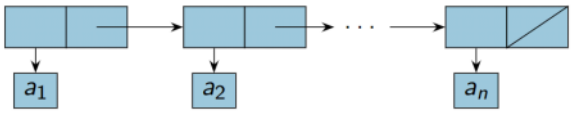
20. Функционално програмиране. Списъци. Потоци и отложено оценяване

1. Списъци. Представяне

Деф: S-израз наричаме
- Атоми (булеви, числа, знаци, символи, низове, функции)
- Наредени двойки $(S_1 \cdot S_2)$, където S_1 и S_2 са S-изрази

S-изразите са най-общият тип данни в Scheme. С тяхна помощ могат да се дефинират произволно сложни структури

Деф: Списъци в Scheme
- Празният списък () е списък
- $(h \cdot t)$ е списък, ако t е списък
  - h - глава на списъка
  - t - опашка на списъка



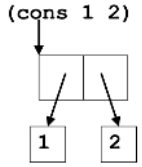
$(a_1 \ . \ (a_2 \ . \ (\ ...(\ a_n \ . \ ()\ )\ )\ )\ ) \iff (a_1 \ a_2 \ ... \ a_n)$

Наредените двойки са частен случай на S-изразите

Наредени двойки в Scheme: $(cons\ \langle израз_1 \rangle\ \langle израз_2 \rangle)$
- Представлява наредена двойка от оценките на $\langle израз_1 \rangle$ и $\langle израз_2 \rangle$

Функции, приемащи наредени двойки в Scheme:
- $(car\ \langle израз \rangle)$ - първият компонент на двойката, която е оценка на $\langle израз \rangle$
- $(cdr\ \langle израз \rangle)$ - вторият компонент на двойката, която е оценка на $\langle израз \rangle$

Всеки обект се представя като указател към клетка.
- Клетка, съответна на примитивен обект, съдържа представянето на този обект
- Клетка, съответна на точкова двойка, съдържа двойка указатели към представянията на *car* и *cdr* на тази точкова двойка



Списък може да се конструира с процедурите cons и list:
- $(list\ \langle a_1 \rangle\ \langle a_2 \rangle\ \ldots\ \langle a_n \rangle) \rightarrow\ ([\langle a_1 \rangle]\ [\langle a_2 \rangle] \ldots [\langle a_n \rangle])$
- Еквивалентно на: $\left(cons\ \langle a_1 \rangle\ \left(cons\ \langle a_2 \rangle\ \left(cons\ \ldots (cons\ \langle a_n \rangle\ '()) \ldots\right)\right)\right)$
- *cons* за добавяне на елемент в началото: $(cons\ 'a\ '(b\ c\ d)) \rightarrow\ (a\ b\ c\ d)$

2. Основни операции със списъци

Достъп до първи елемент и опашка на списък:
- *car* - достъп до първи елемент на списък
- *cdr* - достъп до опашка на списък

Намиране на броя елементи (дължина) на списък
- с примитивна процедура *length*
  $(length\ l) \rightarrow$ число, равно на броя на елементите на списъка l
- Дефинираме сами (в итеративен стил):

```
(define (length l)
   (define (length-iter arg count)
           (if (null? arg)
               count
               (length-iter (cdr arg)
                            (+ 1 count))))
   (length-iter l 0))
```

Обединяване на елементите на произволен брой списъци - примитивна процедура *append*
- $(\text{append}\ l_1\ l_2\ \ldots\ l_n) \rightarrow$ списък, който съдържа елементите на $[l_1]$, следвани от елементите на $[l_2]$, … , елементите на $[l_n]$
- Пример: $(\text{append}\ '(\text{a b})\ '(c\ d)) \rightarrow\ (a\ b\ c\ d)$

Обръщане на реда на елементите на списък - примитивна процедура *reverse*
- $(reverse\ l) \rightarrow$ списък, съставен от елементите на списъка $[l]$, но взети в обратен ред
- Пример: $\left(\text{reverse}\ '((a\ b)\ (c\ d)\ e)\right) \rightarrow (\text{e}\ (c\ d)\ (a\ b))$

Проверка за равенство

- С eq?: $(eq?\ \ s1\ \ s2) \rightarrow \begin{cases} \#t, & \text{ако } s1 \text{ и } s2 \text{ са идентични, т. е. сочат един и същ обект} \\ & \text{(участък в паметта)} \\ \#f, & \text{в противен случай} \end{cases}$

- С equal?: $(equal?\ \ s1\ \ s2) \rightarrow \begin{cases} \#t, & \text{ако } s1 \text{ и } s2 \text{ са еквивалентни } S-\text{изрази} \\ \#f, & \text{в противен случай} \end{cases}$

Проверка за принадлежност към списък

- $(\text{memq item l}) \rightarrow \begin{cases} \#f, & \text{ако } item \text{ не съвпада (в } eq?\text{ смисъл) с никой от елементите на списъка } l \\ & \text{тази част от списъка } l, \text{ която започва с първото срещане на елемент,} \\ & \text{равен (в } eq?\text{ смисъл) на } item \end{cases}$
- $(member\ item\ l)$ - същото действие като memq, но сравнението се извършва с помощта на $equal?$

Примери:

- $(memq\ \ '(a\ b)\ \ '((a\ b)\ c\ d)) \rightarrow \#f$
- $(member\ \ '(a\ b)\ \ '((a\ b)\ \ (c\ d))) \rightarrow ((a\ b)\ (c\ d))$
- $(member\ \ 'a\ \ '(b\ a\ c\ a\ d)) \rightarrow (a\ c\ a\ d)$

Извличане на n-тия пореден елемент от списък в Scheme:

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

В *haskell* списъкът е последователност от елементи от *еднакъв тип* с произволна дължина.
Конструиране, чрез двуместната операция $(:) :: a \rightarrow [a] \rightarrow [a]$ (дясноасоциативна).
Примери:

- $(1 : (2 : (3 : (4 : [\,]))))$
- $[a_1, a_2, \dots, a_n]$ - по-удобен запис на $a_1 : (a_2 : \dots (a_n : [\,]) \dots)$
- [1,2,3,4] = 1:[2,3,4] = 1:2:[3,4] = 1:2:3:[4] = 1:2:3:4:[]

Основни функции в haskell (и съответстващите им в Scheme):

- $head$ - опашка на (непразен) списък - като car
- $tail$ - опашка на (непразен) списък - като cdr
- $null :: [a] \rightarrow Bool$ - проверява дали списъка е празен - като $null?$
- $length :: [\,] \rightarrow Int$ - дължина на списък - като $length$
- $reverse :: [a] \rightarrow [a]$ - обръща списък - като $reverse$
- $elem :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool$ - проверка за принадлежност на елемент към списък - member?
- $init :: [a] \rightarrow [a]$ - списъка без последния му елемент
- $last :: [a] \rightarrow a$ - последния елемент на списъка
- $take :: Int \rightarrow [a] \rightarrow [a]$ - първите n елемента на списъка
- $drop :: Int \rightarrow [a] \rightarrow [a]$ - списъка без първите n елемента
- ...

Отделяне на списъци в haskell:

- Начин за дефиниране на нови списъци, чрез използване на дадени такива
- [⟨израз⟩ | ⟨генератор⟩ {, ⟨генератор⟩ | ⟨условие⟩}]
  - ⟨генератор⟩ е от вида ⟨образец⟩ <− ⟨израз⟩, където
    - ⟨израз⟩ е от тип списък [a]
    - ⟨образец⟩ пасва на елементи от тип a
  - ⟨условие⟩ е произволен израз от тип Bool
  - За всеки от елементите, генерирани от ⟨генератор⟩, които удовлетворяват всички ⟨условие⟩, се пресмята ⟨израз⟩ и резултатите се натрупват в списък
- Пример:
  - $[2 * x \mid x < -[1\,..\,5]\,] \rightarrow [2,4,6,8,10]$
  - $[(x,y) \mid x <- [1,2,3], y <- [5,6,7], x + y <= 8\,] \rightarrow [(1,5),(1,6),(1,7),(2,5),(2,6),(3,5)]$

3. **Функции от по-висок ред за работа със списъци**

Акумулиране (комбиниране) на елементите на списък (Scheme)

- Функция от по-висок ред, натрупваща (комбинираща) по някакво правило елементите на списък.
- Има аргументи:
  - Комбинираща функция combiner
  - Начална стойност init
  - Списък lst
- Пример Scheme:
  $(accum\ *\ 1\ lst)$ → произведението на елементите на lst

```
(define (accum combiner init lst)
   (if (null? lst)
       init
       (combiner (car lst)
                 (accum combiner init (cdr lst)))
))
```

Трансформиране (изобразяване)

- Трансформира списък чрез прилагане на една и съща функция върху всички елементи

```
map :: (a -> b) -> [a] -> [b]
map f l = [ f x | x <- l ]
map _ [] = []
map f (x:xs) = f x : map f xs
```

- Пример haskell:
  - map (^ 2) [1,2,3] → [1,4,9]
  - $map\ (\backslash f\ \rightarrow f\ 2)$ [(^2), (1 +), (∗ 3)] → [4,3,6]

Филтриране

- Филтриране на елементите на списък по предикат

```
filter :: (a -> Bool) -> [a] -> [a]
filter p l = [ x | x <- l, p x ]
```

```
filter _ [] = []
filter p (x:xs)
 | p x       = x : rest
 | otherwise = rest
 where rest = filter p xs
```

- Пример haskell:
  - $filter\ odd$ [1..5] → [1,3,5]
  - $filter$ (\f → f 2 > 3) [(^2), (1 +), (∗ 3)] → [(^2), (∗ 3)]

Акумулиране в haskell - foldr, foldl, foldr1, foldl1

Дясно свиване (foldr)

- $foldr$ :: $\underbrace{(a \rightarrow b \rightarrow b)}_{\text{бинарна операция}} \rightarrow \underbrace{б}_{\text{начална стойност}} \rightarrow \underbrace{[a]}_{\text{списък от стойности за комбиниране}} \rightarrow \underbrace{b}_{\text{тип на резултата}}$
- $foldr\ op\ nv\ [x_1, x_2, \ldots, x_n] = x_1\,'op'\,(x_2\,'op'\,\ldots\,(x_n\,'op'\,nv)\ldots)$
- ```
  foldr _  nv [] = nv
  foldr op nv (x:xs) = x `op` foldr op nv xs
  ```
- Пример:

```
and :: [Bool] -> Bool
and bs = foldr (&&) True bs
```

Ляво свиване (foldl)

- $foldl$ :: $(b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
- $foldl\ op\ nv\ [x_1, x_2, \ldots, x_n] = \left(\ldots\left(\left(nv\ 'op'\ x_1\right)\ 'op'\ x_2\right)\ldots\right)\ 'op'\ x_n$
- ```
  foldl _  nv [] = nv
  foldl op nv (x:xs) = foldl op (nv `op` x) xs
  ```
- Пример:

  $flip\ f\ x\ y = f\ y\ x$

  $reverse = foldl\ \left(flip\ (:)\right)$ [⬚]

Свиване на непразни списъци (foldr1 и foldl1)

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 op [x1, x2, ..., xn] =
x1 `op` (x2 `op` ...(xn−1 `op` xn) ...)
foldr1 _  [x] = x
foldr1 op (x:xs) = x `op` foldr1 op xs
```

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 op [x1, x2, ..., xn] =
(...((x1 `op` x2) ...) `op` xn
foldl1 op (x:xs) = foldl op x xs
```

4. **Отложено оценяване**
Функция, която ще се изчисли и върне някаква стойност в бъдещ момент от изпълнението на програмата се нарича *отложена операция* или *обещание*

Изчислението на обещание може да стане:
- Асинхронно - паралелно с изпълнението на основната програма
- Синхронно - при поискване от основната програма

В Scheme:
Примитивни операции delay и force:
- $(delay\ \langle израз \rangle)$ - връща обещание за оценяването на $\langle израз \rangle$
- $(force\ \langle обещание \rangle)$ - форсира изчислението на $\langle обещание \rangle$ и връща оценката на $\langle израз \rangle$ (примитивна функция)
- Пример:
```
(define undefined (delay (+ a 3)))
(define a 5)
(force undefined) ⟶ 8
```

Обещанията в Scheme имат страничен ефект: "мемоизират" вече изчислената стойност.

5. **Безкрайни потоци и безкрайни списъци. Основни операции и функции от по-висок ред. Работа с безкрайни потоци**
Деф: Поток
Списък, чиито елементи се изчисляват отложено.
По-точно: Поток е празен списък () или двойка $(h \cdot t)$, където:
  - h - произволен елемент (глава на потока)
  - t - обещание за поток (опашка на потока)

При потоците се предполага строго последователен достъп до елементите.

За да дефинираме коректно поток в Scheme ще използваме синтаксиса за дефиниране на специални форми:
(define-syntax ⟨символ⟩
(syntax-rules () {⟨шаблон⟩ ⟨тяло⟩}))
- Дефинира специална форма ⟨символ⟩, т.ч. всякао срещане на ⟨шаблон⟩ се замества с ⟨тяло⟩

Дефиниране на потоци на Scheme:
```
(define-syntax delay
  (syntax-rules () ((delay x) (lambda () x))))

(define-syntax cons-stream
  (syntax-rules () ((cons-stream h t) (cons h (delay t)))))

(define the-empty-stream '())
(define head car)
(define (tail s) (force (cdr s)))
(define empty-stream? null?)
```

Пример за поток в Scheme:
```
(define (enum a b)
  (if (> a b) the-empty-stream
      (cons-stream a (enum (+ a 1) b))))
```

Отлагането на операции позволява създаването на безкрайни потоци:g
- Пример за поток от всички естествени числа:
```
(define (from n) (cons-stream n (from (+ n 1))))
(define nats (from 0))
```

Функции от по-висок ред за потоци:
- Трансформиране (map)
```
(define (map-stream f s) (cons-stream (f (head s))
                                      (map-stream f (tail s))))
```

- Филтриране (filter):

```
(define (filter-stream p? s)
  (if (p? (head s))
    (cons-stream (head s) (filter-stream p? (tail s)))
    (filter-stream p? (tail s))))
```

- Комбиниране (zip):

```
(define (zip-streams op s1 s2)
  (cons-stream (op (head s1) (head s2))
               (zip-streams op (tail s1) (tail s2))))
```

Директна дефиниция на потоци - може да дефинираме потоци с директна рекурсия. Примери:

- `(define ones (cons-stream 1 ones))`
- `(define nats (cons-stream 0 (map-stream 1+ nats)))`

Потоци в haskell:

- Аргументите в Haskell са обещания, които се изпълняват при нужда
- Списъците в Haskell всъщност са потоци

Генериране на безкрайни списъци в Haskell:

- $[a..] \to [a, a+1, a+2, \dots]$
- Примери:
  - $nats = [0..]$
  - $take\ 6\ ['a'..] \to "abcdef"$
- $[a, a+\Delta x..] \to [a, a+\Delta x, a+2\Delta x, \dots]$
- Примери:
  - $evens = [0,2..]$
  - $take\ 7\ ['a','e'..] \to "aeimquy"$