Lista zagadnień nr 6

Zadania na ćwiczenia

Ćwiczenie 1.

Przedstaw w składni konkretnej i abstrakcyjnej z wykładu/notatek wyrażenia, które w składni konkretnej używanej zwykle w matematyce zapisalibyśmy jako:

(a) $\frac{8}{2+3} + 10 + 1$

(b) $1 + 2 + 3 * 4 + 5$

Ćwiczenie 2.

Przypomnij sobie zadanie 7. z listy 2. o obliczaniu przybliżonej wartości liczby π przy użyciu ułamków łańcuchowych. Zdefiniuj jednoargumentową procedurę `pi-expr`, taką że `(pi-expr n)` generuje wyrażenie arytmetyczne (w postaci składni abstrakcyjnej), obliczające liczbę π używając `n`-tego przybliżenia. Następnie oblicz wartość tego wyrażenia przy użyciu procedury `eval` zdefiniowanej w notatkach.

Wskazówka: Najbardziej eleganckie rozwiązanie to przerobić rozwiązanie zadania 6. z listy 2., ale zamiast wykonywać dodawanie i dzielenie, tworzyć wyrażenia, które dodają i dzielą.

Ćwiczenie 3.

Rozszerz składnię abstrakcyjną, składnię konkretną i ewaluator wyrażeń arytmetycznych o:

(a) Operator potęgowania

(b) Operator wartości bezwzględnej

Wskazówka: Żeby wprowadzić operator potęgowania, nie trzeba definiować nowego rodzaju węzła w składni abstrakcyjnej. Żeby zrobić operator wartości bezwzględnej, wypadałoby wprowadzić nowy rodzaj węzła.

Ćwiczenie 4.

Zdefiniuj procedurę `pretty-print`, która bierze jako argument wyrażenie arytmetyczne (oczywiście w składni abstrakcyjnej), a jako rezultat daje ciąg znaków reprezentujący to wyrażenie w standardowej, infiksowej notacji.

- W wersji łatwiejszej wynik musi być po prostu prawidłowy.
- W wersji trudniejszej wynik może zawierać tylko te nawiasy, które są niezbędne.

Przydatne będą racketowe procedury `symbol->string` i `number->string`, a także wieloargumentowa procedura `append-string`.

Ćwiczenie 5.

Rozbudujmy składnię wyrażeń arytmetycznych o jedną zmienną, którą w składni konkretnej nazwiemy `x`:

```
(struct variable () #:transparent)

(define (expr? e)
  (match e
    ...
    [(variable) true]))

(define (parse q)
  (cond
    ...
    [(eq? q 'x) (variable)]))
```

(nie trzeba kopiować z pdf-a, plik `ex-derivative.rkt` z tak rozszerzoną składnią znajduje się na SKOS-ie). W składni konkretnej możemy więc pisać wyrażenia typu `'(+ (* 2 x) (* x x))`.

Takie wyrażenie możemy rozumieć jako definicję funkcji (w matematycznym sensie tego słowa) na zmiennej `x`. Napisz procedurę ∂ (U+2202), która generuje pochodną tej funkcji (także jako funkcję na zmiennej `x` zdefiniowaną w postaci składni abstrakcyjnej). Np.

```
(∂ (binop '* (variable) (binop '* (variable) (variable))))
```

powinno wyprodukować wyrażenie równoważne wyrażeniu

```
(binop '* (const 3) (binop '* (variable) (variable)))
```

(równoważne, a niekoniecznie równe, bo bezpośrednia implementacja szkolnych regułek liczenia pochodnych może wyprodukować nam trochę śmieci w postaci

wyrażeń `(+ e 0)` czy `(* e 1)`, albo długie postaci typu `(+ e (+ e e))` zamiast `(* 3 e)`).

Uwaga: Tu i w następnych zadaniach dwóch zadaniach możesz przyjąć dla uproszczenia, że wyrażenia używają tylko operatorów + i ∗.

Ćwiczenie 6.

Zdefiniuj procedurę `simpl`, która potrafi upraszczać wyrażenia z poprzedniego zadania używając prostych reguł arytmetycznych typu $a + 0 = a$, $1a = a$, czy $a + a + a = 3a$. Zastosuj tę procedurę do wyników generowanych przez procedurę ∂ z poprzedniego zadania. Porównaj swoje rozwiązanie z rozwiązaniami innych uczestników w Twojej grupie ćwiczeniowej: komu udało się najbardziej uprościć wyrażenia?

Ćwiczenie 7.

Rozbuduj składnię konkretną, abstrakcyjną i ewaluator z zadania 5. o *operator* ∂ (czyli ∂ nie jest już procedurą przetwarzającą wyrażenia w wyrażenia, a jest elementem składni wyrażeń). Tak więc teraz prawidłowym wyrażeniem w składni konkretnej może być:

```
(+ (∂ (+ (variable) (∂ (* (variable) 3)))) 8)
```

Zmodyfikowany ewaluator powinien przyjmować dwa argumenty: wyrażenie i wartość zmiennej. Np.

```
(eval (parse '(∂ (+ x (∂ (* x (* x x)))))) 10)
```

daje rezultat 61, bo $\partial(x + \partial x^3) = \partial(x + 3x^2) = 1 + 6x$, a $1 + 6 * 10 = 61$.

Zadania domowe

Zadanie 6a.

W tym zadaniu zmodyfikujemy wyrażenia arytmetyczne zdefiniowane w pliku `arith.rkt` tak, by umożliwiały obliczenia na *liczbach zespolonych*. W szczególności należy:

- Dodać do składni konkretnej stałą `i` (podobnie jak w zadaniu 5. powyżej dodaliśmy zmienną `x`). Chcemy więc móc obliczyć wartość wyrażenia tak jak poniżej:

  ```
  (eval (parse '(+ 2 (* 8 i))))
  ```

- Zmodyfikować odpowiednio składnię abstrakcyjną i dostosować procedurę `parse`,
- Zmodyfikować ewaluator. W szczególności, wynik działania procedury `eval` powinien zwracać wartości zdefiniowane w poniższy sposób:

  ```
  (struct complex (re im) #:transparent)
  (define value? complex?)
  ```

  Jak sugerują nazwy argumentów `re` i `im`, jest to liczba zespolona w reprezentacji kanonicznej (część rzeczywista i urojona).

Na przykład:

```
> (eval (parse '(* i i)))
(complex -1 0)
> (eval (parse '(+ 3 (* i 8))))
(complex 3 8)
```

Wskazówka: Są dwa naturalne sposoby reprezentacji składni abstrakcyjnej w tym zadaniu. Chyba żaden z nich nie jest znacząco lepszy – stąd wymyślenie reprezentacji pozostawiamy Rozwiązującym.

Zadanie 6b.

Napisz kompilator wyrażeń w odwrotnej notacji polskiej (RPN) do wyrażeń arytmetycznych, uzupełniając szablon rozwiązania znajdujący się w pliku `ex-rpn-to-arith.rkt`.

Wskazówka: W notatkach jest uwaga mówiącą, że kompilator z wyrażeń arytmetycznych do RPN to taki ewaluator tylko z wyrażeniami RPN jako wartościami. Może w tym przypadku jest podobnie i kompilator z RPN do wyrażeń arytmetycznych to taki ewaluator z wyrażeniami arytmetycznymi jako wartościami?