**ФУНКЦИОНАЛНО ПРОГРАМИРАНЕ**

**Какво е функционално програмиране?**

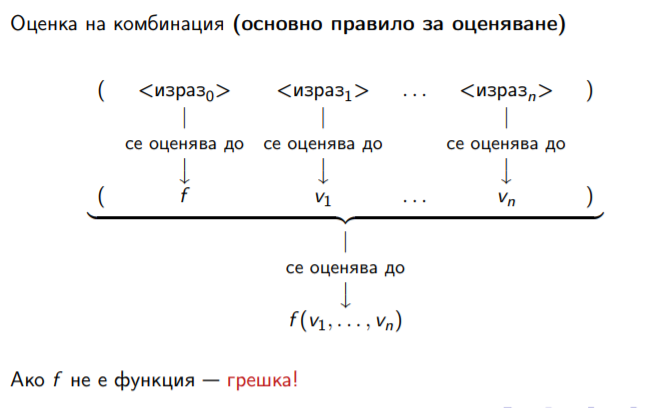
* Фп - Декларативен стил на програмиране (описваме свойствата на желания резултат)
* Във фп има:
* функции с параметри, (абстракция)
* които могат да се прилагат над аргументи, (апликация)
* които могат да са други функции (функции от висок ред)
* и могат да се дефинират чрез себе си, (рекурсия)
* Във фп няма:
* памет
* присвояване
* цикли
* прескачане (goto, break, return)
* видове функционални езици за програмиране :
* Scheme – динамично типизиран (стойностите имат тип), със странични ефекти, стриктно оценяване (първо смята, после връща)
* Haskell – статично типизиран (променливите имат тип), без странични ефекти, лениво оценяване (първо смята, после връща)

**Scheme**

* Програма = списък от дефиниции
* Изпълнение = оценка на израз
* Синтаксис
* Литерали - Булеви константи (#f, #t); Числови константи (15, 2/3, -1.532); Знакови константи (#\a, #\newline); Низови константи ("Scheme", "hi ")
* Символи (f, square, +, find-min)
* Комбинации (<израз1>, <израз2> ... )
* Оценки
* На всеки израз се дава оценка
* Оценката на булевите константи, знаците, числата и низовете са

самите те

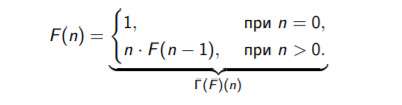
* Оценката на символ е стойността, свързана с него



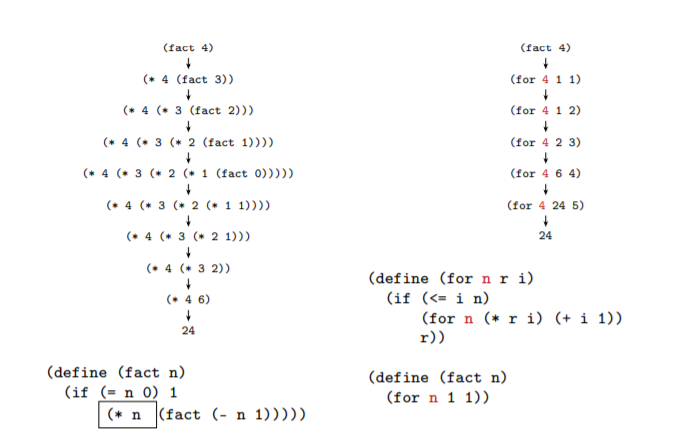
* Специални форми - конструкции, които са изключение от стандартното правило (напр. Define, if, and, or, let, let\*, case, lambda)
* Цитиране - Оценката на (quote <израз>) или ’<израз> е самият <израз>

**Модел на средите и изчислителни процеси**

* Среди в Scheme
* Връзката между символите и техните оценки се записват в речник, който се нарича среда
* Всеки символ има най-много една оценка в дадена среда.
* В даден момент могат да съществуват много среди.
* Символите винаги се оценяват в една конкретна среда.
* Символите могат да има различни оценки в различни среди.
* При стартиране Scheme по подразбиране работи в глобалната среда. В глобалната среда са дефинирани символи за стандартни операции и функции.
* Всяка функция f пази указател към средата E, в която е дефинирана.
* При извикване на f:
* създава се нова среда E1, която разширява E
* в E1 всеки символ означаващ формален параметър се свързва с оценката на фактическия параметър
* тялото на f се оценява в E1
* Всяка среда пази указател към своя “родителска среда”, която разширява
* така се получава дърво от среди
* при оценка на символ в дадена среда E
* първо се търси оценката му в E
* ако символът не е дефиниран в E, се преминава към родителската среда
* при достигане на най-горната среда, ако символът не е дефиниран и в нея се извежда съобщение за грешка
* Рекурсия - Повторение чрез позоваване на себе си



* Теорема (Knaster-Tarski) - Ако Γ е изчислим оператор, то уравнението F = Γ(F) има единствено най-малко решение f (най-малка неподвижна точка на Γ). Нещо повече, решението точно съответства на рекурсивна програма пресмятаща f чрез Γ
* Рекурсивен и итеративен процес



* Функциите, в които има отложени операции генерират същински рекурсивни процеси
* Рекурсивно извикване, при което няма отложена операция се нарича опашкова рекурсия
* Функциите, в които всички рекурсивни извиквания са опашкови генерират итеративни процеси
* При итеративните процеси липсва етап на свиването на рекурсията
* Опашковата рекурсия се използва за симулиране на цикли
* В Scheme опашковата рекурсия по стандарт се интерпретира като цикъл т.е. не се заделя памет за всяко рекурсивно извикване
* Оценка на вложени функции

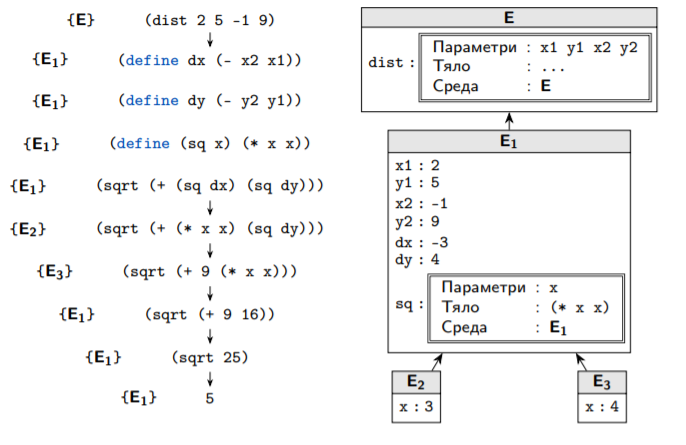
(define (dist x1 y1 x2 y2)

(define dx (- x2 x1))

(define dy (- y2 y1))

(define (sq x) (\* x x))

(sqrt (+ (sq dx) (sq dy))))



* Специална форма let

(let ({(<символ> <израз>)}) <тяло>)

(let ((<символ1> <израз1>)

(<символ2> <израз2>)

. . .

(<символn> <изразn>))

<тяло>)

При оценка на let в среда E:

* Създава се нова среда E1 разширение на текущата среда E
* Оценката на <израз1> в E се свързва със <символ1> в E1
* Оценката на <израз2> в E се свързва със <символ2> в E1
* . . .
* Оценката на <изразn> в E се свързва със <символn> в E1
* Връща се оценката на <тяло> в средата E1

let няма странични ефекти върху средата!

* Специална форма let\*

(let\* ({(<символ> <израз>)}) <тяло>)

(let\* ((<символ1> <израз1>)

(<символ2> <израз2>)

. . .

(<символn> <изразn>))

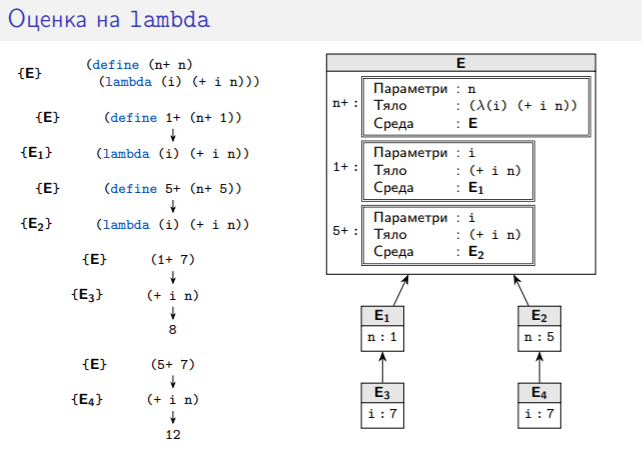
<тяло>)

При оценка на let\* в среда E:

* Създава се нова среда E1 разширение на текущата среда E
* Оценката на <израз1> в E се свързва със <символ1> в E1
* Създава се нова среда E2 разширение на текущата среда E1
* Оценката на <израз2> в E1 се свързва със <символ2> в E2
* . . .
* Създава се нова среда En разширение на текущата среда En−1
* Оценката на <изразn> в En−1 се свързва със <символn> в En
* Връща се оценката на <тяло> в средата En

**Функции от по-висок ред**

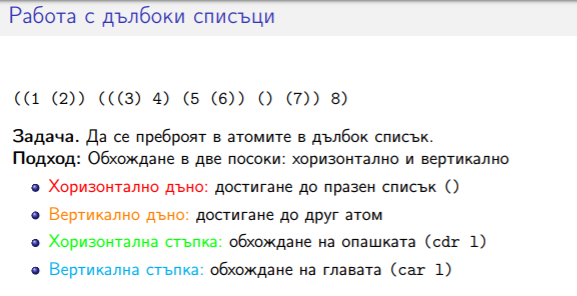
* Функция, която приема функция за параметър се нарича функция от по-висок ред
* Всички функции в λ-смятането са от по-висок ред
* Анонимни функции
* (lambda ({<параметър>}) <тяло>)
* Оценява се до функционален обект със съответните параметри и тяло
* Анонимната функция пази указател към средата, в която е оценена



* Специалната форма lambda е достатъчна за реализацията на (почти) всички конструкции в Scheme!

**Списъци**

* Наредени двойки
* (cons <израз1> <израз2>) – наредена двойка от оценките на израз1 и израз2
* (car <израз>) – първият компонент на двойката, която е оценката на израз
* (cdr <израз>) - вторият компонент на двойката, която е оценката на израз
* (pair? <израз>) – проверява дали оценката на израз е наредена двойка
* S-изрази – атоми (булеви, числа, символи, низове, функции), наредени двойки от S-изрази
* Най-общият вид на данните в Scheme
* С тяхна помощ могат да се дефинират произволно сложни структури от данни
* Списъци
* Празният списък () е списък
* (h . t) е списък, ако t е списък
* (a1 . (a2 . ( ...( an . () ) ) ) ) ⇐⇒ (a1 a2 ... an)
* (null? l) – дали е празният списък
* (list? l) – дали е списък
* (list {<израз>}) – построява списък от {<израз>}
* ( ) не е наредена двойка! (не може car cdr)
* Съкратени форми за до 4 прилагания на car I cdr
* (length l) – връща дължината на списъка
* (append {l}) – конкатенира всички списъци
* (reverse l) – елементите а списъка в обратен ред
* (list-tail l n) – елементите на списъка без първите n
* (list-ref l n) – n-тия елемент на списъка (от 0)
* (member x l) – ако х е елемент на списъка връща списъка от х нататък; ако не е връща false; сравнява с equal
* (memv x l) - eqv
* (memq x l) -eq
* Равенство
* (eq? <израз1> <израз2>) – връща истина, когато оценките на 1 и 2 заемат едно и също място в паметта
* (eqv? <израз1> <израз2>) – връща истина, когато оценките на 1 и 2 заемат едно и също място в паметта или са едни и същи по стойност атоми (дори и да заемат различно място в паметта)
* (equal? <израз1> <израз2>) – връща истина, когато оценките на 1 и 2 са едни и същи по стойност атоми или наредени двойки, чиито компоненти са равни в смисъла на equal? – проверява за равенство на списъци
* Eq? -> eqv? -> equal?



(define (deep-foldr nv term op l)

(cond ((null? l) nv)

((atom? l) (term l))

(else (op (deep-foldr nv term op (car l))

(deep-foldr nv term op (cdr l))))))

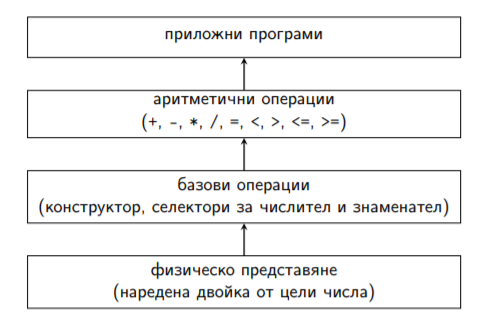
Как работи deep-foldr?

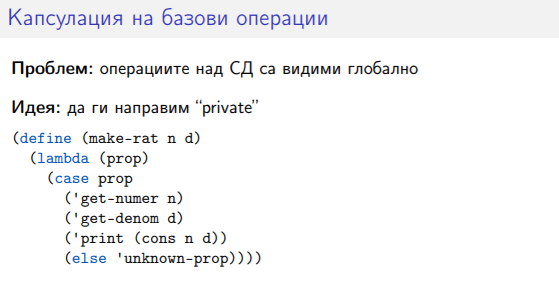
* пуска себе си рекурсивно за всеки елемент на дълбокия списък
* при достигане на вертикално дъно (атоми) прилага term и
* събира резултатите с op
* Приемане на произволен брой аргументи
* (lambda <списък> <тяло>) - създава функция с <тяло>, която получава <списък> от параметри
* (lambda ({<параметър>}+ . <списък>) <тяло>) - създава функция с <тяло>, която получава няколко задължителни <параметър> и <списък> от опционални параметри
* (define (<функция> . <списък>) <тяло>) - еквивалентно на (define <функция> (lambda <списък> <тяло>))
* (define (<функция> {<параметър>}+ . <списък>) <тяло>) еквивалентно на (define <функция> (lambda ({<параметър>}+ . <списък>) <тяло>))
* Оценяване на списък като комбинация
* (eval <списък> <среда>) - връща оценката на оценката на <списък> в <среда>
* (interaction-environment) — текущата среда, в която оценяваме
* (define (evali x) (eval x (interaction-environment)))
* Програмите на Scheme могат да се разглеждат като данни

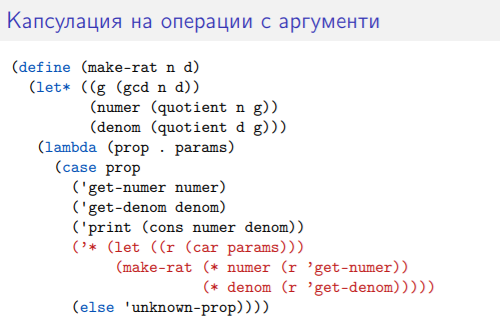
**Структури от данни**

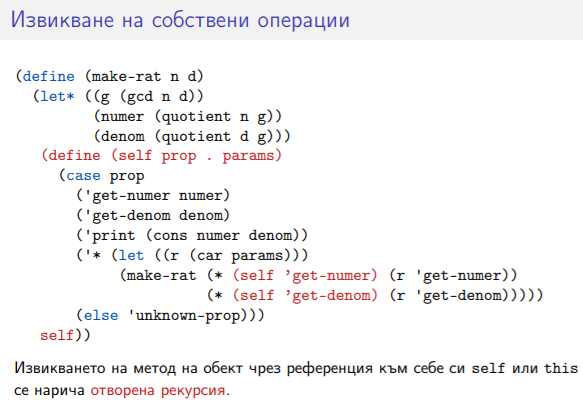
* Абстракция - Принцип за разделянето (“абстрахирането”) на представянето на дадена структура от данни (СД) от нейното използване.
* основен принцип на обектно-ориентираното програмиране
* позволява използването на СД преди представянето и да е ` уточнено
* предимства: програмите работят на по-високо концептуално ниво със СД; позволява алтернативни имплементации на дадена СД, подходящи за различни видове задачи; влиянието на промени по представянето е ограничено до операциите, които “знаят” за него; подобрения при представянето автоматично се разпространяват до по-горните нива на абстракция

Нива на абстракция

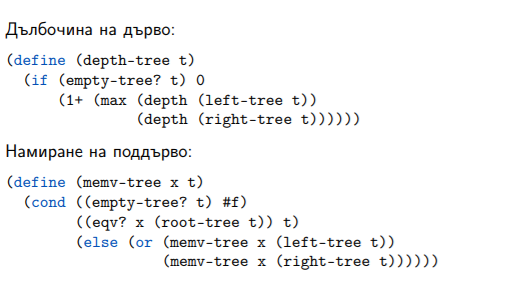
****

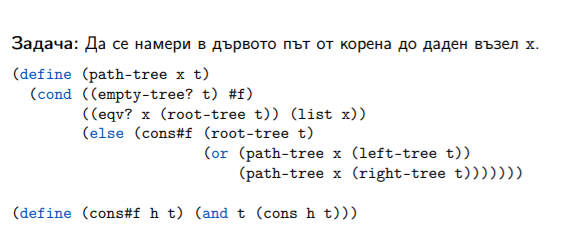
****

****

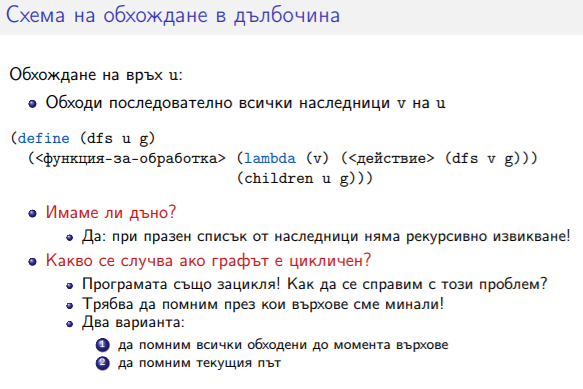
****

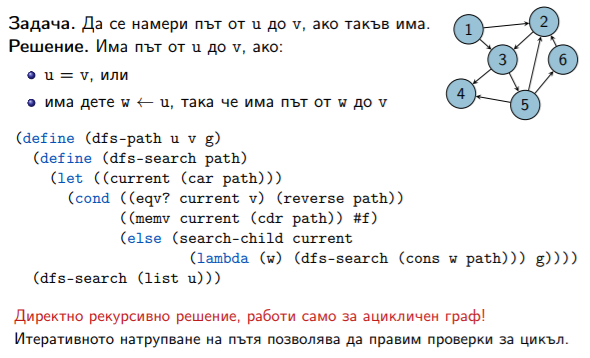
* Представяне на двоични дървета - Представяме двоични дървета като вложени списъци от три елемента : (<корен> <ляво> <дясно>)

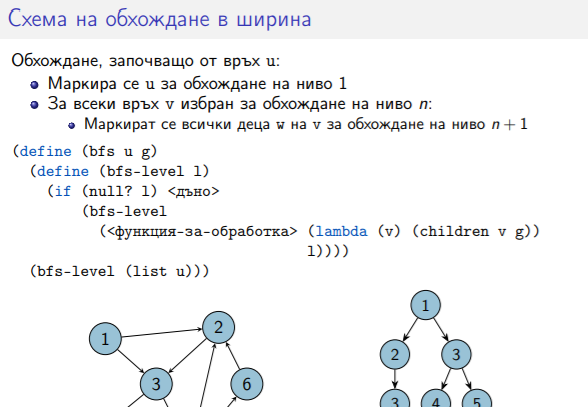




* Асоциативни списъци - (още: речник, хеш, map) са списъци от наредени двойки (<ключ> . <стойност>); <ключ> и <стойност> може да са произволни S-изрази
* (assoc <ключ> <асоциативен-списък>) - Ако <ключ> се среща сред ключовете на <асоциативен-списък>, връща първата двойка (<ключ> . <стойност>); Ако <ключ> не се среща сред ключовете, връща #f; Сравнението се извършва с equal?
* (assv <ключ> <асоциативен-списък>) - също като assoc, но сравнява с eqv?
* (assq <ключ> <асоциативен-списък>) - също като assoc, но сравнява с eq?Графи - Асоциативен списък, в който ключовете са върховете, а стойностите са списъци от техните деца.







**Мутиращи операции –** позволяват въвеждането на странични ефекти

* set! — промяна на оценка, свързана със символ
* set-car!, set-cdr! — промяна на компоненти на точкови двойки
* begin — последователност от действия
* open-input-file, open-output-file — работа с файлове
* read, write, display — вход и изход
* set! - (set! <символ> <израз>) // губи се референциалната прозрачност
* Търси се <символ> във веригата от среди
* Ако бъде намерен, свързва се с оценката на <израз>
* В противен случай — грешка
* (set-car! <двойка> <израз>) / (set-cdr! <двойка> <израз>)
* Съответният компонент на <двойка> се променя да сочи към оценката на <израз>

**Потоци**

* Отложени операции - механизъм да подготвяме операциите и да ги изпълняваме само при нужда
* Обещание - Функция, която ще изчисли и върне някаква стойност в бъдещ момент от изпълнението на програмата. Нарича се още promise и отложена операция.
* Изчислението на дадено обещание може да стане : паралелно с изпълнението на основната програма (асинхронно); при поискване от основната програма (синхронно)
* (delay <израз>) - връща обещание за оценяването на <израз> (специална форма)
* (force <обещание>) - форсира изчислението на <обещание> и връща оценката на <израз> (примитивна функция) (Обещанията в Scheme имат страничен ефект: “мемоизират” вече изчислената стойност)
* Поток - Списък, чиито елементи се изчисляват отложено. По-точно: Поток е празен списък () или двойка (h . t), където
* h — е произволен елемент (глава на потока)
* t — е обещание за поток (опашка на потока)
* (define the-empty-stream '())
* (define (cons-stream h t) (cons h (delay t)))
* (define head car)
* (define (tail s) (force (cdr s)))
* (define empty-stream? null?)
* cons-stream трябва да оценява само първия си аргумент!
* cons-stream трябва да е специална форма
* (define-syntax delay

(syntax-rules () ((delay x) (lambda () x))))

* (define-syntax cons-stream

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

* Дефиниране на специални форми
* (define-syntax <символ>

(syntax-rules () {(<шаблон> <тяло>)}))

* дефинира специална форма <символ> така, че всяко срещане на <шаблон> се замества с <тяло>
* define-syntax има и други, по-сложни форми
* Повечето специални форми на Scheme могат да се дефинират с define-syntax
* Безкрайни потоци

