Функционално програмиране с езика Scheme

Бойко Банчев

boykobb@gmail.com | 0889-392057

Институт по математика и информатика – БАН,
Факултет по математика и информатика – СУ

18.1.2017

Актуално копие на този текст се намира тук

$\Downarrow$

http://www.math.bas.bg/bantchev/teaching/scheme/fp.pdf

(Тук и по-нататък всеки текст в този цвят съдържа препратка)

Същност и роля на ФП

Програмиране чрез функции:
всяко действие е или определение, или пресмятане на израз.

Функции-стойности. Анонимни функции.

Образуване на функции от функции: композиране,
(частично) прилагане, извличане на контекст.

Едноаргументни функции. Разлагане на функция по аргументи.

Съпоставяне на аргумент с образец.

История, корени и развитие на идеите. Ролята на λ- и
комбинàторното смятане и на езиците Lisp, CPL, ISWIM, PAL,
Pop-2, Scheme, FP/FL, ML, Hope, Miranda, Рефал и др.

Функционалност и императивност в програмирането

Критика на императивния (команден) стил на програмиране.

```
// Пример: намиране на средно аритметично
            avg = 0;
            for (i = 0; i < n; ++i)
              avg += a[i];
            avg /= n;
```

- Разкъсване на действието на дребни части.
- Работа със структурни данни на части вместо цялостно.
- Преплитане на действия и данни.
- Елементи не по съществото на задачата.
- Уязвимост към грешки.
- Ниска съчетаемост на съставките на програмата.

Понятията *променлива*, *контекст*, *свързване*, *последствие*.

ФП и формалното третиране на програми.

Внимание: „хлъзгави“ понятия!

В програмирането, особено функционалното, широко се използва понятието стойност и такива, свързани със съотнасяне на стойности. Необходимо е особено внимание към последните, за да се избегне размиване на смисъла и възникване на грешки.

Следните пет свойства са сходни, но различни:

идентично – означава „едно и също“;

неразличимо – отсъства или е целенасочено пренебрегнат признак или начин за различаване;

еднакво – отнася се до строежа и състава, напр. две програмни процедури с еднакъв текст, два еднакво определени даннови типа, две структури от данни с еднакви, равни или идентични една на друга съставки;

равно – от гледна точка на операция за сравнение „равно на“; прилага се напр. за числа и други стойности, но не функции; еднакви структури от данни може да се смятат за равни или не;

еквивалентно – за рефлексивно, симетрично и транзитивно отношение; може да не е нито „равно“, нито „еднакво“, например за остатъци по модул на числа, за редици с една и съща дължина и пр.

Разновидности на ФП

- ФП, основано на λ-смятане – две действия:
  - абстрахиране – извличане, образуване на функция по зададен израз и параметри в него;
  - прилагане на функция.
- Ограничено ФП: като горното, но функциите не са стойности.
- Апликативно или безаргументно ФП: основано на комбинàторно смятане – само прилагане, няма абстрахиране (функциите нямат явни аргументи).
- Конкатенативно ФП: композиране на безаргументни функции (няма явно прилагане).

Динамично и статично типизирани езици

Пример за динамична типова система: Common Lisp.

Пример за статична типова система: Хиндли-Милнър.

Примери: функция за намиране на средно аритметично

безименни функции с явен аргумент

```
(lambda [ns] (/ (fold-left + 0 ns) (length ns)))
```
Scheme

```
fun (Ns) -> sum(Ns) / length(Ns) end
```
Erlang

```
[](vector<N> ns) -> N
  {return accumulate(ns.begin(),ns.end(),0)
          / ns.size();}
```
C++

изрази със стойност функция: безаргументен стил

```
÷∘[/+,length]
```
FP

```
/[sum,tally]
```
Nial

```
+/ % #
```
J

изрази със стойност функция: конкатенативен стил

```
[sum] [size] cleave /
```
Joy

```
[ sum ] [ length ] bi /
```
Factor

```
dup 0 exch {add} forall exch length div
```
PostScript

Разпространение на ФП по езици

Семейството на Lisp:

Scheme (вкл. Guile, Racket), Common Lisp, Clojure, Shen, ...

Съвременни ФЕ с развити типови системи:

строго функционални – Haskell, Clean, Miranda, Elm;
предимно функционални – ML, OCaml, Alice, F#, GeomLab;
трансформационни – Pure.

Функционално паралелно и разпределено програмиране:

Erlang.

Конкатенативни (стекови):

Joy, Factor, PostScript.

Работа с масиви или редици, неявен паралелизъм:

предимно и по специфичен начин функционални –
APL, J, K, Nial, SequenceL, U.

Хибридни – Ф и релационни или обектноориентирани:

Scala, Dylan, Oz, Mercury, Pop-11, Io.

Универсални, за системно и приложно програмиране:

доразвити с елементи на ФП – C++, Java, C#;
(ново)създадени с възможности за ФП – D, Go, Rust.

Съвременни динамични:

JavaScript, Lua, Ruby, Falcon, REBOL, (донякъде) Python.

Програмиране на Scheme

Scheme в контекста на ФП

Диалект на Lisp,
    който е
минималистичен (малко на брой конструкции),
    но
концептуално мощен (мощна основа, висока комбинативност)
    и
пълен (отлична аритметика, стандартен и файлов В/И и пр.),
    предназначен основно за
изследвания и обучение по програмиране.

Първият Lisp с лексично свързване на несобствените имена (дотогава е динамично).

Понастоящем има множество реализации и диалекти.

Стандартизиране. Ролята на R^4RS, R^5RS, R^6RS и R^7RS.

Някои реализации и диалекти на Scheme

Chez Scheme – високопроизводителен компилатор и интерпретатор за R^6RS.

Racket – компилатор, интерпретатор, множество разширения, интегрирана работна среда.

Guile – множество разширения; самостоятелно и скриптове за Linux.

Gauche – команден език (shell) за Linux и подобни ОС.

Bigloo – език и компилатор за системно и приложно програмиране.

BiwaScheme – реализация на (и интегрираност с) JavaScript, за програмиране в уеббраузър или автономно (чрез Node.js).

Използвано в курса

- Chez Scheme – интерпретаторът *Petite Chez Scheme*.

  Настоятелно се препоръчва да се използва този архив – освен интерпретатора, той съдържа текстовия редактор *SciTE*, настроен за използване с *Petite Chez Scheme*. (Указания във връзка с това има на следващата страница.)

- Основен справочник по езика – книгата
  R. Kent Dybvig, *The Scheme programming language*, 4^{th} ed., MIT Press, 2009 (онлайн тук).

- Кратък справочник за голяма част от езика:
  http://www.math.bas.bg/bantchev/teaching/scheme/scheme-reference.txt.

- Множество препратки към литература и други ресурси:
  http://www.math.bas.bg/bantchev/place/scheme.html.

Начало на работа със *SciTE* и *Petite Chez Scheme*

1. Изтеглете този архив.
2. Цялото съдържание на архива е в директория `petite-chez` в него. Разархивирайте на произволно място `...`, при това ще се образува посочената директория: `...\petite\`.
3. В директорията `...\petite\bin\i3nt` намерете файла `SciTE.exe` и направете препратка (shortcut) към него, например на работния плот (Desktop).
4. Стартирайте редактора *SciTE* чрез препратката.
5. Създайте **файл с име, завършващо на** `.scm` **или** `.ss` (стандартни за езика) – този файл трябва да съдържа програмата на Scheme. Файлът може да се намира където и да е.
6. Стартиране или спиране на интерпретатора с програмата от файла става чрез менюто `Tools`.

Основни действия със *SciTE* и *Petite Chez Scheme*

Стартиране на интерпретатора – с `Go` или `Run in true console` от меню `Tools` или съответно с `F5` или `Ctrl+1` (програмата автоматично се записва във файла и изпълнява оттам)

Веднъж стартираният интерпретатор продължава да работи, докато не се изпълни `(exit)` – в програмата или въведено допълнително

Може да се прекрати и със `Stop Executing` от `Tools` (или `Ctrl+Break`)

В истинска конзола (самостоятелен прозорец, `Ctrl+1`), може да се прекрати и с `Ctrl+D`

`Ctrl+F6` премества курсора от полето за редактиране в псевдоконзолата (под-прозорец) или обратно

`Shift+F5` изчиства съдържанието на псевдоконзолата

`F8` скрива или показва псевдоконзолата

`Vertical Split` от `Options` поставя псевдоконзолата отдясно или отдолу

Програма и изпълнение

Програмата е редица от определения и изрази или само второто (или само първото, но такава програма е безсмислена).

Изразите се пресмятат и произвеждат стойности, а възможно и някакви други последствия, като присвоявания, запис във файл или другаде.

Определенията произвеждат свързвания на имена със стойности.

Синтактичен строеж

Всяка конструкция в програмата е непосредствено представена стойност (константа), специален знак или има фòрмата на списък. Същото се отнася и за частите на конструкциите.

Списък е редица от членове, заградени в `( )` или `[ ]` (членовете са константи, специални знаци или списъци).

Списъкът може да бъде и празен.

Двата вида скоби са взаимозаменяеми.

Пресмятане на изрази

Най-простите видове изрази са константи или имена, които цитират стойности, напр.

```
74      "щем!"      mig-29-load
```

Останалите изрази задават пресмятания – на функции или чрез специални форми, и съответно се задават чрез списъци.

Когато списък задава повикване на функция, първият му член задава функцията, а останалите (ако има такива) – аргументи. И функцията, и аргументите са на свой ред изрази, които се пресмятат в неопределен ред. След добиване на съответните стойности се извършва повикването.

```
(+ (* 3 x) 7)
(if (> a b) (f x) (g y))
((f t) x (+ p q r) z)
```

Ограждането на всяко повикване на функция в скоби позволява аритметични и други операции да имат променлив брой аргументи.

Функциите могат да имат небуквени имена – каквито в други езици се наричат операции. В Scheme няма разлика между двете, използват се равнозначно и двата термина.

Специални форми

Когато име или списък трябва да се разглежда като константа, пред него се пише `'` или се образува списък с `quote`, напр.

`'(ab c def)` или `(quote (ab c def))`

Първото е съкратен вид на второто.

`quote` е псевдофункция, т.к. нейният аргумент не се пресмята: действието ѝ се състои именно в потискане на пресмятане, което иначе се подразбира.

От трите аргумента на `if` единият също не се пресмята: `if`, също като `quote`, не е функция.

`if` и `quote` са примери на *специални форми*. За разлика от повикванията на функции, всяка СФ има специфично за нея правило за изпълнение.

Полупресмятане с quasiquote

Специалната форма `quasiquote`, или съкратено `` ` ``, позволява отделно за всяка част на израз да се избира дали да се пресмята или не.

Всяка от подформите `unquote` (съкр. `,`) и `unquote-splicing` (съкр. `,@`) посочва аргумент на `quasiquote`, който да се пресмята. Резултатът се влага на съответното място, като при това `,` съхранява списъчната му структура, а `,@` я разрушава:

```
`(1 (list 2 3 4) 5)      --->   (1 (list 2 3 4) 5)
`(1 ,(list 2 3 4) 5)     --->   (1 (2 3 4) 5)
`(1 ,@(list 2 3 4) 5)    --->   (1 2 3 4 5)
```

Формите `` ` ``, `,` и `,@` могат да се влагат една в друга, така че пресмятаните части да се избират точно, колкото и сложен да е изразът.

Разполагане на програмен текст

За разполагането на текста на програма няма езикови изисквания, но то е много важно за четливостта.

Съществуват обичаи и съглашения:

- части от едно синтактично равнище се записват последователно хоризонтално или вертикално;
- влагане / подчиненост и продължаване на конструкция се изразяват с отстъп;
- затваряща скоба) се поставя непосредствено в края на конструкцията, на същия ред.

(На едно място може да има неограничен брой затварящи скоби.)

```
(define [f t]
  (define [g q]
    (if (null? q) '()
      (cons (map car q) (g (h cdr q)))))
  (if (null? t) '() (g (list t))))
```

Коментари

`; ...`	до края на реда
`#| ... |#`	блокови, могат да се влагат
`#; ...`	за (под)израз

Видове стойности и изобразяване на константите

Числа

число $\supset$ комплексно $\supset$ реално $\supset$ рационално $\supset$ цяло

`3+5i` `2.71` `355/113` `725`

`#b110001011101` – двоично, `#x20fa4` – 16-чно, ...

`#e3.141592` – „точно“ (`392699/125000`)

`#i355/113` – „неточно“ (`3.141592920353982`5)

Булеви

`#t` (истина) `#f` (неистина)

Литери (characters)

`#\k` `#\λ` `#\\` `#\xb7` `#\x03bb`

Именувани:

`#\newline` `#\space` `#\tab` `#\backspace` `#\alarm` `#\esc` `#\nul` и др.

Низове

`"щам H1N1"`

Отделни литери в низ – чрез имена или кодове:

`\n` `\t` `\b` `\a` `\\` `\x042f;`

Например `"\x042f;ndex"` (`"Яndex"`).

Символи

`a`, `z101`, `al-Khwarizmi` `<` `/` `1+` `*10` `...`
(Всеки символ е уникат!)

Двойки

Понятието *двойка* е сред най-важните в Scheme. Именно от двойки се образуват списъците, а те пък са структурата, основна за данните в езика и тази, чрез която се представя самата програма.

Двойка е структура от две (препратки към) стойности. Образува се с функцията `cons` (от construct), а двете стойности-части се извличат съответно с `car` и `cdr`. По традиция второто се произнася „кудър“. И двете имена нямат нищо общо със смисъла на извършваните действия – те са нелепо оцелял атавизъм от първата реализация на Lisp.

Списъци

По принцип частите на двойка са смислово равнозначни, но от гледна точка на образуването на списъци се приема, че първата дава челния член, а втората – остатъка от списък. Ако остатъкът е на свой ред двойка, нейната първа част е вторият член на списъка, втората част препраща към остатъка след втория член на списъка и т. н.

Така е налице свързан списък, в който членове са частите `car` на двойките, а връзки са препратките в частите `cdr`. Списъкът завършва с двойка, частта `cdr` на която препраща към стойност, която не е двойка. Ако тази стойност е празният списък (изобразяван с `()`), списъкът се нарича *истински*, в противен случай – *неистински*.

Езикът предлага голям брой вградени и библиотечни функции за истински списъци.

Записване и примери на списъци

Участието на образуващите списъка двойки може да се изрази явно в програмата чрез т. нар. точков запис. Например:

`(1 . 2)` е неистински списък – двойка от 1 и 2, каквато дава и изразът `(cons 1 2)`;

`(1 . ())` е истински списък с единствен член 1 – това е списъкът `(1)`, който се образува и с `(cons 1 '())`;

`(1 . (2 . (3 . ())))` е истински списък с членове 1, 2 и 3, който може да се запише и като `(1 . (2 . (3)))`, `(1 . (2 3))`, `(1 2 . (3 . ()))`, `(1 2 . (3))`, `(1 2 3 . ())` или `(1 2 3)`;

`(() . ())` е истинският списък `(())` с единствен член празен списък (образува се и с `(cons '() '())`);

`(1 . (+ 2 3))` е списъкът `(1 + 2 3)` (истински).

Списъците `(+ 1 2 . ())` и `(+ . (3 5))` се записват и като `(+ 1 2)` и `(+ 3 5)` (и пресмятат съответно 3 и 8).

Записите `(1 . (2 . 3))` и `(1 2 . 3)` представят един и същи неистински списък, образуван от две двойки, а записите `(1 . 2 3)` и `(1 . 2 . 3)` нямат смисъл и са синтактично погрешни.

Композиции на car и cdr

Композиция от няколко (до 4) `car` и `cdr` може да се запише с едно име, напр. `(caadar x)` е равнозначно на `(car (car (cdr (car x))))`:

```
(caadar '(((a b c d) (e f g) h) i j))   ⟶   e
```

Стек, йерархия

От гледна точка на списъците `cons` образува списък чрез добавяне на първия си аргумент към втория – самият той най-често списък. Казваме, че `cons` удължава списъка откъм началото му. В същия смисъл `cdr` пък „скъсява“ списъка – всъщност дава остатъка след първия член.

Така списъкът е и стек, в който `cons` и `cdr` са действията добавяне и отнемане на членове, а `car` дава върховия член.

Понеже достъпът до последователните членове на списък става само чрез прилагане на последователни `cdr` от началото му, списъкът е асиметрична редица: достъпът до членовете ѝ е толкова по-бавен, колкото по-далеч са те от началото.

Списъкът е и (потенциално) йерархична структура, тъй като всеки негов член може да бъде списък – от гледна точка на йерархията всеки член-списък е наследник или подчинен на този, на който е член.

Други структурни стойности

Вектори и байтвектори

Векторите са редици с пряк достъп до всеки от членовете.

Байтвекторите са разновидност, или подтип на векторите, при който всеки член е цяло неотрицателно число до 255 и следователно се побира в един октет.

`#("qwer" (3 5 8) #\x1a)`	вектор с три члена
`#vu8(113 9 255 12)`	байтвектор с четири члена

Абстрактни структури

Асоциативни списъци и асоциативни таблици.
Тези структури съществуват в лицето на по няколко вградени или библиотечни функции за всяка и в този смисъл са абстрактни.

Различаване на типа на стойност

`number?`	число
`boolean?`	булево
`symbol?`	символ
`char?`	литера
`string?`	низ
`pair?`	двойка (вкл. списък)
`vector?`	вектор (вкл. байтвектор)
`procedure?`	функция (процедура)
`null?`	празният списък (единствена стойност от типа)

Всяка стойност удовлетворява най-много един от тези предикати.
Допълнително има

`list?`	истински списък (подтип на този на двойките)
`bytevector?`	байтвектор (подтип на векторния)
`record?`	смес („скрит“ тип)

Условни пресмятания

Функциите за сравняване, например на числа, <, <=, =, >= и >, пораждат булеви стойности. В ролята на условие освен булева стойност може да се използва и всяка друга: всяка различна от `#f` стойност се тълкува като „истина“.

Следните специални форми по различни начини служат за организиране на условно пресмятане.

`(and` v_1 v_2 ...`)`
: Пресмятат се изразите v_1 v_2 ... докато или някой от тях се окаже равен на `#f`, или се изчерпят. Резултатът е последната пресметната по този начин стойност.

`(or` v_1 v_2 ...`)`
: Пресмятат се изразите v_1 v_2 ... докато или някой от тях се окаже различен от `#f`, или се изчерпят. Резултатът е последната пресметната по този начин стойност.

`(if` v v_1 v_2`)`

Пресмята се v и след това или v_1, или v_2 според това дали v е различно или равно на `#f`. Резултатът е последната пресметната стойност.

`(when` v v_1 v_2 `...)`

Ако стойността на v е различна от `#f`, пресмятат се v_1 v_2 `...` и резултатът е последната пресметната стойност. В противен случай резултатът е неопределен.

`(unless` v v_1 v_2 `...)`

Същото като `(when (not` v`)` v_1 v_2 `...)`.

(`case` v *клаузи*)

Всяка клауза е списък, първият член на който е списък от константи, а останалите – изрази. Стойността на v се сравнява с константите по реда на клаузите и щом се открие равенство, пресмятат се изразите в съответната клауза. Ако равенство за v не е открито, но последната клауза вместо списък от константи съдържа `else`, пресмятат се изразите в нея. Резултатът е стойността на последния пресметнат израз или неопределен, ако равенство не е открито и няма `else`.

Например, ако `x` е равно на 4:

```
(case 4 ((1 2) 'A) ((3 4 5) 'B) (else 'C))   ⟶ B
(case x ((1 2) 'A) ((3 x 5) 'B) (else 'C))   ⟶ C
(case 'x ((1 x) 'A) ((3 4 5) 'B) (else 'C))  ⟶ A
```

(cond *клаузи*)

Всяка клауза е списък, първият член на който е условие. Условията се пресмятат последователно за клаузите и щом се открие истинно (различна от `#f` стойност), пресмятат се изразите в съответната клауза. Ако истинно условие не е намерено, но последната клауза вместо условие съдържа `else`, пресмятат се изразите в нея. Резултатът е стойността на последния пресметнат израз или неопределен, ако няма нито истинно условие, нито `else`.

Следният израз има стойност един от няколко символа в зависимост от типа на стойността на `x`:

```
(cond ([char? x]      'charismatic)
      ([procedure? x] 'orderly)
      ([pair? x]      'espoused)
      ([string? x]    'tense)
      (else           'exceptional))
```

Действия с истински списъци

`length` дължина на списък

```
(length '())                  ⟶  0
(length '((a b) c (d e f)))   ⟶  3
```

`reverse` обръщане на списък

```
(reverse '())                 ⟶  ()
(reverse '((a b) c (d e f)))  ⟶  ((def) c (a b))
```

`list` списък от посочени членове

```
(list)                        ⟶  ()
(list '())                    ⟶  (())
(list '(a b) 'c '(d e f))     ⟶  ((a b) c (d e f))
```

Образуване на истински и неистински списъци

`append` истински или неистински списък от членовете на аргументите; ако първият аргумент не е истински списък, той е единствен аргумент и е резултат-несписък

```
(append 'a)                          ⟶ a
(append '(a b) '(c d e))             ⟶ (a b c d e)
(append '(a b) '(c (d e)) 'f)        ⟶ (a b c (d e) . f)
(append '(a b) '((c d) e) '(f g . h)) ⟶
                                       (a b (c d) e f g . h)
```

`cons*` прилагане на `cons` отдясно наляво

```
(cons* 'a)                           ⟶ a
(cons* 'a 'b 'c 'd)                  ⟶ (a b c . d)
(cons* '(a b) '(c d e))              ⟶ ((a b) c d e)
(cons* '(a b) '(c (d e)) 'f)         ⟶ ((a b) (c (d e)) . f)
(cons* '(a b) '((c d) e) '(f g . h)) ⟶
                                       ((a b) ((c d) e) f g . h)
```

Член и остатък на (истински или неистински) списък

`(list-tail` l k`)`

Равнозначно на `(c`$\underbrace{\texttt{d}\ldots\texttt{d}}_{k}$`r` l`)`.

(Приблизително: остатъкът след първите k члена на списъка l.)

```
(list-tail '(a (b c) d) 0)      ⟶ (a (b c) d)
(list-tail '(a (b c) d) 2)      ⟶ (d)
(list-tail '(a b . c) 1)        ⟶ (b . c)
(list-tail '(a b . c) 2)        ⟶ c
(list-tail '(a b . c) 3)        ⟶ грешка
```

`(list-ref` l k`)`

Равнозначно на `(car (list-tail` l k`))`.

(Приблизително: членът след първите k на списъка l.)

```
(list-ref '(a (b c) d) 0)       ⟶ a
(list-ref '(a (b c) d) 2)       ⟶ d
(list-ref '(a b . c) 1)         ⟶ b
(list-ref '(a b . c) 2)         ⟶ грешка
```

Изрази функции. Безименни функции

Получаваната от пресмятане на израз стойност може да бъде функция. Основното средство за образуване на функционна стойност е специалната форма `lambda`, най-вече във вида

`(lambda (`*параметри*`)` *изрази*`)`

Такъв израз (λ-израз) има за стойност образуваната функция. Когато функцията бъде повикана, тя образува друга стойност чрез пресмятане на дадените изрази – тяло на функцията – при което стойности на параметрите са тези на аргументите от конкретното повикване. Образуваната от повикването стойност е тази на последния пресметнат израз. Например

`(lambda [] 5)`	стойност функция без параметри
`((lambda [] 5))`	повикване, стойност 5
`(lambda [a b] (+ (* 5 a) b))`	стойност функция с два пар-ра
`((lambda [a b] (+ (* 5 a) b)) 7 4)`	повикване, стойност 39

```
(lambda [a] (lambda [b] (+ (* 5 a) b)))
```

функция с един параметър и резултат функция с един параметър (срв. с функцията с два параметъра по-горе)

```
((lambda [a] (lambda [b] (+ (* 5 a) b))) 7)
```

прилагане на горната функция към 7
(резултатът е функция с един параметър, `b`)

```
(((lambda [a] (lambda [b] (+ (* 5 a) b))) 7) 4)
```
$\longrightarrow 39$

прилагане към `4` на резултата от прилагането към 7

Свързване. Именувани функции

Специалната форма `define` създава свързване между име и стойност в текущия контекст. Основният ѝ вид е

`(define` *име стойност*`)`

(Ако стойността е пропусната, създава се „отсъстваща“, но именувана стойност.)

Когато стойността е λ-израз както по-горе, т.е. за

`(define` *име* `(lambda (`*параметри*`)` *изрази*`))`

може да се използва съкратеният запис

`(define (`*име параметри*`)` *изрази*`)`

Примери на именувани функции

Функцията `seq` образува списък от числата $m, m+1, \ldots, m'$, където $m' \leq n < m'+1$ (за $m > n$ – празен списък).

Определението е рекурентно:

```
(define [seq m n]
  (if (<= m n) (cons m (seq (+ 1 m) n)) '()))
```

Функцията `len` намира дължината на списък, не непременно истински (вградената функция `length` е неприложима за неистински списъци). При неистински списък вторият член на последната двойка също се брои за член на списъка. И това определение е рекурентно:

```
(define [len as]
  (cond
    ([pair? as] (+ 1 (len (cdr as))))
    ([null? as] 0)
    (else 1)))
```

Терминологична бележка

Определенията на пряко или косвено повикващи себе си функции, както аналогични определения в математиката, наричаме **рекурентни**.

Пресмятанията на такива функции наричаме **рекурсивни**.

Рекурентността е езиково средство за описване на обекти и съответно – свойство на описанието на един или друг конкретен обект, а *рекурсивността* е характеристика на изчислителния процес, осъществяващ пресмятането на функция. Възможно е да съществуват и се прилагат различни процеси за пресмятане на една и съща функция, включително рекурентноопределена функция да се пресмята нерекурсивно и нерекурентно зададена функция да се пресмята рекурсивно.

Контекст на пресмятанията. Глобален контекст

За да се извършват пресмятания, на цитираните в тях имена се съпоставят свързани с имената стойности. Съвкупността от направените и достъпни в дадена част на програмата свързвания се нарича неин контекст и казваме, че пресмятанията се извършват в този контекст.

Тъй като контекстът винаги съответства на определена област на (текста на) програмата, думата „контекст“ се използва и по отношение на областта.

Предопределените в езика свързвания, както и направените чрез `define` на най-външно равнище в програмата, образуват глобалния контекст. Пресмятанията на най-външно равнище се извършват в него.

Влагане на контексти

Свързванията, направени локално в обособена област на програмата, са достъпни само в нея: цитирането на локално свързано име обозначава съответната стойност само в дадената област. По този начин в рамките на областта множеството от свързвания се обогатява – то включва както наличните извън областта, така и локалните свързвания. В такива случаи казваме, че има влагане на контексти.

Локалното свързване може да се отнася до име, за което има свързване във външната, обхващащата част на програмата. В този случай външното свързване е недостъпно в областта на действие на локалното – казваме, че е скрито от него.

Контекст на функция

Всеки λ-израз образува контекст, локално свързани в който (при всяко конкретно повикване) са параметрите на функцията, както и изрично направените чрез `define` в тялото на функцията свързвания.

При определяне на функция чрез `define` за името ѝ се създава свързване в текущия контекст, а за параметрите – в обособения от съответната `lambda` вложен контекст.

Забележка. За образуване на контекстна област съществуват няколко средства, но основни са `lambda` и `case-lambda` (разглеждана по-нататък). Останалите се свеждат до тях.

Пример за обособяване на контексти чрез `define`

Следната функция пресмята полином по схемата на Хорнър:

$$c_0x^n+c_1x^{n-1}+\cdots+c_{n-1}x+c_n = (\ldots((c_0x+c_1)x+c_2)x+\cdots)x+c_n$$

```
(define [poly cs x]
  (define [poly1 cs v]
    (define [h c]
      (+ (* v x) c))
    (if (null? cs) v (poly1 (cdr cs) (h (car cs)))))
  (poly1 cs 0))
```

Параметри на `poly` са списъкът от коефициенти `cs` и променливата `x`. Едноименният параметър `cs` на локалноопределената функция `poly1` скрива този на `poly`, така че той се цитира само на последния ред – навсякъде другаде `cs` е от `poly1`. Тялото на функцията `h`, локалноопределена в `poly1`, освен до `c`, което е локално за самата `h`, има достъп и до `v` от `poly1`, и до `x` от `poly`.

Впрочем . . .

Какво би станало, ако в горната функция сменим името `poly1` с `poly`?

Тъй като `poly1` е определена в тялото на `poly`, новото свързване за името `poly` скрива външното, което така или инак не се цитира другаде.

Функцията `poly1` бива повиквана от две места – от собственото ѝ тяло и от това на `poly`. И двете принадлежат на контекста, в който е в сила свързването за `poly1` и смяната на това име не променя смисъла на програмата – повиква се все същата функция.

Забележка

Думата „влагане“ във връзка с контекст може да бъде объркваща. Когато област B е текстово вложена в A казваме, че контекстът (на) B е вложен в (този на) A, но множеството на достъпните в B свързвания съдържа това на A, т.е. между множествата е налице влагане в обратната посока!

(Същият вид двусмислие възниква и другаде, например във връзка с отношението клас-подклас в обектноориентираното програмиране: свойствата на „надкласа“ са подмножество на тези на „подкласа“.)

Извличане на контекст

При образуване на функционна стойност заедно със самата функция се съхранява обхващащият контекст – нелокалните за функцията свързвания. Така извлеченият, прикрепен към функцията контекст позволява тя да бъде изпълнявана където и да е, включително извън контекста на образуването ѝ.

В примера всяко повикване на `add` създава свързване за `n` и функция, която го използва. Запомняйки свързването, всяка създадена от `add` функция може да се изпълнява където и да е, със станалата нейна собствена стойност на `n`.

```
(define [add n]
  (lambda [k] (+ n k)))
(define add1 (add 1))
(define add5 (add 5))
```

`(add1 20)` $\longrightarrow$ 21

`(add5 20)` $\longrightarrow$ 25

Стойностите на променливи от извлечен при създаване на функция контекст могат да бъдат променяни в нея. В примера `agen` образува функции, всяка от които при повикване променя своя екземпляр на `n`, добавяйки към стойността му тази на аргумента от повикването (`set!` е императивна процедура, специална форма, която тук присвоява $n+k$ на n).

```
(define [agen n]
  (lambda [k]
    (set! n (+ n k))
    n))
(define f (agen 10))
(define g (agen 100))
```

```
(f 2)   ⟶  12
(f 3)   ⟶  15
(g 4)   ⟶  104
(f 50)  ⟶  65
(g 10)  ⟶  114
```

Ако в даден контекст се образуват повече от една функции, те си го поделят и чрез извличането му. В примера `fgen` при всяко свое повикване създава свързване за `n` и двойка от поделящи си го функции. Повикване на коя да е от двете функции изменя общото за тях `n`.

```
(define [fgen n]
  (cons
    (lambda [k] (set! n (+ n k)) n)
    (lambda [k] (set! n (* n k)) n)))
(define fs (fgen 10))
(define add (car fs))
(define mul (cdr fs))

(add 2)   ⟶  12
(add 3)   ⟶  15
(mul 4)   ⟶  60
(add 50)  ⟶  110
(mul 10)  ⟶  1100
```

Поделянето на съвкупност от стойности от множество функции е същото като в обектноориентираното програмиране. В терминологията на ООП функция като `fgen` е *конструктор*, а повикването ѝ създава *обект*, на който функциите-членове са *методи*.

Пресмятане на изрази в контекст: `let` и `let*`

Специалната форма `let` дава възможност да се извършат пресмятания в обособен за целта контекст:

(`let` ((`$n_1$` `$v_1$`) (`$n_2$` `$v_2$`) ...) *изрази*)

В новообразувания контекст се създават свързвания на имената n_i със съответните стойности v_i и след това се пресмятат изразите. Стойността на последния израз е и стойността на `let`. Пресмятанията на v_i стават във външния (не новообразувания) контекст и в неопределен ред.

Следователно `let` е равнозначно на

((`lambda` (`$n_1$` `$n_2$` ...) *изрази*) `$v_1$` `$v_2$` ...)

Формата `let*` се различава от `let` по това, че пресмятанията на v_i и свързванията стават последователно, всяко следващо в контекста на предишното – като при вложени едно в друго `let`, по едно за всяка двойка n_i v_i.

Примери

```
(let ([x 1] [y 2])
  (let ([x y] [y x])
    (cons x y)))          ⟶ (2 . 1)
```

Намиране на корените на уравнението $ax^2+bx+c=0$:

```
(define [roots a b c]
  (let ([d (sqrt (- (* b b) (* 4 a c)))] [a2 (* 2 a)])
    (cons (/ (- (+ b d)) a2) (/ (- d b) a2))))
```

Сливане на два ненамаляващи числови списъка в също такъв резултат:

```
(define [merge a b]
  (cond ([null? a] b)
        ([null? b] a)
        (else (let ([x (car a)] [y (car b)])
                (if (<= x y)
                  (cons x (merge (cdr a) b))
                  (cons y (merge a (cdr b))))))))
```

Примери

Образуване на двойка от първите n и останалите членове на списък:

```
(define [split-at n xs]
  (if (= 0 n)
    (cons '() xs)
    (let ([rs (split-at (- n 1) (cdr xs))])
      (cons (cons (car xs) (car rs)) (cdr rs)))))
```

(Ефективно) построяване на списък от първия, последния, втория, предпоследния и т. н. членове на списък:

```
(define [volute xs]
  (let ([p (split-at (div (+ 1 (length xs)) 2) xs)])
    (define [interweave as bs]
      (if (null? as) '()
        (cons (car as) (interweave bs (cdr as)))))
    (interweave (car p) (reverse (cdr p)))))
```

Видове рекурентност

В много случаи на рекурентни определения на функции рекурентните обръщения доставят стойност, от която резултатът на функцията се образува с помощта на допълнителни действия – такива са например определенията на `seq`, `len`, `merge` и др. по-горе.

Когато такива действия няма – стойността на рекурентното повикване е и резултат на функцията – казваме, че рекурентността е *финална* (*заключителна*, *„опашкова“* (tail)). Такъв е случаят с `poly1` по-горе.

Нефиналната, същинската рекурентност се реализира с процес, рекурсивен или не, но потребяващ пропорционален на броя на повикванията обем памет. Финалната рекурентност може да се реализира като итеративен (цикличен) процес, заменяйки рекурентните повиквания с повторения и така потребявайки предварително фиксиран обем памет, независим от броя на повторенията. По определение на езика такава реализация е винаги налице в Scheme.

Пример за разклонена (нелинейна) рекурентност

Списък от „крайните членове“ (не двойки) на списък.
За всякакъв (и неистински) списък и дори не списък.

```
(define [flatten a]
  (cond
    ([null? a] '())
    ([pair? a] (append (flatten (car a))
                       (flatten (cdr a))))
    (else (list a))))
```

Разклонената рекурентност по принцип се реализира с рекурсивен процес.

Преобразуване към финалнорекурентна форма – сумиране

Непосредствено определение: за да получим сбора на членовете, добавяме първия от тях към сбора на останалите.

Рекурентността в определението е същинска, не финална.

```
(define [sum-1 ns]
  (if (null? ns)
    0
    (+ (car ns) (sum-1 (cdr ns)))))
```

Натрупване на резултата в променлива.

`f` има странично последствие (тя изобщо и не образува стойност) в контекста на основната функция, но е финалнорекурентна.

```
(define [sum-2 ns]
  (define s 0)
  (define [f ns]
    (unless (null? ns)
      (set! s (+ s (car ns)))
      (f (cdr ns))))
  (f ns)
  s)
```

Чисто функционално определение: резултатът, вместо отделна от помощната функция променлива, е неин параметър.

`f` е финалнорекурентна.

```
(define [sum-3 ns]
  (define [f ns s]
    (if (null? ns)
      s
      (f (cdr ns) (+ s (car ns)))))
  (f ns 0))
```

Итеративност във функционален стил: спец. форма do

```
(do ((n1 v1 w1) (n2 v2 w2) ...)
  (условие резултат)
  изрази)
```

Създават се свързвания, както с `let`, за имената n_i със стойностите v_i. Ако условието е истинно, пресмята се изразът *резултат* – неговата стойност е стойността на `do`. В противен случай се пресмятат *изрази* (тялото на цикъла), след което се създават **нови свързвания** на n_i със стойностите w_i – все едно с `let`, вложен в първия, наново се пресмята условието и т. н., докато то не се окаже истинно.

Въпреки, че формата `do` външно прилича на оператор за цикъл в императивен език, изпълнението ѝ всъщност е пресмятане на израз, т.е. е чисто функционално. То няма странични последствия, стига никой от участващите подизрази да няма такива.

В много случаи частта *изрази* („тялото“) отсъства.

Отново сумиране и пресмятане на полином

Сумиране като при `sum-3`, но тук не се налага въвеждането на помощна функция – повторността се постига чрез `do`:

```
(define [sum-4 ns]
  (do ([ns ns (cdr ns)]
       [s 0 (+ s (car ns))])
    ([null? ns] s)))
```

Пресмятане на полином (сравни с предишното определение):

```
(define [poly cs x]
  (do ([cs (cdr cs) (cdr cs)]
       [v (car cs) (+ (* v x) (car cs))])
    ([null? cs] v)))
```

Преобразуване към финалнорекурентна форма – обръщане

Искаме да съставим функция, аналог на `reverse`.

Тромаво ($\approx n^2$) и със същинска (не финална) рекурентност:

```
(define [rev-1 xs]
  (if (null? xs) '()
    (append (rev-1 (cdr xs)) (list (car xs)))))
```

Натрупване чрез вложена финалнорекурентна функция, в която резултатът е параметър:

```
(define [rev-2 xs]
  (define [f xs ys]
    (if (null? xs) ys
      (f (cdr xs) (cons (car xs) ys))))
  (f xs '()))
```

Важно наблюдение

Превръщането на резултата от пресмятане на функция в неин параметър позволява и рекурентността да се превърне във финална: функцията сама „пренася“ чрез рекурентните повиквания и преобразува резултата си, докато той стане окончателен. Така в много случаи функцията добива по-удобна за (ефективно) пресмятане форма.

Упражнения

Преобразувайте `seq` към финалнорекурентна форма. За целта числовата редица може да се породи в обратен ред и към резултата да се приложи `reverse`.

Използвайки `do`, съставете функция за обръщане на списък.

Рекурсивни локални определения – `letrec` и `letrec*`

Специалната форма `letrec` има синтаксис като на `let` и позволява пряко и косвено рекурентни локални определения: всяко може да цитира себе си и всяко друго.

Формата `letrec*` се различава от `letrec` както `let*` от `let`.

Следната функция е подобна на `rev-2`, но вложената помощна функция е определена чрез `letrec`, а не `define`:

```
(define [rev-3 xs]
  (letrec
    ([f (lambda [xs ys]
          (if (null? xs) ys
            (f (cdr xs) (cons (car xs) ys))))])
    (f xs '())))
```

Забележка. В контекст, различен от глобалния, формата `define` е равнозначна на `letrec*` и се превежда чрез нея.

Рекурентни локални определения – `let` с етикет

Специалната форма `let` допуска задаване на допълнително име пред тези в скоби:

(`let` f `((`n_1 v_1`) (`n_2 v_2`) ...)` *изрази*)

Името – в образеца тук f – обозначава образуваната от `let` функция, създавайки възможност за рекурентни обръщения.

В следното определение вложената помощна функция е определена чрез `let` с етикет. Това спестява изричното ѝ повикване от основната (сравни с `rev-2` и `rev-3`):

```
(define [rev-4 xs]
  (let f ([xs xs] [ys '()])
    (if (null? xs) ys
      (f (cdr xs) (cons (car xs) ys)))))
```

Както другите варианти на `let`, и този се свежда до прилагане на безименна функция. В случая обаче функцията има *локално* име, което позволява тя да се самоцитира.

Функции с променлив брой аргументи

Ако дадена функция трябва да допуска повиквания с *изменчив брой аргументи*, в определението ѝ трябва да се посочи *неопределен брой параметри*. За целта се използва някой от следните варианти на `lambda`:

(`lambda` *параметър изрази*)

(`lambda` (*параметри* . *параметър*) *изрази*)

В първия вариант при повикване на функцията *параметър* получава за стойност списък (възможно празен) от стойностите на аргументите.

Във втория вариант един или повече параметъра са посочени със самостоятелни имена. Функцията трябва да бъде повиквана с поне толкова на брой аргументи, а останалите образуват списък, получаван от *параметър* както в първия вариант.

`(lambda x 5)`	$\longrightarrow$	функция
`((lambda x 5))`	$\longrightarrow$	5 (повикване без аргументи)
`((lambda x 5) 2 3)`	$\longrightarrow$	5 (повикване с два аргумента)

На дадените два варианта на `lambda` отговарят следните варианти на `define`:

(`define` (*име* .*параметър*) *изрази*)

(`define` (*име параметри* .*параметър*) *изрази*)

(Както и другаде, *име* е това на функцията.)

Пример

Функциите с изменчив брой аргументи са удобни при изброявания с незададена отнапред дължина – спестява се образуване на списък при обръщение към функцията.

Следната функция пресмята полином, както по-горе. В случая изрично е посочено, че коефициентите са поне един. По зададени коефициенти функцията образува функция от неизвестното `x`.

```
(define [poly c . cs]
  (lambda [x]
    (do ([cs cs (cdr cs)]
         [v c (+ (* v x) (car cs))])
      ([null? cs] v))))
```

```
((poly 2 -7 4 6) 3)          ⟶ 9 (= 2x^3 − 7x^2 + 4x + 6 |_{x=3})
(define f (poly 2 -7 4 6))   ⟶ функция
(f 3)                        ⟶ 9
```

Пример

Изменчив брой аргументи е удобен също за инициране на пресмятане по рекурентна функция, както и за задаване на стойности на аргументи по подразбиране.

Следната функция е пореден вариант на обръщане на списък. Вместо да си служи с помощна функция, тя сама играе тази роля, когато бъде повикана с втори (натрупващия резултата) аргумент.

```
(define [rev-5 xs . ys]
  (if (null? ys)
    (rev-5 xs '())
    (let ([ys (car ys)])
      (if (null? xs) ys
        (rev-5 (cdr xs) (cons (car xs) ys))))))
```

В случая спестяването на помощната функция едва ли е изгодно – параметърът `ys` е списък в списък и трябва да се извлича оттам, а и всеки път, което е по същество излишно, се проверява дали вторият аргумент присъства в повикването.

case-lambda

Специалната форма `case-lambda` е още един начин да се образува функция с променлив брой аргументи. Общият ѝ вид е следният:

(`case-lambda` *клаузи*)

Всяка клауза има вида

(*образец израз*)

и е аналогична на λ-израз. Образецът в нея задава конкретен или неопределен брой параметри, както в *кой да е от вариантите* на `lambda`.

При обръщение към функцията образците се преглеждат по реда на клаузите до откриване на такъв, който отговаря на броя на аргументите, и се пресмятат изразите от съответната клауза.

Пример

Отново вариант на обръщане на списък – сходен с `rev-5`, но няма недостатъка вторият аргумент да се оказва поставен в списък.

Освен това, използването на образци и съпоставяне прави определението на функцията по-непосредствено и нагледно:

```
(define rev-6
  (case-lambda
    ([xs]    (rev-6 xs '()))
    ([xs ys] (if (null? xs) ys
                 (rev-6 (cdr xs) (cons (car xs) ys))))))
```

Сравнения за „равенство“ и действия със списъци

Сравнения за равенство

`=`	за числа
`char=?`	за литери
.....	
`eq?`	идентичност; работи за еднакви символи, но не непременно за числа и литери
`eqv?`	като `eq?` + за литери и за „почти всички“ числа
`equal?`	като `eqv?` + структурноиндуктивна еднаквост (бавно)

Някои свързани с „равенство“ действия със списъци

```
 ,---------------------- остатък от списък, от намерена ст-ст
 |        ,--------------- премахване от списък
 |        |       ,------- двойка от асоциативен списък
memq    remq    assq  ---- сравняване чрез eq?
memv    remv    assv  ---- сравняване чрез eqv?
member  remove  assoc ---- сравняване чрез equal?
memp    remp    assp  ---- сравняване чрез предикат
```

Асоциативни списъци

Асоциативен списък е списък от двойки. Първият член на всяка двойка се разглежда като ключ за достъп до втория: казваме, че вторият съответства на първия.

Основните действия с асоциативни списъци – образуване, добавяне и премахване на член и др. – се извършват както с всеки друг списък или двойка. За търсене в асоциативен списък служат функциите `assq`, `assv`, `assoc` и `assp`.

```
(define pairs '((a . P) (b . Q) (17 . R) ("om" . S)))
(cdr (assq 17 pairs))  ⟶  R
(assq 'Q pairs)        ⟶  #f
```

Контекстите, образувани чрез `lambda` и др. се реализират чрез асоциативни списъци.

Търсене, извличане и преобразуване на списъци

(`filter` f *списък*)
: Списък от членовете на дадения списък, за които f не е `#f`.

(`partition` f *списък*)
: Два списъка, отделни стойности – различните и равни на `#f` при f.

(`find` f *списък*)
: Първият член, за който f не е `#f`, а ако такъв няма – `#f`.

(`exists` f *списъци*)
: Първият резултат, който не е `#f`, а ако такъв няма – този от последното прилагане на f.

(`for-all` f *списъци*)
: `#f`, ако поне едно прилагане на f дава `#f`, а ако такова няма – резултата от последното прилагане на f.

(`list-sort` f *списък*)
: Устойчиво подреждане според f: два члена x и y разменят местата си ако и само ако x предхожда y в дадения списък и (f y x) не е `#f`.

Примери

```
(filter (lambda [x] (> x 10)) '(5 3 34 2 8 1 13 0 21 1))
    ⟶ (34 13 21)

(find (lambda [x] (> x 10)) '(5 3 34 2 8 1 13 0 21 1))
    ⟶ 34

(partition even? '(5 3 34 2 8 1 13 0 21 1))
    ⟶ (34 2 8 0)
      (5 3 1 13 21 1)

(list-sort (lambda [x y] (< (car x) (car y)))
           '((6 . 7) (2 . 9) (5 . 8) (2 . 4)))
    ⟶ ((2 . 9) (2 . 4) (5 . 8) (6 . 7))
```

Преобразуване на списъци

(`map` f *списъци*)

Почленно преобразуване. Прилагането на f не непременно спазва реда на членовете в списъка, но резултатите от него го спазват.

(`for-each` f *списъци*)

Почленно, последователно прилагане на f. Без стойност.

Примери

```
(map list '(a b c d))
```
$\longrightarrow$ `((a) (b) (c) (d))`

```
(map (lambda [x] (* x x)) '(3 -2 0 1))
```
$\longrightarrow$ `(9 4 0 1)`

```
(map + '(1 2 3 4 5) '(10 20 30 40 50))
```
$\longrightarrow$ `(11 22 33 44 55)`

```
(map list '(a b c d) '(+ - * /) '(A B C D))
```
$\longrightarrow$ `((a + A) (b - B) (c * C) (d / D))`

```
(for-each display '(2 0 1 6))
```
отпечатва 2016

Обхождане на списъци в общ вид

(`fold-left` f u *списъци*)

Обхождане отляво надясно с последователно прилагане на f – схема за един списък: $\underline{\underline{u} \diamond x_1} \diamond x_2 \diamond \ldots \diamond x_n$.

(С $\diamond$ е означено действието на f.)

Примерно определение за един списък:

```
(define [fold-lf f u xs]
  (if (null? xs) u
    (fold-lf f (f u (car xs)) (cdr xs))))
```

`fold-left` е финалнорекурентна.

(`fold-right` f u *списъци*)

Обхождане отдясно наляво с последователно прилагане на f – схема за един списък: $x_1 \diamond \ldots \diamond \underline{x_{n-1} \diamond \underline{x_n \diamond \underline{u}}}$.

Примерно определение за един списък:

```
(define [fold-rt f u xs]
  (if (null? xs) u
    (f (car xs) (fold-rt f u (cdr xs)))))
```

Примери: непосредствени приложения на `fold`

```
(fold-left + 0 '(3 5 8))          ⟶ 16  (сума)
(fold-left * 1 '(3 5 8))          ⟶ 120  (произведение)

(fold-left - 0 '(3 5 8))          ⟶ -16
(fold-right - 0 '(3 5 8))         ⟶ 6 (алтерниращa сума)

(fold-right cons '(x y) '(a b c)) ⟶ (a b c x y)
(fold-left cons '() '(a b c))     ⟶ ((((() . a) . b) . c)

(fold-left merge '() '((1 5) (2 5 8) (1) (3 4 5 6)))
              ⟶(1 1 2 3 4 5 5 5 6 8) (виж merge по-горе)
```

Примери: някои познати действия чрез fold

Намиране на дължина на списък:

```
(define [list-len a]
  (fold-left (lambda [n _] (+ 1 n)) 0 a))
```

Пресмятане на полином:

```
(define [poly cs x]
  (fold-left (lambda [v c] (+ (* v x) c)) 0 cs))
```

Обръщане на списък:

```
(define [rev-7 xs]
  (fold-left (lambda [rs x] (cons x rs)) '() xs))
```

Функция като map във вариант за само един списък:

```
(define [map1 f xs]
  (fold-right (lambda [x ys] (cons (f x) ys)) '() xs))
```

Пример: сливане на множество подредени списъци

Приложено към два списъка, следното определение работи както по-горе определената функция `merge` за сливане на подредени списъци. Приложено към списък от списъци, то ги слива чрез `fold` и свеждане към сливане на два списъка. Последното не е отделна функция, а част от основната и единствена.

```
(define merge
  (case-lambda
    ([a b]
      (cond
        ([null? a] b)
        ([null? b] a)
        (else (let ([x (car a)] [y (car b)])
                (if (<= x y)
                  (cons x (merge (cdr a) b))
                  (cons y (merge a (cdr b)))))))))
    (a (fold-left merge '() a))))
```

Множествени стойности

За някои действия е естествено да образуват в резултат повече от една или неопределен брой стойности, които при това не непременно образуват някаква структура. Например частното и остатъкът при целочислено делене са две стойности, а корените на квадратно уравнение са един или два.

Образуването на множество стойности става с функцията

(values v_1 v_2 ...)

Функцията

(call-with-values f g)

повиква f без аргументи и с образуваните от нея стойности, всяка като отделен аргумент, повиква g. Така всяка функция може да бъде направена потребител на множествени стойности, произведени от коя да е друга функция.

Примери

```
(values)                                                    ⟶

(values 17)                                                 ⟶ 17

(values (+ 3 5) (* 2 3 4) (sqrt -1))                        ⟶ 8
                                                              24
                                                              0+1i

(call-with-values (lambda [] (values map 'f 2))
                  (lambda x (length x)))                    ⟶ 3

(define [roots a b c]
  (let ([d (sqrt (- (* b b) (* 4 a c)))] [a2 (* 2 a)])
    (if (= 0 d)
      (- (/ b a2))
      (values (/ (- (+ b d)) a2) (/ (- d b) a2)))))

(roots 1 -3 2)                                              ⟶ 1
                                                              2
(roots 1 6 9)                                               ⟶ -3
```

Косвени пресмятания

Функцията

(apply f v_1 v_2 ... v)

повиква функция f с аргументи v_1, v_2, ... Последният от тях, v, трябва да е списък и членовете му се превръщат всеки в отделен аргумент. Така `apply`, от една страна, е функцията *пресмятане на функция*, а от друга – предоставя начин за превръщане на списък в отделни стойности – аргументи.

Друга форма на косвено пресмятане дава функцията

(eval v)

Тя пресмята израза, който е резултат от пресмятането на аргумента ѝ v. Така части от програмата могат да се образуват от самата нея.

Примери за прилагане на apply

```
(apply + '(3 8 -4 6)) ⟶ 13

(apply map list '((a b c d) (+ - * /) (A B C D)))
                ⟶ ((a + A) (b - B) (c * C) (d / D))

(apply max '(5 3 34 2 8 1 13 0 21 1)) ⟶ 34

(apply < (list-sort < '(5 3 34 2 8 1 13 0 21))) ⟶ #t
```

Реализация на map в общия случай – за един или повече списъци:

```
(define [mapn f . xss]
  (apply fold-left
         (lambda [ys . xs] (cons (apply f xs) ys))
         '()
         (fold-left
           (lambda [yss xs] (cons (reverse xs) yss))
           '()
           (reverse xss))))
```

Примери за прилагане на eval

```
(define x 3)
(define y 5)
(define a 'x)
(eval a)                    ⟶ 3 (косвено цитиране на x)
(define a 'y)
(* 10 (eval a))             ⟶ 50 (косвено цитиране на y)
(eval `(set! ,a 10))        равнозначно на (set! y 10)
(list x y)                  ⟶ (3 10)

(define x '*)
(procedure? x)                                  ⟶ #f
(symbol? x)                                     ⟶ #t
(procedure? (eval x))                           ⟶ #t
(map symbol? '(sin cos))                        ⟶ (#t #t)
(map procedure? (map eval '(sin cos)))          ⟶ (#t #t)
(equal? (map eval '(sin cos)) `(,sin ,cos))     ⟶ (#t #t)
```

Примерите са в глобалния контекст. `eval` не позволява косвено цитиране на променливи от локален контекст (`lambda`, `let`).

Програмиране с функции от висок ред

„Функции от висок ред“ се наричат тези, които имат функции за свои параметри, резултати или и едното, и другото. Редица предопределени в езика Scheme функции – `find`, `filter`, `map`, `fold-left`/`-right` и др. имат аргумент функция.

Същите функции имат и друго важно качество: те въплъщават действия с цялостни сложни стойности (списъци) и така представляват готови решения на различни основни, характерни задачи, възникващи при работа със сложни данни. Третирането на сложните данни като единни стойности позволява на тях да се гледа абстрактно, без да се обръща внимание на подробностите по устройството им.

Ако на свой ред решенията на по-сложните задачи се свеждат до прилагане на готовите решения за подзадачи, необходимостта от рекурентни определения (и изобщо всякакви форми на повторност) изчезва или силно намалява, а новоопределените функции, както тези, от които са съставени, третират съответните структури от данни като цялостни, единни стойности.

Образуване на функции от висок ред

За ефикасно прилагане на описания стил на програмиране е важно да има разнообразни средства за образуване на функции от вече налични.

Сред предопределените в Scheme функции няма такива, които да произвеждат функции. Единственото предвидено средство за образуване на функционни стойности са λ-изразите. Не е трудно обаче функции – преобразуватели на функции да бъдат съставени самостоятелно.

Особено плодотворни начини за пораждане на функции от функции са:

- частично прилагане (проекция, специализиране) на функция;
- разлагане по аргумент (ешелониране) на функция – прави удобно частичното прилагане;
- различни форми на композиране.

Например функцията `map` по начало бива повиквана с поне два аргумента – функция и списък. Чрез разлагане тя може да бъде превърната във функция само от първия си параметър, произвеждаща функция (от списъци). Това на свой ред дава възможност за образуване на именно такива функции – частични прилагания на `map`.

Функции-преобразуватели – 1

Функция-константа – повикване с каквито и да е аргументи дава `x`:

```
(define [const x]
  (lambda _ x))
```

Размяна на първите два аргумента на `f`:

```
(define [flip f]
  (lambda [x y . xs] (apply f y x xs)))
```

Ешелониране (разлагане по аргументи) на функция:

```
(define [echelon f]
  (lambda [x] (lambda xs (apply f x xs))))
```

Частично прилагане на `f` към `x` като първи аргумент:

```
(define [bind1 f x]
  ((echelon f) x))
```

Частично прилагане на `f` към `x` като втори аргумент:

```
(define [bind2 f x]
  (bind1 (flip f) x))
```

Функции-преобразуватели – 2

Композиция на редица от функции:

```
(define [compose . fs]
  (define [comp2 f g]
    (lambda x (call-with-values
                (lambda [] (apply g x))
                f)))
  (fold-left comp2 values fs))
```

Съединяване на списъци, резултат от `map`:

```
(define [append-map f xs]
  (apply append (map f xs)))
```

Съединяване на множествени резултати от функции:

```
(define [append-values . fs]
  (apply values
    (append-map (lambda [f] (call-with-values f list))
                fs)))
```

Примери

```
((const 5))                          ⟶ 5

((const 5) 100 200)                  ⟶ 5

(let ([f (flip /)])
  (values (/ 2 5) (f 2 5)))          ⟶ 2/5
                                       5/2

(bind1 expt 2)                       ⟶ функцията 2^x

(bind2 expt 2)                       ⟶ функцията x^2

((bind1 expt 2) 10)                  ⟶ 1024

((bind2 expt 2) 10)                  ⟶ 100

(let ([f (bind1 expt 2)])
  (map f '(0 1 3 5 6)))              ⟶ (1 2 8 32 64)

(let ([f (bind2 expt 2)])
  (map f '(0 1 3 5 6)))              ⟶ (0 1 9 25 36)
```

Примери

```
(let ([f (compose (bind1 = 1) abs (bind2 mod0 5))])
   (filter f (seq -8 8)))          ⟶ (-6 -4 -1 1 4 6)

(map vector->list '(#(a b) #(c) #(d e f)))
                                   ⟶ ((a b) (c) (d e f))
(append-map vector->list '(#(a b) #(c) #(d e f)))
                                   ⟶ (a b c d e f)

(append-values (lambda [] (values 'a 'b))
               (lambda [] (values 99))
               (lambda [] (values 'x 'y 'z))) ⟶ a
                                                b
                                                99
                                                x
                                                y
                                                z
```

Действия със списъци като множества – 1

Премахване на повторения на последователни членове в списък. Ако списъкът е подреден, това е образуване на наредено множество от членовете му:

```
(define [uniq xs]
  (if (> 2 (length xs))
    xs
    (let ([x (car xs)] [y (cadr xs)])
      (if (equal? x y)
        (uniq (cons x (cddr xs)))
        (cons x (uniq (cdr xs)))))))

(uniq '(M i s s i s s s s i p p i)) ⟶ (M i s i s i p i)
```

Образуване на множество – списък от членовете на списък без повторения и изброени според реда на първото им срещане:

```
(define [nub xs]
  (if (null? xs) '()
    (let ([x (car xs)])
      (cons x (nub (remove x (cdr xs)))))))

(nub '(a b r a c a d a b r a)) ⟶ (a b r c d)
```

Действия със списъци като множества – 2

Разлика, обединение, сечение и симетрична разлика на списъци-множества:

```
(define [set-difference a b]
  (fold-right remove a b))

(define [set-union a b]
  (append a (set-difference b a)))

(define [set-intersection a b]
  (filter (bind2 member b) a))

(define [set-symmetric-difference a b]
  (append (set-difference a b) (set-difference b a)))
```

Действия със списъци като множества – 3

Симетрична разлика без `set-difference` (или `fold-right`):

```
(define set-symmetric-difference
  (case-lambda
    ([as bs]
      (set-symmetric-difference as bs '()))
    ([as bs rs]
      (if (null? as) (append (reverse rs) bs)
        (let* ([a (car as)] [as (remove a (cdr as))])
          (if (member a bs)
            (set-symmetric-difference as (remove a bs) rs)
            (set-symmetric-difference as bs (cons a rs))))))))

(set-symmetric-difference '(a b c b a d) '(c a e))
                                                ⟶ (b d e)
```

Действия със списъци като множества – 4

Разпределяне на членовете на списък по групи (списък от списъци) според признак за еквивалентност. В групите членовете запазват относителния ред помежду си:

```
(define [group p as]
  (define [add-element zss x]
    (if (null? zss)
      `((,x))
      (let* ([yss (cdr zss)] [ys (car zss)] [y (car ys)])
        (if (p x y)
          (cons (cons x ys) yss)
          (cons ys (add-element yss x))))))
  (map reverse (fold-left add-element '() as)))

(group (lambda [m n] (= (modulo m 5) (modulo n 5)))
       '(5 3 34 2 8 1 13 0 21 1))
                  ⟶ ((5 0) (3 8 13) (34) (2) (1 21 1))
```

Действия със списъци като множества – 5

Декартово произведение на наредени множества

```
(define [cartesian-product ass]
  (fold-right
    (lambda [as bss]
      (append-map
        (lambda [a] (map (lambda [bs] (cons a bs)) bss))
        as))
    '(())
    ass))
```

```
(cartesian-product '())                    ⟶ (())
(cartesian-product '(()))                  ⟶ ()
(cartesian-product '((a b) (:) (x y z))) ⟶
    ((a : x) (a : y) (a : z) (b : x) (b : y) (b : z))
```

Декартово произведение: шедьовър на ФП от 1960-те

Преди повече от половин век Christopher S. Strachey изобретява* функцията, която тук наричаме `fold-right` и дава следния пример за използването ѝ, записан на създадения също от Strachey ЕП CPL (по това време Scheme не съществува, а Lisp не разполага с такива средства). Решението не използва други функции, освен трикратно `fold-right` (и `cons`).

```
(define [cartesian-product ass]
  (define [f as bss]
    (define [g a css]
      (define [h bs dss]
        (cons (cons a bs) dss))
      (fold-right h css bss))
    (fold-right g '() as))
  (fold-right f '(()) ass))
```

---

* D. W. Baron, C. Strachey. *Programming.* In L. Fox, ed. *Advances in programming and non-numerical computation*, pp. 49-82. Pergammon Press, 1966

Двоични дървета

По определение двоично дърво (ДД) или се състои от възел (корен) и ляв и десен наследници – двоични дървета, или е празно. (Празното ДД не съдържа нищо.)

За представяне на ДД като даннова структура да приемем, че:

- всеки възел съдържа (или е) някаква, която и да е възможна, стойност;
- празното ДД представяме с `()`;
- ДД с възел `x`, на което двата наследника не съществуват, т. е. са празни ДД, представяме с `(x)`;
- всяко друго ДД представяме като двойка с първи член възела и втори член – двойка от левия и десния наследници.

Пример

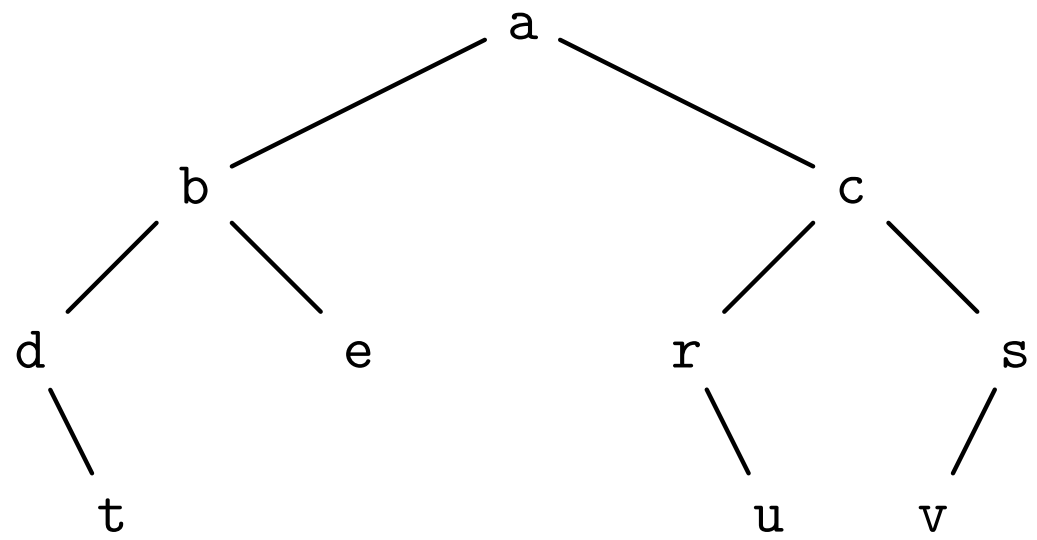
Според приетото представяне на ДД, стойността

```
(a . ((b . ((d . (() . (t))) . (e))) .
      (c . ((r . (() . (u))) . (s . ((v) . ()))))))
```

или в обичайния по-къс запис

```
(a (b (d () t) e) c (r () u) s (v))
```

отговаря на дървото от фигурата:



Обхождане на двоични дървета – 1

Обхождане на ДД означава построяване на списък от всички възли в него и в неговите поддървета или на някои характерни възли.

Обхождане в ред корен-ляво-дясно (префиксен):

```
(define [bt-preorder t]
  (if (or (null? t) (null? (cdr t)))
    t
    (cons (car t) (append (bt-preorder (cadr t))
                          (bt-preorder (cddr t))))))
```

Обхождане в ред ляво-дясно-корен (суфиксен):

```
(define [bt-postorder t]
  (if (or (null? t) (null? (cdr t)))
    t
    (append (bt-postorder (cadr t))
            (bt-postorder (cddr t))
            (list (car t)))))
```

Обхождане на двоични дървета – 2

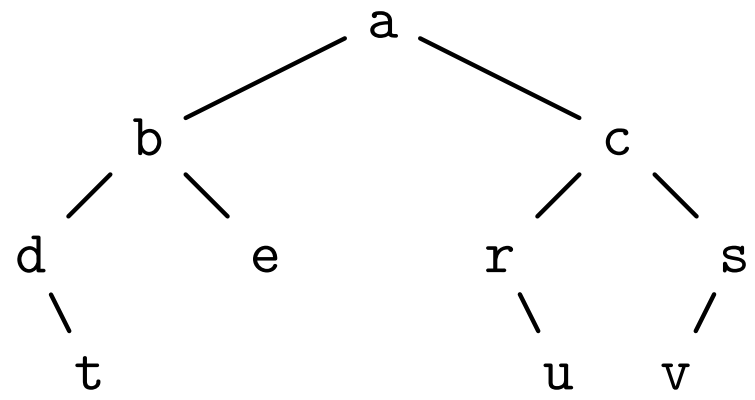
Обхождане в ред ляво-корен-дясно (инфиксен):

```
(define [bt-inorder t]
  (if (or (null? t) (null? (cdr t)))
    t
    (let ([x (car t)] [t1 (cadr t)] [t2 (cddr t)])
      (append (bt-inorder t1)
              (cons x (bt-inorder t2))))))
```

Обхождане на листата:

```
(define [bt-leaves t]
  (if (or (null? t) (null? (cdr t)))
    t
    (append (bt-leaves (cadr t)) (bt-leaves (cddr t)))))
```

Примери




```
(define bt '(a (b (d () t) e) c (r () u) s (v)))
```

```
(bt-preorder bt))      ⟶ (a b d t e c r u s v)
(bt-postorder bt))     ⟶ (t d e b u r v s c a)
(bt-inorder bt))       ⟶ (d t b e a r u c v s)
(bt-leaves bt))        ⟶ (t e u v)
```

Двоични подреждащи дървета

Двоично подреждащо дърво (ДПД) е двоично дърво с наследници ДПД. Стойността на корена на левия наследник не превъзхожда тази на корена на дървото, а последната не превъзхожда стойността на корена на десния наследник (когато наследниците не са празни). Стойностите са сравними по големина, например числа, или предшестване, например низове в лексикографски смисъл.

Следната процедура добавя число към числово ДПД, образувайки ново ДПД:

```
(define [bst-insert t x]
  (if (null? t)
    (list x)
    (let* ([y (car t)] [c (<= x y)] [t1 (cdr t)])
      (if (null? t1)
        (if c `(,y (,x)) `(,y () ,x))
        (let ([t1 (car t1)] [t2 (cdr t1)])
          (if c (cons* y (bst-insert t1 x) t2)
                (cons* y t1 (bst-insert t2 x)))))))))
```

Подреждане чрез ДПД

Инфиксното обхождане на ДПД дава нареден по големина списък.

От друга страна, от кой да е списък (от сравними стойности) може да се образува ДПД с помощта на следната функция:

```
(define [list->bst xs] (fold-left bst-insert '() xs))
```

```
(list->bst '(3 5 4 1 2))        ⟶ (3 (1 () 2) 5 (4))
```

Композицията на двете функции дава подреден списък, например

```
(bt-inorder (list->bst '(3 5 4 1 2))) ⟶ (1 2 3 4 5)
```

Допълнение и въпрос

Представяне на ДД е възможно и по други начини. Един от тях е празното ДД да бъде стойността `#f`, а непразното – двойка с членове стойност-възел и двойка от наследниците. В този случай някои действия с ДД добиват малко по-прост запис. Например добавянето на число в ДПД е

```
(define [bst-insert t x]
  (if t
    (let ([y (car t)] [t1 (cadr t)] [t2 (cddr t)])
      (if (<= x y)
        (cons* y (bst-insert t1 x) t2)
        (cons* y t1 (bst-insert t2 x))))
    `(,x #f . #f)))
```

Какви са сравнителните предимства и недостатъци на използваното по-горе и на това представяне?

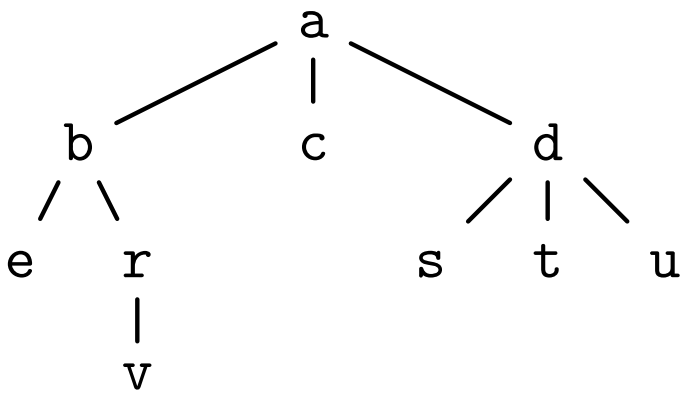
Дървета от общ вид

Дърво (от общ вид) или се състои от възел (корен) и известен (възможно нулев) брой наследници – дървета, или е празно.

(За разлика от двоичните дървета, наследниците на тези от общ вид, дори само един или два, не се отличават като ляв и десен.)

За представяне на дърво като даннова структура използваме списък. Непразен списък отговаря на дърво с корен първия член на списъка и наследници останалите членове – те трябва да са списъци. Следните стойност и фигура представят едно и също дърво:

```
(a (b (e) (r (v))) (c) (d (s) (t) (u)))
```



Обхождане на дървета – 1

Обхожданията са подобни на тези при двоични дървета, но инфиксното обхождане е неприложимо за общи дървета.

Префиксно обхождане:

```
(define [tree-preorder t]
  (if (null? t) '()
    (cons (car t) (append-map tree-preorder (cdr t)))))
```

Суфиксно обхождане:

```
(define [tree-postorder t]
  (if (null? t) '()
    (fold-right append (list (car t))
                (map tree-postorder (cdr t)))))
```

Обхождане на листата:

```
(define [tree-leaves t]
  (if (null? t) '()
    (let ([h (cdr t)])
      (if (null? h) t (append-map tree-leaves h)))))
```

Обхождане на дървета – 2

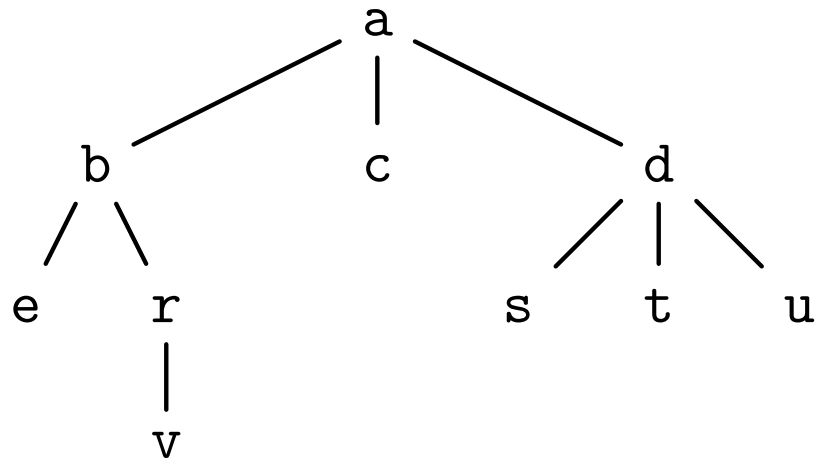
Послойно обхождане – образува се не списък от възлите, а списък от списъци от възлите по слоеве:

```
(define [tree-levelorder t]
  (define [f q]
    (if (null? q) '()
      (cons (map car q) (f (append-map cdr q)))))
  (if (null? t) '() (f (list t))))
```

Списък от пътищата от корена на дърво до всяко от листата:

```
(define [tree-paths t]
  (cond
    ([null? t] t)
    ([null? (cdr t)] (list t))
    (else (map (bind1 cons (car t))
               (append-map tree-paths (cdr t))))))
```

Примери




```
(define tree '(a (b (e) (r (v))) (c) (d (s) (t) (u))))
```

```
(tree-preorder tree)    ⟶ (a b e r v c d s t u)
(tree-postorder tree)   ⟶ (e v r b c s t u d a)
(tree-leaves tree)      ⟶ (e v c s t u)
(tree-levelorder tree)  ⟶ ((a) (b c d) (e r s t u) (v))
(tree-paths tree)
     ⟶ ((a b e) (a b r v) (a c) (a d s) (a d t) (a d u))
```

Задачи

Представянето на двоични дървета по-горе е определено чрез двойки. Обосновете твърдението, че съгласно избраното представяне всяко дърво се задава чрез (истински) списък.

Представянията и на двоични, и на общи дървета са списъци. Обосновете защо е възможно всяка стойност на възел да бъде произволно избрана, включително също списък, празен или не, истински или неистински.

Префиксното обхождане и в двата разгледани конкретни примера на дървета, двоично и общо, може да се получи и чрез функцията `flatten`. За всички дървета ли е вярно това? Обосновете.

Асоциативни таблици

Асоциативните таблици се използват за същото, за което и асоциативните списъци, но работят много по-бързо от тях при голям брой съхранявани връзки. Те са реализирана чрез набор от библиотечни функции абстракция. Ето тази част от функциите, която се използва по-нататък:

(`make-hashtable` h p *размер*)
: Създава асоциативна таблица. Функцията h (в нашия случай – `equal-hash`) по кой да е ключ образува цяло неотрицателно число – хешстойност за ключа, а сравняването на ключове става чрез функцията p. Посочването на размер не е задължително.

(`hashtable-contains?` *таблица ключ*)
: Дава булев резултат – съдържа ли таблицата асоциативна връзка с дадения ключ.

(`hashtable-ref` *таблица ключ стойност-по-подразбиране*)
: Дава наличната стойност или тази по подразбиране.

(`hashtable-set!` *таблица ключ стойност*)
: Променя стара връзка или създава нова.

Излишни пресмятания при рекурсия

Различни числови редици и много други обекти биват естествено, просто и ефикасно задавани чрез рекурентни определения. Често така биват определяни и отнасящите се до тях функции. Понякога обаче това води до голям обем излишни пресмятания.

Например по определението на числата на Фибоначи

$$F_n = \begin{cases} n, & n = 0, 1 \\ F_{n-1} + F_{n-2}, & n = 2, 3, \ldots \end{cases}$$

непосредствено съставяме функцията

```
(define [fib n]
  (if (< n 2)
    n
    (+ (fib (- n 1)) (fib (- n 2)))))
```

рекурсивното пресмятане на която многократно намира едни и същи стойности на функцията.

Запомняне на стойности

Повторното пресмятане може да бъде избегнато, като съхраняваме в асоциативна таблица всяка намерена стойност на функцията заедно със съответния аргумент и за всеки аргумент проверяваме дали стойността е налице или следва да бъде пресмятана.

С това функцията `fib` добива вида

```
(define [fib n]
  (let ([cache (make-hashtable equal-hash equal?)])
    (let f ([n n])
      (if (hashtable-contains? cache n)
        (hashtable-ref cache n #f)
        (let ([res (if (< n 2) n
                       (+ (f (- n 1)) (f (- n 2))))])
          (hashtable-set! cache n res) res)))))
```

и се пресмята бързо за големи стойности на `n`, например

`(fib 200)` $\longrightarrow$ `280571172992510140037611932413038677189525`

Забележка

Конкретно числата на Фибоначи всъщност могат да се намират и не рекурентно, а чрез последователно определяне на членовете на редицата до достигане на нужния:

```
(define [fib n]
  (do ([i 0 j] [j 1 (+ i j)] [n n (- n 1)])
    ([= 0 n] i)))
```

или дори съвсем непосредствено по формулата $F_n = \left[\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}} \right]$:

```
(define [fib n]
  (let* ([rt5 (sqrt 5)] [phi (/ (+ 1 rt5) 2)])
    (exact (round (/ (expt phi n) rt5)))))
```

Описаният метод за усъвършенстване на рекурентни функции обаче е приложим и в много други случаи. Преобразуваните функции се наричат *мемофункции*, от англ. *memoize*.

Автоматизиране на привеждането

Привеждането на функция към мемовариант е еднообразно и е естествено да става автоматично. За целта най-напред снабдяваме функцията, която преобразуваме, с допълнителен аргумент – функция, повиквания към която заемат мястото на рекурентните повиквания на изходната функция. Например `fib` се превръща в

```
(define [fib n f]
  (if (< n 2)
    n
    (+ (f (- n 1)) (f (- n 2)))))
```

Пораждане на мемофункции

От функция като така изменената `fib` функцията-преобразувател `memoize` построява мемофункция. Мемофункцията образува и използва асоциативна таблица за съхраняване на стойностите на изходната функция. Тя повиква тази функция, предавайки ѝ като втори аргумент себе си.

```
(define [memoize f]
  (define fm
    (let ([cache (make-hashtable equal-hash equal?)])
      (lambda [x]
        (if (hashtable-contains? cache x)
          (hashtable-ref cache x #f)
          (let ([res (f x fm)])
            (hashtable-set! cache x res)
            res)))))
  fm)
```

Например

```
(define fibonacci (memoize fib))
(fibonacci 200) ⟶ 280571172992510140037611932413038677189525
```

Друг пример: функцията на Акерман

Функцията на Акерман е „тежкорекурентна“ (опитайте да пресметнете ръчно $A(2,2)$):

$$A(m,n) = \begin{cases} n+1, & m = 0 \\ A(m-1,1), & m > 0,\ n = 0 \\ A(m-1, A(m,n-1)), & m > 0,\ n > 0 \end{cases}$$

```
(define [ack a f]
  (let ([m (car a)] [n (cdr a)])
    (cond
      ([= m 0] (+ 1 n))
      ([= n 0] (f (cons (- m 1) 1)))
      (else (f (cons (- m 1) (f (cons m (- n 1)))))))))

(define ackerman (memoize ack))
(ackerman '(3 . 16))                    ⟶ 524285
```

Действия с числа, литери, низове и вектори. Преобразования

Виж съответните раздели в краткия справочник и стандарта на езика.

За подробности по отделните функции удобна отправна точка е азбучният показалец на книгата T.S.P.L.

Във връзка с числовите функции обърнете внимание на последователното

- различаване и придържане в пресмятанията към *точни* (цели и дробни, реални и комплексни) и *неточни* (с плаваща точка) числа; точните цели, както и числителите и знаменателите на точните дробни числа, са неограничено големи;
- отношение към комплексните числа: функциите `sin`, `log` и пр. са определени, както в мат. анализ, и за комплексни.

```
(exact 2+.5i)                          ⟶ 2+1/2i
(* 2-i 2+i)                            ⟶ 5
(sin +i)                               ⟶ 0.0+1.1752011936438014
(expt 2+i 3)                           ⟶ 2+11i
(imag-part (log -1))                   ⟶ 3.141592653589793
(= (imag-part (log -1)) (acos -1))     ⟶ #t
```

Изменяеми величини

Специалната форма

(`set!` *име израз*)

придава на цитираната по име променлива стойността на израза. Ако в дадения контекст променливата вече съществува, цитира се именно тя – нова не се създава.

Специалните форми

(`set-car!` *двойка израз*)

и

(`set-cdr!` *двойка израз*)

променят първия или втория елемент на двойката, така че той да цитира стойността на израза. Тези две форми дават възможност вече създадени структурни стойности да се изменят и така да се създават разнообразни изменяеми даннови структури.

По принцип изменяемостта на величините противоречи на функционалното програмиране, защото прави възможно даден израз при повторно пресмятане да образува различни стойности. Но изменяемостта не вреди, ако бъде съсредоточена в добре обособени части на програмата.

Изменяне на списъци – 1

Даден списък може да бъде променен, като изменим съдържанието на някоя от образуващите го двойки. Например изразите

```
(set-car! (list-tail a n) v)
(set-cdr! (list-tail a n) v)
```

променят съответно `n`-я член на списъка `a` и остатъка на `a` след `n`-я член.

Добавяне на стойност, в частност истински или неистински списък, към непразен такъв списък става с функцията

```
(define [append! a b]
  (do ([p a a] [a (cdr a) (cdr a)])
    ([null? a] (set-cdr! p b))))
```

Изменяне на списъци – 2

Чрез `append!` можем да образуваме цикличен списък от даден, като „закачим опашката му към тялото“ в една или друга точка.

```
(define x '(2 #\. 0 6 2 8 5))
(append! x (cddddr x))
```

образува цикличен списък `2 . 0 6 2 8 5 2 8 5 2 ...` в който частта `2 8 5` като че се повтаря безкрайно. Разбира се, цикличният списък не е истински списък в смисъла, в който го разбираме (никоя двойка в него не съдържа втори член `()`), а и безкрайното му съдържание е само условно: броят на съставящите го двойки не се променя от образуването на цикъл.

Истински списък може да се извлече от цикличен или от друг истински с функция, отброяваща посочен брой членове:

```
(define [take n xs]
  (if (or (= 0 n) (null? xs)) '()
    (cons (car xs) (take (- n 1) (cdr xs)))))
```

Например за `x` по-горе `(for-each display (take 15 x))` извежда `2.0628528528528`.

Изменяне на списъци – 3

За премахване на част от списък – k члена, броено от n-я или всички до края, ако k не е зададено – може да се използва функцията

```
(define [remove-from! a n . k]
  (let* ([b (list-tail a (- n 1))]
         [r (if (null? k) '()
                (list-tail b (+ 1 (car k))))])
    (set-cdr! b r)))
```

Тя е приложима само за $n>0$. Премахване на начален участък от списък a може да стане с

```
(set! a (list-tail a k))
```

Задачи и допълнение

Обосновете защо функцията `remove-from!` е неприложима за $n = 0$. Може ли тя да се преправи така, че да работи и в този случай?

Съставете функция, която вмъква нов член или подсписък на произволно място в посочен списък. Може ли функцията да обхване всички възможни случаи?

Понякога е нужно да се извършат няколко последователни пресмятания на изрази. Това има смисъл само ако изразите, освен евентуално последния, имат странични последствия: ако те просто образуват стойност, тя би останала неизползвана. Телата на много от формите като `lambda`, `let`, `when`, `cons` и `case` позволяват такова последователно пресмятане, но `if` – не: във всяка от частите ѝ се задава по точно един израз. В такива случаи може да се използва фо̀рмата

(`begin` *изрази*)

в която могат да се пресметнат колкото е нужно изрази. Заедно те формално образуват израз със стойност – тази на последния от тях, ако такава има.

Изменяема структура: опашка

Опашка е редица, на единия край на която може да се добавят членове, а на другия да се отнемат.

Може да се представи като двойка от указатели към списък – единият към началото му, а другият към последния член. (Другояче казано – двойка, единият член на която е самият списък, а другият – списъкът от неговия последен член).

Добавянето на член към опашката е добавяне на съответната стойност като нов последен член на списъка (използвайки втория член на двойката), а отнемането на член от опашката е отнемане от началото на списъка. По този начин и добавянето, и отнемането (прочитането) се изпълняват максимално ефективно – времето за изпълнение на действията е постоянно, не зависи от дължината на опашката. Ефективно реализираме и нескъсяващо опашката четене (прочитаният член не се отнема).

Действия с опашка – реализация

```
(define [queue-make] '(().()))

(define [queue-put s x]
  (let ([z `(,x)] [p (cdr s)])
    (if (null? p)
      (set-car! s z)
      (set-cdr! p z))
    (set-cdr! s z)))

(define [queue-peek s] (caar s))

(define [queue-get s]
  (let ([x (caar s)] [p (cdar s)])
    (set-car! s p)
    (when (null? p) (set-cdr! s '()))
    x))

(define [queue-empty? s] (null? (car s)))
```

Пример

```
(define q (queue-make))
(queue-empty? q)          ⟶ #t
(queue-put q 'I)
(queue-put q 'J)
(queue-put q 'K)
(queue-get q)             ⟶ I
(queue-empty? q)          ⟶ #f
(queue-put q 'L)
(queue-get q)             ⟶ J
(queue-get q)             ⟶ K
(queue-get q)             ⟶ L
(queue-empty? q)          ⟶ #t
(queue-put q 'M)
(queue-peek q)            ⟶ M
(queue-empty? q)          ⟶ #f
```

Упражнение

Реализирайте действията с опашка, като представите опашката с цикличен списък и само един указател към него. Както и по-горе, времето за изпълняване на всяко от действията трябва да е постоянно (независещо от дължината на опашката).

Изменяема структура: дек

Дек (англ. *deq* или *deque* от *double-ended queue*) е редица, на всеки край на която може да се добавя или отнема член. Може да се представи като двойка от списъци, началото на всеки от които отговаря на един от краищата на редицата, а за краищата на списъците се подразбира, че са съединени в редицата. Двата списъка са всъщност стекове със „съединени“ дъна.

По този начин всяко добавяне се изпълнява максимално ефективно. Четенето или извличането на член от кой да е край също става максимално бързо, стига съответният списък да е непразен. Ако е празен, но другият не е, преместваме от него в първия списък половината членове (взети в обратен ред). За разделянето на половини използваме по-горе определената функция `split-at`.

Действия с дек – реализация

```
(define [deque-make] '(().()))

(define [deque-put1 s x]
  (set-car! s (cons x (car s))))

(define [deque-put2 s x]
  (set-cdr! s (cons x (cdr s))))

(define [deque-peek1 s]
  (let ([p1 (car s)] [p2 (cdr s)])
    (if (null? p1)
      (let* ([p (split-at (div (length p2) 2) p2)]
             [p1 (car p)] [p2 (reverse (cdr p))])
        (set-car! s p2)
        (set-cdr! s p1)
        (car p2))
      (car p1))))
```

```
(define [deque-peek2 s]
  (let* ([p1 (car s)] [p2 (cdr s)]
         [s1 (cons p2 p1)] [x (deque-peek1 s1)])
    (set-car! s (cdr s1))
    (set-cdr! s (car s1))
    x))

(define [deque-get1 s]
  (let ([x (deque-peek1 s)])
    (set-car! s (cdar s))
    x))

(define [deque-get2 s]
  (let ([x (deque-peek2 s)])
    (set-cdr! s (cddr s))
    x))

(define [deque-empty? s] (equal? s '(().())))
```

Пример

```
(define d (deque-make))
(deque-empty? d)          ⟶ #t
(deque-put1 d 'I)
(deque-put1 d 'J)
(deque-put1 d 'K)
(deque-put2 d 'L)
(deque-get2 d)            ⟶ L
(deque-empty? d)          ⟶ #f
(deque-get2 d)            ⟶ I
(deque-get2 d)            ⟶ J
(deque-put2 d 'M)
(deque-get1 d)            ⟶ K
(deque-peek2 d)           ⟶ M
(deque-peek1 d)           ⟶ M
(deque-get2 d)            ⟶ M
(deque-empty? d)          ⟶ #t
```

Енергично и лениво пресмятане

При повечето действия, в частност повикванията на функции, аргументите биват пресмятани преди самите действия да започнат да се изпълняват – такова пресмятане се нарича *енергично*.

При други действия част от аргументите не се пресмятат. Това става в зависимост от едно или друго условие, както например при специалните форми `if`, `and` и `or`, чиито аргументи след първия се пресмятат само *при необходимост*. Такова пресмятане се нарича *лениво*.

Една от употребите на ленивостта е за предотвратяване на излишни или дори неправилни пресмятания, както в

```
(and (integer? x) (even? x))
```

където се проверява дали `x` е цяло число и едва след това и само ако наистина е цяло се проверява четността му. Ако второто условие се пресмяташе винаги, при `x` не цяло или изобщо не число изразът вместо резултат би пораждал авария.

Отложено пресмятане

Специалната форма

`(delay` *израз*`)`

не пресмята *израз*, а образува безаргументна безименна функция (λ-израз), която при повикване да го пресметне. Намерената при това стойност се запомня и при повторни повиквания на функцията пресмятането не се извършва, а направо се дава резултатът.

Както при всяка образувана с λ-израз функция, и тук се извлича контекстът на свързванията, наличен в точката на образуването.

В описания смисъл `delay` отлага пресмятането на аргумента си. С нейна помощ могат да се реализират всякакви лениви пресмятания, вкл. формите `if`, `cond`, `and` и пр. Отлагането на пресмятане чрез „опаковане“ във функция е основният механизъм на езика за реализиране на ленивост.

force

За пресмятане на получена от `delay` функция служи функцията `force`. Всъщност

(`force` *ф-израз*)

е равнозначно на

(*ф-израз*)

и служи за повикване на каквато и да е функция – нужно е само *ф-израз* да има функционна стойност.

Както `apply`, `force` въплъщава във вид на функция действието „повикване на функция“, но при `apply` функцията се повиква с аргументи, а при `force` – не.

Потоци

Освен за избягване на неправилни действия и за реализиране на разнообразни схеми с условен избор, ленивото пресмятане е удобен инструмент за образуване на (псевдо)безкрайни структури. Най-прост и заедно с това най-широко използван пример за такава структура е *потокът* – информатично понятие, което най-близко отговаря на понятието *безкрайна редица* в математиката. Числови и други безкрайни редици програмно се моделират чрез потоци.

Всеки поток се състои от начален член и остатък – поток. В това отношение потоците са като списъците и, както и списъците, се представят чрез двойки. По-конкретно, представяме поток чрез двойка от стойност – начален член на потока – и функция, пресмятането на която без аргументи има за резултат потока остатък. Така остатъкът на поток се образува лениво: за да образуваме поток всъщност правим това само за първия му член!

Примери

При образуване на поток най-често знаем как да образуваме остатъка и към съответния израз прилагаме `delay`.

Например

```
(define ones (cons 1 (delay ones)))
```

е поток от единици, а

```
(define [repeat x]
  (letrec ([s (cons x (delay s))]) s))
```

е функция, повикването на която образува поток с членове, равни на аргумента на функцията.

Забележка. Определението на `ones`, а по-нататък и други примери показват, че благодарение на отложеното пресмятане рекурентните (самоцитиращи се) определения имат смисъл не само за функционни, а и за други видове стойности.

Средства за работа с потоци – 1

За цитиране на член на поток по пореден номер и намиране на кой да е остатъчен поток съставяме функции, аналогични на списъчните `list-ref` и `list-tail`:

```
(define [stream-ref s n]
  (if (= n 0) (car s) (stream-ref ((cdr s)) (- n 1))))


(define [stream-tail s n]
  (if (= n 0) s (stream-tail ((cdr s)) (- n 1))))
```

За извличане на списък от посочен брой първи членове на поток:

```
(define [stream->list s n]
  (if (= n 0) '()
    (cons (car s) (stream->list ((cdr s)) (- n 1)))))
```

В много случаи за получаване на краен отрязък от някаква редица е по-удобно да образуваме списък не пряко, а като къс от цялата редица (поток).

Средства за работа с потоци – 2

Както при списъци, различни потоци могат да бъдат породени чрез почленно преобразуване на един или повече потоци, избор на членове по условие и други подобни действия. При потоци това включва и самообразуване на поток от негови части по рекурентна зависимост.

Следната функция е аналог на `fold-right` и прилага функция успоредно по протежение на един или повече потоци:

```
(define [fold-* f . ss]
  (let ([r (delay (apply fold-* f
                         (map force (map cdr ss))))])
    (apply f (append (map car ss) `(,r)))))
```

`fold-*` е определена чрез самоцитиране, но поради отложеното пресмятане на `r` не съдържа самоповикване. От функцията `f` зависи дали при пресмятането ѝ и това на `fold-*` последната се повиква вторично (с остатъците на потоците).

Обръщението `(map force ...)` показва ползата от това `force` да съществува като именувана функция.

Средства за работа с потоци – 3

Функциите `map-*` и `filter-*` са аналози на `map` и `filter` (вкл. `map-*` прилага функция успоредно върху членовете на един или повече потоци). Всяка от двете е повикване на `fold-*`.

Функцията `iterate-*` образува потока $x \;\; f(x) \;\; f^2(x) \;\; f^3(x) \;\; \ldots$

```
(define [map-* f . ss]
  (apply fold-*
         (lambda xs
           (let* ([xs (reverse xs)]
                  [r (car xs)] [xs (reverse (cdr xs))])
             (cons (apply f xs) r)))
         ss))

(define [filter-* p s]
  (fold-* (lambda [x ys] (if (p x) (cons x ys) (ys))) s))

(define [iterate-* f x]
  (letrec ([s (cons x (delay (map-* f s)))]) s))
```

Примери

```
(stream->list ones 5)                                  ⟶ (1 1 1 1 1)

(stream->list (repeat 'a) 5)                           ⟶ (a a a a a)

(define nats (iterate-* (bind1 + 1) 1))
(stream->list nats 15)
                    ⟶ (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)

(stream->list (iterate-* (bind1 * 2) 1) 13)
          ⟶ (1 2 4 8 16 32 64 128 256 512 1024 2048 4096)

(define rats                ; неотрицателните рационални числа
  (iterate-* (lambda [x] (/ (- (* 2 (floor x)) x -1))) 0))
(stream->list rats 15)
 ⟶ (0 1 1/2 2 1/3 3/2 2/3 3 1/4 4/3 3/5 5/2 2/5 5/3 3/4)
(stream->list (stream-tail rats 1001) 7)
              ⟶ (39/28 28/45 45/17 17/40 40/23 23/29 29/6)
```

Пример: поток от числата на Фибоначи

```
(define fibs
  (cons 0 (delay (cons 1 (delay
         (map-* + fibs ((cdr fibs))))))))

(stream->list fibs 15)
       ⟶ (0 1 1 2 3 5 8 13 21 34 55 89 144 233 377)

(stream-ref fibs 200)
       ⟶ 280571172992510140037611932413038677189525
```

Първите шест числа на Фибоначи, кратни на 7:

```
(stream->list
    (filter-* (lambda [x] (= 0 (mod x 7))) fibs) 6)
              ⟶ (0 21 987 46368 2178309 102334155)
```

Пример: поток от числата на Хаминг

Числата на Хаминг (*регулярни числа*) са числата с прости делители само 2, 3 и 5. Потокът `hams` се образува чрез безповторно сливане на три други потока – числата на Хаминг, кратни на 2, тези кратни на 3 и кратните на 5 – които на свой ред се образуват от `hams`:

```
(define [merge-* as bs]
  (let ([a (car as)] [b (car bs)])
    (if (<= a b)
      (cons a (delay (merge-* ((cdr as))
                             (if (= a b) ((cdr bs)) bs))))
      (cons b (delay (merge-* as ((cdr bs))))))))

(define hams (cons 1 (delay
    (merge-* (map-* (bind1 * 2) hams)
             (merge-* (map-* (bind1 * 3) hams)
                      (map-* (bind1 * 5) hams))))))

(stream->list hams 20)
  ⟶ (1 2 3 4 5 6 8 9 10 12 15 16 18 20 24 25 27 30 32 36)
```

Забележка

Понятието „поток“ може да се разглежда и така, че да включва и редици с крайна дължина. Отлагането на пресмятането на остатъка може да бъде полезно и за такива редици. Например, ако се образува твърде голяма редица, а се ползва само малка начална част или последователни малки части от нея, не е нужно да се образува цялата редица във вид на списък – бихме спестили памет, ако си служим с поток.

Допускането на крайни потоци изисква малки изменения във функциите за работа с потоци.

Упражнение

Образувайте поток от подредените по големина прости числа, използвайки следния метод.

Най-напред построяваме *поток от потоци*. Първият поток съдържа естествените числа от 2 нататък. Вторият поток се образува от първия, като премахнем всички кратни на първото му число: получаваме поток, започващ с 3. По същото правило образуваме и третия поток от втория – получавайки поток, започващ с 5 – и всеки следващ поток. Така полученият поток от потоци е

2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
3	5	7	9	11	13	15	17	19	21	23	25	27	29	...
5	7	11	13	17	19	23	25	29	31	35	37	41	43	...
7	11	13	17	19	23	29	31	37	41	43	47	49	53	...

...

и остава да образуваме поток от първите му членове: това са именно подредените по големина прости числа.

Как описаният метод може да се измени с цел ускоряване?

Потребителски типове (смеси)

Специалната форма

`(define-record-type` *тип клаузи*`)`

служи за образуване на нов даннов тип с име *тип* – смес от именувани полета, всяко от които може да носи някаква стойност. Основната клауза има вид

`(fields` *имена*`)`

и задава имената на членове (полета). Стойностите от дадения тип се създават с функцията `make-`*тип*, удовлетворяват предикатите `record?` и *тип*`?` и членовете им се цитират чрез функции с имена от вида *тип*`-`*поле*. Всички тези функции се пораждат автоматично с определението на типа. Аргументите на първата от тях отговарят на полетата, а останалите имат за аргумент стойност от създадения тип.

Всяко поле може да се опише като изменяемо – тогава за него се поражда и функция с име *тип*`-`*поле*`-set!`, обръщения към която сменят стойността на полето.

Всички клаузи освен `fields` са незадължителни. Другите посочват дали полетата на типа са видими, дали той може да бъде наследяван и дали самият той наследява друг тип.

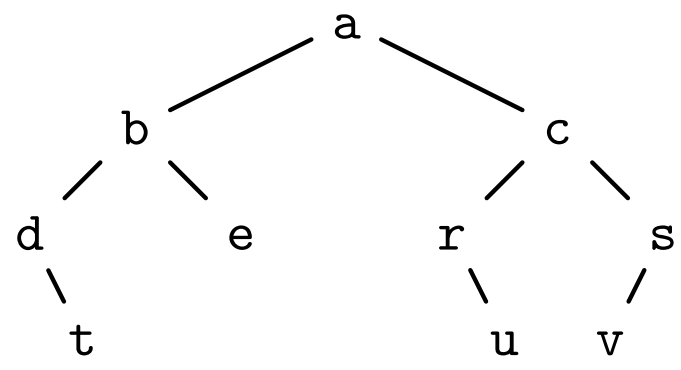
Пример: двоично дърво

Двоично дърво (виж по-горе) можем да представим като тип `btree` с полета `node` – възел и `left` и `right` – ляво и дясно поддървета (празното ДД е `()`):

```
(define-record-type btree (fields node left right))
```

Стойност от тип `btree`:

```
(define bt
  (make-btree 'a
    (make-btree 'b
      (make-btree 'd
        '()
        (make-btree 't '() '()))
      (make-btree 'e '() '()))
    (make-btree 'c
      (make-btree 'r
        '()
        (make-btree 'u '() '()))
      (make-btree 's
        (make-btree 'v '() '())
        '()))))
```




```
(record? bt) ⟶ #t
(btree? bt)  ⟶ #t
```

Функциите за работа с двоично дърво в такова представяне използват функциите доставчици `btree-node`, `btree-left` и `btree-right` за съответните полета. Например функция за обхождане на ДД в инфиксен ред е:

```
(define [bt-inorder t]
  (if (null? t) t
    (let ([n (btree-node t)])
      (if (and (null? (btree-left t))
               (null? (btree-right t)))
        `(,n)
        (append (bt-inorder (btree-left t))
                (cons n (bt-inorder (btree-right t))))))))
```

```
(bt-inorder bt)          ⟶ (d t b e a r u c v s)
```

Продължения

Продължение се нарича изпълнителният контекст в една или друга точка на програмата. Например продълженията на подчертаните с червено и синьо части на израза

```
(+ (* (+ r s) t) u v)
```

са действията с пораждaните от тези части стойности. Означавайки стойностите с □, действията се онагледяват съответно с

```
(+ (* (+ □ s) t) u v)
```

и

```
(+ (* □ t) u v)
```

Под функция-продължение разбираме функция, повиквана на мястото на цветните квадратчета – функция, която доставя съответните стойности в контекста на пресмятането. Такива функции обикновено са неявни в програмата, но могат да бъдат и изрично образувани и повиквани с *различни* аргументи. С това можем да променяме реда на следване на действията, участващите стойности или и двете.

call/cc

Създаването на функция-продължение (казваме и само *продължение*) става чрез повикване

`(call-with-current-continuation` f`)` или кратко `(call/cc` f`)`

където f е функция с един параметър. То води до повикване на f с аргумент функцията-продължение за дадената точка: `call/cc` образува и предава на f собственото си продължение.

Ако и когато бъде повикана, функцията-продължение *осъществява* продължението. Повикването може да стане повече от веднъж и откъдето и да е – често в контекст, различен от този, в който функцията е създадена.

Функцията-продължение приема такъв брой аргументи, какъвто брой стойности очаква продължението (обикновено 1, освен ако продължението е образувано с `call-with-values`) и ги предава като резултат(и) от изпълнението си – все едно че са резултати от обръщението към `call/cc`.

Примери

```
(call/cc (lambda [c] (c c)))
```
$\longrightarrow$ текущото продължение

```
(+ 5 (call/cc (lambda [c] (+ 100 (c 10)))))
```
$\longrightarrow$ 15

повикването на c изоставя собственото си продължение

```
(let ([x 5])
  (+ x (call/cc (lambda [c] (+ 100 (c (* x 3)))))))
```
$\longrightarrow$ 20

x се използва пряко и косвено

```
(define f)
(let ([x 5])
  (+ x (call/cc (lambda [c]
                  (set! f (lambda [n] (c (+ 2 (* x n)))))
                  (c 10)))))
```
$\longrightarrow$ 15 = (+ 5 10)

```
(f 4)
```
$\longrightarrow$ 27 = (+ 5 (+ 2 (* 5 4)))

```
(f 99)
```
$\longrightarrow$ 502 = (+ 5 (+ 2 (* 5 99)))

Съпрограми, генератори

Чрез продължения могат да се реализират различни конструкции с необичайно, т. е. различно от влагането на двойки повикване – връщане предаване на управлението. Една от тези конструкции са съпрограмите – симетрично (вместо йерархично) взаимодействащи функции.

Особен вид съпрограма е *генераторът*: многократно повиквана функция, която при всяко повикване дава стойност(и) от някаква редица, например членовете на дадена структура от данни в определен ред на изброяване. Тъй като генераторът трябва да „помни“ докъде е стигнало изброяването, всъщност повикванията след първото не са наистина повиквания, а възобновявания на единственото направено.

Пример: генератор за обхождане на двоично дърво

Нека дървото е стойност от дадения по-горе тип `btree`. Следната функция образува безаргументна функция-генератор, повикванията на която дават възлите на дърво `t` в инфиксен ред:

```
(define [generate-bt t]
  (letrec
    ([cont (lambda [] (bt-inorder t) (cont '()))]
     [resume (lambda v
               (call/cc
                 (lambda [c]
                   (let ([k cont]) (set! cont c)
                                   (if (null? v) (k) (k v))))))]
     [bt-inorder (lambda [t]
                   (when (not (null? t))
                     (bt-inorder (btree-left t))
                     (resume (btree-node t))
                     (bt-inorder (btree-right t))))])
    resume))
```

Главна роля за съпрограмността тук играе функцията `resume`. Тя съхранява текущото продължение в `cont` и осъществява (повиква) продължението, съхранявано там дотогава.

`resume` бива повиквана двояко. От една страна, тя е именно стойността, която произвежда `generate-bt` и в този смисъл олицетворява генератора: повторните повиквания (без аргумент) към `resume` дават изброяваните стойности. От друга страна, `resume` бива повиквана и в обратната посока – за предаване на стойността на възел от генератора към неговия потребител – в този случай повикването е с аргумент възела и `resume` го пре-предава на повикваното от нея продължение.

`cont` е хранилище за осъществяваното продължение при всеки от двата вида повикване на `resume`. Първоначално в `cont` се поставя функция, която повиква `bt-inorder`, а когато тя завърши – това, което в момента съхранява `cont`, а именно последното продължение на потребителя на генератора.

Собствено изброяването на възлите става от рекурентната, с очеочевидно съдържание функция `bt-inorder`, а `resume` е напълно независима от изброяването и може да се използва без промени за който и да е генератор. С тази цел препредаваната от `resume` към потребителя на генератора стойност е списъчна: така може да се предават множествени стойности, а празният списък е признак за липса на стойност – изчерпване на генератора.

За използване на този или друг генератор може да се прилага следната функция:

```
(define [for gen use]
  (do ([v (gen) (gen)]) ([null? v])
    (use (car v))))
```

параметри на която са генераторът и функция, на която се подават получаваните от него стойности. Например

```
(for (generate-bt bt) (lambda [v] (write v) (display #\ )))
```

обхожда възлите на двоично дърво от тип `btree`, отпечатвайки всяка стойност.

Задачи

Функцията `generate-bt` обединява конкретната изброяваща функция `bt-inorder` с универсалния „генераторен интерфейс“, състоящ се от `resume` и `cont`. Подобрете реализацията, като определите `bt-inorder` като самостоятелна функция и съставите функция `(make-generator g d)`, която за каква да е даннова структура `d` (както двоичното дърво в примера) и изброяваща тази структура функция `g` (както например `bt-inorder`) образува генератор, работещ по същия начин както резултата от `generate-bt` по-горе.

Отложените пресмятания и продължженията са различни механизми, с които могат да се решават някои сходни задачи, например изброяване на членовете на редици с постепенно пораждане. Посочете предимствата на всяко от двете средства пред другото.

Извънредни ситуации и събития

Извънредна ситуация (И) възниква при опит да се извърши действие над неподходящ аргумент, например `(/ x 0)` или `(car '())`, или чрез нарочно, явно действие. При възникване на И текущото действие се изоставя и се стартира функция – *обработчик* на И. По начало има само един обработчик, а изпълнението му се състои в извеждане на подходящо съобщение и завършване на програмата. Обработчици могат да бъдат назначавани по програмен път и различни за различните части на програмата.

Когато И възниква неявно, тя е представена от стойност от особен тип – *събитие*. Явно предизвиканите И се представят също чрез събития или чрез какви да е други стойности.

Съществува голям брой предопределени, именувани събития. Някои от тях са фактически множества, състоящи се от „основни“ събития. Нови видове събития могат да бъдат въвеждани в самата програма.

Предизвикване на събития

За явно предизвикване на И са предназначени основно функциите

`(raise` v`)`

и

`(raise-continuable` v`)`

всяка от които предава стойността v – събитие или друго – на обработчика. `raise` създава „поправима“, а `raise-continuable` – „непоправима“ И. В първия случай е възможно да се достигне продължението на обработчика, т. е. обработчикът се пресмята по обичайния начин. Във втория случай пресмятането на обработчика завършва също с И.

Съществуват и други функции за пораждане на И. Например

`(error` *източник текст*`)`

поражда И, като посочва къде възниква (при невъзможност за това – `#f`) и извеждано съобщение *текст*.

`(assert` *израз*`)`

дава стойността на *израз*, ако тя не е `#f`, в противен случай предизвиква непоправима И.

Пресмятане под надзор

`(with-exception-handler` h f`)`

повиква без аргументи функцията f. Ако тя породи И, повиква се функцията обработчик h с аргумент представящата И стойност (например събитие). Ако при това h достига продължението, тя евентуално му предава стойност-резултат (което при нормален завършек прави f).

`(guard (`*име клаузи*`)` *изрази*`)`

пресмята *изрази*. Ако се породи И, съответното събитие става стойност на *име* и последователно се пресмятат *клаузи* като в `cond`: неявно е образувана функция-обработчик с параметър *име* и тяло `cond`. Ако никоя клауза не сработва, възниква наново същата И.

`(dynamic-wind` g f h`)`

Трите функции f, g и h се повикват без аргументи. Пресмята се основно f, но преди това, както и преди всяко евентуално вторично попадане в f посредством осъществяване на продължение, се пресмята g. Освен това, всеки път щом f завърши (вкл. и вторично, след съответно продължение) се пресмята h. Това става дори ако f завърши с И. Така се осигуряват съпровождащи f действия, които не бива да бъдат пропуснати: освобождаване на зает от f ресурс и др. под.

Примери

```
(define [f n]
  (guard
    (x ([assertion-violation? x] 'badarg)
       ([< x 1]    'zero)
       ([< x 10]   'ones)
       ([< x 100]  'tens)
       ([< x 1000] 'hundreds)
       (else       'huge))
    (assert (integer? n))
    (assert (exact? n))
    (assert (<= 0 n))
    (raise n)))

(f "231")    ⟶ 'badarg
(f 37.)      ⟶ 'badarg
(f 37)       ⟶ 'tens
(f -45)      ⟶ 'badarg
(f 1001)     ⟶ 'huge
```

Примери

И в трите израза, ако стойността на `z` не е `#f`, се дава като резултат тя.

Ако `z` е `#f`, `x` в обработчика на И получава стойността `ha!`. В първия израз стойността се и дава като резултат от `with-exception-handler`. Във втория израз опитът да се направи това поражда нова И от обработчика. В третия израз обработчикът осъществява продължението на `call/cc`, а не на `with-exception-handler`, затова изразът, както първия, произвежда стойността `ha!`, въпреки че И се поражда с `raise`.

```
(with-exception-handler
  (lambda [x] (display x) x)
  (lambda [] (or z (raise-continuable 'ha!))))

(with-exception-handler
  (lambda [x] (display x) x)
  (lambda [] (or z (raise 'ha!))))

(call/cc
  (lambda [c]
    (with-exception-handler
      (lambda [x] (c x))
      (lambda [] (or z (raise 'ha!))))))
```

Примери

При следното определение

```
(define [bound? z]
  (call/cc
    (lambda [c]
      (with-exception-handler
        (lambda [_] (c #f))
        (lambda [] (values #t (z)))))))
```

изразът

```
(bound? (delay име))
```

дава две стойности – `#t` и тази на *име*, ако свързване за последното има. Ако за *име* няма свързване в текущия контекст, цитирането му поражда извънредна ситуация, която се потиска чрез обработчика и като резултат се дава `#f`.

Примерът съчетава прилагане на отложено пресмятане, продължения и извънредни ситуации.

Допълнение

В последния пример аргументът на `bound?` не може да се дава непосредствено, а непременно чрез `delay`. С познатите средства няма как другояче да се потисне преждевременният опит да се извлече стойността на името и съответно възникването на И, която не се прехваща от обработчик. Това може да се направи с помощта на специалната форма `define-syntax`, както по-долу. Така създаваме макроопределение, което на свой ред също е специална форма. По същия начин се определят `delay` и редица други.

```
(define-syntax bound?
  (syntax-rules ()
    [(_ z)
     (call/cc
       (lambda [c]
         (with-exception-handler
           (lambda [_] (c #f))
           (lambda [] (values #t z))))))]))
```

Входно-изходни действия

Scheme е снабден с разнообразни функции за въвеждане и извеждане на информация от и във файлове или т. нар. стандартни вход и изход („терминал“, „конзола“).

Съществуват функции за четене например на отделна литера (`read-char`) или октет (`get-u8`), ред (`get-line`), част от ред или няколко последователни реда (`get-string-n`) и на цял файл (`get-string-all`), както и за извеждане на литера, октет и пр.

`write` и `display` извеждат стойност на стандартния изход – като текст, но по различен начин:

```
(write '(#\a "bcd" ef))    ⟶ (#\a "bcd" ef)
(display '(#\a "bcd" ef)) ⟶ (a bcd ef)
(display '(a "bcd ef"))    ⟶ (a bcd ef)
(display '(a bcd ef))      ⟶ (a bcd ef)
```

`(newline)` е равнозначно на `(write-char #\linefeed)`.

Функцията `read` чете и възприема текст както в програма на Scheme – давайки число, символ, списък или др.

Пример

Следното е програма, отразяваща ред по ред текста от стандартния вход на стандартния изход. Функцията `read-line` чете литери една по една до край на ред и ги натрупва в списък, който накрая превръща в низ, а ако входът е изчерпан, дава `#f`.

```
(define [read-line . cs]
  (unless (null? cs) (set! cs (car cs)))
  (let ([c (read-char)])
    (cond ([eof-object? c] #f)
          ([char=? c #\newline] (list->string (reverse cs)))
          (else (read-line (cons c cs))))))

(define [echo]
  (let ([s (read-line)])
    (when s (display s) (newline) (echo))))

(echo)
```

По-сложна програма с подобна структура е калкулаторът в ОПЗ