19.1. Функционално програмиране: Основни конструкции в езиците за функционално програмиране. Функции.

Характерни особености на функционалния стил на програмиране. Основни компоненти на функционалните програми. Примитивни изрази. Средства за комбиниране и абстракция. Оценяване на изрази. Дефиниране на променливи и процедури. Среди. Специални форми. Модели на оценяване на комбинации. Процедури от по-висок ред. Процедурите като параметри и оценки на обръщения към процедури.

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

Функционалното програмиране е начин за съставяне на програми, при който единственото действие е обръщението към функции, единственият начин за разделяне на програмата на части е въвеждането на име на функция и задаването за това име на израз, който пресмята стойността на функцията, а единственото правило за композиция е суперпозицията на функции. Тук под функция се разбира програмна част, която "връща" резултат (по-точно, във функционалното програмиране се работи с т.нар. строги функции, които не предизвикват никакви странични ефекти, а само връщат стойности). Във функционалното програмиране не се използват оператори за присвояване и за цикъл, блок-схеми, предаване на управлението.

Най-съществени елементи на функционалния стил на програмиране (по-точно, на програмирането във функционален стил) са дефинирането и използването на функции. Не се използват никакви клетки от паметта и оператори за присвояване и за цикъл, не се описват действия като предаване на управлението и т.н.

Програмирането във **функционален стил** се състои от:

- Дефиниране на функции, които пресмятат (връщат) стойности. При това тези стойности еднозначно се определят от стойностите на съответните аргументи (фактически параметри);

- Прилагане (апликация) на тези функции върху подходящи аргументи, които също могат да бъдат обръщения към функции, тъй като всяка функция връща стойност. Затова езиците за функционално програмиране се наричат още апликативни езици.

Основни предимства на функционалния стил на програмиране:

- Може да се извършва лесна проверка и поправка на съответните програми поради липсата на странични ефекти;

- Подходящ при проектирането на езици за програмиране, предназначени за многопроцесорни компютри, в които много от пресмятанията се извършват паралелно (поради липсата на странични ефекти не се налага програмистът да се грижи за евентуални грешки при синхронизацията, причинени от промени на стойности, извършени в неправилен ред);

- Могат да бъдат доказвани точно (с математически средства) свойства на функционалните програми.

Основни недостатъци на функционалното програмиране:

- Строгата функционалност понякога изисква многократно пресмятане на едни и същи изрази;

- Неестествено и често неефективно е използването им при решаване на задачи от процедурен (алгоритмичен) характер.

Примитивни изрази.

Примитивните изрази или само **изрази** са най-простите, атомарните единици на езика. В езика Scheme **примитивни изрази**/**обекти** са: символите (крайни редици от знаци, които не могат да съдържат кавички, скоби и интервали; числата не са символи; чрез символите се означават имена на обекти, данни и процедури), числата (цели и реални), низовете (редици от знаци, оградени в кавички) и списъците (редици от обекти, разделени с интервали, оградени в кръгли скоби). Символите, числата и низовете се наричат **атоми**.

Примери:
```
> 45678                    ; Въведен е примитивен израз – число.
45678                      ; Оценката му съвпада с въведеното число.
> “This is a string.”      ; Въведен е примитивен израз – символен низ.
“This is a string.”        ; Оценката му съвпада с въведения низ.
> +                        ; Въведен е примитивен израз – вградена
#<procedure +>             ; функция (примитивна процедура).
```

Комбинации.

Израз, конструиран като списък от примитивни изрази, най-левият елемент на които е процедура, се нарича **комбинация** (обръщение към функция/процедура). Най-левият елемент се нарича оператор, а другите елементи – операнди. Стойността (оценката) на комбинацията се получава чрез прилагане (апликация) на процедурата, зададена чрез оператора, към аргументите, които са стойности на операндите.

(<означение на функция> <арг1> <арг2> ... <аргn>)
оператор операнди

Забележка: В Scheme термините функция и процедура се използват като синоними.

Примери:
```
> (+ 137 349)              ; Въведена е комбинация – примитивна
486                        ; процедура (+, -, *, ... ) се прилага върху
> (* 5 33 3)               ; числови аргументи.
495
```

В езика Lisp (и в частност в Scheme) е възприет префиксен запис на изразите. По-съществени предимства на префиксния запис:

- Лесно се използват процедури с произволен брой аргументи. Пример: **(+ 2 4 8 7 6)**

- Лесно се записват вложени изрази (всеки от операндите също може да бъде комбинация с достатъчно сложна структура). Пример: **(* (+ 2 (* 2 4)) (+ 3 5 7))**

Резюме на работата на интерпретатора на Scheme: Интерпретаторът работи в цикъл, като на всяка стъпка от този цикъл извършва следните действия: чете израз (read); оценява го (evaluate); отпечатва (извежда) резултата (print). Затова често се казва, че интерпретаторът изпълнява REP-цикъл (цикъл Read-Evaluate-Print). Този цикъл се изпълнява автоматично, без за целта да се дават специални указания от страна на потребителя.

Средства за абстракция.

Чрез **средствата за абстракция** обекти на езика могат да се именуват и обработват като едно цяло. Едно такова средство е ***define***. ***define*** позволява да се именуват резултати от съставни операции (комбинации). ***define*** позволява да се именуват резултати от съставни операции (комбинации).

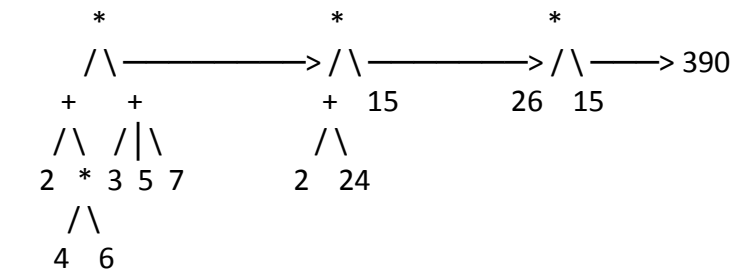
Синтаксис: **(define <име> <израз>)**, където **<име>** е произволен символ (препоръчва се да е различен от имената на примитивните процедури на езика), а **<израз>** е произволен примитивен израз. Семантика: Интерпретаторът чете ***define***-израза, оценява го и извежда върху екрана **<име>**. Процесът на оценяване се състои в следното: в средата, в която define-изразът се оценява, се записва <име> и се свързва с оценката на <израз> в същата среда.

Оценяване на изрази.

Общо правило за оценяване на комбинациите: оценяват се подизразите на комбинацията; прилага се процедурата, която е оценка на най-левия подизраз (т.е. операторът), към операндите (аргументите), които са оценки на останалите подизрази.

Пример: оценяването на **(* (+ 2 (* 4 6)) (+ 3 5 7))** изисква прилагане на общото правило върху четири различни комбинации.

Оценяването на горната комбинация може да бъде представено по следния начин:



Правила за оценяване на числа, низове, имена и вградени оператори:

- Оценката на дадено число е самото число;
- Оценката на даден низ е самият низ;
- Оценката на даден вграден оператор е поредицата от машинни инструкции, които реализират този оператор;
- Оценките на останалите имена (в частност, оценките на променливите) са стойностите, свързани с тези имена в текущата среда.

Оценките на вградените оператори също могат да се смятат за част от средата.

Дефиниране на променливи и процедури.

Дефинирането (задаването, именуването) **на променлива** се извършва с помощта на оператора (примитивната процедура) ***define***.

Пример:

```
> (define size 2)
size
```

> size
2

Синтаксис: Обръщението към ***define*** в случая на дефиниране (именуване) на променлива изглежда по следния начин: **(define <име> <израз>)**

Семантика (механизъм на оценяване): Интерпретаторът свързва името **<име>** със стойността (оценката) на **<израз>** (в горния пример свързва името **size** със стойността **2**). Оценката на обръщението към ***define*** съвпада с името на дефинираната променлива или по-точно, с дефинираното име (дефинирания символ). Както ще покажем по-нататък, ***define*** се използва за дефиниране не само на променливи. Още примери:

> (define pi 3.14159)	**> (define lngth (* 2 pi radius))**
pi	**lngth**
> (define radius 10)	**> lngth**
radius	**62.8318**

Дефиниране на процедура става отново чрез ***define***, както направихме по-горе при дефинирането на **lngth**.

Общ вид на обръщението към ***define*** в случая на **дефиниране на съставна процедура**:

(define (<име> <формални параметри>) <тяло>)

Тук:

- **<име>** е символ (идентификатор), който задава името на процедурата;
- **<формални параметри>** са имената, използвани в тялото на дефиницията за означаване на аргументите на процедурата;
- **<тяло>** е поредица от изрази (най-често – един израз).

Оценката на обръщението към ***define*** съвпада с **<име>**. В процеса на оценяване на това обръщение **<име>** се свързва с тялото на дефиницията в текущата среда. По такъв начин **<име>** вече е име на съставна процедура, която е комбинация на други процедури.

Пример: Процедура, която връща квадрата на дадено число: **(define (square x) (* x x))**

В процеса на оценяване на обръщението към ***define*** името **square** се свързва със зададената дефиниция в текущата среда. В случая се казва, че **square** е име на **съставна процедура** (дефинирана функция). Съставните процедури се използват по същия начин, както и вградените (примитивните) процедури, т.е. дефинираните и вградените процедури са неразличими за интерпретатора (смятат се за записани в средата).

Примери за дефиниране и използване на дефинирани процедури:

> (square (+ 2 5))	**> (define (sum-of-sq x y) (+ (square x) (square y)))**
49	**sum-of-sq**
> (square (square 3))	**> (sum-of-sq 3 4)**
81	**25**

Среди.

С използване на ***define*** е възможно последователно, стъпка по стъпка, да се въвеждат нови дефиниции (свързвания на имена със стойности). Това означава, че интерпретаторът трябва да използва специална работна памет, която съдържа наредените двойки от вида

(име, стойност). Тази работна памет се нарича **среда**. По-точно, възможно е в даден момент да съществуват няколко различни среди, една от които е т. нар. глобална среда.

Специални форми.

Вграденият оператор (примитивната процедура) ***define*** е изключение от общото правило за оценяване, което въведохме по-горе. Например, при оценяване на **(define x 3)** не се прилага ***define*** (по-точно, оценката на ***define***) върху оценките на двата операнда (първият операнд изобщо не се оценява), а променливата **x** се свързва със стойността **3** съгласно основното предназначение на ***define*** да свърже даден символ с определена стойност.

Такива вградени оператори, които са изключения от общото правило за оценяване на комбинации (т.е. оценяването на обръщения към които не се извършва по общото правило за оценяване на комбинации), се наричат **специални форми**.

По същество специалните форми определят синтаксиса на езика Lisp, а общото правило за оценяване и правилата за оценяване на специалните форми определят семантиката му.

Специална форма *cond*

Пример: Функция, която пресмята абсолютната стойност на дадено число:

```
(define (absol x)
        (cond ((> x 0) x)
              ((= x 0) 0)
              ((< x 0) (- x)) ))
```

Общ вид на обръщението към ***cond***:

```
(cond (<тест-1> <посл-от-изр-1>)
      (<тест-2> <посл-от-изр-2>)
      . . .
      (<тест-n> <посл-от-изр-n>))
```

Тук:

- **<тест-i>** е предикат, т.е. израз с оценка "истина" (**true**) или "лъжа" (**false**). В стандарта на Scheme за "лъжа" се смятат само константата **#f** и означението за празен списък (**nil** или **()**), които за интерпретатора са еквивалентни. Всички останали стойности се смятат за "истина";
- **<посл-от-изр-i>** е поредица от изрази, която може да бъде и празна.

Семантика: Последователно се оценяват **<тест-i>**, докато се стигне до (първия) такъв с оценка "истина". След това се оценяват последователно изразите от съответната **<посл-от-изр>** и оценката на последния се връща като оценка на обръщението към ***cond***. Ако няма **<тест>** с оценка ***"истина"***, то по стандарт оценката на ***cond*** е неопределена (в повечето реализации е ***()***, ***nil*** или ***#f*** и така отново съвпада с оценката на последния оценен израз). Ако първият срещнат **<тест>** с оценка "истина" няма съответна **<посл-от-изр>**, то оценката на ***cond*** съвпада с оценката на този **<тест>**. Възможно е вместо **<тест-n>** да се запише специалният символ ***else***, чието използване предизвиква безусловно оценяване на **<посл-от-изр-n>**, ако оценката на всички предходни **<тест-i>** е била "лъжа".

Забележка: Аргументите (**<тест-i> <посл-от-изр-i>**) се наричат клаузи.

Специална форма *if*

Друго решение на примерната задача:

```
(define (absol x)
        (cond ((< x 0) (- x))
              (else x) ))
```

Това решение е еквивалентно на:

```
(define (absol x)
        (if (< x 0) (- x) x))
```

Общ вид на обръщението към ***if***: **(if <тест> <then-израз> <else-израз>)**

Семантика: Оценява се **<тест>** и ако оценката му е "истина", се оценява **<then-израз>** и получената стойност става оценка на обръщението към ***if*** (при това **<else-израз>** не се оценява); в противен случай се оценява **<else-израз>** (без да се оценява **<then-израз>**) и получената стойност става оценка на обръщението към ***if***. Допуска се **<else-израз>** да липсва и този случай е еквивалентен на задаване на **<else-израз>** с оценка ***"лъжа"***.

Примерна програма на езика Scheme, с чиято помощ може да се провери какъв метод на оценяване (апликативен или нормален) е реализиран в използвания интерпретатор.

Нека разгледаме следната система от дефиниции:

```
(define (p) (p))
(define (test x y)
        (if (= x 0)
            0
            y))
```

Интересен в случая е резултатът от оценяването на **(test 0 (p))**:

- Ако методът е апликативен, то **(test 0 (p)) ——> (test 0 (p)) ——> ...** (оценяването на обръщението не завършва, тъй като не завършва оценяването на втория аргумент);
- Ако методът е нормален, то **(test 0 (p)) ——> (if (= 0 0) 0 (p)) ——> 0**.

Забележка: Вярно е следното твърдение: ако процесът на оценяване завърши, то двата метода дават един и същ резултат.

Функции – предикати в Scheme

Предикат е всяка функция, която връща логическа стойност ("истина" или "лъжа"). В този смисъл всички функции в Scheme могат да се разглеждат като предикати. Тук ще имаме предвид само функции, които извършват сравнения или логически операции.

Примитивни предикати за сравнения са: **>**, **=**, **<**, **>=**, **<=**, **<>**. Те са двуаргументни процедури, които оценяват аргументите си (оценките на аргументите трябва да са числа) и връщат стойност ***#t*** точно когато оценките на аргументите удовлетворяват съответното отношение.

Логически функции са: ***and*** (конюнкция), ***or*** (дизюнкция), ***not*** (логическо отрицание).

Оператор ***and***: произволен брой аргументи. Процесът на оценяване на обръщението към ***and*** е следният: Последователно се оценяват аргументите и ако всички аргументи имат стойност "истина", като резултат се връща оценката на последния аргумент. Ако в процеса

на оценяване на аргументите се срещне аргумент, чиято оценка е "лъжа", то останалите аргументи не се оценяват и оценката на обръщението към ***and*** е ***#f.***.

Оператор ***or***: има произволен брой аргументи. Процесът на оценяване на обръщението към ***or*** е следният: Последователно се оценяват аргументите и, ако всичките имат стойност "лъжа", оценката на обръщението към ***or*** е ***#f***. Ако в процеса на оценяване на аргументите се срещне аргумент, чиято оценка е "истина", то останалите аргументи не се оценяват и оценката на обръщението към ***or*** съвпада с оценката на последния оценен аргумент.

Следователно, процедурите ***and*** и ***or*** са специални форми.

Оператор за логическо отрицание (***not***): има един аргумент. Аргументът на ***not*** се оценява. Оценката на обръщението към ***not*** е ***#t***, ако оценката на аргумента е "лъжа"; ако оценката на аргумента е "истина", оценката на обръщението към ***not*** е ***#f***.

Модели на оценяване на комбинации.

При **оценяване на комбинация**, чийто оператор е съставна процедура, интерпретаторът следва същия процес, както при оценяване на комбинация, чийто оператор е примитивна процедура. Същността на този процес е следната: Интерпретаторът 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 примитивните процедури към аргументите е вграден за интерпретатора и е част от семантиката на примитивните процедури.

При прилагане на съставна процедура към аргументите се оценява тялото на процедурата, като предварително формалните параметри се заместват със съответните фактически (със съответните аргументи) и след това се следва механизмът на оценяване на комбинация.

Това правило стои в основата на т. нар. **модел на заместването при оценяване на обръщения към съставни (дефинирани) процедури**. Същността на този модел на оценяване на комбинациите и в частност на обръщенията към съставни процедури е:

- При оценяване на комбинация най-напред се оценяват подизразите на тази комбинация, след което оценката на първия подизраз се прилага върху оценките на останалите подизрази (т.е. прилага се общото правило за оценяване на комбинации);

- Прилагането на дадена съставна процедура към получените аргументи се извършва, като съгласно горното правило за оценяване на комбинации се оценят последователно изразите от тялото на процедурата, в които формалните параметри са заместени със съответните аргументи (фактически параметри). Оценката на последния израз от тялото става оценка на обръщението към съставната процедура.

Пример: При оценяване на **(f 5)** задачата в крайна сметка се свежда до оценяване на (**sum-of-sq (+ 5 1) (* 5 2))**, т.е. задачата се свежда до оценяване на комбинация с два операнда и оператор, наречен **sum-of-sq**. Следователно, необходимо е да се оцени операторът (за да се получи процедурата, която трябва да се приложи) и да се оценят операндите (за да се получат аргументите). При това, в рамките на вече въведения модел на оценяване чрез заместване са възможни два подхода (метода) за оценяване на тази комбинация: "**апликативен**" и "**нормален**".

Същност на апликативния подход (метод) на оценяване: Отначало се оценяват операндите (аргументите) и след това към получените оценки се прилага резултатът от оценяването на оператора.

В нашия пример:

(f 5) —дефиниция на f—> (sum-of-sq (+ a 1) (* a 2)) —заместване на параметри—>
(sum-of-sq (+ 5 1) (* 5 2)) —апликативен метод—> (sum-of-sq 6 10)
—деф. на sum-of-sq—> (+ (square 6) (square 10)) —оц. на комбинатори—>
(+ (* 6 6) (* 10 10)) —семантика на *—> (+ 36 100) —семантика на +—> 136

Същност на нормалния подход (метод) на оценяване: Замества се името на процедурата с тялото ѝ, докато се получат означения на примитивни процедури, и след това се прилагат техните правила за оценка.

В нашия пример:

(f 5) —дефиниция на f—> (sum-of-sq (+ a 1) (* a 2)) —зам. на параметри—>
(sum-of-sq (+ 5 1) (* 5 2)) —нормален метод—> (+ (square (+ 5 1))
(square (* 5 2)))
—нормален метод—> (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2))) —семантика на + и *—>
(+ (* 6 6) (* 10 10)) —семантика на *—> (+ 36 100) —семантика на +—> 136

Забележки:

1) Заместването на формалните параметри със съответните фактически се осъществява в рамките на локална за всяка процедура среда, като в процеса на оценяване всеки символ се заменя с оценката си в тази среда.

2) Оценяването чрез заместване не е валидно при използване на някои оператори (например, на оператора за присвояване).

3) При използване на нормалния подход на оценяване е необходимо да се вземат мерки за избягване на многократното оценяване на един и същ израз.

По правило интерпретаторите на Лисп (Scheme) използват апликативния метод.

Процедури от по-висок ред.

Един от съществените аспекти на всеки достатъчно мощен език за програмиране е възможността да се построяват абстракции, като за целта на избрани имена се присвояват използваните общи образци, след което може да се работи директно в термините на тези абстракции. Процедурите осигуряват тази възможност. Затова повечето езици за

програмиране съдържат механизми за дефиниране на процедури. Досега разглеждахме само процедури с числови аргументи. Друга възможност се състои в следното: Често един и същ програмен образец се използва с различни процедури. За означаване на такива образци като концепции е необходимо да се конструират процедури, които имат за аргументи процедури или връщат процедури като стойности. Такава процедура, която манипулира други процедури (на която формални параметри са процедури или която връща процедура като резултат), се нарича **процедура от по-висок ред**.

Процедурите като параметри.

Пример 1: Пресмятане на сумата на целите числа от **a** до **b**.

Ще използваме следната идея: $\sum_{k=a}^{b} k = a + (a+1) + (a+2) + \ldots + b = a + \sum_{k=a+1}^{b} k$

```
(define (sum-int a b)
        (if (> a b)
                0
                (+ a (sum-int (+ a 1) b))))
```

Пример 2: Пресмятане на сумата от кубовете на целите числа от **a** до **b**.

```
(define (cube x) (* x x x))
(define (sum-cub a b)
        (if (> a b)
                0
                (+ (cube a) (sum-cub (+ a 1) b))))
```

Пример 3: Пресмятане на част от сума, която според известната формула на Лайбниц клони много бавно към π/8: $\frac{\pi}{8} = \frac{1}{1.3} + \frac{1}{5.7} + \frac{1}{9.11} + \ldots + \frac{1}{a.(a+2)} + \frac{1}{(a+4).(a+6)} + \ldots$, където *a* е цяло положително число от вида *4k+1*.

```
(define (sum-pi a b)
        (if (> a b)
                0
                (+ (/ 1 (* a (+ a 2))) (sum-pi (+ a 4) b))))
```

Трите процедури от горните примери използват един и същ образец. Те си приличат много (имат много общи части); различават се по функцията, която се използва за пресмятане на общия член на сумата, и по функцията, с чиято помощ се пресмята следващата стойност на първия аргумент (променливата a). Всяка от тези процедури може да бъде генерирана чрез попълване на специално означените елементи в следния образец:

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

Тук:

- **<term>** е име на процедура (правилото за пресмятане на поредния член на сумата);
- **<next>** също е име на процедура (правилото за получаване на следващата стойност на аргумента).

Този образец отразява идеята за сумиране (намиране на частична сума на ред) по формулата: $\sum_{a}^{b} f(n) = f(a) + ... + f(b)$. Тук *f* съответства на **<term>**, а Σ - на **<name>**.

Ако искаме да получим още по-голяма степен на общност – такава, че да построим процедура, която да отразява идеята за сумиране въобще (сама по себе си), а не да построяваме различни процедури за пресмятане на различни конкретни суми, можем да модифицираме горния образец в дефиниция на процедура от по-висок ред, която има като аргументи въведените по-горе **<term>** (формулата за пресмятане на общия член) и **<next>** (формулата за пресмятане на новата стойност на аргумента). Така се получава следната по-обща дефиниция:

```
(define (sum term a next b)
        (if (> a b)
                0
                (+ (term a) (sum term (next a) next b))))
```

Като прости примери за използване на тази процедура ще покажем как се получават дефинираните по-горе процедури **sum-cub** и **sum-pi**:

```
(define (sum-cub a b) (sum cube a inc b))
(define (inc x) (+ x 1))
(define (sum-pi a b)
        (define (pi-term x) (/ 1 (* x (+ x 2))))
        (define (next x) (+ x 4))
        (sum pi-term a next b))
```

Обща бележка: Дефинираната по-горе процедура **sum** генерира линеен рекурсивен процес. Ако искаме да дефинираме аналогична по резултат процедура, която генерира линеен итеративен процес, можем да използваме следната схема:

```
(define (sum term a next b)
        (define (iter a result)
                (if <?>
                        <?>
                        (iter <?> <?>)))
        (iter <?> <?>))
```

Правила за трансформиране на променливите на състоянието в помощната процедура **iter**: **result := result + term(a), a := next(a),** и търсената дефиниция:

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

Не е задължително да се включва

Ламбда изрази.

Идея: Често е досадно и ненужно да се задават (определят) имена на някои елементарни процедури, например **(define (next x) (+ x 4))**.

В езика Scheme съществуват средства, които позволяват да се дефинират процедури, без те да бъдат именувани, т.е. да се дефинират **анонимни процедури**.

Процедурата от горния пример е функция с аргумент x, която прибавя 4 към аргумента си:

(lambda (x) (+ x 4))

Общ вид на обръщението към специалната форма ***lambda***:

(lambda (<формални параметри>) <тяло>)

Семантика: В резултат на оценяването на обръщението към ***lambda*** се получава процедура, която обаче не се свързва с никакво име в средата. Тази процедура става оценка на обръщението към ***lambda***.

Връзка между дефинирането на процедура с помощта на ***define*** и ламбда дефиниция на процедура: Конструкциите **(define (<име> <формални параметри>) <тяло>)** и **(define <име> (lambda (<формални параметри>) <тяло>))** са еквивалентни.

Общ вид на обръщението към ***define***: **(define <име> <израз>)**

При това, ако **<израз>** е ламбда израз (ламбда дефиниция), ***define*** определя процедура с име **<име>**; иначе ***define*** определя променлива с име **<име>**.

Оценяване на комбинации, чиито оператори са ламбда изрази. В общия случай тези комбинации са от вида: **((lambda (x1 x2 ... xn) <тяло>) a1 a2 ... an)**

В процеса на оценяване на такава комбинация всички срещания (включвания) на **xi** в **<тяло>** се заместват с **[ai]**, след което се оценява така полученият израз.

Пример:

```
> ((lambda (x y z) (+ x y (* z z))) 1 2 3)
12
```

Процедурите като оценки на обръщения към процедури.

Изразителната сила на един език за програмиране се повишава много, ако освен възможността за задаване на процедури като параметри се предвиди и възможността по аналогичен лек и естествен начин да се задават процедури като оценки на обръщенията към други процедури.

Пример: Нека разгледаме изречението: "Производната на функцията x^3 е функцията $3x^2$.". То означава, че производната на функцията, чиято стойност в x е x^3 , е друга функция - тази, чиято стойност в x е $3x^2$. Следователно, понятието "производна" може да бъде разглеждано като определен оператор (в математически смисъл), който за дадена функция f връща друга функция Df. В този смисъл, за да опишем понятието "производна", можем да кажем, че ако f е някаква функция, то производната Df на f е функцията, чиято стойност за всяко число x се получава като

$$Df(x) = \lim_{dx \to 0} \frac{f(x+dx) - f(x)}{dx}.$$

Ако искаме (с известно приближение, като игнорираме означението за граница) да запишем горната формула във вид на дефиниция на процедура на езика Scheme, това може да стане по следния начин:

- Дясната страна на формулата може да се запише като **(lambda (x) (/ (- (f (+ x dx)) (f x)) dx))**, където **dx** е име на променлива, която означава (чиято стойност по идея е) някакво малко положително число;

- Цялата формула може да се запише като

(define (deriv f dx) (lambda (x) (/ (- (f (+ x dx)) (f x)) dx)))

Коментар: Така **deriv** е процедура, която има като аргумент процедура (има и още един аргумент – числото **dx**) и която връща като резултат нова процедура – тази, която е дефинирана чрез използвания ламбда израз. Последната процедура, приложена към някакво число **x**, връща стойността на производната **f'(x)**.

Процедурата **deriv** може да бъде дефинирана също и с използване на локална именувана процедура по следния начин:

```
(define (deriv f dx)
        (define (right_hand_side x)
                (/ (- (f (+ x dx)) (f x)) dx))
        right_hand_side)
```

Пример за използване на процедурата **deriv**:

```
> ((deriv cube 0.001) 5)
75.0150010000255
```

```
> ((deriv cube 0.00001) 5)
75.0001499966402
```

Тук операторът на използваната комбинация също е комбинация, тъй като процедурата, която се прилага към аргумента 5, е стойността на процедурата **deriv** при аргумент **cube**.