ФУНКЦИОНАЛНО ПРОГРАМИРАНЕ

Магдалина Тодорова
magda@fmi.uni-sofia.bg
todorova_magda@hotmail.com
кабинет 517, ФМИ

Работа със символи

1. Предикат eq? за работа със символи

Използването на символи разширява диапазона на структурите от данни, които езикът обработва. Чрез тях се реализират редица важни процедури за символна обработка, като символно диференциране, символно интегриране и др.

За работа със символи се използва предикатът **eq?**. Нарича се още предикат за проверка за ***идентичност***.

Дефиниция: Два символа са идентични (едни и същи), ако се състоят от едни и същи знаци, записани в един и същ ред.

Предикат eq?

Синтаксис:

(eq? c1 c2)

където c1 и c2 са символни изрази.

Семантика:

Оценяват се символните изрази c1 и c2. Ако оценките им са едни и същи символи (печатат се по еднакъв начин), оценката на eq? е #t. В останалите случаи оценката е #f.

Освен за символи, възможно е чрез **eq?** да се провери дали два произволни обекта са идентични. Ще отбележим само, че ако аргументите на eq? са от различни типове, eq? връща #f.

2. Приложения на eq?
Проверка за принадлежност на елемент в списък

Задача. Да се дефинира процедура ***memq***, която има два аргумента – символ и списък. Ако символът не се съдържа в списъка (т.е. не е идентичен с никой елемент на списъка), memq да връща празния списък (**В DrRacket връща #f**). В противен случай memq да връща списък от елементите на дадения списък, започващ с първото срещане на символа и завършващ в края на списъка.

Примери:

```
>(memq 'a '(x 1 2 y))
#f
>(memq 'a '(x t a b c a))
(a b c a)
```

Процедура memq

```
(define (memq item L)
    (cond ((null? L) #f)
          ((eq? item (car L)) L)
          (else (memq item (cdr L)))))
```

Забележка: Повечето реализациите на Scheme имат примитивна процедура **memq**, която реализира същата операция.

Процедура memq

В DrRacket има набор от процедури за проверка за дали елемент принадлежи на списък. Те са подобни на memq.

memq – сравняване чрез eq?

memv – сравняване чрез eqv?

member – сравняване чрез equal?

Примери:

(member 'a '(b a c a d)) ⟶ (a c a d)

(member 'a '(c d e f)) ⟶ #f

(member '(a b) '((a b) (c d e))) ⟶ ((a b) (c d e))

(member '(c d e) '((a b) (c d e))) ⟶ ((c d e))

(member ' a '((a b) (c d e))) ⟶ #f

Проверка за принадлежност на елемент в списък на произволно ниво

Задача. Да се дефинира процедура deep-found?, която установява дали символът item се съдържа в списъка L на произволно ниво.

```
(define (deep_found? item L)
    (cond ((null? L) #f)
          ((atom? (car L)) (if  (eq? item (car L)) #t
                                (deep_found? item (cdr L))))
          (else (or (deep_found? item (car L))
                    (deep_found? item (cdr L))))))
```

Символно диференциране

Символното диференциране има исторически смисъл в езика Lisp. Това е един от примерите, мотивиращи развитието на компютърните езици за символна обработка, какъвто език е Scheme.

Задача. Да се дефинира процедура, която извършва символно диференциране на алгебрични изрази.

Символно диференциране

Искаме процедурата да има за аргументи алгебричен израз и променлива и да връща производната на израза относно променливата.

Например, ако аргументи на процедурата са ax^2+bx+c и x, процедурата трябва да върне алгебричния израз $2ax+b$, който е производната на $ax^2 + bx +$ с относно x.

Символно диференциране

Ограничния:

- ✓ Програмата ще обработва алгебрични изрази, в които се прилагат само операторите за събиране и умножение;
- ✓ Алгебричните изрази ще са само с два аргумента.

Ще използваме подхода абстракция чрез данни.

За целта отначало ще дефинираме програма за символно диференциране, която оперира над абстрактни обекти – суми, произведения, константи и променливи, без да се интересуваме как те се представят. След това ще разгледаме и задачата за представянето.

Първи етап. Диференцирането на произволен израз от посочения вид се извършва чрез прилагане на следните правила:

$$\frac{dc}{dx} = 0, \text{където c е константа или е символ, различен от x}$$

$$\frac{dx}{dx} = 1$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(u.v)}{dx} = u.\frac{dv}{dx} + v\frac{du}{dx}$$

Символно диференциране

За да могат да се включат тези правила в процедура, ще трябва

да може да се разпознава дали един израз е константа, променлива, сума или произведение.

Необходимо е също така:

- на израз, който е сума, да могат да се извличат събираемите;
- на израз, който е произведение, да могат да се извличат множимото и множителя.

Ще трябва също да имаме конструктори на алгебрични изрази, които са сума и произведение.

Символно диференциране

Нека предположим, че са дефинирани процедури, които реализират следните предикати, селектори, конструктори:

а) ***предикати***

(constant? expr) - установява дали изразът expr е константа;

(variable? expr) - установява дали изразът expr е променлива;

(same_variables? v1 v2) - установява дали v1 и v2 са едни и същи променливи;

(sum? expr) - установява дали expr е сума;

(product? expr) - установява дали изразът expr е произведение.

б) ***селектори***

(ad1 expr) - намира първото събираемо на сумата expr;

(ad2 expr) - намира второто събираемо на сумата expr;

(mult1 expr) - намира множимото на произведението expr;

(mult2 expr) - намира множителя на произведението expr.

в) ***конструктори***

(make_sum a1 a2) - конструира сумата на a1 и a2;

(make_product m1 m2) - конструира произведението на m1 и m2.

Като се използват тези процедури, процедурата за намиране производната на израза ***expr*** относно променливата ***var***, има вида:

```
(define (derive expr var)
    (cond ((constant? expr) 0)
          ((variable? expr) (if (same_variables? expr var) 1 0))
          ((sum? expr)
           (make_sum (derive (ad1 expr) var)
                     (derive (ad2 expr) var)))
          ((product? expr)
           (make_sum (make_product (mult1 expr)
                                   (derive (mult2 expr) var))
                     (make_product (derive (mult1 expr) var)
                                   (mult2 expr))))))
```

Втори етап. Конкретизиране на представянето на алгебричните изрази

Съществуват различни начини за представяне на алгебрични изрази чрез списъци.

Едно представяне.

Чрез списъци от символи, които са огледално изображение на обикновените алгебрични изрази. По този начин изразът $ax+b$ се представя чрез списъка (a * x + b).

При това представяне обаче, трудно могат да се реализират конструкторите, селекторите и предикатите, описани по-горе.

Друго представяне.

По-удобно е префиксното представяне на алгебрични изрази.

За израза ax+b, то има вида (+ (* a x) b).

Избираме това представяне за представяне на алгебричните изрази.

а) предикати

```
(define (constant? expr)
    (number? expr))
(define (variable? expr)
    (symbol? expr))
(define (same_variables? v1 v2)
    (and (variable? v1) (variable? v2) (eq? v1 v2)))
(define (sum? expr)
    (if (pair? expr) (eq? (car expr) '+) #f))
(define (product? expr)
    (if (pair? expr) (eq? (car expr) '*) #f))
```

б) селектори

```
(define (ad1 expr)
    (cadr expr))
(define (ad2 expr)
    (caddr expr))

(define (mult1 expr)
    (cadr expr))
(define (mult2 expr)
    (caddr expr))
```

в) конструктори

```
(define (make_sum a1 a2)
    (list '+  a1  a2))

(define (make_product m1 m2)
    (list '*  m1  m2))
```

Символно диференциране

Експерименти:

```
>(derive '(+ x 3) 'x)
(+ 1 0)
>(derive '(* x y) 'x)
(+ (* x 0) (* 1 y))
>(derive '(* (* x y) (+ x 3)) 'x)
 (+ (* (* x y) (+ 1 0)) (* (+ (* x 0) (* 1 y)) (+ x 3)))
```

Вижда се, че резултатите са коректни, но не са опростени. За реализиране на опростяването трябва да се променят само конструкторите.

```
(define (make_sum a1 a2)
    (cond ((and (number? a1)
                (number? a2))
           (+ a1 a2))
          ((number? a1)
           (if (= a1 0) a2 (list '+ a1 a2)))
          ((number? a2)
           (if (= a2 0) a1 (list '+ a1 a2)))
          (else (list '+ a1 a2))))
```

Символно диференциране

```
(define (make_product m1 m2)
  (cond ((and (number? m1)
              (number? m2))
         (* m1 m2))
        ((number? m1)
         (cond ((= m1 0) 0)
               ((= m1 1) m2)
               (else (list '* m1 m2))))
        ((number? m2)
         (cond ((= m2 0) 0)
               ((= m2 1) m1)
               (else (list '* m1 m2))))
        (else (list '* m1 m2))))
```

Намиране стойност на израз

Формула от вида:

<формула> ::= <терминал> |
(<формула><знак><формула>);

<знак> ::= + | - | *| /;

<терминал> ::= 0 | 1 | ... | 9,

може да се представи във вид на двоично дърво, съгласно следното правило:

- формула състояща се от един терминал се представя чрез двоично дърво с един връх – цифрата;
- формула от вида (f1 s f2) се представя чрез двоично дърво, коренът на което е знака s, лявото поддърво съответства на формулата f1, дясното поддърво – на формулата f2.

Да се дефинира процедура, която намира стойността на формула, представена чрез двоично дърво.

Намиране стойност на израз

```
(define (arexp tree)
  (cond ((and  (number? (entry tree))
               (>= (entry tree) 0)
               (<= (entry tree) 9))    (entry tree))
        ((eq? (entry tree) '+)
               (+  (arexp (lefttree tree))
                    (arexp (righttree tree))))
        ((eq? (entry tree) '-)
               (- (arexp (lefttree tree))
                  (arexp (righttree tree))))
        ((eq? (entry tree) '*)
               (* (arexp (lefttree tree))
                   (arexp (righttree tree))))
        ((eq? (entry tree) '/)
               (/ (arexp (lefttree tree))
                  (arexp (righttree tree))))))
```

Намиране стойност на израз

Примерни изпълнения:

Нека

```
(define f1 '(5 () ()))
(define f2 '(+ (- (4 () ()) (2 () ()))
               (* (3 () ()) (/ (8 () ()) (4 ()()))))))
```

```
> (arexp f1)
5
> (arexp f2)
8
```