

## Seminarska naloga PPJ 99/00

Opravljena seminarska naloga, je potreben pogoj za opravljanje izpita PPJ (prof. Bratko). Vsak študent si izbere eno izmed treh opisanih nalog, ki jih mora zagovarjati pred koncem semestra. Po predhodnem dogovoru je možna tudi lastna izbira teme seminarske naloge. Dodatne informacije dobite:

- na avditornih vajah pri predmetu PPJ
- na govorilnih urah (Dorian Šuc v LUI, v petkih 11<sup>h</sup>-12<sup>h</sup>)
- na elektronski naslov: Dorian.Suc@fri.uni-lj.si
- moja spletna stran: <http://ai.fri.uni-lj.si/dorian/>

### 1. Simbolično računanje

Prolog je primeren za simbolično reševanje problemov. Z njim lahko zelo preprosto rešujemo številne algebraične probleme kot napr. simbolično reševanje enačb, simbolično odvajanje in simbolično integriranje. Za preproste probleme zadostuje že, da zapišemo pravila za simbolično manipulacijo v drugačni, to je v prologovi sintaksi. Večina ljudi pa ne rešuje takih problemov le s preprosto uporabo aksiomov algebre, temveč uporabljo številne metode oziroma pravila, ki transformirajo problem v lažje rešljiv problem. Vsako tako pravilo ima nek pogoj, ki preverja če pravilo lahko uporabimo in samo izvedbo pravila (napr. integracijo *per partes uporabimo* le v določenih primerih, substucijo določene vrste uporabimo le če ima enačba/integral primočno obliko).

Včasih je primerno da uporabljam rekurziven zapis števil. Vsako pozitivno naravno število n je n-ti naslednik števila 0, torej lahko število zapišemo kot s(s(...s(0)...)). Tako napr. število 3 zapišemo kot s(s(s(0))). Tak zapis števil nam včasih omogoča dvosmernost aritmetičnih operacij v prologu.

```
% dvosmeren procedura za izračun vsote dveh stevil  
sum( 0, N, N ).  
sum( s(N1), N2, s(Sum) ) :-  
    sum( N, N2, Sum ).  
?- sum( s(s(s(0))), X, s(s(s(s(0)))) ). % sum( 3, X, 5 )  
X = s(s(0))
```

Preprost primer simboličnega računaja je simbolično odvajanje. Pravila odvajanja zapišemo v prologovi sintaksi. Spodaj podani program lahko še razširimo s prologovimi pravili, tako da zna odvajati tudi bolj zapletene izraze.

```
% derive( Expression, X, DifferentiatedExpression )  
% DifferentiatedExpression je odvod izraza Expression po sprem. X  
derive( 0, _, 0 ).  
derive( s(_), X, 0 ).  
derive( X, X, s(0) ).  
derive( X^s(N), X, s(N)*X^N ).  
derive( sin(X), X, cos(X) ).  
derive( cos(X), X, -sin(X) ).  
derive( e^X, X, e^X ).  
% ... pravila za odvod drugih mat. funkcij...
```

```
derive( F+G, X, DF+DG ) :-  
    derive( F, X, DF ),  
    derive( G, X, DG ).  
derive( F-G, X, DF-DG ) :-  
    derive( F, X, DF ),  
    derive( G, X, DG ).  
derive( F*G, X, F*DG+DF*G ) :-  
    derive( F, X, DF ),  
    derive( G, X, DG ).  
derive( 1/F, X, -DF/(F*F) ) :-
```

```

derive( F, X, DF ).  

derive( F/G, X, (G*DF-F*D G)/(G*G) ) :-  

    derive( F, X, DF ),  

    derive( G, X, DG ).
```

```
?-derive( 3*x+2, x, D ).  
D = (3*s(0) + 0*x) + 0
```

Pri odvajjanju izrazov smo vajeni, da se izraz tudi poenostavi; odgovor na zgornje vprašanje bi moral biti  $D=3$ . Naš program tega še ne zna. Poenostavljanje izrazov predstavlja eno izmed možnih razširitev podanega programa.

Nekoliko bolj zapleteno je odvajanje izraza  $f(g(x))$  po  $x$ , kjer sta  $f$  in  $g$  poljubni funkciji. Pravilo pravi, da je odvod takega izraza enak odvodu  $f(g(x))$  po  $g(x)$  krat odvod  $g(x)$  po  $x$ . Zaradi predstavitev funkcij z unarnimi strukturami(napr.  $\sin(x)$ ) je potrebno zapisati pravilo za posredno odvajanje za vsako funkcij posebej.

```
derive( sin(U), X, cos(U)*DU ) :-  
    derive( U, X, DU ).
```

Alternativo predstavlja razstavljanje izrazov ali pa drugačna predstavitev funkcij, kar nam omogoča neposreden dostop do imena funkcije. Tako zapišemo  $\sin(x)$  kot  $\text{unary\_term}(\sin, x)$ . Če zgoraj zapisana pravila odvajanja prilagodimo na novo predstavitev funkcij, lahko dodamo še pravilo za posredno odvajanje:

```
derive( unary_term(F,U), X, DF*DU ) :-  
    derive( unary_term(F,U), U, DF ),  
    derive( U, X, DU ).
```

### Definicija nalog iz simboličnega računanja

V prologu implementiraj vsaj eno izmed naslednjih nalog:

- **simbolično odvajanje s poenostavitvijo izrazov:** program mora znati odvajati poljubno kompleksne matematične izraze, ter jih podati v poenostavljeni obliki. Zaradi manjše zahtevnosti simboličnega odvajanja je pri tej nalogi poudarek tudi na poenostavitvi izrazov.
- **simbolično integriranje:** implementacija osnovnih metod integriranja, računanje določenih in nedoločenih integralov, integracija *per partes*, izvedba ustrezne substitucije, uporaba nekaterih znanih integralov, ...
- **simbolično reševanje enačb:** pri podani enačbi  $Lhs=Rhs$  z sprem.  $X$ , pretvori enačbo v ekvivalentno enačbo oblike  $X=Rhs_1$ , kjer  $Rhs_1$  ne vsebuje spremenljivke  $X$ . Enačbi sta ekvivalentni, če lahko eno pretvorimo v drugo v končnem številu korakov uporabe pravil algebре. Pri tem uporabljam reševanje kvadratnih enačb, vpeljavo novih spremenljivk, faktorizacijo, izolacijo in podobno.

#### Primer 1:

```
?-solve(cos(x) * (1-2*sin(x)) = 0, x, Sol).  
1          cos(x)=0 ali 2. (1-2*sin(x)=0)      (faktorizacija)  
1.1        cos(x) = 0  
1.2        x = arc cos(0)                  (izolacija-na obeh straneh izvedemo  $\cos^{-1}$ )  
Sol = (x =  $\pi/2 + k\pi$ );  
2.1        1 - 2*sin(x) = 0  
2.2        2*sin(x) = 1                      (izolacija izraza s sprem. x)  
2.3        sin(x) = 1/2                     (izolacija)  
2.4        x = arc sin(1/2)                 (izolacija)  
Sol = (x =  $\pi/6 + 2k\pi$ );  
no
```

#### Primer 2:

```
?-solve(2^(2*x) - 5*2^(x+1) + 16 = 0, x, Sol ).  
1          y=2^x                            (vpeljava nove sprem.)  
2          y^2 - 5*y + 16 = 0                (kvadratna ena~ba)
```

$$\det = (-5*2)*(-5*2) - 4*1*16 = 36$$

2.1  $y = (10 - 6)/2 = 2$  (re{ kvadratne ena~be})  
 2.1.1  $2^x = 2$  ( $y=2^x$ )  
 2.1.2  $x = 1$  (izolacija)  
 Sol =  $(x = 1)$ ;  
 2.2  $y = (10 + 6)/2 = 8$  (re{ kvadratne ena~be})  
 2.2.1  $2^x = 8$  ( $y=2^x$ )  
 2.2.2  $x = 3$  (izolacija)  
 Sol =  $(x = 3)$ ;  
 no

Vhod in izhod iz progama naj bo v razumljivi obliki. Program naj poda tudi uporabljeni pravila, podobno kot v primerih za simbolično reševanje enačb. Ocena seminarske naloge je odvisna predvsem od kompleksnosti problemov, ki jih zna rešiti vaš program, ter tudi od izvirnosti rešitev, splošnosti in učinkovitosti implementacije.

## 2. Obravnavanje omejitve v prologu

Namen naloge je razširitev prologa, tako da zna logično pravilno obravnavati omejitve tudi če se te nanašajo na neopredeljene spremenljivke. To naredimo tako, da napišemo meta-interpreter (interpreter za jezik napisan v istem jeziku), ki zna omejitve previlno upoštevati. Pri tem so omejitve relacije enakosti ozziroma neenakosti.

Cilj(omejitev)  $X > 1$  v prologu ne uspe, če je X neopredeljena spremenljivka. To povzroča, da programi, ki uporabljajo take cilje niso dvosmerni.

Primer: Program za izračun Fibonačijevih števil.

```

fib(0, 1).
fib(1, 1).
fib(N, R) :-
  N > 1, N1 is N - 1,
  fib(N1, R1), N2 is N - 2,
  fib(N2, R2), R is R1 + R2.
?-fib(10, R).
R = 89
  
```

Program deluje pravilno, le v primeru če je prvi argument že opredeljen. Prolog želimo razširiti tako, da pravilno odgovori tudi na vprašanja, ko prvi argument ni opredeljen. Meta-interpreter, ki bi znal obravnavati omejitve, bi se pri podobnem programu obnašal približno takole:

```

?-solve( fib(N, 89) ).
N = 10
?-solve( fib(N, R) ).
N = 0, R = 1 ;
N = 1, R = 1 ;
N = 2, R = 2
  
```

Torej imamo dvosmernost tudi pri predikatih, ki uporabljajo aritmetične operatorje. Za implementacijo te razširitve potrebujemo razširjeno prilagajanje, zakasnitev akcij in manipulacijo omejitev. Cilj(omejitev)  $X > 1$  mora uspeti četudi X še nima opredeljene vrednosti, vendar tak cilj omeji domeno spremenljivke X. Vsaka neopredeljena ima na začetku neko domeno, ki pa se z omejitvami nad to spremenljivko oži. Torej potrebujemo tudi sistem za reševanje linearnih enačb in neenačb.

Običajni prolog ima vgrajeni predikat clause(Head, Body), kjer mora biti Head opredeljen, Body pa se prilagodi s telesom prvega stavka v programu, katerega glava se lahko prilagodi s Head.. Z uporabo vgrajenega predikata lahko realiziramo preprost meta-interpreter, ki se obnaša podobno kot sam prologov interpreter.

```

solve( (G, Rest) ) :-
    solve( G ), solve( Rest ).
solve( (G ; Rest) ) :-
    solve( G ) ; solve( Rest ).
solve( G ) :-
    clause( G, Body ),          % poisce telo stavka z glavo G
    solve( Body ).
solve( G ) :-
    call( G ).                  % izvedba vgrajenega predikata G

```

Če želimo obravnavati omejitve moramo dodati predikatu solve še dva argumenta: omejitve, ki so veljale pred izvedbo ciljev in omejitve, ki veljajo po izvedbi ciljev. Začetne omejitve lahko predstavljajo kar domeno spremenljivk, končne omejitve pa določajo množico rešitev. Implementirati je potrebno predikat myClause(G, Body, C), ki pri podanem cilju G poišče stavek, katerega glava se lahko prilagodi s ciljem G, vrne telo tega stavka, ter nove omejitve C, ki morajo veljati, da se je prilagoditev lahko izvršila. V primeru, da je ustrezni stavek dejtvo, je telo enako atomu true. Predikat myClause najlaže implementiramo tako, da najprej v cilju G vse spremenljivke nadomestimo z neopredeljenimi spremenljivkami, uporabimo prologov predikat clause, in potem ustrezno preimujemo spremenljivke v glavi in telesu stavka; spremenljivke, ki v cilju G niso bile opredeljene določajo nove omejitve C.

Primer:

```

p(X, X).
p(X, Y) :- X < Y.
p(1, Y) :- Y is 2 * 3.
?-myClause( p(A,B), Body, C ).
Body = true, C = [A = B] ;
Body = A < B, C = [] ;
Body = B is 2* 3, C=[A=1] ;
no

```

Poleg predikata myClause je potrebno implementirati še predikat mergeConstraints. Ta predikat združi in poenostavi omejitve(poenuostavitev sistema linearnih enačb/neenačb), in uspe le če omejitve niso protislovne. Predikat mergeConstraints poskrbi za pravilno obravnavanje predikatov kot so  $<$ ,  $=<$ ,  $>$ ,  $>=$ ,  $=$ , ter za predikat is, ki mora uspeti tudi v primeru, če spremenljivke na desni strani niso opredeljene.

```

?-mergeConstraints( X is Y + 2, [X >= 0, Y =< 10], C)
C = [X >= 0, X =< 12, Y >= -2, Y =< 10, X = Y + 2, Y = X - 2] ;
no
?-mergeConstraints( X >= Y, [X = Z+2, Z < 10, Y > 10], C)
% splosne omejitve
% C = [X = Z+2, Z = X-2, X < 12, X > 10, Y > 10, Y =< X, Z < 10]
% ce se omejimo na naravna stevila tedaj so omejitve
C = [ X = 11, Y = 11, Z = 9] ;
no

```

Primer: Shema meta-interpreterja, ki zna pravilno obravnavati omejitve

```

%solve( Goals, InitialConstraints, FinalConstraints )
solve( true, C, C ).
solve( (G, Rest) , C0, C ) :-
    solve( G, C0, C1 ),
    solve( Rest, C1, C ).
solve( (G ; Rest), C0, C ) :-
    solve( G, C0, C1 ),
    solve( Rest, C1, C ).
solve( G, C0, C ) :-
    myClause( G, Body, NewC ),           % vrne telo cilja G ter omejitve, ki morajo veljati
    mergeConstraints( C0, NewC, C1 ),     % zdruzitev in poenostavitev omejitev
    solve( Body, C1, C ).
solve( G, C0, C ) :-
    mergeConstraints( G, C0, C ).         % dodajanje omejitev(napr. cilja X > Y - 2)

```

```
?- solve( fib(N, 89), [N >= 0, N =<1000], _).  
N = 10
```

### **Definicija naloge obravnavanja omejitev v prologu**

Napiši meta-interpreter, ki zna pravilno obravnavati omejitve v prologu. Pri tem se zgleduj po podani shemi. Zaradi enostavnosti obravnavanja omejitev, lahko privzamete da imamo opraviti le s pozitivnimi števili. Začetna domena vsake spremenljivke je lahko napr. (0, 10000).

Problem je zahteven, zato zadostuje, tudi če zna tvoj meta-inrepreter reševati le lažje probleme. Ocena seminarske naloge je odvisna od izvirnosti rešitev in kompleksnosti problemov, ki jih lahko rešuje tvoj sistem.

### 3. Interpreter za denotacijsko semantiko

Denotacijska semantika je sintaksno usmerjen način za opisovanje semantike jezika. Jezik podamo z 1.) definicijo sintaktične domene, 2.) def. opisom stanja in semantičnih funkcij in 3.) def. semantične domene.

Na ta način lahko definiramo sintakso in pomen programov nekega jezika. Naloga je implementacija interpretatorja, ki pravilno interpretira poljuben izraz v jeziku, katerega pomen je podan z denotacijsko semantiko.

#### 3.1 Primer zapisa semantike preprostih boolovih izrazov z den. sem.

Sintaksa in semantika preprostih boolovih izrazov: v izrazih lahko nastopajo sprem. x, y in z (ki imajo nedefinirano vrednost true ali false), konstante true in false ter operatorji and, or in not.

Nekaj primerov sintaktično pravilnih izrazov:

z  
x or ( y and not z )  
not x and ( true and ( false or y ) )

##### 1. Sintakticna domena(v BNF notaciji):

```
<exp> ::= <fact> and <exp> | <fact> or <exp> | <fact>  
<fact> ::= true | false | <var> | not <fact> | ( <exp> )  
<var> ::= x | y | z
```

##### 2. Opis stanja in semantičnih funkcij:

```
State = {true, false} x {true, false} x {true, false}  
% stanje opisujejo vrednosti sprem. x, y in z  
Mexp: exp -> (State -> { true, false })  
% izraz preslika trenutno stanje sprem. x, y, in z v vrednost izraza true ali false  
Mfact: fact -> (State -> { true, false })  
MVar: Var -> (State -> { true, false })
```

### 3. Definicija semanticnih funkcij:

$Mexp[e:exp](s:State) \equiv$   
 $\quad <fact> \text{ and } <exp> \Rightarrow Mfact[e.\text{fact}](s) \text{ AND } Mexp[e.\text{exp}](s) |$   
 $\quad <fact> \text{ or } <exp> \Rightarrow Mfact[e.\text{fact}](s) \text{ OR } Mexp[e.\text{exp}](s) |$   
 $\quad <fact> \Rightarrow Mfact[e.\text{fact}](s)$

$Mfact[f:\text{fact}](s:State) \equiv$   
 $\quad \text{true} \Rightarrow \text{true} |$   
 $\quad \text{false} \Rightarrow \text{false} |$   
 $\quad <\text{var}> \Rightarrow Mvar[f.\text{var}](s) |$   
 $\quad \text{not } <\text{fact}> \Rightarrow \text{NOT } Mfact[f.\text{fact}](s) |$   
 $\quad (<\text{exp}>) \Rightarrow Mexp[f.\text{exp}](s)$

$Mvar[v:\text{var}](<xVal, yVal, zVal>:\text{State}) \equiv$   
 $\quad x \Rightarrow xVal |$   
 $\quad y \Rightarrow yVal |$   
 $\quad z \Rightarrow zVal$

Denimo da je vrednost stanja State =  $\langle \text{true}, \text{true}, \text{true} \rangle$ .  
Kakšen je potem pomen izraza:  $x \text{ or } (y \text{ and not } z)$  ?

Torej:

$Mexp[x \text{ or } (y \text{ and not } z)](\langle \text{true}, \text{true}, \text{true} \rangle) = ?$

Resitev:

$Mexp[x \text{ or } (y \text{ and not } z)](\langle \text{true}, \text{true}, \text{true} \rangle) =$   
 $Mfact[x](\langle \text{true}, \text{true}, \text{true} \rangle) \text{ OR } Mexp[(y \text{ and not } z)](\langle \text{true}, \text{true}, \text{true} \rangle) =$   
 $Mvar[x](\langle \text{true}, \text{true}, \text{true} \rangle) \text{ OR } Mexp[(y \text{ and not } z)](\langle \text{true}, \text{true}, \text{true} \rangle) =$   
 $\text{true OR } Mexp[(y \text{ and not } z)](\langle \text{true}, \text{true}, \text{true} \rangle) =$   
 $\text{true OR } Mfact[(y \text{ and not } z)](\langle \text{true}, \text{true}, \text{true} \rangle) =$   
 $\text{true OR } Mexp[y \text{ and not } z](\langle \text{true}, \text{true}, \text{true} \rangle) =$   
 $\text{true OR } (Mfact[y](\langle \text{true}, \text{true}, \text{true} \rangle) \text{ AND } Mexp[\text{not } z](\langle \text{true}, \text{true}, \text{true} \rangle)) =$   
 $\text{true OR } (Mvar[y](\langle \text{true}, \text{true}, \text{true} \rangle) \text{ AND } Mexp[\text{not } z](\langle \text{true}, \text{true}, \text{true} \rangle)) =$   
 $\text{true OR } (\text{true AND } Mfact[\text{not } z](\langle \text{true}, \text{true}, \text{true} \rangle)) =$   
 $\text{true OR } (\text{true AND } (\text{NOT } Mfact[z](\langle \text{true}, \text{true}, \text{true} \rangle))) =$   
 $\text{true OR } (\text{true AND } (\text{NOT } Mvar[z](\langle \text{true}, \text{true}, \text{true} \rangle))) =$   
 $\text{true OR } (\text{true AND } (\text{NOT } \text{true})) =$   
 $\text{true OR } (\text{true AND } \text{false}) =$   
 $\text{true OR } \text{false} =$   
true

## 3.2 Izvedba interpreterja za denotacijsko semantiko v prologu

Naloga je implementacija interpreterja, ki pravilno interpretira poljuben program v opisanem jeziku. Potrebno je:

1. ustrezno definirati operatorje, ki omogočajo def. sinatkse in semantike jezika. Pri tem je dovoljeno uporabiti tak zapis sintakse in semantike jezika, ki olajša samo izvedbo interpreterja.
2. izdelati proceduro  $m(SemFn, Izraz, Stanje, Vrednost)$ , ki interpretira semantično funkcijo  $SemFn$  nad izrazom  $Izraz$  pri začetnem stanju  $Stanje$  ter vrne vrednost  $Vrednost$  izraza.
3. Izdelati nekaj primerov preprostih jezikov, ki jih zna izvajati vaš interpreter, ter pripraviti ustreerne testne primere za vaš interpreter.

V nadaljevanju je s primeri nakazana ena izmed možnih izvedb seminarske naloge.

1. **del: ustrezno definirati operatorje, ki omogočajo def. sinatkse in semantike jezika. Pri tem je dovoljeno uporabiti tak zapis sintakse in semantike jezika, ki olajša samo izvedbo interpreterja.**

Eden izmed možnih načinov zapisa pomena opisanega jezika preprostih bolovih izrazov v prologu (če so prej ustrezno definirani operatorji kot so  $::=$  in  $\Rightarrow$ ) je:

```
% definicija jezika preprostih boolovih izrazov v prologu
:- op(710, xfy, or).
:- op(700, xfy, and).
:- op(690, fy, not).
```

```
% 2. Opis stanja in semanticnih funkcij:
state( [[true, false], [true, false], [true, false]] ).
semFn( exp, state => [ true, false ] ).  
semFn(fact, state => [ true, false ] ).  
semFn(var, state => [ true, false ] ).
```

```
% 3. Definicija semanticnih funkcij:
exp(E:exp, S:state) ::=  
    [fact, and, exp] => fact(E:el(1), S) and exp(E:el(3), S) ;  
    [fact, or, exp] => fact(E:el(1), S) or exp(E:el(3), S) ;  
    [fact] => fact(E:el(1), S).  
fact(F:fact, S:state) ::=  
    [true] => true ;  
    [false] => false ;  
    [var] => var( F:el(1), S ) ;  
    [not, fact] => not fact(F:el(2), S) ;  
    ['(', exp, ')'] => exp(F:el(2), S).
```

```

var(V:var, [XVal, YVal, ZVal]:state) ::= 
  [x] => XVal ;
  [y] => YVal ;
  [z] => ZVal .

meaning(true, true) :- !.
meaning(false, false) :- !.
meaning(not Exp, Val) :-
  meaning(Exp, E),
  ( E = true, Val = false ;
    E = false, Val = true ), !.
meaning(X and Y, Val) :-
  meaning(X, S1),
  meaning(Y, S2), !,
  ( S1 = true, S2 = true, Val=true ;
    Val=false), !.
meaning(X or Y, Val) :-
  meaning(X, S1),
  meaning(Y, S2), !,
  ( (S1 = true ; S2 = true), Val=true ;
    Val=false), !.

meaning(E, M) :-
  meaningDefault(E, M), !.

% meaningDefault definira pomen seznama, pomen stevila in podobno, napr.
% meaningDefault(N, N) :- number(N), !.
% meaningDefault([], []) :- !.
% meaningDefault([X|R], [Xm|Rm]) :- 
%   meaning(X, Xm), meaning(R, Rm), !.

```

Pri podani definiciji jezika zna interpreter odgovoriti na vprašanja kot so:

?- m(exp,[x, or, (, y, and, not, z, ) ],[true,true,true],M).

M = true

?- m(exp,[z, and, y, and, x],[true,false,true],M).

M = false

?- m(exp,[y, and, not, not, x],[true,true,true],M).

M = true

2. del: izdelati proceduro  $m(SemFn, Izraz, Stanje, Vrednost)$ , ki interpretira semantično funkcijo  $SemFn$  nad izrazom  $Izraz$  pri začetnem stanju  $Stanje$  ter vrne vrednost  $Vrednost$  izraza.

Interpreter je možno implementirati na več načinov. Pri samem načinu izvedbe imajo študentje povsem proste roke !

Eden izmed možnih načinov najprej sestavi  $ParseList$  izraza, ki določa na kakšen način gramatika generira dani izraz. V naslednjem koraku  $ParseList razvijemo$ , tako da nadomeščamo vse semantične funkcije z ustreznimi pomeni sem. funkcij. Na koncu še izračunamo pomen razvitega izraza. Tako je lahko predikat m/4 definiran kot:

```
m(SintD, Exp, State, Val) :-  
    p(SintD, Exp, State, ParseList),  
    unfold(SintD, State, ParseList, V0),  
    meaning(V0, Val).
```

Poglejmo kako prolog izvaja posamezne cilje:

```
?- m(exp,[y,and,x],[true,false,true],M).  
M = false  
  
?- p(exp,[y,and,x],[true,false,true],L).  
ParseList = [  
    exp(y,[true,false,true])=> fact(y,[true,false,true])and exp(x,[true,false,true]),  
    fact(y,_A)=>var(y,_A),  
    var(y,[_E,_C,_D])=>_C,  
    exp(x,_B)=>fact(x,_B),  
    fact(x,_F)=>var(x,_F),  
    var(x,[_G,_I,_H])=>_G]  
  
?- unfold(exp,[true,false,true],  
        [exp(y,[true,false,true])=>fact(y,[true,false,true]) and exp(x,[true,false,true]),  
         fact(y,_A)=>var(y,_A), var(y,[_E,_C,_D])=>_C, exp(x,_B)=>fact(x,_B),  
         fact(x,_F)=>var(x,_F), var(x,[_G,_I,_H])=>_G], V0).  
V0 = false and true  
  
?- meaning(false and true, Val).  
Val = false
```

3. del: Izdelati nekaj primerov preprostih jezikov, ki jih zna izvajati vaš interpreter, ter pripraviti ustrezne testne primere za vaš interpreter.

Primer: Definicija dvojiških števil:

```
semFn(bin, _).
semFn(num, _).

bin(B:bin, S:state) ::=
    [num] => num(B:el(1), _) ;
    [bin, num] => num(B:el(2), _) + 2 * bin(B:el(1), S).

num(N:num, _:state) ::=
    [0] => 0 ;
    [1] => 1 .
```

```
meaning(N, N) :- number(N), !.
meaning(2 * Num, Val) :-
    meaning(Num, V1),
    Val is Num * 2, !.
meaning(N1 + N2, Val) :-
    meaning(N1, V1),
    meaning(N2, V2),
    Val is V1 + V2, !.

meaning(E, M) :-
    meaningDefault(E, M), !.
```

Interpreterju lahko zastavljamo vprašanja kot so:

```
?- m(bin, [1,1], [], M).
M = 3
```

```
?- m(bin, [1,0, 0, 0], [], M).
M = 8
```

Primer: Definicija preprostega programskega jezika:

```
: - op(710, xfy, or).  
: - op(700, xfy, and).  
: - op(690, fy, not).
```

```
% 2. Opis stanja in semanticnih funkcij:  
state( [[true, false], [true, false], [true, false]] ).  
semFn( prog, state => state).  
semFn( stmts, state => state).  
semFn( stmt, state => state).  
semFn( lvar, state => [1, 2, 3]).  
semFn( exp, state => [ true, false ]).  
semFn(fact, state => [ true, false ]).  
semFn(var, state => [ true, false ]).
```

```
prog(P:prog, S:state) ::=  
[program, stmts, end] => stmts(P:el(2),S).  
stmts(Sts:stmts, S:state) ::=  
[stmt, ;, stmts] => stmts( Sts:el(3), stmt(Sts:el(1), S) ) ;  
[stmt] => stmt(Sts:el(1), S).  
stmt(Sts:stmt, [V1, V2, V3]:state) ::=  
[lvar, ':', exp] =>  
case lvar(Sts:el(1), _) of (  
1: [exp(Sts:el(3), [V1, V2, V3]), V2, V3] ;  
2: [V1, exp(Sts:el(3), [V1, V2, V3]), V3] ;  
3: [V1, V2, exp(Sts:el(3), [V1, V2, V3])] ) ;  
[if, exp, then, stmt] =>  
case exp(Sts:el(2), [V1, V2, V3]) of (  
true: stmt(Sts:el(4), [V1, V2, V3]) ;  
false: [V1, V2, V3] ).  
lvar(V:lvar, _:state) ::=  
[x] => 1 ;  
[y] => 2 ;  
[z] => 3.
```

```
exp(E:exp, S:state) ::=  
[fact, and, exp] => fact(E:el(1), S) and exp(E:el(3), S) ;  
[fact, or, exp] => fact(E:el(1), S) or exp(E:el(3), S) ;  
[fact] => fact(E:el(1), S).  
fact(F:fact, S:state) ::=  
[true] => true ;  
[false] => false ;  
[var] => var( F:el(1), S) ;  
[not, fact] => not fact(F:el(2), S) ;  
['(', exp, ')'] => exp(F:el(2), S).  
var(V:var, [XVal, YVal, ZVal]:state) ::=  
[x] => XVal ; [y] => YVal ;  
[z] => ZVal .
```

```

meaning(true, true) :- !.
meaning(false, false) :- !.
meaning(not Exp, Val) :- 
    meaning(Exp, E),
    ( E = true, Val = false ;
      E = false, Val = true ), !.
meaning(X and Y, Val) :- 
    meaning(X, S1),
    meaning(Y, S2), !,
    ( S1 = true, S2 = true, Val=true ;
      Val=false), !.
meaning(X or Y, Val) :- 
    meaning(X, S1),
    meaning(Y, S2), !,
    ( (S1 = true ; S2 = true), Val=true ;
      Val=false), !.

meaning(E, M) :- 
    meaningDefault(E, M), !.

```

Interpreterju lahko zastavljam vprašanja kot so:

```
?- m(stmts, [x, ':='], false, ;, y, ':='], x], [true,true,true], M).
M = [false,false,true]
```

```
?- m(prog, [program, y, ':='], false, ';', x, ':='], false, end], [true,true,true], M).
M = [false,false,true]
```

```
?- m(prog, [program, x, ':='], false, ;, y, ':='], x, ;, z, ':='], x, or, not, y, end],
   [true,true,false], M).
M = [false,false,true]
```

```
?- m(prog, [program, x, ':='], false, ;, if, x, then, y, ':='], false, ;,
     z, ':='], x, or, not, y, end], [true,true,true], M).
M = [false,true,false]
```

```
?- m(prog, [program, x, ':='], true, ;, if, x, then, y, ':='], false, ;,
     z, ':='], x, or, not, y, end], [true,true,true], M).
M = [true,false,true]
```

## Dodatek

Napiši predikat  $p(SintD, Exp, State, ParseList)$ , ki poišče  $ParseList$ .  $ParseList$  je seznam uporabljenih pravil, ki določajo, kako podana gramatika generira izraz  $Exp$ , pri čemer je  $SintD$  začetno pravilo,  $State$  pa določa začetno stanje.

**Primer: definicija sem. funkcij za preproste bool. izraze kot dejstva v prologu:**

```
% definicija bool. operatorjev  
:- op(700, xfy, and).  
:- op(690, fy, not).
```

```
exp(E:exp, S:state) ::=  
    [fact, and, exp] => fact(E:el(1), S) and exp(E:el(3), S) ;  
    [fact] => fact(E:el(1), S).  
fact(F:fact, S:state) ::=  
    [true] => true ;  
    [false] => false ;  
    [var] => var( F:el(1), S) ;  
    [not, fact] => not fact(F:el(2), S) ;  
    [', exp, ')'] => exp(F:el(2), S).  
var(V:var, [XVal, YVal, ZVal]:state) ::=  
    [x] => XVal ;  
    [y] => YVal ;  
    [z] => ZVal .
```

**Pri podani definiciji sem. fn. in ustrezem programu za p/4 prolog odgovori:**

```
?- p(exp,[y, and,x],[true, false, true],L).  
L = [exp([y, and,x], [true, false, true])  
     => fact([y, and,x], [true, false, true]) and exp([x], [true, false, true]),  
     fact([y, and,x], _A) => var([y, and,x], _A),  
     var([y, and,x], [_E, _C, _D]) => _C,  
     exp([x], _B) => fact([x], _B),  
     fact([x], _F) => var([x], _F),  
     var([x], [_G, _I, _H]) => _G]
```

**1. Najprej definiramo ustrezne operatorje za zapis denotacijske semantike:**

```
:-op(1110, xfx, ::= ).  
:-op(850, xfx, => ).  
:-op(800, xfx, :: ).  
:-op(750, xfx, of).  
:-op(740, fx, case).
```

**2. Napiši predikat  $sintMeaning(SintD, Rule, SintVar, State)$ , ki skozi avtomatsko vračanje vrne vsa pravila  $Rule$ , ki ustrezajo sem. funkciji spremenljivke  $SintD$ . V  $SintVar$  in  $State$  vrne prvi in drugi argument sem.funkcije  $SintD$ .**

**Pri podani sem. funkciji, lahko zastavimo vprašanje:**

? - *sintMeaning(exp, Rule, SintVar, State)*.

Rule = [fact, and, exp] => fact(E:el(1), S) and exp(E:el(3), S), SintVar = E, State = S ;  
Rule = [fact] => fact(E:el(1), S), SintVar = E, State = S

**Ideja:** *sintMeaning* poišče ustrezeno semantično funkcijo

*SintD(SintVar:SintD, State) ::= SintBody*

in vrne eno izmed pravil iz *SintBody* (v *SintBody* je lahko več pravil ločenih s ;)

*sintMeaning(SintD, SintClause => SemFn, SintVar, State) :-*

% *SintD(SintVar:SintD, State) ::= SintBody*

(SHead ::= *SintBody*),

SHead =.. [SintD, SintVar:SintD, State],

bodyClause(*SintBody*, SintClause, SemFn).

*bodyClause(*SintBody* => Meaning, SintBody, Meaning).*

*bodyClause(( SintBody => Meaning ; \_ ), SintBody, Meaning).*

*bodyClause( \_ ; SintBodyRest), SintBody, Meaning) :-*

*bodyClause( SintBodyRest, SintBody, Meaning).*

### 3. Napiši predikat *p(SintD, Exp, State, ParseList)*, ki poišče ustrezeni *ParseList*.

**Ideja:** na roko poskušamo poiskati *ParseList* za izraz [y, and x]:

Uporabimo prvo pravilo

*exp(E:exp) => fact(E:el(1)) and exp(E:el(3))*

in ga damo v *ParseList*, sedaj pa moramo generirati še *ParseList* za obe spremenljivki (fact in exp) - torej moramo vzdrževati seznam spremenljivk, ki jih moramo še razviti v končne simbole z uporabo sem. funkcij. Na začetku je v seznamu le začetno pravilo, zato:

*p(SintD, Exp, State, ParseList) :-*

*p2([SintD], Exp, State, ParseList).*

Ko pridemo do sem. funkcije, ki generira naslednji končni simbol, pravilo uporabimo, končni simbol pa odstranimo iz seznama preostalih terminalov. Robni pogoj je, ko je seznam terminalov prazen.

*p2([], [], State, []). % robni pogoj*

% redukcija terminala

*p2([T|Terms], [T|Rexp], State, ParseList) :-*

*p2( Terms, Rexp, State, ParseList), !.*

Sicer moramo uporabiti pravilo za prvo sprem., tako da sprem. nadomestimo z desnim delom ustreznega pravila *SemFn0*. V *SemFn0* moramo še nadomestiti E:el(3) – to naredi *replaceSelectors(SemFn0, SemFn)*:

?- *replaceSelectors( fact( E:el(1), S) and exp( E:el(3), S ), SemFn).*

E = [ \_A, \_B, \_C|\_Rest ],

SemFn = (fact( \_A, S) and exp( \_C, S))

Sprogramirati moramo ustrezni **replaceSelectors/2** in lahko napišemo:

```
% neterminal nadomestim z desno stranjo pripadajocega pravila
p2([Sint|Rsint], Terms, State, [SemHead => SemFn|PL]) :-  
    SemHead=..[Sint, Terms, State],  
    sintMeaning(Sint, SintBody => SemFn0, SintBodyVar, State:state ),  
    replaceSelectors(SemFn0, SemFn),  
    varList(SintBody, SintBodyVar),  
    conc(SintBody, Rsint, NewSint),  
    p2(NewSint, Terms, NewState, PL).
```

```
% ?-varList([fact, and, exp], L)  
% L=[_1, _2, _3]  
varList([], []).  
varList([_|R], [_|VarListRest]):-  
    varList(R, VarListRest).
```

Nas predikat še ne dela povsem pravilno, ampak tako:

```
?- p2([exp],[y, and, x],[true, false, true],L).  
L = [exp([y, and, x],[true, false, true]) =>  
     fact(_B,[true, false, true])and exp(_C,[true, false, true]),  
     fact([y, and, x],_A)=>var(_D,_A),  
     var([y, and, x],[_H,_F,_G])=>_F,  
     exp([x],_E)=>fact(_J,_E),  
     fact([x],_I)=>var(_K,_I),  
     var([x],[_L,_N,_M])=>_L]
```

Kako popravimo problem ? Uvedemo dodatno sprem. v p/2....