# 1. Характерни особености на функционалния стил на програмиране. Основни компоненти на функционалните програми.

Функционалното програмиране е начин за съставяне на програми, при който:

1) единственото действие е обръщението към функция

2) единственият начин за разделяне на програмата на части е въвеждането на име на функция и задаването на това име на израз, който пресмята стойността на функцията

3) единственото правило за композиция е суперпозиция на функции

Функция разбираме програмна част (единица), която връща резултат (тук разбираме „чисти“, тоест без странични ефекти още се наричат строги – strict). Във ФП не се ползват цикли, оператори за присвояване, блок-схеми, заделяне на клетки в паметта и т.н.

Дефиниране и използване на функции

Накратко функционалният стил се състои от:
- дефиниране на функции пресмятащи (връщащи) стойности. При това стойности еднозначно се определят от стойностите на съответните аргументи (deterministic and consistent)

- прилагане на функции върху подходящи аргументи, които също могат да бъдат обръщения към функции, тъй като всяка функция връща стойност. Още фунционалните езици се наричат апликативни.

Основни плюсове на ФС на програмиране:

- лесно тестване на модификация на програма тъй като няма странични ефекти

- подходящ за проектиране на езици за програмиране

- математически могат да бъдат доказвани свойства на програмите написани на такива езици

Основни минуси:

- многократно пресмятане на повтарящи се подзадачи

- неефективно решаване на процедурни/алгоритмични задачи

# Примитивни изрази.

Това са най-простите (атоми) на езика

В Scheme(Racket) са:

- символите – крайни редици от знаци, които не съдържат кавички, скоби и интервали. Пазят се за имена на обекти, данни и процедури, атом

- числа – цели и реалним, атом

- низове – редица от знаци оградени в кавички, атом

- списъци – редица от обекти разделени от интервали/оградени в скоби

## Средства за комбиниране и абстракция.

Комбинации – израз, конструиран като списък от примитивни изрази, най-левият елемент е процедура

Обръщение към функция/процедура

(<означение на функция>, <arg1> …. <argn>)

(<оператор>, <операнди>)

Стойността на обръщението към функцията (комбинация) се получава чрез апликация на процедурата (оператор) към аргументите

Приет е префикесн запис – (+ 137 343); (* 5 333)

Как работи интерпретаторът на Scheme

- read
- evaluate
- print

Средства за абстракция – чрез тях обекти на езика могат да се именуват и обработват като едно цяло.

Пример е (define <name> <expr> )

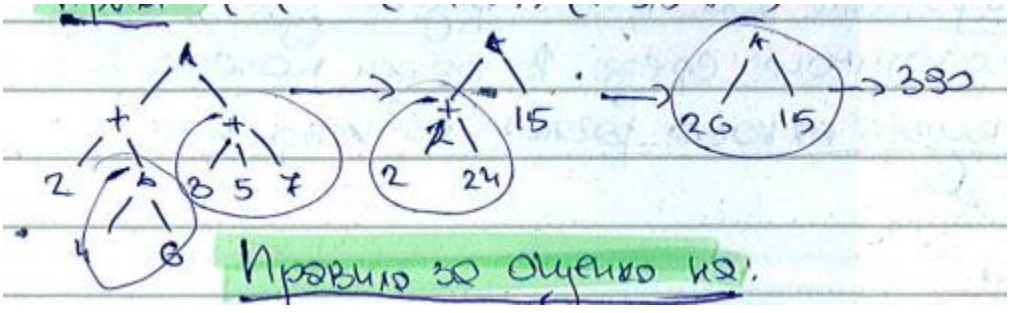
Позволява да се именуват резултати от обращения към функции-комбинации. Интерпретатор чете define <expr> и го оценява като извеща върху екрана <name>. Оценяването се извършва като <expr> се оценява в средата, в която е define и в същата среда се записва <name> заедно със стойността на evaluate(<expr>) [<expr>] така пишем оценката на израза

## Оценяване на израз.

Общото правило за оценяване на комбинация:

- оценяват се подизразите на комбинацията
- прилага се операторът към операндите, които са оценките на предните подизрази

Пример: (*(+2 (*4 6)) (+ 3 5 7))



Правило за оценка на:

- число – оценката е самото число

- низ – оценката е самият низ

- имена (променливи) – от стойностите свързани с тези имена в текущата среда

- вграден оператор – оценката е поредица от машинни инструкции, които реализират оператора

## Дефиниране на функция и оценяване на приложение на функция.

Дефиниране на променливи се извършва с define

> (define pi 3.14)

pi

> pi

3.14

За дефиницията на процедура се използва отново define

(define <name> <formal parameters> <body>)

Оценката на обръщението към дефине съвпада с <name>. В процеса на оценяване <name> се свързва с тялото на дефиницията в текущата среда. Така <name> е име на съставна процедура.

Пример:

> (define (square x) (x*x))

Съставните процедури се използват по същия начин като вградените или още примитивните (неразличими за интерпретатора)

## Модели на оценяване.

Интерпретаторът не различава примитивните от съставните процедури. Съотшетно при оценяването на комбинация интерпретатора оценява елементите на комбинацията и прилага процедурата, която е стойност на оператора на комбинацията към аргументите, които са стойности на операндите на комбинацията.

Механизмът за прилагане на примитивната процедура към аргументите! е вграден в интерпретатора и е част от семантиката на примитивните процедури. При прилагане на съставна процедура към аргументите се оценява тялото на процедурата като предварително формалните се заместват с фактическите параметри и след това се следва меахнизма на оценяване на комбинацията. Това правило стои в основоната на модела на заместването при оценка на обръщение към съставни (дефинирани) процедури. Същността му е:

- при оценяване на комбинация най-напред се оценяват подизразите на тази комбинация и след това оценката на оператора се прилага към оценката на операндите.

- прилагането на дадена съставна процедура към получените аргументи се извършва като съгласно предното правило за оценка на комбинация се оценяват последователно изразите на тялото на процедура като формалните параметри са заместени със съставните фактически. Оценката на последния израз от тялото става оценка на обръщението към съставната процедура.

## Апликативно (стриктно, call-by-value) и нормално (лениво, call-by-name) оценяване.

Възможни са два подхода за оценка на комбинация във въведения модел на оценяване чрез заместване:

- апликативен подход (strict): Отначало се оценяват операндите (аргументите) и след това към получените оценки се прилага оценката на оператора

- нармален подход (lazy): Замества се името на процедурата с тялото ѝ докато се получат означения на примивни процедури и след това се прилагат техните правила за оценка

Забележка:

1) Заместването на формалните с фактически се извършва в рамките на локалната среда

2) Оценяването чрез заместване не е валидно при използване на някои оператори (прим. За присвояване?)

3) При нормалния подход трябва да се (нещо си? – внимава?) да не се оценяват едни и същи изрази многократно

Lisp използва апликативен подход

Пример:

(define (square x) (* x x))

(define (sumOfSq x y) (+ (square x) (square y)))

(define (f x) (sumOfSq (+ x 1) (* x 2)))

Апликативен подход:

(f 5) -> (sumOfSq (+ x 1) (* x 2)) -> (sumOfSq (+ 5 1) (* 5 2)) -> (sumOfSq 6 10) -> (+ (square 6) (square 10)) -> (+ (* 6 6) (* 10 10)) -> (+ 36 100) -> 136

Нормален подход:

(f 5) -> (sumOfSq (+ x 1) (* x 2)) -> (sumOfSq (+ 5 1) (* 5 2)) -> (+ (square (+ 5 1)) (square (* 5 2))) -> (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2 ))) -> (+ (* 6 6) (* 10 10)) -> (+ 36 100) -> 136

## 2. Функции от по-висок ред.

Основните средства за абстракция във ФП са процедурите. Те също могат да поемат като аргументи процедури или да връщат като стойност процедури. Тези процедури, които манипулират други процедури се наричат процедури от по-висок ред.

# Функциите като параметри и оценки на обръщения към функции.

Примерно процедурите имат един и също образец

```
(define (sumInt a b) (if (> a b) 0 (+ a (sumInt (+ a 1) b))))
```

```
(define (sumCube a b) (if (> a b) 0 (+ (* a a a) (sumCube (+ a 1) b))))
```

```
(define (<name> a b) if (> a b) 0 (+ (<f> a) (<name> (<next> a) b))))
```

Така (define (sum f a next b) (if (> a b) 0 (+ (f a) (sum f (next a) b)))) и

```
(define (sumCube a b)
        (define (cube x) (* x x x))
        (define (inc x) (+ x 1))
        (sum cube a inc b))
```

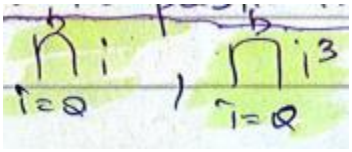
Sum дефинира линеен рекурсивен процес.

Може и итеративно:

```
(define (sumIter term a next b)
        (define (iter a result)
                (if (> a b)
                        result (iter (next a) (+ (term a) result))))
        (iter a 0))
```

Като оценки на обръщения към процедури:

Още по-голяма изразителна сила. Примерно, ако искаме да направим шаблонна процедура, в която да опаковаме sum 0 и да можем и по-различни процедури да получим. Примерно



Тогава:

```
(define (accum operation base)
        (define (returnfunc f a next b)
                (if (> a b) base
                        (operation (f a) (returnfunc f
                                                (next a) next b))))
        returnfunc)
```

> (display ((accum + 0) id 5 inc 8))

```
(define (accumulate op init)
        (define (result f a next b)
                (if (> a b)
                        init
                        (op (f a) (result f
                                (next a) next b))))
```

> ((accumulate + 0) id 5 inc 8))

## Анонимни (ламбда) функции

(lambda (<formal parameters>) <body>)

Оценката на обръщения към ламбда функции е:

Процедурата не се свързва с име в средата. Оценяването на комбинация, чийто оператор е lambda израз става така – ((lambda (x1 ... xn) <body>) a1 … an). В процеса на оценка на комбинацията всички срещания на xi се заменят с [ai] и се оценява получения израз.

(define <name> (lambda (<formal params>) <body>)) <->

(define (<name> <formal params>) <body>) //syntactic sugar