Lista zadań nr 9

Zadanie 1. (2 pkt)

Rozszerz kalkulator z wykładu o operacje potęgowania, silni i liczby przeciwnej (unarny minus). W tym celu najpierw uzupełnij składnię abstrakcyjną i interpreter, a następnie rozbuduj parser, tab by obsługiwał nowe konstrukcje. Postaraj się, żeby nowe operatory wiązały w sposób zgodny z przyjętymi konwencjami, np. silnia wiąże najsilniej, potęgowanie wiąże w prawo i silniej niż mnożenie.

Zadanie 2. (2 pkt)

Zaproponuj składnię abstrakcyjną fragmentu języka Racket i zdefiniuj ją jako odpowiedni typ danych w języku Plait. Na fragment języka Racket rozważany w tym zadaniu zawiera tylko wyrażenia (nie zawiera definicji), na które mogą składać się zmienne, liczby, lambda-wyrażenia, aplikacje, formy specjalne **`let`**, **`if`** i **`cond`**. Napisz parser dla tego fragmentu języka. Do reprezentacji zmiennych użyj typu `Symbol`.

Zadanie 3. (3 pkt)

Na stronie przedmiotu znajduje się implementacja parsera wyrażeń, która tylko nieznacznie różni się od parsera pokazanego na wykładzie. Oto lista najważniejszych różnic:

- parser obsługuje tylko operatory addytywne (dodawanie i odejmowanie);
- parsowane wyrażenie jest od razu interpretowane — nie jest tworzone żadne wyrażenie w składni abstrakcyjnej;
- parser nie używa żadnego silnika parsującego, tylko jest zestawem wzajemnie rekurencyjnych funkcji, które w istocie są wyspecjalizowaną wersją funkcji z napisanego na wykładzie silnika parsującego.

Zwróć uwagę na to, że funkcja parsująca `parse-expr-rest` w przypadku sukcesu, oprócz listy tokenów zwróci pewną funkcję `f`. Co więcej, w miejscu wołania funkcji `parse-expr-rest` jest już znana wartość, przekazywana później jako parametr funkcji `f`. Zmodyfikuj parser

tak, by funkcja `parse-expr-rest` zamiast zwracać funkcję, przyjmowała dodatkowy parametr. Zauważ, że po tej zmianie wywołania funkcji `parse-expr-rest` mogą być ogonowe — rekursję zamieniliśmy na iterację!

Zadanie 4. (2 pkt)

Pokazane na wykładzie pierwsze rozwiązanie problemu n hetmanów, z wykorzystaniem list od razu konstruuje listę wszystkich rozwiązań. Takie podejście jest bardzo nieefektywne jeśli interesuje nas tylko jakiekolwiek rozwiązanie. Problem ten możemy rozwiązać używając leniwych strumieni zamiast list. Leniwy strumień, to bezparametrowa funkcja, która zwraca albo koniec strumienia (`sempty`) albo parę: element (głowę) i kolejny strumień (ogon). W języku Plait takie strumienie możemy zdefiniować za pomocą następującego typu.

```
(define-type-alias (Stream 'a) (-> (StreamData 'a)))
(define-type (StreamData 'a)
  (sempty)
  (scons [head : 'a] [tail : (Stream 'a)]))
```

Operowanie na takich strumieniach jest podobne do operowania na zwykłych listach, z tą różnicą że w miejscu gdzie konstruujemy strumień, odraczamy obliczenia za pomocą bezparametrowej lambdy, a tam gdzie go dekonstruujemy, trzeba wykonać odroczone obliczenie.

Przepisz program rozwiązujący problem n hetmanów na taki który zwraca strumień wszystkich rozwiązań. Porównaj czas znalezienia pierwszego rozwiązania z czasami znalezienia pierwszego rozwiązania dla innych wersji tego programu pokazanych na wykładzie.

Zadanie 5.

Przepisz poniższą funkcję wyliczającą n-tą liczbę Fibonacciego na styl kontynuacyjny.

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Zadanie 6. (2 pkt)

Napisz program rozwiązujący problem skoczka szachowego dla szachownicy $n \times n$, tj. znajdujący taką sekwencję ruchów skocza szachowego, że zaczynając w rogu szachownicy odwiedzi wszystkie pola dokładnie raz. Nie przejmuj się, jeśli Twój program działa zbyt wolno dla szachownicy 8×8, wystarczy, że będzie umiał znaleźć rozwiązanie dla szachownicy 5×5 w rozsądnym czasie.