

Лекция 1

**Основни принципи на
функционалното програмиране.
Начални сведения за Haskell и
Hugs**

компютърна система = хардуер + софтуер

Софтуерът (програмното осигуряване)
управлява поведението на хардуера.

софтуер
(програмно
осигуряване) = програми + данни

Данни: информация, която може да бъде съхранена в паметта на компютъра.

Примери за данни:

- ✓ числа
- ✓ букви
- ✓ съобщения по електронната поща
- ✓ карти
- ✓ песни върху CD
- ✓ видео клипове
- ✓ щраквания на мишката
- ✓ програми

Програми: описания на начини за манипулиране на данните.

Езици за програмиране: изкуствени езици, създадени специално за записване на програми.

Изграждане на софтуерни системи

Една голяма система може да съдържа милиони редове програмен код.

Софтуерните системи са измежду най-сложните артефакти, които някога са създавани.

Изграждат се чрез комбиниране на съществуващи компоненти, доколкото това е възможно.

Езици за програмиране

Формални изкуствени езици, предназначени за записване на програми.

Съществуват стотици езици за програмиране, всеки от които има своите силни и слаби страни.

Класификация: съществуват различни **класификационни признаци** (машинно-ориентирани езици и езици от високо ниво)

Типове езици за програмиране ОТ ВИСОКО НИВО

- Процедурни (императивни)
как?

програма = алгоритъм + структури от данни

Типове езици за програмиране ОТ ВИСОКО НИВО

- Декларативни (дескриптивни)
какво?

програма = списък от дефиниции на функции
или
списък от равенства
или
факти + правила

Основни характеристики на функционалното програмиране

Програмирането във функционален стил се
състои от:

- дефиниране на функции, които пресмятат (връщат) стойности. При това тези стойности се определят еднозначно от стойностите на съответните аргументи (фактически параметри)

- прилагане (апликация) на тези функции върху подходящи аргументи, които също могат да бъдат обръщения към функции

Пример 1: програма за намиране на сумата на естествените числа от 1 до n

- В процедурен (императивен) стил:

count := 0

total := 0

repeat

count := count + 1

total := total + count

until

count = n

- Във функционален стил (на езика Haskell):

`sum [] = 0`

`sum (x : xs) = x + sum xs`

`sum [1..n]`

Пример 2: “бързо” сортиране

qsort [] = []

qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger

where

smaller = [a | a ← xs, a ≤ x]

larger = [b | b ← xs, b > x]

Основни предимства на функционалния стил на програмиране

- може да се извършва лесна проверка и поправка на съответните програми поради липсата на странични ефекти
- могат да бъдат доказвани строго (с математически средства) свойства на функционалните програми

Основни недостатъци на функционалния стил на програмиране

- строгата функционалност понякога изисква многократно пресмятане на едни и същи изрази (в Haskell този проблем по принцип е преодолян)
- неестествено и често неефективно е използването им при решаване на задачи от процедурен (алгоритмичен) характер

Още за функционалния стил на програмиране

Функционалният стил на програмиране е съществено различен от този, поддържан от популярните езици за програмиране като Java, C++, C и Visual Basic. В частност, идеологията на посочените езици за програмиране е тясно свързана с архитектурата на съответния хардуер в смисъл, че програмирането се основава на идеята за променяне на съхранени стойности.

Обратно, Haskell поддържа по-абстрактен стил на програмиране, основан на идеята за прилагане на функции към аргументи.

Преминаването към това по-високо ниво на абстракция води до възможност за съставяне на значително по-прости програми и поддържа множество от мощни подходи за конструиране на програми и изследване на техните свойства.

Факти от историята на функционалното програмиране

- 30-те години на 20-ти век: *Alonzo Church*, *лямбда смятане* - математическа теория на функциите, дала тласък в развитието на множество езици за функционално програмиране.

- 50-те години на 20-ти век: *John McCarthy*, език за програмиране *LISP* (“LISt Processor”) – първият език за функционално програмиране, влияние върху който оказва ламбда смятането. Допуска се присвояване на стойности и промяна на стойността на променлива.

- 60-те години на 20-ти век: *Peter Landin*, език за програмиране *ISWIM* (“If you See What I Mean”) – първият език за строго функционално програмиране, основан на ламбда смятането и изключващ присвояването на променливи.
- 70-те години на 20-ти век: *John Backus*, *FP* (“Functional Programming”) – език за функционално програмиране, който набляга на дефинирането и използването на функции от по-висок ред и изследването на свойствата на програмите.

- 70-те години на 20-ти век: *Robin Milner* и др., *ML* (“Meta-Language”) – първият модерен език за функционално програмиране, в който се въвежда идеята за полиморфни типове и извод на типове.
- 70-те и 80-те години на 20-ти век: *David Turner, Miranda* (“admirable”) и множество от езици за “мързеливо” (lazy) функционално програмиране.

- 1987 г.: *международен комитет, Haskell* (на името на Haskell B. Curry, един от пионерите на ламбда смятането) – модерен език за “мързеливо” строго функционално програмиране.
- 2003 г.: *международен комитет, the Haskell Report* – дефиниция на стабилна версия на езика Haskell, която започва да играе ролята на негов стандарт.

Нашият подход

Haskell – език за строго функционално програмиране:

- език от много високо ниво (грижата за много детайли се поема автоматично)
- с голяма изразителна сила, позволява писане на много кратък и компактен код
- подходящ за работа със сложни данни и за комбиниране на готови компоненти
- дава приоритет на времето на програмиста, а не на компютърното време

Най-популярната среда за програмиране на Haskell е **Hugs** (1998). Тя предоставя много добри средства за обучение и се разпространява безплатно за множество платформи.

Haskell home page:

<http://www.haskell.org/haskellwiki/Haskell>

Функции

Функцията е програмна част, която връща стойност (резултат).

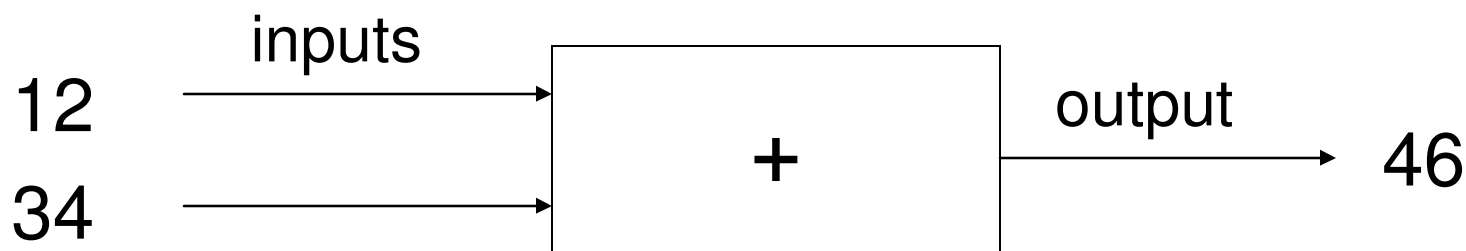
Изобразяване на функциите: чрез диаграми от вида



inputs: входни стойности (аргументи или параметри)

output: изход (върнатата стойност, резултат)

Пример: функцията + за събиране на числа



Процесът на задаване на конкретни стойности на аргументите на функцията и пресмятане на съответната ѝ стойност се нарича **апликация** (прилагане на функцията).

Може да се създават (дефинират) функции с различен брой аргументи (0, 1, 2 или повече).

Аргументите на функцията, както и върнатата от нея стойност, могат да бъдат от различни **типове** (не е задължително да бъдат числа).

Величини. Типове

Величините са основно средство за изразяване на стойностите, с които се работи в една програма.

Всяка величина се характеризира с име, тип и (текуща) стойност.

Типът представлява множество от допустими стойности заедно с определена съвкупност от операции, приложими върху тези стойности.

Дефиниции

Всяка функционална програма представлява поредица от дефиниции на функции и други величини.

Всяка дефиниция на Haskell свързва (асоциира) дадено **име** (идентификатор) със **стойност** от определен **тип**.

В най-простия случай дефинициите имат вида

`name :: type`

`name = expression`

Една дефиниция от посочения вид свързва името от лявата страна на равенството със стойността на **израза** от дясната страна.

Символът “::” би трябвало да се чете “е от тип”.

Имената на функциите и останалите величини започват с малки букви, докато имената на типовете започват с главни букви.

Пример

```
size :: Int
```

```
size = 12+13
```


Изрази

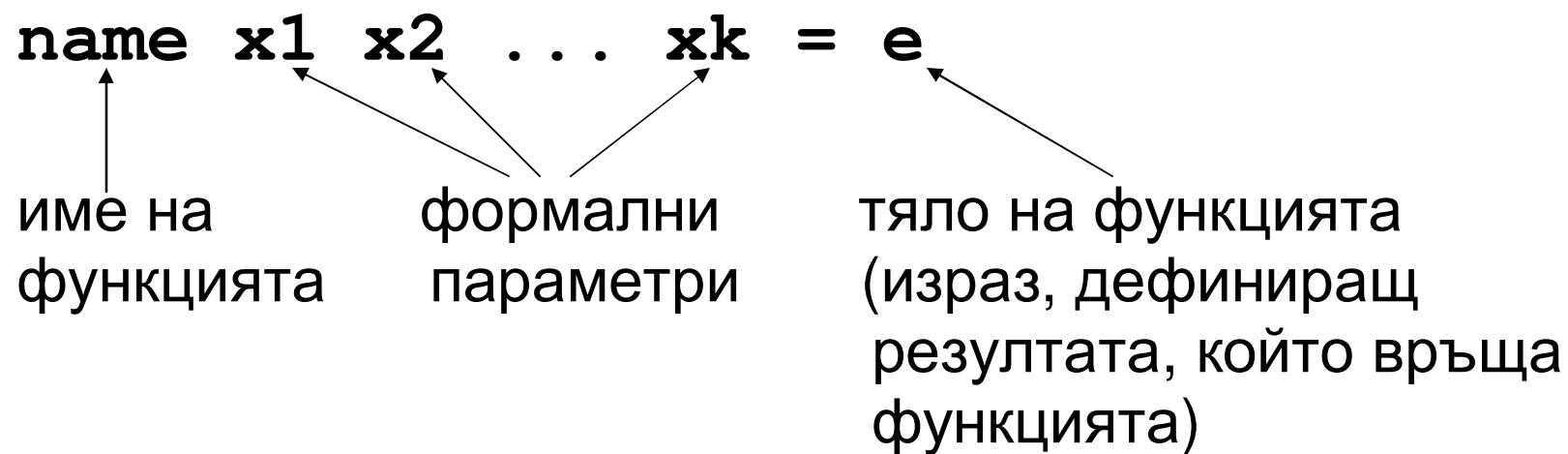
Всеки израз представлява правило за пресмятане (намиране) на стойност.

Изразите се образуват чрез композиция на операции над определени величини.

Операциите се означават с определени знакове или с идентификатори (пореждици от букви и цифри, които започват с буква). Всяка операция се прилага към стойности от определен(и) тип(ове) и формира стойност от определен тип.

Дефиниции на функции

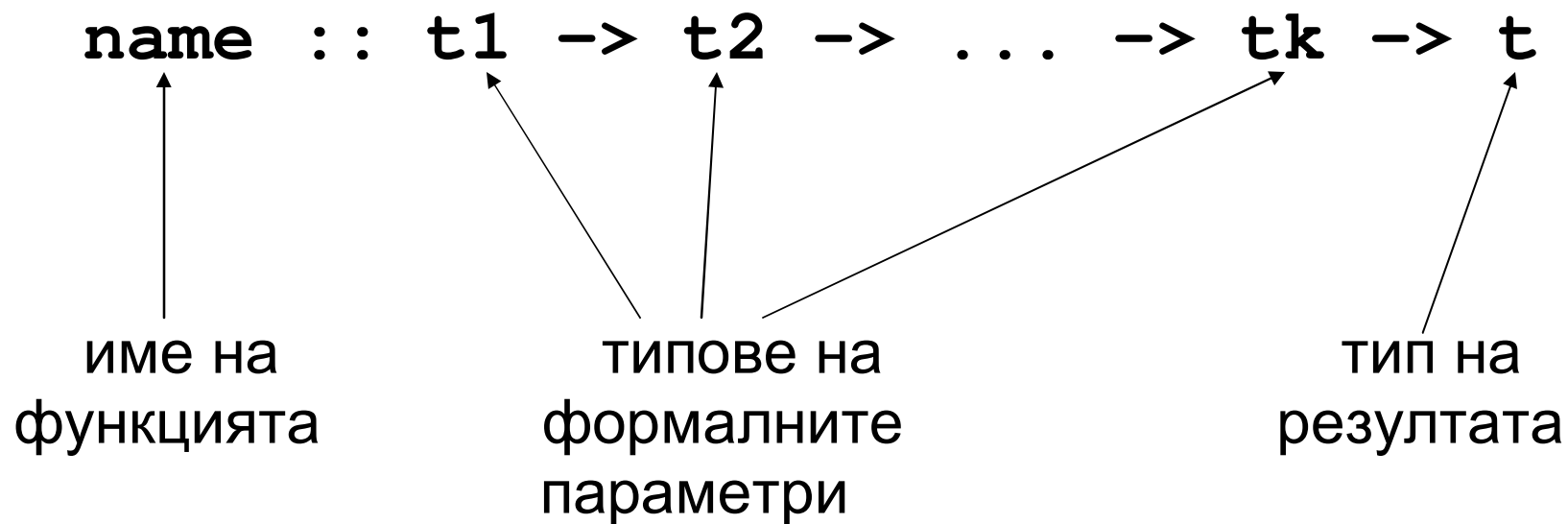
Дефинициите на функции имат следния общ вид:



Формални параметри (аргументи): имена на независимите променливи на функцията (в математическия смисъл на това понятие).

Дефиницията на функция трябва да бъде предшествана от декларация на нейния тип (на типовете на аргументите и типа на резултата, който връща функцията).

Общ вид на декларация на типа на функция:



Примери

```
square :: Int -> Int
```

```
square n = n*n
```

```
> square 3
```

```
9
```

```
> square 5
```

```
25
```



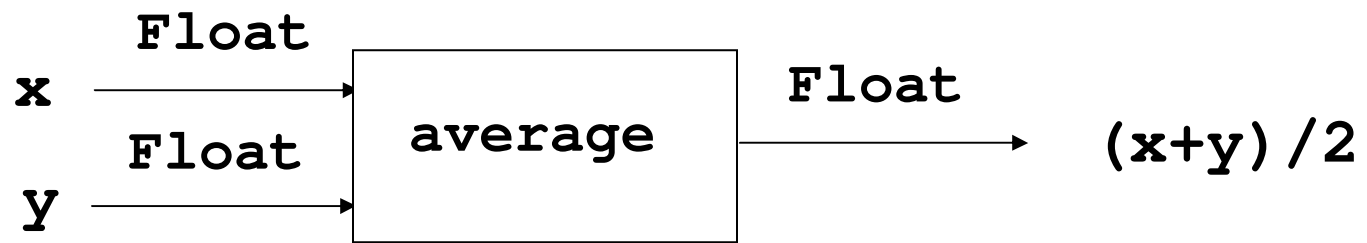
```
average :: Float -> Float -> Float  
average x y = (x+y)/2
```

```
> average 3.4 5.6
```

```
4.5
```

```
> average 3 4
```

```
3.5
```



Общ вид на програмата на Haskell

Програмите на Haskell обикновено се наричат **скриптове (scripts)**. Освен програмния код (поредица от дефиниции на функции) един скрипт може да съдържа и коментари.

Има два различни стила на писане на скриптове, които съответстват на две различни философии на програмиране.

Традиционно всичко в един програмен файл (файл с изходния код на програма на Haskell) се интерпретира като програмен текст (код), освен ако за нещо е отбелязано специално, че представлява коментар. Скриптовете, написани в такъв (traditional) стил, се съхраняват във файлове с разширение “.hs”.

Традиционно коментари се означават по два начина. Символът “--“ означава начало на коментар, който продължава от съответната позиция до края на текущия ред. Коментари, които съдържат произволен брой знакове и евентуално заемат повече от един ред, могат да бъдат заключени между символите “{-“ и “-}”.

Алтернативният (literate) подход предполага, че всичко във файла е коментар освен частите от текста, специално означени като програмен код.

В Пример 2 програмният текст е само в редовете, започващи с “>” и отделени от останалия текст с празни редове.

Този вид скриптове се съхраняват във файлове с разширение “.lhs”.

Пример 1. A traditional script

```
{- #####  
    MyFirstScript.hs  
##### -}  
  
-- The value size is an integer (Int), defined to be  
-- the sum of 12 and 13.  
  
size :: Int  
size = 12+13  
  
-- The function to square an integer.  
  
square :: Int -> Int  
square n = n*n
```

```
-- The function to double an integer.
```

```
double :: Int -> Int
```

```
double n = 2*n
```

```
-- An example using double, square and size.
```

```
example :: Int
```

```
example = double (size - square (2+2))
```

Пример 2. A literate script

```
{- #####  
    MyFirstLiterate.lhs  
##### -}
```

The value `size` is an integer (`Int`), defined to be the sum of 12 and 13.

```
> size :: Int  
> size = 12+13
```

The function to square an integer.

```
> square :: Int -> Int  
> square n = n*n
```

The function to double an integer.

```
> double :: Int -> Int  
> double n = 2*n
```

An example using double, square and size.

```
> example :: Int  
> example = double (size - square (2+2))
```


Библиотеки на Haskell

Haskell поддържа множество вградени типове данни: цели и реални числа, булеви стойности, низове, списъци и др., както и предлага вградени функции за работа с данни от тези типове.

Дефинициите на основните вградени функции в езика се съдържат във файл (**the standard prelude**) с името `Prelude.hs`. По подразбиране при стартиране на Hugs (или друга среда за програмиране на Haskell) най-напред се зарежда `the standard prelude`, след което потребителят може да започне своята работа.

Напоследък, с цел намаляване на обема на the standard prelude, дефинициите на част от вградените функции се преместват от the standard prelude в множество **стандартни библиотеки**, които могат да бъдат включени от потребителя в средата на Haskell при необходимост.

Модули

Възможно е текстът на една програма на Haskell да бъде разделен на множество компоненти, наречени **модули**.

Всеки модул има свое **име** и може да съдържа множество от дефиниции на Haskell. За да се дефинира даден модул, например `Aut`, е необходимо в началото на програмния текст в съответния файл да се включи ред от типа на

```
module Aut where
```

```
.....
```

Един модул може да **импортира** дефиниции от други модули. Например модулът Bee ще може да импортира дефиниции от модула Aut чрез включване на оператор `import` както следва:

```
module Bee where
import Aut
. . . . .
```

В случая операторът `import` означава, че при дефинирането на функции в `Вее` могат да се използват всички (**видими**) дефиниции от `Aut`.

Механизмът на модулите поддържа споменатите по-горе библиотеки.

Механизмът на модулите позволява да се определи кои дефиниции да бъдат достъпни чрез **експортиране** от даден модул за употреба от други модули.