

ФУНКЦИОНАЛНО ПРОГРАМИРАНЕ

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

Двоични дървета.
Процедури от по-висок ред
за работа със списъци

1. Двоично дърво. Дефиниране

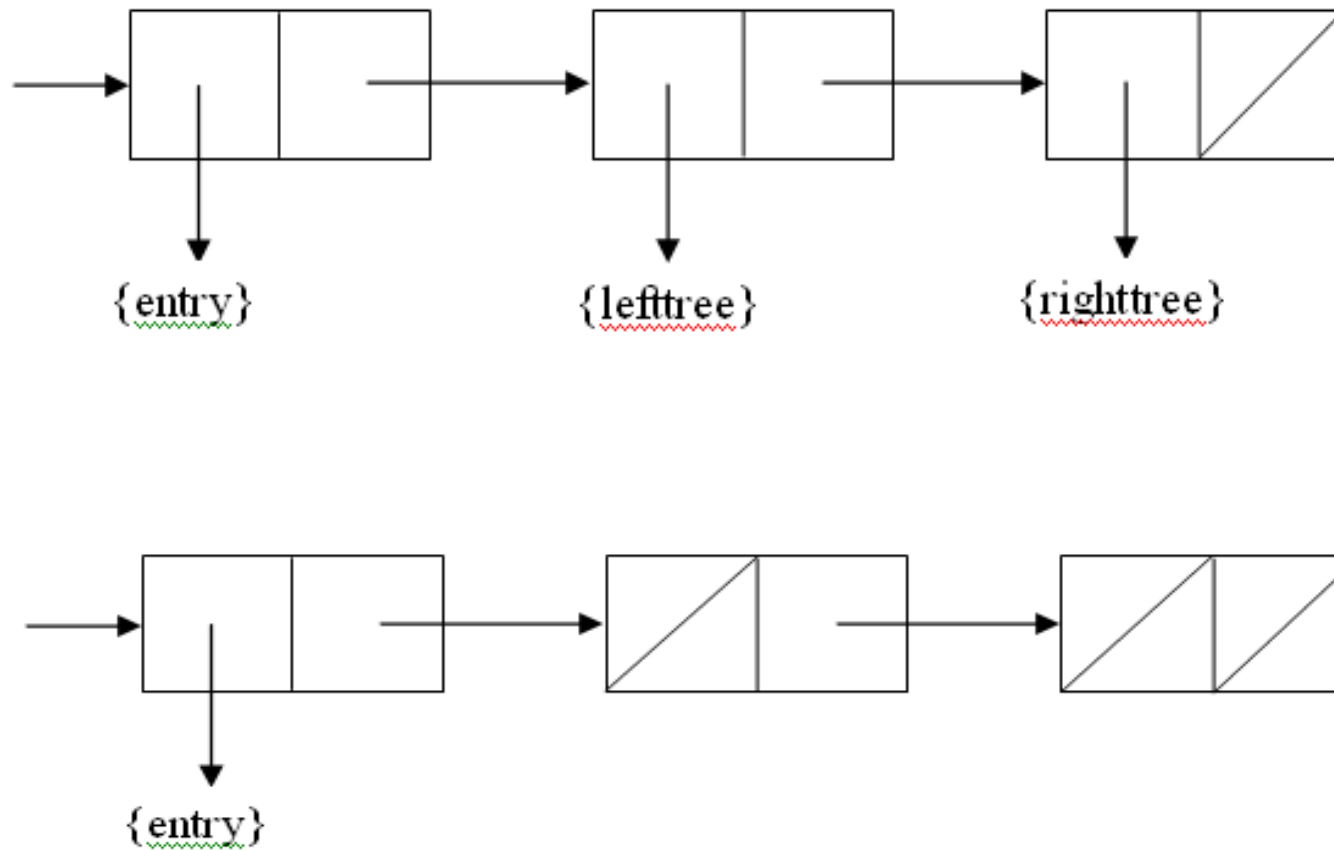
Двоично дърво (ДД)– това е структура, която е или празна, или се състои от данна, наречена *корен*, двоично дърво, наречено *ляво поддърво* (ЛПД) и двоично дърво, наречено *дясно поддърво* (ДПД) на двоичното дърво.

Примитивни операции за работа с двоично дърво:

- ✓ make-tree – конструира ДД по дадени: **корен**, **ЛПД** и **ДПД**;
- ✓ entry – намира корена на непразно ДД;
- ✓ lefttree – намира лявото поддърво на непразно ДД;
- ✓ righttree – намира дясното поддърво на непразно ДД;
- ✓ leaf? – проверява дали непразно ДД е листо;
- ✓ empty-tree? – проверява дали ДД е празно.

Представяне

Непразното двоично дърво може да се представи по следния начин:



Реализация на конструктора и селекторите

Конструктор:

```
(define (make-tree root left right)
  (list root left right))
```

Селектори:

```
(define (entry tree)
  (car tree))

(define (lefttree tree)
  (cadr tree))

(define (righttree tree)
  (caddr tree))
```

Реализация на предикатите

Предикати:

```
(define (empty-tree? tree)
  (null? tree))
```

```
(define (leaf? tree)
  (and (empty-tree? (lefttree tree))
        (empty-tree? (righttree tree))))
```

2. Работа с двоични дървета от числа

Проверка дали дадена стойност x се съдържа във върховете на двоичното дърво t

```
(define (member-tree? x t)
  (cond ((empty-tree? t) #f)
        ((= x (entry t)) #t)
        (else (or (member-tree? x (lefttree t))
                    (member-tree? x (righttree t))))))
```

2. Работа с двоични дървета от числа

Намиране на дълбочината на двоичното дърво t

```
(define (depth t)
  (if (empty-tree? t) 0
      (+ 1 (max (depth (lefttree t))
                 (depth (righttree t))))))
```

Забележка: (max e_1 e_2 ... e_n) намира максималния елемент на редицата от елементи $[e_1]$, $[e_2]$, ..., $[e_n]$. Вградена е в повечето реализации на езика.

```
>(max 12 1 234 15 -78 21)
234
```


2. Работа с двоични дървета от числа

Проверка дали дадена стойност x се съдържа в листата на двоичното дърво t

```
(define (member_leaf? x t)
  (cond ((empty-tree? t) #f)
        ((leaf? t) (= x (entry t)))
        (else (or (member_leaf? x (lefttree t))
                    (member_leaf? x (righttree t))))))
```

2. Работа с двоични дървета от числа

Проверка дали има път от върха a до върха b в двоичното дърво t

```
(define (path? a b t)
  (cond ((empty-tree? t) #f)
        ((= a (entry t)) (or (member-tree? b (lefttree t))
                              (member-tree? b (righttree t))))
        (else (or (path? a b (lefttree t))
                   (path? a b (righttree t))))))
```

3. Процедури от по-висок ред за работа със списъци

➤ *Акумулиране на елементите на даден списък →
accumulate*

Примери:

Събиране на елементите на даден списък от числа

```
(define (sum-list L)
  (if (null? L) 0
      (+ (car L) (sum-list (cdr L)))))
```

Умножаване на елементите на даден списък от числа

```
(define (product-list L)
  (if (null? L) 1
      (* (car L) (product-list (cdr L)))))
```

3. Процедури от по-висок ред за работа със списъци

Двете процедури са написани по следния общ шаблон:

```
(define (<name> L)
  (if (null? L) <null_value>
      (<op> (car L) (<name> (cdr L)))))
```

3. Процедури от по-висок ред за работа със списъци

Могат да бъдат обобщени по следния начин:

```
(define (accumulate op null_value L)
  (if (null? L) null_value
      (op (car L) (accumulate op null_value (cdr L)))))
```

Ако $L = (a_1 a_2 \dots a_n)$, `accumulate` реализира
 $(op a_1 (op a_2 \dots (op a_n \text{null_value}) \dots))$.

accumulate е процедура от по-висок ред, реализираща рекурсивен процес.

3. Процедури от по-висок ред за работа със списъци

итеративен вариант на accumulate

```
(define (accumulate2 op null_value L)
  (define (accum_iter Li acc)
    (if (null? Li) acc
        (accum_iter (cdr Li) (op (car Li) acc))))
  (accum_iter L null_value))
```

Ако $L = (a_1 a_2 \dots a_{n-1} a_n)$, `accumulate2` реализира
(`op` a_n (`op` a_{n-1} ... (`op` a_2 (`op` a_1 `null_value`)) ...)).

3. Процедури от по-висок ред за работа със списъци

Примери:

$(\text{accumulate} + 0 L) \rightarrow$ намира сумата от елементите на $[L]$
(работи като `sum-list`);

$(\text{accumulate} * 1 L) \rightarrow$ намира произведението на елементите
на $[L]$ (работи като `product-list`);

$(\text{accumulate cons '() } L) \rightarrow$ копира елементите на $[L]$ на друго
място в паметта;

$(\text{accumulate append '() } L) \rightarrow$ конкатенира елементите на $[L]$.

3. Процедури от по-висок ред за работа със списъци

Примери:

$(\text{accumulate2} + 0\ L) \rightarrow$ намира сумата от елементите на $[L]$;

$(\text{accumulate2} * 1\ L) \rightarrow$ намира произведението на елементите на $[L]$;

$(\text{accumulate2 cons '()}\ L) \rightarrow$ копира в **обратен ред** елементите на $[L]$ на друго място в паметта;

$(\text{accumulate2 append '()}\ L) \rightarrow$ конкатенира в **обратен ред** елементите на $[L]$.

3. Процедури от по-висок ред за работа със списъци

Примерни изпълнения:

```
> (define L1 (accumulate cons '() '(1 2 3)))
```

```
> L1
```

```
(1 2 3)
```

```
> (define L2 (accumulate2 cons '() '(1 2 3)))
```

```
> L2
```

```
(3 2 1)
```

3. Процедури от по-висок ред за работа със списъци

Примерни изпълнения:

```
> (define L1 (accumulate append '() '((1 2 3) (4 5 6) (7 8 9))))
```

```
> L1
```

```
(1 2 3 4 5 6 7 8 9)
```

```
> (L2 (accumulate2 append '() '((1 2 3) (4 5 6) (7 8 9))))
```

```
> L2
```

```
(7 8 9 4 5 6 1 2 3)
```

3. Процедури от по-висок ред за работа със списъци

➤ *Преобразуване на списък чрез прилагане на една и съща процедура към всеки от елементите на списъка*

map (вградена е)

Ако $L = (a_1 a_2 \dots a_n)$, то $(\text{map } f \ L)$ намира
 $((f \ a_1) (f \ a_2) \dots (f \ a_n))$

```
(define (map f L)
  (if (null? L) '()
      (cons (f (car L)) (map f (cdr L)))))
```

3. Процедури от по-висок ред за работа със списъци

Примери:

```
> (map (lambda (x) (/ 1 (+ 1 (* x x)))) '(1 2 3))  
(1/2 1/5 1/10)
```

```
>(map (lambda (x) (* x x)) '(1 2 3 4 5))  
(1 4 9 16 25)
```

```
(> (map sqrt '(4 25 -16 -25))  
(2 5 0+4i 0+5i))
```


3. Процедури от по-висок ред за работа със списъци

➤ *Филтриране елементите на списък*

Примери:

Даден е списък L от естествени числа. Да се дефинира процедура, която връща списък, съдържащ само онези елементи на L, които са прости числа.

```
(define (prime_list L)
  (cond ((null? L) '())
        ((prime? (car L)) (cons (car L) (prime_list (cdr L))))
        (else (prime_list (cdr L)))))
```



не е вградена,
трябва да я
дефинирате.

3. Процедури от по-висок ред за работа със списъци

Даден е списък от естествени числа L . Да се дефинира процедура, която връща списък, съдържащ онези елементи на L , които са числа на Фибоначи.

```
(define (fib_list L)
  (cond ((null? L) '())
        ((fib? (car L)) (cons (car L) (fib_list (cdr L))))
        (else (fib_list (cdr L)))))
```



не е вградена.

3. Процедури от по-висок ред за работа със списъци

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

```
> (filter odd? '(1 2 3 4 5 6))
```

```
(1 3 5)
```

```
> (filter even? '(1 2 3 4 5 6))
```

```
(2 4 6)
```

4. Прилагане на процедура към списък от аргументи – примитивна процедура *apply*

Синтаксис:

(*apply* <процедура> <списък-от-аргументи>)

Семантика: Оценяват се <процедура> и <списък-от-аргументи>.

Нека [<списък-от-аргументи>] е (*arg1 arg2 ... argn*).

Процедурата *apply* предизвиква прилагане на процедурата [<процедура>] над аргументите *arg1, arg2, ... , argn* и връща получения резултат. При прилагането на [<процедура>] към *arg1, arg2, ... , argn*, тези аргументи не се оценяват още един път.

Примитивна процедура apply

Примери:

(apply + '(2 8)) —> 10

(apply + '(2 3 4 5)) —> 14

(apply + (list (+ 2 3) (- 5 2) (* 3 5))) —> 23

(apply + '(2 (* 3 4))) —> грешка заради комбинацията (* 3 4)

(apply max '(8 4 2 12 3)) —> 12

(apply max '(8 4 2 12 (* 1 3))) —> грешка заради (* 1 3)

(apply append '((1 2 3) (4 5 6) (7) (8 9 10))) —>
(1 2 3 4 5 6 7 8 9 10)

Примитивна процедура *apply*

Пример за използване на apply:

Реализация на процедурата *filter* чрез *apply*

```
(define (filter pred? L)
  (apply append
    (map (lambda (x) (if (pred? x) (list x) '())) L) ))
```

Примитивна процедура `apply`

Пример: Намиране броя на елементите на даден списък, които удовлетворяват даден предикат.

```
(define (count pred? L)  
  (apply + (map (lambda (x) (if (pred? x) 1 0)) L) ))
```

```
> (count even? '(1 2 3 4 5 6 7 8 9))
```

```
4
```

Двукратно оценяване на даден израз – примитивна процедура **eval**

Синтаксис (в по-простия му вариант):

(eval <израз>)

Семантика: Оценява се <израз> в текущата среда и оценката на получената оценка (отново в текущата среда) се връща като резултат, т.е.

(eval <израз>) \longrightarrow [[<израз>]]

Двукратно оценяване на даден израз – примитивна процедура eval

Примери:

(eval (cons '+ '(1 2 3))) \longrightarrow 6

> (define L '(+ 1 2 3))

L

> (define m L)

m

> (eval m)

6

Двукратно оценяване на даден израз – примитивна процедура eval

Примери:

```
> (eval '(+ 1 2))
```

3

```
> (map eval '((* 3 4) (+ 4 2) (- 3 1)))
```

(12 6 2)

(apply + '(2 (* 3 4))) —> грешка заради комбинацията (* 3 4)

(apply + (map eval '(2 (* 3 4)))) —> 14

(apply max '(8 4 2 12 3)) —> 12

(apply max '(8 4 2 12 (* 1 3))) —> грешка заради (* 1 3)

(apply max (map eval '(8 4 2 12 (* 1 3)))) —> 12

5. Дефиниране на процедури с произволен (променлив) брой параметри

Дефинирането на процедури с произволен (променлив) брой параметри може да се извърши с помощта на т. нар. *unrestricted lambda* дефиниция. Използва се някой от следните варианти на *lambda*:

(lambda <параметър> <тяло>)

(lambda (<параметри> . <параметър>) <тяло>)

където <параметър> и <параметри> изпълняват ролята на формални параметри. Съответните им define-специални форми имат вида:

**(define (<име> . <параметър>)
 <тяло>)**

**(define (<име> <параметри> . <параметър>)
 <тяло>)**

5. Дефиниране на процедури с произволен (променлив) брой параметри

Семантика:

(define (<име> . <параметър>)
 <тяло>)

При оценката на обръщение към процедура от горния вид в средата E, се генерира нова среда E1, разширение на E. В E1 параметърът <параметър> се свързва със списък от оценките в E на съответните фактически параметри и в E1 се оценява <тяло>.

5. Дефиниране на процедури с произволен (променлив) брой параметри

Семантика:

(define (<име> <параметри> . <параметър>)
 <тяло>)

Обръщението към дефиниция от този вид трябва да съдържа поне толкова фактически параметъра, колкото са параметрите в частта <параметри>.

5. Дефиниране на процедури с произволен (променлив) брой параметри

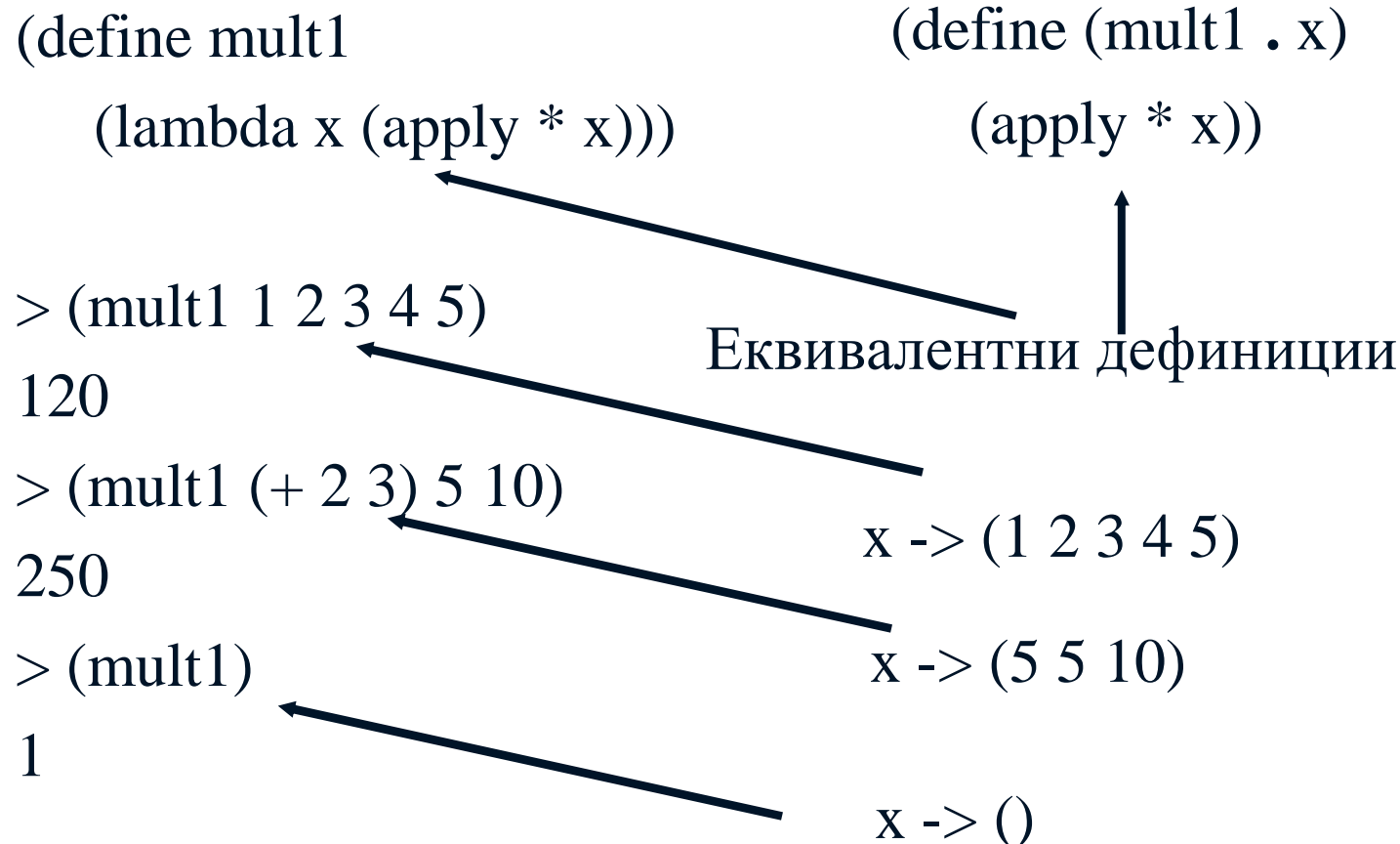
Семантика:

(define (<име> <параметри> . <параметър>)
 <тяло>)

При оценката на обръщение към процедура от този вид в средата E, се генерира нова среда E1, разширение на E. В E1 <параметри> се свързват с оценките в E на зададените фактически параметри, а <параметър> се свързва със списък от оценките в E на останалите фактически параметри и в E1 се оценява <тяло>.

5. Дефиниране на процедури с променлив брой параметри

Пример 1: Да се дефинира процедура **mult1**, която има произволен брой аргументи (чиито оценки са числа) и връща като резултат произведението от оценките на аргументите си.



5. Дефиниране на процедури с променлив брой параметри

Пример 2: Да се дефинира процедура **new-and**, която има произволен брой аргументи и връща конюнкцията от оценките на аргументите си.

```
(define (new-and . as)
```

```
  (help-and as)) ←———— as е списък
```

```
(define (help-and as)
```

```
  (cond ((null? as) #t)
```

```
        ((not (car as)) #f)
```

```
        (else (help-and (cdr as))))))
```

5. Дефиниране на процедури с променлив брой параметри

```
(define (new-and . as)  
  (help-and as))
```

```
(new-and)
```

```
as -> ()
```

```
(new-and (> 3 2))
```

```
as -> (#t)
```

```
(new-and (< 3 5) (= 3 4))
```

```
as -> (#t #f)
```

```
> (new-and)  
#t  
> (new-and (> 3 2))  
#t  
> (new-and (< 3 5) (= 3 4))  
#f  
> (new-and (= 3 3) (> 3 1) #t)  
#t  
> (new-and #t #t #t (< 3 8) (<= 3 3))  
#t  
,
```

5. Дефиниране на процедури с променлив брой параметри

Пример 3: Да се дефинира процедура **my-append**, която предефинира примитивната процедура `append`, конкатенираща произволен брой списъци.

а) *позволяваща 0-ла на брой аргументи*

```
(define (append2 L1 L2)
  (if (null? L1) L2
      (cons (car L1) (append2 (cdr L1) L2))))
```

```
(define (my-append . Li)
  (help-app Li))
```

```
(define (help-app Li)
  (if (null? Li) '()
      (append2 (car Li) (help-app (cdr Li)))))
```

Примери:

б) с поне 1 аргумент

```
(define (append2 L1 L2)
  (if (null? L1) L2
      (cons (car L1) (append2 (cdr L1) L2))))
```

```
(define (my-append L1 . Lr)
  (help-app L1 Lr))
```

```
(define (help-app L1 Lr)
  (if (null? Lr) L1
      (append2 L1 (help-app (car Lr) (cdr Lr)))))
```