Programare declarativă[1]

Intrare/Ieșire

Ioana Leuștean
Traian Florin Șerbănuță

Departamentul de Informatică, FMI, UNIBUC
traian.serbanuta@unibuc.ro

[1] bazat pe cursul Informatics 1: Functional Programming de la University of Edinburgh

Despre intenție și acțiune

Mind-Body Problem — Comandă vs. Execuție

Care e legătura dintre intenție și acțiune, dintre percepție și înțelegere?

- [1] A purely functional program implements a function; it has no side effect.

[1] S. Peyton-Jones, Tackling the Awkward Squad: ...

Mind-Body Problem — Comandă vs. Execuție

Care e legătura dintre intenție și acțiune, dintre percepție și înțelegere?

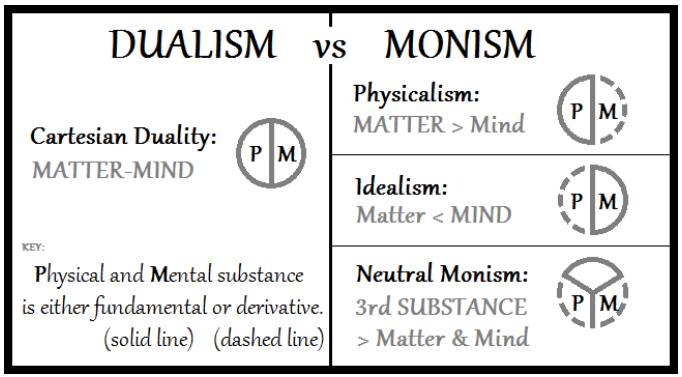
- [1] A purely functional program implements a function; it has no side effect.
- [1] Yet the ultimate purpose of running a program is invariably to cause some side effect: a changed file, some new pixels on the screen, a message sent, ...

[1] S. Peyton-Jones, Tackling the Awkward Squad: ...

Mind-Body Problem — Comandă vs. Execuție

Care e legătura dintre intenție și acțiune, dintre percepție și înțelegere?

- [1] A purely functional program implements a function; it has no side effect.
- [1] Yet the ultimate purpose of running a program is invariably to cause some side effect: a changed file, some new pixels on the screen, a message sent, ...
- [2] Interaction is the mind-body problem of computing.

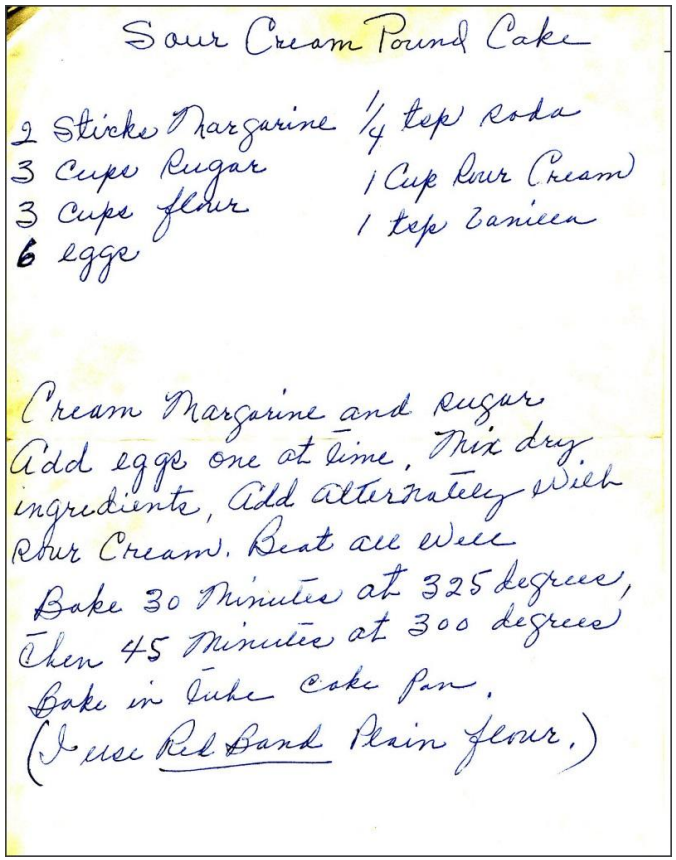


[1] S. Peyton-Jones, Tackling the Awkward Squad: ...

[2] P. Wadler, How to Declare an Imperative

Mind-Body Problem

Rețetă vs Prăjitură



Sour Cream Pound Cake

2 Sticks Margarine
3 Cups Sugar
3 Cups flour
6 eggs
1/4 tsp soda
1 Cup Sour Cream
1 tsp Vanilla

Cream Margarine and sugar
Add eggs one at time, Mix dry ingredients, Add alternately with Sour Cream. Beat all well
Bake 30 Minutes at 325 degrees,
Then 45 Minutes at 300 degrees
Bake in Tube Cake Pan.
(I use Red Band Plain flour.)

http://www.seas.upenn.edu/~cis194/fall16/lectures/06-io-and-monads.html

Comenzi în Haskell

Comanda: afișează un caracter!

```
putChar :: Char -> IO ()
```

Exemplu

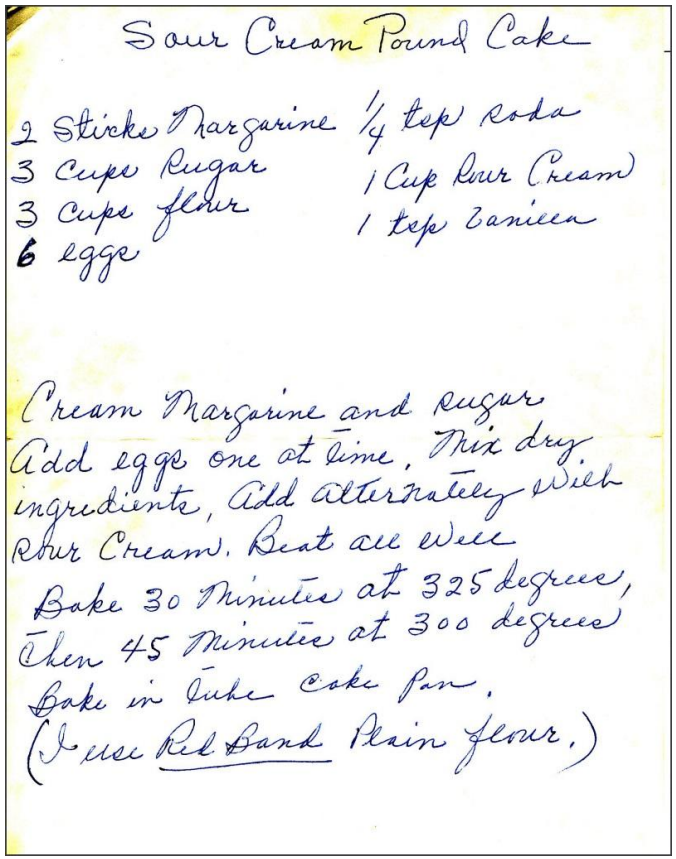
```
putChar '!'
```

reprezintă o comandă care, dacă va fi executată, va afișa un semn de exclamare.

Mind-Body Problem - Rețetă vs Prăjitură

http://www.seas.upenn.edu/~cis194/fall16/lectures/06-io-and-monads.html

c :: Cake

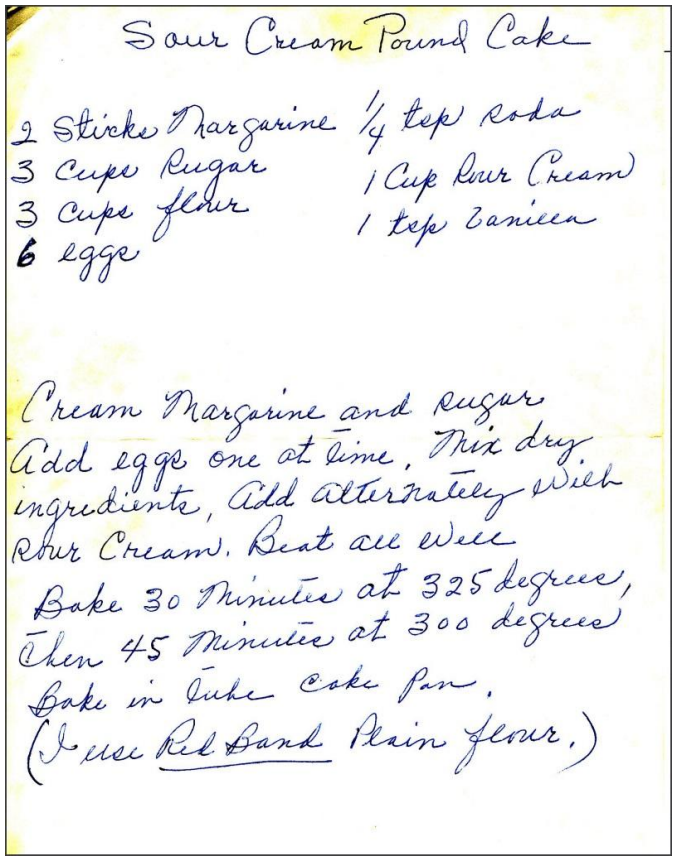


r :: Recipe Cake

Mind-Body Problem - Rețetă vs Prăjitură

http://www.seas.upenn.edu/~cis194/fall16/lectures/06-io-and-monads.html

c :: Cake


Sour Cream Pound Cake

2 Sticks Margarine
3 Cups Sugar
3 Cups flour
6 eggs
1/4 tsp Soda
1 Cup Sour Cream
1 tsp Vanilla

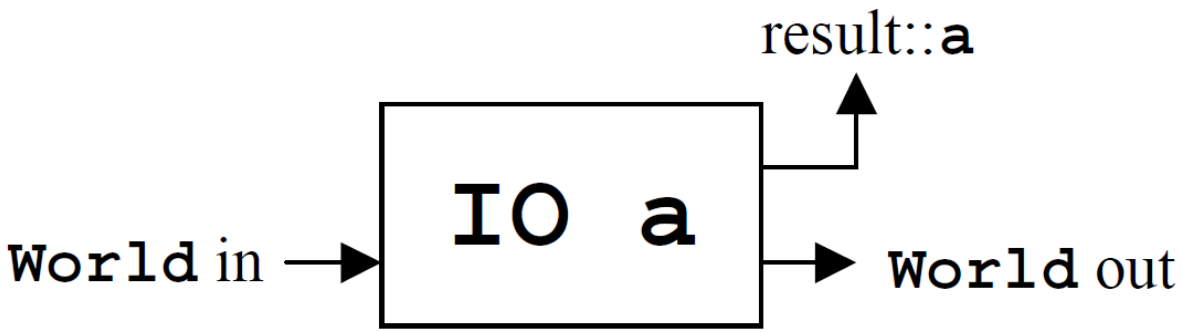
Cream Margarine and Sugar
Add eggs one at time, Mix dry ingredients, Add alternately with Sour Cream. Beat all well
Bake 30 Minutes at 325 degrees, Then 45 Minutes at 300 degrees
Bake in Tube Cake Pan.
(I use Red Band Plain flour.)

r :: Recipe Cake

IO este o rețetă care produce o valoare de tip c

Comenzi în Haskell

```
type IO a = RealWorld -> (a, RealWorld)
```



S. Peyton-Jones, Tackling the Awkward Squad: ...

Combină două comenzi!

```
(>>) :: IO () -> IO () -> IO ()
putChar :: Char -> IO ()
```

Exemplu

```
putChar '?' >> putChar '!'
```

rerpezintă o comandă care, dacă va fi executată, va afișa un semn de întrebare urmat de un semn de exclamare .

Afișează un șir de caractere

```
putStr :: String -> IO ()
putStr []     = done
putStr (x:xs) = putChar x >> putStr xs
```

Observație:

```
done :: IO ()
```

reprezintă o comandă care, dacă va fi executată, nu va face nimic.

Exemplu

```
putStr "?!" == putChar '?' >> (putChar '!' >> done)
```

rerpezintă o comandă care, dacă va fi executată, va afișa un semn de întrebare urmat de un semn de exclamare.

putStr folosind funcționale

```
putStr      :: String -> IO ()
putStr      = foldr (>>) done . map putChar
```

Afișează și treci pe rândul următor

```
putStrLn :: String -> IO ()
putStrLn xs = putStr xs >> putChar '\n'
```

(**IO** (), (>>), done) e monoid

```
m >> done          =  m
done >> m          =  m
(m >> n) >> o      =  m >> (n >> o)
```

Și totuși, când sunt executate comenzile?

main

Orice comandă **IO a** poate fi executată în interpretor, dar

Programele Haskell pot fi compilate

Fișierul scrie.hs:

```
main :: IO ()
main = putStrLn "?!"
```

```
08-io$ ghc scrie.hs
[1 of 1] Compiling Main  (scrie.hs, scrie.o)
Linking scrie.exe ...
08-io$ ./scrie
?!
```

Funcția executată este **main**

Când sunt executate comenzile?

Fișierul PutStr.hs

```
module PutStr where

main :: IO ()
main = putStr "?!"
```

Rularea programului are ca efect executarea comenzii specificate de main:

```
08-io$ runghc PutStr.hs
?!08-io$
```

Observație:

runghc rulează programul fără a-l compila inainte

Validitatea raționamentelor

Raționamentele substitutive își pierd valabilitatea

În limbaje cu efecte laterale

Program 1

```
int main() { cout << "HA!"; cout << "HA!"; }
```

Program 2

```
void dup(auto& x) { x ; x; }
int main() { dup(cout << "HA!"); }
```

Raționamentele substitutive își pierd valabilitatea

În limbaje cu efecte laterale

Program 1

```
int main() { cout << "HA!"; cout << "HA!"; }
```

Program 2

```
void dup(auto& x) { x ; x; }
int main() { dup(cout << "HA!"); }
```

Program 3

```
void dup(auto x) { x() ; x(); }
int main() { dup( []() { cout << "HA!"; } ); }
```

Raționamentele substitutive sunt valabile

În Haskell

Expresii

```
(1+2) * (1+2)
```

este echivalentă cu expresia

```
let x = 1+2 in x * x
```

și se evaluează amândouă la 9

Comenzi

```
putStr "HA!" >> putStr "HA!"
```

este echivalentă cu

```
let m = putStr "HA!" in m >> m
```

si amândouă afișează "HA!HA!".

Raționamentele substitutive sunt valabile

https://en.wikibooks.org/wiki/Haskell/Prologue:_IO,_an_applicative_functor

Referential transparency

orice expresie poate fi înlocuită cu valoare ei

```
addExclamation :: String -> String
addExclamation s = s ++ "!"

main = putStrLn (addExclamation "Hello")
Prelude> main
Hello!

main = putStrLn ("Hello" ++ "!")
Prelude> main
Hello!
```

Raționamentele substitutive sunt valabile

https://en.wikibooks.org/wiki/Haskell/Prologue:_IO,_an_applicative_functor

```
addExclamation :: String -> String
addExclamation s = s ++ "!"
```

Observație

Dacă **getLine** ar avea tipul **String** atunci am putea scrie

```
main = putStrLn (addExclamation getLine)  -- cod eronat !!!
```

Raționamentele substitutive sunt valabile

https://en.wikibooks.org/wiki/Haskell/Prologue:_IO,_an_applicative_functor

```
addExclamation :: String -> String
addExclamation s = s ++ "!"
```

Observație

Dacă **getLine** ar avea tipul **String** atunci am putea scrie

```
main = putStrLn (addExclamation getLine)  -- cod eronat !!!
```

Nu putem înlocui **getLine** cu valoarea ei!

Raționamentele substitutive sunt valabile

https://en.wikibooks.org/wiki/Haskell/Prologue:_IO,_an_applicative_functor

```
addExclamation :: String -> String
addExclamation s = s ++ "!"
```

Observație

Dacă **getLine** ar avea tipul **String** atunci am putea scrie

```
main = putStrLn (addExclamation getLine)  -- cod eronat !!!
```

Nu putem înlocui **getLine** cu valoarea ei!

Soluția: **getLine** are tipul **IO String**

Comenzi cu valori

Comenzi cu valori

- **IO** () corespunde comenzilor care nu produc rezultate
  - () este tipul unitate care conține doar valoarea ()

- În general, **IO** a corespunde comenzilor care produc rezultate de tip a.
  - **IO Char** corespunde comenzilor care produc rezultate de tip **Char**

Citește un caracter!

getChar :: IO Char

- Dacă „șirul de intrare” conține "abc"
- atunci **getChar** produce:
  - 'a'
  - șirul rămas de intrare "bc"

Produu o valoare fără să faci nimic!

Din pălărie

```
return :: a -> IO a
```

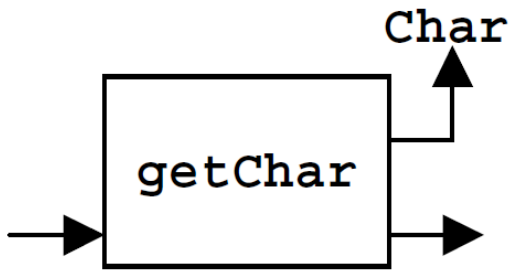
Asemănatoar cu done, nu face nimic, dar produce o valoare.

Exemplu

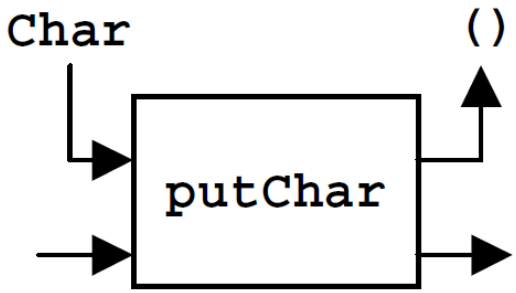
```
return ""
```

- Dacă „șirul de intrare” conține "abc"
- atunci **return** "" produce:
  - valoarea ""
  - șirul (neschimbat) de intrare "abc"

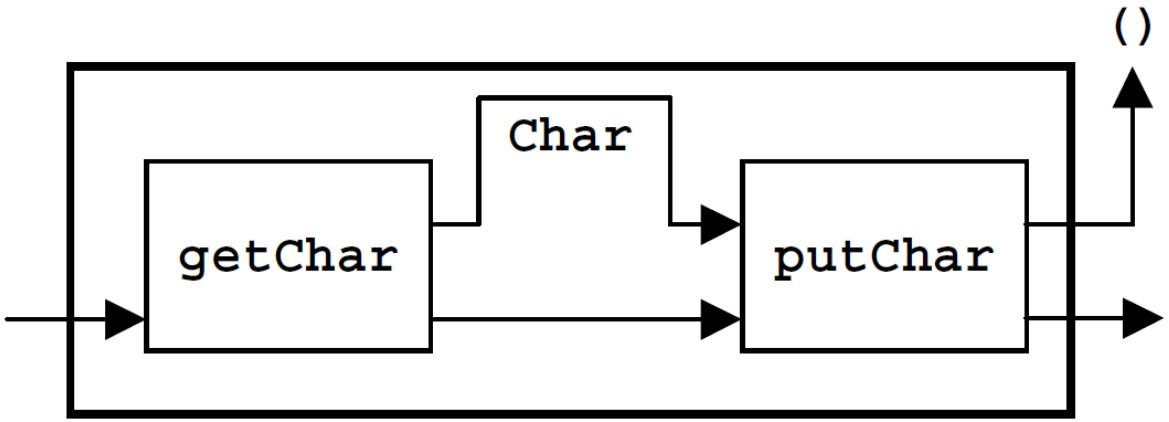
Operatorul de legare




```
getChar :: IO Char
```




```
putChar :: Char -> IO ()
```




```
getChar >>= putChar
```

getChar $>>=$ **putChar**

S. Peyton-Jones, Tackling the Awkward Squad: ...

Combinarea comenzilor cu valori

Operatorul de legare / bind

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Exemplu

```
getChar >>= \x -> putChar (toUpper x)
```

- Dacă „șirul de intrare” conține "abc"
- atunci comanda de mai sus, atunci când se execută, produce:
  - ieșirea "A"
  - șirul rămas de intrare "bc"

Operatorul de legare / bind

Mai multe detalii

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

- Dacă fiind o comandă care produce o valoare de tip a
  m :: **IO** a
- Data fiind o funcție care pentru o valoare de tip a se evaluează la o comandă de tip b
  k :: a -> **IO** b
- Atunci
  m >>= k :: **IO** b
  este comanda care, dacă se va executa:
  - Mai întâi efectuează m, obținând valoarea x de tip a
  - Apoi efectuează comanda k x obținând o valoare y de tip b
  - Produce y ca rezultat al comenzii

Citește o linie!

```
getLine :: IO String
getLine = getChar >>= \x ->
            if x == '\n' then
              return []
            else
              getLine >>= \xs ->
              return (x:xs)
```

Exemplu

Dat fiind șirul de intrare "abc\ndef", getLine produce șirul "abc" și șirul rămas de intrare e "def"

Comenzile sunt cazuri speciale de comenzi cu valori

done e caz special de return

```
done        :: IO ()
done        = return ()
```

>> e caz special de >>=

```
(>>)        :: IO () -> IO () -> IO ()
m >> n      = m >>= \ () -> n
```

Operatorul de legare e similar cu **let**

Operatorul **let**

```
let x = m in n
```

let ca aplicație de funcții

```
(\ x -> n) m
```

Operatorul de legare

```
m >>= \ x -> n
```

De la intrare la ieșire

```
echo :: IO ()
echo = getLine >>= \line ->
       if line == "" then
         return ()
       else
         putStrLn (map toUpper line) >>
         echo

main :: IO ()
main = echo
```

De la intrare la ieșire

```
echo :: IO ()
echo = getLine >>= \line ->
       if line == "" then
         return ()
       else
         putStrLn (map toUpper line) >>
         echo

main :: IO ()
main = echo
```

Test

```
$ runghc Echo.hs
One line
ONE LINE
And, another line!
AND, ANOTHER LINE!
```

Notația do

Citirea unei linii în notație „do”

```
getLine :: IO String
getLine = getChar >>= \x ->
            if x == '\n' then
              return []
            else
              getLine >>= \xs ->
              return (x:xs)
```

Echivalent cu:

```
getLine :: IO String
getLine = do {
            x <- getChar;
            if x == '\n' then
              return []
            else do {
              xs <- getLine;
              return (x:xs)
            }
          }
```

Echo în notația „do”

```
echo :: IO ()
echo = getLine >>= \line ->
       if line == "" then
         return ()
       else
         putStrLn (map toUpper line) >>
         echo
```

Echivalent cu

```
echo :: IO ()
echo = do {
         line <- getLine;
         if line == "" then
           return ()
         else do {
           putStrLn (map toUpper line);
           echo
         }
       }
```

Notația „do” în general

- Fiecare linie x <– e; ... devine e >>= \x –> ...
- Fiecare linie e; ... devine e >> ...

De exemplu

```
do { x1 <– e1;
     x2 <– e2;
     e3;
     x4 <– e4;
     e5;
     e6 }
```

e echivalent cu

```
e1   >>= \x1 –>
e2   >>= \x2 –>
e3   >>
e4   >>= \x4 –>
e5   >>
e6
```

Citire/Scriere din fișiere

Citirea/ Scrierea din fișiere

Operații de bază

```
type FilePath = String

readFile :: FilePath -> IO String

writeFile :: FilePath -> String -> IO ()

appendFile :: FilePath -> String -> IO ()
```

```
echof = do
        s <- readFile "fis1.txt"
        putStrLn s
        writeFile "fis2.txt" s
```

Citirea/ Scrierea din fișiere

Operații de bază

```
import Data.Char(toUpper)

main = do
       s <- readFile "Input.txt"
       putStrLn $ "Intrare\n" ++ s

       let sprel = map toUpper s  -- prelucrare date citite

       putStrLn $ "Iesire\n" ++ sprel
       writeFile "Output.txt" sprel   -- appendFile
```

Observații:

- **readFile** citește (leneș) conținutul fișierului
- **writeFile** și **appendFile** crează fișierele dacă acestea nu există.

Citirea/ Scrierea numerelor din fișiere

Operații de bază

```
listNo str = concat $ map words $ lines str

readNumbers file1 file2 = do
                  str <- readFile file1
                  putStrLn $ "Intrare\n"
                  let numbers = (map read $ listNo s) :: [Int]
                  print numbers
                  writeFile file2 (show numbers)

> readNumbers "fio.txt"  "fout.txt"
```

Observații:

- **print** este **putStrLn . show**
- putem scrie in fisier numai date care sunt instanta a clasei **Show**

??????

https://crypto.stanford.edu/~blynn/haskell/

Temă pentru vacanță

Temă pentru vacanță

Simon Peyton Jones — Haskell is Useless

http://www.youtube.com/watch?v=iSmkqocn0oQ

Temă pentru vacanță

Simon Peyton Jones — Haskell is Useless

http://www.youtube.com/watch?v=iSmkqocn0oQ

Atenție! Înregistrarea este din 2011.