Записки по функционално програмиране. Езикът Scheme

# ФП – въведение

Двата най-основни стила на програмиране (програмни парадигми) са императивния и декларативния (imperative, declarative). В императивния стил обикновено програмистът точно определя стъпките, които трябва да извърши програмата, на пример: създай променлива, събери две променливи, върти цикъл от 0 до N, задели памет за масив. Общо взето „заповядваме“ на компютъра какво точно да прави. Процедурният стил и ООП са императивни парадигми. При декларативния стил програмистът описва свойствата на желания резултат. Логическото програмиране и функционалното програмиране са подстилове на декларативния. На пример искаме да сметнем НОД-а на числата a и b с функционален стил, просто описваме свойствата на функцията НОД:

* НОД(a, a) = a - (свойство 1)
* НОД(a, b) = НОД(a – b, b), ако a > b - (свойство 2)
* НОД(a, b) = НОД(a, b – a), ако b > a - (свойство 3)

Ако искаме да сметнем НОД-а на 9 и 3 на пример, ще имаме следното:

НОД(9, 3) = /\* с-во 2 \*/ = НОД(6, 3) = /\* с-во 2 \*/ = НОД(3, 3) = /\* с-во 1 \*/ = 3

Друг пример: Да се намери сумата на квадратите на нечетните числа в списъка L. Кодът на Haskell ще изглежда така:

* foldr1 (+) [ xˆ2 | x <- l, odd x ], което много прилича на стандартния запис на множества.

Също може да се каже, че при императивният стил основният метод на изчисление е променянето на стойности в паметта, а при функционалният е прилагането на функции върху аргументи[[1]](#footnote-1).

Във функционалното програмиране разполагаме със следното:

* функции с параметри, (абстракция)
* които могат да се прилагат над аргументи, (апликация)
* които могат да са други функции (функции от висок ред)
* и могат да се дефинират чрез себе си (рекурсия)

Но нямаме:

* памет
* присвояване
* цикли[[2]](#footnote-2)
* прескачане (goto, break, return)

Предимства на ФП:

* Кратки и ясни програми (изразителност).
* Лесна проверка за коректност (в сравнение с императивни програми).
* При еднакви входни данни връщат един и същ резултат (референциална прозрачност), което позволява...
* Избягване на повторно пресмятане на резултати чрез запомняне (мемоизация).
* Премахване на части от програмата, които не участват в крайния резултат (мъртъв
* код).
* Пренареждане на програмата за по-ефективно изпълнение (стратегия за оценяване).
* Паралелно изпълнение на независими части от програмата (паралелизация).

Недостатъци:

* Ако е нужно да имаме достъп до машината, която програмираме, на много ниско ниво ФП няма много средства за достъп (тъй като в основата си компютрите са императивни (местят/записват/модифицират клетки в паметта, въртят цикли и т.н.).
* От горното следва, че ако искаме да правим оптимизация на ниско ниво ФП не е отговорът.

***Видове функционални езици***

* според типовата система
* динамично типизирани (стойностите имат тип) **[Scheme]**
* статично типизирани (променливите имат тип) **[Haskell]**
* според страничните ефекти
* нечисти (със странични ефекти) **[Scheme]**
* чисти (без странични ефекти) **[Haskell]**
* според стратегията за оценяване
* стриктно (първо сметни, после предай) **[Scheme]**
* лениво (първо предай, после смятай) **[Haskell]**

# Основни понятия в Scheme

В курса по ФП ще научим функционалните езици Scheme и Haskell. Първата половина от семестъра ще сме на Scheme. Средата за програмиране на Scheme, която ще ползваме се казва DrRacket. Това всъщност е интерпретатор, който работи на принципа REPL (read-evaluate-print-loop). Програмите ни ще представляват списък от дефиниции на функции[[3]](#footnote-3), а изпълнението на програма е оценка на израз. Scheme е език създаден основно с учебна цел.

Синтаксис:

* Литерали
* Булеви константи: #t, #f
* Числови константи: -4, 15, 2/3, -2.356 (да, наистина числото 2/3)
* Знакови константи (char-ове): #\a, #\newline
* Низови константи: “Hello world!”
* Символи: f, square, +, find-min. Всъщност това са имената, които в C++ се наричат идентификатори. В Scheme се наричат символи (ще гледам да напомням, че се казват символи за напред, защото е малко объркващо в началото). Изключителна свобода имаме за имената. Можем да ползваме в имената:
  + aA9 - главни/малки букви и цифри.
  + +-\*/ - някои аритметични знаци (в императивните езици са забранени).
  + =<> - оператори за сравнение (пак забранени в императивните езици).
  + ?!@# - някои знаци, които съвсем са забранени в императивните.
  + **Нямаме право да ползваме** ()[]{}, както и интервал.
  + Примери:
    - ala\_bala
    - 12tova+e-str@nen\*no/pozvolen=identifikator
* Комбинации: (<израз\_1> <израз\_2> . . . <израз\_N>). Други изрази оградени в скоби. Много скоби се слагат в този език. И това е общо взето целия синтаксис. Всичко останало е скобуване на изрази по правилния начин.

Оценки на литерали и символи(идентификатори/имена)

* Оценката на булевите константи, знаците, числата и низовете са самите те:
* 5 🡪 5
* #t 🡪 #t
* #\a 🡪 #\a
* "scheme" 🡪 "scheme"
* Оценката на символ е стойността, свързана с него:
* + 🡪 #<procedure:+> (те. + е функция).
* (define a 5), a 🡪 5

Основно правило за оценяване (оценка на комбинация)

( <израз\_0> <израз\_1> . . . <израз\_n> )

| | |

Се оценява до Се оценява до Се оценява до

| | |

F v1 . . . vn

Което се оценява до F(v1, v2, . . . , vn), те. функция на n аргумента. Ако F не е функция става грешка. Всъщност Scheme „разбира“, че даден символ е функция, ако стои точно след някоя отваряща скоба (на мястото на <израз\_1>). Не всички „неща“ се оценяват по основното правило; не всички неща са функции.

Пример за оценяване:

(+ (\* 2 2 2) (\* 3 4)) 🡪 (+ 8 12) 🡪 20

(1 2 3) 🡪 Error

Дефиниция на символи(идентификатори)

(define <символ> <израз>)

* Ключовата дума define оценява <израз> и свързва <символ> с оценката му.
* define НЕ се оценява по основното правило. Конструкции, които не се оценяват по основното правило се наричат **специални форми**.
* Примери:



Дефиниция на функция

(define (<функция> {<параметър>}) <тяло>)

* <функция> и <параметър> са символи
* <тяло> е <израз>
* Символът <функция> се свързва с поредица от инструкции, които пресмятат <тяло> при подадени стойности на <параметър>.
* Примери:



* Вградени числови функции:
  + Аритметични операции: +, -, \*, /
  + Други числови функции: remainder, quotient, max, min, gcd, lcm.
    - quotient – целочислено деление.
    - gcd – greatest common devisor.
    - lcm – least common multiple
* Функции за закръгляне: floor, ceiling, round.
  + round – закръгля число с плаваща точка до цяло число.
* Функции над дробни числа: exp, log, sin, cos, tan, asin, acos, atan, expt, sqrt.
  + exp – експонента при основа e.
  + expt – експонента при подадена основа.
* Вградени предикати:
  + За сравняване на числа: <, >, =, <=, >=
  + Числови предикати: zero?, negative?, positive?, odd?, even?
  + Предикати за проверка на тип: boolean?, number?, char?, string?, symbol?, procedure?

Цитиране

(quote <израз>) или алтернативно ’<израз>

* Оценката на (quote <израз>) или ’<израз> е самият <израз>.
* Примери:



Условна форма if

(if <условие> <израз\_1> <израз\_2>)

* Оценява се условие. Ако оценката е #t, връща се оценката на <израз\_1>. В противен случай се връща оценката на <израз\_2>.
* if е специална форма, защото само един от двата израза се оценява (не се оценява по основното правило).
* Примери:



Форма за многозначен избор cond[[4]](#footnote-4)

(cond (<условие\_1> <израз\_1>)

(<условие\_2> <израз\_2>)

. . .

(<условие\_n> <израз\_n>)

(else <израз\_n+1>))

* Това е аналогично на else if в C++. Оценява се <условие\_1>. При #t връща се <израз\_1>, при #f, оценява се <условие\_2>. При #t, връща се <израз\_2>, при #f, оценява се <условие\_3> и т.н. Ако всички условия са #f, връща се <израз\_n+1>.
* Пример:



Форма за разглеждане на случаи case

(case <тест> ((<случай1,1> . . . <случай1,K1>) <израз\_1>)

((<случай2,1> . . . <случай2,K2>) <израз\_2>)

. . .

((<случайn,1> . . . <случайn,Kn>) <израз\_n>)

(else <израз\_n+1>))

* Аналогично на switch в C++.
  + При някое от <случай1,1>. . . <случай1,k1> → <израз1>, иначе:
  + при някое от <случай2,1>. . . <случай2,k2> → <израз2>, иначе:
  + . . .
  + при някое от <случайn,1>. . . <случайn,kn> → <изразn>, иначе:
  + Връща се <изразn+1>[[5]](#footnote-5)
* Пример:



Логически операции

(not <булев-израз>)

(and <булев-израз\_1> <булев-израз\_2> . . . <булев-израз\_n>)

(or <булев-израз\_1> <булев-израз\_2> . . . <булев-израз\_n>)

* and и or са специални форми (защото не е задължително всички изрази в тях да се оценят).
* and и or работят аналогично на && и || в C++. Тоест, ако при and срещнем израз, който е #f, то останалите не се оценяват и се връща #f. За or, когато срещнем #t, връщаме #t и останалите не се оценяват.

# Модел на средите[[6]](#footnote-6)

Връзката между символите (идентификаторите) и техните оценки (стойности) се записват в **речник[[7]](#footnote-7)**, който се нарича **среда**.

* Всеки символ има най-много една оценка в дадена среда.
* В даден момент могат да съществуват много среди.
* Символите винаги се оценяват в конкретна среда.
* Символите могат да имат различни оценки в различни среди (нещо като област на видимост в императивните езици).
* По подразбиране Scheme работи в **глобалната среда**. В **глобалната среда** са дефинирани символите за стандартни операции и функции (на пример +, \*, /, sqrt, quotient и т.н.).
* Всяка среда пази указател към своята „родителска среда“ (освен **глобалната среда**). Така при изпълнение на програма се получава дърво от среди.
* При срещане на символ при изпълнението на програма първо се търси оценката на символа в текущата среда. Ако не се намери, отива се в родителската среда и се търси там. Ако не се намери, отива се в следващата родителска и т.н. до глобалната среда. Ако и там няма оценка извежда се грешка.

Нека разгледаме следния код за пример:



Имаме просто четири дефиниции на четири символа: x от тип естествено число, y – дробно, str – низ, square – процедура (функция). Те са дефинирани в глобалната среда (кръстена E), която ще изглежда така:



Средата E пази оценките на x, y, str и square. Оценката на square е процедура и процедурите са наредената тройка (параметри | тяло на функцията | указател към средата, в която е оценена функцията (в случая E)). Стрелките са там, за да покажат, че в средите се пазят указатели към оценките, а оценките не са „във“ средата. Тоест на мястото на „…“ стоят указатели.

Нека разгледаме същия код, но с добавено едно извикване на функцията square.



След извикването на (square 7) картинката на средите ще изглежда така:



За по-лесно четене съм сложил оценките на символите директно в средите, но не забравяме, че всъщност на тези места стоят указатели, сочещи към оценките. Стрелката от средата E1 към E означава, че E1 пази указател към E (към „родителската“ си среда).

Какво е станало, когато сме извикали (square 7)? При извикване на функция имаме следното:

* При извикването на функция направи нова среда, която разширява(сочи към) средата, която е записана в наредената тройка, дефинираща функцията. В случая това е средата E и затова E1 има стрелка към E.
* След това в новата средa (E1) запиши параметрите на функцията с техните оценки. В случая записваме, че x е 7 в E1.
* Накрая в новопостроената среда оцени тялото на функцията (сметни функцията) и след оценката върни резултата.

Тук възниква въпросът: какво става, ако извикаме някоя функция 100‘000 пъти? Ще направим 100‘000 отделни среди и ще си напълним оперативната памет? Вярно е, че ще направим 100‘000 отделни среди за всяко отделно викане на ф-ята[[8]](#footnote-8), но за щастие Scheme има garbage collector, който може да определи дали дадена среда ще се използва някога в бъдеще или не[[9]](#footnote-9). И така, ако викнем 100’000 функцията square, garbage collector-ът в някакъв момент ще изчисти получените среди от паметта.

Оценка на вложени функции

Щом вече знаем как се променят средите при дефиниране на символи и извикване на функции да разгледаме следния пример:



Средите ще изглеждат така:



# Специални форми let и let\*

Обща форма на let:



Примери с let:



Обща форма let\*:



Примери с let\*



# Рекурсивни, итеративни и дървовидни процеси

Рекурсия

Най-просто казано рекурсията е повторение чрез позоваване на себе си. Рекурсивна функция е функция, която дава отговор на най-простата задача (база; може и да е повече то една) и показва как сложна задача се свежда до една или повече по-прости задачи от същия тип (стъпка).

Да разгледаме следната функция, която смята факториел:



При изпълнението си тя поражда **линеен рекурсивен процес**. Първо вика себе си докато удари дъното за n = 0, a после на връщане пресмята стойността 1\*2\*…\*n. Има линейна сложност по време и памет.

Да разгледаме и следната функция, която също смята факториел:



При изпълнени си тя поражда **линеен итеративен процес**. Насочваме вниманието си към помощната функция fact, която започва „отдолу нагоре“ и акумулира резултата в result. След като i надвиши n резултатът директно се връща без допълнителни сметки в миналите рекурсивни викания. Това е защото викаме fact рекурсивно без отложена операция (не умножаваме върнатия резултат както в fact-rec). Да няма отложена операция е същото като рекурсивното викане да е в „края“, тоест да е последното нещо, което ще се извърши във функцията. Рекурсивна функция, в която всяко рекурсивно викане е без отложена операция, се нарича **опашкова рекурсия** и за нея имаме:

* Някаква стойност/стойности result, в която се акумулира резултатът.
* Рекурсивната функция е обвита в друга функция, за да има подходящ интерфейс (в примера fact е обвита в fact-iter).
* По стандарт в Scheme **опашковата рекурсия** се интерпретира като цикъл, тоест не се заделя нова памет за рекурсивните викания.
* Сложността по памет е константна (за примера с fact-iter), (правим цикъл, а не рекурсивни викания).

Дървовидна рекурсия

Разглеждаме следните функции, които смятат числата на Фибоначи:



fib-slow вика себе си два пъти и смята едни и същи числа на Фибоначи ненужно много пъти. Сложността ѝ по време е експоненциална с основа златното сечение. Можем да я докараме до линейна като пазим последните две числа и започнем да ги изчисляваме „отдолу нагоре“[[10]](#footnote-10) (fib-fast прави точно това).

# Функции от по-висок ред

Във функционалните езици функциите се третират, като обикновени стойности, както числата, символите и символните низове. Това означава, че може да ги подаваме като параметри или да ги връщаме като стойност[[11]](#footnote-11). Функция, която приема функция като параметър или връща функция като резултат, се нарича **функция от по-висок ред**. Функция от нулев ред приема и връща само стойности, които не са функции. Ф-я от първи ред приема или връща ф-я от нулев ред. Ф-я от втори ред приема/връща от първи и т.н. В Scheme функциите могат да са от произволно висок ред. За примери виж папката „Код“.

1. Разбира се това е на ниво език за програмиране. В крайна сметка компютрите ни изчисляват като променят стойности в паметта. Не мисля, че все още е измислена машина, която смята с функции. [↑](#footnote-ref-1)
2. Можем да ги „симулираме“ с опашкова рекурсия (виж [тук](#_Рекурсивни,_итеративни_и)). [↑](#footnote-ref-2)
3. В Scheme функциите (functions) се казват процедури (procedures). [↑](#footnote-ref-3)
4. Не знам какво точно става като се изпусне else на края? Какво точно ще върне? [↑](#footnote-ref-4)
5. Тук също не знам какво ще върне, ако изпусна else. [↑](#footnote-ref-5)
6. Обясненията в тази глава не са напълно формални, но са достатъчни за добиване на добра представа как работи Scheme „under-the-hood”. [↑](#footnote-ref-6)
7. Речник означава символи и техните дефиниции/стойности. [↑](#footnote-ref-7)
8. Предполагам е възможно да има някакви чудовищни оптимизации в компилатора, които да са неразбираеми за мен. Абстрахираме се от това и мислим за нещата на основно ниво. [↑](#footnote-ref-8)
9. Работата на garbage collector-ът е извън обсега на този курс. Просто приемаме, че той си върши работата правилно. [↑](#footnote-ref-9)
10. Тук има термини като ***динамично програмиране*** и ***мемоизация***, но трябва да погледна какво точно значат. [↑](#footnote-ref-10)
11. В императивните езици също може да подаваме и връщаме функции, на пример с указатели към функции в **C++**, но в повечето случаи това е мазохистично (само погледнете синтаксиса на указател към функция в **C++**). Функционалните езици правят тези операции толкова лесни, колкото да правим аритметични операции с числа. [↑](#footnote-ref-11)