Declarative Programming

Logic Programming and Prolog

Yann-Michaël De Hauwere
ydehauwe@vub.ac.be
Procedural and declarative meaning

\[ a :- b, c. \]

declarative meaning realized by model semantics

to determine whether \( a \) is a logical consequence of the clause, order of atoms in body is irrelevant

procedural meaning realized by proof theory

order of atoms may determine whether \( a \) can be derived

\[ a :- b, c. \]

to prove \( a \), prove \( b \) and then prove \( c \)

\[ a :- c, b. \]

to prove \( a \), prove \( c \) and then prove \( b \)
Procedural meaning enables programming

algorithm = logic + control
Logic Programming

SLD Resolution refutation

```
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(a,b).
mother(b,c).
```

```
:- grandfather(a,X),           goal (query)
    grandfather(C,D) :- father(C,E), parent(E,D).
    {C/a,D/X}
:- father(a,E), parent(E,X),   derived goal
    father(a,b).
    {E/b}
:- parent(b,X),                computed substitution
    parent(U,V) :- mother(U,V).
    {U/b,V/X}
:- mother(b,X).
    mother(b,c).
    {X/c}

{X/c,C/a,D/c,E/b,U/b,V/c}:

computation answer substitution
```
Logic Programming

SLD Trees

\[
\begin{align*}
\text{grandfather}(X,Z) & :\text{ father}(X,Y), \text{ parent}(Y,Z). \\
\text{parent}(X,Y) & :\text{ father}(X,Y). \\
\text{parent}(X,Y) & :\text{ mother}(X,Y). \\
\text{father}(a,b). \\
\text{mother}(b,c). \\
\end{align*}
\]

Every leaf corresponds to a successful refutation (a success branch). A blocked leaf corresponds to a failed branch.

Prolog does a depth-first traversal of an SLD tree.

What if an SLD tree has infinite branches?
Prolog loops on this query; renders it **incomplete**!
Because of depth-first traversal and not because of resolution as all answers are represented by success branches in the SLD-tree
Logic Programming

Infinite SLD Trees

\[
\text{sibling}(a,b).
\text{sibling}(b,c).
\text{sibling}(X,Y) :- \text{sibling}(X,Z), \text{sibling}(Z,Y).
\]

\[
\begin{align*}
\text{:-sibling}(a,X) \\
\text{:-sibling}(a,Z), \text{sibling}(Z,Y) \\
\text{:-sibling}(b,Y) & \quad \text{:-sibling}(a,U), \text{sibling}(U,Z), \text{sibling}(Z,Y) \\
\text{:-sibling}(a,Z), \text{sibling}(Z,Y) & \quad \cdots \\
\cdots
\end{align*}
\]
Infinite SLD Trees with lists

Logic Programming

?-list(L).
L = [];
L = [A];
L = [A, B];
...

list([]).
list([H|T]) :- list(T).

? :- list(L)

?-list(L)

L = []

[] :- list(T1)

L = [A]

[] :- list(T2)

L = [A, B]

[] :- list(T3)
Infinite subtrees with lists

Logic Programming

\[
\text{plist}([], []). \\
\text{plist}([H|T]) :- \text{p}(H), \text{plist}(T). \\
p(1). \\
p(2). \\
\]

?- \text{plist}(L).
L = []; \\
L = [1]; \\
L = [1,1]; \\

\[
\text{:- p}(H1), \text{plist}(T1) \\
\text{:- \text{plist}}(T1) \\
\text{:- p}(H1), \text{plist}(T1) \\
\text{:- \text{plist}}(T1) \\
\text{:- \text{plist}}(T1) \\
\text{:- \text{plist}}(T1) \\
\text{:- \text{plist}}(T1) \\
\text{:- \text{plist}}(T1) \\
\text{:- \text{plist}}(T1) \\
\text{:- \text{plist}}(T1) \\
\]

\[
\text{L} = [2] \\
\text{L} = [1,2] \\
\text{L} = [2,1] \\
\text{L} = [2,2] \\
\]

\[
\text{L} = [1,1] \\
\text{L} = [1] \\
\text{L} = [] \\
\text{...}
\]
SLD Resolution refutation as a proof procedure

SLD:
• Selection of literal
• Linear Resolution
• Definite CLauses
Backtracking

• When a failure branch is reached (non-empty resolvent which cannot be reduced further), next alternative for the last-chosen program clause has to be tried

• Amounts to going up one level in SLD-tree and descending into the next branch to the right (backtracking = popping resolvent from stack and exploring next alternative)

• Requires remembering previous resolvents for which not all alternatives have been explored together with the last program clause that has been explored at that point
“Once you’ve reached me, stick with all variable substitutions you’ve found after you entered my clause”

Prolog won’t try alternatives for:

- literals left to the cut
- the clause in which the cut is found
Logic Programming

Cut (example)

p(X,Y) :- q(X,Y).
p(X,Y) :- r(X,Y).
q(X,Y) :- s(X), !, t(Y).
r(c,d).
s(a).
s(b).
t(a).
t(b).

?-p(X,Y)

:-q(X,Y)

:-r(X,Y)

:-s(X), !, t(Y)

:-t(Y)

:-t(Y)

:-t(Y)

[]

[]
Different kinds of cut

- does not prune away success branches
- stresses that the conjuncts to its left are deterministic and therefore do not have alternative solutions
- and that the clauses below with the same head won’t result in alternative solutions either

- prunes success branches
- some logical consequences of the program are not returned
- has the declarative and procedural meaning of the program diverge
Logic Programming

Different kinds of cut - green or red?

parent(X,Y):-father(X,Y),!.
parent(X,Y):-mother(X,Y).
father(john,paul).
father(john,peter).
mother(mary,paul).
mother(mary,peter).

?-parent(john,C) !.
:-father(john,C),!.
:-mother(john,C) !.

same query, but John has multiple children in this program
Different kinds of cut - green or red?

```prolog
parent(X,Y):-father(X,Y),!.
parent(X,Y):-mother(X,Y).
father(john,paul).
mother(mary,paul).

?-parent(P,paul)
   :-father(P,paul),!.
   :-mother(P,paul).

{P/mary}

same program, but query quantifies over parents rather than children
```
Logic Programming

Placement of cut

```
likes(peter,Y):-friendly(Y).
likes(T,S):-student_of(S,T).
student_of(maria,peter).
student_of(paul,peter).
friendly(maria).

?-likes(A,B)

[]
A=peter
B=maria

?-likes(A,B)

[]
A=peter
B=maria

!:,-!,friendly(B)
:=-student_of(B,A)

[]
A=peter
B=maria
B=paul

!:,-!
:=-student_of(B,A),!
:=-friendly(B)
:=-student_of(B,A),!

[]
A=peter
B=maria
B=paul

likes(peter,Y):-!,friendly(Y).
likes(T,S):-student_of(S,T),!.
```
Logic Programming

Other dangers with cut

```
max(M,N,M) :- M>=N.
max(M,N,N) :- M=<N.
```

clauses are not mutually exclusive
two ways to solve query `?-max(3,3,5)`

```
max(M,N,M) :- M>=N, !.
max(M,N,N).
```
could use red cut to prune second way

```
max(M,N,M) :- M>=N, !.
max(M,N,N).
```

problem:

```
?-max(5,3,3)
succeeds
```
Logic Programming

Cutting choice points

parent(X,Y):-father(X,Y).
parent(X,Y):-mother(X,Y).
father(john,paul).
mother(mary,paul).

?-parent(john,C)
  !:-mother(john,C)
  !:-father(john,C)

Cut is often used to ensure clauses are mutually exclusive
Logic Programming

Negation as failure

\[ p :\!-\: q, r. \]
\[ p :\!-\: s. \]

such uses are equivalent to the higher-level

\[ p :\!-\: q, r. \]
\[ p :\!-\: \text{not}_q, s. \]

where

\[ \text{not}_q :\!-\: q, !, \text{fail}. \]
\[ \text{not}_q :\!-\: \text{not}_q. \]

Prolog’s not/1 meta-predicate captures this idea:

\[ \text{not}(\text{Goal}) :\!-\: \text{Goal}, !, \text{fail}. \]
\[ \text{not}(\text{Goal}). \]
Logic Programming

Negation as failure (SLD tree)

not(q) succeeds because q fails

\begin{verbatim}
p:-q,r.
p:-not(q),s.
s.
not(Goal):-Goal,! fail.
not(Goal).
\end{verbatim}
Negation as failure (SLD tree)

p :- not(q), r.
p :- q.
q.
r.
not(Goal) :- Goal, !, fail.
not(Goal).

?- p
?- not(q), r
?- q,
?- fail, r
?- !, fail, r
?- fail, r
bachelor(X):-not(married(X)),man(X).
man(fred).
man(peter).
mARRIED(fred).

?-bachelor(X)

:-not(married(X)),man(X)

:-married(X),!,fail,man(X)

:-!,fail,man(fred)

:-fail,man(fred)

not(Goal):-Goal,!,fail.
not(Goal).
Logic Programming

Avoid floundering

correct implementation of SLDNF-resolution:
not(Goal) fails only if Goal has a refutation with an **empty** answer substitution

work-around: if Goal is ground, only empty answer substitutions are possible

```prolog
bachelor(X):- man(X),
            not(married(X)).
man(fred).
man(peter).
married(fred).
```
Logic Programming

Avoid floundering

?-bachelor(X)

:-man(X),not(married(X))

:-not(married(fred))

:-married(fred),!,fail []

:-married(peter),!,fail []

:-!,fail

:-fail

bachelor(X):- man(X),not(married(X)).
mixed(fred).
man(peter).
mixed(fred).
Logic Programming

if-then-else

such uses are equivalent to

```prolog
p:-q,r,s,!,t.
p:-q,r,u.
q.
r.
u.
```

```prolog
p:-q,r,if_s_then_t_else_u.
if_s_then_t_else_u:-s,!,t.
if_s_then_t_else_u:-u.
q.
r.
u.
```
Logic Programming

if-then-else built-in

\[
\begin{align*}
p &: - q, r, s, !, t. \\
p &: - q, r, u. \\
q &. \\
r &. \\
u &.
\end{align*}
\]

\[
\begin{align*}
p &: - q, r, \text{if}_\text{then}_\text{else}(S,T,U). \\
\text{if}_\text{then}_\text{else}(S,T,U) &: - S, !, T. \\
\text{if}_\text{then}_\text{else}(S,T,U) &: - U.
\end{align*}
\]

built-in as \( P \rightarrow Q; R \)

\[
\begin{align*}
\text{diagnosis}(\text{Patient}, \text{Condition}) &: - \\
& \text{temperature}(\text{Patient}, T), \\
& ( T \leq 37 \rightarrow \text{blood}_\text{pressure}(\text{Patient}, \text{Condition}) ) \\
& ; T > 37, T < 38 \rightarrow \text{Condition} = \text{ok} \\
& ; \text{otherwise} \rightarrow \text{diagnose}_\text{fever}(\text{Patient}, \text{Condition})
\end{align*}
\]
Tail recursion

```
play(Board,Player):-
    lost(Board,Player).
play(Board,Player):-
    find_move(Board,Player,Move),
    make_move(Board,Move,NewBoard),
    next_player(Player,Next),!,
    play(NewBoard,Next).

:-play(starconfiguration,first).
```

Cut ensures that no previous moves are reconsidered and optimizes tail recursion to iteration.
Arithmetic in prolog

?-X is 5+7-3.
X = 9

?-X is 5*3+7/2.
X = 18.5

?-9 is 5+7-3.
Yes

?-9 is X+7-3.
Error in arithmetic expression

is(Result,Expression) is true if Expression can be evaluated as an arithmetic expression and its resulting value unifies with Result
The last example illustrates that Prolog does not implement the occur check.

Prolog also has other built-in arithmetic predicates: <,>,=<,>=. 
\=/2 succeeds if its arguments are not unifiable.
Accumulators

length([],0).
length([H|T],N) :- length(T,N1), N is N1+1.

?- length([a,b,c],N)
length([H|T],N1):-length(T,M1), N1 is M1+1
   {H->a, T->[b,c], N1->N}

:-length([b,c],M1), N is M1+1
   length([H|T],N2):-length(T,M2), N2 is M2+1
      {H->b, T->[c], N2->M1}

:-length([c],M2), M1 is M2+1, N is M1+1
   length([H|T],N3):-length(T,M3), N3 is M3+1
      {H->c, T->[], N3->M2}

:-length([],M3), M2 is M3+1, M1 is M2+1, N is M1+1
   length([],0)
      {M3->0}

:-M2 is 0+1, M1 is M2+1, N is M1+1
   {M2->1}

:-M1 is 1+1, N is M1+1
   {M1->2}

:-N is 2+1
   {N->3}
[]
Logic Programming

Accumulators

```prolog
length(L,N) :- length_acc(L,0,N).
length_acc([],N,N).
length_acc([H|T],N0,N) :- N1 is N0+1, length_acc(T,N1,N).
```

?- length_acc([a,b,c],0,N).

:-length_acc([a,b,c],0,N0) :- N1 is N0+1,
  length_acc([b,c],N11,N1)
  :-N11 is 0+1,
  length_acc([b,c],N11,N1)
  {H->a, T->[b,c], N10->0, N1->N}
  :-N11 is 0+1,
  length_acc([b,c],N11,N1)
  :-length_acc([b,c],1,N) :-N21 is N20+1,
  length_acc([c],N21,N)
  :-N21 is 1+1,
  length_acc([c],N21,N)
  {N21->2}
  :-length_acc([c],2,N) :-N31 is N30+1,
  length_acc([],N31,N)
  :-N31 is 2+1,
  length_acc([],N31,N)
  {N31->3}
  :-length_acc([],3,N) length_acc([],N,N)
  {N->3}
```

[32]
Reverse with accumulators

naive_reverse([], []).
naive_reverse([H|T], R) :- naive_reverse(T, R1), append(R1, [H], R).

append([], Y, Y).
append([H|T], Y, [H|Z]) :- append(T, Y, Z).

reverse(X, Z) :- reverse(X, [], Z).
reverse([], Z, Z).
reverse([H|T], Z, Z) :- reverse(T, [H|Y], Z).
reverse([H|T], Y, Z) :- reverse(T, [H|Y], Z).
Logic Programming

**Difference lists**

![Diagram showing difference between two lists](image)

**represent a list by a term L1-L2.**

- \([a, b, c, d] - [d]\) \[a, b, c]\n- \([a, b, c, 1, 2] - [1, 2]\) \[a, b, c]\n- \([a, b, c|X] - X\) \[a, b, c]\n
---

34
Appending difference lists

append_dl(XPlus-XMinus,YPlus-YMinus,XPlus-YMinus) :- XMinus=YPlus.

or

append_dl(XPlus-YPlus,YPlus-YMinus,XPlus-YMinus).

?-append_dl([a,b|X]-X,[c,d|Y]-Y,Z).
X = [c,d|Y], Z = [a,b,c,d|Y]-Y
Second order predicates: map

map(R, [], []).  
map(R, [X|Xs], [Y|Ys]):- R(X, Y), map(R, Xs, Ys).
?- map(parent, [a, b, c], X)  

or, when atoms with variable as predicate symbol are not allowed:

map(R, [], []).  
map(R, [X|Xs], [Y|Ys]):- Goal =.. [R, X, Y], 
    call(Goal), 
    map(R, Xs, Ys).

univ operator =.. can be used to construct terms:  
?- Term=..[parent,X,peter]
   Term=parent(X,peter)

and decompose terms:  
?- parent(maria, Y)=..List
   List=[parent, maria, Y]

Term=..List succeeds  
if Term is a constant and List is the list [Term]  
if Term is a compound term f(A1,..,An) 
   and List is a list with head f and whose tail unifies with [A1,..,An]
findall(Template, Goal, List) succeeds if List unifies with a list of the terms Template is instantiated to successively on backtracking over Goal. If Goal has no solutions, List has to unify with the empty list.

?-findall(C, parent(john, C), L).
L = [peter, paul, mary]

?-findall(f(C), parent(john, C), L).
L = [f(peter), f(paul), f(mary)]

?-findall(C, parent(P, C), L).
L = [peter, paul, mary, davy, dee, dozy]
Logic Programming

Second order predicates: bagof & setof

The construct $\text{Var}^\text{Goal}$ tells bagof/3 not to bind $\text{Var}$ in $\text{Goal}$.

findall/3 is same as bagof/3 with all free variables existentially quantified using ^

setof/3 is same as bagof/3 without duplicate elements in List
asserta(Clause)
    adds Clause at the beginning of the Prolog database.
assertz(Clause) and assert(Clause)
    adds Clause at the end of the Prolog database.
retract(Clause)
    removes first clause that unifies with Clause from the Prolog database.

retractall(Term):-
    retract(Term), fail.
retractall(Term):-
    retract([(Term:- Body)), fail.
retractall(Term).
Logic Programming

Second order predicates: assert & retract

Powerful: enable run-time program modification
Harmful: code hard to understand and debug, often slow

Sometimes used as global variables, “boolean” flags or to cache:

\[
\begin{align*}
\text{fib}(0,0). \\
\text{fib}(1,1). \\
\text{fib}(N,F) :&- \\
&\quad N > 1, \\
&\quad N1 \text{ is } N-1, \\
&\quad N2 \text{ is } N1-1, \\
&\quad \text{fib}(N1,F1), \\
&\quad \text{fib}(N2,F2), \\
&\quad F \text{ is } F1+F2.
\end{align*}
\]

\[
\begin{align*}
\text{mfib}(N, F) :&- \text{memo_fib}(N, F), !. \\
\text{mfib}(N, F) :&- \\
&\quad N > 1, \\
&\quad N1 \text{ is } N-1, \\
&\quad N2 \text{ is } N1-1, \\
&\quad \text{mfib}(N1,F1), \\
&\quad \text{mfib}(N2,F2), \\
&\quad F \text{ is } F1+F2, \\
&\quad \text{assert(memo_fib}(N, F)).
\end{align*}
\]

:- dynamic memo_fib/2.
\[
\begin{align*}
\text{memo_fib}(0,0). \\
\text{memo_fib}(1,1).
\end{align*}
\]
Higher order programming using call

a more flexible form of call/1, which takes additional arguments that will be added to the Goal that is called

<table>
<thead>
<tr>
<th>call(p(X1,X2,X3))</th>
<th>call(p(X1,X2), X3)</th>
<th>call(p(X1), X2, X3)</th>
<th>call(p, X1, X2, X3)</th>
</tr>
</thead>
</table>

all equal
Higher order programming using call implementing map and filter

map(F,[],[]).  
map(F, [A₀|As₀], [A₁|As]) :-  
    call(F, A₀, A₁),  
    map(F, As₀, As).  

filter(P,[],[]).  
filter(P, [A₀|As₀], As) :-  
    (call(P, A₀) ->  
    As = [A₀|As₁]  
    ;As = As₁),  
    filter(P, As₀, As₁)
Logic Programming

Higher order programming using call using map and filter

?- filter(>(5),[3,4,5,6,7],As).
As=[3,4]

?- map(plus(1),[2,3,4],As).
As=[3,4,5]

?- map(between(1),[2,3],As).
As=[1,1]; As=[1,2]; As=[1,3];
As=[2,1]; As=[2,2]; As=[2,3]

?- map(plus(1),As,[3,4,5]).
As=[2,3,4]

?- map(plus(X),[2,3,4],[3,4,5]).
X=1

?- map(plus(X),[2,A,4],[3,4,B]).
X=1,A=3,B=5
Logic Programming

**Var**

\[ \text{var}(\text{Term}) \]

succeeds when Term is an uninstantiated variable

\[ \text{nonvar}(\text{Term}) \]

has opposite behavior

?- var(X).
true.
?- X=3,var(X).
false.

\[ \text{plus}(X,Y,Z) :- \text{nonvar}(X),\text{nonvar}(Y), Z \text{ is } X+Y. \]
\[ \text{plus}(X,Y,Z) :- \text{nonvar}(X),\text{nonvar}(Z), Y \text{ is } Z-X. \]
\[ \text{plus}(X,Y,Z) :- \text{nonvar}(Y),\text{nonvar}(Z), X \text{ is } Z-Y. \]

\[ \text{grandparent}(X,Z) :- \text{nonvar}(X),\text{parent}(X,Y),\text{parent}(Y,Z). \]
\[ \text{grandparent}(X,Z) :- \text{nonvar}(Z),\text{parent}(Y,Z),\text{parent}(X,Y). \]
Logic Programming

Var

?- var(X).
true.
?- X=3,var(X).
false.

len1([],0).
len1([_|L],N) :-
    len1(L,N1),
    N is N1 + 1.

len2([],0).
len2([_|L],N) :-
    N > 0,
    N1 is N - 1,
    len2(L, N1).

len(L,N) :-
    ( var(N) ->
    len1(L,N)
    ; len2(L,N)
    ).
Logic Programming

Args & Functor

arg(N, Term, Arg)
succeeds when Arg is the Nth argument of Term

functor(Term, F, N)
succeeds when the Term starts with the functor F of arity N

ground(Term) :-
    nonvar(Term), constant(Term).
ground(Term) :-
    nonvar(Term),
    compound(Term),
    functor(Term, F, N),
    ground(N, Term).
ground(N, Term) :-
    N > 0,
    arg(N, Term, Arg),
    ground(Arg),
    Nl is N - 1,
    ground(Nl, Term).
ground(0, Term).
Logic Programming

Operators

:- op(500,xfx,'has_color').
a has_color red.
b has_color blue.

?- b has_color C.
C = blue.

?- What has_color red.
What = a

:- op(Precedence, Type, Name)

integer between 1 and 1200;
smaller integer binds stronger
a+b/c ≡ a+(b/c) ≡ +(a,/(b,c)) if / smaller than +

prefix:   fx, fy
infix:    xfx, xfy, yfx
postfix:  xf, yf

associative    not    right    left
xfx             xfy       yfx

X op Y op Z / op(X,op(Y,Z)) op(op(X,Y),Z)
:- op(900,xfx,to).
hanoi(0,A,B,C,[]).
hanoi(N,A,B,C,Moves):-
    N1 is N-1,
    hanoi(N1,A,C,B,Moves1),
    hanoi(N1,B,A,C,Moves2),
    append(Moves1,[A to C|Moves2],Moves).

?- hanoi(3,left,middle,right,M)
M = [left to right,  
    left to middle,  
    right to middle,  
    left to right,  
    middle to left,  
    middle to right,  
    left to right ]
Logic Programming

Built-in Operators

1200 xfy -->, :-
1200 fx :-, ?-
1150 fx dynamic, discontiguous, initialization, meta_predicate, module_transparent, multifile, thread_local, volatile
1100 xfy ;, |
1050 xfy -->, op*-->
1000 xfy ,
900 fy \ +
900 fx -
700 xfy <, =, ==, =@=, =:=, ==<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \=, is
600 xfy :
500 yfx +, -, \/, \/, xor
500 fx ?
400 yfx *, /, //, rdiv, <<, >>, mod, rem
200 xfy **
200 xfy ^
200 fy +, -, \,

+(a, /'(b, c))
a+b/c

is(X, mod(34, 7))
X is 34 mod 7

<(’+’(3,4),8)
3+4<8

’=’(X,f(Y))
X=f(Y)

’-’(3)
-3

’:-’(p(X),q(Y))
p(X) :- q(Y)

’:-’(p(X),’,’(q(Y),r(Z)))
p(X) :- q(Y), r(Z)
Extending prolog

term_expansion(+In,-Out)

useful for generation code, e.g.:

given compound term representation of data

\[\text{student}(\text{Name}, \text{Id})\]

want to use accessor predicates

\[
\begin{align*}
\text{student_name}(\text{student}(\text{Name}, _), \text{Name}). \\
\text{student_id}(\text{student}(_, \text{Id}), \text{Id}).
\end{align*}
\]

instead of explicit unifications throughout the code

\[\text{Student} = \text{student}(\text{Name}, _)\]

to ensure independence of one particular representation of the data
Extending prolog

:- struct student(name,id).

:- op(1150, fx, (struct)).

| student_name(student(Name, _), Name). |
| student_id(student(_, Id), Id). |

declares struct as a prefix operator

:- op(1150, fx, (struct)).

term_expansion((:- struct Term), Clauses) :-
    functor(Term, Name, Arity),
    functor(Template, Name, Arity),
    gen_clauses(Arity, Name, Term, Template, Clauses).
gen_clauses(N, Name, Term, Template, Clauses) :-
(N =:= 0 -> Clauses = [] ;
arg(N, Term, Argname),
arg(N, Template, Arg),
atom_codes(Argname, Argcodes),
atom_codes(Name, Namecodes),
append(Namecodes, [0'|_|Argcodes], Codes),
atom_codes(Pred, Codes),
Clause =.. [Pred, Template, Arg],
Clauses = [Clause|Clauses1],
N1 is N - 1,
gen_clauses(N1, Name, Term, Template, Clauses1).

?-_X=0'_.
X = 95.
?-_char_code(X,95).
X = '_.'

Logic Programming
Extending prolog

N-th argument recursed upon

trick to merge recursive and base clause

conversion from atom to list of character codes

When trying out, put gen_clauses/5 before term_expansion/2

creates fact
Meta interpreter

prove(Goal):-
    clause(Goal, Body),
    prove(Body).

prove((Goal1, Goal2)):-
    prove(Goal1),
    prove(Goal2).

prove(true):- !.

prove((A, B)): - !,
    prove(A),
    prove(B).

prove(not(Goal)): - !,
    not(prove(Goal)).

prove(A): -
    % not (A=true; A=(X,Y); A=not(G))
    clause(A, B),
    prove(B).

clause(:Head, ?Body)

True if Head can be unified with a clause head and Body with the corresponding clause body. Gives alternative clauses on backtracking. For facts Body is unified with the atom true.
meta-level vs object-level in meta-interpreter

**META-LEVEL**

<table>
<thead>
<tr>
<th>KNOWLEDGE</th>
<th>REASONING</th>
</tr>
</thead>
<tbody>
<tr>
<td>clause(p(X),q(X)).</td>
<td>?-prove(p(X)).</td>
</tr>
<tr>
<td>clause(q(a),true).</td>
<td>X=a</td>
</tr>
</tbody>
</table>

**OBJECT-LEVEL**

| p(X):-q(X).                  | ?-p(X).                       |
|                               | X=a                           |
| q(a).                        |                               |

**Reified** unification explicit at meta-level:

prove(A):- 
  clause(Head,Body), 
  unify(A,Head, MGU, Result), 
  apply(Body, MGU, NewBody), 
  prove_var(NewBody).

Canonical meta-interpreter still absorbs backtracking, unification and variable environments implicitly from the object-level.
Logic Programming

Example: partition

1. Write down declarative specification

```prolog
% partition(L,N,Littles,Bigs) <- Littles contains numbers
% in L smaller than N,
% Bigs contains the rest

partition([],N,[],[]).
partition([Head|Tail],N,?Littles,?Bigs):-
  /* do something with Head */
  partition(Tail,N,Littles,Bigs).
```

2. Identify recursion and “output” arguments

what is the recursion argument?
what is the base case?

3. Write down implementation skeleton

```prolog
partition([],N,[],[]).
partition([Head|Tail],N,?Littles,?Bigs):-
  /* do something with Head */
  partition(Tail,N,Littles,Bigs).
```
Example: partition/4 (2)

4 Complete bodies of clauses

```
partition([],N,[],[]).
partition([Head|Tail],N,?Littles,?Bigs):-
  Head < N,
  partition(Tail,N,Littles,Bigs),
  ?Littles = [Head|Littles],?Bigs = Bigs.
partition([Head|Tail],N,?Littles,?Bigs):-
  Head >= N,
  partition(Tail,N,Littles,Bigs),
  ?Littles = Littles,?Bigs = [Head|Bigs].
```

5 Fill in “output” arguments

```
partition([],N,[],[]).
partition([Head|Tail],N,[Head|Littles],Bigs):-
  Head < N,
  partition(Tail,N,Littles,Bigs).
partition([Head|Tail],N,Littles,[Head|Bigs]):-
  Head >= N,
  partition(Tail,N,Littles,Bigs).
```
Logic Programming

Example: sort/2

1. Write down declarative specification

   % sort(L,S) <- S is a sorted permutation of list L

2. Identify recursion and “output” arguments

3. Write down implementation skeleton

   sort([],[]).
   sort([Head|Tail],?Sorted):-
     /* do something with Head */
     sort(Tail,Sorted).

4. Complete bodies of clauses

   sort([],[]).
   sort([Head|Tail],WholeSorted):-
     sort(Tail,Sorted),
     insert(Head,Sorted,WholeSorted).
Example: insert/3

1. Write down declarative specification

   % insert(X,In,Out) <- In is a sorted list, Out is In
   % with X inserted in the proper place

2. Identify recursion and “output” arguments

3. Write down implementation skeleton

   insert(X, [], ?Inserted).
   insert(X, [Head|Tail], ?Inserted):-
      /* do something with Head */
      insert(X, Tail, Inserted).
Logic Programming

Example: insert/3 (2)

4  Complete bodies of clauses

```prolog
insert(X, [], ?Inserted):-
    ?Inserted = [X].
insert(X, [Head|Tail], ?Inserted):-
    X > Head,
    insert(X, Tail, Inserted),
    ?Inserted = [Head|Inserted].
insert(X, [Head|Tail], ?Inserted):-
    X =< Head,
    ?Inserted = [X, Head|Tail].
```

5  Fill in “output” arguments

```prolog
insert(X, [], [X]).
insert(X, [Head|Tail], [X, Head|Tail]):-
    X =< Head.
insert(X, [Head|Tail], [Head|Inserted]):-
    X > Head,
    insert(X, Tail, Inserted).
```
**Logic Programming**

**Example: quicksort/2**

```prolog
quicksort([],[]).
quicksort([X|Xs],Sorted):-
    partition(Xs,X,Littles,Bigs),
quicksort(Littles,SortedLittles),
quicksort(Bigs,SortedBigs),
    append(SortedLittles,[X|SortedBigs],Sorted).
```

with difference lists:

```prolog
quicksort(Xs,Ys) :- qsort(Xs,Ys-[]).
qsort([],Ys-Ys).
qusort([X0|Xs],Ys-Zs) :-
    partition(Xs,X0,Ls,Bs),
qusort(Bs,Ys2-Zs),
qusort(Ls,Ys-[X0|Ys2]).
```
„I am unhappy.“
„How long have you been unhappy?“
„Six months. Can you help me?“
„What makes you think I help you?“
„You remind me of my sister."
„Can you tell me more about your sister?“
„I like teasing my sister."
„Does anyone else in your family like teasing your sister?“
„No. Only me."
„Please go on.“
Eliza Classic in prolog

Core algorithm

driven by stimulus-response patterns

I am <statement>.
How long have you been <statement>?

while the input is not „bye“
choose a stimulus-response pair
match the input to the stimulus
generate the reply from the response and the match
output the response
Dictionary lookup

as association list for arbitrary keys:

```prolog
lookup(Key, [(Key, Value)|Dict], Value).
lookup(Key, [(Key1, Value1)|Dict], Value) :-
    Key \= Key1,
    lookup(Key, Dict, Value).
```

as binary tree for integer keys:

```prolog
lookup2(Key, dict(Key, X, Left, Right), Value) :- !,
    X = Value.
lookup2(Key, dict(Key1, X, Left, Right), Value) :-
    Key < Key1,
    lookup2(Key, Left, Value).
lookup2(Key, dict(Key1, X, Left, Right), Value) :-
    Key > Key1,
    lookup2(Key, Right, Value).
```

will be used to store matches between stimulus and input
Eliza Classic in prolog
Representing stimulus-response patterns

pattern([i,am,1],["How",long,have,you,been,1,?]).
pattern([1,you,2,me],["What",makes,you,think,\'I',2,you,?]).
pattern([i,like,1],["Does",anyone,else,in,your,family,like,1,?]). pattern([i,feel,1],["Do",you,often,feel,that,way,?]).
pattern([1,X,2],["Please",you,tell,me,more,about,X]) :-
    important(X).
pattern([1],["Please",go,on,'.']].

important(father).
important(mother).
important(sister).
important(brother).
important(son).
important(daughter).
Eliza Classic in prolog

Main loop

reply([]) :- nl.
reply([Head|Tail]) :- write(Head), write(' '), reply(Tail).

eliza :- read(Input),
        eliza(Input),
        !.
eliza([bye]) :-
        writeln(['Goodbye. I hope I have helped you']).
eliza(Input) :-
        pattern(Stimulus, Response),
        match(Stimulus, Table, Input),
        match(Response, Table, Output),
        reply(Output),
        read(Input1),
        !,
        eliza(Input1).
Logic Programming

Eliza Classic in prolog

Actual matching

match([N|Pattern],Table,Target) :-
    integer(N),
    lookup(N,Table,LeftTarget),
    append(LeftTarget,RightTarget,Target),
    match(Pattern,Table,RightTarget).
match([Word|Pattern],Table,[Word|Target]) :-
    atom(Word),
    match(Pattern,Table,Target).
match([],Table,[]).

The incomplete datastructure does not have to be initialized!

Suppose \( D = [(a,b),(c,d)|X] \)

?- lookup(a,D,V)
V=b
?- lookup(c,D,e)
no
?- lookup(e,D,f)
yes
% \( D = [(a,b),(c,d),(e,f)|X] \)