Процедурите като абстракции. Вложени дефиниции и блокова структура на програмите

Същност на абстрахирането чрез процедури

Пример. Да се дефинира функция, която намира пълната повърхнина на правилна пирамида, чиято основа е правилен **n**-ъгълник с дължина на страната **a**, а апотемата ѝ е равна на **l**.

Първи вариант на решение

```
(define (full-area n a l)
   (+ (s n a) (area n a l)))
```

```
(define (area n a l)
;;; Намира околната повърхнина
   (/ (* (perim n a) l) 2))
(define (perim n a)
;;; Намира периметъра на основата
   (* n a))
(define (s n a)
;;; Намира лицето на основата. S=p.r/2,
;;; r=a/(2.tg(pi/n))
   (/ (* (perim n a) a) (* 4 (tan (/ pi n))) ))
```

Забележка. **pi** е системна (вградена) константа в DrRacket със стойност 3,141592653589793.

Анализ. По този начин се постига декомпозиция на дадената задача на подзадачи. Процедурата **full-area** (а и не само тя) по същество представлява абстракция – означение на съставна операция, която включва резултатите от няколко по-прости. В такива случаи се говори за използване на подход, известен като ***абстракция чрез/на процедури*** (или още, модулен подход). Най-съществения елемент на абстрахирането е разделянето на задачата на подзадачи, които могат да бъдат решавани поотделно.

Основни преимущества на този подход:

- голям програмен проект може да се раздели между няколко изпълнители, които да работят относително независимо един от друг;
- по-лесно се извършват проверката и поправката на програмите.

Вложени дефиниции и блокова структура на програмите на Racket. Локални имена

Втори вариант на решение на примерната задача (с използване на вложени дефиниции)

```
(define (full-area n a l)
   (define (area) (/ (* (perim) l) 2))
   (define (perim) (* n a))
   (define (s) (/ (* (perim) a) (* 4 (tan (/ pi n))) ))
   (+ (s) (area)) )
```

Забележки

1) Вложените дефиниции трябва да фигурират една след друга непосредствено след името и формалните параметри на основната процедура. Вложението на дефиниции определя блоковата структура на програмите на езика Racket.

2) Редът на записване на дефинициите на вложени процедури на едно и също ниво е без значение.

3) Областта на действие на една променлива съвпада с блока, в който променливата е дефинирана. Търсенето на стойността на дадена променлива се извършва в следния ред: ако в процеса на оценяване интерпретаторът срещне име, което не е описано в текущия блок, той търси оценката му в блока, обграждащ (съдържащ) лексически текущия, и ако отново не я намери, продължава аналогично до достигане на глобалната среда.

Процедури и процесите, които описват те

На всяка система от (взаимно-рекурсивни) функционални уравнения може да се гледа както *декларативно* (като на описание на свойствата на обекта, който трябва да бъде получен), така и *процедурно* (като на описание на процеса, който ще доведе до получаване на търсения резултат). Същото се отнася и за всяка функция във функционалното програмиране, в частност при програмирането на езика Racket. С други думи, функционалните програми и в частност програмите на езика Racket имат *двойнствена семантика* (говори се за наличие на *декларативна и процедурна семантика на програмите на езика Racket*). Затова и термините функция и процедура се използват като синоними в литературата за езика Racket.

Дефиниция на понятието процес. Процесът е абстрактно понятие, което се използва за описание на *развитието във времето на пресмятанията върху определени входни данни до получаване на крайния резултат*.

Например, изчислителният процес, реализиран от интерпретатора, дава възможност за реализиране на други процеси, които програмистът описва във вид на дефиниции на процедури на езика Racket. В програмирането развитието на изчислителния процес се планира и контролира чрез програми, съставени от процедури.

Следователно, в много случаи е полезно да се отдели понятието процес от понятието процедура.

Линейна рекурсия и итерация

Примерна задача. Да се състави програма за пресмятане на **n!**.

Решение

Първи начин

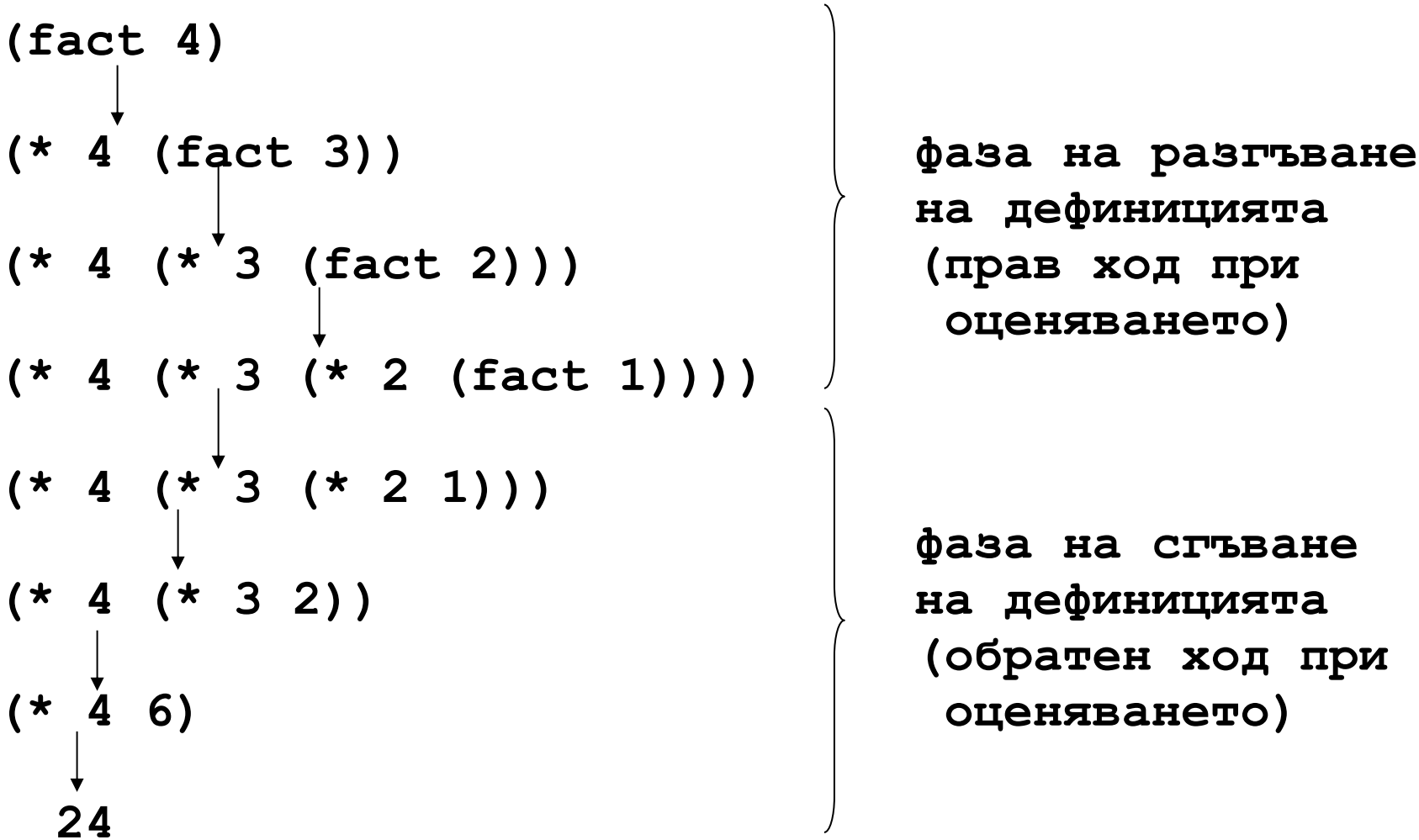
Използва се дефиницията на **n!**, според която:

n! = n.(n-1)!, n≥2,

1! = 1.

```
(define (fact n)
   (if (= n 1)
       1
       (* n (fact (- n 1)))))
```

Проследяване на развитието на процеса
(хода на изпълнението):



Втори начин

Използва се дефиницията на **n!**, според която:

n! = 1.2.3. ... .n.

Според тази дефиниция **n!** може да се получи, като най-напред се умножи **1.2**, след това полученият резултат се умножи по **3**, след това - по **4** и т.н., докато се достигне **n** (умножава се накрая по **n** и когато поредният множител стане по-голям от **n**, пресмятането се прекратява). Необходимо е да се съхраняват частичният резултат (**product**) и поредният множител (**counter**). Правилата за промяна на стойностите на **product** и **counter** са:

product := product*counter, counter := counter+1.

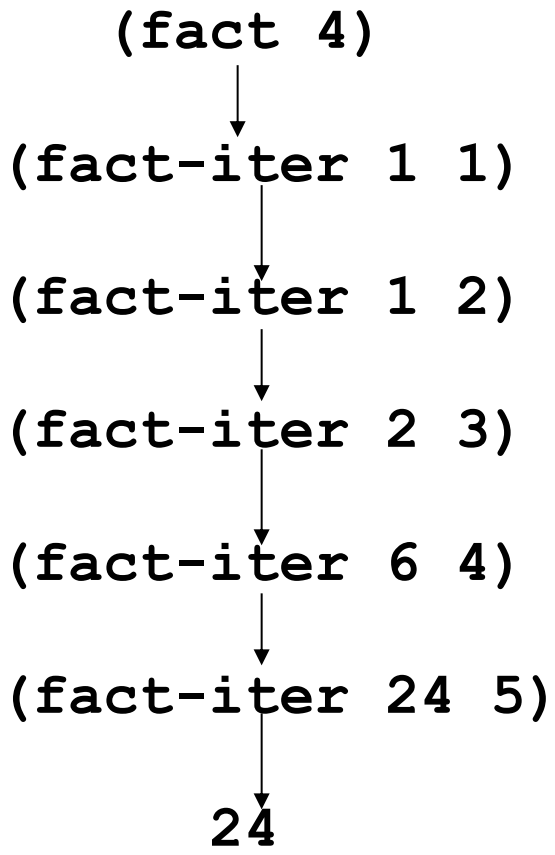
Първоначално **product = counter = 1**, а когато **counter** получи стойност, по-голяма от **n** (т.е. когато counter стане **n+1**), **product** ще има стойност **n!**.

```
(define (fact n)
   (define (fact-iter product counter maxcount)
      (if (> counter maxcount)
          product
          (fact-iter (* product counter)
                     (+ counter 1)
                     maxcount)))
   (fact-iter 1 1 n))
```

Забележка. По-добре е процедурата **fact-iter** да се дефинира не като глобална, а като локална за **fact**, както е показано по-горе. Всъщност третият аргумент на процедурата **fact-iter** е излишен, поради което горната дефиниция може да придобие вида:

```
(define (fact n)
   (define (fact-iter product counter)
      (if (> counter n)
          product
          (fact-iter (* product counter) (+ counter 1))))
   (fact-iter 1 1))
```

Проследяване на развитието на процеса
(хода на изпълнението):



Сравнение между двете решения

Общи черти:

- реализират пресмятането на една и съща математическа функция;
- броят на стъпките е пропорционален на **n**, т.е. и двата процеса са линейни;
- извършва се една и съща поредица от умножения (1.2, 2.3, ...) и съответно се получават еднакви междинни резултати.

Различия:

- при първия процес има *фаза на разгъване* (увеличаване на броя на участващите операции – в случая умножения) и след това – *фаза на сгъване* (намаляване на броя на операциите). При втория процес броят на участващите (означените) операции е постоянен;
- при изпълнението на първия процес (при използването на първата дефиниция) интерпретаторът трябва да запазва формираната верига от умножения, за да може по-късно да ги изпълни. При изпълнението на втория процес (при използването на втората дефиниция) текущите стойности на променливите **product** и **counter** дават пълна информация за текущото състояние.

Дефиниция. Процесите от първия тип се наричат ***рекурсивни***. При тях се поражда верига от обръщения към дефинираната функция с все по-прости в определен конкретен смисъл аргументи, докато се стигне до обръщение с т. нар. базов (прост, граничен) вариант на аргументите, след което започва последователно пресмятане на генерираните вече обръщения. Процесите от втория тип се наричат ***итеративни***. При тях във всеки момент състоянието на изчисленията се описва (като при това може при необходимост да бъде прекъснато и после – възстановено) от няколко променливи (state variables, променливи на състоянието) и правило, с чиято помощ се извършва преходът от дадено състояние към следващото.

Ако при даден рекурсивен процес дължината на генерираната верига (и, следователно, обемът на необходимата памет за нейното съхраняване) расте линейно с нарастването на аргумента **n** (т.е. е линейна функция на **n**), процесът се нарича ***линейно рекурсивен*** (говори се още за *линеен рекурсивен процес*).

Ако при даден итеративен процес броят на стъпките (и, следователно, времето за съответните пресмятания) расте линейно с нарастването на аргумента **n** (т.е. е линейна функция на **n**), процесът се нарича ***линейно итеративен*** (говори се още за *линеен итеративен процес*).

Забележки

1) За описание както на рекурсивния, така и на итеративния процес използвахме процедури, които синтактично са рекурсивни (в тялото на такава процедура има обръщение към нея самата, т.е. в тялото на процедурата се посочва нейното име).

2) От отбелязаното току-що следва, че е възможно итерацията да се емулира, т.е. по същество в езика няма нужда от итеративни конструкции. Такива обаче са предвидени за удобство.

3) В програмирането по отношение на синтаксиса на процедурите се говори за т. нар. ***линейна рекурсия*** и ***опашкова рекурсия***. При линейната рекурсия дефиницията включва (поражда) само едно рекурсивно обръщение към същата процедура с опростени аргументи. Опашковата рекурсия (tail recursion) е линейна рекурсия, при която общата задача се трансформира до нова, по-проста, като при това решението на общата задача съвпада с решението на по-простата, а не се получава от него с помощта на допълнителни операции.

Така общата задача директно се редуцира до по-проста и няма нужда от обратен ход за получаването на решението на общата задача. Затова по принцип използването на опашкова рекурсия създава условия за по-голяма ефективност на съответните процедури, отколкото използването на линейна рекурсия от по-общ вид. Такава ефективност е налице при опашково рекурсивните процедури на езика Racket, тъй като интерпретаторите на Racket се реализират така, че при изпълнението на опашково рекурсивните процедури не използват допълнително пространство от системния стек. Затова се казва, че интерпретаторите на езика Racket имат т. нар. *опашково рекурсивна семантика*.

В разгледаната примерна задача опашкова рекурсия се използва във втория вариант на дефиницията на **fact**, който описва итеративния процес (по-точно, в дефиницията на процедурата **fact-iter**). С помощта на първия вариант на дефиницията в рамките на използваната преди години във ФМИ реализация на PC Scheme успешно се пресмята 3100!, но при пресмятането на 3150! се получава препълване на системния стек (Stack Overflow); с помощта на втория вариант на дефиницията успешно се пресмята 5000!, но при пресмятането на 5100! се получава препълване на списъчната памет (Heap Space Exhausted). В рамките на използваната реализация на DrRacket не са установени видими различия при изпълнението на двата варианта.

Нелинейни рекурсивни и итеративни процеси

Примерна задача. Да се състави програма за намиране на най-големия общ делител (gcd) на две цели числа.

Дефиниция. Най-големият общ делител (gcd) на две цели числа u и v е най-голямото цяло число d, на което u и v се делят без остатък.

В езика Racket има вградена функция ***gcd*** за намиране на най-големия общ делител на две цели числа. Ще дефинираме нов вариант на такава функция, който се базира на следните свойства на най-големия общ делител:

gcd(u,v) = gcd(v,u),
gcd(0,v) = |v|,
gcd(u,v) = gcd(|u|,|v|),
ако u≥v>0, то gcd(u,v) = gcd(v, u mod v).

Следователно, общият случай на намиране на **gcd** на две произволни цели числа **u** и **v** лесно се свежда до базовия, в който **u≥v≥0**. В този случай се използва известният алгоритъм на Евклид, при който **gcd(u,v) —> gcd(v,u mod v) —> ...** , докато второто число стане равно на **0** и тогава резултатът е равен на текущата стойност на първото число.

Решение

```
(define (user-gcd u v)
   (define (pos-gcd u v)
      (if (> u v)
          (bas-gcd u v)
          (bas-gcd v u)))
   (define (bas-gcd u v)
      (if (= v 0)
          u
          (bas-gcd v (remainder u v))))
   (cond [(= u 0) (abs v)]
         [(= v 0) (abs u)]
         [else (pos-gcd (abs u) (abs v))]))
```

Забележка. remainder е вградена в Racket функция за намиране на остатъка при целочисленото деление на две положителни цели числа.

Обща оценка. Процедурата **user-gcd** генерира итеративен процес, при който броят на стъпките на изпълнението расте, както расте log n, където n = min(|u|,|v|). По-точно, броят на стъпките в случая не надвишава $\log_{\Phi} n$, където n = min(|u|,|v|) и $\Phi = (1+\sqrt{5})/2 \approx 1.6180$ (Φ е положителният корен на уравнението $\Phi^2 = \Phi+1$). В такива случаи се говори за ***логаритмични итеративни процеси***.

Дървовидна рекурсия

Примерна задача. Да се състави програма на намиране на n-тото число на Фибоначи.

Както е известно, редицата от числата на Фибоначи има следния вид: 0, 1, 1, 2, 3, 5, 8, 13, 21, ... n-тото число на Фибоначи се пресмята по формулата:

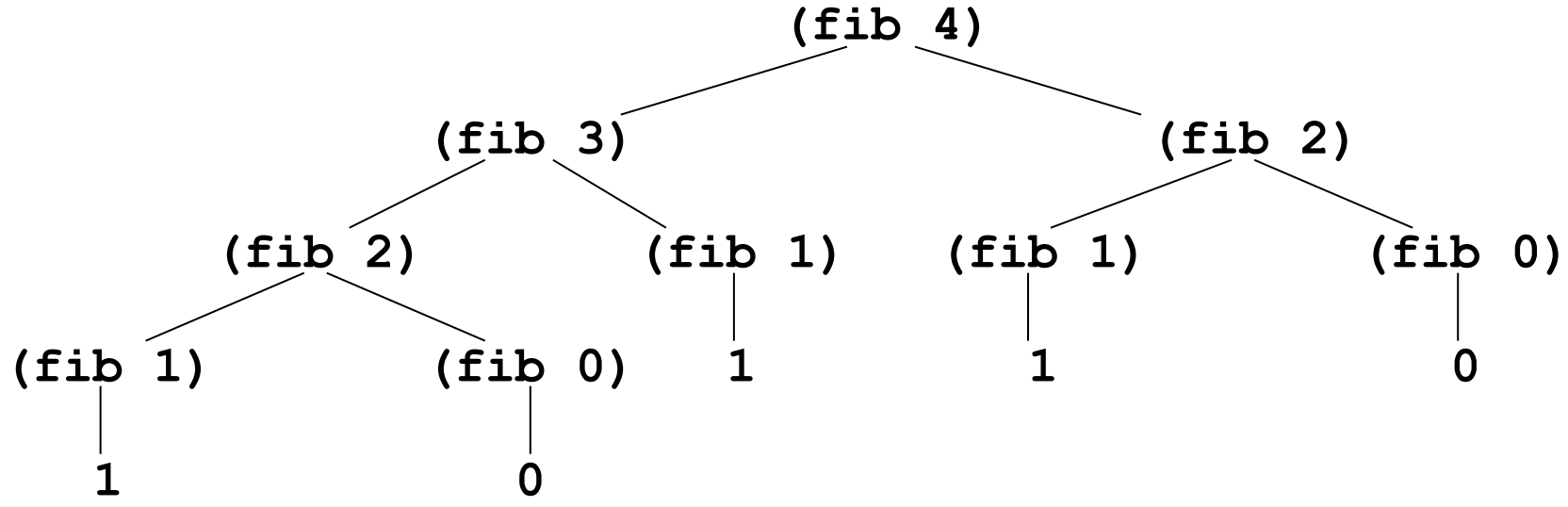
$$\mathbf{Fib(n)} = \begin{cases} \mathbf{0} & \mathbf{, n=0} \\ \mathbf{1} & \mathbf{, n=1} \\ \mathbf{Fib(n-1)+Fib(n-2)} & \mathbf{, n>1} \end{cases}$$

Решение

Първи начин – чрез рекурсивна процедура, която реализира рекурсивен процес

```
(define (fib n)
   (cond [(= n 0) 0]
         [(= n 1) 1]
         [else (+ (fib (- n 1)) (fib (- n 2)))]))
```

Процесът на оценяване на обръщението към тази процедура може да бъде разглеждан като процес на обхождане в дълбочина на дърво от вида:



Затова процесът, породен от тази процедура, се нарича ***дървовидно рекурсивен*** (говори се още за ***дървовиден рекурсивен процес***).

Анализ:

- Fib(n) расте експоненциално с нарастването на n. По-точно, Fib(n) е най-близкото цяло число до $\Phi^n/\sqrt{5}$, където $\Phi = (1+\sqrt{5})/2 \approx 1.6180$ е положителният корен на уравнението

$\Phi^2 = \Phi+1$;

- времето за намиране на Fib(n) е пропорционално на броя на генерираните рекурсивни обръщения (броя на стъпките, за които се извършва пресмятането), който е равен на броя на възлите в дървото, а той от своя страна е от порядъка на 2^{n-1};

- обемът на необходимата памет е пропорционален на дълбочината на дървото, която е от порядъка на n.

Втори начин – чрез рекурсивна процедура, която реализира итеративен процес

Идея. Използват се две цели числа a и b, инициализирани съответно с 0 и 1, върху които n-кратно се прилага следната едновременна трансформация: a' := b, b' := a+b.

```
(define (fib n)
   (define (fib-iter a b count)
      (if (= count 0)
          a
          (fib-iter b (+ a b) (- count 1))))
   (fib-iter 0 1 n))
```

Анализ. Тази процедура поражда линеен итеративен процес, броят на стъпките на който е пропорционален на **n**.

Следователно, времето, необходимо за пресмятане на Fib(n) чрез двете предложени процедури, е твърде различно (в първия случай то расте експоненциално, а във втория - линейно). При това, разликите са съществени дори при малки стойности на n.

От показаното по-горе обаче не следва, че дървовидната рекурсия е нещо безполезно. Тя е особено подходяща при работа с рекурсивни типове данни. Самият процес на работа на интерпретатора на Racket (процесът на оценяване на изрази в Racket) е дървовиден рекурсивен процес. Дори и при числови данни дървовидната рекурсия често е много удобно средство. Например, в задачата за намиране на Fib(n) решението чрез дървовидна рекурсия е по-неефективно, но е много по-естествено (представлява точен запис чрез средствата на езика Racket на дефиницията на числата на Фибоначи).

Пример, който демонстрира ползата от дървовидна рекурсия при числови данни: намиране на броя на възможните начини за „разваляне“ на дадена парична сума на стотинки (например, на монети от 1, 2, 5, 10, 20 и 50 ст.).

Решение

Идея. Избираме и фиксираме една от възможните монети (най-добре е да се въведе наредба между монетите в зависимост от означаваната от тях сума и да се фиксира най-едрата монета). Тогава общият брой на търсените начини е равен на броя на начините за „разваляне“ на сумата без използване на избрания вид монети плюс броя на начините за „разваляне“ на дадената сума с използване на поне една монета от избрания вид.

Следователно,

брой начини за „разваляне“ на сумата a, използвайки n типа монети	**=**	**брой начини за „разваляне“ на сумата a, използвайки n-1 типа монети**	**+**	**брой начини за „разваляне“ на сумата a-d, използвайки n типа монети**

Тук **d** е номиналната стойност на фиксирания тип монети.

Базови (гранични) случаи:
- **a=0**: 1 начин;
- **a<0**: 0 начина;
- **n=0** (и **a>0**): 0 начина.

```
(define (count-change amount)
   (define (cc amount kinds)
      (cond [(= amount 0) 1]
            [(or (< amount 0) (= kinds 0)) 0]
            [else (+ (cc (- amount (denom kinds)) kinds)
                     (cc amount (- kinds 1)))]))
   (define (denom kinds)
      (cond [(= kinds 1) 1]
            [(= kinds 2) 2]
            [(= kinds 3) 5]
            [(= kinds 4) 10]
            [(= kinds 5) 20]
            [(= kinds 6) 50]))
   (cc amount 6))
```

Забележка. При тази дефиниция също се получава дублиране на някои пресмятания, както и при първата дефиниция на функцията **fib**, но тук не е така очевидно как би изглеждал един по-ефективен алгоритъм.

Процедури от по-висок ред

Същност на процедурите 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.

Дефиниция. Процедури, които манипулират други процедури, се наричат ***процедури от по-висок ред***.

Използване на процедури като параметри

Нека разгледаме като примери следните три процедури.

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

Решение. 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**.

Решение

```
(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*.

Решение

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

Забележка. Дефинираната току-що процедура може да се използва за пресмятане на приближения на числото π примерно по следния начин:

```
> (* 8 (sum-pi 1 9000))
3.14137043137031
```

Както се вижда, полученото по този начин приближение не е много добро (точната стойност на числото π е 3.14159265358979...). Нещо повече, при увеличаване на стойността на b не се получава съществено подобрение на точността.

Коментар. 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:

$$\sum_{a}^{b} f(n) = f(a) + ... + f(b)$$

В тази формула *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))
```

Като по-сложен пример за използване на тази процедура ще покажем как може да се дефинира процедура за приближено пресмятане на определен интеграл $\int_a^b f(x)dx$ по формулата (метода) на правоъгълниците. Ще смятаме, че са дадени границите на интегриране *a* и *b*, функцията *f* и стъпка *h* - такава, че *(b-a)/h* е цяло число. Тогава пресмятането на търсения интеграл по метода на правоъгълниците се извършва с помощта на следната формула:

$$\int_a^b f(x)dx \approx f(a+h/2)*h + f(a+h+h/2)*h + f(a+2h+h/2)*h + \ldots =$$

$$[f(a+h/2) + f(a+h+h/2) + f(a+2h+h/2) + \ldots]*h$$

Тогава процедурата за приближено пресмятане на $\int_a^b f(x)dx$ по формулата на правоъгълниците изглежда както следва:

```
(define (integral f a b h)
   (define (next x) (+ x h))
   (* (sum f (+ a (/ h 2)) next b) h))
```

Пример за използване на тази процедура:

```
> (integral cube 0 1 0.001)
0.24999875000001
```

Точната стойност на интеграла е 0,25.

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))
```