Lista zagadnień nr 8

Zadania na ćwiczenia

Wartości boolowskie i pary

Ćwiczenie 1.

Fakt, że reprezentujemy zarówno *stałą* `true` jak i *wartość* `true` w postaci racketowego `true` (czy też `#t`) może wywołać trochę zamieszania. Być może warto więc odróżnić te trzy sposoby istnienia prawdy i fałszu. Zmień implementację w kodzie z wykładu tak, by:

- W składni abstrakcyjnej prawda była reprezentowana jako `(const 'true)`, a nie jak dotychczas `(const true)` (upewnij się, że rozumiesz różnicę). Analogicznie z fałszem.
- Do reprezentowania wartości będącej wynikiem interpretacji, użyj liczb: niech `0` reprezentuje fałsz, a `1` reprezentuje prawdę.

Przetestuj nowe rozwiązanie, żeby upewnić się, że dostosował[e/a]ś wszystkie elementy systemu do nowej reprezentacji. Jakie widzisz wady i zalety takiego rozwiązania?

Ćwiczenie 2.

Dodaj do języka konstrukcje (w Rackecie nazwalibyśmy je „formami specjalnymi") `boolean?`, `number?` i `pair?`, które pozwalają sprawdzić czy dana wartość jest liczbą, boolem, czy też parą. Zwróć uwagę, że w Rackecie te dwie konstrukcje to procedury — my na razie pozostaniemy przy formach specjalnych. Czy decyzje podjęte w poprzednim zadaniu ułatwiają czy utrudniają dodanie tych operatorów?

Ćwiczenie 3.

Nasze operatory `and` i `or` są gorliwe, bo nasz interpreter zawsze oblicza oba argumenty, jak dla *każdego* operatora binarnego (zastanów się, czy na pewno

rozumiesz dlaczego tak się dzieje). Dodaj do języka `and` i `or` jako formy specjalne, które będą odpowiednio leniwe w stylu Racketa. Czy musisz zmieniać składnię abstrakcyjną? Czy chcesz zmieniać składnię abstrakcyjną?

Ćwiczenie 4.

Rozszerz język o konstrukcję `unop` dla operatorów unarnych (jednoargumentowych). Rozszerz ewaluator o operator jednoargumentowy `not` i spraw, by `fst` i `snd` (a także predykaty z Ćwiczenia 2) nie były formami specjalnymi a operatorami unarnymi. Jakie widzisz zalety i wady takiego rozwiązania (w szczególności w kontekście funkcji wyższego rzędu)? Jak napisać procedurę `parse`, żeby ewaluator nie pomylił wywołania funkcji z operatorem unarnym?

Ćwiczenie 5.

Jedną z nielicznych zalet nazw `car` i `cdr` w Rackecie jest wygodna składnia służąca do składania tych procedur. Można więc napisać np. `cadar` żeby uzyskać wartość pierwszego elementu drugiego elementu pierwszego elementu jakieś pary. Wadą (?) racketowej implementacji jest to, że konstrukcje `c[a/d]+r` to zwykłe procedury zdefiniowane w bibliotece standardowej, więc jest ich skończona liczba. Możemy więc użyć procedury `cadaar`, ale procedury `cadaadr` już w bibliotece nie znajdziemy.

Rozbuduj nasz język o formy specjalne postaci `c[a/d]+r` dowolnej długości, poprawiając w ten sposób racktowe podejście. Prawdopodobnie przydadzą Ci się racketowe procedury `symbol->string` i `string->list`. Pojedyncze znaki możemy porównywać procedurą `eq?`, np.

```
> (symbol->string 'cadaadadr)
"cadaadadr"
> (string->list (symbol->string 'cadaadadr))
'(#\c #\a #\d #\a #\a #\d #\a #\d #\r)
> (eq? #\c #\a)
#f
> (eq? #\c #\c)
#t
```

Listy

Ćwiczenie 6.

Rozbudujmy nasz język z wykładu o listy których komórki *nie* są tożsame z parami. Dodaj do składni języka stałą `null`, operację binarną `cons`, operacje unarne

`head`, `tail`, `cons?` i `null?` reprezentujące odpowiednio stałą końca listy, konstrukcję komórki składającej się z głowy i ogona, odpowiednie selektory dla tej komórki i predykaty, a następnie odpowiednio rozszerz ewaluator. Zadbaj żeby komórki skonstruowane przez `cons` były odróżnialne od par skonstruowanych przez `pair`!

Ćwiczenie 7.

Do naszego języka dodaj lukier syntaktyczny `list` z semantyką taką jak w Rackecie. Oznacza to, że procedura `parse` powinna zmienić wyrażenie

```
(list e1 e2 e3 e4)
```

na składnię abstrakcyjną odpowiadającą wyrażeniu

```
(cons e1 (cons e2 (cons e3 (cons e4 null))))
```

Funkcje i domknięcia

Ćwiczenie 8.

Zdefiniuj w naszym języku funkcję `not`.

Ćwiczenie 9.

Zdefiniuj w naszym języku funkcje `curry` i `uncurry`, spełniające następujące równania:

```
(curry f x y) ≡ (f (pair x y))
(uncurry f (pair x y)) ≡ (f x y)
```

Ćwiczenie 10.

Dodaj do naszego języka konstrukcję `let-lazy`, która działa podobnie do zwykłego let-wyrażenia, ale wartość liczona jest leniwie, czyli dopiero w momencie, gdy jest potrzebna. Proszę pamiętać o statycznym wiązaniu zmiennych! Wynik nie musi być spamiętany, więc w programie

```
(let-lazy [x (+ 3 5)]
  (+ x x))
```

dodawanie `3` do `5` może być wykonane dwa razy. Wartość zmiennej powinna być obliczana przy każdym użyciu zmiennej, w szczególności aplikacja funkcji do argumentu powinna być gorliwa, np. program

```
(let-lazy [x (+ 3 5)]
  ((lambda (a) 1) x)
```

powinien obliczyć się do wartości 1 a wartość zmiennej x powinna być obliczona w momencie aplikacji.

Zadania domowe

Zadanie 8a.

Dodaj do języka z wykładu formę dwuargumentową formę specjalną (albo, wedle uznania, operator binarny) `apply`, który aplikuje funkcję do listy argumentów, np. (wykorzystując składnię dodaną w Zad. 7):

```
> (eval (parse
    '(apply (lambda (x y) (+ x y))
            (list 1 2))))
3
> (eval (parse
    '(apply (lambda (x y z) (+ x (+ y z)))
            (list 1 2))))
#<clo>
> (eval (parse
    '(apply (lambda (x y) (lambda (z) (+ x (+ y z))))
            (list 1 2 3))))
6
> (eval (parse
    '(apply (lambda (x y) (+ x y))
            (list 1 2 3))))
ERROR
```

Zadanie 8b.

Zmodyfikuj język z wykładu tak, by w naszym języku argumenty funkcji i formy specjalnej `pair` były liczone leniwie. Np.

```
> (eval (parse
    '((lambda (x) (+ 3 3)) (/ 5 0))))
6
> (eval (parse
    '(let [if-fun (lambda (b t e) (if b t e))]
       (if-fun true 4 (/ 5 0)))))
4
> (eval (parse
    '(fst (pair (+ 2 2) (/ 5 0)))))
4
```

Wskazówka: W języku z wykładu jawnie obliczamy wartość argumentu. Zamiast tego, można utworzyć odroczone obliczenie — tak jak w rozwiązaniu Ćwiczenia 10.