Lista zagadnień nr 7

Przed zajęciami

Należy wiedzieć, co to jest **składnia konkretna** i co to jest **składnia abstrakcyjna**. Należy rozumieć, jak reprezentujemy składnię abstrakcyjną w postaci Racketowych struktur danych oraz jak reprezentujemy konstrukcje, które wiążą zmienne (np. `let`-wyrażenia). Należy wiedzieć, co to są **wyrażenia**, a co to są **wartości**. Należy rozumieć pojęcia **parser**, **pretty-printer**, **ewaluator** i **kompilator**.

Na zajęciach (składnia)

Ćwiczenie 1.

Rozważ poniższe przykłady składni **konkretnej** używanej w matematyce. Dla każdego przykładu wymyśl składnię **abstrakcyjną** i rozszerz o nią składnię wyrażeń arytmetycznych ze zmiennymi i `let`-wyrażeniami przedstawioną na wykładzie, tzn. zdefiniuj odpowiednie struktury i rozszerz predykat `expr?`:

- Potęgowanie: a^b
- Suma wartości dla kolejnych liczb naturalnych: $\sum_{i=n}^{m} f(i)$.
- Całka oznaczona: $\int_a^b f(x)dx$
- Minimum zbioru: $\min\{f(i) \mid i \in \mathbb{N}\}$

Dobrze przemyśl, które przykłady składni wiążą zmienne, co jest stałą, a co jest wyrażeniem. Czy w zaproponowana składnia abstrakcyjna umie reprezentować wyrażenie $\int_{1+1}^{\infty} \frac{1}{2^x} dx$?

Ćwiczenie 2.

Zaproponuj składnię abstrakcyjną następujących konstrukcji występujących w Rackecie:

- `if`
- `cond`
- `lambda`

Ćwiczenie 3.

Zdefiniuj procedurę `from-quote`, która umie przetworzyć zacytowane Racketowe wyrażenie arytmetyczne na składnię abstrakcyjną wyrażeń arytmetycznych wprowadzoną na wykładzie. Np.

```
(from-quote '(+ 2 (* 1 2) 4))
```

może wyprodukować wynik w postaci

```
(op '+ (const 2) (op '+ (op '* (const 1) (const 2)) (const 4)))
```

Zwróć uwagę, że są także inne możliwe prawidłowe odpowiedzi, ponieważ nasze wyrażenia arytmetyczne zawierają jedynie operatory binarne.

Ćwiczenie 4.

Przedstawiona na wykładzie procedura ∂ jest bez wątpienia poprawna, ale produkowane przez nią wyrażenia są nieprzyjemnie duże (np. `(∂ (∂ (∂ f)))` w kodzie zamieszczonym w SKOS-ie). Bardziej dokładne spojrzenie na te wyrażenia ujawnia jednak, że zawierają one dużo podwyrażeń, które można uprościć, np. `(op '* (const 0) (variable))`, które (używając praw arytmetyki) można uprościć do `(const 0)`. Napisz procedurę, która korzystając z najprostszych praw arytmetyki (typu $x + 0 = x$, $1x = x$, $0x = 0$, $x + x = 2x$ itp.) przetwarza wyrażenie na trochę bardziej czytelne. Przetestuj na wyrażeniu `(∂ (∂ (∂ f)))`.

Ćwiczenie 5.

Zaimplementuj optymalizację, która działa na wyrażeniach arytmetycznych z `let`-wyrażeniami usuwając z wyrażenia nieużywane `let`-wyrażenia. Na przykład w programie

```
(let 'x (op '+ (const 2) (const 2))
  (let 'y (op '* (const 3) (variable 'x))
    (op '+ (const 7) (variable 'x))))
```

definicja `'y` nie jest używana i całość może być uproszczona do

```
(let 'x (op '+ (const 2) (const 2))
  (op '+ (const 7) (variable 'x)))
```

Zastanów się, czy w prawdziwym języku programowania taka optymalizacja jest poprawna.

Ćwiczenie 6.

Na wykładzie widzieliśmy prosty **kompilator**, który umie tłumaczyć nasze wyrażenia arytmetyczne (czyli wartości spełniające predykat `expr?`) na wyrażenia w odwrotnej notacji polskiej. Napisz **ewaluator** takich wyrażeń, zgodnie ze (znanym Ci zapewne) modelem obliczeń ze stosem:

1. Jeśli pierwszym elementem wyrażenia jest stała, wrzuć ją na stos i kontynuuj obliczenie.
2. Jeśli pierwszym elementem wyrażenia jest operator, zdejmij ze stosu dwie wartości, oblicz wartość zgodnie z tym, co to za operator, wrzuć wynik na stos i kontynuuj obliczenie.
3. Jeśli po przejściu całego wyrażenia na stosie znajduje się dokładnie jedna wartość, jest to wartość końcowa obliczenia.

Zacznij od zaimplementowania stosu jako struktury danych.

Na zajęciach (stan)

Ćwiczenie 7.

Zdefiniuj procedurę `make-cycle`, która zmieni listę modyfikowalnę podaną jako jedyny parametr na listę cykliczną, nadpisując `mcdr` ostatniej pary należącej do listy, aby wskazywała na pierwszą.