Logic Programming

Georg Moser

Institute of Computer Science @ UIBK

Summer 2015

Summary of Last Lecture

Definition

- *functor*(*Term*,*F*,*Arity*) is true, if *Term* is a compound term, whose principal functor is *F* with arith *Arity*
- *arg*(*N*,*Term*,*Arg*) is true, if *Arg* is the N^{th} argument of *Term*

Definition

- *Term* =.. *List* is true if *List* is a list whose head is the principal functor of *Term*, and whose tail is the list of arguments of *Term*
- the operator =.. is also called *univ*

Definition

- *var*(*Term*) is true if *Term* is at present an uninstantiated variable
- *nonvar*(*Term*) is true if *Term* is at present not a variable
- *ground*(*Term*) is true if *Term* does not contain variables

Comparing Nonground Terms

Definition

- $X == Y$ is true if X and Y are identical constants, variables, or compound terms
- $X \backslash== Y$ is true if X and Y are not identical

Example (Unification with Occurs Check)

```
← unify_with_occurs_check(X,f(X)).
false
```

Remark

SWI-Prolog provides the following predicate that implements unification with occurs check:

$$unify\_with\_occurs\_check/2$$

Outline of the Lecture

Logic Programs

introduction, basic constructs, database and recursive programming, theory of logic programs

The Prolog Language

programming in pure prolog, arithmetic, structure inspection, meta-logical predicates, cuts, extra-logical predicates, how to program efficiently

Advanced Prolog Programming Techniques

nondeterministic programming, incomplete data structures, definite clause grammars, meta-programming, constraint logic programming

Outline of the Lecture

Logic Programs

introduction, basic constructs, database and recursive programming, theory of logic programs

The Prolog Language

programming in pure prolog, arithmetic, structure inspection, meta-logical predicates, cuts, extra-logical predicates, how to program efficiently

Advanced Prolog Programming Techniques

nondeterministic programming, incomplete data structures, definite clause grammars, meta-programming, constraint logic programming

Example

```
substitute(Old,New,Old,New).
substitute(Old,New,Term,Term) ←
    constant(Term),
    Term ≠ Old.
substitute(Old,New,Term,Term1) ←
    compound(Term),
    functor(Term,F,N),
    functor(Term1,F,N),
    substitute(N,Old,New,Term,Term1).

substitute(N,Old,New,Term,Term1) ←
    N > 0,
    arg(N,Term,Arg),
    substitute(Old,New,Arg,Arg1),
    arg(N,Term1,Arg1),
    N1 is N - 1,
    substitute(N1,Old,New,Term,Term1).
substitute(0,Old,New,Term,Term1).
```

Variables as Objects

Observation

(logical) variables can be accidentally instantiated

Variables as Objects

Observation

(logical) variables can be accidentally instantiated

Example

```
← substitute(a,b,X,X).
false
```

Variables as Objects

Observation

(logical) variables can be accidentally instantiated

Example

```
← substitute(a,b,X,X).
false
```

Example (cont'd)

```
substitute(Old,New,Term,New) ←
    ground(Term), Old = Term.
substitute(Old,New,Term,Term) ←
    constant(Term), Term ≠ Old.
substitute(Old,New,Var,Var) ←
    var(Var).
⋮
```

Observation

- the problem comes from a mixing of object-level and meta-level variables
- one (crude) solution is to avoid logical variables on object level
- another solution is to freeze logical variable so that they become objects

Observation

- the problem comes from a mixing of object-level and meta-level variables
- one (crude) solution is to avoid logical variables on object level
- another solution is to freeze logical variable so that they become objects

Freeze and Melt

- the predicate *freeze*(*Term*, *Frozen*) makes a copy of *Term* into *Frozen*
- all variables in *Term* become constants in *Frozen*
- *melt*(*Frozen*, *Thawed*) is the reversed funcion to *freeze*

Observation

- the problem comes from a mixing of object-level and meta-level variables
- one (crude) solution is to avoid logical variables on object level
- another solution is to freeze logical variable so that they become objects

Freeze and Melt

- the predicate *freeze*(*Term*, *Frozen*) makes a copy of *Term* into *Frozen*
- all variables in *Term* become constants in *Frozen*
- *melt*(*Frozen*, *Thawed*) is the reversed funcion to *freeze*

Example

```
← freeze(f(X,Y),Frozen), ground(Frozen)
Frozen ↦ ...
```

Example

```
occurs_in(X,Term) ←
    subterm(Sub,Term),
    X == Sub.
```

Example

```
occurs_in(X,Term) ←
    subterm(Sub,Term),
    X == Sub.
```

Example

```
occurs_in(X,Term) ←
    freeze(X,Xf),
    freeze(Term,Termf),
    subterm(Xf,Termf).
```

Example

```
occurs_in(X,Term) ←
    subterm(Sub,Term),
    X == Sub.
```

Example

```
occurs_in(X,Term) ←
    freeze(X,Xf),
    freeze(Term,Termf),
    subterm(Xf,Termf).
```

Observations

- two frozen terms X and Y unify iff $X{==}Y$ holds
- freeze and melt allow to implement *substitute*/4 without unintended variable instantiation

Meta-Variable Facility

Definition

the meta-variable facility allows a variable to appear as a goal or in the body

Meta-Variable Facility

Definition

the meta-variable facility allows a variable to appear as a goal or in the body

Example

```
X; Y ← X.
X; Y ← Y.
```

Meta-Variable Facility

Definition

the meta-variable facility allows a variable to appear as a goal or in the body

Example

```
X; Y ← X.
X; Y ← Y.
```

Other Control Predicates

- *fail*/0 *false*/0

```
← fail.        ← false.
false          false
```

Meta-Variable Facility

Definition

the meta-variable facility allows a variable to appear as a goal or in the body

Example

$X;\ Y \leftarrow X.$
$X;\ Y \leftarrow Y.$

Other Control Predicates

- *fail*/0 *false*/0

  $\leftarrow$ fail. $\leftarrow$ false.
  false false

- *true*/0

  $\leftarrow$ true.
  true

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs),
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←
    no_doubles(Xs,Ys).
```

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs),
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←
    no_doubles(Xs,Ys).
```

```
← no_doubles([a,b,a,c,b],X).
```

X ↦ [a,c,b]

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs),
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←
    no_doubles(Xs,Ys).
```

```
← no_doubles([a,b,a,c,b],X).
```

```
X ↦ [a,c,b] ;
X ↦ [b,a,c,b]
```

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs),
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←
    no_doubles(Xs,Ys).
```

```
← no_doubles([a,b,a,c,b],X).
```

```
X ↦ [a,c,b] ;
X ↦ [b,a,c,b] ;
X ↦ [a,a,c,b]
```

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs),
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←
    no_doubles(Xs,Ys).
```

```
← no_doubles([a,b,a,c,b],X).
X ↦ [a,c,b] ;
X ↦ [b,a,c,b] ;
X ↦ [a,a,c,b] ;
X ↦ [a,b,a,c,b]
```

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs),
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←
    no_doubles(Xs,Ys).
```

```
← no_doubles([a,b,a,c,b],X).
X ↦ [a,c,b] ;
X ↦ [b,a,c,b] ;
X ↦ [a,a,c,b] ;
X ↦ [a,b,a,c,b] ;
false
```

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs),
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←
    \+ member(X,Xs),
    no_doubles(Xs,Ys).
```

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs),
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←
    \+ member(X,Xs),
    no_doubles(Xs,Ys).
```

```
← no_doubles([a,b,a,c,b],X).
```

X ↦ [a,c,b]

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs),
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←
    \+ member(X,Xs),
    no_doubles(Xs,Ys).
```

```
← no_doubles([a,b,a,c,b],X).
X ↦ [a,c,b] ;
false
```

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs),
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←
    \+ member(X,Xs),                negation as failure
    no_doubles(Xs,Ys).
```

```
← no_doubles([a,b,a,c,b],X).
X ↦ [a,c,b] ;
false
```

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs), !,              cut
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←

    no_doubles(Xs,Ys).
```

```
← no_doubles([a,b,a,c,b],X).
X ↦ [a,c,b] ;
false
```

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs), !,              cut
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←

    no_doubles(Xs,Ys).
```

Effect of Cut

! succeeds

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs), !,              cut
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←

    no_doubles(Xs,Ys).
```

Effect of Cut

- ! succeeds
- ! fixes all choices between (and including) moment of matching rule's head with parent goal and cut

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs), !,              cut
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←

    no_doubles(Xs,Ys).
```

Effect of Cut

- ! succeeds
- ! fixes all choices between (and including) moment of matching rule's head with parent goal and cut

  if backtracking reaches !, the cut fails and the search continues from the last choice made before the clause containing ! was chosen

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs), !,           cut
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←

    no_doubles(Xs,Ys).
```

Effect of Cut

$$\mathtt{p}(t_{11}, \dots, t_{1n}) \leftarrow A_1, \dots, A_k\,.$$
$$\vdots$$
$$\mathtt{p}(t_{i1}, \dots, t_{in}) \leftarrow B_1, \dots, B_i\,,\ \mathtt{!}\,,\ C_1, \dots, C_j\,.$$
$$\vdots$$
$$\mathtt{p}(t_{m1}, \dots, t_{mn}) \leftarrow D_1, \dots, D_l\,.$$

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs), !,            cut
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←

    no_doubles(Xs,Ys).
```

Effect of Cut

$$\mathtt{p}(t_{11}, \dots, t_{1n}) \leftarrow A_1, \dots, A_k\,.$$

$$\vdots$$

$$\mathtt{p}(t_{i1}, \dots, t_{in}) \leftarrow B_1, \dots, B_i\,,\ !\,,\ C_1, \dots, C_j\,.$$

$$\vdots$$

$$\mathtt{p}(t_{m1}, \dots, t_{mn}) \leftarrow D_1, \dots, D_l\,.$$

Example (Removal of Duplicates)

```
no_doubles([],[]).
no_doubles([X|Xs],Ys) ←
    member(X,Xs), !,            cut
    no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) ←

    no_doubles(Xs,Ys).
```

Effect of Cut

$\mathtt{p}(t_{11}, \ldots, t_{1n}) \leftarrow A_1, \ldots, A_k\,.$

$\vdots$

$\mathtt{p}(t_{i1}, \ldots, t_{in}) \leftarrow B_1, \ldots, B_i\,,\ \mathtt{!}\,,\ C_1, \ldots, C_j\,.$

$\vdots$ blocked

$\mathtt{p}(t_{m1}, \ldots, t_{mn}) \leftarrow D_1, \ldots, D_l\,.$

Examples of (Green) Cuts

Example (Without Cuts)

```
merge([X|Xs], [Y|Ys],[X|Zs]) ←
    X < Y,  merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[Y|Ys],[X,Y|Zs]) ←
    X = Y,  merge(Xs,Ys,Zs).
merge([X|Xs],[Y|Ys],[Y|Zs]) ←
    X > Y,  merge([X|Xs],Ys,Zs).
merge(Xs,[],Xs) .
merge([],Ys,Ys) .
```

Examples of (Green) Cuts

Example (With Cuts)

```
merge([X|Xs], [Y|Ys],[X|Zs]) ←
    X < Y, !, merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[Y|Ys],[X,Y|Zs]) ←
    X = Y, !, merge(Xs,Ys,Zs).
merge([X|Xs],[Y|Ys],[Y|Zs]) ←
    X > Y, !, merge([X|Xs],Ys,Zs).
merge(Xs,[],Xs) ← !.
merge([],Ys,Ys) ← !.
```

Examples of (Green) Cuts

Example (With Cuts)

```
merge([X|Xs], [Y|Ys],[X|Zs]) ←
    X < Y, !, merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[Y|Ys],[X,Y|Zs]) ←
    X = Y, !, merge(Xs,Ys,Zs).
merge([X|Xs],[Y|Ys],[Y|Zs]) ←
    X > Y, !, merge([X|Xs],Ys,Zs).
merge(Xs,[],Xs) ← !.
merge([],Ys,Ys) ← !.
```

Example

```
minimum(X,Y,X) ← X ⩽ Y, !.
minimum(X,Y,Y) ← X > Y, !.
```

Fact

(Green) cuts can greatly increase the efficiency by removing redundant computations

Fact

(Green) cuts can greatly increase the efficiency by removing redundant computations

Example

```
ordered([X]).
ordered([X,Y|Xs]) ← X ⩽ Y, ordered([Y|Xs]).

sort(Xs,Ys) ←
    append(As,[X,Y|Bs],Xs),
    X > Y,
    append(As,[Y,X|Bs],Xs1),
    sort(Xs1,Ys1).
```

Fact

(Green) cuts can greatly increase the efficiency by removing redundant computations

Example

```
ordered([X]).
ordered([X,Y|Xs]) ← X ⩽ Y, ordered([Y|Xs]).

sort(Xs,Ys) ←
    append(As,[X,Y|Bs],Xs),
    X > Y,
    append(As,[Y,X|Bs],Xs1),
    sort(Xs1,Ys1).
 sort(Xs,Xs) ←
    ordered(Xs).
```

Fact

(Green) cuts can greatly increase the efficiency by removing redundant computations

Example

```
ordered([X]).
ordered([X,Y|Xs]) ← X ⩽ Y, ordered([Y|Xs]).

sort(Xs,Ys) ←
    append(As,[X,Y|Bs],Xs),
    X > Y,
    append(As,[Y,X|Bs],Xs1),
    sort(Xs1,Ys1).
 sort(Xs,Xs) ←
    ordered(Xs).

 ← sort([3,2,1],Xs)
Xs ↦ [1,2,3]
```

Fact

(Green) cuts can greatly increase the efficiency by removing redundant computations

Example

```
ordered([X]).
ordered([X,Y|Xs]) ← X ⩽ Y, ordered([Y|Xs]).

sort(Xs,Ys) ←
    append(As,[X,Y|Bs],Xs),
    X > Y, !,
    append(As,[Y,X|Bs],Xs1),
    sort(Xs1,Ys1).
 sort(Xs,Xs) ←
    ordered(Xs), !.

 ← sort([3,2,1],Xs)
Xs ↦ [1,2,3]
```

Definition (Negation as Failure)

- negation \+ is implemented using cut

Definition (Negation as Failure)

- negation \+ is implemented using cut
- the principle of negation is limited and known as negation as failure

Definition (Negation as Failure)

- negation \+ is implemented using cut
- the principle of negation is limited and known as negation as failure

Example

```
not X ← X, !, fail.
not X.
```

Definition (Negation as Failure)

- negation \+ is implemented using cut
- the principle of negation is limited and known as negation as failure

Example

```
not X ← X, !, fail.
not X.
```

Observation

if G does not terminate, not(G) may or may not terminate

Definition (Negation as Failure)

- negation \+ is implemented using cut
- the principle of negation is limited and known as negation as failure

Example

```
not X ← X, !, fail.
not X.
```

Observation

if G does not terminate, not(G) may or may not terminate

Example

```
married(abraham,sarah).
married(X,Y) ← married(Y,X)
← not married(abraham,sarah).
```

Green vs Red Cuts

Definition

- a cut is green if the addition of the cut doesn't cange the meaning of the program; removing it makes the program potentially inefficient, but not wrong

Green vs Red Cuts

Definition

- a cut is green if the addition of the cut doesn't cange the meaning of the program; removing it makes the program potentially inefficient, but not wrong
- a cut is red if its presence changes the meaning of the program; removing it, changes the meaning and thus may make the program wrong

Green vs Red Cuts

Definition

- a cut is green if the addition of the cut doesn't cange the meaning of the program; removing it makes the program potentially inefficient, but not wrong
- a cut is red if its presence changes the meaning of the program; removing it, changes the meaning and thus may make the program wrong

Example (...)

```
minimum(X,Y,X) ← X ⩽ Y, .
minimum(X,Y,Y).
```

```
← minimum(2,5,X)
```

$X = 2$
$X = 5$

Green vs Red Cuts

Definition

- a cut is green if the addition of the cut doesn't cange the meaning of the program; removing it makes the program potentially inefficient, but not wrong
- a cut is red if its presence changes the meaning of the program; removing it, changes the meaning and thus may make the program wrong

Example (...)

```
minimum(X,Y,X) ← X ⩽ Y, !.
minimum(X,Y,Y).
```

```
← minimum(2,5,X)
```

$X = 2$

Green vs Red Cuts

Definition

- a cut is green if the addition of the cut doesn't cange the meaning of the program; removing it makes the program potentially inefficient, but not wrong
- a cut is red if its presence changes the meaning of the program; removing it, changes the meaning and thus may make the program wrong

Example (Bad Cut)

```
minimum(X,Y,X) ← X ⩽ Y, !.
minimum(X,Y,Y).

← minimum(2,5,5)

true
```

Example of Green and Red Cuts

Example (. . .)

```
delete([X|Ys],X,Zs) ← !, delete(Ys,X,Zs).
delete([Y|Ys],X,[Y|Zs]) ← Y ≠ X, !, delete(Ys,X,Zs).
delete([],X,[]).
```

Example of Green and Red Cuts

Example (Green Cut)

```
delete([X|Ys],X,Zs) ← !, delete(Ys,X,Zs).
delete([Y|Ys],X,[Y|Zs]) ← Y ≠ X, !, delete(Ys,X,Zs).
delete([],X,[]).
```

Example of Green and Red Cuts

Example (Green Cut)

```
delete([X|Ys],X,Zs) ← !, delete(Ys,X,Zs).
delete([Y|Ys],X,[Y|Zs]) ← Y ≠ X, !, delete(Ys,X,Zs).
delete([],X,[]).
```

Example (. . .)

```
delete([X|Xs],X,Zs) ← !, delete(Ys,X,Zs).
delete([Y|Ys],X,[Y|Zs]) ← !, delete(Ys,X,Zs).
delete([],X,[]).
← \+ delete([a,b],b,[a,b]).
```

Example of Green and Red Cuts

Example (Green Cut)

```
delete([X|Ys],X,Zs) ← !, delete(Ys,X,Zs).
delete([Y|Ys],X,[Y|Zs]) ← Y ≠ X, !, delete(Ys,X,Zs).
delete([],X,[]).
```

Example (Red Cut)

```
delete([X|Xs],X,Zs) ← !, delete(Ys,X,Zs).
delete([Y|Ys],X,[Y|Zs]) ← !, delete(Ys,X,Zs).
delete([],X,[]).
← \+ delete([a,b],b,[a,b]).
```

Example of Green and Red Cuts

Example (Green Cut)

```
delete([X|Ys],X,Zs) ← !, delete(Ys,X,Zs).
delete([Y|Ys],X,[Y|Zs]) ← Y ≠ X, !, delete(Ys,X,Zs).
delete([],X,[]).
```

Example (Red Cut)

```
delete([X|Xs],X,Zs) ← !, delete(Ys,X,Zs).
delete([Y|Ys],X,[Y|Zs]) ← !, delete(Ys,X,Zs).
delete([],X,[]).

← \+ delete([a,b],b,[a,b]).
```

Example (. . .)

```
member(X,[X|Xs]) ← !.
member(X,[Y|Ys]) ← member(X,Ys).
```

Example of Green and Red Cuts

Example (Green Cut)

```
delete([X|Ys],X,Zs) ← !, delete(Ys,X,Zs).
delete([Y|Ys],X,[Y|Zs]) ← Y ≠ X, !, delete(Ys,X,Zs).
delete([],X,[]).
```

Example (Red Cut)

```
delete([X|Xs],X,Zs) ← !, delete(Ys,X,Zs).
delete([Y|Ys],X,[Y|Zs]) ← !, delete(Ys,X,Zs).
delete([],X,[]).

← \+ delete([a,b],b,[a,b]).
```

Example (Red Cut)

```
member(X,[X|Xs]) ← !.
member(X,[Y|Ys]) ← member(X,Ys).
```

Example (Truth Tables for Propositional Formulas)

```
and(A,B) ← A, B.
or(A,B) ← A; B.
implies(A,B) ← or(not(A),B).
```

Example (Truth Tables for Propositional Formulas)

```
and(A,B) ← A, B.
or(A,B) ← A; B.
implies(A,B) ← or(not(A),B).

bind(true).
bind(false).

table(A,B,E) ← bind(A), bind(B), row(A,B,E), fail.
```

Example (Truth Tables for Propositional Formulas)

```
and(A,B) ← A, B.
or(A,B) ← A; B.
implies(A,B) ← or(not(A),B).

bind(true).
bind(false).

table(A,B,E) ← bind(A), bind(B), row(A,B,E), fail.
table(_,_,_) ← nl.
```

Example (Truth Tables for Propositional Formulas)

```
and(A,B) ← A, B.
or(A,B) ← A; B.
implies(A,B) ← or(not(A),B).

bind(true).
bind(false).

table(A,B,E) ← bind(A), bind(B), row(A,B,E), fail.
table(_,_,_) ← nl.

row(A,B,_) ← wr(A), write(' '), wr(B), write(' '), fail.
row(_,_,E) ← E, !, wr(true), nl.
row(_,_,_) ← wr(false), nl.

wr(true) ← write('T').
wr(false) ← write('F').
```

Example (Truth Tables for Propositional Formulas)

```
and(A,B) ← A, B.
or(A,B) ← A; B.
implies(A,B) ← or(not(A),B).

bind(true).
bind(false).

table(A,B,E) ← bind(A), bind(B), row(A,B,E), fail.
table(_,_,_) ← nl.

row(A,B,_) ← wr(A), write(' '), wr(B), write(' '), fail.
row(_,_,E) ← E, !, wr(true), nl.
row(_,_,_) ← wr(false), nl.

wr(true) ← write('T').
wr(false) ← write('F').

← table(A,B,or(A,implies(B,or(B,and(A,B))))).
```

Example (Truth Tables for Propositional Formulas)

```
and(A,B) ← A, B.
or(A,B) ← A; B.
implies(A,B) ← or(not(A),B).

bind(true).
bind(false).

table(A,B,E) ← bind(A), bind(B), row(A,B,E), fail.
table(_,_,_) ← nl.

row(A,B,_) ← wr(A), write(' '), wr(B), write(' '), fail.
row(_,_,E) ← E, !, wr(true), nl.
row(_,_,_) ← wr(false), nl.

wr(true) ← write('T').
wr(false) ← write('F').

← table(A,B,or(A,implies(B,or(B,and(A,B))))).

← table(A,B,false).
```

Cut-Fail Combinations

Example (Implementing $\neq$)

```
X ≠ X → !, fail.
X ≠ Y.
```

Cut-Fail Combinations

Example (Implementing ≠)

$\texttt{X} \neq \texttt{X} \rightarrow \texttt{!, fail.}$
$\texttt{X} \neq \texttt{Y.}$

Example (Implementing if_then_else)

```
if_then_else(P,Q,R) ← P, !, Q.
if_then_else(P,Q,R) ← R.
```

Cut-Fail Combinations

Example (Implementing $\neq$)

```
X ≠ X → !, fail.
X ≠ Y.
```

Example (Implementing if_then_else)

```
if_then_else(P,Q,R) ← P, !, Q.
if_then_else(P,Q,R) ← R.
```

Example (Implementing same_vars)

```
same_var(foo,Y) ← var(Y), !, fail.
same_var(X,Y) ← var(X), var(Y).
```

Extra-Logical Predicates

Definition

predicates in Prolog outside of the logic programming model are called extra-logical predicates

1. predicates concerned with I/0
2. predicates for accessing and manipulating the program
3. predicates for interfacing the operatiing system

Extra-Logical Predicates

Definition

predicates in Prolog outside of the logic programming model are called extra-logical predicates

1. predicates concerned with I/O
2. predicates for accessing and manipulating the program
3. predicates for interfacing the operatiing system

input/output

Extra-Logical Predicates

Definition

predicates in Prolog outside of the logic programming model are called extra-logical predicates

1. predicates concerned with I/O
2. predicates for accessing and manipulating the program
3. predicates for interfacing the operatiing system

input/output

- *read*(X) is true if X unifies with term read from input stream

Extra-Logical Predicates

Definition

predicates in Prolog outside of the logic programming model are called extra-logical predicates

1. predicates concerned with I/0
2. predicates for accessing and manipulating the program
3. predicates for interfacing the operatiing system

input/output

- *read*(X) is true if X unifies with term read from input stream
- *write*(X) writes X to output stream; always succeeds

Extra-Logical Predicates

Definition

predicates in Prolog outside of the logic programming model are called extra-logical predicates

1. predicates concerned with I/0
2. predicates for accessing and manipulating the program
3. predicates for interfacing the operatiing system

input/output

- *read*(X) is true if X unifies with term read from input stream
- *write*(X) writes X to output stream; always succeeds
- *get*(X) is true if X unifies with the ASCII code of the first character

Extra-Logical Predicates

Definition

predicates in Prolog outside of the logic programming model are called extra-logical predicates

1. predicates concerned with I/0
2. predicates for accessing and manipulating the program
3. predicates for interfacing the operatiing system

input/output

- *read*(X) is true if X unifies with term read from input stream
- *write*(X) writes X to output stream; always succeeds
- *get*(X) is true if X unifies with the ASCII code of the first character
- *put*(N) writes character corresponding to ASCII code N to output stream

Example

```
read_word_list(Ws) ←
  get(C),
  read_word_list(C,Ws).

read_word_list(C,[W|Ws]) ←
  word_char(C),
  read_word(C,W,C1),
  read_word_list(C1,Ws).
read_word_list(C,Ws) ←
  fill_char(C),
  get(C1),
  read_word_list(C1,Ws).
read_word_list(C,[]) ←
  end_of_words_char(C).

read_word(C,W,C1) ←
  word_chars(C,Cs,C1),
  name(W,Cs).

word_chars(C,[C|Cs],C0) ←
  word_char(C),
  !,
  get(C1),
  word_chars(C1,Cs,C0).
word_chars(C,[],C) ←
  \+ word_char(C).

word_char(C) ← 97 ⩽ C, C ⩽ 122.
word_char(C) ← 65 ⩽ C, C ⩽ 90.
word_char(95).

fill_char(32).

end_of_words_char(46).
```