Laboratorul 9 Prolog

Implementarea IMP - semantica operationala

In acest laborator vom implementa un limbaj care contine:

- expresii *aritmetice* si *booleene*;
  - `x + 3`
  - `x >= 7`
- instructiuni *de atribuire*, *conditionale* si *de ciclare*;
  - `x = 5`
  - `if(x >= 7, x = 5, x = 0)`
  - `while(x >= 7, x = x - 1)`
- compunerea instructiunilor;
  - `x = 7; while(x >= 0; x = x - 1)`
- blocuri de instructiuni.
  - `{x = 7; while(x >= 0, x = x - 1)}`

Un exemplu de program in IMP este

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

9.1. Definitia limbajului (BNF)

```
    E ::= n | x
        | E + E | E - E | E * E
```

```
B ::= true | false
    | E =< E | E >= E | E == E
    | not(B) | and(B, B) | or(B, B)

C ::= skip
    | x = E
    | if(B, C, C)
    | while(B, C)
    | { C }
    | C ; C

P ::= { C }, E
```

9.2. Implementarea limbajului in Prolog

Avem urmatorii doi operatori:

```
:- op(100, xf, {}).
:- op(1100, yf, ;).
```

Definim un predicat pentru fiecare neterminal din descrierea BNF de mai sus. Vom avea:

- `aexp/1` pentru expresii aritmetice;
- `bexp/1` pentru expresii booleene;
- `stmt/1` pentru instructiunile propriu-zise;
- `program/1` pentru structura programului.

9.2.1. Implementarea `aexp/1`

```
E ::= n | x | E + E | E - E | E * E
```

```
aexp(I) :- integer(I).      % intregii sunt cei din Prolog
aexp(X) :- atom(X).         % identificatorii din IMP sunt atomii
aexp(A1 * A2) :- aexp(A1), aexp(A2).
aexp(A1 + A2) :- aexp(A1), aexp(A2).
aexp(A1 - A2) :- aexp(A1), aexp(A2).
```

9.2.2. Implementarea `bexp/1`

```
B ::= true | false
    | E =< E | E >= E | E == E
    | not(B) | and(B, B) | or(B, B)
```

```
bexp(true).
bexp(false).
bexp(A1 =< A2) :- aexp(A1), aexp(A2).
bexp(A1 >= A2) :- aexp(A1), aexp(A2).
bexp(A1 == A2) :- aexp(A1), aexp(A2).
bexp(and(BE1,BE2)) :- bexp(BE1), bexp(BE2).
bexp(or(BE1,BE2)) :- bexp(BE1), bexp(BE2).
bexp(not(BE)) :- bexp(BE).
```

9.2.3. Implementarea `stmt/1`

```
C ::= skip
    | x = E
    | if(B, C, C)
    | while(B, C)
    | { C }
    | C ; C
```

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

9.2.4. Implementarea `program/1`

```
P ::= { C }, E
```

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

Urmatorul predicat trebuie sa raspunda `true`:

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

```
?- test0. => true
```

9.3. Semantica operationala

- descrie cum se executa un program pe o masina abstracta;
- semantica **small-step** descrie cum avanseaza o executie in functie de reduceri succesive. Avem mereu un cuplu (cod, starea memoriei), notat in general $< cod, \sigma >$, iar o tranzitie este $< cod, \sigma > \rightarrow < cod', \sigma' >$

Vom defini care sunt regulile structurale pentru toate neterminalele din programul nostru, dand astfel o semantica a programului.

Avem deja implementate predicatele auxiliare de mai jos:

```
% get ne da informatii despre starea memoriei
% vi/2 este o pereche variabila - valoare
% de exemplu, putem avea S = [vi(X, 0), vi(Y, 1)] care spune
% ca X |-> 0 si Y |-> 1 in stadiul curent
% apelam get(S, X, -I) - dam starea memoriei, o variabila, si in I gasim valoa
get(S,X,I) :- member(vi(X,I),S).
get(_,_,0).

% elimina vi(X, I) din starea curenta a memoriei
set(S,X,I,[vi(X,I)|S1]) :- del(S,X,S1).
del([vi(X,_)|S],X,S).
del([H|S],X,[H|S1]) :- del(S,X,S1) .
del([],_,[]).
```

9.3.1. Regula ID

$$(\text{ID}) < x, \sigma > \rightarrow < i, \sigma >$$

daca $i = \sigma(x)$

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

9.3.2. Regula ADD

$$(\text{ADD1}) < i_1 + i_2, \sigma > \rightarrow < i, \sigma >$$

daca $i = i_1 + i_2$

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

$$(\text{ADD2}) \frac{< a_1, \sigma > \rightarrow < a_1', \sigma >}{< a_1 + a_2, \sigma > \rightarrow < a_1' + a_2, \sigma >}$$

```
smallstepA(AE1 + I, S, AE2 + I, S) :-
  integer(I),
  smallstepA(AE1, S, AE2, S).
```

$$(\text{ADD3}) \frac{< a_2, \sigma > \rightarrow < a_2', \sigma >}{< a_1 + a_2, \sigma > \rightarrow < a_1 + a_2', \sigma >}$$

```
smallstepA(I + AE1,S,I + AE2,S) :-
  integer(I),
  smallstepA(AE1, S, AE2, S).
```

9.3.3. Regula DIFF [de lucrat]

$$(\text{DIFF1}) < i_1 - i_2, \sigma > \rightarrow < i, \sigma >$$

daca $i = i_1 - i_2$

```
smallstepA(...) :- ...
```

$$(\text{DIFF2}) \frac{< a_1, \sigma > \rightarrow < a_2, \sigma >}{< i - a_1, \sigma > \rightarrow < i - a_2, \sigma >}$$

```
smallstepA(...) :- ...
```

$$(\text{DIFF3})\frac{< a_1, \sigma > \to < a_2, \sigma >}{< a_1 - i, \sigma > \to < a_2 - i, \sigma >}$$

```
smallstepA(...) :- ...
```

9.3.4. Regula MUL [de lucrat]

De scris atat regulile, cat si implementarea lor in Prolog.

9.3.5. Reguli de comparatie LEQ

$$(\text{LEQ-FALSE}) < i_1 =< i_2, \sigma > \to < \text{false}, \sigma >$$

daca $i_1 > i_2$

```
smallstepB(I1 =< I2, S, false, S) :-
    integer(I1), integer(I2),
    I1 > I2.
```

$$(\text{LEQ-TRUE}) < i_1 =< i_2, \sigma > \to < \text{true}, \sigma >$$

daca $i_1 \leq i_2$

```
smallstepB(I1 =< I2, S, true, S):-
    integer(I1), integer(I2),
    I1 =< I2.
```

$$(\text{LEQ1})\frac{< a_1, \sigma > \to < a_1', \sigma >}{< a_1 =< a_2, \sigma > \to < a_1' =< a_2, \sigma >}$$

```
smallstepB(AE1 =< I, S, AE2 =< I, S) :-
    integer(I),
    smallstepA(AE1, S, AE2, S).
```

$$(\text{LEQ2})\frac{< a_2, \sigma > \to < a_2', \sigma >}{< a_1 =< a_2, \sigma > \to < a_1 =< a_2', \sigma >}$$

```
smallstepB(I =< AE1, S, I =< AE2, S) :-
     integer(I),
     smallstepA(AE1, S, AE2, S).
```

9.3.6. Reguli de comparatie GEQ [de lucrat]

Sa se scrie atat regulile de derivare, cat si implementarea in Prolog

9.3.7. Reguli de comparatie EQ [de lucrat]

Sa se scrie atat regulile de derivare, cat si implementarea in Prolog

9.3.8. Reguli pentru conjunctie

$$(\text{AND1}) < and(\text{true}, BE_2), \sigma > \rightarrow < BE_2, \sigma >$$

```
smallstepB(and(true, BE2), S, BE2, S).
```

$$(\text{AND2}) < and(\text{false}, \_), \sigma > \rightarrow < \text{false}, \sigma >$$

```
smallstepB(and(false,_), S, false, S).
```

$$(\text{AND3}) \frac{< BE_1, \sigma > \rightarrow < BE_2, \sigma >}{< and(BE_1, BE), \sigma > \rightarrow < and(BE_2, BE), \sigma >}$$

```
smallstepB(and(BE1, BE),S,and(BE2, BE),S) :-
    smallstepB(BE1, S, BE2, S).
```

9.3.9. Reguli pentru disjunctie [de lucrat]

Sa se scrie regulile pentru disjunctie si sa se implementeze in Prolog.

9.3.10. Regulile pentru negatie

$$\text{!-TRUE} < not(\text{true}), \sigma > \rightarrow < \text{false}, \sigma >$$

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

$$\text{!-FALSE} < not(\text{false}), \sigma > \rightarrow < \text{true}, \sigma >$$

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

$$(\text{NEG})\frac{< a, \sigma > \rightarrow < a', \sigma >}{< not(a), \sigma > \rightarrow < not(a'), \sigma >}$$

```
smallstepB(not(BE1),S,not(BE2),S) :-
    smallstepB(BE1,S,BE2,S).
```

9.3.11. Regula ASGN

$$(\text{ASGN1}) < x = i, \sigma > \rightarrow < \text{skip}, \sigma' >$$

daca $\sigma' = \sigma[i/x]$

```
smallstepS(X = AE, S, skip, S1) :-
  integer(AE), set(S, X, AE, S1) .
```

$$(\text{ASGN2})\frac{< e_1, \sigma > \rightarrow < e_2, \sigma >}{< x = e_1, \sigma > \rightarrow < x = e_2, \sigma >}$$

```
smallstepS(X = AE1, S, X = AE2, S) :-
  smallstepA(AE1, S, AE2, S).
```

9.3.12. Regula NEXT-STMT

$$(\text{NEXT-STMT1}) < \text{skip}; s_2, \sigma > \rightarrow < s_2, \sigma >$$

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

$$(\text{NEXT-STMT2})\frac{< s_1, \sigma > \rightarrow < s_1', \sigma' >}{< s_1; s_2, \sigma > \rightarrow < s_1'; s_2, \sigma' >}$$

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

9.3.13. Regula pentru blocuri de cod

$$(\text{BLOCK}) < \{E\}, \sigma > \rightarrow < E, \sigma >$$

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

8.3.14. Reguli pentru IF

$$(\text{IF-TRUE}) < if(\text{true}, St_1, \_), \sigma > \to < St_1, \sigma >$$

```
smallstepS(if(true, St1, _), S, St1, S).
```

$$(\text{IF-FALSE}) < if(\text{false}, \_, St_2), \sigma > \to < St_2, \sigma >$$

```
smallstepS(if(false, _, St2), S, St2, S).
```

$$(\text{IF}) \frac{< BE_1, \sigma > \to < BE_2, \sigma >}{< if(BE_1, St_1, St_2), \sigma > \to < if(BE_2, St_1, St_2), \sigma >}$$

```
smallstepS(if(BE1,St1,St2),S,if(BE2,St1,St2),S) :-
    smallstepB(BE1, S, BE2, S).
```

9.3.15. Regula pentru WHILE

$$\begin{gathered}(\text{WHILE}) < while(BE, St), \sigma > \\ \to < if(BE, (St; while(BE, St)), \text{skip}), \sigma >\end{gathered}$$

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

9.3.16. Reguli pentru programe

```
smallstepP(skip, AE1, S1, skip, AE2, S2) :-
    smallstepA(AE1, S1, AE2, S2).

smallstepP(St1, AE, S1, ST2, AE, S2) :-
    smallstepS(St1, S1, St2, S2).


run(skip, I, _, I) :- integer(I).
run(St1, AE1, S1, I) :-
    smallstepP(St1, AE1, S1, ST2, AE2, S2),
        run(St2, AE2, S2, I).


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

```
defpg(pg1, {nr = 0; while(nr =< 10, nr=nr+1)}, nr).
```