Lista zadań nr 13

Poniższe zadania, z wyjątkiem 4, rozwiąż w języku Plait.

Zadanie 1. (1 pkt)

Rozszerz interpreter z pliku `letrec-state.rkt` o symbole, a także o operację porównania symboli `symbol=?` i predykaty `symbol?` oraz `number?`.

Zadanie 2. (2 pkt)

Rozszerz rozwiązanie poprzedniego zadania o listy (zadanie 5 z listy 10) i *cytowanie*. Wartością wyrażenia (**`quote`** *s*), gdzie *s* jest S-wyrażeniem, a zatem liczbą, symbolem lub listą S-wyrażeń, jest, odpowiednio, wartość reprezentująca liczbę, symbol lub wartość reprezentująca listę wartości.

Napisz w tak rozszerzonym języku *interpretowanym* ewaluator wyrażeń arytmetycznych zbudowanych ze stałych liczbowych, dodawania i mnożenia, reprezentowanych jako S-wyrażenia (utożsamiamy składnię konkretną z abstrakcyjną).

Zadanie 3. (2 pkt)

W interpreterze z pliku `state-store-macros.rkt` mamy do czynienia z niejawnymi referencjami, co oznacza, że rozszerzenie środowiska o wiązanie automatycznie powoduje utworzenie modyfikowalnej komórki pamięci związanej z wiązaną zmienną. Zmień ten interpreter tak by utworzenie modyfikowalnej komórki pamięci było realizowane jawnie i wyłącznie na życzenie programisty. W tym celu należy sprawić by środowisko ponownie przechowywało wartości a nie referencje na wartości, poza jednym przypadkiem – `boxV` – wartością reprezentującą referencję do sterty. Sam język powinien zostać wzbogacony o konstrukcje `box` (utworzenie referencji), `unbox` (odczytanie wartości ze sterty, na którą wskazuje dana referencja) oraz `set-box!` (modyfikacja sterty dla danej referencji). Możesz w tym zadaniu usunąć z języka **`letrec`**, którego implementacja korzysta z niejawnych referencji.

Zadanie 4. (2 pkt)

Zdefiniuj w języku Racket strumień wszystkich liczb pierwszych, opierając się na następującej obserwacji: dana liczba naturalna jest pierwsza (a więc powinna sie znaleźć

w strumieniu), jeżeli nie dzieli sie przez żadną odpowiednio mniejszą liczbę pierwszą. Jest to podejście inne od tego opartego na sicie Eratostenesa, zaprezentowanego na wykładzie.

Zadanie 5. (2 pkt)

Wzorując się na implementacji strumieni z wykładu (plik `letrec-streams.rkt`) popraw swoje rozwiązanie zadania 1 z listy 12, tak by uczynić swój interpreter prawdziwie leniwym. Twój interpreter powinien liczyć wartość argumentu funkcji oraz wyrażenia definiującego w wyrażeniu **`let`** co najwyżej raz, tylko przy pierwszym użyciu. Obliczona wartość powinna zostać zapamiętana, a kolejne odwołania do zmiennej z nią związanej powinny polegać na zwróceniu raz obliczonej wartości.

Zadanie 6. (1 pkt)

Rozszerz interpreter z pliku `cps.rkt` o konstrukcje **`begin`** oraz `set!`.

Zadanie 7. (2 pkt)

Dodaj do języka z pliku `cps.rkt` (lub do tego z poprzedniego zadania) wyrażenie postaci $\{$**`call/cc`** x $e\}$.[1] W tym celu, rozszerz gramatykę wartości o konstruktor przechowujący kontynuacje interpretera, czyli funkcje typu `(Value -> 'a)`. Ewaluacja nowego wyrażenia sprowadza się do ewaluacji wyrażenia e w środowisku, w którym zmienna x jest związana z wartością reprezentującą bieżącą kontynuację, oraz z niezmienioną bieżącą kontynuacją (bieżąca kontynuacja zostaje zwyczajnie skopiowana do środowiska). Taka przechwycona kontynuacja może zostać zaaplikowana do innej wartości przy użyciu funkcji `apply`, która w tym przypadku zapomina o swojej kontynuacji i używa aplikacji z meta języka (tak jak przy operacjach prymitywnych) do zaaplikowania przechwyconej kontynuacji do danej wartości.

Jaka jest wartość następujących wyrażeń (zakładając, że w języku występują listy)?

```
(+ 1 (call/cc k (+ 10 (k 100))))

(+ 1 (call/cc k (+ 10 (k (k 100)))))

(+ 1 (call/cc k (+ 10 100)))
```

[1] **`call/cc`** jest skrótem od **`call-with-current-continuation`**. Jest to tzw. operator sterowania występujący m. in. w językach Scheme i Racket.

```
(call/cc break
  (letrec product
    (lambda (xs)
      (if (null? xs)
          1
          (if (= (car xs) 0)
              (break 0)
              (* (car xs) (product (cdr xs))))))
    (product (list 1 2 0 4 5))))

(let arg-fc
  (lambda (d)
    (call/cc k
      (lambda (x)
        (k (lambda (y) x)))))
  ((map (arg-fc 42)) (list 1 2 3 4)))
```