MP19 @ II UWr 9 maja 2019 r.

# Lista zagadnień nr 10

## Przed zajęciami

Należy wiedzieć, co to jest **środowisko** oraz **domknięcie** i jak te dwie rzeczy mają się do siebie. Należy wiedzieć, czym jest **dynamiczne**, a czym **leksykalne** wiązanie zmiennych i jak te pojęcia mają się do środowisk i domknięć. Należy rozumieć, jak działają języki i ich interpretery przedstawione na wykładzie. Rozumieć, po co nam **czarne dziury**.

## Zadania, które można rozwiązać „na szybko" przy tablicy

### Ćwiczenie 1.

Napisz procedurę, która rozwiązuje zadanie 1. z listy 9. To znaczy: argumentem tej procedury jest wyrażenie (w składni abstrakcyjnej naszego języka wyrażeń), które oblicza się do lambdy. Wynikiem zwracanym przez procedurę jest domknięcie, które jest wartością tego wyrażenia.

### Ćwiczenie 2.

Dlaczego wynikeim wyrażenia

```
(run (cons-expr (const 1) (cons-expr (const 2) (cons-expr (const 3)
    (null-expr)))))
```

jest

```
'(1 2 3 . null)
```

a nie

```
'(1 2 3)
```

tak jak w Rackecie?

# Zadania bardziej implementacyjne

### Ćwiczenie 3.

Zmodyfikuj interpreter języka WHILE tak, by działał jak prosty **debugger** – jego wynikiem powinien być nie ostateczny stan pamięci, ale lista wszystkich stanów pamięci, przez które przechodzi ewaluacja. Np. przypomnij sobie program liczący wartość silni zmiennej i znajdującej się w początkowej pamięci:

```
(define fact-in-WHILE
  (var-block 'x (const 0)
    (comp (assign 'x (const 1))
    (comp (while (op '> (variable 'i) (const 0))
             (comp (assign 'x (op '* (variable 'x) (variable 'i)))
                   (assign 'i (op '- (variable 'i) (const 1)))))
          (assign 'i (variable 'x))))))
```

Teraz, jeśli każemy Racketowi obliczyć wartość wyrażenia

```
(debug fact-in-WHILE (env-from-assoc-list '((i 5))))
```

gdzie debug to nasz zmodyfikowany interpreter, powinniśmy dostać odpowiedź

```
'(((i 5))
  ((x 0) (i 5))
  ((x 1) (i 5))
  ((x 5) (i 5))
  ((x 5) (i 4))
  ((x 20) (i 4))
  ((x 20) (i 3))
  ((x 60) (i 3))
  ((x 60) (i 2))
  ((x 120) (i 2))
  ((x 120) (i 1))
  ((x 120) (i 1))
  ((x 120) (i 0))
  ((x 120) (i 120))
  ((i 120)))
```

*Wskazówka:* Dzięki temu, że zachowaliśmy abstrakcję danych, tak naprawdę nie musimy dotykać samego interpretera; wystarczy alternatywna reprezentacja środowisk, być może z drobnym *post-processingiem* ostatecznego wyniku.

### Ćwiczenie 4.

Na wykładzie 16 kwietnia próbowaliśmy zrobić leniwe let-wyrażenia, ale nam nie wyszło (to znaczy: wyszło, ale z dynamicznym wiązaniem zmiennych). Teraz mamy już narzędzia (domknięcia), żeby sobie z tym poradzić. Dodaj

do języka konstrukcję `let-lazy`, która implementuje leniwe let-wyrażenia z leksykalnym wiązaniem zmiennych.

### Ćwiczenie 5.

Funkcja rekurencyjna jest **ogonowa**, jeśli wartość rekurencyjnego wywołania jest zawsze zwracaną wartością – a więc nie możemy np. dodać do wyniku 1. Funkcje ogonowe znamy w Rackecie pod nazwą *procesy iteracyjne* i widzieliśmy je już wielokrotnie. Napisz procedurę, która bierze jako swój argument wyrażenie postaci

```
(letrec 'foo (lambda (x) e1) e2)
```

i mówi, czy tak zdefiniowana jednoargumentowa funkcja `foo` jest ogonowa. (Tak, wyrażenie `e2` nie bierze udziału w grze, ale nie mamy w języku konstrukcji `define`.)

### Ćwiczenie 6.

Rozbuduj język i interpreter wyrażeń z wykładu o definicje **wzajemnie rekurencyjne**, które można wprowadzić przez definiowanie wielu wartości w wyrażeniu `letrec`, tak jak w poniższym wyrażeniu danym w składni konkretnej Racketa:

```
(letrec ((even (lambda (x) (if (= x 0) true  (odd  (- x 1)))))
         (odd  (lambda (x) (if (= x 0) false (even (- x 1))))))
  (even 1234))
```

Dla uproszczenia możesz przyjąć, że zawsze definiujemy tylko dwie wartości, tzn. pracujemy na składni abstrakcyjnej w rodzaju:

```
(struct letrec-mutual (x e1 y e2 e3))

(define (expr? e)
  (match e
    ...
    [(letrec-mutual x e1 y e2 e3) (and (symbol? x)
                                       (symbol? y)
                                       (expr? e1)
                                       (expr? e2)
                                       (expr? e3))]
    ...))
```

*Uwaga:* To zadanie nie jest bardzo łatwe i jest dość subtelne semantycznie. Np. rozważ racketowe wyrażenie

```
(letrec ((y 3) (x y)) (+ x y))
```

Ma ono, zgodnie z oczekiwaniami, wartość 6. Ale próba obliczenia wartości wyrażenia

```
(letrec ((x y) (y 3)) (+ x y))
```

kończy się błędem. Czy tak Twoim zdaniem powinno być? Np. w OCamlu odpowiednieki obu definicji **nie** są akceptowane jako poprawne wyrażenia, a w Haskellu obie są Ok. Tak więc rozwiązując to zadanie masz możliwość wyboru szczegółów semantyki (która nie powinna się różnić, jeśli definiujemy wzajmenie rekurencyjne **funkcje**, jak w przykładzie z `even` i `odd`).

### Ćwiczenie 7.

Przypomnij sobie pojęcie makro instrukcji z ćwiczenia 7. z listy 8. W tym wypadku będziemy raczej mówić o **makro wyrażeniach**. Pokaż, że wzajemnie rekurencyjny `letrec` z poprzedniego zadania może być wyrażony jako makro wyrażenie (więc nie trzeba modyfikować składni abstrakcyjnej ani interpretera żeby napisać wzajemnie rekurencyjne funkcje).

*Wskazówka:* Nie przejmuj się, jeśli Twoja makro instrukcja będzie duplikować kod i/lub być mniej wydajna niż ewentualne wbudowanie wzajemnie rekurencyjnego wyrażenia `letrec` bezpośrednio w język i interpreter. W tym zadaniu chodzi jedynie o uzyskanie tej samej (albo przynajmniej podobnej) semantyki.