1. Списъци. Представяне.

Основна конструкция в lisp-подобните езици е S-изразът (точкови двойки?)

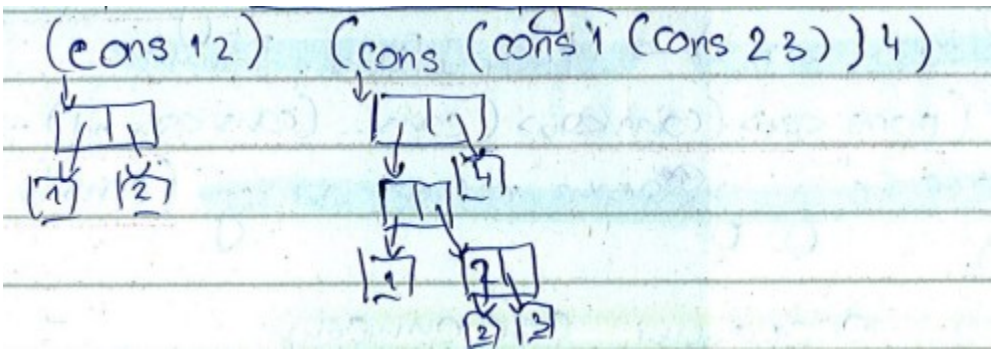
1) атомите (символите, числата, низовете, #t и #f) са S-изрази

2) ако X и Y са S-изрази, то (X.Y) – точковата двойка е S-израз;

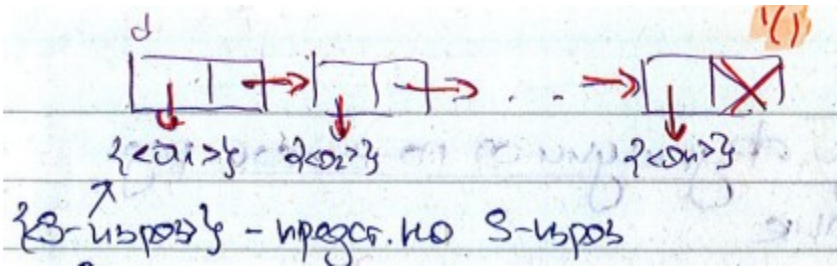
3) няма други S-изрази освен тези, описани в 1) и 2)

S-изразите са най-общият тип данни в Scheme. Списъкът е частен случай на S-израз. S-изразът (X.Y) се нарича точкова двойка. Тоест всеки S-израз е или атом или точкова двойка. Чрез тях можем да представяхме йерархични данни. Чрез процедурата cons(construct) се конструира двойка. (cons a1 a2) е синтаксисът, където a1 и a2 са изрази, чиито оценки са S-изрази.

Примери: Cons е конструктор



Процедурата car(contents address register) намира първи обект от двойката, а cdr (contents decrement register) втория обект тоест те са селектори. (car x), (cdr x), където оценката на x е точкова двойка. Списъците са двойки, които предоставят крайни редици от елементи по следния начин:



Специалната форма quote не оценява аргумента си и го връща като оценка такъв какъвто е без да го променя. Общ вид (quote <S-израз>) или ‘<S-израз>. Например ‘a->a, ‘(a.b)->(a.b), ‘’a->’a.

Записът във вид на точкови двойки дава още един запис

(<a1>.(<a2>.(...).(<an>.nil)....))) или

(cons ‘<a1> (cons ‘<a2> (cons ... (cons ‘<an> ()...))))

Според стандарта на Scheme nil, () и #f са еквивалентни. Също така записваме и като (<a1> <a2> … <an>)

Ако искаме да видим дали един S-израз е еквивалентен на nil ползваме null?

(null? obj) -> {#t, ако [obj] е nil (или (), или #f)

{ #f, иначе

Списъците се конструират и с примитивната процедура list приемаща произволен брой аргументи. Тя конструира и връща като резултат списък от оценките на аргументите си:

(list <a1> <a2> … <an>) -> ([<a1>] … [<an>])

Еквивалентен е на (cons <a1> ….)

Можем с cons да добавяме елемент отпред (cons ‘a ‘(bcd)) -> ‘(abcd)

Основни операции със списъци.

a) Намиране на дължината на списък

```
(define (length l)
    (if (null? l)
        0
        (+ 1 (length (cdr l)))))
```

b) Извличане на n-тия пореден елемент от списък

```
(define (nth n l)
    (if (= n 1)
        (car l)
        (nth (- n 1) (cdr l))))
```

c) Конкатенация на произволен брой списъци (конкретно сега за 2)

```
(define  (append l1 l2)
    (if (null? l1)
        l2
        (cons (car l1) (append (cdr l1) l2)))))
```

d) Обръщане реда на елементите в списък

```
(define (reverse l)
        (define (loop l1 l2)
                (if (null? l1)
                        l2
                        (loop (cdr l1) (cons (car l1) l2))))
        (loop l '()))
```

e) Предикати за проверка на равенство

(eq? s1 s2) -> {#t, ако [s1] и [s2] са идентични (ако са обозначение на един и същ обект, тоест на един и същ участък в паметта)

{#f, иначе

Забележка:

1) Обикновено се използва дали [s1] и [s2] са еднакви символни атоми

2) За сравнение на числа се ползва „=“ или eqv? (обединението на eq? и =)

(equal? s1 s2)-> {#t, ако [s1] и [s2] са еквиваленти S-изрази (тоест или са еднакви символи или са еднакви низове, или равни числа от същия тип, или са точкови двойки с еквивалентни car и cdr части)

{#f, иначе

f) Проверка за принадлежност към списък

(memq item l)->{(), ако [item] не съвпада по eq? с никой от елементите в [l];

{тази част от [l] започваща с това срещане на елемент равен по eq? на [item]

```
(define (memq item l)
        (cond ((null? l) '())
                ((eq? item (car l)) l)
                (else (memq item (cdr l)))))
```

(member item l) – същото като memq, но в смисъл на equal?

Функции от по-висок ред за работа със списъци.

a) Акумулиране на елементите на даден списък

примерно обобщение на:

```
(define (sumList l)
        (if (null? l)
                0
                (+ (car l) (sumList (cdr l))))
(define (accum operation base l)
        (if (null? l)
                base
                (operation (car l) (accum operation base (cdr l)))))
```

b) Изобразяване (трансформиране) на даден списък чрез прилагане на една и съща процедура към всеки от неговите елементи

```
(define (map proc l)
        (if (null? l)
                '()
                (cons (proc (car l)) (map proc (cdr l))))
```

При map няма акумулиране на резултатите

Map връща списък от получените резултати

(map <procedure> <list>)

Действие: оценяват се <procedure> и <list>. Процедурата [<procedure>] се прилага паралелно (едновременно) към всеки от елементите на списъка [<list>] (не се оценяват повторно елементите на [<list>]) и като оценка се връща списък от получения резултат

Пример: (map (lambda (x) (* x x)) '(1 2 3)) -> '(1 4 9)

Процедурата map стои в остновата на една (трета поред след рекурсият и итерацията) от основните стратегии (за управление на изчислителния процес при програмиране на Lisp – т.нар. mapping. Основните характеристики на изоборазяването се свеждат до псевдопаралелно извършване на един и същ тип обработка върху елементите на даден списък и формиране на списък от получените резултати.

c) Филтриране на елементите на списъка

```
(define (filter pred l)
        (cond ((null? l) '())
                ((pred (car l))
                (cons (car l) (filter pred (cdr l)))
```

(else (filter pred (cdr l)))))

d) Прилагане на процедура към списък от аргументи

(apply <procedure> <listOfArgs>)

Действие: Оценяват се <procedure> и <listOfArgs>. Нека [<listOfArgs>] е (arg1, …, argn)

Процедурата apply предизвиква прилагане на процедурите [<procedure>] върху аргументите като при това тези аргументи не се оценяват още един път и връща резултат.

2. Безкрайни потоци и безкрайни списъци.

Потокът е редица от елементи, която може да се дефинира с помощта на основните примитивни операции (процедури) за работа с потоци. Ако x има стойност, равна на (cons-stream a b), то (head x) -> [a], а (tail x) -> [b]. Поток без елементи се означава с the-empty-stream, а примитивен предикат за проверка за празен поток е empty-stream?

Тоест потоците са съставен тип данни с конструкция cons-stream, селектори head – първи елемент от поток и tail – опашката на даден поток тоест потокът получен чрез премахването на 1ви нещо?

Важно е, че при потоците се предполага строго последователен достъп до елементите (от началото до края)

Употребяват се най-вече, когато имаме много (дори безкрайните) елементи

Основни операции и функции от повисок ред.

Конкатенация на потоци

(define (appendStream s1 s2)

(if (empty-stream? s1)

(cons-stream (head s1) (appendStream (tail s1) s2))))

Функции от по-висок ред:

a) Трансформиране на поток (изобразяване)

(define (mapStream f st)

(if (empty-stream? st) the-empty-stream

(cons-stream (f (head st))

(mapStreak f (tail st)))))

Пример:

(define st (cons-stream (cons-stream 2 the-empty-stream)))

(mapStream (lambda (x) (* x x)) st) -> [1 4]

b) Филтриране на поток по дадено условие

```
(define (filterStream pred st)
        (cond ((empty-stream? st) the-empty-stream)
                ((pred (head st)) (cons-stream
                        (head st) (filterStream pred (tail st))
                (else (filterStream pred (tail st)))))
```

c) Формиране на поток от числа в интервал [low, high] от Ints

```
(define (range low high)
        (if (> low high) the-empty-stream
                (cons-stream low (range (+ low 1) high))))
```

Отложено оценяване.

Основава се на „забавено (отложено)“ оценяване, които „пакетират“ съответните изрази (аргументи и резултати) и позволяват тези изрази да бъдат оценени едва кгоато има нужда да бъдат оценени.

В този смисъл cons-stream конструира потока само частично и да предава така конструирания от нея специален обект на програмата, която използва. Ако тази програма се опитва да получи достъп до неконструирана част, то се конструира допълнително само тази част от потока, която е реално необходима за продължаване на процеса на оценяване. Оценяването на останалата част от потока отново се отлага за евентуално възникване на необходимост от достъп до нови елементи и т.н.

За целта се използва специална форма delay. В резултат на оценяването на обръщението (delay <expr>) се получава „пакетиран“ вариант на <expr>, който позволява реално оценяване на <expr> да се извърши едва тогава, коогато това е абсолютно необходимо. По-точно пакетираният израз, който е оценка към обръщението към delay може да бъде използван за възобновяване на оценяването на <expr> с помощта на процедурата fore. Процедурите delay и force са вградени: Още:

- delay “пакетират“ даден израз, така че той да бъде оценен при повикване. Следователно delay е специална форма която не оценява аргумента си и (delay <expr>) -> (lambda () <?>)

- обратно force просто извиква процедурата без аргументи получено чрез delay

```
(define (force do)
        do)
```

Тоест (force (delay expr)) -> (force (lambda () expr)) -> ((lambda () expr)) -> [expr]

Тогава cons-stream може да се престави изцяло като специална форма! (cons-stream a b) -> (cons a (delay b))

head и tail { (define (head st) (car st))

{ (define (tail st) (force (cdr st)))

Работа с безкрайни потоци.

Има два подхода за конструирането им:

- неявно/индиректно – използват се специални процедури - генериран

- явно/директно – рекурсия върху съответния генериран поток (по отношение на вече генерираните части на потока)

а) неявно конструиране на безкрайни потоци

Пример 1: безкраен поток от цели числа

```
(define (integersFrom n)
        (cons-stream n (integersFrom (+ 1 n))))
(define nats (integersFrom 0))
```

Пример 2: безкраен поток от числата на Фибоначи

```
(deifne (fib a b)
        (cons-stream a (fib b (+ a b))))
(define fibs (fib 0 1))
```

nats има стойност точкова двойка с head -> 0 И с tail (lambda () Поток започващ с 1). Така генерирания поток е безкраен, но във всеки един момент работим с негова крайна част. Самата ни програма не знае, че той не е напълно наличен (тъй като никога няма да се наложи да разгледа целия поток)

Намиране на n-ти елемент на безкраен поток

```
(define (nthStream n st)
        (if (= n 1)
                (head st)
                (nthStream (- n 1) (tail st))))
```

Събиране на елементите на 2 потока

```
(define (addStreams s1 s2)
        (cond ((empty-stream? s1) s2)
                ((empty-stream? s2) s1)
                (else cons-stream (+ (head s1 (head s2))
```

(addStreams (tail s1) (tail s2)))))

Умножаване на всички елементи на потока с константа

```
(define (scaleStream c st)
        (mapStream (lambda (x) (*x c)) st))
```

b) Явно конструиране на безкрайни потоци

Използва се рекурсия върху самия генериран поток, а не се използват процедури – генератори

Пример 1: безкраен поток от 1-ци

```
(define ones (cons-stream 1 ones))
```

Пример 2: безкраен поток от естествени числа

```
(define nats
        (cons-stream 1 (addStreams ones nats)))
```

Пример 3: безкраен поток от Fibs

```
(define fibs (cons-stream 0 (cons-stream 1
                (addStreams (tail fibs) fibs))))
```