Haskell

Prezentacja języka

Magdalena Zięba

Informatyka Stosowana 4 rok
spec.:Informatyka w Sterowaniu i Zrządzaniu

Tarnów, 26 stycznia 2009

Wybrane zagadnienia - czyli co warto wiedzieć

Wybrane zagadnienia - czyli co warto wiedzieć

Wybrane zagadnienia - czyli co warto wiedzieć

Wybrane zagadnienia - czyli co warto wiedzieć

Wybrane zagadnienia - czyli co warto wiedzieć

Wybrane zagadnienia - czyli co warto wiedzieć

Wybrane zagadnienia - czyli co warto wiedzieć

Wybrane zagadnienia - czyli co warto wiedzieć

Wybrane zagadnienia - czyli co warto wiedzieć

Wybrane zagadnienia - czyli co warto wiedzieć

Wybrane zagadnienia - czyli co warto wiedzieć

1. Krótkie wprowadzenie
   - Co to właściwie jest Haskell
   - Historia języka
   - Używać czy nie używać?
   - Haskell i błędy
   - Haskell a programwanie obiektowe
2. Dostępne kompilatory
3. Podstawowe zagadnienia
   - Praca interaktywna
   - Pierwszy program
   - Operatory
   - Typy danych
   - Funkcje
   - Listy
4. Przykłady

Wybrane zagadnienia - czyli co warto wiedzieć

1. Krótkie wprowadzenie
   - Co to właściwie jest Haskell
   - Historia języka
   - Używać czy nie używać?
   - Haskell i błędy
   - Haskell a programwanie obiektowe
2. Dostępne kompilatory
3. Podstawowe zagadnienia
   - Praca interaktywna
   - Pierwszy program
   - Operatory
   - Typy danych
   - Funkcje
   - Listy
4. Przykłady

Wybrane zagadnienia - czyli co warto wiedzieć

Wybrane zagadnienia - czyli co warto wiedzieć

Wybrane zagadnienia - czyli co warto wiedzieć

1. Krótkie wprowadzenie
   - Co to właściwie jest Haskell
   - Historia języka
   - Używać czy nie używać?
   - Haskell i błędy
   - Haskell a programwanie obiektowe
2. Dostępne kompilatory
3. Podstawowe zagadnienia
   - Praca interaktywna
   - Pierwszy program
   - Operatory
   - Typy danych
   - Funkcje
   - Listy
4. Przykłady

Haskell - co to takiego?

Haskell jest jedym z wielu funkcyjnych języków programowania, do których należą m.in Lisp, Erlang i inne. Jednak w odróżnieniu od innych jest językiem *czysto funkcyjnym*, dzięki czemu nie pozwala na skutki uboczne. Głównym założeniem języków czysto funkcyjnych jest to, że wynik działania funkcji jest uzależniony od przekazanych jej argumentów i tylko od nich. Ocena programu funkcyjnego jest więc równoważna z oceną funkcji w czystym matematycznym sensie.

Haskell jest często nazywany językiem *"leniwym"* (ang. "lazy" or "non-strict"), gdyż wyrażenia, które nie są potrzebne, by ustalić odpowiedź na dany problem nie są wyznaczane. Zatem leniwe wartościowanie gwarantuje, że nic nie zostanie policzone niepotrzebnie.

Warto wspomnieć również, że Haskell jest językiem stosującym silne typowanie. Niemożliwa jest więc przypadkowa konwersja np. Double do Int.

Jak powstał Haskell

W 1987 roku w Portland w stanie Oregon, odbyła się konferencja dotycząca funkcyjnych języków programowania i architektury komputera (Functional Programming Languages and Computer Architecture FPCA '87). Poruszono na niej temat niezbyt dobrej sytuacji w środowisku programowania funkcyjnego. Jednomyślnie zdecydowano o utworzoniu specjalnego komitetu, którego zadaniem będzie zaprojektowanie języka według ustalonych zasad.
Odzwierciedleniem wysiłku i pracy komitetu jest właśnie czysto funkcyjny język programowania - Haskell.
Nazwa języka pochodzi od znanego logika Haskell'a Brooks'a.
W 1997 roku Haskell Workshop w Amsterdamie zadecydowało o potrzebie stworzenia stabilnej wersji, która ukazała się później jako "Haskell 98". Oryginalny Haskell Report powiązał jedynie język wraz ze standardową biblioteką o nazwie Prelude.

Dlaczego warto stosować język Haskell

Istnieje dużo motywów, dla których warto używać język Haskell. Jedną z przyczyn jest niezaprzeczalnie to, że program napisany w języku Haskell jest w większym stopniu wolny od bugów niż napisany w innym języku. Program może być również bardziej czytelny. Jednak najbardziej znaczące może być to, że środowisko Haskell jest niewiarygodnie pomocne. Język stale się rozwija (nie dlatego że jest niestabilny, raczej że są liczne rozszerzenia, które zostały dodane do różnych kompilatorów, a są bardzo przydatne) oraz uwzględnia sugestie użytkowników kiedy nowe rozszerzenia mają zostać zaimplementowane.

Powody, dla których nie wykorzystuje się języka Haskell

Do najbardziej znanych zastrzeżeń użytkowników Haskell'a należą przede wszystkim:

- wygenerowany kod jest powolniejszy niż równoważy program napisany w innym języku, np. C.
- trudno zdebugować program - jednakże nie jest to wielki problem, większość kodu nie zawiera żadnych bugów.

Powody, dla których nie wykorzystuje się języka Haskell

Do najbardziej znanych zastrzeżeń użytkowników Haskell'a należą przede wszystkim:

- wygenerowany kod jest powolniejszy niż równoważy program napisany w innym języku, np. C.
- trudno zdebugować program - jednakże nie jest to wielki problem, większość kodu nie zawiera żadnych bugów.

Haskell i błędy

Programy napisane w Haskellu zawierają mniej błędów, ponieważ Haskell:

- jest językiem czystym – nie ma efektów ubocznych – za każdym razem gdy wywołamy funkcję z takimi samymi parametrami, zwróci ona taką samą wartość;
- stosuje silne typowanie – nie ma niejawnych przekształceń typów;
- jest zwięzły – programy są krótsze, dzięki czemu łatwiej jest analizować poszczególne funkcje i lokalizować błędy;
- jest językiem wysokiego poziomu – programy napisane w Haskellu często są bardzo podobne do opisu algorytmu. Pisząc funkcje na wyższym poziomie abstrakcji, zostawiając szczegóły kompilatorowi, zmniejszamy szanse pojawienia się błędów;
- zarządza pamięcią – programista zajmuje się jedynie implementacją algorytmu, a nie zarządzaniem pamięcią;
- jest modularny – Haskell oferuje bardzo wiele metod łączenia modułów.

Haskell i błędy

Programy napisane w Haskellu zawierają mniej błędów, ponieważ Haskell:

- jest językiem czystym – nie ma efektów ubocznych – za każdym razem gdy wywołamy funkcję z takimi samymi parametrami, zwróci ona taką samą wartość;
- stosuje silne typowanie – nie ma niejawnych przekształceń typów;
- jest zwięzły – programy są krótsze, dzięki czemu łatwiej jest analizować poszczególne funkcje i lokalizować błędy;
- jest językiem wysokiego poziomu – programy napisane w Haskellu często są bardzo podobne do opisu algorytmu. Pisząc funkcje na wyższym poziomie abstrakcji, zostawiając szczegóły kompilatorowi, zmniejszamy szanse pojawienia się błędów;
- zarządza pamięcią – programista zajmuje się jedynie implementacją algorytmu, a nie zarządzaniem pamięcią;
- jest modularny – Haskell oferuje bardzo wiele metod łączenia modułów.

Haskell i błędy

Programy napisane w Haskellu zawierają mniej błędów, ponieważ Haskell:

- jest językiem czystym – nie ma efektów ubocznych – za każdym razem gdy wywołamy funkcję z takimi samymi parametrami, zwróci ona taką samą wartość;
- stosuje silne typowanie – nie ma niejawnych przekształceń typów;
- jest zwięzły – programy są krótsze, dzięki czemu łatwiej jest analizować poszczególne funkcje i lokalizować błędy;
- jest językiem wysokiego poziomu – programy napisane w Haskellu często są bardzo podobne do opisu algorytmu. Pisząc funkcje na wyższym poziomie abstrakcji, zostawiając szczegóły kompilatorowi, zmniejszamy szanse pojawienia się błędów;
- zarządza pamięcią – programista zajmuje się jedynie implementacją algorytmu, a nie zarządzaniem pamięcią;
- jest modularny – Haskell oferuje bardzo wiele metod łączenia modułów.

Haskell i błędy

Programy napisane w Haskellu zawierają mniej błędów, ponieważ Haskell:

- jest językiem czystym – nie ma efektów ubocznych – za każdym razem gdy wywołamy funkcję z takimi samymi parametrami, zwróci ona taką samą wartość;
- stosuje silne typowanie – nie ma niejawnych przekształceń typów;
- jest zwięzły – programy są krótsze, dzięki czemu łatwiej jest analizować poszczególne funkcje i lokalizować błędy;
- jest językiem wysokiego poziomu – programy napisane w Haskellu często są bardzo podobne do opisu algorytmu. Pisząc funkcje na wyższym poziomie abstrakcji, zostawiając szczegóły kompilatorowi, zmniejszamy szanse pojawienia się błędów;
- zarządza pamięcią – programista zajmuje się jedynie implementacją algorytmu, a nie zarządzaniem pamięcią;
- jest modularny – Haskell oferuje bardzo wiele metod łączenia modułów.

Haskell i błędy

Programy napisane w Haskellu zawierają mniej błędów, ponieważ Haskell:

- jest językiem czystym – nie ma efektów ubocznych – za każdym razem gdy wywołamy funkcję z takimi samymi parametrami, zwróci ona taką samą wartość;
- stosuje silne typowanie – nie ma niejawnych przekształceń typów;
- jest zwięzły – programy są krótsze, dzięki czemu łatwiej jest analizować poszczególne funkcje i lokalizować błędy;
- jest językiem wysokiego poziomu – programy napisane w Haskellu często są bardzo podobne do opisu algorytmu. Pisząc funkcje na wyższym poziomie abstrakcji, zostawiając szczegóły kompilatorowi, zmniejszamy szanse pojawienia się błędów;
- zarządza pamięcią – programista zajmuje się jedynie implementacją algorytmu, a nie zarządzaniem pamięcią;
- jest modularny – Haskell oferuje bardzo wiele metod łączenia modułów.

Haskell i błędy

Programy napisane w Haskellu zawierają mniej błędów, ponieważ Haskell:

- jest językiem czystym – nie ma efektów ubocznych – za każdym razem gdy wywołamy funkcję z takimi samymi parametrami, zwróci ona taką samą wartość;
- stosuje silne typowanie – nie ma niejawnych przekształceń typów;
- jest zwięzły – programy są krótsze, dzięki czemu łatwiej jest analizować poszczególne funkcje i lokalizować błędy;
- jest językiem wysokiego poziomu – programy napisane w Haskellu często są bardzo podobne do opisu algorytmu. Pisząc funkcje na wyższym poziomie abstrakcji, zostawiając szczegóły kompilatorowi, zmniejszamy szanse pojawienia się błędów;
- zarządza pamięcią – programista zajmuje się jedynie implementacją algorytmu, a nie zarządzaniem pamięcią;
- jest modularny – Haskell oferuje bardzo wiele metod łączenia modułów.

Haskell a programowanie obiektowe

Język Haskell, podobnie jak języki obiektowe, umożliwia tworzenie abstrakcyjnych struktur danych, *polimorfizm*i *enkapsulację*.
Enkapsulacja danych jest realizowana poprzez umieszczenie każdego typu danych w osobnym module, z którego eksportowany jest tylko interfejs i tylko ten interfejs jest widziany na zewnątrz modułu. Dzięki temu, że tylko funkcje będące częścią interfejsu mogą być użyte „na zewnątrz" – szczegóły implementacji mogą zostać ukryte wewnątrz modułu. W Haskellu typ danych oraz funkcje działające na tych danych nie są zgrupowane wewnątrz „obiektu", ale wewnątrz modułu.
Polimorfizm jest realizowany poprzez zastosowanie tzw. klas typów. Klasy typów są w Haskellu dokładnie tym, czego można się spodziewać po ich nazwie. Są zbiorem zasad, które musi spełnić typ danych, aby mógł być traktowany jako instancja tej klasy.
Haskell nie pozwala jednak na grupowanie danych i funkcji na nich działających w pojedynczy obiekt.

Wśród dostępnych kompilatorów wyróżniamy

- GHC
- Hugs
- nhc98
- Yhc
- HBI i HMC
- Helium
- Jhc
- Yale Haskell
- DDC

Do najbardziej popularnych zalicza się **GHC** i **Hugs**

GHC the Glasgow Haskell Compiler

GHC jest optymalnym kompilatorem języka Haskell, dostarczającym jego wiele rozszerzeń. GHC jest de facto standardowym kompilatorem, dzięki któremu uzyskujemy szybki kod. GHC został napisany w Haskell'u (wraz z rozszerzeniami) i jego wielkość oraz złożoność sprawiają, że nie jest tak "przenośny"jak Hugs, oraz działa wolniej i potrzebuje więcej pamięci. Jednakże programy, które są wynikiem jego działania są znacznie szybsze. Posiada także interaktywne środowisko GHCi, które jest podobne do Hugs ale posiada interaktywne ładowanie skompilowanego kodu. Kompilator ten jest dostepny dla najbardziej znanych platform, takich jak Windows, Mac OS X, i kilku wariantów Unix'a (Linux, *BSD, Solaris). **Dostępny pod adresem http://www.haskell.org/ghc/**

The Haskel Interpreter Hugs

Jest to mały, przenośny interpreter języka Haskell, napisany w języku C i działający na prawie każdej maszynie. W swojej kategorii jest najlepszym kompilatorem używanym jako system opracowywania programów. Może pochwalić się niezwykle szybką kompilacją oraz wygodnym interaktywnym interpreterem. Jest to najlepszy system dla nowicjuszy do nauki języka Haskell.
Dostępny jest dla wszystkich Unixowych platform (włączając Linuxa), Dosa, Windowsa i Macintosha. Posiada wiele bibliotek, także biblioteki Win32, zaś wersja pod Windows o nazwie winHugs posiada graficzny interfejs użytkownika.
Dostępny pod adresem http://haskell.org/hugs/

nhc98

Jest to mały, prosty w instalacji kompilator dostosowany do wersji Haskell 98. Dostarcza mnóstwo nowoczesnych profili, które nie zostały wykorzytane w żadnym innym kompilatorze Haskella. Produkuje średnio-szybkie kody, zaś kompilacja jest dość szybka. Kompilator charakteryzuje się oszczędnością przestrzeni, produkuje małe programy, które potrzebują niewielką ilość przestrzeni w trybie runtime.
Dostepny jest dla każdego rodzaju Unixów (including Mac OS X, Cygwin/Windows, Linux, Solaris, *BSD, AIX, HP-UX, Ultrix, IRIX, etc.)
Został napisany w języku Haskell 98, ale może być szybko i łatwo załadowany z źródeł C.
Dostępny pod adresem http://haskell.org/nhc98/

Praca interaktywna

W odróżnieniu od typowych języków kompilowanych, Haskell pozwala na pracę interaktywną. Praca taka, zwana sesją, polega na wpisywaniu wyrażeń, które Haskell oblicza i wypisuje ich wynik. Zatem w najprostszym przypadku możemy używać Haskella podobnie jak kalkulatora.

przykłady

- > 5*4+3
- > sqrt 4
- > pi^2*8

Jakie to proste!

Naszym pierwszym programem będzie standardowe wypisanie tekstu "Hello Worldńa ekranie. W tym celu w lini poleceń należy wpisać polecenie **putStr "Hello World"**
W wyniku otrzymamy:

```
> putStr "Hello World"
Hello World
(20 reductions, 41 cells)
>
```

Pierwsza linia jest to oczywiście wpisane przez nas polecenie. Druga linia jest wynikiem działania programu. Ostatnia linia dostarcza informacji o oszacowaniu polecenia. Interpreter zastępuje funkcje łatwiejszymi funkcjami dopóki każda część nie będzie prostym krokiem w bezpośredniej implementacji.

Operatory matematyczne i logiczne

Poniżej przedstawione są dostępne operatory matematyczno logiczne:

Operator	Działanie
+	dodawanie
-	odejmowanie
*	mnożenie
/	dzielenie
^	potęgowanie
&&	AND
||	OR
<	mniejszy niż
<=	mniejszy lyb równy
>	większy
>=	większy lub równy
==	równy
/=	różny

Operatory działające na listach

Wśród operatorów działających na listach mozemy wyróżnić:

Operator	**Działanie**
++	konkatenacja list
:	dodanie elementu (głowy) do listy, "cons" operator
!!	operator indeksowania
..	specyfikacja zasięgu listy

Typy danych

Ponieważ Haskell jest językiem czysto funkcyjnym, wszelkie obliczenia są dokonywane poprzez ewaluowanie wyrażeń (członów syntaktycznych), w celu dostarczenia wartości. Każda wartość ma przypisany typ.
Tak jak wyrażenia wskazują na wartości, wyrażenia typujące (ang. type expressions) są syntaktycznymi członami, które wskazują na typy.

Aby sprawdzić typ danego wyrażenia posługujemy się poleceniem interpretera **:t**

Przykładowa deklaracja typów

Symbol **::** możemy odczytać jako "jest typu"

- > 2 :: Integer
- > 2.3 :: Double
- > True :: Bool
- > 'b' :: Char
- > 'Czesc' :: String

Podstawowe typy danych

Typy całkowite

- Int (fixed-precision) - liczby całkowite z zakresu
- Integer (arbitrary-precision) - wartością Integer może być dowolna liczba całkowita (zarówno ujemna jak i dodatnia).

Typy rzeczywiste

- Float - liczba zmiennoprzecinkowa pojedynczej precyzji,
- Double - liczba zmiennoprzecinkowa podwójnej precyzji.

Typ znakowy

- Char - typ pojedynczego znaku. Należy pamiętać, że pojedyncze znaki zapisuje się w apostrofach
- String - łańcuch znakowy. Łańcuchy znakowe zapisuje się w cudzysłowach.

Podstawowe typy danych (2)

Typ logiczny

- Bool - reprezentuje zmienne logiczne. Jest to typ wyliczeniowy zawierający dwie wartości False (fałsz - 0) i True (prawda - 1).

Typy relacji między elementami

Typ wyliczeniowy posiadający trzy wartości:

- LT (less then - mniejszy niż)
- EQ (equal - równy)
- GT (greater then - większy niż)

Wartości tego typu są zwracane między innymi przez funkcję compare porównującą dwa elementy

Przykład

```
> compare 2 4
LT
```

Typy strukturalne

Listy

Listy to struktury homogeniczne. Rozmiar listy nie jest określony - można dołączać do niej kolejne elementy.

np.

```
> :t ['a','b','c','d','e'] ['a','b','c','d','e'] :: [Char]
```

Krotki

Krotki to struktury heterogeniczne. Rozmiar krotki jest ściśle określony podczas jej tworzenia. Nie jest możliwe dołączanie elementów do istniejącej krotki.

np.

```
> :t (True,"Haskell",'a',1::Integer)
(True,"Haskell",'a',1) :: (Bool,[Char],Char,Integer)
```

Typy funkcji

Nie tylko zmienne, ale również funkcje mają w Haskellu swój typ. Na typ funkcji składają się typy przyjmowanych przez nią parametrów oraz typ wartości zwracanej przez funkcję. Typy te podajemy w następujący sposób:
nazwa_funkcji :: TypParam_1 -> TypParam_2 -> ... -> TypParam_n -> TypWartosciZwracanej

przykład

```
inkrementacja :: Int -> Int
dodajTrzyLiczby :: Double -> Double -> Double -> Double
```

Funkcje - podstawy

Deklaracja funkcji. Zajmijmy się funkcją dodającą trzy liczby, której typ już określiliśmy. Cała funkcja będzie prezentować się następująco

Dodawanie 3 liczb

```
dodajTrzyLiczby :: Double -> Double -> Double -> Double
dodajTrzyLiczby x y z = x + y + z
```

Listy

Są to homogeniczne struktury danych. Lista pozwala na przechowywanie dowolnej liczby elementów tego samego typu. Listy są otoczone nawiasami kwadratowymi, a ich elementy oddzielone przecinkami.

przykład

> [] - - pusta lista
> [1,2,3,4,5,6,7,8,9] - - lista numeryczna
> [1,2,3,4,5,6,7,8,9::Integer] - - lista całkowitoliczbowa
> ['a','b','c','d'] - - lista znaków
> ["Kasia","Tomek","Adam"] - - lista stingów

Przejdźmy teraz do operacji wykonywanych na listach.

Listy - operacje

head

Funkcja head wypisuje głowę, czyli pierwszy element listy. Przykład: // > head [1,2,3,4,5,6,7,8,9] // 1

tail

UWAGA! Funkcja tail nie wypisuje ostatniego elementu listy! Wypisuje wszystkie elementy, poza głową. // > tail [1,2,3,4,5,6,7,8,9] // [2,3,4,5,6,7,8,9]

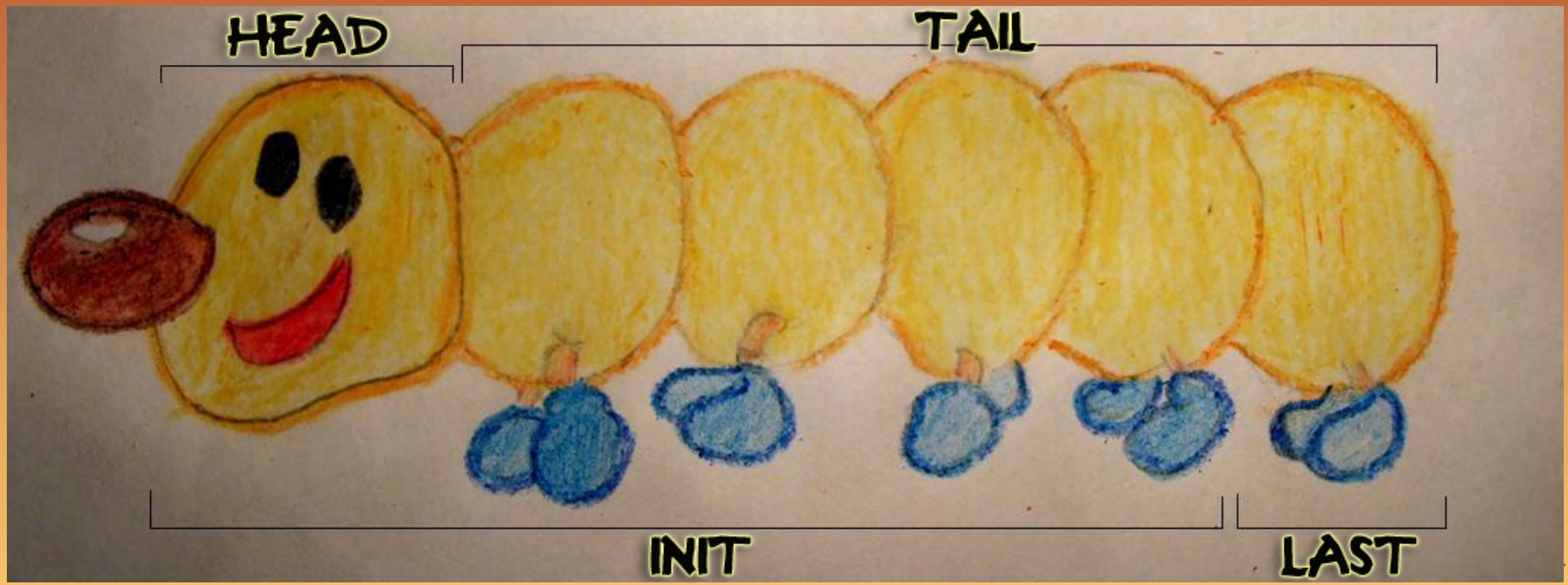
last

Wypisuje ostatni element listy. // > last [1,2,3,4,5,6,7,8,9] // 9

init

Wypisuje wszystkie elementy, prócz ostatniego. // > init [1,2,3,4,5,6,7,8,9] // [1,2,3,4,5,6,7,8]

Listy



Silnia

Program w języku Haskell

```
silnia 0 = 1 - - pierwszy krok
silnia n = n * silnia (n-1)
```

lub

```
silnia2 n = if n == 0 then 1 else n *
silnia2 (n-1)
```

Program w języku C

```
int silnia(int n)
{
if(n==0) return 1;
return n * silnia(n-1);
}
```

Fibonacci

Program w języku Haskell

```
module Main where
- - Rekurencja binarna
f n |n < 2 = 1
|n >= 2 = f (n-1) + f (n-2)
f_print n = print(show n ++
"Liczby Fibonacciego to " ++ show
(f n))
main = f_print 46
```

Program w języku C

```
int fib (int n) {
int a = 0, b = 1, i, temp;
for (i = 0; i < n; i++) {
temp = a + b;
a = b;
b = temp;
}
return a;
}
```

Quicksort

Program w języku Haskell

```
qsort [] = []
qsort (x:xs) = qsort (filter (< x) xs)
++ [x] ++ qsort (filter (>= x) xs)
```

Rozumowanie

Rezultatem sortowania pustej listy, będzie również pusta lista.

Aby posortować listę, której pierwszy element oznaczono jako x, a pozostałą część jako xs, należy najpierw posortować elementy, które

Program w języku C

```
void qsort(int a[], int lo, int hi) {
{ int h, l, p, t;
if (lo < hi) {
l = lo;
h = hi;
p = a[hi];
do {
while ((l < h) && (a[l] <= p))
l = l+1;
while ((h > l) && (a[h] >= p))
h = h-1;
if (l < h) {
t = a[l];
a[l] = a[h];
a[h] = t;
```

Bibliografia

- http://www.haskell.org/
- http://darcs.haskell.org/yaht/yaht.pdf
- http://gnosis.cx/publish/programming/Haskell.pdf
- http://learnyouahaskell.com/
- http://pl.wikibooks.org/wiki/Haskell

Koniec

To już jest koniec.

Serdecznie dziękuję za uwagę.