

Основни принципи на функционалното програмиране. Езици за функционално програмиране

Два стила на програмиране

Според един от разпространените класификационни признаци съществуват два основни стила на програмиране:

- процедурен (императивен);
- декларативен (дескриптивен).

В **процедурните езици** програмата се реализира по известната схема

програма = алгоритъм + структури от данни

В основата на програмата при императивното програмиране стои **алгоритъмът**. Той определя последователните стъпки на обработката на данните с цел получаване на търсения резултат. Решаването на една задача се свежда до точно описание на това **как** се получава съответният резултат.

При **декларативните езици** за програмиране в програмата явно се посочва *какви свойства има желаният резултат* (или **какво** е известно за свойствата на предметната област и в частност на резултата), без да се посочва как точно се получава този резултат. С други думи, при декларативните езици се посочва **какво** се пресмята, без да се определя **как** да се пресмята то.

В декларативните езици програмите се изграждат по схемата

програма = списък от дефиниции на функции	}	при функционалното програмиране
или		
списък от равенства		
или	}	при логическото програмиране
факти + правила		

Обща характеристика на функционалния стил на програмиране

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).

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 езиките за функционално програмиране се наричат още **апликативни езици**.

Пример 1. Програма за намиране на сумата на естествените числа от 1 до n.

- Решение в процедурен (императивен) стил:

```
count := 0
total := 0
repeat
    count := count + 1
    total := total + count
until
    count = n
```

- Решение във функционален стил (на езика Haskell):

```
sum [ ] = 0
sum (x : xs) = x + sum xs
sum [1..n]
```

Пример 2. Програма за „бързо“ сортиране (псевдокод със синтаксис и семантика, близки до тези на езика Haskell).

```
qsort [ ] = [ ]
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [a | a ← xs, a ≤ x ]
    larger  = [b | b ← xs, b > x ]
```

Силни страни 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 – потенциален проблем, който има добри теоретични и практически решения;
- неестествено и в някои случаи неефективно е използването му при решаване на задачи от процедурен (алгоритмичен) характер.

Теоретичен модел на функционалното програмиране е **λ-смятането**, въведено от A. Church през 1930-те години.

Факти от историята

На базата на λ -смятането през 1958-1961 г. J. McCarthy разработва езика за функционално програмиране Lisp. В Lisp програмата е списък от дефиниции на функции. Дефиницията на функция се осъществява с помощта на едно равенство, но чрез използване на множество от примитивни функции.

През 1977 г. M. O'Donnell обосновава математически изчисленията в системи, описани с равенства. O'Donnell ги нарича системи за програмиране с равенства, а също и системи за заместване на поддървета. При тях програмата е съвкупност от равенства от вида $A = B$. Равенствата са ориентирани в смисъл, че B може да замести A , но A не може да замести B . Сред най-популярните езици за програмиране с равенства са HOPE, ML и др.

През 1978 г. J. Backus поставя основите на т. нар. FP-стил на програмиране. FP-програмите са или примитивни форми, или дефиниции, или функционални форми. Backus доказва, че FP-системите имат следните важни свойства:

а) проста формална семантика;

б) ясна йерархична структура на FP-програмите, при която програми от високо ниво могат да се комбинират и да образуват програми от още по-високо ниво;

в) основните FP-форми за комбиниране са операции в съответна алгебра на програми, която може да се използва за трансформиране на програми, за доказване на свойства на програми и др.

Типове езици за функционално програмиране

Езиците за функционално програмиране могат да се разделят на:

- **строги („чисти“) функционални езици** – при тях единствените управляващи структури са функциите. Странични ефекти не се допускат. Такива езици са: Pure Lisp, HOPE, Miranda, Haskell, Erlang и др.;
- **нестроги функционални езици** – те съдържат някои императивни конструкции и допускат използването на нестроги функции. Такива езици са: Common Lisp, Scheme, Racket, ML и др.

Цел на курса по ФП: да се усвои методологията на програмиране в красив функционален стил на базата на два конкретни езика за функционално програмиране, които имат обща теоретична основа (лямбда смятането), но различни характеристики – нестрого/строغو функционален стил; апликативен модел на оценяване/модел на т. нар. мързеливо оценяване.

Основни понятия в програмирането на езика Scheme/Racket

Основни сведения за езика Lisp. Диалекти на Lisp. Обща характеристика на Scheme и Racket

Езикът Lisp (LISP = LISt Processing) е създаден от J. McCarthy в края на 1950-те години (през 1960 г. е публикувано съобщението за езика). Той е интерактивен език за обработка на символни (нечислови) данни, записани в списъчен вид. Lisp може да бъде характеризиран също като функционален език за символни (нечислени) пресмятания (преобразования).

Lisp все още е един от широко използваните езици за програмиране в областта на изкуствения интелект. Причина за това са изброените по-горе особености на езика, както и фактът, че програмите и данните в Lisp имат еднакъв вид, т.е. една програма на Lisp по естествен начин може да бъде третирана като данни за друга програма на Lisp и т.н.

Диалекти на езика Lisp: Lisp 1.5, MacLisp, Franz Lisp, Standard Lisp, Scheme, Common Lisp и др. Common Lisp е утвърден като търговски стандарт за езика Lisp. Scheme е създаден с учебна цел. Той е с по-ограничени възможности и запазва в по-голяма степен и в по-чист вид идеите на функционалното програмиране. **Racket** (<https://racket-lang.org/>) се сочи за наследник на Scheme ("We use "Racket" to refer to a specific dialect of the Lisp language, and one that is based on the Scheme branch of the Lisp family").

Среда за програмиране на Racket, която ще бъде използвана на упражненията по ФП: DrRacket (<http://docs.racket-lang.org/drracket/>).

Препоръчвана версия на DrRacket: 8.2 от август 2021 г. или 7.8 от август 2020 г.

Изрази

Синтактично изразът е или **атом** (атомарна константа или променлива), или **обръщение към функция**, което има следния вид:

(<означение на функция> <арг₁> <арг₂> ... <арг_n>)

└──┘

оператор

└──┘

операнди

Забележка. В Scheme/Racket заедно с термина „функция“ като негов синоним се използва и терминът „процедура“. Терминът „процедура“ в смисъл на описание на даден изчислителен процес понякога дори се предпочита, тъй като в Scheme/Racket, както и в другите съвременни диалекти на езика Lisp, няма изисквания за строга функционалност. В този смисъл терминът „функция“ понякога не е съвсем точен. Затова в по-нататъшното изложение на езика Racket термините „функция“ и „процедура“ обикновено ще бъдат използвани като синоними.

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) .
> (+ 137 349) ; Въведена е комбинация - примитивна
486 ; процедура (+, -, *, ... ) се прилага върху
> (- 1000 334) ; числови аргументи.
666
> (* 5 99)
495
(+ 3 5 7)
15
> (/ 10 6)
 $\frac{5}{3}$ 
> (+ 2.7 10 2)
14.7
> (sqrt -16)
0+4i
```

Дефиниция. Комбинация (обръщение към функция/процедура) се нарича израз, конструиран като списък от изрази, заградени в скоби. Най-левият елемент на списъка се нарича *оператор*, а останалите - *операнди*. Стойността (**оценката**) на комбинацията се получава чрез прилагане (апликация) на процедурата, зададена чрез оператора, към аргументите, които са стойности на операндите.

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

- лесно се използват процедури, които имат 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).

Примери:

(+ (* 3 5) (- 10 6))

(* (+ 2 (* 2 4)) (+ 3 5 7))

Резюме на работата на интерпретатора на Racket.

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.

Дефиниране на променливи и процедури в езика Racket

Именуване и среда

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

Пример

```
(define size 2)  
> 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). Както ще покажем по-нататък, **define** се използва за дефиниране не само на променливи.

По-нататъшни примери

```
> (* 5 size)
10
(define pi 3.14159)
(define radius 10)
> (* pi (* radius radius))
314.159
(define lngth (* 2 pi radius))
> lngth
62.8318
```

Изводи

1. **define** е първото средство за абстракция, което разглеждаме. **define** позволява да се именуват резултати от съставни операции (комбинации), както направихме по-горе при дефиницията на променливата **length**.

2. 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 (управлява) 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**.

Оценяване на комбинациите

Общо правило за оценяване на комбинациите:

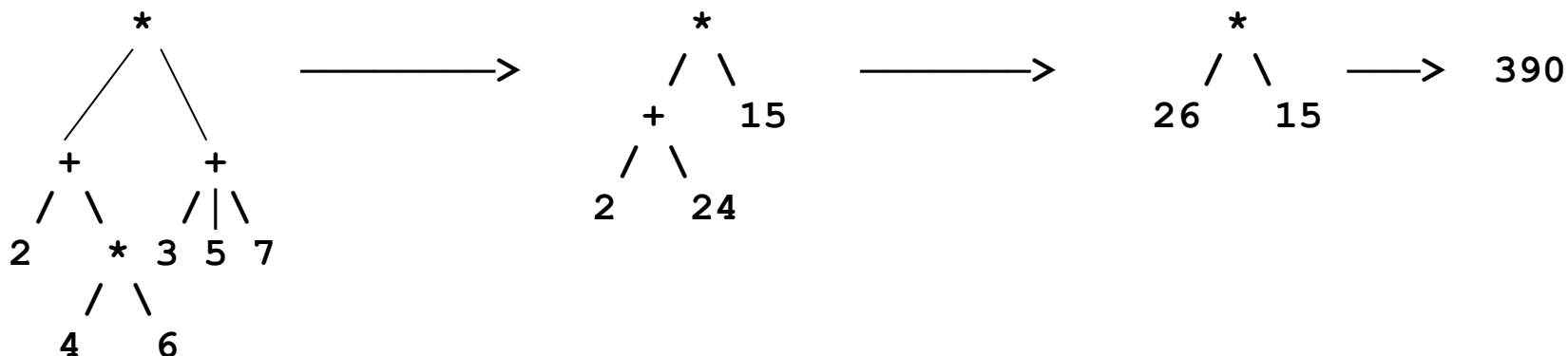
- оценяват се подизразите на комбинацията;
- прилага се процедурата, която е оценка на най-левия подизраз (т.е. операторът), към операндите (аргументите), които са оценки на останалите подизрази.

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 (т.е. на примитивните процедури) също могат да се смятат за част от средата.

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**.

Следователно, **define** е пример за специална форма.

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

Дефиниране на съставни процедури

Пример. Да се дефинира процедура, която връща като резултат квадрата на дадено число.

Решение

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

Анализ. 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)**.

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).

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

Дефинираните (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 4)
```

```
16
```

```
> (square (+ 2 5))
```

```
49
```

```
> (square (square 3))
```

```
81
```

```
(define (sum-of-sq x y) (+ (square x) (square y)))  
> (sum-of-sq 3 4)  
25  
(define (f a) (sum-of-sq (+ a 1) (* a 2)))  
> (f 5)  
136
```

Резюме на някои термини в езика Racket. Съществуват два типа изрази в Racket: примитивни изрази и съставни изрази. Примитивни изрази са атомите в езика. Съставните изрази са комбинации, т.е. обръщения към функции. Термините функция и процедура често се разглеждат като синоними. Функциите в езика са вградени (наричат се още *примитивни процедури*) и дефинирани (наричат се още *съставни процедури*, тъй като представляват комбинации на други процедури).

Оценяване на обръщение към процедура чрез заместване

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 процедури (дефинирани функции)**. Същността на този модел на оценяване на комбинациите и в частност на обръщенията към съставни процедури е следната:

- 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 $(\text{sum-of-sq}\ (+\ 5\ 1)\ (*\ 5\ 2))$, 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:

	деф. на f		зам. на парам.
(f 5)	—————>	(sum-of-sq (+ a 1) (* a 2))	—————>
		апликат. метод	
(sum-of-sq (+ 5 1) (* 5 2))	—————>	(sum-of-sq 6 10)	
деф. на sum-of-sq		оц. на комб.	
—————>	(+ (square 6) (square 10))	—————>	
	сем. на *	сем. на +	
(+ (* 6 6) (* 10 10))	—————>	(+ 36 100)	—————> 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:

	деф. на f		зам. на парам.
(f 5)	—————>	(sum-of-sq (+ a 1) (* a 2))	—————>
		норм. метод	
(sum-of-sq (+ 5 1) (* 5 2))	—————>	(+ (square (+ 5 1)) (square (* 5 2)))	
	норм. метод		сем. на + и *
—————>	(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))	—————>	
	сем. на *	сем. на +	
(+ (* 6 6) (* 10 10))	—————>	(+ 36 100)	—————> 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.

Кой метод (подход) на оценяване се използва от интерпретаторите на Racket?

Условни изрази и предикати в езика Racket

Специална форма `cond`

Пример. Да се дефинира функция, която пресмята абсолютната стойност на дадено число по правилото:

$$\text{absol}(x) = \begin{cases} x, & x > 0 \\ 0, & x = 0 \\ -x, & x < 0 \end{cases}$$

Първо решение с помощта на условния оператор (специалната форма) ***cond***:

```
(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 Racket логическата стойност „истина“ се означава с константата **#t** (или **#T**), а логическата стойност „лъжа“ – с константата **#f** (или **#F**). При оценяване на обръщения към специалните форми **cond**, **if**, **and**, **or** и др. обаче всички стойности, различни от **#f**, се интерпретират като представители на логическата стойност „истина“;***

- **<посл-от-изр-*i е поредица от изрази, която може да бъде и празна (т.е. да не съдържа нито един елемент).***

Семантика (механизъм на оценяване). Последователно се оценяват **<тест-і>**, 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 (**#<void>**). 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-п>** да се запише специалният символ **else**, чието използване предизвиква безусловно оценяване на **<посл-от-изр-п>**, ако оценката на всички предходни **<тест-і>** е била „лъжа“.

Забележка. Аргументите [**<тест-і>** **<посл-от-изр-і>**] 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 (механизъм на оценяване). 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*. В Racket не се допуска <else-израз> да липсва.

Примерна програма на езика Racket, 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))` \longrightarrow `(test 0 (p))` \longrightarrow ... (оценяването на обръщението не завършва, тъй като не завършва оценяването на втория аргумент);

- ако методът е нормален, то

`(test 0 (p))` \longrightarrow `(if (= 0 0) 0 (p))` \longrightarrow 0.

Забележки:

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

2. Моделът на оценяване чрез заместване е частен случай на по-общия (валидния в действителност) модел на оценяване на изрази в Scheme/Racket, известен под името ***модел на средите***, който ще бъде разгледан в детайли по-нататък в курса.

Функции – предикати в Racket

В тази група се включват функциите, които връщат логическа (булева) стойност – „истина“ (**#t**) или „лъжа“ (**#f**). В частност такива са функциите, които извършват сравнения или логически операции.

Примитивни предикати за аритметични сравнения

Такива са предикатите: **>**, **=**, **<**, **>=**, **<=**. 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

Такива са процедурите: **and** (конюнкция), **or** (дизюнкция), **not** (логическо отрицание).

Оператор за конюнкция (**and**): има 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 за дизюнкция (**or**): има произволен брой аргументи. Процесът на оценяване на обръщението към **or** е следният. Последователно се оценяват аргументите и ако всички аргументи 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***.

Примери

```
> (not #t)
```

```
#f
```

```
> (not #f)
```

```
#t
```

```
> (not 0)
```

```
#f
```

Примерна задача. Да се дефинира функция на езика Racket, която пресмята

$$\sum_{i=1}^n \frac{i}{i^2 + 1}$$

където n е дадено естествено число.

Решение:

```
(define (suma n)
  (if (= n 0)
      0
      (+ (/ n (+ (* n n) 1)) (suma (- n 1))))))
```