

Suport Proiect 1

- Un limbaj de programare -


2013-2014

Programare Logică

Un limbaj de programare

- Vom defini un limbaj de programare simplu, care are doar atribuiri si bucle.
- Metoda dupa care vom construi limbaje de programare este urmatoarea:
 - 1 Definim *sintaxa* limbajului
 - 2 Definim *semantica* limbajului

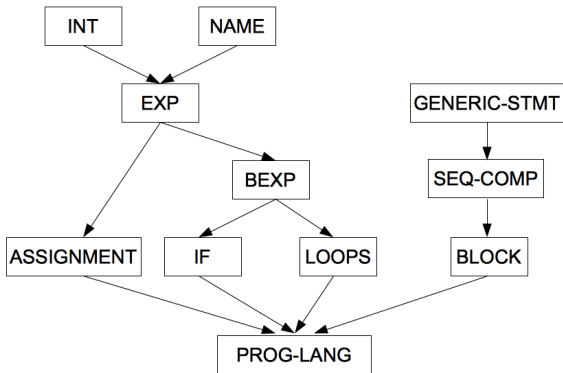
Sintaxa

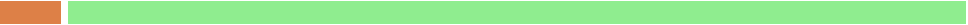


Limbajul pe care il vom defini va contine urmatoarele elemente de baza:

- ☐ numere intregi
- ☐ variabile
- ☐ expresii formate din intregi, variabile si operatori aritmetici
- ☐ instructiunea de atribuire
- ☐ expresii booleene
- ☐ instructiuni conditionate
- ☐ bucle

Aceste elemente vor fi introduse modular dupa schema de mai jos:



- 
- incepem prin a defini sintaxa limbajului dorit
 - un avantaj al definirii limbajului de programare in Maude este acela ca obtinem un parser pentru sintaxa aproape fara niciun efort
 - aceasta facilitate este data de notatia mix-fix, de precedenta operatorilor si de parserul predefinit pentru aceasta notatie

Modulul NAME-SYNTAX

- incepem prin a defini numele pentru variabilele ce pot fi folosite in limbajul dorit
- putem folosi orice identificator oferit de modulul predefinit **QID**
- in plus, definim prin constante de sort **Name** numele formate dintr-o litera

```
fmod NAME-SYNTAX is protecting QID .  
    sort Name .  
    subsort Qid < Name .  
    ops a b c d e f g h i j k l m n  
        o p q r s t u v x y z w : -> Name .  
endfm
```

Modulul EXP-SYNTAX

- urmatorul modul introduce expresiile din limbaj
- expresiile sunt construite din numere intregi si nume folosind operatorii aritmetici uzuali

```
fmod EXP-SYNTAX is protecting NAME-SYNTAX .  
  protecting INT .  
  sort Exp .  
  subsorts Int Name < Exp .  
  op _+_ : Exp Exp -> Exp [ditto] .  
  op _-_ : Exp Exp -> Exp [ditto] .  
  op _*_ : Exp Exp -> Exp [ditto] .  
  op _/_ : Exp Exp -> Exp [prec 31] .  
endfm
```


Modulul GENERIC-STMT-SYNTAX

- daca nu stim de la inceput ce instructiuni vrem sa folosim in limbajul de programare dezvoltat, vrem sa permitem includerea de noi instructiuni in orice moment al dezvoltarii limbajului
- vom defini un modul **GENERIC-STMT-SYNTAX** ce va fi importat de alte module ce definesc cazuri particulare de instructiuni

```
fmod GENERIC-STMT-SYNTAX is
  sort Stmt .
  op skip : -> Stmt .
endfm
```

Modulul ASSIGNMENT-SYNTAX

- Urmatorul modul defineste instructiunea de atribuire:

```
fmod ASSIGNMENT-SYNTAX is
    extending GENERIC-STMT-SYNTAX .
    protecting EXP-SYNTAX .
    op _=_ : Name Exp -> Stmt [prec 40] .
endfm
```

Modulul SEQ-COMP-SYNTAX

- compunerea secventiala de instructiuni este esentiala in orice limbaj de programare
- aceasta se obtine, de obicei, folosind ;
- preferam ca in limbajul definit, folosirea ; sa fie optionala

```
fmod SEQ-COMP-SYNTAX is
  protecting GENERIC-STMT-SYNTAX .
  sort StmtList .
  subsort Stmt < StmtList .
  op _ _ : StmtList StmtList -> StmtList [assoc] .
  op _;_ : StmtList StmtList -> StmtList [assoc] .
endfm
```

Modulul BLOCK-SYNTAX

- introducem in continuare blocuri
- acestea sunt obtinute inchizand o secventa de instructiuni, ca in C, C++, Java etc., folosind acolade
- in limbajul de fata, blocurile sunt parsate ca instructiuni obisnuite

```
fmod BLOCK-SYNTAX is
    extending SEQ-COMP-SYNTAX .
    op {_} : StmtList -> Stmt .
endfm
```

Modulul BEXP-SYNTAX

- expresiile booleene sunt necesare in definirea unor instructiuni importante, cum ar fi cele conditionate sau buclele

```
fmod BEXP-SYNTAX is protecting EXP-SYNTAX .  
  sort BExp .  
  op _equals_ : Exp Exp -> BExp .  
  op zero? : Exp -> BExp .  
  op even? : Exp -> BExp .  
  op not_ : BExp -> BExp .  
  op _and_ : BExp BExp -> BExp .  
endfm
```

Modulul IF-SYNTAX

- orice limbaj de programare ar trebui sa aiba expresii conditionate

```
fmod IF-SYNTAX is
  protecting BEXP-SYNTAX .
  extending GENERIC-STMT-SYNTAX .
  op if_then_else_ : BExp Stmt Stmt -> Stmt .
endfm
```

- observati ca importam atat modulul **BEXP-SYNTAX**, cat si **GENERIC-STMT-SYNTAX**
- metodologia noastra este aceea de a importa cat mai putine module atunci cand definim facilitati noi
- in acest mod, putem pastra arhitectura cat mai flexibila si cat mai modulara

Modulul LOOPS-SYNTAX

- repetitia se obtine, de obicei, prin bucle **for**, bucle **while**, bucle **do...until** etc.
- in limbajul de fata vom considera doar primele doua tipuri de bucle enumerate

```
fmod LOOPS-SYNTAX is
  extending BEXP-SYNTAX .
  extending GENERIC-STMT-SYNTAX .
  op for(_;;_)_ : Stmt BExp Stmt Stmt -> Stmt .
  op while__ : BExp Stmt -> Stmt .
endfm
```

Modulul PROG-LANG-SYNTAX

- putem reuni toate elementele pentru care am definit sintaxa cu scopul de a obtine primul limbaj de programare

```
fmod PROG-LANG-SYNTAX is
  extending ASSIGNMENT-SYNTAX .
  extending BLOCK-SYNTAX .
  extending IF-SYNTAX .
  extending LOOPS-SYNTAX .
  sort Pgm .
  op __ : StmtList Exp -> Pgm .
  op _;_ : StmtList Exp -> Pgm .
endfm
```


Cum arata programele in acest limbaj?

- programele in limbajul dezvoltat constau intr-o serie de instructiuni urmate de o expresie
- intuitiv, aceste programe pot fi gandite astfel:
instructiunile sunt executate, expresia finala este evaluata, iar apoi rezultatul sau este returnat ca rezultat al executiei

Parsarea - Program 1

Urmatorul program foloseste o bucla **for** pentru a calcula puterea x^y :

parse

```
x = 17 ;  
y = 100 ;  
p = 1 ;  
for(i = y ; not zero?(i) ; i = i - 1) {  
    p = p * x  
}  
p
```

.

***> should be Pgm

□ observati ca cele patru instructiuni sunt separate prin ;

□ expresia rezultat este adaugata folosind operatorul

$_{_} : \text{StmList Exp} \rightarrow \text{Pgm}$

□ daca nu am fi permis concatenarea fara ; ca operator, ar fi trebuit sa folosim ; chiar si dupa }

Parsarea - Program 2

- *Sirul lui Fibonacci* are urmatoarea proprietate:

$$f_0 = 0, f_1 = 1 \text{ si } f_{n+2} = f_{n+1} + f_n, \text{ pentru orice } n \geq 2$$

- urmatorul program este o implementare clasica a sirului lui Fibonacci folosind doar doua variabile

parse

```
x = 0 ;
y = 1 ;
n = 1000 ;
for(i = 0 ; not(i equals n) ; i = i + 1) {
    y = y + x ;
    x = y - x
}
y
```

.

***> should be Pgm

parse

$$c = 0 ;$$

```
c = c + 1 ;
```

```
if even?(n)
```

then $n = n / 2$

```
else n = 3 * n + 1
```

}

C


```

.          ***> should be Pgm

```



Semantica

- 
- definim semantica limbajul de programare ca o serie de specificatii in Maude
 - deoarece Maude este executabil, vom obtine un interpretor pentru limbajul de programare construit

Stare

- pentru a vorbi despre semantica programelor, trebuie sa introducem intai notiunea de stare
- o *stare* poate fi gandita intuitiv ca o structura de date ce stocheaza toate informatiile necesare pentru a defini 'intelesul' fiecarui limbaj de programare construit
- pentru limbajul nostru simplificat, avem nevoie doar de valoarea asociata fiecarui nume
- astfel, putem evalua orice expresie cautand in stare valorile corespunzatoare numelor din expresie
- deoarece avem definite atribuirii in limbaj, vom avea nevoie sa adaugam o noua valoare sau sa schimbam una deja existenta asociata numelor intr-o stare

Stare

- este evident ca notiunea de stare este necesara in procesul de definire al semanticii oricarui limbaj de programare
- nu vom defini modulul **STATE** pentru sintaxa unui limbaj anume, ci vom lucra la modul general
- astfel, vom considera ca o stare asociaza o valoare intreaga unui *index* generic
- apare natural ideea de a defini o stare ca o multime de perechi (index,intreg), impreuna cu operatiile de cautare si actualizare

Modulul STATE

```
fmod STATE is protecting INT .
  sorts Index State .
  op empty : -> State .
  op [_,_] : Index Int -> State .
  op __ : State State -> State [assoc comm id: empty] .
  op _[_] : State Index -> Int .
  op _[_<-_] : State Index Int -> State .
  var X : Index . vars I I : Int . var S : State .
  eq ([X,I] S)[X] = I .
  eq ([X,I] S)[X <- I] = [X,I] S .
  eq S[X <- I] = S [X,I] [owise] .
endfm
```

Semantica numelor


- fiind data o stare S , semantica unui nume X in S este valoarea lui X in S , adica $S[X]$
- definim o noua operatie $eval$ ce da semantica oricarui nume in orice stare

```
fmod NAME-SEMANTICS is
  protecting NAME-SYNTAX .
  protecting STATE .
  subsort Name < Index .
  op eval : Name State -> Int .
  var X : Name . var S : State .
  eq eval(X, S) = S[X] .
endfm
```

Semantica expresiilor

- operatorul se extinde la expresii in mod natural

```
fmod EXP-SEMANTICS is
  protecting EXP-SYNTAX .
  protecting NAME-SEMANTICS .
  op eval : Exp State -> Int .
  vars E E : Exp . var I : Int . var S : State .
  eq eval(I, S) = I .
  eq eval(E + E, S) = eval(E, S) + eval(E, S) .
  eq eval(E - E, S) = eval(E, S) - eval(E, S) .
  eq eval(E * E, S) = eval(E, S) * eval(E, S) .
  eq eval(E / E, S) = eval(E, S) quo eval(E, S) .
endfm
```

- 
- pentru a evalua $E + E'$ intr-o stare S , evaluam intai E , apoi E' , iar apoi adunam cei doi intregi obtinuti (operatiile din termenii din partea dreapta sunt definite in modulul predefinit `INT`)

Semantica instructiunilor

- semantica instructiunilor difera de cea a expresiilor, deoarece instructiunile schimba starea programului.
- definim o operatie *state* care are ca argumente o instructiune si o stare, si intoarce starea dupa ce instructiunea este executata in starea data

```
fmod GENERIC-STMT-SEMANTICS is
  protecting GENERIC-STMT-SYNTAX .
  protecting STATE .
  op state : Stmt State -> State .
  eq state(skip, S:State) = S:State .
endfm
```

- instructiunea `skip` nu are efect, deci nu schimba starea programului
- observati ca am declarat variabila `S` de sort `State` "on-the-fly", fara a folosi cuvantul `var`
- aceasta facilitate este utila intr-o ecuatie care nu contine multe variabile, sau cand o variabila este folosita doar intr-o ecuatie

Semantica unei atribuirii

- pentru a atribui o expresie unui nume, trebuie intai sa evaluam expresia in starea curenta, si apoi sa actualizam starea

```
fmod ASSIGNMENT-SEMANTICS is
  protecting ASSIGNMENT-SYNTAX .
  extending GENERIC-STMT-SEMANTICS .
  extending EXP-SEMANTICS .
  var X : Name . var E : Exp . var S : State .
  eq state(X = E, S) = S[X <- eval(E,S)] .
endfm
```


Semantica compunerii secventiale

O secventa de instructiuni modifica starea inductiv:

```
fmod SEQ-COMP-SEMANTICS is
  protecting SEQ-COMP-SYNTAX .
  extending GENERIC-STMT-SEMANTICS .
  op state : StmtList State -> State .
  var St : Stmt . var Stl : StmtList .
  var S : State .
  eq St Stl = St ; Stl .
  eq state(St ; Stl, S) =
    state(Stl, state(St, S)) .
endfm
```

Semantica unui bloc

Un bloc este echivalent cu executia secventei de instructiuni din blocul respectiv:

```
fmod BLOCK-SEMANTICS is
  protecting BLOCK-SYNTAX .
  extending SEQ-COMP-SEMANTICS .
  var Stl : StmtList . var S : State .
  eq state({Stl}, S) = state(Stl, S) .
endfm
```

Semantica expresiilor booleene

- definim o noua operatie **eval** pe expresii booleene, care foloseste operatia **eval** pe expresii

```
fmod BEXP-SEMANTICS is protecting BEXP-SYNTAX .  
  protecting EXP-SEMANTICS .  
  protecting STATE .  
  op eval : BExp State -> Bool .  
  vars E E : Exp . vars BE BE : BExp .  
  var S : State .  
  eq eval(E equals E, S) =  
    eval(E, S) == eval(E, S) .  
  eq eval(zero?(E), S) = eval(E, S) == 0 .  
  eq eval(even?(E), S) = eval(E, S) rem 2 == 0 .  
  eq eval(not BE, S) = not eval(BE, S) .  
  eq eval(BE and BE, S) =  
    eval(BE, S) and eval(BE, S) .  
endfm
```

Semantica instructiunilor conditionate

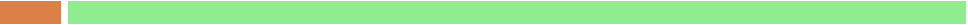
- o instructiune conditionata schimba starea programului in functie de valoarea expresiei booleene
- daca este **true**, atunci semantica instructiunii conditionate este echivalenta cu a evalua partea de dupa **then**, altfel este echivalenta cu a evalua partea de dupa **else**


```
fmod IF-SEMANTICS is
  protecting IF-SYNTAX .
  protecting BEXP-SEMANTICS .
  extending GENERIC-STMT-SEMANTICS .
  var BE : BExp . vars St St : Stmt .
  var S : State .
  eq state(if BE then St else St, S) =
    if eval(BE, S) then state(St, S)
      else state(St, S) fi .
endfm
```

Semantica buclelor

- pentru usurinta, vom exprima intai bucla **for** ca o bucla **while**, si apoi vom defini doar semantica lui **while**

```
fmod LOOPS-SEMANTICS is
  protecting LOOPS-SYNTAX .
  protecting BEXP-SEMANTICS .
  extending BLOCK-SEMANTICS .
  op for(_;;_)_ : Stmt BExp Stmt Stmt -> Stmt .
  op while__ : BExp Stmt -> Stmt .
  vars St St1 St2 St3 : Stmt .
  var BE : BExp . var S : State .
  eq for(St1 ; BE ; St2) St3 =
    St1 ; while BE {St3 ; St2} .
  eq state(while BE St, S) =
    if eval(BE, S)
      then state(while BE St, state(St, S))
      else S fi .
endfm
```

- 
- ecuatia pentru **while** surprinde esenta acestui tip de bucla: corpul sau este executat cat timp conditia este satisfacuta

- 
- acum putem reuni toate piesele pentru a defini semantica limbajului de programare dezvoltat
 - 'sensul' unui program este acela de a evalua expresia rezultat in starea generata de secventa de instructiuni ce o precede

```
fmod PROG-LANG-SEMANTICS is
  protecting PROG-LANG-SYNTAX .
  extending ASSIGNMENT-SEMANTICS .
  extending BLOCK-SEMANTICS .
  extending IF-SEMANTICS .
  extending LOOPS-SEMANTICS .
  op eval : Pgm -> Int .
  var Stl : StmtList . var E : Exp .
  eq Stl E = Stl ; E .
  eq eval(Stl ; E) = eval(E, state(Stl, empty)) .
endfm
```


Obtinerea unui interpretor

- scopul a fost sa *definim* (*specificam*) un limbaj de programare simplu, nu sa il *implementam*
- totusi, deoarece Maude este executabil, avem un *model* al limbajului de programare specificat
- putem folosi acest model ca un *interpretor* pentru limbajul de programare construit, adica avem un 'program' special care executa un program analizand si interpretand instructiunile sale

```
red eval( skip ; 3 + y) .
    ***> should be NzNat: 3 + empty[y]
red eval( x = 1 ; y = x ; y ).
    ***> should be NzNat: 1
red eval( x = 1 ; y + 1 = x ; y) .
    ***> should be [Index,Exp,FindResult]:
        eval((x = 1 ; y + 1 = x) ; y)
red eval( x = 1 ;
    y = (1 + x) * 2 ;
    z = x * 2 + x * y + y * 2 ;
    x + y + z) .
    ***> should be NzNat: 19
```

Program care calculeaza 17^{1000} :

```
red eval(  
    x = 17 ;  
    y = 1000 ;  
    p = 1 ;  
    for(i = y ; not zero?(i) ; i = i - 1) {  
        p = p * x  
    }  
    p  
) .
```

Program care calculeaza al 1000-lea element din sirul lui Fibonacci:

```
red eval(  
    x = 0 ;  
    y = 1 ;  
    n = 1000 ;  
    for(i = 0 ; not(i equals n); i = i + 1) {  
        y = y + x ;  
        x = y - x  
    }  
    y  
) .
```

Conjectura lui Collatz se termina:

```
red eval(  
  n = 1783783426478237597439857348095823098297983475834 ;  
  c = 0 ;  
  while not (n equals 1) {  
    c = c + 1 ;  
    if even?(n)  
      then n = n / 2  
      else n = 3 * n + 1  
  }  
  c  
) .
```



Vezi cerinta Proiect 1!