Curs 1

2020-2021

Fundamentele Limbajelor de Programare

Cuprins

Organizare

Instructori

Curs:

- **Ioana Leuștean (seria 24), Traian-Florin Șerbănuță (seria 23)**

Laborator

Seria 24
- **Ioana Leuștean (241, 244)**
- **Natalia Ozunu (242, 243)**

Seria 23
- **Ana Țurlea (231, 232, 233)**
- **Traian Șerbănuță (234)**

Suport curs/seminar/laborator

Seria 24
- https://cs.unibuc.ro//~ileustean/FLP.html
- Moodle: https://moodle.unibuc.ro/course/view.php?id=4635
- Materiale Curs/Laborator: https://bit.ly/3de0SOF

Seria 23
- Materiale Curs/Laborator: http://bit.do/unibuc-flp
- Moodle (teste, note): https://moodle.unibuc.ro/course/view.php?id=4634

O parte din materiale sunt realizate în colaborare cu Denisa Diaconescu.

Notare

- **Testare parțială: 40 puncte**
- **Testare finală: 50 puncte**
- Se acordă 10 puncte din oficiu!

Notare

- **Testare parțială: 40 puncte**
- **Testare finală: 50 puncte**
- Se acordă 10 puncte din oficiu!

- Condiţie minimă pentru promovare:
  testare parțială: minim 20 puncte şi
  testare finală: minim 20 puncte.

Notare

- **Testare parțială: 40 puncte**
- **Testare finală: 50 puncte**
- Se acordă 10 puncte din oficiu!

- Condiţie minimă pentru promovare:
  testare parțială: minim 20 puncte şi
  testare finală: minim 20 puncte.

- Se poate obţine punctaj suplimentar pentru activitatea din timpul laboratorului:
  maxim 10 puncte.

Testare parțială: 40 puncte

- Data: 23 aprilie
- Timp de lucru: 1,5 ore
- Prezenţa este obligatorie pentru a putea promova!
- Pentru a trece această probă, trebuie să obţineţi minim 20 de puncte.

Testare finală: 50 puncte

- Data: În sesiune
- Timp de lucru: 2 ore
- Prezenţa este obligatorie pentru a putea promova!
- Pentru a trece această probă, trebuie să obţineţi minim 20 de puncte.

Curs/seminar/laborator

- Curs

  Semantica limbajelor de programare
  - Parsare, Verificarea tipurilor și Interpretare
  - Semantică operațională, statică și axiomatică
  - Inferarea automată a tipurilor

  Bazele programării funcționale
  - Lambda Calcul, Codificări Church, combinatori
  - Lambda Calcul cu tipuri de date

  Bazele programării logice
  - Logica clauzelor Horn, Unificare, Rezoluţie

- Laborator:

  Haskell Limbaj pur de programare funcțională
  - Interpretoare pentru mini-limbaje

  Prolog Cel mai cunoscut limbaj de programare logică
  - Verificator pentru un mini-limbaj imperativ
  - Inferența tipurilor pentru un mini-limbaj funcțional

Bibliografie

- B.C. Pierce, **Types and programming languages**. MIT Press.2002
- G. Winskel, **The formal semantics of programming languages**. MIT Press. 1993
- H. Barendregt, E. Barendsen, **Introduction to Lambda Calculus**, 2000.
- J. Lloyd. **Foundations of Logic Programming**, second edition. Springer, 1987.
- P. Blackburn, J. Bos, and K. Striegnitz, **Learn Prolog Now!** (Texts in Computing, Vol. 7),College Publications, 2006
- M. Huth, M. Ryan, **Logic in Computer Science (Modelling and Reasoning about Systems)**, Cambridge University Press, 2004.

Privire de ansamblu

Semantica Limbajelor de Programare

Ce definește un limbaj de programare?

Sintaxa Simboluri de operație, cuvinte cheie, descriere (formală) a programelor/expresiilor bine formate

Practica Un limbaj e definit de modul cum poate fi folosit

- Manual de utilizare și exemple de bune practici
- Implementare (compilator/interpretor)
- Instrumente ajutătoare (analizor de sintaxă, verificator de tipuri, depanator)

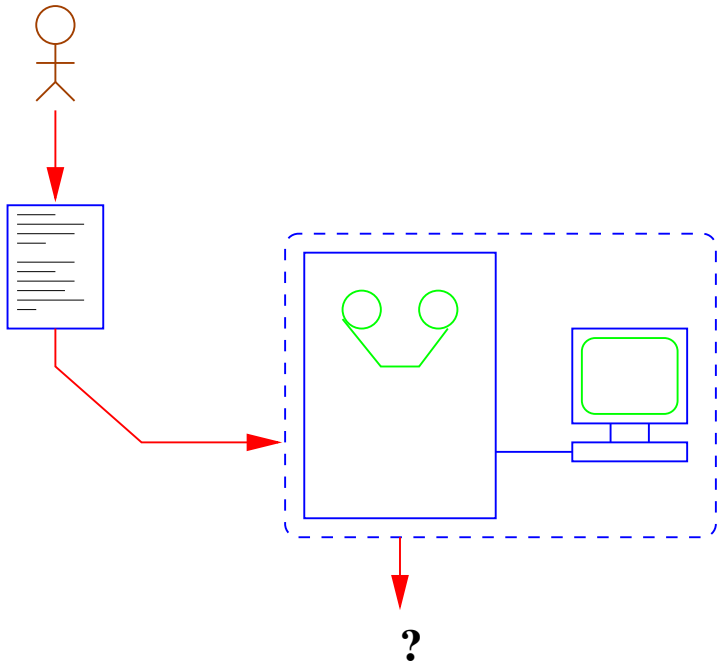
Semantica? Ce înseamnă / care e comportamentul unei instrucțiuni?

- De cele mai multe ori se dă din umeri și se spune că Practica e suficientă
- Limbajele mai utilizate sunt standardizate

La ce folosește semantica

- Să înțelegem un limbaj în profunzime
  - Ca programator: pe ce mă pot baza când programez în limbajul dat
  - Ca implementator al limbajului: ce garanții trebuie să ofer
- Ca instrument în proiectarea unui nou limbaj / a unei extensii
  - Înțelegerea componentelor și a relațiilor dintre ele
  - Exprimarea (și motivarea) deciziilor de proiectare
  - Demonstrarea unor proprietăți generice ale limbajului
    E.g., execuția nu se va bloca pentru programe care trec de analiza tipurilor
- Ca bază pentru demonstrarea corectitudinii programelor.

Problema corectitudinii programelor



- Pentru anumite metode de programare (e.g., imperativă, orientată pe obiecte), nu este uşor să stabilim că un program este corect sau să înţelegem ce înseamnă că este corect (e.g, în raport cu ce?!).
- Corectitudinea programelor devine o problemă din ce în ce mai importantă, nu doar pentru aplicaţii "safety-critical".
- Avem nevoie de metode ce asigură "calitate", capabile să ofere "garanţii".

C

```
#include <iostream>
using namespace std;
int main()
{
  int square;
  for(int i = 1; i <= 5; ++i)
  {
    square = i * i;
    cout <<  square << endl;
  }
}
```

C

```
#include <iostream>
using namespace std;
int main()
{
  int square;
  for(int i = 1; i <= 5; ++i)
  {
    square = i * i;
    cout <<  square << endl;
  }
}
```

- Este corect?

C

```
#include <iostream>
using namespace std;
int main()
{
  int square;
  for(int i = 1; i <= 5; ++i)
  {
    square = i * i;
    cout <<  square << endl;
  }
}
```

- Este corect? În raport cu ce?

C

```
#include <iostream>
using namespace std;
int main()
{
  int square;
  for(int i = 1; i <= 5; ++i)
  {
    square = i * i;
    cout <<  square << endl;
  }
}
```

- Este corect? În raport cu ce?
- Un formalism adecvat trebuie:
  - să permită descrierea problemelor (specificaţii), şi
  - să raţioneze despre implementarea lor (corectitudinea programelor).

Care este comportamentul corect?

```
int main(void) {
  int x = 0;
  return (x = 1) + (x = 2);
}
```

Care este comportamentul corect?

```
int main(void) {
  int x = 0;
  return (x = 1) + (x = 2);
}
```

Conform standardului C, comportamentul programului este nedefinit.

- GCC4, MSVC: valoarea întoarsă e 4
- GCC3, ICC, Clang: valoarea întoarsă e 3

Care este comportamentul corect?

```
int r;
int f(int x) {
  return (r = x);
}
int main() {
  return f(1) + f(2), r;
}
```

Care este comportamentul corect?

```
int r;
int f(int x) {
  return (r = x);
}
int main() {
  return f(1) + f(2), r;
}
```

Conform standardului C, comportamentul programului este corect, dar subspecificat:
poate întoarce atât valoarea 1 cât și 2.

Tipuri de semantică

Semantica dă "înţeles" unui program.

Tipuri de semantică

Semantica dă "înţeles" unui program.

- Operaţională:
  - Înţelesul programului este definit în funcţie de paşii (transformări dintr-o stare în alta) care apar în timpul execuţiei.

Tipuri de semantică

Semantica dă "înţeles" unui program.

- Operaţională:
  - Înţelesul programului este definit în funcţie de paşii (transformări dintr-o stare în alta) care apar în timpul execuţiei.
- Denotaţională:
  - Înţelesul programului este definit abstract ca element dintr-o structură matematică adecvată.

Tipuri de semantică

Semantica dă "înţeles" unui program.

- Operaţională:
  - Înţelesul programului este definit în funcţie de paşii (transformări dintr-o stare în alta) care apar în timpul execuţiei.
- Denotaţională:
  - Înţelesul programului este definit abstract ca element dintr-o structură matematică adecvată.
- Axiomatică:
  - Înţelesul programului este definit indirect în funcţie de axiomele şi regulile pe care le verifică.

Tipuri de semantică

Semantica dă "înţeles" unui program.

- Operaţională:
  - Înţelesul programului este definit în funcţie de paşii (transformări dintr-o stare în alta) care apar în timpul execuţiei.
- Denotaţională:
  - Înţelesul programului este definit abstract ca element dintr-o structură matematică adecvată.
- Axiomatică:
  - Înţelesul programului este definit indirect în funcţie de axiomele şi regulile pe care le verifică.
- Statică / a tipurilor
  - Reguli de bună-formare pentru programe
  - Oferă garanții privind execuția (e.g., nu se blochează)

Bazele programării funcționale / logice

Principalele paradigme de programare

- Imperativă (cum calculăm)
  - Procedurală
  - Orientată pe obiecte
- Declarativă (ce calculăm)
  - Logică
  - Funcţională

Fundamentele paradigmelor de programare

Imperativă	Execuția unei Mașini Turing
Funcțională	Beta-reducție în Lambda Calcul
Logică	Rezoluția în logica clauzelor Horn

Programare declarativă

- Programatorul spune ce vrea să calculeze, dar nu specifică concret cum calculează.
- Este treaba interpretorului (compilator/implementare) să identifice cum să efectueze calculul respectiv.
- Tipuri de programare declarativă:
  - Programare funcţională (e.g., Haskell)
  - Programare logică (e.g., Prolog)
  - Limbaje de interogare (e.g., SQL)

Programare funcțională

Esență: funcții care relaționează intrările cu ieșirile

Caracteristici:
- funcții de ordin înalt – funcții parametrizate de funcții
- grad înalt de abstractizare (e.g., functori, monade)
- grad înalt de reutilizarea codului — polimorfism

Fundamente:
- Teoria funcțiilor recursive
- Lambda-calcul ca model de computabilitate (echivalent cu mașina Turing)

Inspirație:
- Inferența tipurilor pentru templates/generics in POO
- Model pentru programarea distribuită/bazată pe evenimente (callbacks)

Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.

Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.
- Unul din sloganurile programării logice:

  Program = Logică + Control (R. Kowalski)

Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.
- Unul din sloganurile programării logice:

  Program = Logică + Control (R. Kowalski)

- Programarea logică poate fi privită ca o deducţie controlată.

Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.
- Unul din sloganurile programării logice:

  Program = Logică + Control (R. Kowalski)

- Programarea logică poate fi privită ca o deducţie controlată.
- Un program scris într-un limbaj de programare logică este

  o listă de formule într-o logică

  ce exprimă fapte şi reguli despre o problemă.

Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.
- Unul din sloganurile programării logice:

  Program = Logică + Control (R. Kowalski)

- Programarea logică poate fi privită ca o deducţie controlată.
- Un program scris într-un limbaj de programare logică este

  o listă de formule într-o logică

  ce exprimă fapte şi reguli despre o problemă.
- Exemple de limbaje de programare logică:
  - Prolog
  - Answer set programming (ASP)
  - Datalog

Programare logică & Prolog

Programare logică - în mod idealist

- Un "program logic" este o colecţie de proprietăţi presupuse (sub formă de formule logice) despre lume (sau mai degrabă despre lumea programului).
- Programatorul furnizează şi o proprietate (o formula logică) care poate să fie sau nu adevărată în lumea respectivă (întrebare, query).
- Sistemul determină dacă proprietatea aflată sub semnul întrebării este o consecinţă a proprietăţilor presupuse în program.
- Programatorul nu specifică metoda prin care sistemul verifică dacă întrebarea este sau nu consecinţă a programului.

Exemplu de program logic

$$
\begin{aligned}
\texttt{oslo} &\rightarrow \texttt{windy} \\
\texttt{oslo} &\rightarrow \texttt{norway} \\
\texttt{norway} &\rightarrow \texttt{cold} \\
\texttt{cold} \wedge \texttt{windy} &\rightarrow \texttt{winterIsComing} \\
&\qquad\quad \texttt{oslo}
\end{aligned}
$$

Exemplu de program logic

$$
\begin{aligned}
\texttt{oslo} &\rightarrow \texttt{windy} \\
\texttt{oslo} &\rightarrow \texttt{norway} \\
\texttt{norway} &\rightarrow \texttt{cold} \\
\texttt{cold} \wedge \texttt{windy} &\rightarrow \texttt{winterIsComing} \\
&\rightarrow \texttt{oslo}
\end{aligned}
$$

Exemplu de întrebare

Este adevărat `winterIsComing`?

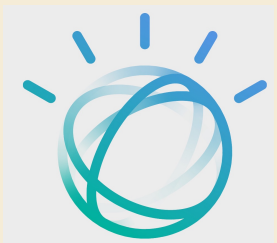
Prolog

- bazat pe logica clauzelor Horn
- semantica operaţională este bazată pe rezoluţie
- este Turing complet

Prolog

- bazat pe logica clauzelor Horn
- semantica operaţională este bazată pe rezoluţie
- este Turing complet

Limbajul Prolog este folosit pentru programarea sistemului IBM Watson!



Puteţi citi mai multe detalii aici.

Exemplul de mai sus în SWI-Prolog

Program:

```
windy :- oslo.
norway :- oslo.
cold :- norway.
winterIsComing :- windy, cold.
oslo.
```

Intrebare:

```
?- winterIsComing.
true
```

http://swish.swi-prolog.org/

Curs 1

2020-2021

Fundamentele Limbajelor de Programare

Cuprins

Haskell: Clasa de tipuri **Monad**

Clasa de tipuri **Monad**

```
class Applicative m => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a - > m b -> m b
    return :: a -> m a

    ma >> mb = ma >>= \_ -> mb
```

- m a este tipul computațiilor care produc rezultate de tip a (și au efecte laterale)
- a -> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= este operația de „secvențiere” a computațiilor

Functor și Applicative definiți cu **return** și >>=

```
instance Monad M where
  return a = ...
  ma >>= k = ...

instance Applicative M where
  pure = return
  mf <*> ma = do
    f <- mf
    a <- ma
    return (f a)
   -- mf >>= (\f -> ma >>= (\a -> return (f a)))

instance Functor F where          -- ma >>= \a -> return (f a)
  fmap f ma = pure f <*> ma       -- ma >>= (return . f)
```

Notația **do** pentru monade

Notația cu operatori	Notația **do**
e >>= \x -> rest	x <- e rest
e >>= _ -> rest	e rest
e >> rest	e rest

Notația **do** pentru monade

Notația cu operatori	Notația **do**
e >>= \x -> rest	x <- e rest
e >>= _ -> rest	e rest
e >> rest	e rest

De exemplu

```
e1    >>= \x1 ->
e2    >> e3
```

devine

Notația **do** pentru monade

Notația cu operatori	Notația **do**
e >>= \x -> rest	x <- e rest
e >>= _ -> rest	e rest
e >> rest	e rest

De exemplu

```
e1    >>= \x1 ->
e2    >> e3
```

devine

```
do
    x1 <- e1
    e2
    e3
```

Haskell: Monade standard

Exemple de efecte laterale

I/O	Monada **IO**
Parțialitate	Monada **Maybe**
Excepții	Monada **Either**
Nedeterminism	Monada [] (listă)
Logging	Monada Writer
Memorie read-only	Monada Reader
Stare	Monada State

Monada **Maybe** (a rezultatelor parțiale)

```
data Maybe a = Nothing | Just a
```

Monada **Maybe** (a rezultatelor parțiale)

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
  return = Just
  Just va  >>= k   = k va
  Nothing >>= _    = Nothing
```

Monada **Maybe** (a rezultatelor parțiale)

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
  return = Just
  Just va  >>= k   = k va
  Nothing >>= _    = Nothing

radical :: Float -> Maybe Float
radical x | x >= 0 = return (sqrt x)
          | x < 0  = Nothing

solEq2 :: Float -> Float -> Float -> Maybe Float
solEq2 0 0 0 = return 0          -- a * x^2 + b * x + c = 0
solEq2 0 0 c = Nothing
solEq2 0 b c = return ((negate c) / b)
solEq2 a b c = do
                 rDelta <- radical (b * b - 4 * a * c)
                 return (negate b + rDelta) / (2 * a)
```

Monada **Either** (a excepțiilor)

```
data Either err a = Left err | Right a
```

Monada **Either** (a excepțiilor)

```
data Either err a = Left err | Right a

instance Monad (Either err) where
  return = Right
  Right va >>= k  = k va
  err      >>= _  = err    -- Left verr >>=_ = Left verr
```

Monada **Either** (a excepțiilor)

```
data Either err a = Left err | Right a

instance Monad (Either err) where
  return = Right
  Right va >>= k  = k va
  err      >>= _  = err    -- Left verr >>=_ = Left verr

radical :: Float -> Either String Float
radical x | x >= 0 = return (sqrt x)
          | x < 0  = Left "radical: argument negativ"

solEq2 :: Float -> Float -> Float -> Either String Float
solEq2 0 0 0 = return 0          -- a * x^2 + b * x + c = 0
solEq2 0 0 c = Left "Nu are solutii"
solEq2 0 b c = return ((negate c) / b)
solEq2 a b c = do
                 rDelta <- radical (b * b - 4 * a * c)
                 return (negate b + rDelta) / (2 * a)
```

Monada listelor (a rezultatelor nedeterministe)

```
instance Monad [] where
  return va = [va]
  ma >>= k = [vb | va <- ma, vb <- k va]
```

Rezultatul nedeterminist e dat de lista tuturor valorilor posibile.

Monada listelor (a rezultatelor nedeterministe)

```
instance Monad [] where
  return va = [va]
  ma >>= k = [vb | va <- ma, vb <- k va]
```

Rezultatul nedeterminist e dat de lista tuturor valorilor posibile.

```
radical :: Float -> [Float]
radical x | x >= 0 = [negate (sqrt x), sqrt x]
          | x < 0  = []

solEq2 :: Float -> Float -> Float -> [Float]
solEq2 0 0 c = []                    -- a * x^2 + b * x + c = 0
solEq2 0 b c = return ((negate c) / b)
solEq2 a b c = do
                  rDelta <- radical (b * b - 4 * a * c)
                  return (negate b + rDelta) / (2 * a)
```

Monada Writer (variantă simplificată)

```
newtype Writer log a = Writer { runWriter :: (a, log) }
-- a este parametru de tip

tell :: log -> Writer log ()
tell msg = Writer ((), msg)
```

Monada Writer (variantă simplificată)

```
newtype Writer log a = Writer { runWriter :: (a, log) }
-- a este parametru de tip

tell :: log -> Writer log ()
tell msg = Writer ((), msg)

instance Monad (Writer String) where
  return va =  Writer (va, "")
  ma >>= k =   let (va, log1) = runWriter ma
                   (vb, log2) = runWriter (k va)
               in Writer (vb, log1 ++ log2)
```

Monada Writer - Exemplu logging

```
newtype Writer log a = Writer { runWriter :: (a, log) }

tell :: log -> Writer log ()
tell msg = Writer ((), msg)
```

Monada Writer - Exemplu logging

```
newtype Writer log a = Writer { runWriter :: (a, log) }

tell :: log -> Writer log ()
tell msg = Writer ((), msg)

logIncrement :: Int -> Writer String Int
logIncrement x = do
  tell ("increment: " ++ show x ++ "\n")
  return (x + 1)

logIncrement2 :: Int -> Writer String Int
logIncrement2 x = do
  y <- logIncrement x
  logIncrement y


Main> runWriter (logIncrement2 13)
(15,"increment: 13\nincrement: 14\n")
```

Monada Writer (varianta lungă)

Clasa de tipuri Semigroup

O mulțime, cu o operație <> care ar trebui să fie asociativă

```
class Semigroup a where
  (<>) :: a -> a -> a
```

Clasa de tipuri Monoid

Un semigrup cu unitatea mempty. mappend este alias pentru <>.

```
class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mappend = (<>)
```

Foarte multe tipuri sunt instanțe ale lui Monoid. Exemplul clasic: listele.

Monada Writer (varianta lungă)

```
newtype Writer log a = Writer { runWriter :: (a, log) }

instance Monoid log => Monad (Writer log) where
  return a = Writer (a, mempty)
  ma >>= k = let (va, log1) = runWriter ma
                 (vb, log2) = runWriter (k va)
             in Writer (vb, log1 <> log2)
```

Monada Reader (stare nemodificabilă)

```
newtype Reader env a = Reader { runReader :: env -> a }
-- inspecteaza starea curenta
ask :: Reader env env
ask = Reader id
-- modifica starea doar pentru computatia data
local :: (env -> env) -> Reader env a -> Reader env a
local f r = Reader (runReader r . f)
```

Monada Reader (stare nemodificabilă)

```
newtype Reader env a = Reader { runReader :: env -> a }
-- inspecteaza starea curenta
ask :: Reader env env
ask = Reader id
-- modifica starea doar pentru computatia data
local :: (env -> env) -> Reader env a -> Reader env a
local f r = Reader (runReader r . f)

instance Monad (Reader env) where
  return = Reader const  -- return x = Reader (\_ -> x)
  ma >>= k = Reader f
             where
                f env = let va = runReader ma env
                        in  runReader (k va) env
```

Monada Reader- exemplu: mediu de evaluare

```
newtype Reader env a = Reader { runReader :: env -> a }
ask :: Reader env env
ask = Reader id

data Prop ::= Var String | Prop :&: Prop
type Env = [(String, Bool)]

var :: String -> Reader Env Bool
var x = do
          env <- ask
          fromMaybe False (lookup x env)

eval :: Prop -> Reader Env Bool
eval (Var x) = var x
eval (p1 :&: p2) = do
  b1 <- eval p1
  b2 <- eval p2
  return (b1 && b2)
```

Monada State

```
newtype State state a =
    State { runState :: state -> (a, state) }
```

Monada State

```
newtype State state a =
    State { runState :: state -> (a, state) }

instance Monad (State state) where
  return va = State (\s -> (va, s))
-- return va = State f where f s = (va, s)

  ma  >>= k =
      State $ \s -> let (va, news) = runState ma s
                    in runState (k va) news
-- ma :: State state a
-- k :: a -> State state b
-- s :: state
-- runState ma :: state -> (a, state)
-- (va, news) :: (a, state)  = runState ma s
-- k va :: State state b
-- runState (k va) news :: (b, state)
-- ma >>= k :: State state b
```

Monada State

```
newtype State state a =
    State { runState :: state -> (a, state) }

instance Monad (State state) where
  return va = State (\s -> (va, s))
  ma  >>= k =
      State $ \s -> let (va, news) = runState ma s
                    in runState (k va) news
```

Funcții ajutătoare:

```
get :: State state state          -- obtine starea curenta
get = State (\s -> (s, s))

set :: state -> State state () -- seteaza starea curenta
set s = State (const ((), s))

modify :: (state -> state) -> State state ()
modify f = State (\s -> ((), f s))    -- modifica starea
```

Monada State - exemplu "random"

```
newtype State state a = State{runState :: state->(a,state)}
get :: State state state
get = State (\s -> (s,s))
modify :: (state -> state) -> State state ()
modify f = State (\s -> ((), f s))
```

Monada State - exemplu "random"

```
newtype State state a = State{runState :: state->(a,state)}
get :: State state state
get = State (\s -> (s,s))
modify :: (state -> state) -> State state ()
modify f = State (\s -> ((), f s))

cMULTIPLIER, cINCREMENT :: Word32
cMULTIPLIER = 1664525 ; cINCREMENT = 1013904223

rnd, rnd2 :: State Word32 Word32
rnd = do modify (\seed -> cMULTIPLIER * seed + cINCREMENT)
         get
rnd2 = do r1 <- rnd
          r2 <- rnd
          return (r1 + r2)

Main> runState rnd2 0
(2210339985,1196435762)
```

Pe săptămâna viitoare!

Programare declarativă

Evaluare cu efecte laterale

În acest curs:

- vom defini un mini-limbaj asemănător cu limbajul Mini Haskell definit în cursurile trecute
- vom defini semantica limbajului folosind o monadă generică M
- înlocuind M cu monadele standard studiate anterior vom obține variații ale semanticii generale, care vor fi particularizate prin tipul de efecte surprins de fiecare monadă

Sintaxă abstractă

Lambda calcul cu întregi Sintaxă

```
type Name = String

data Term = Var Name
          | Con Integer
          | Term :+: Term
          | Lam Name Term
          | App Term Term
```

Program - Exemplu

λ-expresia $(\lambda x.x + x)(10 + 11)$
este definită astfel:

```
pgm :: Term
pgm = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Con 10) :+: (Con 11))
```

Program - Exemplu

```
pgm :: Term
pgm = App
   (Lam "y"
      (App
         (App
            (Lam "f"
               (Lam "y"
                  (App (Var "f") (Var "y"))
               )
            )
            (Lam "x"
               (Var "x" :+: Var "y")
            )
         )
         (Con 3)
      )
   )
   (Con 4)
```

Valori și medii de evaluare

```
data Value = Num Integer
           | Fun (Value -> M Value)
           | Wrong

instance Show Value where
 show (Num x) = show x
 show (Fun _) = "<function>"
 show Wrong   = "<wrong>"
```

Observații

- Vom interpreta termenii în valori ‘M Value‘, unde ‘M‘ este o monadă; variind se obțin comportamente diferite;
- ‘Wrong‘ reprezintă o eroare, de exemplu adunarea unor valori care nu sunt numere sau aplicarea unui termen care nu e funcție.

Evaluare - variabile și valori

```
type Environment = [(Name, Value)]
```

Interpretarea termenilor în monada ‘M‘

```
interp :: Term -> Environment -> M Value
interp (Var x) env = lookupM x env
interp (Con i) _ = return $ Num i
interp (Lam x e) env = return $
  Fun $ \ v -> interp e ((x,v):env)

lookupM :: Name -> Environment -> M Value
lookupM x env = case lookup x env of
  Just v  -> return v
  Nothing -> return Wrong
```

Evaluare - adunare

```
interp (t1 :+: t2) env = do
  v1 <- interp t1 env
  v2 <- interp t2 env
  add v1 v2
```

Interpretarea adunării în monada ‘M‘

```
add :: Value -> Value -> M Value
add (Num i) (Num j) = return (Num $ i + j)
add _ _             = return Wrong
```

Evaluare - aplicarea funcțiilor

```
interp (App t1 t2) env = do
  f <- interp t1 env
  v <- interp t2 env
  apply f v
```

Interpretarea aplicării funcțiilor în monada 'M'

```
apply :: Value -> Value -> M Value
apply (Fun k) v = k v
apply _ _       = return Wrong

-- k :: Value -> M Value
```

Testarea interpretorului

```
test :: Term -> String
test t = showM $ interp t []
```

unde

```
showM :: Show a => M a -> String
```

este o funcție definită special pentru fiecare tip de efecte laterale dorit.

Testarea interpretorului

```
test :: Term -> String
test t = showM $ interp t []
```

unde

```
showM :: Show a => M a -> String
```

este o funcție definită special pentru fiecare tip de efecte laterale dorit.

Exemplu de program

```
pgmW :: Term
pgmW = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Con 10) :+:  (Con 11))

test pgmW  -- apelul pentru testare
```

Interpretor monadic

```
data Value = Num Integer
           | Fun (Value -> M Value)
           | Wrong

interp :: Term -> Environment -> M Value
```

În continuare vom înlocui monada M cu:

- Identity
- **Maybe**
- **Either String**
- monada listelor
- Writer
- Reader
- State

Interpretare în monada ‘Identity‘

Monada ‘Identity‘ este "efectul identitate".

```
newtype Identity a = Identity { runIdentity :: a }

instance Monad Identity where
    return a              = Identity a
    ma >>= k = k (runIndentity ma)
```

Interpretare în monada ‘Identity‘

Monada ‘Identity‘ este "efectul identitate".

```
newtype Identity a = Identity { runIdentity :: a }

instance Monad Identity where
    return a              = Identity a
    ma >>= k = k (runIndentity ma)
```

Pentru a particulariza interpretorul definim

```
type M a = Identity a

showM :: Show a => M a -> String
showM = show . runIdentity
```

Interpretare în monada ‘Identity‘

Monada ‘Identity‘ este "efectul identitate".

```
newtype Identity a = Identity { runIdentity :: a }

instance Monad Identity where
    return a              = Identity a
    ma >>= k = k (runIndentity ma)
```

Pentru a particulariza interpretorul definim

```
type M a = Identity a

showM :: Show a => M a -> String
showM = show . runIdentity
```

Obținem interpretorul standard, asemănător celui discutat pentru limbajul Mini-Haskell.

Interpretare folosind monada ‘Identity‘

```
type M a = Identity a

showM :: Show a => M a -> String
showM = show . runIdentity

pgm = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Con 10) :+:  (Con 11))

*Var0> test pgm
"42"
```

Interpretare în monada ‘Maybe‘ (opțiune)

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
  return = Just
  Just a  >>= k  = k a
  Nothing >>= _  = Nothing
```

Putem renunța la valoarea ‘Wrong‘, folosind monada ‘Maybe‘

```
type M a = Maybe a

showM :: Show a => M a -> String
showM (Just a) = show a
showM Nothing  = "<wrong>"
```

Interpretare în monada ‘Maybe‘

Putem acum înlocui rezultatele ‘Wrong‘ cu ‘Nothing‘

```
type M a = Maybe a

lookupM :: Name -> Environment -> M Value
lookupM x env = case lookup x env of
  Just v  -> return v
  Nothing -> Nothing

add :: Value -> Value -> M Value
add (Num i) (Num j) = return (Num $ i + j)
add _ _             = Nothing

apply :: Value -> Value -> M Value
apply (Fun k) v = k v
apply _ _       = Nothing
```

Interpretare în monada ‘Either String‘

```
data Either a b = Left a | Right b

instance Monad (Either err) where
  return = Right
  Right a >>= k  = k a
  err     >>= _  = err
```

Putem nuanța erorile folosind monada ‘Either String‘

```
type M a = Either String a

showM :: Show a => M a -> String
showM (Left s)  = "Error: " ++ s
showM (Right a) = "Success: " ++ show a
```

Interpretare în monada ‘Either String‘

Putem acum înlocui rezultatele ‘Wrong‘ cu valori ‘Left‘

```
type M a = Either String a

lookupM :: Name -> Environment -> M Value
lookupM x env = case lookup x env of
  Just v  -> return v
  Nothing -> Left ("unbound variable " ++ x)

add :: Value -> Value -> M Value
add (Num i) (Num j) = return $ Num $ i + j
add v1 v2           = Left $
  "Expected numbers: " ++ show v1 ++ ", " ++ show v2

apply :: Value -> Value -> M Value
apply (Fun k) v = k v
apply v _       = Left $
  "Expected function: " ++ show v
```

Interpretare în monada ‘Either String‘

```
type M a = Either String a

showM :: Show a => M a -> String
showM (Left s)  = "Error: " ++ s
showM (Right a) = "Success: " ++ show a

pgm = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Con 10) :+:  (Con 11))

*Var2> test pgm
"Success: 42"
```

Interpretare în monada ‘Either String‘

```
type M a = Either String a

showM :: Show a => M a -> String
showM (Left s)  = "Error: " ++ s
showM (Right a) = "Success: " ++ show a

pgm = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Con 10) :+:  (Con 11))

*Var2> test pgm
"Success: 42"

pgmE = App (Var "x") ((Con 10) :+:  (Con 11))

*Var2> test pgmE
"Error: unbound variable x"
```

Monada listelor (a funcțiilor nedeterministe)

```
instance Monad [] where
  return a = [a]
  ma >>= k = [b | a <- ma, b <- k a]
```

Rezultatul funcției e lista tuturor valorilor posibile.

```
> [4,9,25] >>= \x -> [(sqrt x), -(sqrt x)]
[2.0,-2.0,3.0,-3.0,5.0,-5.0]
```

Interpretare în monada listelor

Adăugarea unei instrucțiuni nedeterministe

```
data Term = ... | Amb Term Term | Fail

type M a = [a]

showM :: Show a => M a -> String
showM = show

interp Fail _ = []
interp (Amb t1 t2) env = interp t1 env ++ interp t2
    env

pgm = (App (Lam "x" (Var "x" :+: Var "x"))
       (Amb (Con 1) (Con2)))
> test pgm
"[2,4]"
```

Monada ‘Writer‘

Este folosită pentru a acumula (logging) informație produsă în timpul execuției.

```
newtype Writer log a = Writer { runWriter :: (a, log)
    }

instance Monoid log => Monad (Writer log) where
  return a = Writer (a, mempty)
  ma >>= k = let (a, log1) = runWriter ma
                 (b, log2) = runWriter (k a)
             in Writer (b, log1 `mappend` log2)
```

Funcție ajutătoare

```
tell :: log -> Writer log ()
tell log = Writer ((), log)  -- produce mesajul
```

Interpretare în monada ‘Writer‘

Adăugarea unei instrucțiuni de afișare
data Term = ... | Out Term

```
type M a = Writer String a

showM :: Show a => M a -> String
showM ma = "Output: " ++ w ++ "  Value: " ++ show a
  where (a, w) = runWriter ma


interp (Out t) env = do
  v <- interp t env
  tell (show v ++ "; ")
  return v
```

- Out t se evaluează la valoarea lui t, cu efectul lateral de a adăuga valoarea la șirul de ieșire.

Interpretare în monada ‘Writer‘

```
data Term = ... | Out Term

type M a = Writer String a

showM :: Show a => M a -> String
showM ma = "Output: " ++ w ++ "  Value: " ++ show a
  where (a, w) = runWriter ma

pgmW = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Out (Con 10)) :+: (Out (Con 11)))

> test pgm
"Output: 10; 11; Value: 42"
```

Monada ‘Reader‘

Face accesibilă o memorie (environment) nemodificabilă (imuabilă)

```
newtype Reader env a = Reader {runReader :: env -> a}

instance Monad (Reader env) where
  return = Reader const
  ma >>= k = Reader f
    where f env = let a = runReader ma env
                  in  runReader (k a) env
```

Monada ‘Reader‘

Face accesibilă o memorie (environment) nemodificabilă (imuabilă)

```
newtype Reader env a = Reader {runReader :: env -> a}

instance Monad (Reader env) where
  return = Reader const
  ma >>= k = Reader f
    where f env = let a = runReader ma env
                  in  runReader (k a) env
```

Funcții ajutătoare

```
ask :: Reader r r    -- obtine memoria
ask = Reader id  -- Reader (\r -> r)

-- modifica local memoria
local :: (r -> r) -> Reader r a -> Reader r a
local f ma = Reader $ (\r -> (runReader ma)(f r)
```

Interpretare în monada ‘Reader‘

Eliminarea argumentului ‘Environment‘

```
type Environment = [(Name, Value)]
type M a = Reader Environment a

showM :: Show a => M a -> String
showM ma = show $ runReader ma []
```

Funcția de interpretare era definită astfel:

```
interp :: Term -> Enviroment -> M Value
```

Deoarece interpretăm în monada ‘Reader Enviroment a‘ signatura funcției de interpretare este:

```
interp :: Term ->  M Value
```

Interpretare în monada ‘Reader‘

Interpretarea expresiilor de bază și căutare(‘lookup‘)

```
type Environment = [(Name, Value)]
type M a = Reader Environment a

interp :: Term -> M Value
interp (Var x) = lookupM x
interp (Con i) = return $ Num i


lookupM :: Name -> M Value
lookupM x = do
  env <- ask
  case lookup x env of
    Just v  -> return v
    Nothing -> return Wrong
```

Interpretare în monada ‘Reader‘

```
type Environment = [(Name, Value)]
interp :: Term -> M Value
```

Operatori binari și funcții

```
interp (t1 :+: t2) = do
  v1 <- interp t1
  v2 <- interp t2
  add v1 v2
interp (App t1 t2) = do
  f <- interp t1
  v <- interp t2
  apply f v
interp (Lam x e) = do
  env <- ask
  return $ Fun $ \ v ->
    local (const ((x,v):env)) (interp e)
```

Interpretare în monada ‘Reader‘

```
type Environment = [(Name, Value)]
type M a = Reader Environment a

showM :: Show a => M a -> String
showM ma = show $ runReader ma []

interp :: Term ->  M Value

pgm = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Con 10) :+:  (Con 11))

> test pgm
"42"
```

Monada ‘State‘

```
newtype State state a =
  State { runState :: state -> (a, state) }
instance Monad (State state) where
  return a = State (\ s -> (a, s))
  ma >>= k = State (\state ->
      let (a, aState) = runState ma state
       in runState (k a) aState)
```

Funcții ajutătoare

```
get :: State state state
get = State (\s -> (s, s))  -- starea curenta

put :: s -\> State s ()
put s = State (\_ ->((), s)) -- schimba starea

modify :: (state -> state) -> State state ()
modify f = State (\s -> ((), f s))
```

Interpretare în monada ‘State‘

Adăugăm un contor de instrucțiuni ‘Count‘, valoarea acestui contor reprezentând starea.
Astfel variabilele care reprezintă starea sunt numere întregi.

```
data Term = ... | Count

type M a = State Integer a

showM :: Show a => M a -> String
showM ma = show a ++ "\n" ++ "Count: " ++ show s
           where (a, s) = runState ma 0

interp Count _ = do
                   i <- get
                   return (Num i)
```

Interpretare în monada ‘State‘

Creștem starea (contorul) la fiecare instrucțiune

```
tickS :: M ()
tickS = modify (+1)  -- \s ->((), (s+1))

add :: Value -> Value -> M Value
add (Num i) (Num j) = tickS >> return (Num $ i + j)
add _ _             = return Wrong

apply :: Value -> Value -> M Value
apply (Fun k) v = tickS >> k v
apply _ _       = return Wrong
```

Interpretare în monada ‘State‘

```
data Term = ... | Count

type M a = State Integer a

showM :: Show a => M a -> String
showM ma = show a ++ "\n" ++ "Count: " ++ show s
           where (a, s) = runState ma 0

pgm = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Con 10) :+:  (Con 11))

> test pgm
"42\nCount: 3"
```

Pe săptămâna viitoare!

Curs 4

2020-2021

Fundamentele Limbajelor de Programare

Analiză sintactică

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

- Dar cel puțin pentru rezultate parțiale, va mai rămâne ceva de analizat

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

- Dar cel puțin pentru rezultate parțiale, va mai rămâne ceva de analizat

A doua încercare

```
type Parser a = String -> (a, String)
```

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

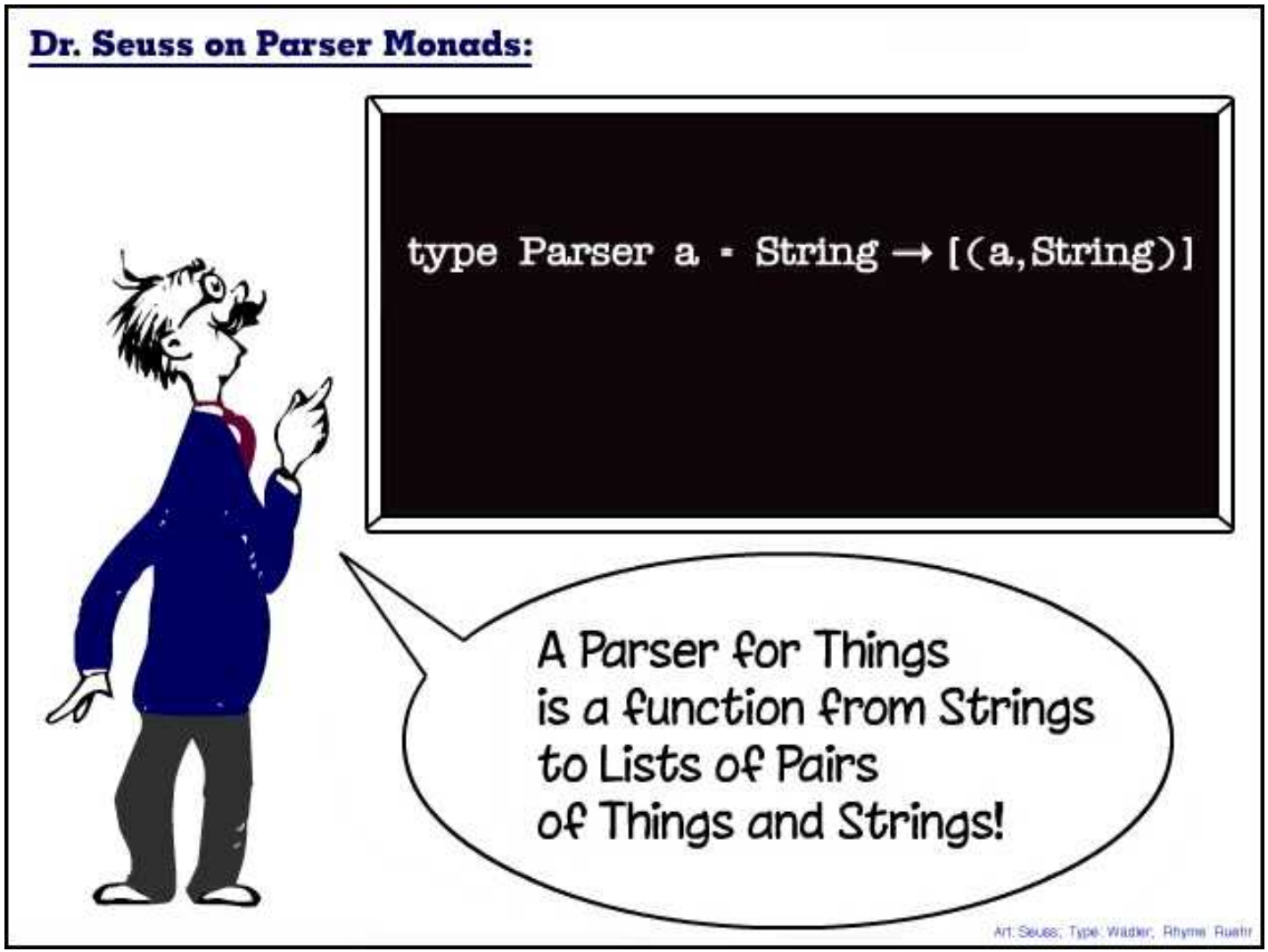
- Dar cel puțin pentru rezultate parțiale, va mai rămâne ceva de analizat

A doua încercare

```
type Parser a = String -> (a, String)
```

- Dar dacă gramatica e ambiguă?
- Dar dacă intrarea nu corespunde nici unui element din a?

Tipul unui analizor sintactic



Tipul Parser

Tipul Parser

```
newtype Parser a =
  Parser { apply :: String -> [(a, String)] }
```

```
-- Folosirea unui parser
-- apply :: Parser a -> String -> [(a, String)]
-- apply (Parser f) s = f s

-- Daca exista parsare, da prima varianta
parse :: Parser a -> String -> a
parse m s = head [ x | (x,t) <- apply m s, t == "" ]
```

Parsare pentru caractere

```
-- Recunoasterea unui caracter
anychar :: Parser Char
anychar = Parser f
  where
  f []     = []
  f (c:s) = [(c,s)]
```

Parsare pentru caractere

```
-- Recunoasterea unui caracter
anychar :: Parser Char
anychar = Parser f
  where
  f []     = []
  f (c:s) = [(c,s)]
```

```
*Main> parse anychar "a"
'a'
*Main> parse anychar "ab"
*** Exception: Prelude.head: empty list

*Main> apply anychar "abc"
[('a',"bc")]
```

Parsare pentru caractere

```
-- Recunoasterea unui caracter cu o proprietate
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = Parser f
  where
  f []                  = []
  f (c:s) | p c         = [(c, s)]
          | otherwise = []
```

Parsare pentru caractere

```
-- Recunoasterea unui caracter cu o proprietate
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = Parser f
  where
  f []                  = []
  f (c:s) | p c         = [(c, s)]
          | otherwise = []
```

```
*Main> parse (satisfy isUpper) "A"
'A'
(0.01 secs, 52,760 bytes)
*Main> parse (satisfy isUpper) "a"
*** Exception: Prelude.head: empty list

*Main> apply (satisfy isUpper) "Ab"
[('A',"b")]
```

Parsare pentru caractere

```
-- Recunoasterea unui anumit caracter
char :: Char -> Parser Char
char c = satisfy (== c)
```

Parsare pentru caractere

```
-- Recunoasterea unui anumit caracter
char :: Char -> Parser Char
char c = satisfy (== c)
```

```
*Main> parse (char 'a') "a"
'a'
(0.00 secs, 52,824 bytes)
*Main> parse (char 'a') "ab"
*** Exception: Prelude.head: empty list

*Main>  apply (char 'a') "ab"
[('a',"b")]
```

Parsarea unui cuvânt cheie

```
-- Recunoasterea unui cuvant cheie
string :: String -> Parser String
string [] = Parser (\s -> [([],s)])
string (x:xs) = Parser f
 where
   f s = [(y:z,zs)| (y,ys) <- apply (char x) s,
                    (z,zs) <- apply (string xs) ys]
```

Parsarea unui cuvânt cheie

```
-- Recunoasterea unui cuvant cheie
string :: String -> Parser String
string [] = Parser (\s -> [([],s)])
string (x:xs) = Parser f
 where
   f s = [(y:z,zs)| (y,ys) <- apply (char x) s,
                    (z,zs) <- apply (string xs) ys]

*Main> parse (string "abc") "abc"
"abc"
*Main> parse (string "abc") "abcd"
*** Exception: Prelude.head: empty list

"*Main> apply  (string "abc") "abcd"
[("abc","d")]
```

Monada Parser

```
--    class Monad m where
--      return :: a -> m a
--      (>>=) :: m a -> (a -> m b) -> m b

instance Monad Parser where
  return x  = Parser (\s -> [ (x, s) ])
  m >>= k   = Parser (\s -> [ (y, u)
                            | (x, t) <- apply m s
                            , (y, u) <- apply (k x) t
                            ])
```

Monada Parser

```
-- Recunoasterea unui cuvant cheie
string :: String -> Parser String
string [] = Parser (\s -> [([],s)])
string (x:xs) = Parser f
 where
   f s = [(y:z,zs)| (y,ys) <- apply (char x) s,
                    (z,zs) <- apply (string xs) ys]
```

e echivalent cu

```
string :: String -> Parser String
string []     = return []
string (x:xs) = do  y  <- char x
                    ys <- string xs
                    return (y:ys)
```

Combinarea variantelor

```
digit = satisfy isDigit
abcP = satisfy (`elem` ['A','B','C'])

alt :: Parser a -> Parser a -> Parser a
alt p1 p2 = Parser f
          where f s = apply p1 s ++ apply p2 s
```

Combinarea variantelor

```
digit = satisfy isDigit
abcP = satisfy (`elem` ['A','B','C'])

alt :: Parser a -> Parser a -> Parser a
alt p1 p2 = Parser f
          where f s = apply p1 s ++ apply p2 s

*Main> apply (alt digit abcP) "1sd"
[('1',"sd")]
*Main> apply (alt digit abcP) "Asd"
[('A',"sd")]
*Main> apply (alt digit abcP) "dsd"
[]
*Main> parse (alt digit abcP) "A"
'A'
*Main> parse (alt digit abcP) "1"
'1'
```

Parser e monadă cu plus

```
--    class MonadPlus m where
--      mzero :: m a
--      mplus :: m a -> m a -> m a

instance MonadPlus Parser where
  mzero       = Parser (\s -> [])
  mplus m n   = Parser (\s -> apply m s ++ apply n s)
                -- === alt m n
```

- mzero reprezintă analizorul sintactic care eșuează tot timpul
- mplus reprezintă combinarea alternativelor

```
instance Alternative Parser where
  empty   = mzero
  (<|>) = mplus
```

Parser e monadă cu plus

```
instance MonadPlus Parser where
    mzero       = Parser (\s -> [])
    mplus m n   = Parser (\s -> apply m s ++ apply n s)

instance Alternative Parser where
  empty  = mzero
  (<|>) = mplus
```

```
*Main> apply (digit <|> abcP) "1www"
[('1',"www")]
*Main> apply (digit <|> abcP) "Awww"
[('A',"www")]
*Main> parse (digit <|> abcP) "B"
'B'
*Main> parse (digit <|> abcP) "2"
'2'
```

Recunoașterea unui caracter cu o proprietate

Alternative și Gărzi

```
guard :: MonadPlus f => Bool -> f ()
guard True = return ()
guard False = mzero
```

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = Parser f
  where
  f []                  = []
  f (c:s) | p c         = [(c, s)]
          | otherwise = []
```

e echivalentă cu

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do   c <- anychar
                 guard (p c)
                 return c
```

Recunoașterea unei secvențe repetitive

```
-- Steluta Kleene (zero, una sau mai multe repetitii)
many :: Parser a -> Parser [a]
many p = some p `mplus` return []

-- cel putin o repetitie
some :: Parser a -> Parser [a]
some p = do x <- p
            xs <- many p
            return (x:xs)
```

Recunoașterea unui numar întreg

```
-- Recunoasterea unui numar natural
decimal :: Parser Int
decimal = do  s <- some digit
              return (read s)

-- Recunoasterea unui numar negativ
negative :: Parser Int
negative = do   char '-'
                n <- decimal
                return (-n)

-- Recunoasterea unui numar intreg
integer :: Parser Int
integer = decimal `mplus` parseNeg
```

Recunoașterea unui identificator

Cum arată un identificator

Un identificator este definit de doi parametri

- felul primului caracter (e.g., începe cu o literă)
- felul restului caracterelor (e.g., literă sau cifră)

Dat fiind un parser pentru felul primului caracter și un parser pentru felul următoarelor caractere putem parsa un identificator:

```
-- Recunoasterea unui identificator
identifier :: Parser Char -> Parser Char -> Parser String
identifier firstCh nextCh = do    c <- firstCh
                                  s <- many nextCh
                                  return (c : s)
```

Exemplu

```
myId = identifier (satisfy isAlpha) (satisfy isAlphaNum)
```

Eliminarea spațiilor

Ignorarea spațiilor

```
skipSpace :: Parser ()
skipSpace = do  _ <- many (satisfy isSpace)
                return ()
```

Ignorarea spațiilor de dinainte și după

```
token :: Parser a -> Parser a
token p = do  skipSpace
              x <- p
              skipSpace
              return x
```

Modulul Exp

```
module Exp where

import Monad
import Parser

data Exp = Lit Int
         | Exp :+: Exp
         | Exp :*: Exp
         deriving (Eq,Show)

evalExp    :: Exp -> Int
evalExp    (Lit n)     = n
evalExp    (e :+: f) = evalExp e + evalExp f
evalExp    (e :*: f) = evalExp e * evalExp f
```

Recunoașterea unei expresii

```
parseExp :: Parser Exp
parseExp = parseLit `mplus` parseAdd `mplus` parseMul
  where
  parseLit = do   n <- integer
                  return (Lit n)
  parseAdd = do   char '('
                  d <- token parseExp
                  char '+'
                  e <- token parseExp
                  char ')'
                  return (d :+: e)
  parseMul = do   char '('
                  d <- token parseExp
                  char '*'
                  e <- token parseExp
                  char ')'
                  return (d :*: e)
```

Recunoașterea și evaluarea unei expresii

```
*Exp> parse parseExp "(1 + (2 * 3))"
Lit 1 :+: (Lit 2 :*: Lit 3)
*Exp> evalExp (parse parseExp "(1 + (2 * 3))")
7
*Exp> parse parseExp "( ( 1 + 2 ) * 3 )"
(Lit 1 :+: Lit 2) :*: Lit 3
*Exp> evalExp (parse parseExp "( ( 1 + 2 ) * 3 )")
9
```

Pe săptămâna viitoare!

Fundamentele Limbajelor de Programare

Gramatici de operatori cu precedență

Traian-Florin Șerbănuță

UNIBUC

19 martie 2021

Gramatici de operatori cu precedențe

Definiție

O gramatică independenta de context se numește gramatică de operatori dacă:

- Nu are producții vide — `A ::=`
- Nu are terminale adiacente în partea dreaptă — `A ::= BC`

Exemplu rău

```
E ::= E A E | -E | (E) | x
A ::= + | - | * | / | ^
```

Exemplu bun

```
E ::= E + E | E - E | E * E | E / E | E ^ E | - E | T
T ::= (E) | id | nat
```

Adăugăm precedențe și atribute de asociativitate

```
E ::= T
    > E ^ E (right)
    > - E
    > E * E (left) | E / E (left)
    > E + E (left) | E - E (left)
```

Calculăm tabela de precedențe

```
E ::= T
    > E ^ E (right)
    > - E
    > E * E (left) | E / E (left)
    > E + E (left) | E - E (left)
```

	T	^	0-	*	/	+	-	$
T		>		>	>	>	>	>
^	<	<	>	>	>	>	>	>
0-	<	<	<	>	>	>	>	>
	<	<	<	>	>	>	>	>
/	<	<	<	>	>	>	>	>
+	<	<	<	<	<	>	>	>
-	<	<	<	<	<	>	>	>
$	<	<	<	<	<	<	<	<

Adăugăm precedențele în șirul de analizat

	T	^	0-	*	/	+	-	$
T		>		>	>	>	>	>
^	<	<	>	>	>	>	>	>
0-	<	<	<	>	>	>	>	>
	<	<	<	>	>	>	>	>
/	<	<	<	>	>	>	>	>
+	<	<	<	<	<	>	>	>
-	<	<	<	<	<	>	>	>
$	<	<	<	<	<	<	<	<

- Dacă vrem să analizăm `-2^2^x + 3 * 5 - 2 + 4`
- Transformăm în `$<-<2>^<2>^<x>+<3>*<5>-<2>+<4>$`

Algoritm

1. Punem pe stivă până la primul >
2. Când întâlnim > scoatem din stivă până la <, și evaluăm
   - Am scos din stivă < `V1` o `V2` >
     - unde `V`-urile sunt valori deja obținute
   - Calculăm valoarea nouă `V` (arbore de parsare, număr)
3. Vedem relația dintre operatorul de pe stivă și cel din șir
   - dacă <, punem < `V` pe stivă și mergem la (1)
   - dacă >, punem `V` > pe stiva și mergem la (2)
   - dacă = (aceeași producție), punem `V` pe stivă și mergem la (1)

Până când avem doar $ în șir și operatorul rămas în stivă e $

Exemplu

- $<-<2>^<2>^<x>+<3>*<5>-<2>+<4>$
- ^<2>^<x>+<3>*<5>-<2>+<4>$ $<-<2>
- ^<2>^<x>+<3>*<5>-<2>+<4>$ $<-<2 (0- < ^)
- ^<x>+<3>*<5>-<2>+<4>$ $<-<2^<2>
- ^<x>+<3>*<5>-<2>+<4>$ $<-<2^<2 (^ < ^)
- +<3>*<5>-<2>+<4>$ $<-<2^<2^<x>
- +<3>*<5>-<2>+<4>$ $<-<2^<2^x> (^ > +)
- +<3>*<5>-<2>+<4>$ $<-<2^(2^x)> (^ > +)
- +<3>*<5>-<2>+<4>$ $<-(2^(2^x))> (0- > +)
- +<3>*<5>-<2>+<4>$ $<(-(2^(2^x))) ($ < +)

Algoritm

1. Punem pe stivă până la primul >
2. Când întâlnim > scoatem din stivă până la <, și evaluăm
3. Vedem relația dintre operatorul de pe stivă și cel din șir

- dacă <, punem < apoi valoarea pe stivă si mergem la (1)
- dacă >, punem valoarea, apoi > pe stiva și mergem la (2)

Exemplu

- `+<3>*<5>-<2>+<4>$ $<(-(2^(2^x)))`
- `*<5>-<2>+<4>$ $<(-(2^(2^x)))+<3>`
- `*<5>-<2>+<4>$ $<(-(2^(2^x)))+<3` `(+ < *)`
- `-<2>+<4>$ $<(-(2^(2^x)))+<3*<5>`
- `-<2>+<4>$ $<(-(2^(2^x)))+<3*5>` `(* > -)`
- `-<2>+<4>$ $<(-(2^(2^x)))+(3*5)>` `(+ > -)`
- `-<2>+<4>$ $<((-(2^(2^x)))+(3*5))` `($ < -)`
- `+<4>$ $<((-(2^(2^x)))+(3*5))-<2>`
- `+<4>$ $<((-(2^(2^x)))+(3*5))-2>` `(- > +)`
- `+<4>$ $<(((-(2^(2^x)))+(3*5))-2)` `($ < +)`

Algoritm

1. Punem pe stivă până la primul >
2. Când întâlnim > scoatem din stivă până la <, și evaluăm
3. Vedem relația dintre operatorul de pe stivă și cel din șir

- dacă <, punem < apoi valoarea pe stivă si mergem la (1)
- dacă >, punem valoarea, apoi > pe stiva și mergem la (2)

Exemplu

- `+<4>$ $<(((-(2^(2^x)))+(3*5))-2)`
- `$ $<(((-(2^(2^x)))+(3*5))-2)+<4>`
- `$ $<(((-(2^(2^x)))+(3*5))-2)+4>` (+ > $)
- `$ $((((-(2^(2^x)))+(3*5))-2)+4)` (gata)

Algoritm

1. Punem pe stivă până la primul >
2. Când întâlnim > scoatem din stivă până la <, și evaluăm
3. Vedem relația dintre operatorul de pe stivă și cel din șir

- dacă <, punem < apoi valoarea pe stivă si mergem la (1)
- dacă >, punem valoarea, apoi > pe stiva și mergem la (2)

Până când avem doar $ în șir și operatorul rămas în stivă e $

Surse

- Gatevidyalay
- Wikipedia
- Text.Parsec.Expr