

Curs 1

2020-2021

Fundamentele Limbajelor de Programare

Cuprins

1 Organizare

2 Privire de ansamblu

- Semantica Limbajelor de Programare
- Bazele programării funcționale / logice

3 Programare logică & Prolog

Organizare

Instructori

Curs:

- **Ioana Leuştean (seria 24), Traian-Florin Şerbănuţă (seria 23)**

Laborator

Seria 24 □ **Ioana Leuştean (241, 244)**

□ **Natalia Ozunu (242, 243)**

Seria 23 □ **Ana Țurlea (231, 232, 233)**

□ **Traian Şerbănuţă (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/3de0S0F>
- 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!

- Condiție minimă pentru promovare:
testare parțială: minim 20 puncte și
testare finală: minim 20 puncte.

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

Testare parțială: 40 puncte

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

Testare finală: 50 puncte

- ☐ Data: În sesiune
- ☐ Timp de lucru: 2 ore
- ☐ Prezența este obligatorie pentru a putea promova!
- ☐ Pentru a trece această probă, trebuie să obțineți 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, Rezoluție

□ 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.

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, verificador de tipuri, depanator)

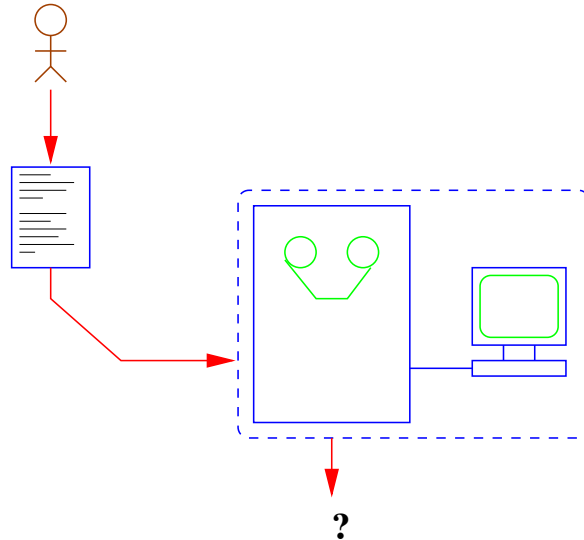
Semantica? Ce înseamnă / care e comportamentul unei instrucțiuni?

- ☐ 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 ușor să stabilim că un program este **corect** sau să înțelegem 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 aplicații "safety-critical".
- Avem nevoie de metode ce asigură "calitate", capabile să ofere "garanții".

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 (specificații), și
- să raționeze despre implementarea lor (corectitudinea programelor).

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;
}
```

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

Tipuri de semantică

Semantica dă "înțeles" unui program.

- Operațională:

- Înțelesul programului este definit în funcție de pașii (transformări dintr-o stare în alta) care apar în timpul execuției.

- Denotațională:

- Înțelesul programului este definit abstract ca element dintr-o structură matematică adecvată.

- Axiomatică:

- Înțelesul programului este definit indirect în funcție de axiomele și regulile pe care le verifică.

- Statică / a tipurilor

- Reguli de bună-formare pentru programe
- Oferă garanții privind execuția (e.g., nu se blochează)

Principalele paradigme de programare

- Imperativă (cum calculăm)

- Procedurală

- Orientată pe obiecte

- Declarativă (ce calculăm)

- Logică

- Funcțională

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 (compiler/implementare) să identifice cum să efectueze calculul respectiv.
- Tipuri de programare declarativă:
 - Programare funcțională (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.
- Unul din sloganurile programării logice:
Program = Logică + Control (R. Kowalski)
- Programarea logică poate fi privită ca o deducție controlată.
- Un program scris într-un limbaj de programare logică este
o listă de formule într-o logică
ce exprimă fapte și 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 colecție de proprietăți presupuse (sub formă de formule logice) despre lume (sau mai degrabă despre lumea programului).
- Programatorul furnizează și 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 consecință a proprietăților presupuse în program.
- Programatorul nu specifică metoda prin care sistemul verifică dacă întrebarea este sau nu consecință a programului.

Exemplu de program logic

oslo → windy
oslo → norway
norway → cold
cold \wedge windy → winterIsComing
oslo

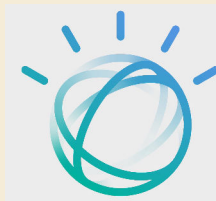
Exemplu de întrebare

Este adevărat `winterIsComing`?

Prolog

- bazat pe logica clauzelor Horn
- semantica operațională este bazată pe rezoluție
- este Turing complet

Limbajul Prolog este folosit pentru programarea sistemului IBM Watson!



Puteți 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



1 Haskell: Clasa de tipuri **Monad**

2 Haskell: Monade standard

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 **compuțațiilor** 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 compuțațiilor

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)
```

Notăția **do** pentru monade

Notăția cu operatori	Notăția do
$e \gg= \backslash x \rightarrow \text{rest}$	$x \leftarrow e$ rest
$e \gg= _ \rightarrow \text{rest}$	e rest
$e \gg \text{rest}$	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
```

```
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
```

```
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.

```
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)
```

```
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)
```

```
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)  
  
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) }  
  
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))
```

```
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)
```

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 - Exemplan

```
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.

Exemplu de program

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

```
test pgmW  -- apelul pentru testare
```


Interpreter 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 (runIdentity 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"
```

```
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 nedeterminate

```
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
```

Funcții ajutătoare

```
ask :: Reader r r    -- obține 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 -> Environment -> M Value
```

Deoarece interpretăm în monada 'Reader Environment a' semnătura 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"
```

Curs 4

2020-2021

Fundamentele Limbajelor de Programare

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

Dr. Seuss on Parser Monads:



```
type Parser a - String → [(a,String)]
```

A Parser for Things
is a function from Strings
to Lists of Pairs
of Things and Strings!

Art: Seuss; Type: Wadler, Rhyme: Rueter

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)]
```

```
*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 = []
```

```
*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)

```
*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 cuvânt 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

*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 puțin 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
```

Fundamentele Limbajelor de Programare

Gramatici de operatori cu precedență

Traian-Florin Șerbănuță

UNIBUC

19 martie 2021

Gramatici de operatori cu precedențe

Definiție

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

- ▶ Nu are producții vide $A ::=$
- ▶ Nu are terminale adiacente în partea dreaptă $A ::= BC$

Exemplu rău

$$E ::= E A E \mid -E \mid (E) \mid x$$
$$A ::= + \mid - \mid * \mid / \mid ^$$

Exemplu bun

$$E ::= E + E \mid E - E \mid E * E \mid E / E \mid E ^ E \mid - E \mid T$$
$$T ::= (E) \mid \text{id} \mid \text{nat}$$

Adăugăm precedente și attribute de asociativitate

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

Calculăm tabela de precedențe

$$E ::= T$$
$$> E \wedge E \text{ (right)}$$

> - E

$$> E * E \text{ (left)} \mid E / E \text{ (left)}$$
$$> E + E \text{ (left)} \mid E - E \text{ (left)}$$
[illegible]

Adăugăm precedențele în șirul de analizat

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

- ▶ Dacă vrem să analizăm $-2^2x + 3 * 5 - 2 + 4$
- ▶ 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 stivă 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

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

Algorithm

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

Exemplu

- ▶ $+<3>*<5>-<2>+<4>\$ \$<(-(2^{(2^x)}))>$
- ▶ $*<5>-<2>+<4>\$ \$<(-(2^{(2^x)}))+<3>$
- ▶ $*<5>-<2>+<4>\$ \$<(-(2^{(2^x)}))+<3$ (+ < *)
- ▶ $-<2>+<4>\$ \$<(-(2^{(2^x)}))+<3*<5>$
- ▶ $-<2>+<4>\$ \$<(-(2^{(2^x)}))+<3*5>$ (* > -)
- ▶ $-<2>+<4>\$ \$<(-(2^{(2^x)}))+(3*5)>$ (+ > -)
- ▶ $-<2>+<4>\$ \$<((- (2^{(2^x)}))+(3*5))>$ (\$ < -)
- ▶ $+<4>\$ \$<((- (2^{(2^x)}))+(3*5))-<2>$
- ▶ $+<4>\$ \$<((- (2^{(2^x)}))+(3*5))-2>$ (- > +)
- ▶ $+<4>\$ \$<(((- (2^{(2^x)}))+(3*5))-2)$ (\$ < +)

Algorithm

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 și mergem la (1)
 - ▶ dacă >, punem valoarea, apoi > pe stivă 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 și mergem la (1)
 - ▶ dacă >, punem valoarea, apoi > pe stivă și mergem la (2)
- Până când avem doar \$ în șir și operatorul rămas în stivă e \$

Surse

- ▶ Gatevidyalay
- ▶ Wikipedia
- ▶ Text.Parsec.Expr

Curs 5

Semantica limbajelor de programare

- Limbaj de programare: sintaxă și semantică
- Feluri de semantică
 - Limbaj natural — descriere textuală a efectelor
 - Operațională — asocierea unei demonstrații a execuției
 - Axiomatică — Descrierea folosind logică a efectelor unei instrucțiuni
 - Denotațională — prin asocierea unui obiect matematic (denotație)
 - Statică — Asocierea unui sistem de tipuri care exclude programe eronate

IMP: un limbaj IMPerativ foarte simplu

Ce conține

□ Expresii

- Aritmetice
- Booleene

□ Blocuri de instrucțiuni

- De atribuire
- Condiționale
- De ciclare
- De interacțiune I/O

```
var n = 0; read("n=", n);
var prime = true;
var i = 1;
while (prime && i * i < n) {
    i = i + 1;
    if (n % i == 0) prime = false
    else {}
};
if (prime) print ("Is prime:", n)
else print ("Is not prime:", n)
```


Sintaxă formală

$$\begin{aligned} E ::= & n \mid x \mid b \\ & \mid E + E \mid E * E \mid E / E \\ & \mid E \leq E \mid E == E \\ & \mid ! E \mid E \&\& E \end{aligned}$$
$$\begin{aligned} C ::= & \text{var } x = E \mid x = E \\ & \mid \text{if } (B) C \text{ else } C \\ & \mid \text{while } (B) C \\ & \mid \text{read } (str, x) \mid \text{print } (str, E) \\ & \mid \{ \{ C ; \}^* \} \end{aligned}$$
$$P ::= \{ C ; \}^*$$

unde n reprezintă numere întregi, b constante de adevăr, x identificatori și str șiruri de caractere.

Semantică Statică - Motivație

- Este sintaxa unui limbaj de programare prea expresivă?
- Sunt programe care n-aș vrea să le pot scrie, dar le pot?
 - Pot aduna întregi cu Booleeni
 - $2 \leq 3 \leq 5$
- Putem detecta programe greșite înainte de rulare?
- Putem garanta că execuția programului nu se va bloca?
 - folosirea variabilelor fără a le declara
 - tipuri incompatibile (variabile, dar și expresii)
- Soluție: Sistemele de tipuri

Sisteme de tipuri

- Descriu programele „bine formate“
- Pot preveni anumite erori
 - folosirea variabilelor nedeclarate/neinițializate
 - detectarea unor bucați de cod inaccesibile
 - erori de securitate
- Ajută compilatorul
- Pot influența proiectarea limbajului

Scop (ideal)

Programele „bine formate“, i. e., cărora li se poate asocia un tip nu eșuează

Reguli intuitive pentru IMP

- Variabilele trebuie declarate înainte de a fi folosite
- Variabilele nu își schimbă tipul în timpul execuției
- operanzii asociați unui operator într-o expresie trebuie să se evalueze la valori corespunzătoare tipurilor așteptate de operator
- Condițiile din if și while se evaluează la valori Booleene
- Se fac operații I/O doar cu valori de tip întreg

Sisteme de tipuri

- Vom defini o relație $\Gamma \vdash \text{frag} : T$
- Citim *frag* are tipul T dacă Γ , unde
- Γ — tipuri asociate variabilelor din e

Exemple

$\vdash \text{if } \text{true} \text{ then } \{\} \text{ else } \{\} : \text{stmt}$

$x : \text{int} \vdash x + 13 : \text{int}$

$x : \text{int} \not\vdash x = y + 1 : T$ pentru orice T

Tipuri în limbajul IMP

Tipurile expresiilor = tipurile gramaticale

$$T ::= \mathbf{int} \mid \mathbf{bool} \mid \mathbf{stmt}$$

Γ — Mediul de tipuri

- Asociază tipuri variabilelor
- **Notăție:** o listă de perechi locație-tip $x_1 : t_1, \dots, x_n : t_n$

Observații pentru limbajul IMP

- variabile din Γ au tipul fie **int** fie **bool**
- Apariția unei variabile în Γ înseamnă că variabila a fost declarată

IMP: Reguli formale pentru tipuri

Tipul constantelor

(INT) $\Gamma \vdash n : \mathbf{int}$ dacă $n \in \mathbb{Z}$

(BOOL) $\Gamma \vdash b : \mathbf{bool}$ dacă $b \in \{true, false\}$

Tipul operatorilor

Operanzii asociați unui operator într-o expresie trebuie să se evalueze la valori corespunzătoare tipurilor așteptate de operator

(OP₊)
$$\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}}$$

(OP_≤)
$$\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 \leq e_2 : \mathbf{bool}}$$

(OP_!)
$$\frac{\Gamma \vdash e : \mathbf{bool}}{\Gamma \vdash !e : \mathbf{bool}}$$

IMP: Reguli formale pentru tipuri

Tipul variabilelor și al declarațiilor

- Variabilele trebuie declarate înainte de a fi folosite
- Variabilele nu își schimbă tipul în timpul execuției

$$(\text{LOC}) \quad \Gamma \vdash x : t \quad \text{dacă } \Gamma(x) = t$$

$$(\text{DECL}) \quad \frac{\Gamma \vdash e : t \quad \Gamma, x \mapsto t \vdash sts : stmt}{\Gamma \vdash \mathbf{var} \ x = e; sts : stmt}$$

Instrucțiuni: Atribuire și secvențiere

- Variabilele nu își schimbă tipul în timpul execuției

$$(\text{ATRIB}) \quad \frac{\Gamma \vdash e : t}{\Gamma \vdash x = e : \mathbf{stmt}} \quad \text{dacă } \Gamma(x) = t$$

$$(\text{SEQ}) \quad \frac{\Gamma \vdash st : \mathbf{stmt} \quad \Gamma \vdash sts : stmt}{\Gamma \vdash st; sts : \mathbf{stmt}} \quad \text{dacă } st \text{ nu e declarație}$$

IMP: Reguli formale pentru tipuri

Tipuri pentru instrucțiuni

- Condițiile din if și while se evaluează la valori Booleene

$$\text{(IF)} \quad \frac{\Gamma \vdash c : \mathbf{bool} \quad \Gamma \vdash t : \mathbf{stmt} \quad \Gamma \vdash e : \mathbf{stmt}}{\Gamma \vdash \mathbf{if } c \mathbf{ then } t \mathbf{ else } e : \mathbf{stmt}}$$

$$\text{(WHILE)} \quad \frac{\Gamma \vdash c : \mathbf{bool} \quad \Gamma \vdash b : \mathbf{stmt}}{\Gamma \vdash \mathbf{while } (c) b : \mathbf{stmt}}$$

- Se fac operații I/O doar cu valori de tip întreg

$$\text{(READ)} \quad \Gamma \vdash \mathbf{read } (s, x) : \mathbf{stmt} \quad \text{dacă } \Gamma(x) = \mathbf{int}$$

$$\text{(PRINT)} \quad \frac{\Gamma \vdash e : \mathbf{int}}{\Gamma \vdash \mathbf{print } (s, e) : \mathbf{stmt}}$$

Haskell: limbajul SIMPLE

Limbajul SIMPLE

SIMPLE - exemplu

```
pFact= Block [  
    Asgn "n" (1 5),  
    Asgn "fact " (Id "n"),  
    Asgn "i" (1 1),  
    While (BinE Neq (Id "n") (Id "i"))  
        (Block [ Asgn "fact" (BinA Mul (Id "  
            fact") (Id "i")),  
                Asgn "i" (BinA Add (Id "i") (1  
                    1))  
            ])  
    ]
```

```
*SIMPLE> pFact  
Block [Asgn "n" 5,Asgn "fact " n,Asgn "i" 1,While n!=  
    i (Block [Asgn "fact" (fact*i),Asgn "i" (i+1)])]
```

Limbajul SIMPLE - operatori

```
type Name = String
```

```
data BinAop = Add | Mul | Sub | Div | Mod
```

```
data BinCop = Lt | Lte | Gt | Gte
```

```
data BinEop = Eq | Neq
```

```
data BinLop = And | Or
```

Limbajul SIMPLE - expresii

```
data Exp
  = Id Name
  | I Integer
  | B Bool
  | UMin Exp
  | BinA BinAop Exp Exp
  | BinC BinCop Exp Exp
  | BinE BinEop Exp Exp
  | BinL BinLop Exp Exp
  | Not Exp
```

Limbajul SIMPLE - instrucțiuni

data Stmt

= Asgn Name Exp

| If Exp Stmt Stmt

| While Exp Stmt

| Block [Stmt]

| Decl Name Exp

SIMPLE: operatori binari

- operatori aritmetici

```
data BinAop = Add | Mul
```

```
instance Show BinAop where
```

```
  show Add = "+"
```

```
  show Mul = "*"
```

- operatori logici

```
data BinLop = And | Or
```

```
instance Show BinLop where
```

```
  show And = "&&"
```

```
  show Or = "||"
```

SIMPLE: relații

□ relații de ordine

```
data BinCop =  Lte |  Gte
```

```
instance Show BinCop where  
  show Lte = "<="   
  show Gte = ">="
```

□ relația de egalitate

```
data BinEop = Eq |  Neq
```

```
instance Show  BinEop where  
  show Eq = "=="   
  show Neq = "!="
```


SIMPLE: expresii simple

```
data Exp
  = Id Name
  | I Integer
  | B Bool

instance Show Exp where
  show (Id x) = x
  show (I i) = show i
  show (B True) = "true"
  show (B False) = "false"
```

SIMPLE: expresii (complet)

```
data Exp
  = Id Name
  | I Integer
  | B Bool
  | UMin Exp
  | BinA BinAop Exp Exp
  | BinC BinCop Exp Exp
  | BinE BinEop Exp Exp
  | BinL BinLop Exp Exp
  | Not Exp
```

SIMPLE: expresii (complet)

```
instance Show Exp where
```

```
  show (Id x) = x
```

```
  show (I i) = show i
```

```
  show (B True) = "true"
```

```
  show (B False) = "false"
```

```
  show (UMin e) = "-" ++ show e
```

```
  show (BinA op e1 e2) = addParens $ show e1 ++  
    show op ++ show e2
```

```
  show (BinL op e1 e2) = addParens $ show e1 ++  
    show op ++ show e2
```

```
  show (BinC op e1 e2) = show e1 ++ show op ++ show  
    e2
```

```
  show (BinE op e1 e2) = show e1 ++ show op ++ show  
    e2
```

```
  show (Not e) = "!" ++ show e
```

```
addParens :: String -> String
```

```
addParens e = "(" ++ e ++ ")"
```

Exemplu: SIMPLE expresii

```
> :t BinC Lte (Id "prime") (BinA Add (I 2) (Id "x"))
BinC Lte (Id "prime") (BinA Add (I 2) (Id "x")) ::
  Exp
```

```
> :t BinL Lte (Id "prime") (BinA Add (I 2) (Id "x"))
error: ...
```

```
> BinC Lte (Id "prime") (BinA Add (I 2) (Id "x"))
prime <= (2+x)
```

Exemplu: SIMPLE expresii

Observați că:

```
> :t BinC Lte (B True) (BinA Add (I 2) (Id "x"))  
BinC Lte (B True) (BinA Add (I 2) (Id "x")) :: Exp
```

```
> BinC Lte (B True) (BinA Add (I 2) (Id "x"))  
true <=(2+x)
```

Expresia de mai sus este **coerectă sintactic** dar **greșită** din punctul de vedere al verificării tipurilor.

SIMPLE: instrucțiuni

```
data Stmt
    =   Asgn Name Exp
      | If Exp Stmt Stmt
      | While Exp Stmt
      | Block [Stmt]
      | Decl Name Exp
```

deriving (Show)

Exemplu: SIMPLE program

```
pFact= Block [  
  Asgn "n" (1 5),  
  Asgn "fact" (Id "n"),  
  Asgn "i" (1 1),  
  While (BinE Neq (Id "n") (Id "i"))  
    (Block [ Asgn "fact" (BinA Mul (Id "  
      fact") (Id "i")),  
      Asgn "i" (BinA Add (Id "i") (1  
        1))  
    ])  
  ]
```

```
*SIMPLE> :t pFact  
pFact :: Stmt
```

Exemplu: SIMPLE program

```
pFact= Block [  
    Decl "n" (1 5),  
    Decl "fact " (Id "n"),  
    Decl "i" (1 1),  
    While (BinE Neq (Id "n") (Id "i"))  
        (Block [ Asgn "fact" (BinA Mul (Id "  
            fact") (Id "i")),  
                Asgn "i" (BinA Add (Id "i") (1  
                    1))  
                ])  
    ]
```

```
*SIMPLE> pFact  
Block [Asgn "n" 5,Asgn "fact " n,Asgn "i" 1,  
While n!=i (Block [Asgn "fact" (fact*i),  
Asgn "i" (i+1)])]
```


SIMPLE: verificarea sistemului de tipuri

SIMPLE type checker

- vom folosi o "stare" in care fiecare variabila are asociat un tip; "stările" sunt definite folosind Data.Map

```
import Data.Map.Strict as Map
type CheckerState = Map Name Type
```

```
emptyCSt :: CheckerState  -- "starea" vida
emptyCSt = Map.empty
```

- funcția de verificare va asocia unei construcții sintactice o valoare M Type, unde M este o monadă iar Type este un tip:

```
data Type = TInt | TBool
```

```
checkExp :: Exp -> M Type
```

Tipul "unit" () va fi tipul instrucțiunilor: unde

```
checkStmt :: Stmt -> M ()
checkBlock :: [Stmt] -> M ()
```

SIMPLE type checker: monada

- Monada M este o combinație între monada Reader și monada **Either**

```
newtype EReader a =  
    EReader {runEReader :: CheckerState ->(Either  
        String a)}
```



```
instance Monad EReader where  
    return a = EReader (\env -> Right a)  
    act >=> k = EReader f  
        where  
            f env = case (runEReader act env)  
                of  
                    Left s -> Left s  
                    Right va -> runEReader (k  
                        va) env
```

```
type M = EReader
```

SIMPLE type checker: monada

- Monada EReader este o combinație între monada Reader și monada **Either**

```
newtype EReader a =  
    EReader {runEReader :: CheckerState ->(Either  
        String a)}
```

```
askEReader :: EReader CheckerState  
askEReader =EReader (\env -> Right env)
```

```
localEReader ::( CheckerState ->CheckerState) ->  
    EReader a -> EReader a  
localEReader f ma = EReader (\env -> (runEReader ma)  
    (f env))
```

SIMPLE type checker: funcții auxiliare

```
throwError :: String -> EReader a  
throwError e = EReader (\_ -> (Left e))
```

```
expect :: (Show t, Eq t, Show e) => t -> t -> e -> M ()  
expect tExpect tActual e =  
    if (tExpect /= tActual)  
    then      (throwError  
        $ "Type mismatch. Expected " ++ show tExpect  
        ++ " but got " ++ show tActual ++ " for "  
        ++ show e)  
    else (return ())
```

SIMPLE type checker: **lookup**

```
data Type = TInt | TBool  
    deriving (Eq)
```

```
type CheckerState = Map Name Type  
type M = EReader
```

```
lookupM :: Name -> M Type
```

```
lookupM x = do  
    env <- askEReader  
    case (Map.lookup x env) of  
        Nothing -> throwError $ "Variable " <> x <> "  
            not declared"  
        Just t -> return t
```

SIMPLE type checker: structura generală

`checkExp :: Exp -> M Type`

`checkStmt :: Stmt -> M ()`

`checkBlock :: [Stmt] -> M ()`

`checkPgm :: [Stmt] -> Bool`

`checkPgm pgm =`

`case (runEReader (checkBlock pgm)) emptyCSt of
 Left err -> error err
 Right _ -> True`

`*Checker> checkPgm [pFact]
True`

SIMPLE type checker: structura generală

`checkExp :: Exp -> M Type`

`checkStmt :: Stmt -> M ()`

`checkBlock :: [Stmt] -> M ()`

`checkPgm :: [Stmt] -> Bool`

```
checkPgm pgm =  
  case (runEReader (checkBlock pgm)) emptyCSt of  
    Left err -> error err  
    Right _ -> True
```

```
*Checker> runEReader (checkExp (BinC Lte (B True)  
  (BinA Add (I 2) (Id "x")))) emptyCSt  
Left "Type mismatch. Expected int but got bool for  
  true"
```


SIMPLE type checker: verificarea blocurilor

```
checkBlock :: [Stmt] -> M ()
```

```
checkBlock [] = return ()
```

```
checkBlock (Decl x e : ss) = do  
    t <- checkExp e  
    localEReader (Map.insert x t) (checkBlock ss)
```

```
checkBlock (s : ss) = checkStmt s >> checkBlock ss
```

SIMPLE type checker: verificarea instrucțiunilor

```
checkStmt :: Stmt -> M ()  
checkStmt (Decl _ _) = return ()  
checkStmt (Block ss) = checkBlock ss
```

```
checkStmt (Asgn x e) = do  
    tx <- lookupM x  
    te <- checkExp e  
    expect tx te e
```

```
checkStmt (If e s1 s2) = do  
    te <- checkExp e  
    expect TBool te e  
    checkStmt s1  
    checkStmt s2
```

```
checkStmt (While e s) = ...
```

SIMPLE type checker: verificarea expresiilor

```
checkExp :: Exp -> M Type
checkExp (I _) = return TInt
checkExp (B _) = return TBool
checkExp (Id x) = lookupM x
```

```
checkExp (BinA _ e1 e2) = do
    t1 <- checkExp e1
    expect TInt t1 e1
    t2 <- checkExp e2
    expect TInt t2 e2
    return TInt
```

```
checkExp (Not e) = do
    t <- checkExp e
    expect TBool t e
    return TBool
```

...

Curs 6

2020-2021

Fundamentele Limbajelor de Programare

λ -calcul

- În 1929-1932 Church a propus λ -calculul ca sistem formal pentru logica matematică. În 1935 a argumentat că orice funcție calculabilă peste numere naturale poate fi calculată în λ -calcul.
- În 1935, independent de Church, Turing a dezvoltat mecanismul de calcul numit astăzi Mașina Turing. În 1936 și el a argumentat că orice funcție calculabilă peste numere naturale poate fi calculată de o mașină Turing. De asemenea, a arătat echivalența celor două modele de calcul. Această echivalență a constituit o indicație puternică asupra "universalității" celor două modele, conducând la ceea ce numim astăzi "Teza Church-Turing".

Referințe

- Benjamin C. Pierce, Types and Programming Languages, The MIT Press 2002
- J.R. Hindley, J.P. Seldin, Lambda-Calculus and Combinators, an Introduction, Cambridge University Press, 2008
- R. Nederpelt, H. Geuvers, Type Theory and Formal Proof, an Introduction, Cambridge University Press 2014

λ -calcul: sintaxa

Lambda Calcul - sintaxă

$t =$	x	(variabilă)
	$ \lambda x. t$	(abstractizare)
	$ t t$	(aplicare)

λ -termeni

Fie $Var = \{x, y, z, \dots\}$ o mulțime infinită de variabile.

Mulțimea λT termenilor λT este definită inductiv astfel:

[Variabilă] $Var \subseteq \lambda T$

[Aplicare] dacă $t_1, t_2 \in \lambda T$ atunci $(t_1 t_2) \in \lambda T$

[Abstractizare] dacă $x \in Var$ și $t \in \lambda T$ atunci $(\lambda x. t) \in \lambda T$

Lambda termeni

λ -termeni: exemple

- x, y, z
- $(xy), (yx), (x(yx))$
- $(\lambda x.x), (\lambda x.(xy)), (\lambda z.(xy)), (\lambda x.(\lambda z.(xy)))$
- $((\lambda x.x)y), ((\lambda x.(xz))y), ((\lambda x.x)(\lambda y.y))$

Convenții:

- se elimină parantezele exterioare
- aplicarea este asociativă la stînga: $t_1 t_2 t_3$ este $(t_1 t_2) t_3$
- corpul abstractizării este extins la dreapta: $\lambda x.t_1 t_2$ este $\lambda x.(t_1 t_2)$ (nu $(\lambda x.t_1) t_2$)
- scriem $\lambda xyz.t$ în loc de $\lambda x.\lambda y.\lambda z.t$

Lambda termeni / Funcții anonime

λ -termeni: exemple

- x, y, z
- $(xy), (yx), (x(yx))$
- $(\lambda x.x), (\lambda x.(xy)), (\lambda z.(xy)), (\lambda x.(\lambda z.(xy)))$
- $((\lambda x.x)y), ((\lambda x.(xz))y), ((\lambda x.x)(\lambda y.y))$

λ -termeni/ funcții anonime în Haskell

În Haskell, \backslash e folosit în locul simbolului λ și \rightarrow în locul punctului.

$\lambda x.x * x$ este $\backslash x \rightarrow x * x$

$\lambda x.x > 0$ este $\backslash x \rightarrow x > 0$

Variabile libere și legate

Apariții libere și legate

Pentru un termen $\lambda x.t$ spunem că:

- aparițiile variabilei x în t sunt legate (*bound*)
- λx este legătura (*binder*), iar t este domeniul (*scope*) legării
- o apariție a unei variabile este liberă (*free*) dacă apare într-o poziție în care nu e legată.

Un termen fără variabile libere se numește închis (*closed*).

Exemplu:

- $\lambda x.x$ este un termen închis
- $\lambda x.xy$ nu este termen închis, x este legată, y este liberă
- în termenul $x(\lambda x.xy)$ prima apariție a lui x este liberă, a doua este legată.

Variabile libere

Mulțimea variabilelor libere $FV(t)$

Pentru un λ -termen t mulțimea variabilelor libere este definită astfel:

$$\text{[Variabilă]} \quad FV(x) = x$$

$$\text{[Aplicare]} \quad FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$$

$$\text{[Abstractizare]} \quad FV(\lambda x.t) = FV(t) \setminus \{x\}$$

Exemplu:

$$\begin{aligned} FV(\lambda x.xy) &= FV(xy) \setminus \{x\} \\ &= (FV(x) \cup FV(y)) \setminus \{x\} \\ &= (\{x\} \cup \{y\}) \setminus \{x\} \\ &= \{y\} \end{aligned}$$

$$FV(x\lambda x.xy) = \{x, y\}$$

Substituții

Fie t un λ -termen $x \in Var$.

Definiție intuitivă

Pentru un λ -termen u vom nota prin $[u/x]t$ rezultatul înlocuirii tuturor aparițiilor libere ale lui x cu u în t .

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$
- $[(\lambda z.zw)/x](\lambda y.x) = \lambda y.\lambda z.zw$

Substituții

Definirea substituției

Rezultatul substituirii lui x cu u în t este definit astfel:

[Variabilă] $[u/x]x = u$

[Variabilă] $[u/x]y = y$ dacă $x \neq y$

[Aplicare] $[u/x](t_1 t_2) = [u/x]t_1 [u/x]t_2$

[Abstractizare] $[u/x]\lambda y.t = \lambda y.[u/x]t$ unde
 $y \neq x$ și $y \notin FV(u)$

Substituții

Exemple: Dacă x, y, z sunt variabile distincte atunci

□ $[y/x]\lambda z.x = \lambda z.y$

□ Cine este $[y/x]\lambda y.x$?

Dacă folosim definiția intuitivă obținem

$[y/x]\lambda y.x = \lambda y.y$ ceea ce este greșit!

Cum procedăm pentru a repara greșeala? Observăm că $\lambda y.x$ desemnează o funcție constantă, aceeași funcție putând fi reprezentată prin $\lambda z.x$. Aplicarea corectă a substituției este:

$$[y/x]\lambda y.x = [y/x]\lambda z.x = \lambda z.y$$

Avem libertatea de a redenumi variabilele legate!

α -conversie (α -echivalență)

α -conversia $=_\alpha$

[Reflexivitate] $t =_\alpha t$

[Simetrie] $t_1 =_\alpha t_2$ implică $t_2 =_\alpha t_1$

[Tranzitivitate] $t_1 =_\alpha t_2$ și $t_2 =_\alpha t_3$ implică $t_1 =_\alpha t_3$

[Redenumire] $\lambda x.t =_\alpha \lambda y.[y/x]t$ dacă $y \notin FV(t)$

[Compatibilitate] $t_1 =_\alpha t_2$ implică

$tt_1 =_\alpha tt_2$, $t_1t =_\alpha t_2t$ și $\lambda x.t_1 =_\alpha \lambda x.t_2$

α -conversie (α -echivalență)

α -conversia $=_{\alpha}$

[Reflexivitate] $t =_{\alpha} t$

[Simetrie] $t_1 =_{\alpha} t_2$ implică $t_2 =_{\alpha} t_1$

[Tranzitivitate] $t_1 =_{\alpha} t_2$ și $t_2 =_{\alpha} t_3$ implică $t_1 =_{\alpha} t_3$

[Redenumire] $\lambda x.t =_{\alpha} \lambda y.[y/x]t$ dacă $y \notin FV(t)$

[Compatibilitate] $t_1 =_{\alpha} t_2$ implică

$tt_1 =_{\alpha} tt_2$, $t_1t =_{\alpha} t_2t$ și $\lambda x.t_1 =_{\alpha} \lambda x.t_2$

$t_1 =_{\alpha} t_2$ și $u_1 =_{\alpha} u_2$ implică $[u_1/x]t_1 =_{\alpha} [u_2/x]t_2$

Exemplu:

$[xy/x](\lambda y.yx) =_{\alpha} [xy/x](\lambda z.zx) =_{\alpha} \lambda z.z(xy)$

Vom lucra modulo α -conversie, doi termeni α -echivalenți vor fi considerați "egali".

Substituție

Exemplu:

$$\square [xy/x]\lambda x.yx =_{\alpha} [xy/x]\lambda z.yz = \lambda z.[xy/x](yz) = \lambda z.yz$$

Observăm că $\lambda z.yz =_{\alpha} \lambda x.yx$

$$\square [y/z]\lambda xy.zzx = \lambda x.[y/z]\lambda y.zzx =_{\alpha} \lambda x.[y/z]\lambda v.zzx = \lambda x.\lambda v.[y/z](zzx) = \lambda xv.yyx$$

β -reducție

β -reducția este o relație pe mulțimea α -termenilor.

β -reducția $\rightarrow_\beta, \rightarrow_\beta^*$

- un singur pas $\rightarrow_\beta \subseteq \Lambda T \times \Lambda T$
 - [Aplicarea] $(\lambda x.t)u \rightarrow_\beta [u/x]t$
 - [Compatibilitatea] $t_1 \rightarrow_\beta t_2$ implică
 $tt_1 \rightarrow_\beta tt_2, t_1 t \rightarrow_\beta t_2 t$ și $\lambda x.t_1 \rightarrow_\beta \lambda x.t_2$
- zero sau mai mulți pași $\rightarrow_\beta^* \subseteq \Lambda T \times \Lambda T$
 - $t_1 \rightarrow_\beta^* t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât
 $t_1 =_\alpha u_0 \rightarrow_\beta u_1 \rightarrow_\beta \dots \rightarrow_\beta u_n =_\alpha t_2$

β -reducție

Să considerăm termenul $(\lambda x.(\lambda y.yx)z)v$

$$\square (\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda y.yv)z \rightarrow_{\beta} zv$$

$$\square (\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda x.zx)v \rightarrow_{\beta} zv$$

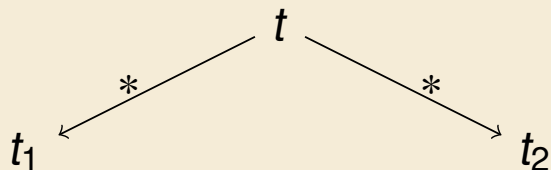
Observăm că un termen poate fi β -redus în mai multe moduri.

Proprietatea de confluență ne asigură că vom ajunge întotdeauna la același rezultat.

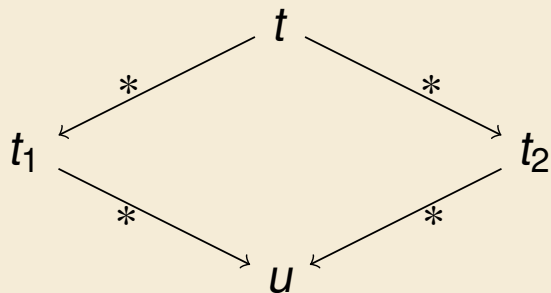
Confluența β -reducției

Teorema Church-Rosser

Dacă $t \xrightarrow{*}_{\beta} t_1$ și $t \xrightarrow{*}_{\beta} t_2$



atunci există u astfel încât $t_1 \xrightarrow{*}_{\beta} u$ și $t_2 \xrightarrow{*}_{\beta} u$.



β -forma normală

Intuitiv, o formă normală este un termen care nu mai poate fi redus (sau punctul final al unui calcul).

Formă normală

- un λ -termen căruia nu i se mai poate aplica reducerea într-un pas \rightarrow_β se numește β -formă normală
- dacă $t \xrightarrow{*}_\beta u_1$, $t \xrightarrow{*}_\beta u_2$ și u_1, u_2 sunt η -forme normale atunci, datorită confluentei, $u_1 =_\alpha u_2$
- un λ -termen poate avea cel mult o β -formă normală (modulo α -echivalență)

β -forma normală

Formă normală

- un λ -termen căruia nu i se mai poate aplica reducerea într-un pas \rightarrow_β se numește *β -formă normală*
- dacă $t \xrightarrow{*}_\beta u_1$, $t \xrightarrow{*}_\beta u_2$ și u_1, u_2 sunt η -forme normale atunci, datorită confluentei, $u_1 =_\alpha u_2$
- un λ -termen poate avea cel mult o β -formă normală (modulo α -echivalență)

Exemplu:

- zv este β -formă normală pentru $(\lambda x.(\lambda y.yx)z)v$
 $(\lambda x.(\lambda y.yx)z)v \rightarrow_\beta (\lambda y.yv)z \rightarrow_\beta zv$
- există termeni care **nu** pot fi reduși la o β -formă normală, de exemplu $(\lambda x.xx)(\lambda x.xx)$

β -conversia

Intuitiv, β -conversia extinde β -reducția în ambele direcții.

- $(\lambda y. yv)z \rightarrow_{\beta} zv \leftarrow_{\beta} (\lambda x. zx)v$
- $(\lambda y. yv)z \leftarrow_{\beta} (\lambda x. (\lambda y. yx)z)v \rightarrow_{\beta} (\lambda x. zx)v$

β -conversia $=_{\beta}$

- $=_{\beta} \subseteq \Lambda T \times \Lambda T$
 $t_1 =_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât
 $t_1 =_{\alpha} u_0$, $u_n =_{\alpha} t_2$ și, pentru orice i , $u_i \rightarrow_{\beta} u_{i+1}$ sau $u_{i+1} \rightarrow_{\beta} u_i$

Exemplu: $(\lambda y. yv)z =_{\beta} (\lambda x. zx)v$

β -conversia

β -conversia $=_{\beta}$

□ $=_{\beta} \subseteq \Lambda T \times \Lambda T$

$t_1 =_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât

$t_1 =_{\alpha} u_0$, $u_n =_{\alpha} t_2$ și, pentru orice i , $u_i \rightarrow_{\beta} u_{i+1}$ sau $u_{i+1} \rightarrow_{\beta} u_i$

Observații

□ $=_{\beta}$ este o relație de echivalență

□ pentru t_1, t_2 λ -termeni și u_1, u_2 β -forme normale

dacă $t_1 \xrightarrow{*}_{\beta} u_1$, $t_2 \xrightarrow{*}_{\beta} u_2$ și $u_1 =_{\alpha} u_2$ atunci $t_1 =_{\beta} t_2$

β -conversia

β -conversia $=_{\beta}$

- $=_{\beta} \subseteq \Lambda T \times \Lambda T$
 $t_1 =_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât
 $t_1 =_{\alpha} u_0$, $u_n =_{\alpha} t_2$ și, pentru orice i , $u_i \rightarrow_{\beta} u_{i+1}$ sau $u_{i+1} \rightarrow_{\beta} u_i$
- $=_{\beta}$ este o relație de echivalență
- pentru t_1, t_2 λ -termeni și u_1, u_2 β -forme normale
dacă $t_1 \xrightarrow{*}_{\beta} u_1$, $t_2 \xrightarrow{*}_{\beta} u_2$ și $u_1 =_{\alpha} u_2$ atunci $t_1 =_{\beta} t_2$

β -conversia reprezintă "egalitatea prin calcul", iar β -reducția (modulo α -conversie) oferă o procedură de decizie pentru aceasta.

Curs 7

λ -calcul și calculabilitate

- În 1929-1932 Church a propus λ -calculul ca sistem formal pentru logica matematică. În 1935 a argumentat că orice funcție calculabilă peste numere naturale poate fi calculată în λ -calcul.
- În 1935, independent de Church, Turing a dezvoltat mecanismul de calcul numit astăzi Mașina Turing. În 1936 și el a argumentat că orice funcție calculabilă peste numere naturale poate fi calculată de o mașină Turing. De asemenea, a arătat echivalența celor două modele de calcul. Această echivalență a constituit o indicație puternică asupra "universalității" celor două modele, conducând la ceea ce numim astăzi "Teza Church-Turing".

λ -calcul: sintaxa

$$\begin{array}{ll} t = & x \quad \text{(variabilă)} \\ & | (\lambda x. t) \quad \text{(abstractizare)} \\ & | (t \ t) \quad \text{(aplicare)} \end{array}$$

Convenții:

- se elimină parantezele exterioare
- aplicarea este asociativă la stînga: $t_1 t_2 t_3$ este $(t_1 t_2) t_3$
- corpul abstractizării este extins la dreapta: $\lambda x. t_1 t_2$ este $\lambda x. (t_1 t_2)$ (nu $(\lambda x. t_1) t_2$)
- scriem $\lambda xyz. t$ în loc de $\lambda x. \lambda y. \lambda z. t$

Întrebare

Ce putem exprima / calcula folosind **doar** λ -calculul?

Rezumat

- Reprezentarea valorilor de adevăr și a expresiilor condiționale
- Reprezentarea perechilor (tuplurilor) și a funcțiilor proiecție
- Reprezentarea numerelor și a operațiilor aritmetice de bază
- Recursie

Ideea generală

Intuiție

Tipurile de date sunt codificate de capabilități

Boole capacitatea de a alege între două alternative

Perechi capacitatea de a calcula ceva bazat pe două valori

Numere naturale capacitatea de a itera de un număr dat de ori

Valori de adevăr

Intuiție: Capabilitatea de a alege între două alternative.

Codificare: Un Boolean este o funcție cu 2 argumente reprezentând ramurile unei alegeri.

true ::= $\lambda t f.t$ — din cele două alternative o alege pe prima

false ::= $\lambda t f.f$ — din cele două alternative o alege pe a doua

Operații Booleene

true ::= $\lambda t f.t$ — din cele două alternative o alege pe prima

false ::= $\lambda t f.f$ — din cele două alternative o alege pe a doua

if ::= $\lambda c \text{ then } else.c \text{ then } else$ — pur și simplu folosim valoarea de adevăr pentru a alege între alternative

if false $(\lambda x.x \ x) (\lambda x.x) \rightarrow_{\beta}^3$

false $(\lambda x.x \ x) (\lambda x.x) \rightarrow_{\beta}^2 \lambda x.x$

and ::= $\lambda b1 \ b2. \text{if } b1 \ b2 \text{ false sau } \lambda b1 \ b2.b1 \ b2 \ b1$
and true false $\rightarrow_{\beta}^2 \text{true false true} \rightarrow_{\beta}^2 \text{false}$

or ::= $\lambda b1 \ b2. \text{if } b1 \ \text{true } b2 \text{ sau } \lambda b1 \ b2.b1 \ b1 \ b2$
or true false $\rightarrow_{\beta}^2 \text{true true false} \rightarrow_{\beta}^2 \text{true}$

not ::= $\lambda b. \text{if } b \ \text{false true sau } \lambda b \ t \ f.b \ f \ t$
not true $\rightarrow_{\beta} \lambda t \ f.\text{true } f \ t \rightarrow_{\beta} \lambda t \ f.f$

Perechi

Intuiție: Capabilitatea de a aplica o funcție componentelor perechii

Codificare: O funcție cu 3 argumente reprezentând componentele perechii și funcția ce vrem să o aplicăm lor.

pair ::= $\lambda x y. \lambda f. f\ x\ y$
Constructorul de perechi

Exemplu: **pair** 3 5 $\rightarrow_{\beta}^2 \lambda f. f\ 3\ 5$

perechea (3,5) reprezintă capabilitatea de a aplica o funcție de două argumente lui 3 și apoi lui 5.

Operații pe perechi

pair ::= $\lambda x y. \lambda f. f \ x \ y$

pair $xy \equiv_{\beta} f \ x \ y$

fst ::= $\lambda p. p \ true$ — *true* alege prima componentă

fst (**pair** 3 5) \rightarrow_{β} **pair** 3 5 *true* \rightarrow_{β}^3 *true* 3 5 \rightarrow_{β}^2 3

snd ::= $\lambda p. p \ false$ — *false* alege a doua componentă

snd (**pair** 3 5) \rightarrow_{β} **pair** 3 5 *false* \rightarrow_{β}^3 *false* 3 5 \rightarrow_{β}^2 5

Numere naturale

Intuiție: Capabilitatea de a itera o funcție de un număr de ori peste o valoare inițială

Codificare: Un număr natural este o funcție cu 2 argumente

s funcția care se iterează

z valoarea inițială

0 ::= $\lambda s z.z$ — s se iterează de 0 ori, deci valoarea inițială

1 ::= $\lambda s z.s z$ — funcția iterată o dată aplicată valorii inițiale

2 ::= $\lambda s z.s(s z)$ — s iterată de 2 ori, aplicată valorii inițiale

...

8 ::= $\lambda s z.s(s(s(s(s(s(s z))))))$

...

Observație: $0 = false$

Operații aritmetice de bază

0 ::= $\lambda s z.z$ — s se iterează de 0 ori, deci valoarea inițială

8 ::= $\lambda s z.s(s(s(s(s(s(s z))))))$

S ::= $\lambda n s z.s (n s z)$ sau $\lambda n s z.n s (sz)$

$S 0 \rightarrow_{\beta} \lambda s z.0s(sz) \rightarrow_{\beta}^2 \lambda s z.sz = 1$

+ ::= $\lambda m n.m S n$ sau $\lambda m n.\lambda s z.m s (n s z)$

$+ 3 2 \rightarrow_{\beta}^2 \lambda s z.3 s (2 s z) \rightarrow_{\beta}^2$

$\lambda s z.s(s(s(2 s z))) \rightarrow_{\beta}^2 \lambda s z.s(s(s(s z))) = 5$

***** ::= $\lambda m n.m (+ n) 0$ sau $\lambda m n.\lambda s.m (n s)$

$* 3 2 \rightarrow_{\beta}^2 3 (+ 2) 0 \rightarrow_{\beta}^2 + 2(+ 2(+ 2 0)) \rightarrow_{\beta}^4$

$+ 2(+ 2 2) \rightarrow_{\beta}^4 + 2 4 \rightarrow_{\beta}^4 6$

exp ::= $\lambda m n.n (* m) 1$ sau $\lambda m n.n m$

exp $3 2 \rightarrow_{\beta}^2 2 3 \rightarrow_{\beta}^2 \lambda z.3(3 z) \rightarrow_{\beta}$

$\lambda z.\lambda z'.3 z(3 z(3 z z')) \equiv_{\alpha} \lambda s z.3 s(3 s(3 s z)) \rightarrow_{\beta}^6$

$\lambda s z.s(s(s(s(s(s(s z)))))) = 9$

Scăderi și comparații

Presupunem că avem o funcție predecesor $\mathbf{P} \ x = \begin{cases} 0 & x = 0 \\ x - 1 & x \neq 0 \end{cases}$

$- ::= \lambda m \ n. n \ \mathbf{P} \ m$ — dă 0 dacă $m \leq n$

$?0 = ::= \lambda n. n(\lambda x. false) true$ — testează dacă n e 0

$<= ::= \lambda m \ n. ?0 = (- \ m \ n)$

$> ::= \lambda m \ n. \mathbf{not} (<= \ m \ n)$

$>= ::= \lambda m \ n. <= \ n \ m$

$< ::= \lambda m \ n. > \ n \ m$

$= ::= \lambda m \ n. \mathbf{and} (<= \ m \ n) (>= \ m \ n)$

$<> ::= \lambda m \ n. \mathbf{not} (= \ m \ n)$

Problemă

Cum definim funcția \mathbf{P} ?

Definirea funcției **P** folosind perechi

$$\mathbf{P} \ x = \begin{cases} 0 & x = 0 \\ x - 1 & x \neq 0 \end{cases}$$

$$\mathbf{P''} = \lambda n. n \ \mathbf{S''} \ (\mathbf{pair} \ 0 \ 0)$$

$$\mathbf{S''} = \lambda p. (\lambda x. \mathbf{pair} \ x (\mathbf{S} \ x)) \ (\mathbf{snd} \ p)$$

$$\mathbf{P} = \lambda n. \mathbf{fst} \ (\mathbf{P''} \ n)$$

$$\begin{aligned} \mathbf{P} \ 2 &\rightarrow_{\beta} \mathbf{fst} \ (\mathbf{P''} \ 2) \rightarrow_{\beta} \mathbf{fst} \ (2 \ \mathbf{S''} \ (\mathbf{pair} \ 0 \ 0)) \rightarrow_{\beta}^2 \\ &\mathbf{fst} \ (\mathbf{S''} \ (\mathbf{S''} \ (\mathbf{pair} \ 0 \ 0))) \rightarrow_{\beta} \mathbf{fst} \ (\mathbf{S''} \ (\mathbf{S''} \ (\mathbf{pair} \ 0 \ 0))) \rightarrow_{\beta} \\ &\mathbf{fst} \ (\mathbf{S''} \ ((\lambda x. \mathbf{pair} \ x (\mathbf{S} \ x)) \ (\mathbf{snd} \ (\mathbf{pair} \ 0 \ 0)))) \rightarrow_{\beta}^6 \\ &\mathbf{fst} \ (\mathbf{S''} \ ((\lambda x. \mathbf{pair} \ x (\mathbf{S} \ x)) \ 0)) \rightarrow_{\beta} \mathbf{fst} \ (\mathbf{S''} \ (\mathbf{pair} \ 0 (\mathbf{S} \ 0))) \rightarrow_{\beta}^8 \\ &\mathbf{fst} \ (\mathbf{pair} \ (\mathbf{S} \ 0) (\mathbf{S} \ (\mathbf{S} \ 0))) \rightarrow_{\beta}^6 \mathbf{S} \ 0 \rightarrow_{\beta}^3 1 \end{aligned}$$

Funcția predecesor — definiție alternativă directă

Idee 1: $P ::= \lambda n. ?0 = n \ 0 \ (P' \ n)$ — am tratat primul caz. acum vrem o funcție P' care calculează $n - 1$ dacă $n \neq 0$

Idee 2: folosim iterația $P' ::= \lambda n. n \ S' \ Z'$, unde

- S' e un fel de succesor
- Z' e un fel de -1 , i.e. $S' \ Z' = 0$

$S' ::= \lambda n. n \ S \ 1$

$Z' ::= \lambda s \ z. 0$ — Z' nu e codificarea unui număr
Dar se verifică că $S' \ Z' \rightarrow_{\beta} Z' \ S \ 1 \rightarrow_{\beta}^2 0$

Totul e OK deoarece P' e aplicată **doar** pe numere diferite de 0.

$P ::= \lambda n. ?0 = n \ 0 \ (n \ (\lambda n. n \ S \ 1) \ (\lambda s \ z. 0))$

Liste

Intuiție: Capabilitatea de a agrega o listă

Codificare: O funcție cu 2 argumente
funcția de agregare și valoarea inițială

null ::= $\lambda a \ i.i$ — lista vidă

cons ::= $\lambda x \ l.\lambda a \ i.a \ x \ (l \ a \ i)$
Constructorul de liste

Exemplu: $\text{cons } 3 \ (\text{cons } 5 \ \text{null}) \rightarrow_{\beta}^2 \lambda a \ i.a \ 3 \ (\text{cons } 5 \ \text{null} \ a \ i) \rightarrow_{\beta}^4 \lambda a \ i.a \ 3 \ (a \ 5 \ (\text{null} \ a \ i)) \rightarrow_{\beta}^2 \lambda a \ i.a \ 3 \ (a \ 5 \ i)$

Lista $[3, 5]$ reprezintă capabilitatea de a agrega elementele 3 și apoi 5 dată fiind o funcție de agregare a și o valoare implicită i .
Pentru aceste liste, operația de bază este `foldr`.

Operații pe liste

null ::= $\lambda a i.i$ — lista vidă

cons ::= $\lambda x l.\lambda a i.a x (l a i)$

?null ::= $\lambda l.l (\lambda x v.false) true$

head ::= $\lambda d l.l (\lambda x v.x) d$
primul element al listei, sau d dacă lista e vidă

tail ::= $\lambda l.fst (l (\lambda x p.pair (snd p) (cons x (snd p))) (pair null null))$
coada listei, sau lista vidă dacă lista e vidă

foldr ::= $\lambda f i l.l f i$

map ::= $\lambda f l.l (\lambda x t.cons (f x) t) null$

filter ::= $\lambda p l.l (\lambda x t.p x (cons x t) t) null$

Factorial, Fibonacci, împărțire folosind perechi

fact ::= $\lambda n. \text{snd } (n (\lambda p. \text{pair } (\text{S } (\text{fst } p)) (* (\text{fst } p) (\text{snd } p))) (\text{pair } 1 \ 1))$

fib ::= $\lambda n. \text{fst } (n (\lambda p. \text{pair } (\text{snd } p) (+ (\text{fst } p) (\text{snd } p))) (\text{pair } 0 \ 1))$

divMod ::= $\lambda m \ n. m (\lambda p. > n (\text{snd } p) \ p (\text{pair } (\text{S } (\text{fst } p)) (- (\text{snd } p) \ n))) (\text{pair } 0 \ m)$

Observații

- ❑ Nu toate funcțiile pot fi definite prin iterare fixată
- ❑ Pentru **divMod** obținem răspunsul (de obicei) din mult mai puțin de m iterații

Recursie

- există termeni care **nu** pot fi reduși la o β -formă normală, de exemplu

$$(\lambda x. x x)(\lambda x. x x) \rightarrow_{\beta} (\lambda x. x x)(\lambda x. x x)$$

În λ -calcul putem defini calcule infinite!

- Dacă notăm $Af ::= \lambda x. f (x x)$ atunci
 $Af Af =_{\beta} (\lambda x. f (x x)) Af =_{\beta} f (Af Af)$
- Dacă notăm $Yf ::= Af Af$ atunci $Yf =_{\beta} f Yf$.
- Fie $Y ::= \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$
Atunci $Y f =_{\beta} f (Y f)$

Puncte fixe

□ pentru o funcție $f : X \rightarrow X$ un **punct fix** este un element $x_0 \in X$ cu $f(x_0) = x_0$.

□ $f : \mathbb{N} \rightarrow \mathbb{N}, f(x) = x + 1$ nu are puncte fixe

□ $f : \mathbb{N} \rightarrow \mathbb{N}, f(x) = 2x$ are punctul fix $x = 0$

□ În λ -calcul

$$\mathbf{Y} ::= \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

are proprietatea că $Y f =_{\beta} f (Y f)$, deci $Y f$ este un **punct fix** pentru f .

Y se numește **combinator de punct fix**.

$$\text{Avem } Y f =_{\beta} f (Y f) =_{\beta} f (f (Y f)) =_{\beta} \dots$$

Putem folosi Y pentru a obține apeluri recursive!

Puncte fixe - funcția factorial

fact ::= $\lambda n. \text{if } (?0 = n) \ 1 \ (* \ n \ (\text{fact} \ (\mathbf{P} \ n)))$

Această definiție nu este corectă conform regulilor noastre, cum procedăm?

- Pasul 1: abstractizăm, astfel încât construcția să fie corectă

factA ::= $\lambda f. \lambda n. \text{if } (?0 = n) \ 1 \ (* \ n \ (\textcolor{red}{f} \ (\mathbf{P} \ n)))$

- Pasul 2: aplicăm combinatorul de punct fix

fact ::= $Y \ \text{factA}$

Deoarece $Y \ \text{factA} =_{\beta} \text{factA} \ (Y \ \text{factA})$ obținem

$\text{fact} =_{\beta} \lambda n. \text{if } (?0 = n) \ 1 \ (* \ n \ (\text{fact} \ (\mathbf{P} \ n)))$

divMod — definiție recursivă

Definiția primitiv recursivă (fără Y)

divMod ::= $\lambda m n. m (\lambda p. > n (\mathbf{snd} p) p (\mathbf{pair} (\mathbf{S} (\mathbf{fst} p)) (- (\mathbf{snd} p) n))) (\mathbf{pair} 0 m)$

Definiția recursivă (incorectă)

divMod ::= $\lambda m n. ?0 = n (\mathbf{pair} 0 m) (\mathbf{divMod}' 0 m)$ where
divMod' ::= $\lambda q r. > n r (\mathbf{pair} q r) (\mathbf{divMod}' (\mathbf{S} q) (- m n))$

Definiția recursivă (corectă, folosind Y)

divMod ::= $\lambda m n. ?0 = n (\mathbf{pair} 0 m)$
 $(\mathbf{Y} (\lambda f. \lambda q r. > n r (\mathbf{pair} q r) (f (\mathbf{S} q) (- m n))))$
 $0 m)$

Apel prin valoare (Call by Value)

Pentru a reduce $e_1 \ e_2$:

- Reducem e_1 până la o funcție $\lambda x.e$
- Apoi reducem e_2 până la o valoare v
- Apoi reducem $(\lambda x.e) \ v$ la $[v/x]e$

Nu simplificăm sub λ

$\begin{aligned} &?0 = 0 \ (+ \ 2 \ 1) \ (+ \ 3 \ 4) \\ &\rightarrow_{\beta(V)} 0 \ (\lambda x.false) \ true \ (+ \ 2 \ 1) \ (+ \ 3 \ 4) \\ &\rightarrow_{\beta(V)}^2 true \ (+ \ 2 \ 1) \ (+ \ 3 \ 4) \\ &\rightarrow_{\beta(V)}^2 true \ (\lambda s \ z.2 \ s \ (1 \ s \ z)) \ (+ \ 3 \ 4) \\ &\rightarrow_{\beta(V)} (\lambda f.\lambda s \ z.2 \ s \ (1 \ s \ z)) \ (+ \ 3 \ 4) \\ &\rightarrow_{\beta(V)}^2 (\lambda f.\lambda s \ z.2 \ s \ (1 \ s \ z)) \ (\lambda s \ z.3 \ s \ (4 \ s \ z)) \\ &\rightarrow_{\beta(V)} \lambda s \ z.2 \ s \ (1 \ s \ z) \\ &\rightarrow_{\beta(V)} \end{aligned}$	$\begin{aligned} &?0 ::= (\lambda n.n \ (\lambda x.false) \ true) \\ &0 ::= (\lambda s \ z.z) \\ &+ ::= (\lambda m \ n \ s \ z.m \ s \ (n \ s \ z)) \\ &true ::= \lambda t \ f.t \end{aligned}$
---	---

Apel prin nume (Limbaje pur funcționale)

Pentru a reduce $e_1 \ e_2$:

- Reducem e_1 până la o funcție $\lambda x.e$
- Apoi reducem $(\lambda x.e) \ e_2$ la $[e_2/x]e$

Nu simplificăm sub λ nici în dreapta aplicației

?0 = $0 \ (+ \ 2 \ 1) \ (+ \ 3 \ 4)$

$\rightarrow_{\beta(N)}^2 0 \ (\lambda x.false) \ true \ (+ \ 2 \ 1) \ (+ \ 3 \ 4)$

$\rightarrow_{\beta(N)}^2 true \ (+ \ 2 \ 1) \ (+ \ 3 \ 4)$

$\rightarrow_{\beta(N)}^2 (+ \ 2 \ 1)$

$\rightarrow_{\beta(N)}^2 (\lambda s \ z.2 \ s \ (1 \ s \ z))$

$\rightarrow_{\beta(N)}^2$

?0 ::= $(\lambda n.n \ (\lambda x.false) \ true)$

$0 ::= (\lambda s \ z.z)$

$true ::= \lambda t \ f.t$

$+ ::= (\lambda m \ n \ s \ z.m \ s \ (n \ s \ z))$

Evaluare leneșă

Pentru a reduce $e_1 \ e_2$:

- Reduc e_1 până la o funcție $\lambda x.e$
- Apoi reduc corpul funcției e până la un e' care are nevoie de x
- Apoi reduc e_2 până la o valoare v
- Apoi reduc $(\lambda x.e')$ v la $[v/x]e'$

Evaluare nerestricționată

Pentru a reduce $e_1 \ e_2$

- Reducem fie e_1 fie e_2
- Putem reduce corpurile funcțiilor
- Oricând avem $(\lambda x.e')e''$, o putem (sau nu) reduce la $[e''/x]e'$
- Reduce până la o formă normală

Nu garantează găsirea unei forme normale

Evaluare „normală“

Pentru a reduce $e_1 \ e_2$

- Reducem mereu cel mai din stânga redex din cele de mai sus
- Reducem $(\lambda x.e')e''$ la $[e''/x]e'$
- Reducem e_1 (putem reduce și corpurile funcțiilor)
- Dacă $e_1 \rightarrow$, reducem e_2
- Reduce până la o formă normală
- Garantează găsirea unei forme normale

Apel prin valoare

Pentru a reduce $e_1 \ e_2$:

- Reducem e_1 până la o funcție $\lambda x.e$
- Apoi reducem e_2 până la o valoare v
- Apoi reducem $(\lambda x.e) \ v$ la $[v/x]e$

Reguli

$$(V@S) \quad \frac{e_1 \rightarrow_{\beta} e'_1}{e_1 \ e_2 \rightarrow_{\beta} e'_1 \ e_2}$$

$$(V@D) \quad \frac{e_2 \rightarrow_{\beta} e'_2}{(\lambda x.e_1) \ e_2 \rightarrow_{\beta} (\lambda x.e_1) \ e'_2}$$

$$(V@) \quad (\lambda x.e_1) \ v_2 \rightarrow_{\beta} e \quad \text{dacă } e = [v_2/x]e_1$$

Apel prin nume

Pentru a reduce $e_1 \ e_2$:

- Reducem e_1 până la o funcție $\lambda x.e$
- Apoi reducem $(\lambda x.e) \ e_2$ la $[e_2/x]e$

Reguli

$$(N@S) \quad \frac{e_1 \rightarrow_{\beta} e'_1}{e_1 \ e_2 \rightarrow_{\beta} e'_1 \ e_2}$$

$$(N@) \quad (\lambda x.e_1) \ e_2 \rightarrow_{\beta} e \quad \text{dacă } e = [e_2/x]e_1$$

Evaluare nerestricționată

Pentru a reduce $e_1 \ e_2$

- Reducem fie e_1 fie e_2
- Putem reduce corpurile funcțiilor
- Oricând avem $(\lambda x.e')e''$, o putem (sau nu) reduce la $[e''/x]e'$

Reguli

$$(NR@S) \quad \frac{e_1 \rightarrow e'_1}{e_1 \ e_2 \rightarrow e'_1 \ e_2}$$

$$(NR@D) \quad \frac{e_2 \rightarrow e'_2}{e_1 \ e_2 \rightarrow e_1 \ e'_2}$$

$$(NR_{\text{FUND}}) \quad \frac{e \rightarrow e'}{\lambda x.e \rightarrow \lambda x.e'}$$

$$(NR@) \quad (\lambda x.e_1) \ e_2 \rightarrow e \quad \text{dacă } e = [e_2/x]e_1$$

Evaluare „normală“

Pentru a reduce $e_1 \ e_2$

- Reducem mereu cel mai din stânga redex din cele de mai sus
- Reducem $(\lambda x.e')e''$ la $[e''/x]e'$
- Reducem e_1 (putem reduce și corpurile funcțiilor)
- Dacă $e_1 \rightarrow$, reducem e_2

Reguli

$$(\text{NOR@}) \quad (\lambda x.e_1) \ e_2 \rightarrow e \quad \text{dacă } e = [e_2/x]e_1$$

$$(\text{NOR@S}) \quad \frac{e_1 \rightarrow e'_1}{e_1 \ e_2 \rightarrow e'_1 \ e_2} \quad \text{dacă } e_1 \text{ nu e încă funcție}$$

$$(\text{NORFUND}) \quad \frac{e \rightarrow e'}{\lambda x.e \rightarrow \lambda x.e'}$$

$$(\text{NOR@D}) \quad \frac{e_1 \rightarrow \quad e_2 \rightarrow e'_2}{e_1 \ e_2 \rightarrow e_1 \ e'_2}$$

Evaluare leneșă

Pentru a reduce $e_1 \ e_2$:

- ❑ Reduc e_1 până la o funcție **fun** $(x : T) \rightarrow e$
- ❑ Apoi reduc corpul funcției e până la un e' care are nevoie de x
- ❑ Apoi reduc e_2 până la o valoare v
- ❑ Apoi reduc $(\lambda x.e') \ v$ la $[v/x]e'$

Reguli?

E mai complicat decât pare, deoarece trebuie să ne dăm seama că e' are nevoie de x .

Contexte de evaluare

- Găsirea redex-ului prin analiză sintactică
- Putem înlocui regulile structurale cu reguli gramaticale

Sintaxă: $e ::= x \mid \lambda x.e \mid e e$

Reguli structurale

$$\frac{e_1 \longrightarrow_{\beta} e'_1}{e_1 e_2 \longrightarrow_{\beta} e'_1 e_2}$$
$$\frac{e_2 \longrightarrow_{\beta} e'_2}{\langle (\lambda x.e_1) e_2 \rangle \longrightarrow_{\beta} (\lambda x.e_1) e'_2}$$

Contexte de evaluare

$$c ::= \blacksquare$$
$$| c e$$
$$| (\lambda x.e) c$$

Instanțierea unui context c cu expresia e

$$c[e] = [e/\blacksquare]c$$

Contexte de evaluare

Sintaxă: $e ::= x \mid \lambda x.e \mid e e$

Contexte: $c ::= \blacksquare \mid c e \mid (\lambda x.e) c$

Exemple de contexte

Corecte

\blacksquare

$3 \blacksquare$

Greșite

5

$true x \blacksquare$

Exemple de contexte instanțiate

$\square \blacksquare[x \ 1] = x \ 1$

$\square (9 (\blacksquare 7))[x] = 9 (x \ 7)$

$\square (9 (\blacksquare 7))[5] = 9 (5 \ 7)$

Semantica Operațională Contextuală

Un pas de execuție folosind contexte de evaluare

- Descompune expresia în contextul c și redex-ul r
- Aplică o regulă operațională asupra lui r obținând e
- Pune e în contextul inițial, obținând $c[e]$

$$\frac{r \longrightarrow_{\beta} e}{c[r] \longrightarrow_{\beta} c[e]}$$

Evaluare leneșă folosind Semantica Contextuală

Contexte de evaluare pentru aplicație

$$\begin{array}{lcl} c & ::= & \blacksquare \\ & | & c \ e \\ & | & (\lambda x. c) \ e \\ & | & (\lambda x. c[x]) \ c \end{array}$$

Regulă de evaluare pentru aplicație

$$(\lambda x. c[x]) \ v \rightarrow (\lambda x. c[v]) \ v$$

Curs 8

2020-2021

Fundamentele limbajelor de programare

Cuprins

- 1 Programare logică & Prolog
- 2 Tipuri de date compuse
- 3 Liste și recursie
- 4 Exemplu: reprezentarea unei GIC

Programare logică & Prolog

Programare logică

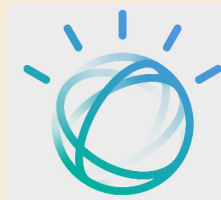
- Programarea logică este o paradigmă de programare bazată pe logică formală.
- Unul din sloganurile programării logice:
Program = Logică + Control (R. Kowalski)
- Programarea logică poate fi privită ca o deducție controlată.
- Un program scris într-un limbaj de programare logică este
o listă de formule într-o logică
ce exprimă fapte și reguli despre o problemă.
- Exemple de limbaje de programare logică:
 - Prolog
 - Answer set programming (ASP)
 - Datalog

Ce veți vedea la laborator

Prolog

- ☐ bazat pe logica clauzelor Horn
- ☐ semantica operațională este bazată pe rezoluție
- ☐ este Turing complet
- ☐ vom folosi implementarea [SWI-Prolog](#)

Limbajul Prolog este folosit pentru programarea sistemului IBM Watson!



Puteți citi mai multe detalii [aici](#).

[Learn Prolog Now!http://www.let.rug.nl/bos/lpn/](http://www.let.rug.nl/bos/lpn/)

Programare logică - în mod idealist

- Un "program logic" este o colecție de proprietăți presupuse (sub formă de formule logice) despre lume (sau mai degrabă despre lumea programului).
- Programatorul furnizează și 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 consecință a proprietăților presupuse în program.
- Programatorul nu specifică metoda prin care sistemul verifică dacă întrebarea este sau nu consecință a programului.

Exemplu de program logic

oslo \rightarrow windy
oslo \rightarrow norway
norway \rightarrow cold
cold \wedge windy \rightarrow winterIsComing
oslo

Exemplu de întrebare

Este adevărat `winterIsComing`?

Putem să testăm în SWI-Prolog

Program:

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

Intrebare:

```
?- winterIsComing.  
true
```

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

Sintaxă: constante, variabile, termeni compuși

- **Atomii**: `sansa`, `'Jon Snow'`, `jon_snow`
- **Numere**: `23`, `23.03`, `-1`
Atomii și **numerele** sunt **constante**.
- **Variabile**: `X`, `Stark`, `_house`
- Termeni **compuși**: `father(eddard, jon_snow)`,
`and(son(bran, eddard), daughter(arya, eddard))`
 - forma generală: `atom(termin,..., termen)`
 - atom-ul care denumește termenul se numește **functor**
 - numărul de argumente se numește **aritate**



Un mic exercițiu sintactic

Care din următoarele șiruri de caractere sunt **constante** și care sunt **variabile** în Prolog?

- ☐ vINCENT
- ☐ Footmassage
- ☐ variable23
- ☐ Variable2000
- ☐ big_kahuna_burger
- ☐ 'big kahuna burger'
- ☐ big kahuna burger
- ☐ 'Jules'
- ☐ _Jules
- ☐ '_Jules'

Un mic exercițiu sintactic

Care din următoarele șiruri de caractere sunt **constante** și care sunt **variabile** în Prolog?

- ☐ vINCENT – **constantă**
- ☐ Footmassage – **variabilă**
- ☐ variable23 – **constantă**
- ☐ Variable2000 – **variabilă**
- ☐ big_kahuna_burger – **constantă**
- ☐ 'big kahuna burger' – **constantă**
- ☐ big kahuna burger – **nici una, nici alta**
- ☐ 'Jules' – **constantă**
- ☐ _Jules – **variabilă**
- ☐ '_Jules' – **constantă**

Program în Prolog = bază de cunoștințe

Exemplu

Un program în Prolog:

```
father(eddard, sansa).  
father(eddard, jon_snow).
```

```
mother(catelyn, sansa).  
mother(wylla, jon_snow).
```

```
stark(eddard).  
stark(catelyn).
```

```
stark(X) :- father(Y,X), stark(Y).
```



Un program în Prolog este o **bază de cunoștințe** (Knowledge Base).

Program în Prolog = mulțime de predicate

Practic, gândim un program în Prolog ca o mulțime de **predicate** cu ajutorul cărora descriem *lumea (universul)* programului respectiv.

Exemplu

```
father(eddard, sansa) .  
father(eddard, jon_snow) .
```

```
mother(catelyn, sansa) .  
mother(wylla, jon_snow) .
```

```
stark(eddard) .  
stark(catelyn) .
```

```
stark(X) :- father(Y,X), stark(Y) .
```

Predicate:

father/2
mother/2
stark/1

Un program în Prolog

Program

Fapte + Reguli

Program

- Un **program** în Prolog este format din **reguli** de forma
Head :- Body.
- **Head** este un predicat, iar **Body** este o secvență de predicate separate prin virgulă.
- Regulile fără Body se numesc **fapte**.

Exemplu

- Exemplu de regulă: `stark(X) :- father(Y,X), stark(Y).`
- Exemplu de fapt: `father(eddard, jon_snow).`

Interpretarea din punctul de vedere al logicii

- operatorul `:-` este implicația logică \leftarrow

Exemplu

`winterfell(X) :- stark(X)`

dacă `stark(X)` *este adevărat, atunci* `winterfell(X)` *este adevărat.*

- virgula `,` este conjuncția \wedge

Exemplu

`stark(X) :- father(Y,X), stark(Y)`

dacă `father(Y,X)` *și* `stark(Y)` *sunt adevărate,*
atunci `stark(X)` *este adevărat.*

Interpretarea din punctul de vedere al logicii

- mai multe reguli cu același Head definesc același predicat, între definiții fiind un sau logic.

Exemplu

```
got_house(X) :- stark(X).  
got_house(X) :- lannister(X).  
got_house(X) :- targaryen(X).  
got_house(X) :- baratheon(X).
```

dacă

stark(X) este adevărat sau lannister(X) este adevărat sau
targaryen(X) este adevărat sau baratheon(X) este adevărat,
atunci
got_house(X) este adevărat.

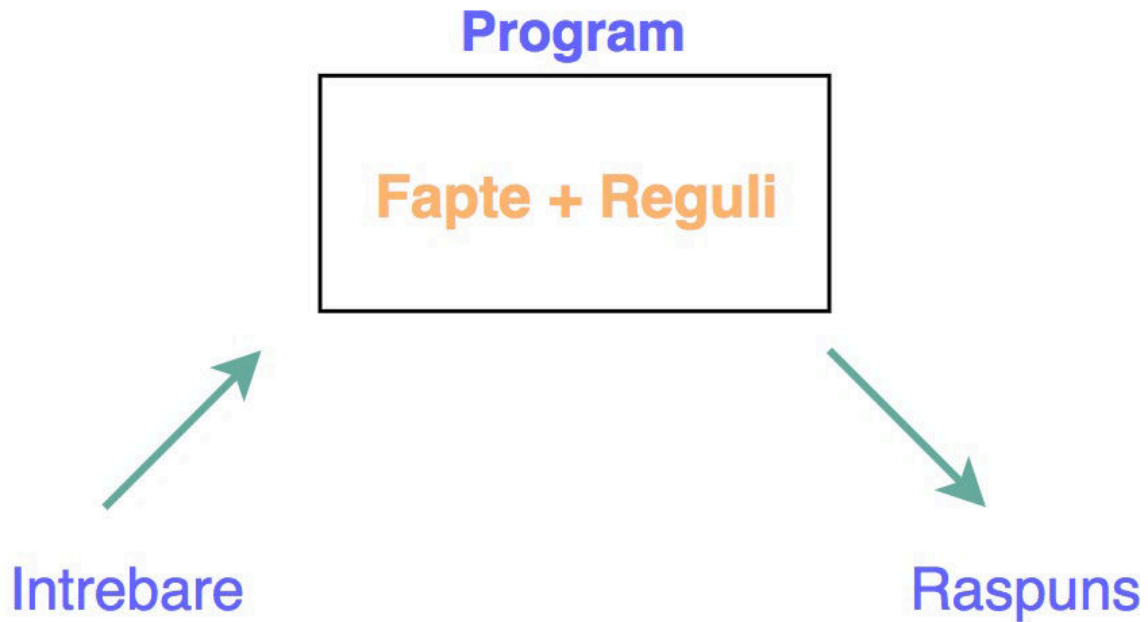
Un program în Prolog

Program

Fapte + Reguli

Cum folosim un program în Prolog?

Întrebări în Prolog



Întrebări și ținte în Prolog

- Prolog poate răspunde la întrebări legate de consecințele relațiilor descrise într-un program în Prolog.
- **Întrebările** sunt de forma:
$$?- \text{predicat}_1(\dots), \dots, \text{predicat}_n(\dots).$$
- Prolog verifică dacă întrebarea este o consecință a relațiilor definite în program.
- Dacă este cazul, Prolog caută valori pentru variabilele care apar în întrebare astfel încât întrebarea să fie o consecință a relațiilor din program.
- un predicat care este analizat pentru a se răspunde la o întrebare se numește **țintă** (**goal**).

Întrebări în Prolog

Prolog poate da 2 tipuri de răspunsuri:

- ❑ **false** – în cazul în care întrebarea nu este o consecință a programului.
- ❑ **true** sau **valori pentru variabilele din întrebare** în cazul în care întrebarea este o consecință a programului.

Exemplu

```
?- stark(jon_snow)
true
?- stark(wylla)
false
```

```
?- stark(X)
X = eddard ;
X = catelyn ;
X = sansa ;
X = jon_snow ;
false
```

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, Prolog încearcă regulile în ordinea apariției lor.

Exemplu

Să presupunem că avem programul:

```
foo(a).  foo(b).  foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).
```

```
X = a.
```

Pentru a răspunde la întrebare se caută o potrivire (unificator) între scopul `foo(X)` și baza de cunoștințe. Răspunsul este substituția care realizează potrivirea, în cazul nostru `X = a`.

Răspunsul la întrebare este găsit prin unificare!

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, Prolog încearcă regulile în ordinea apariției lor.

Exemplu

Să presupunem că avem programul:

```
foo(a).  foo(b).  foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).
```

```
X = a.
```

```
?- foo(d).
```

```
false
```

Dacă nu se poate face potrivirea, răspunsul este **false**.

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, Prolog încearcă regulile în ordinea apariției lor.

Exemplu

Să presupunem că avem programul:

```
foo(a).  foo(b).  foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).
```

```
X = a.
```

Dacă dorim mai multe răspunsuri, tastăm ;

```
?- foo(X).
```

```
X = a ;
```

```
X = b ;
```

```
X = c.
```

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, **Prolog încearcă regulile în ordinea apariției lor.**

Exemplu

Să presupunem că avem programul:

`foo(a).`

`foo(b).`

`foo(c).`

și că punem următoarea întrebare:

`?- foo(X).`

```
?- trace.  
true.  
  
[trace] ?- foo(X).  
    Call: (8) foo(_4556) ? creep  
    Exit: (8) foo(a) ? creep  
X = a ;  
    Redo: (8) foo(_4556) ? creep  
    Exit: (8) foo(b) ? creep  
X = b ;  
    Redo: (8) foo(_4556) ? creep  
    Exit: (8) foo(c) ? creep  
X = c.
```

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, Prolog încearcă regulile în ordinea apariției lor.

Exemplu

Să presupunem că avem programul:

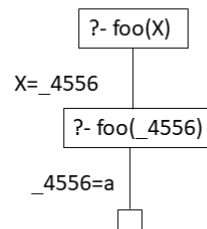
`foo(a).`

`foo(b).`

`foo(c).`

și că punem următoarea întrebare:

`?- foo(X).`



În acest moment, a fost găsită prima soluție: `X=_4556=a`.

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, **Prolog încearcă clauzele în ordinea apariției lor.**

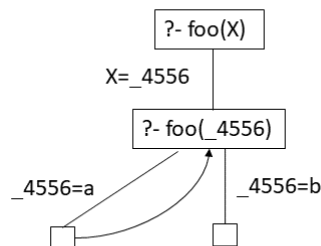
Exemplu

Să presupunem că avem programul:

```
foo(a).  
foo(b).  
foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).
```



Dacă se dorește încă un răspuns, atunci se face un pas înapoi în **arborele de căutare** și se încearcă satisfacerea țintei cu o nouă valoare.

Cum găsește Prolog răspunsul

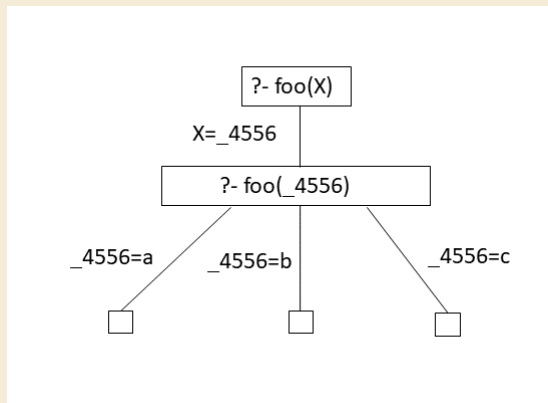
Pentru a găsi un răspuns, Prolog încearcă clauzele în ordinea apariției lor.

Exemplu

Să presupunem că avem programul:

```
foo(a).  
foo(b).  
foo(c).
```

și că punem următoarea întrebare:
`?- foo(X).`



arborele de căutare

Cum găsește Prolog răspunsul

Prolog se întoarce la ultima alegere dacă o sub-țintă eșuează.

Exemplu

Să presupunem că avem programul:

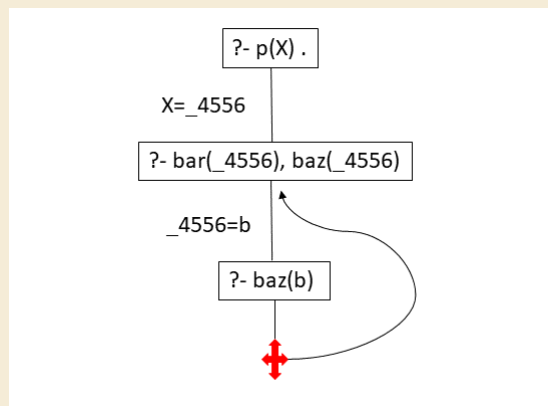
`bar(b) .`

`bar(c) .`

`baz(c) .`

și că punem următoarea întrebare:

`?- bar(X), baz(X) .`



Cum găsește Prolog răspunsul

Prolog se întoarce la ultima alegere dacă o sub-țintă eșuează.

Exemplu

Să presupunem că avem programul:

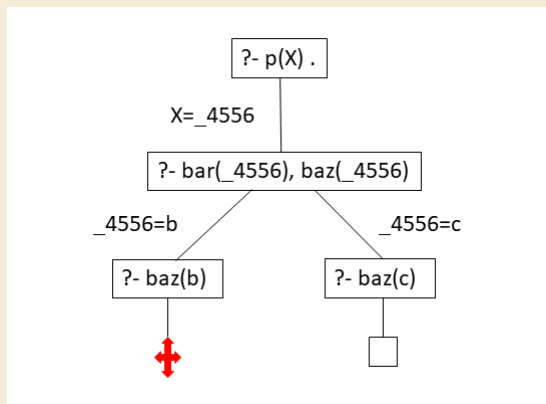
`bar(b) .`

`bar(c) .`

`baz(c) .`

și că punem următoarea întrebare:

`?- bar(X), baz(X) .`



Soluția găsită este: `X=_4556=c`.

Cum găsește Prolog răspunsul

Ce se întâmplă dacă schimbăm ordinea regulilor?

Exemplu

Să presupunem că avem programul:

```
bar(c) .
```

```
bar(b) .
```

```
baz(c) .
```

și că punem următoarea întrebare:

```
?- bar(X), baz(X) .
```

```
X = c ;
```

```
false
```

Vă explicați ce s-a întâmplat? Desenați arborele de căutare!

Un program mai complicat

Problema colorării hărților

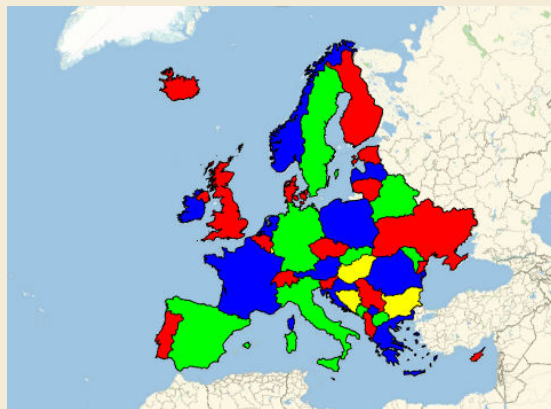
Să se coloreze o hartă dată cu un număr minim de culori astfel încât oricare două țări vecine să fie colorate diferit.

Cum modelăm această problemă în Prolog?

Exemplu

Trebuie să definim:

- ☐ culorile
- ☐ harta
- ☐ constrângerile



Sursa imaginii

Problema colorării hărților

Exemplu

```
culoare(albastru).
```

```
culoare(rosu).
```

```
culoare(verde).
```

```
culoare(galben).
```

```
harta(R0,SE,MD,UA,BG,HU) :-    vecin(R0,SE), vecin(R0,UA),  
                                vecin(R0,MD), vecin(R0,BG),  
                                vecin(R0,HU), vecin(UA,MD),  
                                vecin(BG,SE), vecin(SE,HU).
```

```
vecin(X,Y) :- culoare(X),  
               culoare(Y),  
               X \== Y.
```

```
?- harta(R0,SE,MD,UA,BG,HU).
```

```
R0 = albastru,
```

```
SE = UA, UA = rosu,
```

```
MD = BG, BG = HU, HU = verde ■
```

Compararea termenilor: $=, \backslash=, ==, \backslash==$

$T = U$	reuşeşte dacă există o potrivire (termenii se unifică)
$T \backslash= U$	reuşeşte dacă nu există o potrivire
$T == U$	reuşeşte dacă termenii sunt identici
$T \backslash== U$	reuşeşte dacă termenii sunt diferiţi

Exemplu

?- $X = Y$.

$X = Y$.

?- $X == Y$.

false

?- $p(X, q(Z)) = p(Y, X)$.

$X = Y, Y = q(Z)$.

?- $p(X, Y) == p(X, Y)$.

true

?- $2 = 1 + 1$

false

?- $2 == 1 + 1$

false

- În exemplul de mai sus, $1+1$ este privită ca o expresie, nu este evaluată. Există şi predicate care forţează evaluarea.

Negarea unui predicat: $\backslash + \text{pred}(X)$

Exemplu

```
animal(dog). animal(elephant). animal(sheep).
```

```
?- animal(cat).
```

```
false
```

```
?-  $\backslash +$  animal(cat).
```

```
true
```

- Clauzele din Prolog dau doar condiții suficiente, dar nu și necesare pentru ca un predicat să fie adevărat.
- Pentru a da un răspuns pozitiv la o țintă, Prolog trebuie să construiască o "demonstrație" pentru a arată că mulțimea de fapte și reguli din program implică acea țintă.
- Astfel, un răspuns **false** nu înseamnă neapărat că ținta nu este adevărată, ci doar că **Prolog nu a reușit să găsească o demonstrație**.

Operatorul \+

- Negarea unei ținte se poate defini astfel:

```
neg(Goal) :- Goal, !, fail.  
neg(Goal)
```

unde **fail/0** este un predicat care eșuează întotdeauna.

- În PROLOG acest predicat este predefinit sub numele \+.
- Operatorul \+ se folosește pentru a nega un predicat.
- **!(cut)** este un predicat predefinit (de aritate 0) care restricționează mecanismul de backtracking: execuția subținteii ! se termină cu succes, deci alegerile (instantierile) făcute înaintea de a se ajunge la ! nu mai pot fi schimbate.
- O țintă \+ Goal reușește dacă Prolog nu găsește o demonstrație pentru Goal. Negatia din Prolog este definită ca incapacitatea de a găsi o demonstrație.
- Semantica operatorului \+ se numește **negation as failure**.

Negația ca eșec ("negation as failure")

Exemplu

Să presupunem că avem o listă de fapte cu perechi de oameni căsătoriți între ei:

```
married(peter, lucy).  
married(paul, mary).  
married(bob, juliet).  
married(harry, geraldine).
```

Negația ca eșec

Exemplu (cont.)

Putem să definim un predicat `single/1` care reușește dacă argumentul său nu este nici primul nici al doilea argument în faptele pentru `married`.

```
single(Person) :-  
    \+ married(Person, _),  
    \+ married(_, Person).  
  
?- single(mary).      ?- single(anne).      ?- single(X).  
false                 true                  false
```

Răspunsul la întrebarea `?- single(anne).` trebuie gândit astfel:

Presupunem că Anne este single,
deoarece nu am putut demonstra că este maritată.

Predicatul $\rightarrow /2$ (if-then-else)

□ if-then

`If \rightarrow Then :- If, !, Then.`

□ if-then-else

`If \rightarrow Then; _Else :- If, !, Then.`

`If \rightarrow Then; Else :- !, Else.`

Se încearcă demonstrarea predicatului If. Dacă întoarce true atunci se încearcă demonstrarea predicatului Then, iar dacă întoarce false se încearcă demonstrarea predicatului Else.

`max(X,Y,Z) :- (X =< Y) \rightarrow Z = Y ; Z = X`

`?- max(2,3,Z).`

`Z = 3.`

Observăm că `If \rightarrow Then` este echivalent cu `If \rightarrow Then ; fail.`

Tipuri de date compuse

Termeni compuși $f(t_1, \dots, t_n)$

- **Termenii** sunt unitățile de bază prin care Prolog reprezintă datele.
- Sunt de 3 tipuri:
 - **Constante**: 23, sansa, 'Jon Snow'
 - **Variable**: X, Stark, _house
 - **Termeni compuși**:
 - predicate
 - termeni prin care reprezentăm datele

Exemplu

- `born(john, date(20,3,1977))`
 - `born/2` și `date/3` sunt functori
 - `born/2` este un predicat
 - `date/3` definește date compuse

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
 - [] este listă
 - [X|L] este listă, unde X este element și L este listă

- Cum definim arborii binari în Prolog? Soluție posibilă:
 - void este arbore
 - tree(X,A1,A2) este arbore, unde X este un element, iar A1 și A2 sunt arbori

`tree(X,A1,A2)` este un termen compus, dar nu este un predicat!

Arbori binari în Prolog

- Cum arată un arbore?

```
tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))
```

- Cum dăm un "nume" arborelui de mai sus? Definim un predicat:

```
def(arb, tree(a, tree(b,  
                    tree(d,void,void),  
                    void),  
    tree(c, void,  
        tree(e,void,void)))).
```

Deoarece în Prolog nu avem declarații explicite de date, pentru a defini arborii vom scrie un predicat care este adevărat atunci când argumentul său este un arbore.

Arbori binari în Prolog

- Scrieți un predicat care verifică că un termen este arbore binar.

```
binary_tree(void).  
binary_tree(tree(Element,Left,Right)) :- binary_tree(Left),  
                                         binary_tree(Right).
```

Eventual putem defini și un predicat pentru elemente:

```
binary_tree(void).  
binary_tree(tree(Element,Left,Right)) :- binary_tree(Left),  
                                         binary_tree(Right),  
                                         element_binary_tree(Element)  
  
element_binary_tree(X):- integer(X). /* de exemplu */  
  
test:- def(arb,T), binary_tree(T).
```


Arbori binari în Prolog

Exercițiu

Scrieți un predicat care verifică că un element aparține unui arbore.

```
tree_member(X,tree(X,Left,Right)).
```

```
tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Left).
```

```
tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Right).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care determină parcurgerea în preordine a unui arbore binar.

```
preorder(tree(X,L,R),Xs) :- preorder(L,Ls),
                                preorder(R,Rs),
                                append([X|Ls],Rs,Xs).
```

```
preorder(void,[]).
```

```
test(Tree,Pre):- def(arb, Tree), preorder(Tree,Pre).
```

```
?- test(T,P).
```

```
T = tree(a, tree(b, tree(d, void, void), void), tree(c,
void, tree(e, void, void))),
```

```
P = [a, b, d, c, e]
```

Liste și recursie

Listă $[t_1, \dots, t_n]$

- O listă în Prolog este un șir de elemente, separate prin virgulă, între paranteze drepte:

`[1,cold, parent(jon),[winter,is,coming],X]`

- O listă poate conține termeni de orice fel.
- Ordinea termenilor din listă are importanță:

?- `[1,2] == [2,1] .`

`false`

- Lista vidă se notează `[]`.
- Simbolul `|` delimitează coada listei:

?- `[1,2,3,4,5,6] = [X|T] .`

`X = 1, T = [2, 3, 4, 5, 6] .`

?- `[1,2,3|[4,5,6]] == [1,2,3,4,5,6] .`

`true.`

Listă $[t_1, \dots, t_n] == [t_1 \mid [t_2, \dots, t_n]]$

- Simbolul \mid delimitează coada listei:

?- $[1, 2, 3, 4, 5, 6] = [X \mid T]$.

$X = 1,$

$T = [2, 3, 4, 5, 6].$

- Variabila anonimă $_$ este unificată cu orice termen Prolog:

?- $[1, 2, 3, 4, 5, 6] = [X \mid _]$.

$X = 1.$

- Deoarece Prologul face unificare poate identifica șabloane mai complicate:

?- $[5, 1, 1, 3, 2] = [_ \mid [X \mid [X \mid _]]]$.

$X = 1.$

?- $[5, 1, 4, 3, 2] = [_ \mid [X \mid [X \mid _]]]$.

$false.$

Exercițiu

- Definiți un predicat care verifică că un termen este lista.

```
is_list([]).
```

```
is_list([_|_]).
```

- Definiți predicate care verifică dacă un termen este primul element, ultimul element sau coada unei liste.

```
head([X|_],X).
```

```
last([X],X).
```

```
last([_|T],Y):- last(T,Y).
```

```
tail([],[]).
```

```
tail([_|T],T).
```

Exercițiu

- Definiți un predicat care verifică dacă un termen aparține unei liste.
`member(H, [H|_]) .`
`member(H, [_|T]) :- member(H,T) .`
- Definiți un predicat `append/3` care verifică dacă o listă se obține prin concatenarea altor două liste.
`append([],L,L) .`
`append([X|T],L, [X|R]) :- append(T,L,R) .`

Există predicatele predefinite `member/2` și `append/3`.

Liste append/3

□ Funcția append/3:

```
?- listing(append/3).
```

```
append([],L,L).
```

```
append([X|T],L, [X|R]) :- append(T,L,R).
```

```
?- append(X,Y,[a,b,c]).
```

```
X = [],
```

```
Y = [a, b, c] ;
```

```
X = [a],
```

```
Y = [b, c] ;
```

```
X = [a, b],
```

```
Y = [c] ;
```

```
X = [a, b, c],
```

```
Y = [] ;
```

false

- Funcția astfel definită poate fi folosită atât pentru verificare, cât și pentru generare.

Exercițiu

- Definiți un predicat `elim/3` care verifică dacă o listă se obține din alta prin eliminarea unui element.

```
elim(X, [X|T], T).  
elim(X, [H|T], [H|L]) :- elim(X,T,L).
```

- Definiți un predicat care `perm/2` care verifică dacă două liste sunt permutări.

```
perm([], []). perm([X|T],L) :- elim(X,L,R), perm(R,T).
```

Predicatele predefinite `select/3` și `permutation/2` au aceeași funcționalitate.

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: word(relay). word(early). word(layer).

Predicat util:

```
?- name(relay,L). % conversie între atomi și liste  
L = [114, 101, 108, 97, 121]
```

Două abordări posibile:

- ☐ se generează o posibilă, soluție apoi se testează dacă este în KB.
- ☐ se parcurge KB și pentru fiecare termen se testează dacă e soluție.

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

```
KB: word(relay).  word(early).  word(layer).
```

```
anagram1(A,B) :- name(A,L), permutation(L,W),  
                  name(B,W), word(B).
```

```
anagram2(A,B) :- name(A,L), word(B),  
                  name(B,W), permutation(L,W).
```

```
?- anagram1(layre,X).
```

```
X = layer ;
```

```
X = relay ;
```

```
X = early ;
```

```
false.
```

```
?- anagram2(layre,X).
```

```
X = relay ;
```

```
X = early ;
```

```
X = layer ;
```

```
false.
```

Exercițiu

- Definiți un predicat `rev/2` care verifică dacă o listă este inversa altei liste.

```
rev([], []).
```

```
rev([X|T], L) :- rev(T, R), append(R, [X], L).
```

Soluția de mai sus este corectă, dar foarte costisitoare computațional, datorită stilului de programare declarativ.

Cum putem defini o variantă mai rapidă?

O metodă care prin care recursia devine mai rapidă este folosirea **acumulatorilor**, în care se păstrează rezultatele parțiale.

Recursie cu acumulatori

- Varianta inițială:

```
rev([], []).
```

```
rev([X|T], L) :- rev(T, R), append(R, [X], L).
```

- Varianta cu acumulator

```
rev(L, R) :- revac(L, [], R).
```

```
% la momentul inițial nu am acumulat nimic.
```

```
revac([], R, R).
```

```
% cand lista inițială a fost consumată,
```

```
% am acumulat rezultatul final.
```

```
revac([X|T], Acc, R) :- revac(T, [X|Acc], R).
```

```
% Acc conține inversa listei care a fost deja parcursă.
```

- Complexitatea a fost redusă de la $O(n^2)$ la $O(n)$, unde n este lungimea listei.

Recursie

- Multe implementări ale limbajului Prolog aplică "last call optimization" atunci când un apel recursiv este ultimul predicat din corpul unei clauze (tail recursion).
- Atunci când este posibil, se recomandă utilizare recursiei la coadă (tail recursion).
- Vom defini un predicat care generează liste lungi în două moduri și vom analiza performanța folosind predicatul `time/1`.

```
biglist(0, []).
```

```
biglist(N, [N|T]) :- N >= 1, M is N-1, biglist(M, T), M=M.
```

```
biglist_tr(0, []).
```

```
biglist_tr(N, [N|T]) :- N >= 1, M is N-1, biglist_tr(M, T).
```

Recursie la coadă

- Predicat **fără** recursie la coadă:

```
biglist(0, []).
```

```
biglist(N, [N|T]) :- N >= 1, M is N-1, biglist(M, T), M=M.
```

Apelul recursiv întoarce valoarea găsită în predicatul apelant, această valoare urmând a fi prelucrată.

```
?- time(biglist(50000, X)).
```

```
100,000 inferences, 0.016 CPU in 0.038 seconds  
(41% CPU, 6400000 Lips)
```

```
X = [50000, 49999, 49998|...] .
```

- Predicatul **cu** recursie la coadă:

```
biglist_tr(0, []).
```

```
biglist_tr(N, [N|T]) :- N >= 1, M is N-1, biglist_tr(M, T).
```

```
?- time(biglist_tr(50000, X)).
```

```
100,000 inferences, 0.000 CPU in 0.007 seconds  
(0% CPU, Infinite Lips)
```

```
X = [50000, 49999, 49998|...]
```

Exemplu: reprezentarea unei GLC

Structura frazelor

- Aristotel, On Interpretation,
<http://classics.mit.edu/Aristotle/interpretation.1.1.html>:
"Every affirmation, then, and every denial, will consist of a noun and a verb, either definite or indefinite."
- N. Chomsky, Syntactic structure, Mouton Publishers, First printing 1957 - Fourteenth printing 1985 [Chapter 4 (Phrase Structure)]
 - (i) $Sentence \rightarrow NP + VP$
 - (ii) $NP \rightarrow T + N$
 - (iii) $VP \rightarrow Verb + NP$
 - (iv) $T \rightarrow the$
 - (v) $N \rightarrow fman, ball, etc.$
 - (vi) $V \rightarrow hit, took, etc.$

Gramatică independentă de context

- Definim structura propozițiilor folosind o gramatică independentă de context:

S	→	NP VP
NP	→	Det N
VP	→	V
VP	→	V NP
Det	→	<i>the</i>
Det	→	<i>a</i>
N	→	<i>boy</i>
N	→	<i>girl</i>
V	→	<i>loves</i>
V	→	<i>hates</i>

- Neterminalele definesc categorii gramaticale:

S (propozițiile),
NP (expresiile substantivale),
VP (expresiile verbale),
V (verbele),
N (substantivele),
Det (articolele).

- Terminalele definesc cuvintele.

Gramatică independentă de context

GIC

S → NP VP
NP → Det N
VP → V
VP → V NP

Det → *the*
Det → *a*
N → *boy*
N → *girl*
V → *loves*
V → *hates*

Ce vrem să facem?

- ☐ Vrem să scriem un program în Prolog care să recunoască propozițiile generate de această gramatică.
- ☐ Reprezentăm propozițiile prin liste.

?- atomic_list_concat(SL, ' ', 'a boy loves a girl').
SL = [a, boy, loves, a, girl]

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

SL = [a, boy, loves, a, girl]

- Fiecărui neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective. `n([boy])`.

`n([girl])`. `det([the])`. `v([loves])`.

- Lista asociată unei propoziții se obține prin concatenarea listelor asociate elementelor componente.

De exemplu, interpretăm regula $S \rightarrow NP VP$ astfel:

o propoziție este o listă L care se obține prin concatenarea a două liste, X și Y, unde X reprezintă o expresie substantivală și Y reprezintă o expresie verbală.

`s(L) :- np(X), vp(Y), append(X,Y,L).`

Definirea unei gramatici în Prolog

Gramatică independentă de context

S → NP VP
NP → Det N
VP → V
VP → V NP

Det → *the*
Det → *a*
N → *boy*
N → *girl*
V → *loves*
V → *hates*

Prolog

```
s(L) :- np(X), vp(Y),  
        append(X,Y,L).  
np(L) :- det(X), n(Y),  
        append(X,Y,L).  
vp(L) :- v(L).  
vp(L) :- v(X), np(Y),  
        append(X,Y,L).
```

```
det([the]).  
det([a]).  
n([boy]).  
n([girl]).  
v([loves]).  
v([hates]).
```

Definirea unei gramatici în Prolog

```
s(L) :- np(X), vp(Y),  
        append(X,Y,L).
```

```
np(L) :- det(X), n(Y),  
        append(X,Y,L) .
```

```
vp(L) :- v(L).
```

```
vp(L):- v(X), np(Y),  
        append(X,Y,L) .
```

```
det([the]).
```

```
det([a]).
```

```
n([boy]).
```

```
n([girl]).
```

```
v([loves]).
```

```
v([hates]).
```

```
?- s([a,boy,loves, a,  
girl]).  
true .
```

```
?- s[a, girl|T].  
T = [loves] ;  
T = [hates] ;  
T = [loves, the, boy] ;  
:
```

```
?- s(S).  
S = [the, boy, loves] ;  
S = [the, boy, hates] ;  
:
```

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

SL = [a, boy, loves, a, girl]

- Fiecărui neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective. `n([boy])`.

`n([girl])`. `det([the])`. `v([loves])`.

- Lista asociată unei propoziții se obține prin concatenarea listelor asociate elementelor componente.
- Deși corectă, reprezentarea anterioară este ineficientă, arborele de căutare este foarte mare. Pentru a optimiza, folosim reprezentarea listelor ca diferențe.

Bibliografie

- M. Ben-Ari, **Mathematical Logic for Computer Science**, Springer, 2012.
- P. Blackburn, J. Bos, K. Striegnitz, **Learn Prolog now**, College Publications, 2006.
- J.W. Lloyd, Foundations of Logic Programming, Springer, 1987.
- L.S. Sterling and E.Y. Shapiro, The Art of Prolog <https://mitpress.mit.edu/books/art-prolog-second-edition>
- Logic Programming, The University of Edinburgh, <https://www.inf.ed.ac.uk/teaching/courses/lp/>

Curs 9

2020-2021

Fundamentele limbajelor de programare

Cuprins



- 1 Limbajul IMP
- 2 O implementare a limbajului IMP în Prolog
- 3 O implementare a semanticii small-step
- 4 Semantica Small-Step pentru Lambda Calcul

Limbajul IMP

Limbajul IMP

Vom implementa un limbaj care conține:

- Expresii

- Aritmetice

`x + 3`

- Booleene

`x >= 7`

- Instrucțiuni

- De atribuire

`x = 5`

- Condiționale

`if(x >= 7, x = 5, x = 0)`

- De ciclare

`while(x >= 7, x = x - 1)`

- Compunerea instrucțiunilor

`x=7;while(x>=0,x=x-1)`

- Blocuri de instrucțiuni

`{x=7;while(x>=0,x=x-1)}`

Limbajul IMP

Exemplu

Un program în limbajul IMP

```
{x = 10 ; sum = 0;  
while(0 =< x,  
      {sum = sum + x; x = x-1}  
)},sum
```

□ Semantica

după execuția programului, se evaluează sum

Sintaxa BNF a limbajului IMP

$E ::= n \mid x$
 $\mid E + E \mid E - E \mid E * E$

$B ::= \text{true} \mid \text{false}$
 $\mid E < E \mid E > E \mid E == E$
 $\mid \text{not}(B) \mid \text{and}(B, B) \mid \text{or}(B, B)$

$C ::= \text{skip}$
 $\mid x = E$
 $\mid \text{if}(B, C, C)$
 $\mid \text{while}(B, C)$
 $\mid \{ C \} \mid C ; C$

$P ::= \{ C \}, E$

O implementare a limbajului IMP în Prolog

Decizii de implementare

- `{}` și `;` sunt operatori
 - `:- op(100, xf, {}).`
 - `:- op(1100, yf, ;).`
- definim un predicat pentru fiecare categorie sintactică
 - `stmt(while(BE,St)) :- bexp(BE), stmt(St).`
- `while`, `if`, `and`, etc sunt functori în Prolog
 - `while(true,skip)` este un termen compus
- `,` are semnificația obișnuită
- pentru valori numerice folosim întregii din Prolog
 - `aexp(I) :- integer(I).`
- pentru identificatori folosim atomii din Prolog
 - `aexp(X) :- atom(X).`

Expresiile aritmetice

$$E ::= n \mid x \\ \mid E + E \mid E - E \mid E * E$$

Prolog

```
aexp(I) :- integer(I).  
aexp(X) :- atom(X).  
aexp(A1 + A2) :- aexp(A1), aexp(A2).  
aexp(A1 - A2) :- aexp(A1), aexp(A2).  
aexp(A1 * A2) :- aexp(A1), aexp(A2).
```

Expresiile aritmetice

Exemplu

?- aexp(1000).

true.

?- aexp(id).

true.

?- aexp(id + 1000).

true.

?- aexp(2 + 1000).

true.

?- aexp(x * y).

true.

?- aexp(- x).

false.

Expressiile booleene

$B ::= \text{true} \mid \text{false}$
 $\mid E \leq E \mid E \geq E \mid E == E$
 $\mid \text{not}(B) \mid \text{and}(B, B) \mid \text{or}(B, B)$

Prolog

```
bexp(true). bexp(false).  
bexp(and(BE1, BE2)) :- bexp(BE1), bexp(BE2).  
bexp(or(BE1, BE2)) :- bexp(BE1), bexp(BE2).  
bexp(not(BE)) :- bexp(BE).
```

```
bexp(A1 <= A2) :- aexp(A1), aexp(A2).  
bexp(A1 >= A2) :- aexp(A1), aexp(A2).  
bexp(A1 == A2) :- aexp(A1), aexp(A2).
```

Expresiile booleene

Exemplu

?- bexp(true).

true.

?- bexp(id).

false.

?- bexp(not(1 =< 2)).

true.

?- bexp(or(1 =< 2,true)).

true.

?- bexp(or(a =< b,true)).

true.

?- bexp(not(a)).

false.

?- bexp(!(a)).

false.

Instrucțiunile

```
 $C ::= \text{skip}$   
|  $x = E$  ;  
| if(  $B$  )  $C$  else  $C$   
| while(  $B$  )  $C$   
| {  $C$  } |  $C$  ;  $C$ 
```

Prolog

```
stmt(skip).  
stmt(X = AE) :- atom(X), aexp(AE).  
stmt(St1;St2) :- stmt(St1), stmt(St2).  
stmt((St1;St2)) :- stmt(St1), stmt(St2).  
stmt({St}) :- stmt(St).  
stmt(if(BE,St1,St2)) :- bexp(BE), stmt(St1), stmt(St2).  
stmt(while(BE,St)) :- bexp(BE), stmt(St).
```

Instrucțiunile

Exemplu

?- stmt(id = 5).

true.

?- stmt(id = a).

true.

?- stmt(3 = 6).

false.

?- stmt(if(true, x=2;y=3, x=1;y=0)).

true.

?- stmt(while(x =< 0,skip)).

true.

?- stmt(while(x =< 0,)).

false.

?- stmt(while(x =< 0,skip)).

true .

Programele

$P ::= \{ C \}, E$

Prolog

```
program(St,AE) :- stmt(St), aexp(AE).
```

Exemplu

```
test0 :- program( {x = 10 ; sum = 0;
                  while(0 =< x,
                        {sum = sum + x; x = x-1}
                      )}
        , sum).
```

```
?- test0.
true.
```

O implementare a semanticii small-step

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
 - Semantică Operațională Structurală
 - semantică prin tranziții
 - semantică prin reducere
- Definește cel mai mic pas de execuție ca o relație „de tranziție” între configurații:

$$\langle cod, \sigma \rangle \rightarrow \langle cod, \sigma' \rangle$$

- Execuția se obține ca o succesiune de astfel de tranziții:

$$\begin{aligned} \langle \text{int } x = 0 ; x = x + 1 ; , \perp \rangle &\longrightarrow \langle x = x + 1 ; , x \mapsto 0 \rangle \\ &\longrightarrow \langle x = 0 + 1 ; , x \mapsto 0 \rangle \\ &\longrightarrow \langle x = 1 ; , x \mapsto 0 \rangle \\ &\longrightarrow \langle \{\} , x \mapsto 1 \rangle \end{aligned}$$

Semantica small-step

- Definește cel mai mic pas de execuție ca o relație de tranziție între configurații:
$$\langle cod, \sigma \rangle \rightarrow \langle cod', \sigma' \rangle \quad \text{step}(Cod, S1, Cod', S2)$$
- Execuția se obține ca o succesiune de astfel de tranziții.
- Starea execuției unui program IMP la un moment dat este o funcție parțială: $\sigma = n \mapsto 10, sum \mapsto 0$, etc.

Reprezentarea stărilor în Prolog

```
get(S,X,I) :- member(vi(X,I),S).  
get(_,_,0).  
set(S,X,I,[vi(X,I)|S1]) :- del(S,X,S1).  
  
del(S,X,S1) :- select(vi(X,_), S, S1), !.  
del(S, _, S).
```

Semantica expresiilor aritmetice

□ Semantica unei variabile

$\langle x, \sigma \rangle \rightarrow \langle i, \sigma \rangle$ *dacă* $i = \sigma(x)$

Prolog

```
step(X,S,I,S) :-  
    atom(X),  
    get(S,X,I).
```

Semantica expresiilor aritmetice

□ Semantica adunării a două expresii aritmetice

$\langle i_1 + i_2, \sigma \rangle \rightarrow \langle i, \sigma \rangle$ *dacă* $i = i_1 + i_2$

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma' \rangle}{\langle a_1 + a_2, \sigma \rangle \rightarrow \langle a'_1 + a_2, \sigma' \rangle}$$

$$\frac{\langle a_2, \sigma \rangle \rightarrow \langle a'_2, \sigma' \rangle}{\langle a_1 + a_2, \sigma \rangle \rightarrow \langle a_1 + a'_2, \sigma' \rangle}$$

Prolog

```
step(I1 + I2,S,I,S):- integer(I1),integer(I2),  
                        I is I1 + I2.
```

```
step(AE + AE1,S1,AE + AE2,S1):- step(AE1,S1,AE2,S2).
```

```
step(AE1 + AE,S1,AE2 + AE,S2):- step(AE1,S1,AE2,S2).
```

Semantica expresiilor aritmetice

Exemplu

?- step(a + b, [vi(a,1),vi(b,2)],AE, S).

AE = 1+b,

S = [vi(a, 1), vi(b, 2)] .

?- step(1 + b, [vi(a,1),vi(b,2)],AE, S).

AE = 1+2,

S = [vi(a, 1), vi(b, 2)] .

?- step(1 + 2, [vi(a,1),vi(b,2)],AE, S).

AE = 3,

S = [vi(a, 1), vi(b, 2)]

□ Semantica * și – se definesc similar.

Semantica expresiilor booleene

□ Semantica operatorului de comparație

$\langle i_1 = < i_2, \sigma \rangle \rightarrow \langle \mathbf{false}, \sigma \rangle$ dacă $i_1 > i_2$

$\langle i_1 = < i_2, \sigma \rangle \rightarrow \langle \mathbf{true}, \sigma \rangle$ dacă $i_1 \leq i_2$

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma' \rangle}{\langle a_1 = < a_2, \sigma \rangle \rightarrow \langle a'_1 = < a_2, \sigma' \rangle} \quad \frac{\langle a_2, \sigma \rangle \rightarrow \langle a'_2, \sigma' \rangle}{\langle a_1 = < a_2, \sigma \rangle \rightarrow \langle a_1 = < a'_2, \sigma' \rangle}$$

Prolog

```
step(I1 = < I2,S,true,S):- integer(I1),integer(I2),
                           (I1 = < I2).
step(I1 = < I2,S,false,S):- integer(I1),integer(I2),
                           (I1 > I2).
step(AE = < AE1,S1,AE = < AE2,S2):- step(AE1,S1,AE2,S2).
step(AE1 = < AE,S1,AE2 = < AE,S2):- step(AE1,S1,AE2,S2).
```

Semantica expresiilor Booleene

□ Semantica negației

$\langle \text{not}(\text{true}) , \sigma \rangle \rightarrow \langle \text{false} , \sigma \rangle$

$\langle \text{not}(\text{false}) , \sigma \rangle \rightarrow \langle \text{true} , \sigma \rangle$

$$\frac{\langle a , \sigma \rangle \rightarrow \langle a' , \sigma' \rangle}{\langle \text{not} (a) , \sigma \rangle \rightarrow \langle \text{not} (a') , \sigma' \rangle}$$

Prolog

```
step(not(true),S,false,S) .
```

```
step(not(false),S,true,S) .
```

```
step(not(BE1),S1,not(BE2),S2) :- step(BE1,S1,BE2,S2) .
```

Semantica compunerii și a blocurilor

□ Semantica blocurilor

$$\langle \{s\}, \sigma \rangle \rightarrow \langle s, \sigma \rangle$$

□ Semantica compunerii secvențiale

$$\langle \{\}; s_2, \sigma \rangle \rightarrow \langle s_2, \sigma \rangle \quad \frac{\langle s_1, \sigma \rangle \rightarrow \langle s'_1, \sigma' \rangle}{\langle s_1; s_2, \sigma \rangle \rightarrow \langle s'_1; s_2, \sigma' \rangle}$$

Prolog

```
step({E},S,E,S).
```

```
step(skip;St2),S,St2,S).
```

```
step((St1;St),S1,(St2;St),S2) :-  
    step(St1,S1,St2,S2) .
```


Semantica atribuirii

□ Semantica atribuirii

$\langle x = i, \sigma \rangle \rightarrow \langle \{\} , \sigma' \rangle$ dacă $\sigma' = \sigma[i/x]$

$$\frac{\langle a, \sigma \rangle \rightarrow \langle a', \sigma' \rangle}{\langle x = a, \sigma \rangle \rightarrow \langle x = a' ; , \sigma' \rangle}$$

Prolog

```
step(X = I,S,skip,S1) :- integer(I),set(S,X,I,S1).
```

```
step(X = AE1,S1,X = AE2,S2) :-  
                                step(AE1,S1,AE2,S2).
```

Semantica lui if

□ Semantica lui if

$\langle \mathbf{if}(\mathbf{true}, bl_1, bl_2), \sigma \rangle \rightarrow \langle bl_1, \sigma \rangle$

$\langle \mathbf{if}(\mathbf{false}, bl_1, bl_2), \sigma \rangle \rightarrow \langle bl_2, \sigma \rangle$

$$\frac{\langle b, \sigma \rangle \rightarrow \langle b', \sigma' \rangle}{\langle \mathbf{if}(b, bl_1, bl_2), \sigma \rangle \rightarrow \langle \mathbf{if}(b', bl_1, bl_2), \sigma' \rangle}$$

Prolog

```
step(if(true,St1,_),S,St1,S).  
step(if(false,_,St2),S,St2,S).
```

```
step(if(BE1,St1,St2),S1,if(BE2,St1,St2),S2) :-  
    step(BE1,S1,BE2,S2) .
```

Semantica lui while

□ Semantica lui while

$\langle \text{while } (b, bl) , \sigma \rangle \rightarrow \langle \text{if } (b, bl ; \text{while } (b, bl), \text{skip}) , \sigma \rangle$

Prolog

`step(while(BE, St), S, if(BE, (St; while(BE, St)), skip), S).`

Semantica programelor

□ Semantica programelor

$$\frac{\langle a_1, \sigma_1 \rangle \rightarrow \langle a_2, \sigma_2 \rangle}{\langle (\mathbf{skip}, a_1), \sigma_1 \rangle \rightarrow \langle (\mathbf{skip}, a_2), \sigma_2 \rangle}$$

$$\frac{\langle s_1, \sigma_1 \rangle \rightarrow \langle s_2, \sigma_2 \rangle}{\langle (s_1, a), \sigma_1 \rangle \rightarrow \langle (s_2, a), \sigma_2 \rangle}$$

Prolog

```
step((skip,AE1),S1,(skip,AE2),S2) :-  
    step(AE1,S1,AE2,S2) .  
step((St1,AE),S1,(St2,AE),S2) :-  
    step(St1,S1,St2,S2) .
```

Execuția programelor

Prolog

```
all_steps(P1, S1, PF, SF) :-  
    step(P1, S1, P2, S2)  
-> all_steps(P2, S2, PF, SF)  
;   PF = P1, SF = S1.
```

```
run_program(Name) :- defpg(Name, {P}, E),  
                      all_steps((P,E), [], (skip, I), _),  
                      write(I).
```

Exemplu

```
defpg(pg2, {x = 10 ; sum = 0; while(0 =< x, {  
                                         sum = sum + x;  
                                         x = x - 1})}, sum)
```

```
?- run_program(pg2).
```

```
55
```

```
true
```

Execuția programelor: trace

Putem defini o funcție care ne permite să urmărim execuția unui program în implementarea noastră?

Prolog

```
all_steps(P1, S1, PF, SF, [(P1, S1)| Trace]) :-  
    step(P1, S1, P2, S2)  
-> all_steps(P2, S2, PF, SF, Trace)  
;   PF = P1, SF = S1, Trace=[].
```

```
trace_program(Name) :- defpg(Name, {P}, E),  
                        all_steps((P,E), [], _, _, Trace),  
                        write(Trace).
```

Execuția programelor: trace_program

Exemplu

?- trace_program(pg2).

...

```
((if(0=<x,(sum=sum+x;x=x-1;while(0=<x,sum=sum+x;x=x-1)),skip), sum),  
[vi(x,-1),vi(sum,55)]),  
((if(0=<-1,(sum=sum+x;x=x-1;while(0=<x,sum=sum+x;x=x-1)),skip), sum),  
[vi(x,-1),vi(sum,55)]),  
((if(false,(sum=sum+x;x=x-1;while(0=<x,sum=sum+x;x=x-1)),skip), sum),  
[vi(x,-1),vi(sum,55)]),  
((skip, sum), [vi(x,-1),vi(sum,55)]),  
((skip, 55), [vi(x,-1),vi(sum,55)]),
```

Sintaxa limbajului LAMBDA

BNF

```
e ::= x
    | λx.e
    | e e
    | let x = e in e
```

Verificarea sintaxei în Prolog

```
exp(Id) :- atom(Id). % identifier
exp(Id -> Exp) :- atom(Id), exp(Exp). % lambda
exp(Exp1 $ Exp2) :- exp(Exp1), exp(Exp2). % application
exp(let(Id, Exp1, Exp2)) :- atom(Id), exp(Exp1), exp(Exp2).
```


Semantica small-step pentru Lambda

- Definește cel mai mic pas de execuție ca o relație de tranziție între expresii dată fiind o stare cu valori pentru variabilele libere

$\rho \vdash cod \rightarrow cod'$

`step(Env, Cod1, Cod2)`

- Execuția se obține ca o succesiune de astfel de tranziții.

Semantica variabilelor

$$\rho \vdash x \rightarrow v \quad \text{dacă } \rho(x) = v$$

Prolog

```
step (Env, X, V) :- atom(X), get (Env, X, V).
```

Semantica λ -abstracției

$$\rho \vdash \lambda x.e \rightarrow \text{closure}(x, e, \rho)$$

λ -abstracția se evaluează la o valoare specială numită closure care capturează valorile curente ale variabilelor pentru a se putea executa în acest mediu atunci când va fi aplicată.

Prolog

```
step (Env, X -> E, closure (X, E, Env)).
```

Semantica construcției **let**

$$\rho \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \rightarrow (\lambda x. e_2) e_1$$

A îi da lui x valoarea lui e_1 în e_2 este același lucru cu a aplica funcția de x cu corpul e_2 expresiei e_1 .

Prolog

```
step (_, let (X, E1, E2), (X -> E2) $ E1).
```

Semantica operatorului de aplicare

$$\frac{\rho_e[v/x] \vdash e \rightarrow e'}{\rho \vdash \text{closure}(x, e, \rho_e) v \rightarrow \text{closure}(x, e', \rho_e) v} \quad \text{dacă } v \text{ valoare}$$

$$\rho \vdash \text{closure}(x, v, \rho_e) e \rightarrow v \quad \text{dacă } v \text{ valoare}$$

$$\frac{\rho \vdash e_1 \rightarrow e'_1}{\rho \vdash e_1 e_2 \rightarrow e'_1 e_2} \quad \frac{\rho \vdash e_2 \rightarrow e'_2}{\rho \vdash e_1 e_2 \rightarrow e_1 e'_2}$$

Prolog

```
step(Env, E $ E1, E $ E2) :- step(Env, E1, E2).
step(Env, E1 $ E, E2 $ E) :- step(Env, E1, E2).
step(Env, closure(X, E, EnvE) $ V, Result) :-
    \+ step(Env, V, _),
    set(EnvE, X, V, EnvEX),
    step(EnvEX, E, E1)
-> Result = closure(X, E1, EnvE) $ V
; Result = E.
```

Curs 10

Cuprins

1 Semantica Small-Step pentru Lambda Calcul

2 Determinarea tipurilor

- Asociere de tipuri
- Proprietăți
- Exemplu
- Implementare în Prolog

3 Funcții polimorfice

Sintaxa limbajului LAMBDA

BNF

```
e ::= x | n | true | false
    | e + e | e < e | not (e)
    | if e then e else e
    | λx.e | e e
    | let x = e in e
```

Verificarea sintaxei în Prolog

```
exp(Id) :- atom(Id).                % identifier
exp(Lit) :- Lit = true ; Lit = false ; integer(Lit).
exp(E1 + E2) :- exp(E1), exp(E2).    % same for any op
exp(if(E1, E2, E3)) :- exp(E1), exp(E2), exp(E3).
exp(Id -> Exp) :- atom(Id), exp(Exp). % lambda
exp(Exp1 $ Exp2) :- exp(Exp1), exp(Exp2). % application
exp(let(Id, Exp1, Exp2)) :- atom(Id), exp(Exp1), exp(Exp2).
```


Semantica small-step pentru Lambda

- Definește cel mai mic pas de execuție ca o relație de tranziție între expresii dată fiind o stare cu valori pentru variabilele libere
 $\rho \vdash cod \rightarrow cod'$ step(Env, Cod1, Cod2)
- Execuția se obține ca o succesiune de astfel de tranziții.

Semantica variabilelor

$$\rho \vdash x \rightarrow v \quad \text{dacă } \rho(x) = v$$

Prolog

```
step (Env, X, V) :- atom(X), get (Env, X, V).
```

Semantica expresiilor aritmetice

□ Semantica adunării a două expresii aritmetice

$\langle i_1 + i_2, \sigma \rangle \rightarrow \langle i, \sigma \rangle$ *dacă* $i = i_1 + i_2$

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma' \rangle}{\langle a_1 + a_2, \sigma \rangle \rightarrow \langle a'_1 + a_2, \sigma' \rangle} \quad \frac{\langle a_2, \sigma \rangle \rightarrow \langle a'_2, \sigma' \rangle}{\langle a_1 + a_2, \sigma \rangle \rightarrow \langle a_1 + a'_2, \sigma' \rangle}$$

□ Pentru alți operatori (aritmetici, de comparație, booleani, condițional)

□ Similar cu regulile din IMP

Prolog

```
step(_, I1 + I2, I):- integer(I1),integer(I2),  
                      I is I1 + I2.
```

```
step(Env, AE + AE1, AE + AE2):- step(Env, AE1, AE2).
```

```
step(Env, AE1 + AE, AE2 + AE):- step(Env, AE1, AE2).
```

Semantica λ -abstracției

$$\rho \vdash \lambda x.e \rightarrow \text{closure}(x, e, \rho)$$

λ -abstracția se evaluează la o valoare specială numită closure care capturează valorile curente ale variabilelor pentru a se putea executa în acest mediu atunci când va fi aplicată.

Prolog

```
step (Env, X -> E, closure (X, E, Env)).
```

Semantica construcției **let**

$$\rho \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \rightarrow (\lambda x. e_2) e_1$$

A îi da lui x valoarea lui e_1 în e_2 este același lucru cu a aplica funcția de x cu corpul e_2 expresiei e_1 .

Prolog

```
step (_, let (X, E1, E2), (X -> E2) $ E1).
```

Semantica operatorului de aplicare

$$\frac{\rho_e[v/x] \vdash e \rightarrow e'}{\rho \vdash \text{closure}(x, e, \rho_e) v \rightarrow \text{closure}(x, e', \rho_e) v} \quad \text{dacă } v \text{ valoare}$$

$$\rho \vdash \text{closure}(x, v, \rho_e) e \rightarrow v \quad \text{dacă } v \text{ valoare}$$

$$\frac{\rho \vdash e_1 \rightarrow e'_1}{\rho \vdash e_1 e_2 \rightarrow e'_1 e_2} \quad \frac{\rho \vdash e_2 \rightarrow e'_2}{\rho \vdash e_1 e_2 \rightarrow e_1 e'_2}$$

Prolog

```
step(Env, E $ E1, E $ E2) :- step(Env, E1, E2).
step(Env, E1 $ E, E2 $ E) :- step(Env, E1, E2).
step(Env, closure(X, E, EnvE) $ V, Result) :-
    \+ step(Env, V, _),
    set(EnvE, X, V, EnvEX),
    step(EnvEX, E, E1)
-> Result = closure(X, E1, EnvE) $ V
; Result = E.
```

Problemă: Sintaxa este prea permisivă

Problemă: Mulți termeni acceptați de sintaxă nu pot fi evaluați

- $2 (\lambda x.x)$ — expresia din stânga aplicației trebuie să reprezinte o funcție
- $(\lambda x.x) + 1$ — adunăm funcții cu numere
- $(\lambda x.x + 1) (\lambda x.x)$ — pot face o reducere, dar tot nu pot evalua

Soluție: Identificarea (precisă) a programelor corecte

- Definim tipuri pentru fragmente de program corecte (e.g., int, bool)
- Definim (recursiv) o relație care să lege fragmente de program de tipurile asociate

$((\lambda x.x + 1) ((\lambda x.x) 3)) : \text{int}$

Relația de asociere de tipuri

Definim (recursiv) o relație de forma $\Gamma \vdash e : \tau$, unde

- τ este un tip

$\tau ::= \text{int}$ [întregi]
| bool [valori de adevăr]
| $\tau \rightarrow \tau$ [funcții]
| a [variabile de tip]

- e este un termen (potențial cu variabile libere)
- Γ este **mediul de tipuri**, o funcție parțială finită care asociază tipuri variabilelor (libere ale lui e)
- Variabilele de tip sunt folosite pentru a indica polimorfismul

Cum citim $\Gamma \vdash e : \tau$?

Dacă variabila x are tipul $\Gamma(x)$ pentru orice $x \in \text{dom}(\Gamma)$, atunci termenul e are tipul τ .

Axiome

(:VAR) $\Gamma \vdash x : \tau$ *dacă* $\Gamma(x) = \tau$

(:INT) $\Gamma \vdash n : int$ *dacă* n întreg

(:BOOL) $\Gamma \vdash b : bool$ *dacă* $b = true$ or $b = false$

Expresii

$$(:\text{OP}) \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \ o \ e_2 : \text{int}} \quad \text{dacă } o \in \{+, -, *, /\}$$

$$(:\text{COP}) \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \ o \ e_2 : \text{bool}} \quad \text{dacă } o \in \{\leq, \geq, <, >, =\}$$

$$(:\text{BOP}) \quad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \ o \ e_2 : \text{bool}} \quad \text{dacă } o \in \{\mathbf{and}, \mathbf{or}\}$$

$$(:\text{IF}) \quad \frac{\Gamma \vdash e_b : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{if } e_b \mathbf{ then } e_1 \mathbf{ else } e_2 : \tau}$$

Fragmentul funcțional

$$(\text{:FN}) \quad \frac{\Gamma' \vdash e : \tau'}{\Gamma \vdash \lambda x. e : \tau \rightarrow \tau'} \quad \text{dacă } \Gamma' = \Gamma[x \mapsto \tau]$$

$$(\text{:APP}) \quad \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau}$$

Programe în execuție

Problemă:

- În timpul execuției programul conține valori de tip closure
- Care este tipul lor?

Soluție

- Adăugăm regula ($:_{CL}$)
$$\frac{\Gamma_\rho \vdash \lambda x.e : \tau}{\Gamma \vdash closure(x, e, \rho) : \tau} \quad unde$$
- Mediul de tipuri Γ_ρ asociat unui mediu de execuție ρ satisface:
 - $Dom(\Gamma_\rho) = Dom(\rho)$
 - Pentru orice variabilă $x \in Dom(\rho)$, există τ tip și v valoare astfel încât $\Gamma_\rho(x) = \tau, \rho(x) = v$ și $\vdash v : \tau$

Proprietăți

Theorem (Proprietatea de a progresa)

Dacă $\Gamma_\rho \vdash e : \tau$ atunci e este valoare sau e poate progresa în ρ : există e' astfel încât $\rho \vdash e \rightarrow e'$.

Theorem (Proprietatea de conservare a tipului)

Dacă $\Gamma_\rho \vdash e : \tau$ și $\rho \vdash e \rightarrow e'$, atunci $\Gamma'_\rho \vdash e' : \tau$.

Theorem (Siguranță—programele bine formate nu se împotmolesc)

Dacă $\Gamma_\rho \vdash e : \tau$ și $\rho \vdash e \longrightarrow^ e'$, atunci e' este valoare sau există e'' , astfel încât $\rho \vdash e' \rightarrow e''$.*

Probleme computaționale

Verificarea tipului

Date fiind Γ , e și τ , verificați dacă $\Gamma \vdash e : \tau$.

Determinarea (inferarea) tipului

Date fiind Γ și e , găsiți (sau arătați ce nu există) un τ astfel încât $\Gamma \vdash e : \tau$.

- A doua problemă e mai grea în general decât prima
- Algoritmi de inferare a tipurilor
 - Colectează constrângeri asupra tipului
 - Folosesc metode de rezolvare a constrângerilor (programare logică)

Probleme computaționale

Theorem (Determinarea tipului este decidabilă)

Date fiind Γ și e , poate fi găsit (sau demonstrat că nu există) un τ astfel încât $\Gamma \vdash e : \tau$.

Theorem (Verificarea tipului este decidabilă)

Date fiind Γ , e și τ , problema $\Gamma \vdash e : \tau$ este decidabilă.

Theorem (Unicitatea tipului)

Dacă $\Gamma \vdash e : \tau$ și $\Gamma \vdash e : \tau'$, atunci $\tau = \tau'$.

Exemplu

Care este tipul expresiei următoare (dacă are)

$\lambda x.\lambda y.\lambda z. \text{ if } y = 0 \text{ then } z \text{ else } x/y$

Aplicăm regula

(:FN) $\frac{\Gamma' \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'} \quad \text{dacă } \Gamma' = \Gamma[x \mapsto \tau]$

$\vdash \lambda x.\lambda y.\lambda z. \text{ if } y = 0 \text{ then } z \text{ else } x/y : t_x \rightarrow t$ dacă

$x \mapsto t_x \vdash \lambda y.\lambda z. \text{ if } y = 0 \text{ then } z \text{ else } x/y : t$

Mai departe: $x \mapsto t_x \vdash \lambda y.\lambda z. \text{ if } y = 0 \text{ then } z \text{ else } x/y : t_y \rightarrow t_0$ dacă

$x \mapsto t_x, y \mapsto t_y \vdash \lambda z. \text{ if } y = 0 \text{ then } z \text{ else } x/y : t_0$ și, de mai sus,

$t = t_y \rightarrow t_0$

Mai departe: $x \mapsto t_x, y \mapsto t_y \vdash \lambda z. \text{ if } y = 0 \text{ then } z \text{ else } x/y : t_z \rightarrow t_1$

dacă $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash \text{ if } y = 0 \text{ then } z \text{ else } x/y : t_1$ și, de mai sus, $t_0 = t_z \rightarrow t_1$

Exemplu

Aplicăm regula

$$(\text{:COP}) \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \text{ o } e_2 : \text{bool}} \quad \text{dacă } o \in \{\leq, \geq, <, >, =\}$$

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y = 0 : \text{bool}$ dacă

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y : \text{int}$ și $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash 0 : \text{int}$

Aplicăm regula $(\text{:INT}) \quad \Gamma \vdash n : \text{int}$ dacă n întreg

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash 0 : \text{int}$ este adevărat

Aplicăm regula

$$(\text{:IOP}) \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \text{ o } e_2 : \text{int}} \quad \text{dacă } o \in \{+, -, *, /\}$$

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash x/y : \text{int}$ dacă

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash x : \text{int}$ și $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y : \text{int}$
și, de mai sus, $t_1 = \text{int}$

Exemplu

Recapitulăm

$\vdash \lambda x. \lambda y. \lambda z. \text{if } y = 0 \text{ then } z \text{ else } x/y : t_x \rightarrow t$ dacă

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y : \text{int}$ și $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash z : t_1$
 $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash x : \text{int}$ și $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y : \text{int}$
și $t_0 = t_z \rightarrow t_1, t = t_y \rightarrow t_0, t_1 = \text{int}$.

Aplicăm regula (VAR) $\Gamma \vdash x : \tau$ dacă $\Gamma(x) = \tau$

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y : t_y$ adevărat și, de mai sus $t_y = \text{int}$
 $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash z : t_z$ adevărat și, de mai sus, $t_1 = t_z$
 $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash x : t_x$ adevărat și, de mai sus, $t_x = \text{int}$

Exemplu

Finalizăm

$\vdash \lambda x.\lambda y.\lambda z. \text{if } y = 0 \text{ then } z \text{ else } x/y : t_x \rightarrow t$ dacă
 $t_0 = t_z \rightarrow t_1, t = t_y \rightarrow t_0, t_1 = \text{int}, t_y = \text{int}, t_1 = t_z$ și $t_x = \text{int}$.

Rezolvăm constrângerile și obținem

$\vdash \lambda x.\lambda y.\lambda z. \text{if } y = 0 \text{ then } z \text{ else } x/y : \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int}$

Relația de asociere de tipuri în Prolog

Definim (recursiv) o relație de forma `type(Gamma, E, T)`, unde

- `Gamma` este o listă de perechi de forma (X, T) unde X este un identificator și T este o expresie de tip cu variabile
- E este o λ -expresie scrisă cu sintaxa descrisă mai sus
- T este o expresie de tip cu variabile

Sintaxa limbajului LAMBDA

BNF

```
e ::= x | n | true | false
    | e + e | e < e | not (e)
    | if e then e else e
    | λx.e | e e
    | let x = e in e
```

Verificarea sintaxei în Prolog

```
exp(Id) :- atom(Id).                % identifier
exp(Lit) :- Lit = true ; Lit = false ; integer(Lit).
exp(E1 + E2) :- exp(E1), exp(E2).    % same for any op
exp(if(E1, E2, E3)) :- exp(E1), exp(E2), exp(E3).
exp(Id -> Exp) :- atom(Id), exp(Exp). % lambda
exp(Exp1 $ Exp2) :- exp(Exp1), exp(Exp2). % application
exp(let(Id, Exp1, Exp2)) :- atom(Id), exp(Exp1), exp(Exp2).
```

Sintaxa tipurilor

BNF

```
 $\tau ::= \text{int}$  [întregi]  
|  $\text{bool}$  [valori de adevăr]  
|  $\tau \rightarrow \tau$  [funcții]  
|  $a$  [variabile de tip]
```

Verificarea sintaxei tipurilor în Prolog

```
is_type(X) :- variable(X).           % variabile  
is_type(int).                        % intregi  
is_type(bool).                       % valori de adevar  
is_type(T1 -> T2) :-                 % functii  
    is_type(T1), is_type(T2).
```

Axiome

(:VAR) $\Gamma \vdash X : \tau$ *dacă* $\Gamma(X) = \tau$

`type(Gamma, X, T) :- atom(X), get(Gamma, X, T).`

(:INT) $\Gamma \vdash n : int$ *dacă* n întreg

`type(_, I, int) :- integer(I).`

(:BOOL) $\Gamma \vdash b : bool$ *dacă* $b = true$ *or* $b = false$

`type(_, true, bool).`

`type(_, false, bool).`

Expresii

$$(:\text{IOP}) \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \text{ } o \text{ } e_2 : \text{int}} \quad \text{dacă } o \in \{+, -, *, /\}$$

type(Gamma, E1 + E2, int) :-
type(Gamma, E1, int), type(Gamma, E2, int).

$$(:\text{COP}) \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \text{ } o \text{ } e_2 : \text{bool}} \quad \text{dacă } o \in \{\leq, \geq, <, >, =\}$$

type(Gamma, E1 < E2, bool) :-
type(Gamma, E1, int), type(Gamma, E2, int).

$$(:\text{BOP}) \quad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \text{ } o \text{ } e_2 : \text{bool}} \quad \text{dacă } o \in \{\mathbf{and}, \mathbf{or}\}$$

type(Gamma, and(E1, E2), bool) :-
type(Gamma, E1, bool), type(Gamma, E2, bool).

Expresia condițională

$$(\text{:IF}) \quad \frac{\Gamma \vdash e_b : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e_b \text{ then } e_1 \text{ else } e_2 : \tau}$$

```
type (Gamma, if (E, E1, E2), T) :-  
  type (Gamma, E, bool),  
  type (Gamma, E1, T),  
  type (Gamma, E2, T).
```

Fragmentul funcțional

$$(\text{:FN}) \quad \frac{\Gamma' \vdash e : \tau'}{\Gamma \vdash \lambda x. e : \tau \rightarrow \tau'} \quad \text{dacă } \Gamma' = \Gamma[x \mapsto \tau]$$

type (Gamma, X \rightarrow E, TX \rightarrow TE) :-
 atom(X), set (Gamma, X, TX, GammaX), type (GammaX, E, TE).

$$(\text{:APP}) \quad \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau}$$

type (Gamma, E1 \$ E2, T) :-
 type (Gamma, E, TE2 \rightarrow T), type (Gamma, E2, TE2).

Tipurile variabile nu sunt suficiente

Tipurile variabile sunt destul de flexibile

- $\vdash \lambda x.x : t \rightarrow t$ pentru orice t
- $\vdash \mathbf{if} (\lambda x.x) \text{ true then } (\lambda x.x) \text{ 3 else 4 :int}$

Dar tipul unei expresii este fixat:

$\not\vdash (\lambda id.\mathbf{if} \text{ id true then id 3 else 4})(\lambda x.x) : \mathbf{int}$

Soluție

Pentru funcțiile cu nume, am vrea să fie ca și cum am calcula mereu tipul

$\vdash \mathbf{let} \text{ id} = (\lambda x.x) \text{ in } \mathbf{if} \text{ id true then id 3 else 4) :int}$

Operațional: redenumim variabilele de tip când instanțiem numele funcției

Scheme de tipuri

- Numim schemă de tipuri o expresie de forma $\langle \tau \rangle$, unde τ este o expresie tip cu variabile
- variabilele dintr-o schemă nu pot fi constrânse și cum ar fi cuantificate universal
- O schemă poate fi concretizată la un tip obișnuit substituindu-i fiecare variabilă cu orice tip (poate fi și variabilă)
 - Pentru orice substituție θ de la variabile de tip la tipuri cu variabile

Reguli pentru scheme

$$(\text{:LET}) \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma_1 \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \quad \text{dacă } \Gamma_1 = \Gamma[\langle \tau_1 \rangle / x]$$

```
type(Gamma, let(X, E1, E2), T) :-  
    type(Gamma, E1, T1),  
    copy_term(T1, FreshT1),      % redenumeste variabilele  
                                % ca sa nu poata fi constranse  
    set(Gamma, X, scheme(FreshT1), GammaX),  
    type(GammaX, E2, T).
```

$$(\text{:SCH}) \quad \Gamma \vdash x : \tau' \quad \text{dacă } \Gamma(x) = \langle \tau \rangle \text{ și } \tau' = \theta(\tau)$$

```
type(Gamma, X, T) :-  
    atom(X), get(Gamma, X, T), is_type(T), !.  
type(Gamma, X, T) :-  
    atom(X), get(Gamma, X, scheme(TX)),  
    copy_term(T1, T).           % redenumeste variabilele  
                                % ca sa poata fi constranse
```

Curs 11

Cuprins

- 1 Logica propozițională (recap.)
- 2 Logica de ordinul I (recap.)
- 3 Algoritmul de unificare
- 4 Formă clauzală. Rezoluție
 - Rezoluția în logica propozițională
- 5 Logica Horn
- 6 Rezoluția SLD

Logica propozițională (recap.)

Limbaajul și formulele PL

□ Limbaajul PL

- variabile propoziționale: $Var = \{p, q, v, \dots\}$
- conectori logici: \neg (unar), \rightarrow , \wedge , \vee , \leftrightarrow (binari)

□ Formulele PL

$$\begin{aligned} var &::= p \mid q \mid v \mid \dots \\ form &::= var \mid (\neg form) \mid form \wedge form \mid form \vee form \\ &\quad \mid form \rightarrow form \mid form \leftrightarrow form \end{aligned}$$

- Conectorii sunt împărțiți în conectori **de bază** și conectori **derivați** (în funcție de formalism).
- Legături între conectori:

$$\begin{aligned} \varphi \vee \psi &::= \neg \varphi \rightarrow \psi \\ \varphi \wedge \psi &::= \neg(\varphi \rightarrow \neg \psi) \\ \varphi \leftrightarrow \psi &::= (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi) \end{aligned}$$

Sintaxa și semantica

Un sistem logic are două componente:

□ Sintaxa

- noțiuni sintactice: **demonstrație**, **teoremă**
- notăm prin $\vdash \varphi$ faptul că φ este teoremă
- notăm prin $\Gamma \vdash \varphi$ faptul că formula φ este demonstrabilă din mulțimea de formule Γ

□ Semantica

- noțiuni semantice: **adevăr**, **model**, **tautologie** (formulă universal adevărată)
- notăm prin $\models \varphi$ faptul că φ este tautologie
- notăm prin $\Gamma \models \varphi$ faptul că formula φ este adevărată atunci când toate formulele din mulțimea Γ sunt adevărate

Completitudine: Γ -teoremele și Γ -tautologiile coincid

$\Gamma \vdash \varphi$ dacă și numai dacă $\Gamma \models \varphi$

Logica propozițională

Exemplu

Formalizați următorul raționament:

If winter is coming and Ned is not alive then Robb is lord of Winterfell. Winter is coming. Rob is not lord of Winterfell. Then Ned is alive.

O posibilă formalizare este următoarea:

p = winter is coming

q = Ned is alive

r = Robb is lord of Winterfel

$\{(p \wedge \neg q) \rightarrow r, p, \neg r\} \models q$

Logica de ordinul I (recap.)

Logica de ordinul I - sintaxa

Limbaj de ordinul I \mathcal{L}

- unic determinat de $\tau = (\mathbf{R}, \mathbf{F}, \mathbf{C}, \text{ari})$

Termenii lui \mathcal{L} , notați $\text{Trm}_{\mathcal{L}}$, sunt definiți inductiv astfel:

- orice variabilă este un termen;
- orice simbol de constantă este un termen;
- dacă $f \in \mathbf{F}$, $\text{ar}(f) = n$ și t_1, \dots, t_n sunt termeni, atunci $f(t_1, \dots, t_n)$ este termen.

Formulele atomice ale lui \mathcal{L} sunt definite astfel:

- dacă $R \in \mathbf{R}$, $\text{ar}(R) = n$ și t_1, \dots, t_n sunt termeni, atunci $R(t_1, \dots, t_n)$ este formulă atomică.

Formulele lui \mathcal{L} sunt definite astfel:

- orice formulă atomică este o formulă
- dacă φ este o formulă, atunci $\neg\varphi$ este o formulă
- dacă φ și ψ sunt formule, atunci $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \rightarrow \psi$ sunt formule
- dacă φ este o formulă și x este o variabilă, atunci $\forall x \varphi$, $\exists x \varphi$ sunt formule

Logica de ordinul I - semantică

O **structură** este de forma $\mathcal{A} = (A, \mathbf{F}^{\mathcal{A}}, \mathbf{R}^{\mathcal{A}}, \mathbf{C}^{\mathcal{A}})$, unde

- A este o mulțime nevidă
- $\mathbf{F}^{\mathcal{A}} = \{f^{\mathcal{A}} \mid f \in \mathbf{F}\}$ este o mulțime de operații pe A ; dacă f are aritatea n , atunci $f^{\mathcal{A}} : A^n \rightarrow A$.
- $\mathbf{R}^{\mathcal{A}} = \{R^{\mathcal{A}} \mid R \in \mathbf{R}\}$ este o mulțime de relații pe A ; dacă R are aritatea n , atunci $R^{\mathcal{A}} \subseteq A^n$.
- $\mathbf{C}^{\mathcal{A}} = \{c^{\mathcal{A}} \in A \mid c \in \mathbf{C}\}$.

O **interpretare a variabilelor** lui \mathcal{L} în \mathcal{A} (**\mathcal{A} -interpretare**) este o funcție $I : V \rightarrow A$.

Inductiv, definim **interpretarea termenului** t în \mathcal{A} sub I notat $t_I^{\mathcal{A}}$.

Inductiv, definim când o **formulă este adevărată în \mathcal{A} în interpretarea I** notat $\mathcal{A}, I \models \varphi$.

În acest caz spunem că (\mathcal{A}, I) este **model** pentru φ .

O formulă φ este **adevărată într-o structură \mathcal{A}** , notat $\mathcal{A} \models \varphi$, dacă este adevărată în \mathcal{A} sub orice interpretare. Spunem că \mathcal{A} este **model** al lui φ .

O formulă φ este **adevărată în logica de ordinul I**, notat $\models \varphi$, dacă este adevărată în orice structură. O formulă φ este **validă** dacă $\models \varphi$.

O formulă φ este **satisfiabilă** dacă există o structură \mathcal{A} și o \mathcal{A} -interpretare I astfel încât $\mathcal{A}, I \models \varphi$.

Algoritmul de unificare

Unificare

- O **substituție** σ este o funcție (parțială) de la variabile la termeni,

$$\sigma : V \rightarrow Trm_{\mathcal{L}}$$

- Doi termeni t_1 și t_2 **se unifică** dacă există o substituție θ astfel încât

$$\theta(t_1) = \theta(t_2).$$

- În acest caz, θ se numește **unificatorul** termenilor t_1 și t_2 .

- Un unificator ν pentru t_1 și t_2 este un **cel mai general unificator** (**cgu, mgu**) dacă pentru orice alt unificator ν' pentru t_1 și t_2 , există o substituție μ astfel încât

$$\nu' = \nu; \mu.$$

Algoritmul de unificare

- Pentru o mulțime finită de termeni $\{t_1, \dots, t_n\}$, $n \geq 2$, algoritmul de unificare stabilește dacă există un cgu.
- Algoritmul lucrează cu două liste:
 - Lista soluție: S
 - Lista de rezolvat: R
- Inițial:
 - Lista soluție: $S = \emptyset$
 - Lista de rezolvat: $R = \{t_1 \dot{=} t_2, \dots, t_{n-1} \dot{=} t_n\}$
- $\dot{=}$ este un simbol nou care ne ajută să formăm perechi de termeni (ecuații).

Algoritmul de unificare

Algoritmul constă în aplicarea regulilor de mai jos:

□ SCOATE

□ orice ecuație de forma $t \doteq t$ din R este eliminată.

□ DESCOMPUNE

□ orice ecuație de forma $f(t_1, \dots, t_n) \doteq f(t'_1, \dots, t'_n)$ din R este înlocuită cu ecuațiile $t_1 \doteq t'_1, \dots, t_n \doteq t'_n$.

□ REZOLVĂ

□ orice ecuație de forma $x \doteq t$ sau $t \doteq x$ din R , unde variabila x nu apare în termenul t , este mutată sub forma $x \doteq t$ în S .
În toate celelalte ecuații (din R și S), x este înlocuit cu t .

Algoritmul de unificare

Algoritmul se termină normal dacă $R = \emptyset$. În acest caz, S dă cgu.

Algoritmul este oprit cu concluzia inexistenței unui cgu dacă:

1 În R există o ecuație de forma

$$f(t_1, \dots, t_n) \doteq g(t'_1, \dots, t'_k) \text{ cu } f \neq g.$$

2 În R există o ecuație de forma $x \doteq t$ sau $t \doteq x$ și variabila x apare în termenul t .

Algoritmul de unificare - schemă

	Lista soluție S	Lista de rezolvat R
Inițial	\emptyset	$t_1 \doteq t'_1, \dots, t_n \doteq t'_n$
SCOATE	S	$R', t \doteq t$
	S	R'
DESCOMPUNE	S	$R', f(t_1, \dots, t_n) \doteq f(t'_1, \dots, t'_n)$
	S	$R', t_1 \doteq t'_1, \dots, t_n \doteq t'_n$
REZOLVĂ	S	$R', x \doteq t$ sau $t \doteq x$, x nu apare în t
	$x \doteq t, S[x/t]$	$R'[x/t]$
Final	S	\emptyset

$S[x/t]$: în toate ecuațiile din S, x este înlocuit cu t

Exemplu

Exemplu

□ Ecuațiile $\{g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)\}$ au gcu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(g(y)), y) \doteq f(g(z), w, z)$	DESCOMPUNE
$x \doteq g(y)$	$g(y) \doteq g(z), h(g(y)) \doteq w, y \doteq z$	REZOLVĂ
$w \doteq h(g(y)),$ $x \doteq g(y)$	$g(y) \doteq g(z), y \doteq z$	REZOLVĂ
$y \doteq z, x \doteq g(z),$ $w \doteq h(g(z))$	$g(z) \doteq g(z)$	SCOATE
$y \doteq z, x \doteq g(z),$ $w \doteq h(g(z))$	\emptyset	

□ $\nu = \{y/z, x/g(z), w/h(g(z))\}$ este cgu.

Exemplu

Exemplu

- Ecuațiile $\{g(y) \doteq x, f(x, h(y), y) \doteq f(g(z), b, z)\}$ au gcu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(y), y) \doteq f(g(z), b, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(y), y) \doteq f(g(z), b, z)$	DESCOMPUNE
$x \doteq g(y)$	$g(y) \doteq g(z), h(y) \doteq b, y \doteq z$	- EȘEC -

- h și b sunt simboluri de operații diferite!
- Nu există unificator pentru ecuațiile din U .

Exemplu

Exemplu

- Ecuațiile $\{g(y) \doteq x, f(x, h(x), y) \doteq f(y, w, z)\}$ au gcu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(x), y) \doteq f(y, w, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(g(y)), y) \doteq f(y, w, z)$	DESCOMPUNE
$x \doteq g(y)$	$g(y) \doteq y, h(g(y)) \doteq w, y \doteq z$	- EȘEC -

- În ecuația $g(y) \doteq y$, variabila y apare în termenul $g(y)$.
- Nu există unificator pentru ecuațiile din U .

Validitate și satisfiabilitate

Propoziție

Dacă φ este o formulă atunci

φ este validă dacă și numai dacă $\neg\varphi$ nu este satisfiabilă.

Formă clauzală. Rezoluție

Literali. FNC

- În logica propozițională un literal este o variabilă sau negația unei variabile.

$$\text{literal} := p \mid \neg p \quad \text{unde } p \text{ este variabilă propozițională}$$

- În logica de ordinul I un literal este o formulă atomică sau negația unei formule atomice.

$$\text{literal} := P(t_1, \dots, t_n) \mid \neg P(t_1, \dots, t_n)$$

unde $P \in \mathbf{R}$, $\text{ari}(P) = n$, și t_1, \dots, t_n sunt termeni.

- Pentru un literal L vom nota cu L^c literalul complement.

O formulă este în formă normală conjunctivă (FNC) dacă este o conjuncție de disjuncții de literali.

Forma clauzală în logica propozițională

- Pentru orice formulă α există o FNC α^{fc} astfel încât $\alpha \models \alpha^{fc}$.
- Pentru o formulă din logica propozițională determinăm FNC corespunzătoare prin următoarele transformări:

1 înlocuirea implicațiilor și echivalențelor

$$\varphi \rightarrow \psi \quad \models \quad \neg\varphi \vee \psi$$

$$\varphi \leftrightarrow \psi \quad \models \quad (\neg\varphi \vee \psi) \wedge (\neg\psi \vee \varphi)$$

2 regulile De Morgan

$$\neg(\varphi \vee \psi) \quad \models \quad \neg\varphi \wedge \neg\psi$$

$$\neg(\varphi \wedge \psi) \quad \models \quad \neg\varphi \vee \neg\psi$$

3 principiul dublei negații

$$\neg\neg\psi \quad \models \quad \psi$$

4 distributivitatea

$$\varphi \vee (\psi \wedge \chi) \quad \models \quad (\varphi \vee \psi) \wedge (\varphi \vee \chi)$$

$$(\psi \wedge \chi) \vee \varphi \quad \models \quad (\psi \vee \varphi) \wedge (\chi \vee \varphi)$$

Forma clauzală în logica de ordinul I

- O formulă este **formă normală conjunctivă prenex (FNCP)** dacă
 - este în formă prenex $Q_1x_1 \dots Q_nx_n\psi$ ($Q_i \in \{\forall, \exists\}$ oricare i)
 - ψ este **FNC**
- O formulă este **formă clauzală** dacă este **enunț universal** și **FNCP**:
$$\forall x_1 \dots \forall x_n \psi$$
 unde ψ este **FNC**
- Pentru orice formulă φ din logica de ordinul I există o formă clauzală φ^{fc} astfel încât
$$\varphi \text{ este satisfiabilă dacă și numai dacă } \varphi^{fc} \text{ este satisfiabilă}$$
- Pentru o formulă φ , **forma clauzală** φ^{fc} se poate calcula astfel:
 - 1 se determină forma rectificată
 - 2 se cuantifică universal variabilele libere
 - 3 se determină forma prenex
 - 4 se determină forma Skolem

în acest moment am obținut o formă Skolem $\forall x_1 \dots \forall x_n \psi$

 - 5 se determină o FNC ψ' astfel încât $\psi \models \psi'$
 - 6 φ^{fc} este $\forall x_1 \dots \forall x_n \psi'$

Clauze

- O **clauză** este o **disjuncție de literali**.
- Dacă L_1, \dots, L_n sunt literali atunci clauza $L_1 \vee \dots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \dots, L_n\}$
clauză = mulțime de literali
- Clauza $C = \{L_1, \dots, L_n\}$ este **satisfiabilă** dacă $L_1 \vee \dots \vee L_n$ este satisfiabilă.
- O clauză C este **trivială** dacă conține un literal și complementul lui.
- Când $n = 0$ obținem **clauza vidă**, care se notează \square
- Prin definiție, **clauza \square nu este satisfiabilă**.

Forma clauzală

- Observăm că o FNC este o **conjuncție de clauze**.
- Dacă C_1, \dots, C_k sunt clauze atunci $C_1 \wedge \dots \wedge C_k$ o vom scrie ca mulțimea $\{C_1, \dots, C_k\}$
FNC = mulțime de clauze
- O mulțime de clauze $\mathcal{C} = \{C_1, \dots, C_k\}$ este **satisfiabilă** dacă $C_1 \wedge \dots \wedge C_k$ este satisfiabilă
- Când $k = 0$ obținem **mulțimea de clauze vidă**, pe care o notăm $\{\}$
- Prin definiție, mulțimea de clauze vidă $\{\}$ este **satisfiabilă**.

$\{\}$ este satisfiabilă, dar $\{\square\}$ nu este satisfiabilă

Forma clauzală

- Dacă φ este o formulă în **calculul propozițional**, atunci

$$\varphi^{fc} = \bigwedge_{i=1}^k \bigvee_{j=1}^{n_i} L_{ij} \text{ unde } L_{ij} \text{ sunt literali}$$

- Dacă φ o formulă în **logica de ordinul I**, atunci

$$\varphi^{fc} = \forall x_1 \dots \forall x_n \left(\bigwedge_{i=1}^k \bigvee_{j=1}^{n_i} L_{ij} \right) \text{ unde } L_{ij} \text{ sunt literali}$$

φ **este satisfiabilă** dacă și numai dacă

φ^{fc} este satisfiabilă dacă și numai dacă

$\{\{L_{11}, \dots, L_{1n_1}\}, \dots, \{L_{k1}, \dots, L_{kn_k}\}\}$ **este satisfiabilă**

Rezoluția în logica propozițională

Regula rezoluției

$$\text{Rez} \quad \frac{C_1 \cup \{p\}, C_2 \cup \{\neg p\}}{C_1 \cup C_2}$$

unde C_1, C_2 clauze, iar p este variabila propozițională astfel încât $\{p, \neg p\} \cap C_1 = \emptyset$ și $\{p, \neg p\} \cap C_2 = \emptyset$.

Fie \mathcal{C} o mulțime de clauze. O **derivare prin rezoluție** din \mathcal{C} este o secvență finită de clauze astfel încât fiecare clauză este din \mathcal{C} sau rezultă din clauzele anterioare prin rezoluție (este **rezolvent**).

Derivare prin rezoluție

Fie \mathcal{C} o mulțime de clauze. O **derivare prin rezoluție** din \mathcal{C} este o secvență finită de clauze astfel încât fiecare clauză este din \mathcal{C} sau rezultă din clauzele anterioare prin rezoluție (este **rezolvent**).

Exemplu

Fie $\mathcal{C} = \{\{\neg q, \neg p\}, \{q\}, \{p\}\}$ o mulțime de clauze. O derivare prin rezoluție pentru \square din \mathcal{C} este

$$C_1 = \{\neg q, \neg p\}$$

$$C_2 = \{q\}$$

$$C_3 = \{\neg p\} \quad (\text{Rez}, C_1, C_2)$$

$$C_4 = \{p\}$$

$$C_5 = \square \quad (\text{Rez}, C_3, C_4)$$

Teorema de completitudine

$\models \varphi$ dacă și numai dacă există o derivare prin rezoluție a lui \square din $(\neg\varphi)^{fc}$.

Procedura Davis-Putnam DPP (informal)

Intrare: o mulțime \mathcal{C} de clauze

Se repetă următorii pași:

- se elimină clauzele triviale
- se alege o variabilă p
- se adaugă la mulțimea de clauze toți rezolvenții obținuți prin aplicarea *Rez* pe variabila p
- se șterg toate clauzele care conțin p sau $\neg p$

Ieșire: dacă la un pas s-a obținut □, mulțimea \mathcal{C} nu este satisfiabilă; altfel \mathcal{C} este satisfiabilă.

Logica Horn

Clauze definite. Programe logice. Clauze Horn

□ clauză:

$$\{\neg Q_1, \dots, \neg Q_n, P_1, \dots, P_k\} \quad \text{sau} \quad Q_1 \wedge \dots \wedge Q_n \rightarrow P_1 \vee \dots \vee P_k$$

unde $n, k \geq 0$ și $Q_1, \dots, Q_n, P_1, \dots, P_k$ sunt formule atomice.

□ clauză program definită: $k = 1$

□ cazul $n > 0$: $Q_1 \wedge \dots \wedge Q_n \rightarrow P$

□ cazul $n = 0$: $\top \rightarrow P$ (clauză unitate, fapt)

Program logic definit = mulțime finită de clauze definite

□ scop definit (țintă, întrebare): $k=0$

□ $Q_1 \wedge \dots \wedge Q_n \rightarrow \perp$

□ clauza vidă □: $n = k = 0$

Clauza Horn = clauză program definită sau clauză scop ($k \leq 1$)

Programare logica

- Logica clauzelor definite/Logica Horn: un fragment al logicii de ordinul I în care singurele formule admise sunt clauze Horn
 - formule atomice: $P(t_1, \dots, t_n)$
 - $Q_1 \wedge \dots \wedge Q_n \rightarrow P$
unde toate Q_i, P sunt formule atomice, \top sau \perp
- Problema programării logice: reprezentăm cunoștințele ca o mulțime de clauze definite KB și suntem interesați să aflăm răspunsul la o întrebare de forma $Q_1 \wedge \dots \wedge Q_n$, unde toate Q_i sunt formule atomice
$$KB \models Q_1 \wedge \dots \wedge Q_n$$
 - Variabilele din KB sunt cuantificate universal.
 - Variabilele din Q_1, \dots, Q_n sunt cuantificate existențial.

Limbajul PROLOG are la bază logica clauzelor Horn.

Sistem de deducție *backchain*

Sistem de deducție pentru clauze Horn

Pentru un program logic definit KB avem

- **Axiome:** orice clauză din KB
- **Regula de deducție:** regula *backchain*

$$\frac{\theta(Q_1) \quad \theta(Q_2) \quad \dots \quad \theta(Q_n) \quad (Q_1 \wedge Q_2 \wedge \dots \wedge Q_n \rightarrow P)}{\theta(Q)}$$

unde $Q_1 \wedge Q_2 \wedge \dots \wedge Q_n \rightarrow P \in KB$, iar θ este cgu pentru Q și P .

Sistem de deducție

$$\frac{\theta(Q_1) \quad \theta(Q_2) \quad \dots \quad \theta(Q_n) \quad (Q_1 \wedge Q_2 \wedge \dots \wedge Q_n \rightarrow P)}{\theta(Q)}$$

unde $Q_1 \wedge Q_2 \wedge \dots \wedge Q_n \rightarrow P \in KB$, iar θ este cgu pentru Q și P .

Exemplu

KB conține următoarele clauze definite:

father(jon, ken). father(ken, liz).

father(X, Y) → ancestor(X, Y)

daugther(X, Y) → ancestor(Y, X)

ancestor(X, Y) ∧ ancestor(Y, Z) → ancestor(X, Z)

atunci

$$\frac{\frac{father(ken, liz)}{father(ken, Z)} \quad (father(Y, X) \rightarrow ancestor(Y, X))}{ancestor(ken, Z)}$$

Rezoluția SLD

Rezoluția SLD

Fie T o mulțime de clauze definite.

$$\text{SLD} \quad \boxed{\frac{\neg Q_1 \vee \dots \vee \neg Q_i \vee \dots \vee \neg Q_n}{\theta(\neg Q_1 \vee \dots \vee \neg P_1 \vee \dots \vee \neg P_m \vee \dots \vee \neg Q_n)}}$$

unde

- $Q \vee \neg P_1 \vee \dots \vee \neg P_m$ este o clauză definită din KB (în care toate variabilele au fost redenumite) și
- variabilele din $Q \vee \neg P_1 \vee \dots \vee \neg P_m$ și Q_i se redenumesc
- θ este c.g.u pentru Q_i și Q

Rezoluția SLD

Fie KB o mulțime de clauze definite și $Q_1 \wedge \dots \wedge Q_m$ o întrebare, unde Q_i sunt formule atomice.

- O **derivare** din KB prin rezoluție SLD este o secvență

$$G_0 := \neg Q_1 \vee \dots \vee \neg Q_m, \quad G_1, \quad \dots, \quad G_k, \dots$$

în care G_{i+1} se obține din G_i prin regula **SLD**.

- Dacă există un k cu $G_k = \square$ (clauza vidă), atunci derivarea se numește **SLD-respingere**.

Rezoluția SLD

Exercițiu

Găsiți o SLD-respingere pentru următorul program Prolog și ținta:

1. $p(X) :- q(X, f(Y)), r(a).$ $?- p(X), q(Y, Z).$
2. $p(X) :- r(X).$
3. $q(X, Y) :- p(Y).$
4. $r(X) :- q(X, Y).$
5. $r(f(b)).$

Soluție

$G_0 = \neg p(X) \vee \neg q(Y, Z)$	
$G_1 = \neg r(X_1) \vee \neg q(Y, Z)$	(2 cu $\theta(X) = X_1$)
$G_2 = \neg q(Y, Z)$	(5 cu $\theta(X_1) = f(b)$)
$G_3 = \neg p(Z_1)$	(3 cu $\theta(X) = Y_1$ și $\theta(Y) = Z_1$)
$G_4 = \neg r(X)$	(2 cu $\theta(Z_1) = X$)
$G_5 = \square$	(5 cu $\theta(X) = f(b)$)

Rezoluția SLD - arbori de căutare

Arbori SLD

- Presupunem că avem o mulțime de clauze definite KB și o țintă $G_0 = \neg Q_1 \vee \dots \vee \neg Q_m$
- Construim un arbore de căutare (arbore SLD) astfel:
 - Fiecare nod al arborelui este o țintă (posibil vidă)
 - Rădăcina este G_0
 - Dacă arborele are un nod G_i , iar G_{i+1} se obține din G_i folosind regula SLD folosind o clauză $C_i \in KB$, atunci nodul G_i are copilul G_{i+1} . Muchia dintre G_i și G_{i+1} este etichetată cu C_i .
- Dacă un arbore SLD cu rădăcina G_0 are o frunză \square (clauza vidă), atunci există o SLD-respingere a lui G_0 din KB .

Rezoluția SLD - arbore de căutare complet

Exercițiu

Desenați arborele SLD pentru programul Prolog de mai jos și ținta
 $?- p(X,X).$

- | | |
|--------------------------------|----------------------|
| 1. $p(X,Y) :- q(X,Z), r(Z,Y).$ | 7. $s(X) :- t(X,a).$ |
| 2. $p(X,X) :- s(X).$ | 8. $s(X) :- t(X,b).$ |
| 3. $q(X,b).$ | 9. $s(X) :- t(X,X).$ |
| 4. $q(b,a).$ | 10. $t(a,b).$ |
| 5. $q(X,a) :- r(a,X).$ | 11. $t(b,a).$ |
| 6. $r(b,a).$ | |

Rezoluția SLD - arbore SLD complet

1. $p(X,Y) :- q(X,Z), r(Z,Y).$
2. $p(X,X) :- s(X).$
3. $q(X,b).$

$p(X, Y) \vee \neg q(X, Z) \vee \neg r(Z, Y)$
 $p(X, X) \vee \neg s(X)$
 $q(X, b)$

4. $q(b,a).$
5. $q(X,a) :- r(a,X).$
6. $r(b,a).$

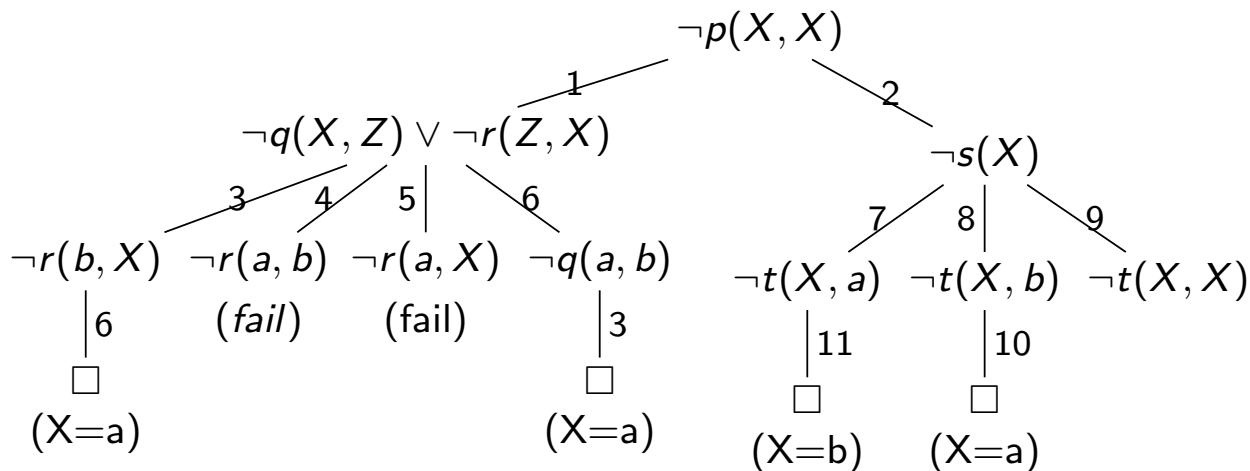
$q(b, a)$
 $q(X, a) \vee \neg r(a, X)$
 $r(b, a)$

7. $s(X) :- t(X,a).$
8. $s(X) :- t(X,b).$
9. $s(X) :- t(X,X).$

$s(X) \vee \neg t(X, a)$
 $s(X) \vee \neg t(X, b)$
 $s(X) \vee \neg t(X, X)$

10. $t(a,b).$
11. $t(b,a).$

$t(a, b)$
 $t(b, a)$



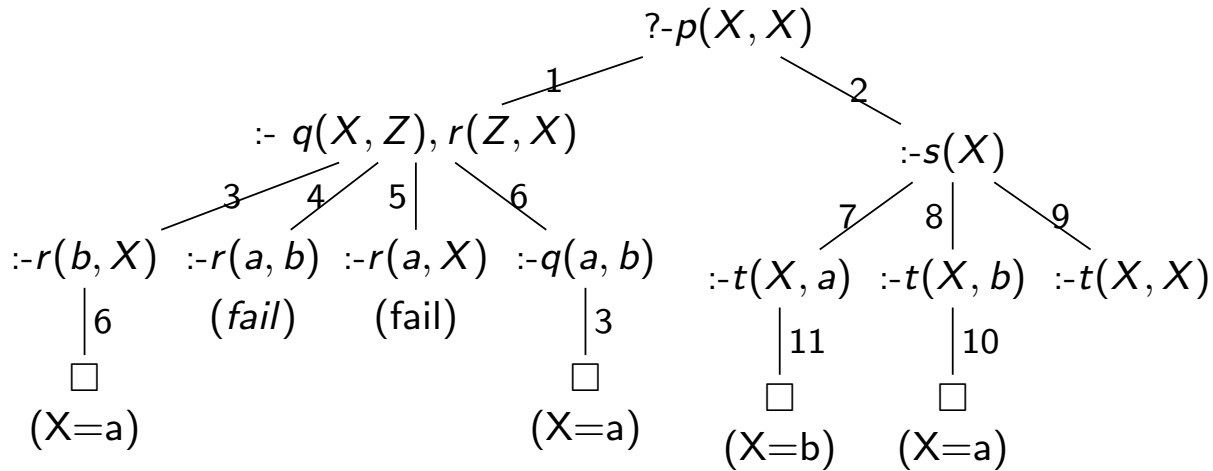
Rezoluția SLD - arbori de execuție

1. $p(X,Y) :- q(X,Z), r(Z,Y).$
 2. $p(X,X) :- s(X).$
 3. $q(X,b).$

4. $q(b,a).$
 5. $q(X,a) :- r(a,X).$
 6. $r(b,a).$

7. $s(X) :- t(X,a).$
 8. $s(X) :- t(X,b).$
 9. $s(X) :- t(X,X).$

10. $t(a,b).$
 11. $t(b,a).$



Rezoluția SLD

Exercițiu

Fie KB următoarea bază de cunoștințe definită în Prolog:

1. $r(a, a)$
2. $q(X, a)$
3. $p(X, Y) :- q(X, Z), r(Z, Y)$

(a) Desenați arborele SLD și arborele de execuție pentru întrebarea $?- p(X, Z)$

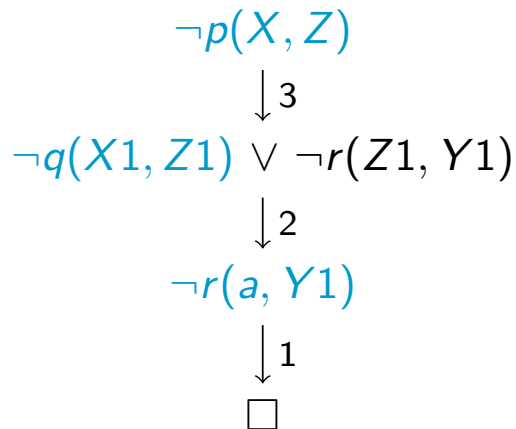
(b) Exprimați KB ca o mulțime de formule în logica de ordinul I demonstrați folosind rezoluția că din KB se deduce $p(X, Z)$, adică $KB \vdash \exists x \exists z p(x, z)$.

Rezoluția SLD

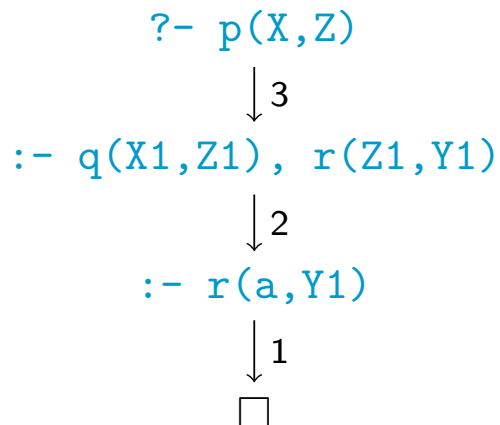
(a) Soluție:

1. $r(a, a).$
2. $q(X, a).$
3. $p(X, Y) :- q(X, Z), r(Z, Y)$

Arborele SLD:



Arborele de execuție:



Rezoluția SLD

(cont.)

Fie KB următoarea bază de cunoștințe definită în Prolog:

1. $r(a, a)$. 2. $q(X, a)$. 3. $p(X, Y) :- q(X, Z), r(Z, Y)$

(b) Soluție:

$KB = \{r(a, a), \forall x q(x, a), \forall x \forall y \forall z (\neg q(x, y) \vee \neg r(z, y) \vee p(x, y))\}$

$KB \vdash \exists x \exists z p(x, z)$ dacă și numai dacă există o derivare prin rezoluție pentru \square din forma clauzală a mulțimii $KB \cup \{\neg(\exists x \exists z p(x, z))\}$.

Forma clauzală a mulțimii $KB \cup \{\neg(\exists x \exists z p(x, z))\}$ este

$\mathcal{C} = \{\{r(a, a)\}, \{q(x, a)\}, \{\neg q(x, y), \neg r(z, y), p(x, y)\}, \{\neg p(x, z)\}\}$. Se

face derivarea direct sau se construiește arborele SLD.