MP19 @ II UWr 22 kwietnia 2019 r.

Lista zagadnień nr 8

Przed zajęciami

Należy wiedzieć, jak używać procedur do programowania ze zmiennym stanem w Rackecie: `set!`, `mcons`, `mcar`, `mcdr`, `set-mcar!`, `set-mcdr!` itd. Należy wiedzieć, co to znaczy, że dwa programy są α**-równoważne**, rozumieć, co to jest **środowisko**.

Należy też zapoznać się z „kodem z wykładu" zamieszczonym na SKOS-ie, który jest uporządkowaną wersją tego, co naprawdę zobaczyliśmy na wykładzie (np. zachowujemy konwencjonalną abstrakcję). Rozumieć, jak działa język WHILE, jak reprezentujemy jego składnię abstrakcyjną i jak działa interpreter.

Na zajęciach

Ćwiczenie 1.

Napisz procedurę, która bierze jako swój argument (zwykłą) listę i zmienia ją na cykl złożony z `mcons`-ów. Cykl to „lista", w której `mcdr` ostatniego „pudełka" wskazuje na pierwsze „pudełko" listy.

Ćwiczenie 2.

Zdefiniuj procedurę `set-nth!` taką, że `(set-nth! n xs v)` zamienia `n`-ty element modyfikowalnej listy `xs` na wartość wyrażenia `v`.

Ćwiczenie 3.

Przedstaw w składni abstrakcyjnej program w języku WHILE, który oblicza n-tą liczbę Fibonacciego.

Ćwiczenie 4.

Czy koniunkcja w interpreterze wyrażeń z wykładu jest leniwa czy gorliwa? Innymi słowy, czy jeśli pierwszy argument jest fałszem, to czy wartość drugiego argumentu jest w ogóle obliczana?

Ćwiczenie 5.

Rozszerz składnię abstrakcyjną języka WHILE i interpreter z wykładu o instrukcję $++$. Składnia konkretna tej instrukcji mogłaby być dana jako $x{+}{+}$, gdzie x to zmienna. Semantyka to: powiększ wartość zmiennej x o 1.

Ćwiczenie 6.

Rozszerz składnię abstrakcyjną języka WHILE o konstrukcję **for**. Składnia konkretna takiej instrukcji mogłaby wyglądać tak: **for**$(x := e_1,\ e_2,\ s_1)\ s_2$. Semantyka może być opisana nieformalnie tak:

1. Utwórz nową zmienną x widoczną dla wyrażenia e_2 i instrukcji s_1 i s_2. Początkową wartością zmiennej x jest wartość wyrażenia e_1.
2. Jeśli wartość wyrażenia e_2 to fałsz, zakończ wykonywanie całej instrukcji **for**.
3. Wykonaj instrukcję s_2.
4. Wykonaj instrukcję s_1.
5. Powtórz kroki 2–5.

Sformalizuj semantykę poprzez rozszerzenie interpretera języka WHILE z wykładu o obsługę instrukcji **for**.

Ćwiczenie 7.

Zaletą tego, że działamy na składni abstrakcyjnej języka WHILE jest to, że możemy traktować ją jak każdą inną strukturę danych w Rackecie i wykonywać na niej różne obliczenia. W szczególności możemy generować program lub fragmenty programu. Jest to użyteczne do wyrażania bardziej zaawansowanych konstrukcji w języku przy użyciu już istniejących instrukcji bez potrzeby jakiejkolwiek ingerencji w składnię abstrakcyjną lub interpreter. Takie konstrukcje nazwiemy *makro instrukcjami*. Nie są one fragmentem jezyka, który interpretujemy, ale są racketowymi procedurami, które tworzą fragmenty programów

w języku WHILE. Dla przykładu, instrukcja `comp` jest dość niewygodna, bo pozwala składać jedynie dwie instrukcje. Można wyrazić makro instrukcję, która składa wiele instrukcji jako ciąg binarnych złożeń:

```
(define (comp* . xs)
  (cond [(null? xs) (skip)]
        [else (comp (car xs) (apply comp* (cdr xs)))]))
```

Dzięki niej program, który w składni konkretnej zapisalibyśmy jako $x := 1; y := 2; z := 3; t := 4$ możemy wyrazić w składni abstrakcyjnej jako

```
(define prog
  (comp*
   (assign 'x (const 1))
   (assign 'y (const 2))
   (assign 'z (const 3))
   (assign 't (const 4))))
```

Pokaż, że instrukcja **for** z poprzedniego zadania może być wyrażona jako makro instrukcja (więc nie ma tak naprawdę potrzeby rozszerzać składni abstrakcyjnej ani interpretera).

Ćwiczenie 8.

Dla ustalonego n, problem n hetmanów można rozwiązać w języku WHILE przy pomocy następującego programu zapisanego w jakieś tam składni konkretnej:

$x_1 := 0; ...; x_n := 0$
$done := \textbf{false};$
$\textbf{for}(y_1 := 1,\ y_1 \leq n \ \&\&\ \mathsf{not}(done),\ y_1 := y_1 + 1)$
$\ \textbf{if}(\textbf{true})$
$\ \ \textbf{for}(y_2 := 1,\ y_2 \leq n \ \&\&\ \mathsf{not}(done),\ y_2 := y_2 + 1)$
$\ \ \ \textbf{if}(y_2 \neq y_1 \ \&\&\ \mathsf{abs}(y_2 - y_1) \neq 2 - 1)$
$\ \ \ \ ...$
$\ \ \ \ \ \textbf{for}(y_k := 1,\ y_k \leq n \ \&\&\ \mathsf{not}(done),\ y_k := y_k + 1)$
$\ \ \ \ \ \ \textbf{if}(y_k \neq y_1 \ \&\&\ y_k \neq y_2 \ \&\&\ ... \ \&\&\ y_k \neq y_{k-1} \ \&\&\ \mathsf{abs}(y_k - y_1) \neq k - 1$
$\ \ \ \ \ \ \ \ \&\&\ \mathsf{abs}(y_k - y_2) \neq k - 2 \ \&\&\ ... \ \&\&\ \mathsf{abs}(y_k - y_{k-1}) \neq k - (k-1))$
$\ \ \ \ \ \ \ ...$
$\ \ \ \ \ \ \ \ \textbf{for}(y_n := 1,\ y_n \leq n \ \&\&\ \mathsf{not}(done),\ y_n := y_n + 1)$
$\ \ \ \ \ \ \ \ \ \textbf{if}(y_n \neq y_1 \&\&\ ... \ \&\&\ y_n \neq y_{n-1} \ \&\&\ \mathsf{abs}(y_n - y_n) \neq n - 1$
$\ \ \ \ \ \ \ \ \ \ \ \&\&\ \ \neq n - 2 \ \&\&\ ... \ \&\&\ \mathsf{abs}(y_n - y_{n-1}) \neq n - (n-1))$
$\ \ \ \ \ \ \ \ \ \ \ \ done := \textbf{true}; x_1 := y_1; ...; x_n := y_n$

Po zakończeniu programu numer wiersza, w którym można postawić hetmana w k-tej kolumnie, zapisany jest w zmiennej x_k. Zauważ, że n nie jest

argumentem wejściowym programu – to dla każdego n istnieje osobny program (zawierający n pętli i n instrukcji **if**), który rozwiązuje problem dla szachownicy $n \times n$.

Zdefiniuj procedurę `n-queens-in-WHILE`, która bierze jednen argument (liczbę n) i generuje program w języku WHILE rozwiązujący problem n hetmanów odpowiadający programowi powyżej. Następnie zdefiniuj jednoargumentową procedurę `n-queens`, która generuje odpowiedni program w języku WHILE i uruchamia go w interpreterze.

Oczywiście możesz użyć pętli **while** zamiast **for**. Możesz też najpierw rozbudować język o konstrukcje (albo zdefiniować odpowiednie makro instrukcje!), które ułatwią rozwiązanie zadania, np. implementujące funkcje `not` i `abs`. Do rozwiązania na pewno przyda się procedura dynamicznie generująca nazwy zmiennych. Można ją zaimplementować następująco:

```
(define (gen-symbol s i)
  (string->symbol (string-append s (number->string i))))
```

Na przykład wartością `(gen-symbol "x" 3)` jest symbol `'x3`.