Lista zagadnień nr 6

Przed zajęciami

Przypominamy, że materiał części drugiej wykładu wprowadzany jest trochę wolniej niż to, co znajdziemy w podręczniku w rozdziale 4 i omawiane interpretery będą się trochę różnić od tego przedstawionego w podrozdziale 4.1. Przed zajęciami należy rozumieć różnice między **składnią konkretną** a **składnią abstrakcyjną**, a także umieć opowiedzieć, co to znaczy, że Racket jest **homoikoniczny**, wiedzieć czym jest **cytowanie** i co to znaczy, że liczby są wartościami **samocytującymi** (cytowania opisane są w podręczniku w podrozdziale 2.3). Ważne jest także zapoznanie się z kodem przedstawionym na wykładzie. W czasie zajęć wygodnie ten kod mieć wydrukowany lub otwarty w edytorze. W celu zrobienia Ćwiczeń 1-4 należy przypomnieć sobie, co to jest **notacja polska** i **odwrotna notacja polska**, a także znać algorytm obliczania wartości wyrażeń w odwrotnej notacji polskiej za pomocą **stosu**.

(Odwrotna) notacja polska

Ćwiczenie 1.

Wyrażenia w *odwrotnej notacji polskiej* (ONP) będziemy reprezentować jako listy, których każdy element jest albo liczbą, albo jednym z symboli `'+`, `'-`, `'*`, `'/`. Zdefiniuj jednoargumentową procedurę `arith->rpn`, która przekształca wyrażenia arytmetyczne (tj. struktury spełniające zdefiniowany na wykładzie predykat `arith-expr?`) na wyrażenia w ONP. Zadbaj, by ta procedura nie generowała nieużytków.

Ćwiczenie 2.

Zdefiniuj strukturę danych reprezentującą stos poprzez zadanie predykatu `stack?` (mówiącego czy dana wartość jest dobrze sformowanym stosem), konstruktor `push` (argmenty to element wrzucany na szczyt stosu i stos, na który wrzucamy; wartością wywołania procedury jest stos z dorzuconym elementem)

i selektor `pop` (zwracający parę złożoną z elementu znajdującego się na szczycie stosu i stosu pozostałego po zdjęciu tego elementu).

Ćwiczenie 3.

Zdefiniuj jednoargumentową procedurę `eval-rpn`, która przy użyciu stosu z poprzedniego zadania oblicza wartość wyrażenia w ONP.

Ćwiczenie 4.

Porównaj prefiksową *notację polską* z notacją prefiksową S-wyrażeń w Rackecie. Skoro notacja polska nie wymaga nawiasów, to czemu S-wyrażenia wymagają nawiasów w składni konkretnej?

Rozszerzanie kalkulatora

Ćwiczenie 5.

Rozszerz składnię i ewaluację `let`-wyrażeń o konstrukcję `if-zero`, np.

```
> (eval '(if-zero (- 2 2) 7 9))
7

> (eval '(if-zero (let (x 3) x) 7 (+ 1 2)))
3
```

Jak zachowa się Twoja ewaluacja dla wyrażenia

```
(if-zero 0 1 (/ 5 0))
```

Czy dostrzegasz dwie możliwe semantyki konstrukcji `if-zero` (leniwą i gorliwą)?

Ćwiczenie 6.

Rozszerz składnię i ewaluację wyrażeń przedstawionych na wykładzie tak, by operacje arytmetyczne mogły dostawać dowolną liczbę argumentów.

Wskazówka: Dobrym pomysłem jest użycie procedry `apply` dostępnej w bibliotece Racketa. Bierze ona dwa argumenty: procedurę i listę. Jej wynikiem jest funkcja zaaplikowana do argmentów na liście, np.

```
> (apply append '((a b c) (x y z)))
'(a b c x y z)

> (apply cons '(10 (20 30)))
'(10 20 30)
```

```
> (apply + '(5 10 20))
35

> (apply (lambda (x y) (+ x y)) '(5 10))
15

> (apply (lambda (x y) (+ x y)) '(5 10 20))
ERROR: #<procedure>: arity mismatch;

> (apply + '(1 2 3 4 5 6 7 8 9))
45
```

Ćwiczenie 7.

Zmień składnię i ewaluację wyrażeń tak, by `let`-wyrażenia nie brały jednej definicji (typu `(x (+ 1 3))`), a listę definicji, tak jak ma to miejsce w Rackecie, np.

```
'(let ((x 3) (y 5) (z 10)) (+ x y z))
```

Zauważ, że teraz lepszą nazwą dla procedury `let-def` byłoby `let-defs`. Czy umiesz poprosić DrRacketa, by zmienił nazwę procedury w bezpieczny sposób, to znaczy taki, który rzecyzwiście zmieni tylko nazwę w definicji procedury i miejscach jej wywołania, a nie ślepo zamieni jedno wystąpienie ciągu znaków na drugi ciąg znaków w tekście programu?

Indeksy De Bruijna

Przedstawiona na wykładzie składnia abstrakcyjna `let`-wyrażeń reprezentuje zmienne za pomocą symboli. Jednak w tym wypadku nie ma większego znaczenia, jaki to właściwie symbol, byle był to ten sam symbol w miejscu wiązania i wystąpienia. Dla przykładu, wyrażenie

```
'(let (x (+ 2 1)) (* x x))
```

nie różni się semantycznie od wyrażenia

```
'(let (y (+ 2 1)) (* y y))
```

Podobnie dzieje się w przypadku innych konstrukcji w językach programowania, np. nazw procedur czy nazw argumentów – z **semantycznego** punktu widzenia nazwy nie mają żadnego znaczenia. To zjawisko ma nawet swoją nazwę: mówimy, że dwa powyższe wyrażenia są *α-równoważne*.

Skoro nazwy zmiennych związanych nie mają znaczenia, czy można się ich pozbyć w składni abstrakcyjnej? Jedną z możliwości są zaproponowane w 1972

roku przez Nicolaasa Goverta de Bruijna *indeksy De Bruijna*. W tym podejściu zmienne nie są wymieniane w konstrukcji wiążącej, a wystąpienie zmiennej zastąpione jest jej indeksem, czyli liczoną od 0 liczbą konstrukcji wiążących, które trzeba „ominąć" idąc w górę drzewa, by odnaleźć odpowiednie wiązanie. Na przykład, składnię abstrakcyjną `let`-wyrażeń z indeksami De Bruijna można wyrazić następująco:

```
(define (let-db? t)
  (and (list? t)
       (= (length t) 3)
       (eq? (car t) 'let)))

(define (let-db-def e)
  (cadr e))

(define (let-db-expr e)
  (caddr e))

(define (db-index? e)
  (and (list? e)
       (= (length e) 2)
       (eq? (car e) 'index)
       (number? (cadr e))))

(define (arith/let-expr-db? t)
  (or (self-evaluating? t)
      (and (binop? t)
           (arith/let-expr-db? (binop-left  t))
           (arith/let-expr-db? (binop-right t)))
      (and (expr-let-db? t)
           (arith/let-expr-db? (let-db-def  t))
           (arith/let-expr-db? (let-db-expr t)))
      (db-index? t)))
```

Oto kilka przykładów wyrażeń zapisanych na dwa sposoby:

```
'(let (x 3) x)              '(let 3 (index 0))

'(let (x 3)                 '(let 3
 (let (y 7)                  (let 7
 (+ x y)))                   (+ (index 1) (index 0))))

 '(let (x                   '(let
        (let (y 3) y))            (let 3 (index 0))
  (let (z 8)                 (let 8
  (+ x z)))                  (+ (index 1) (index 0))))
```

Ćwiczenie 8.

Napisz procedurę konwerującą `let`-wyrażenia zdefiniowane na wykładzie do reprezentacji używającej indeksów De Bruijna.

Wskazówka: Wystarczy prosta procedura rekurencyjna, przypominająca kształtem obliczanie wartości wyrażeń ze środowiskiem. Tym razem środowi-

skiem jest lista nazw zmiennych związanych, którą rozszerzamy rekurencyjnie „wchodząc" do wyrażenia, w którym wiązana jest nowa zmienna. Wówczas indeksem zmiennej jest pozycja tej zmiennej w środowisku (licząc od 0).

Ćwiczenie 9.

Napisz procedurę konwerującą `let`-wyrażenia reprezentowane przy pomocy indeksów De Bruijna do reprezentacji zdefiniowanej na wykładzie.

Wskazówka: Tu również wystarczy procedura rekurencyjna ze środowiskiem. Tym razem środowiskiem jest lista nazw zmiennych kolejnych zagnieżdżonych `let`-ów, a indeks n zastępujemy n-tą zmienną ze środowiska. Trudnością jest konwersja samych `let`-ów, bo musimy skądś wziąć nazwę zmiennej. Możliwym rozwiązaniem jest wprowadzenie dodatkowego licznika, który liczy, ile zmiennych związanych jest w danym kontekście, i tworzenie nowej zmiennej za pomocą następującej procedury:

```
(define (number->symbol i)
  (string->symbol (string-append "x" (number->string i))))
```

Licznik może być dodatkowym argumentem (zwiększanym o 1 z każdym dodaniem zmiennej do środowiska), ewentualnie (mniej wydajnie) być po prostu liczoną w razie potrzeby długością środowiska.