Тема 19. Функционално програмиране. Обща характеристика на функционалния стил на програмиране. Дефиниране и използване на функции. Модели на оценяване. Функции от по-висок ред

1. Характерни особености на функционалния стил на програмиране.

Функционалният стил на програмиране е **декларативен**, т.е. програмите, написани във функционален стил **описват свойствата на желания резултат,** за разлика от програмите, написани в императивен стил, които описват последователността от изчислителни стъки, които водят до резултата.

Съответно, при функционалния стил **отсъстват** традиционните елементи от процедурния стил като цикли, директен достъп до паметта, присвояване, прескачане(go to, break, return).

При ФП стила **основният логически компонент** са **функциите**, които:

- можем да параметризираме, създават ниво на **абстракция**
- могат да бъдат от произволно високо ниво(*First Class Citizens*), т.е.
  - аргументите им могат да бъдат функции
  - могат да връщат като стойност функция
  - могат да бъдат композирани
- могат се прилагат над аргументи(**апликация**)
- дефинират се чрез свързване на **израз** с име
- могат да бъдат дефинирани рекурентно(**рекурсия**), което е основен начин за реализиране на по-сложна логика

ФП стила се базира на изчислителния модел на **λ-смятането**, който дефинира **абстрактното понятие за израз** така:

деф. Нека *x, y, z, …* - изброимо много променливи. Тогава изрази са:

- *x*(променлива)
- *E1(E2)* (апликация), където *E1, E2* - изрази
- *λx E* (абстракция), където *E* - израз

Едно **изчислително правило** - апликация + субституция:
(λx E1)(E2) → E1[x := E2]

Изразителната сила на ФП идва от използването на **чисти функции**, т.е. функции **без странични ефекти**, независимо че някои ФП езици позволяват странични ефекти(напр. Scheme), а при други можем да ги моделираме(напр. в Haskell).

Основни компоненти на функционалните програми.

ФП програмите най-общо казано **се състоят от дефиниции на функции**, а **изпълнението им представлява оценка на израз**.
Ще използваме ***Scheme***, за да покажем основните компоненти на функционалните програми.

i. Примитивни изрази

- **Литерали** - атомарни константи
  - Булеви константи (**#f**, **#t**)
  - Числови константи (**15**, **2/3**, **-1.532**)
  - Знакови константи (**#\a**, **#\newline**)
  - Низови константи (**"Scheme"**, **"hi"**)
- **Символи** (**f**, **square**, **+**, **find-min**) - атомарна променлива

ii. Средства за комбиниране и абстракция

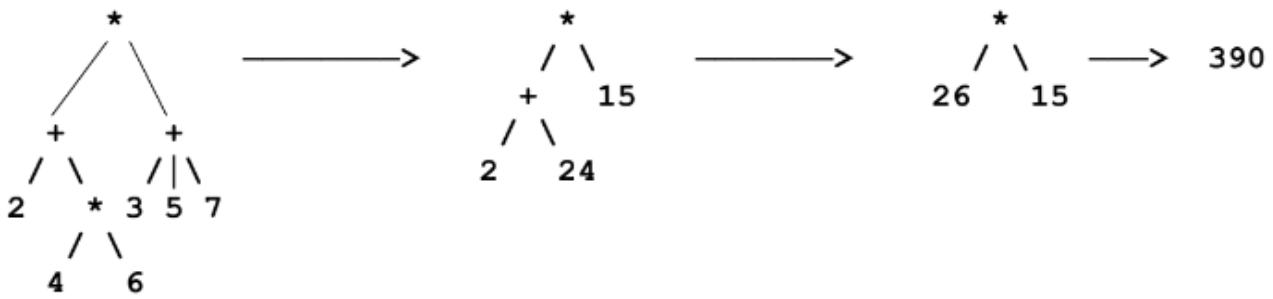
Синтактично изразът е или **атом**(атомарна константа или променлива), или **комбинация**, която има следния вид:
(**<израз$_1$> <израз$_2$> … <израз$_n$>**)
Първият израз наричаме оператор, а останалите - операнди.

iii. Оценяване на израз

На всеки израз се дава оценка.

- Оценката на булевите константи, знаците, числата и низовете са самите те
- Оценката на символ е стойността, свързана с него
- Оценката на комбинация (**<израз$_0$> <израз$_1$> … <израз$_n$>**)
  - **израз$_0$** се оценява до функция ***f***, в противен случай се получава грешка
  - всеки останал израз **израз$_i$** се оценява до **v$_i$**
  - окончателно, комбинацията се оценява до ***f***(**v$_1$, …, v$_1$**)

Например (* (+ 2 (* 4 6)) (+ 3 5 7))



iv. Дефиниране на функция и оценяване на приложение на функция

Дефинирането на функция става чрез специалната форма **define**
забл. **define** не се оценява по основното правило за оценяване на комбинация, затова го наричаме *специална форма*

- Дефиниране на **символи**
  (define <символ> <израз>)
  - Оценява **<израз>** и свързва **<символ>** с оценката му

```
(define s "Scheme is cool")
(define x 2.5)
(define y (+ x 3.2))
```

- Дефиниране на **функции**
  (define (<функция> {<параметър>}) <тяло>)
  - **<функция>** и **<параметър>** са символи
  - **<тяло>** е израз

```
(define (square x) (* x x))
(define (f x y) (+ (square (1+ x)) (square y) 5)
(define (h) (+ 2 3)) ; h е от тип procedure
```

Дефинирането на функции е основно **средство за абстракция**, тъй като ни позволява да ***модуляризираме*** програмата.

Има и компоненти, които са вградени в езика и ни позволяват да създаваме по-сложни програми, например:

- **(if <условие> <израз$_1$> <израз$_2$>) специална форма**
  - ако **<условие>** се оцени до #t, връща се оценката на **<израз$_1$>**
  - ако **<условие>** се оцени до #f, връща се оценката на **<израз$_2$>**

```
(if (< 3 6) (- 4 2) (- 6 9))
```

- **(cond (<условие$_1$> <израз$_1$>) специална форма**
  (<условие$_2$> <израз$_2$>)
  . . .
  (<условие$_n$> <израз$_n$>)
  (else <израз$_{n+1}$>))
  - Оценява се **<условие$_1$>**, при **#t** се връща **<израз$_1$>**, а при **#f:**
  - Оценява се **<условие$_2$>**, при **#t** се връща **<израз$_2$>**, а при **#f:**
  - **. . .**
  - Оценява се **<условие$_n$>,** при **#t** се връща **<израз$_n$>,** а при **#f:**
  - Връща се **<израз$_{n+1}$>**

```
(define (grade x)
(cond ((>= x 5.5) "Отличен")
((>= x 4.5) "Много добър")
((>= x 3.5) "Добър")
((>= x 3) "Среден")
```

```
(else "Слаб")))
```

- **Вложени дефиниции**
  (define (<функция> {<параметър}) {<дефиниция>} <тяло>)
  При извикване на **<функция>** първо се оценяват всички **<дефиниция>** и след това се оценява **<тяло>**
  - Първо се създава среда E_1, в която формалните параметри се свързват с оценките на фактическите
  - След това се създава среда E_2, която разширява E_1, за вложените дефиниции
  - В средата E_2 се записват всички символи от вложените дефиниции без стойности
  - Всички вложени дефиниции се **оценяват** в E_2
  - Накрая получените оценки се свързват със съответните си символи в E_2

```
(define (dist x1 y1 x2 y2)
(define dx (- x2 x1)
(define dy (- y2 y1)
(define (sq x) (-x2 x1))
(sqrt (+ (sq dx) (sq dy))))
```

- **let специална форма**
  - (**let** (**{(<символ> <израз>)}**) **<тяло>**)
  - (**let** ((**<символ$_1$> <израз$_1$>**)
    (**<символ$_2$> <израз$_2$>**)
    . . .
    (**<символ$_n$> <израз$_n$>**))
    <тяло>)

  При оценка на **let** в среда E:
  - Създава се нова среда E_1 - разширение на текущата среда E
  - Оценката на **<израз$_1$>** в E се свързва със символ **<символ>** в E_1
  - Оценката на **<израз$_2$>** в E се свързва със символ **<символ>** в E_2
  - …
  - Оценката на **<израз$_n$>** в E се свързва със символ **<символ>** в E_n
  - Връща се оценката на **<тяло>** в средата E_n

  let няма странични ефекти върху средата

v. Апликативно(стриктно, call-by-value) и нормално оценяване(лениво, call-by-name)

Апликативна(стриктна, call-by-value) стратегия за оценяване - извършва се оценка на фактическите параметри преди прилагане - такава е в ***Scheme***

- използва се в повечето езици за програмиране

- позволява лесно да се контролира редът на изпълнение
- пестеливо е откъм памет

Нормална(ленива, call-by-name) стратегия за оценяване - замества се името на процедурата с тялото ѝ, докато се получат означения на примитивни процедури, и след това се прилагат техните правила за оценка
- по-рядко използвана, макар че нейни елементи съществуват в повечето езици - напр. short circuit evaluation, тернарен оператор
- може да спести време, прекарано в излишно оценяване на фактически параметри, но е за сметка на памет

Такава е стратегията в ***Haskell***, само че той ползва и мемоизация като оптимизация(**call-by-need**).
По теоремата за нормализация на Curry, ако има някакъв ред на оценяване на програма, който стига до резултат, то и с нормална стратегия на оценяване ще достигнем до същия резултат. *Има случаи, когато нормалното оценяване дава резултат, но апликативното - не.*

2. Функции от по-висок ред. Функциите като параметри и оценки на обръщения към функции. Анонимни(ламбда) функции

Както вече обърнахме внимание, във ФП функциите са *първокласни* стойности, те могат да бъдат параметри на функции и оценки на обръщения към функции.

```
(define (fix-point? f x) (= (f x) x))
; (fixed-point? sin 0) → #t
; (fixed-point? exp 1) → #f
```

Анонимни(ламбда функции)
Можем да конструираме параметрите на функциите от по-висок ред “на място”, без да им даваме имена.
(lambda ({<параметър>}) <тяло>)
- Оценява се до функционален обект със съответните параметри и тяло
- Анонимната функция пази указател към средата, в която е оценена

```
(lambda (x) (+ x 3)); → #<procedure>
((lambda (x) (+ x 3)) 5); → 8
```

Вече можем да кажем, че следните са еквивалентни:
```
(define ((<име> <параметри>) <тяло>)
(define <име> (lambda (<параметри>) <тяло>))
```

Примери:
```
(define (compose f g) (lambda (x) (f (g x))))
(define (add1 x) (+ x 1))
(define (square x) (* x x))
; ((compose square add1) 3) → 16
```
Но ако `(define (compose f g x) (f (g x)))`, то извикването следва да бъде:

```
; (compose square add1 3) → 16
```