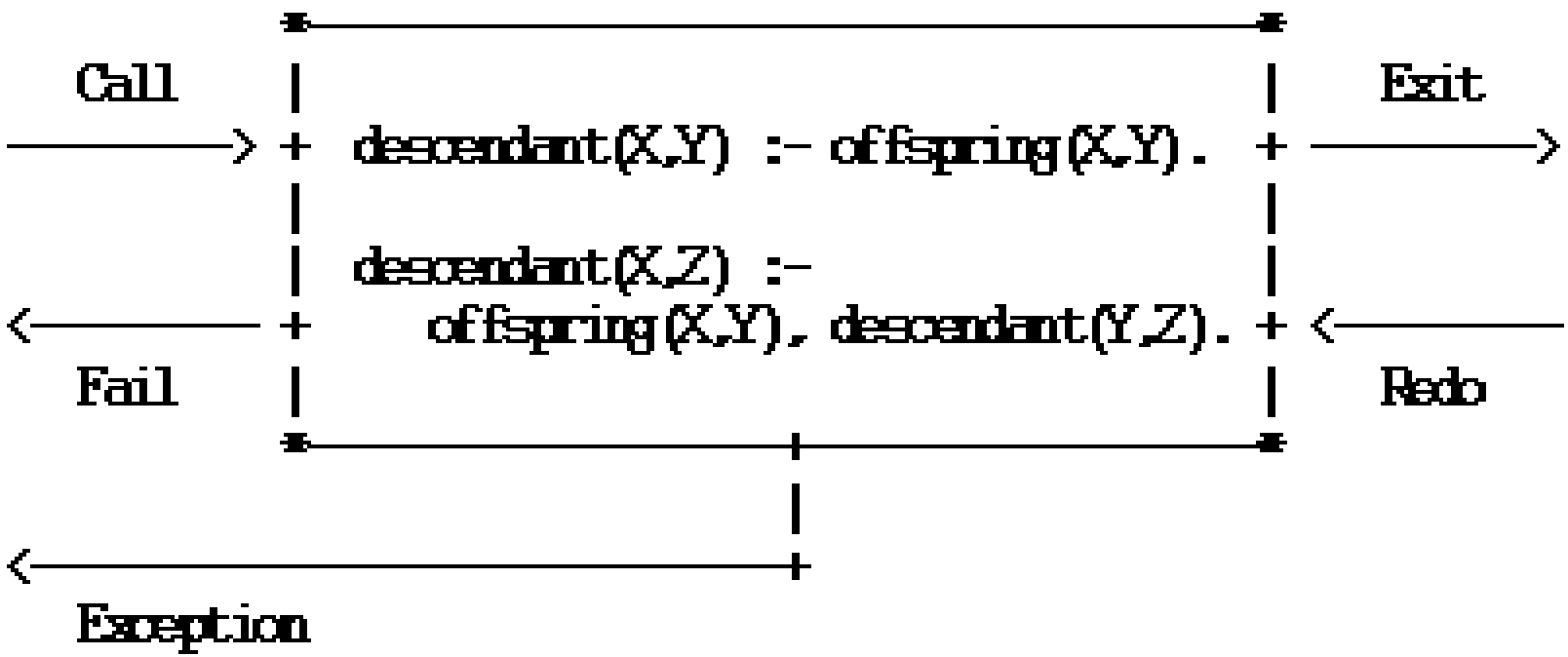
# *Prolog*

# Continuation

# Debugging Prolog Programs

- Offers some challenges
  - Backtracking provides additional complexity
  - Prolog interpreter maintains more states
- Offers some advantages as well
  - Interactive, dynamic envriornment
  - Easy to examine and modify on the fly
- Tools:
  - tracing
  - spy points
  - break

# The Procedure Box Control Flow Model



# The Ports

- **Call** This arrow represents initial invocation of the predicate.
- **Exit** This arrow represents a successful return from the predicate. This occurs when the initial goal has been unified with one of the component clauses and any subgoals have been satisfied.
- **Redo** This arrow indicates that a subsequent goal has failed and that the system is backtracking in an attempt to find alternatives to previous solutions. An attempt will be made to resatisfy one of the component subgoals in the body of the clause that last succeeded; or, if that fails, to completely rematch the original goal with an alternative clause and then try to satisfy any subgoals in the body of this new clause.
- **Fail** This arrow represents a failure of the initial goal, which might occur if no clause is matched, or if subgoals are never satisfied, or if any solution produced is always rejected by later processing.
- **Exception** This arrow represents an exception which was raised in the initial goal.

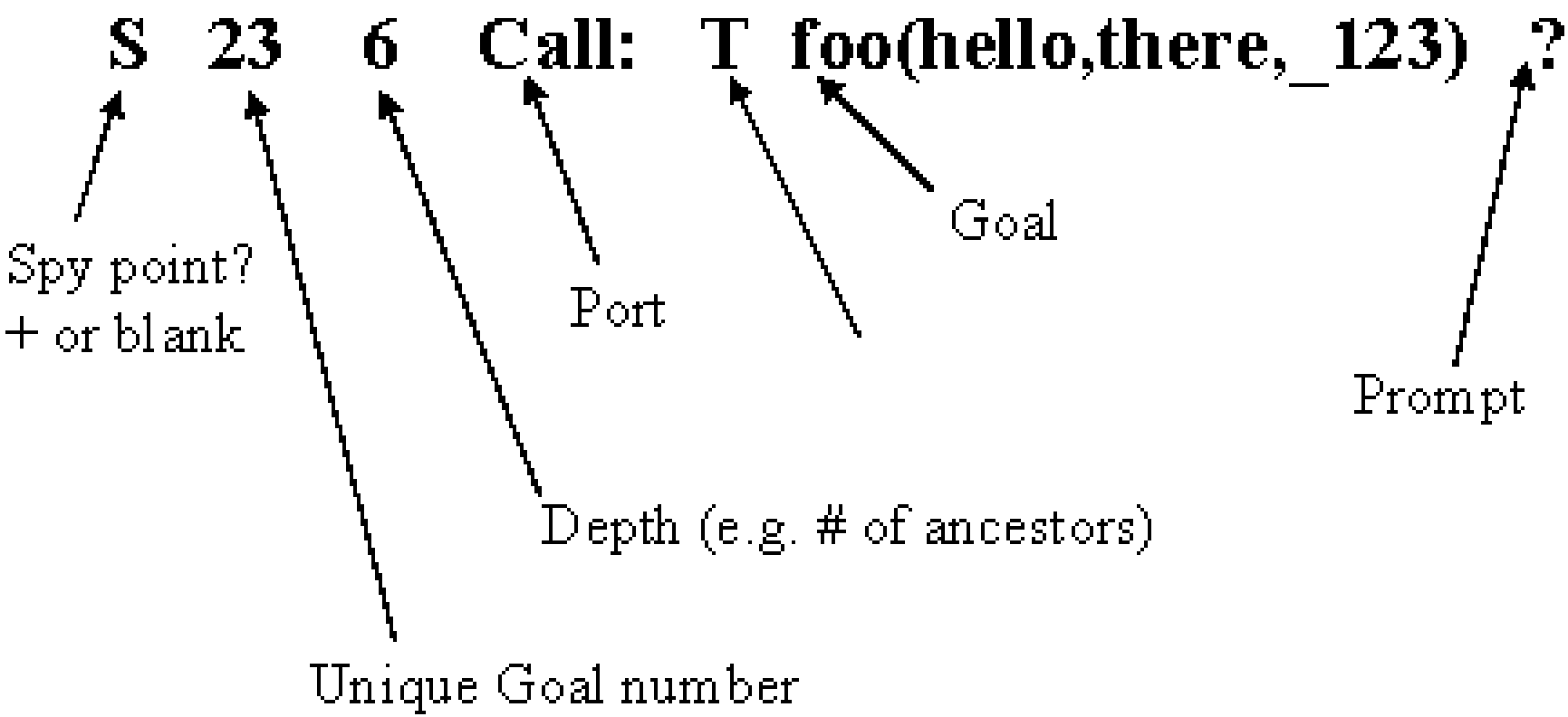
# Some Debugging Predicates

- trace/0
  - turns tracing on
- notrace/0
  - turns tracing off
- leash/1
  - leash(X) controls when the user is prompted during tracing
- spy/1
  - spy(foo/2) sets a "spy point" to stop whenever foo/2 is called
- nospy/1
  - nospy(foo/2) removes a spy point from foo/2
- nospyall/0
  - removes all spy points

# A simple trace

```
| ?- trace, member(3,[1,2,3]).
{The debugger will first creep -- showing everything (trace)}
   1  1  Call: member(3,[1,2,3]) ?
   2  2  Call: member(3,[2,3]) ?
   3  3  Call: member(3,[3]) ?
   3  3  Exit: member(3,[3]) ?
   2  2  Exit: member(3,[2,3]) ?
   1  1  Exit: member(3,[1,2,3]) ?
yes
{trace}
```

# Format of Debugging messages



# A more complicated trace

```
| ?- member(X,[a,b,1,c,9]), integer(X),
     X>5.
   1  1  Call: member(_183,[a,b,1,c,9]) ?
   1  1  Exit: member(a,[a,b,1,c,9]) ?
   2  1  Call: integer(a) ?
   2  1  Fail: integer(a) ?
   1  1  Redo: member(a,[a,b,1,c,9]) ?
   2  2  Call: member(_183,[b,1,c,9]) ?
   2  2  Exit: member(b,[b,1,c,9]) ?
   1  1  Exit: member(b,[a,b,1,c,9]) ?
   3  1  Call: integer(b) ?
   3  1  Fail: integer(b) ?
   1  1  Redo: member(b,[a,b,1,c,9]) ?
   2  2  Redo: member(b,[b,1,c,9]) ?
   3  3  Call: member(_183,[1,c,9]) ?
   3  3  Exit: member(1,[1,c,9]) ?
   2  2  Exit: member(1,[b,1,c,9]) ?
   1  1  Exit: member(1,[a,b,1,c,9]) ?
   4  1  Call: integer(1) ?
   4  1  Exit: integer(1) ?
   5  1  Call: 1>5 ?
   5  1  Fail: 1>5 ?
   4  1  Redo: integer(1) ?
   4  1  Fail: integer(1) ?
   1  1  Redo: member(1,[a,b,1,c,9]) ?
   2  2  Redo: member(1,[b,1,c,9]) ?
   3  3  Redo: member(1,[1,c,9]) ?
   4  4  Call: member(_183,[c,9]) ?
   4  4  Exit: member(c,[c,9]) ?
   3  3  Exit: member(c,[1,c,9]) ?
   2  2  Exit: member(c,[b,1,c,9]) ?
   1  1  Exit: member(c,[a,b,1,c,9]) ?
   5  1  Call: integer(c) ?
   5  1  Fail: integer(c) ?
   1  1  Redo: member(c,[a,b,1,c,9]) ?
   2  2  Redo: member(c,[b,1,c,9]) ?
   3  3  Redo: member(c,[1,c,9]) ?
   4  4  Redo: member(c,[c,9]) ?
   5  5  Call: member(_183,[9]) ?
   5  5  Exit: member(9,[9]) ?
   4  4  Exit: member(9,[c,9]) ?
   3  3  Exit: member(9,[1,c,9]) ?
   4  1 2  2  Exit: member(9,[b,1,c,9]) ?
   1  1  Exit: member(9,[a,b,1,c,9]) ?
   6  1  Call: integer(9) ?
   6  1  Exit: integer(9) ?
   7  1  Call: 9>5 ?
   7  1  Exit: 9>5 ?
X = 9 ?

yes
{trace}
| ?-
```

# Options Available during Debugging

| | | | |
|---|---|---|---|
| RET | creep | c | creep |
| l | leap | s | skip |
| r | retry | r <i> | retry i |
| f | fail | f <i> | fail i |
| d | display | w | write |
| p | print | p <i> | print partial |
| g | ancestors | g <n> | ancestors n |
| & | blocked goals | & <n> | nth blocked goal |
| n | nodebug | = | debugging |
| + | spy this | + <i> | spy conditionally |
| - | nospy this | . | find this |
| a | abort | b | break |
| @ | command | u | unify |
| e | pending exception | | |
| < | reset printdepth | < <n> | set printdepth |
| ^ | reset subterm | ^ <n> | set subterm |
| ? | help | h | help |

# Spy Points

- For real programs it is impractical to creep through the entire program.
- Spy-points make it possible to stop the program whenever it gets to a particular predicate of interest. Once there, one can set further spy-points in order to catch the control flow a bit further on, or one can start creeping.
- Setting a spy-point on a predicate indicates that you wish to see all control flow through the various ports of its invocation boxes, except during skips.
- When control passes through any port of a procedure box with a spy-point set on it, a message is output and the user is asked to interact.

# Example: Towers of Hanoi

```
hanoi(N) :- move(N,left,center,right).


move(0,_,_,_).
move(N,A,B,C) :-
    N>0,
    M is N-1,
    move(M,A,C,B),
    move1disk(A,B),
    move(M,C,B,A).


move1disk(From,To) :-
    format("Move disk from ~p peg to ~p peg.~n",[From,To]).
```

# Spypoint example

```
| ?- spy(move_ldisk).
{The debugger will first leap -- showing spypoints (debug)}
yes
{debug}
| ?- hanoi(4).
 + 15  6  Call: move_ldisk(left,right) ?1
Move disk from left peg to right peg.
 + 15  6  Exit: move_ldisk(left,right) ?1
 + 18  5  Call: move_ldisk(left,center) ?
   19  6  Call: format([77,111,118,101,32,100,105,115,107|...],[left,center]) ?
Move disk from left peg to center peg.
   19  6  Exit: format([77,111,118,101,32,100,105,115,107|...],[left,center]) ?
 + 18  5  Exit: move_ldisk(left,center) ?
   20  5  Call: move(_1,right,center,left) ?
   21  6  Call: _1>0 ?
   21  6  Exit: 1>0 ?
   22  6  Call: _2634 is 1-1 ?
   22  6  Exit: 0 is 1-1 ?
   23  6  Call: move(0,right,left,center) ?1
 + 24  6  Call: move_ldisk(right,center) ?
```

# Leash

- **leash(+Mode)** Leashing Mode determines the ports where you are prompted when creeping through your program.
- At unleashed ports a tracing message is still output, but program execution does not stop to allow user interaction.
- Mode can be a subset of the following, specified as a list:
  - call - Prompt on Call.
  - exit - Prompt on Exit.
  - redo - Prompt on Redo.
  - fail - Prompt on Fail.
  - exception - Prompt on Exception.
- leash( [call,exit,redo,fail,exception]). Is the default.

Assert

Asserta

Assertz

# Modification of the Program

- These predicates allow modification of dynamic predicates
  - Dynamic clauses can be added (asserted) or removed from the program (retracted).
  - Declarations:

    ```
    :- dynamic foo/2, bar/3.
    ```
- *Head* must be instantiated to an atom or a compound term (with an optional module prefix) and *Clause* must be instantiated either to a term *Head* :- *Body* or, if the body part is empty, to *Head* (with an optional module prefix). An empty body part is represented as true.
- A term **Head :- Body** must be enclosed in parentheses when it occurs as an argument of a compound term, as `:-' is a standard infix operator with precedence greater than 1000 (see section Operators), e.g.:
  - `| ?- assert((Head :- Body)).`

# Assert/1 and assert/2

**assert(:Clause)**

**assert(:Clause,-Ref)**

- The current instance of Clause is interpreted as a clause and is added to the current interpreted program.
- The predicate concerned must currently be dynamic or undefined and the position of the new clause within it is implementation-defined.
- Ref is a unique identifier of the asserted clause. Any uninstantiated variables in the Clause will be replaced by new private variables, along with copies of any subgoals blocked on these variables.

# Asserta and assertz

**asserta(:Clause)**

**asserta(:Clause,-Ref)**

- Like assert/2, except that the new clause becomes the first clause for the predicate concerned.

**assertz(:Clause)**

**assertz(:Clause,-Ref)**

- Like assert/2, except that the new clause becomes the last clause for the predicate concerned.

# Asserta and assertz

**asserta(:Clause)**

**asserta(:Clause,-Ref)**

- Like assert/2, except that the new clause becomes the first clause for the predicate concerned.

**assertz(:Clause)**

**assertz(:Clause,-Ref)**

- Like assert/2, except that the new clause becomes the last clause for the predicate concerned.

# Retract/1

**retract(:Clause)**

- The first clause in the current interpreted program that matches Clause is erased.
- The predicate concerned must currently be dynamic.
- retract/1 may be used in a non-determinate fashion, i.e. it will successively retract clauses matching the argument through backtracking.
- If reactivated by backtracking, invocations of the predicate whose clauses are being retracted will proceed unaffected by the retracts.
- This is also true for invocations of clause for the same predicate.

# Retractall/1

**retractall(:Head)**

- Erases all clauses whose head matches Head, where Head must be instantiated to an atom or a compound term.
- The predicate concerned must currently be dynamic.
- The predicate definition is retained.

# abolish/1 and abolish/2

abolish(:Spec)

abolish(:Name,+Arity)

- Erases all clauses of the predicate specified by Spec or Name/Arity. The predicate definition and all associated information such as spy-points is also erased.
- The predicates concerned must all be user defined.

erase(+Ref)

- The dynamic clause or recorded term whose implementation-defined identifier is Ref is effectively erased from the internal database or interpreted program.

# instance/2

**instance(+Ref,?Term)**

- A (most general) instance of the dynamic clause or recorded term whose implementation-defined identifier is Ref is unified with Term.
- Ref must be instantiated to a legal identifier.

# All Solutions in Prolog

# Finding all solutions

- When there are many solutions to a problem, and when all those solutions are required to be collected together, this can be achieved by repeatedly backtracking and gradually building up a list of the solutions.
- The following built-in predicates are provided to automate this process.
  - setof/3
  - bagof/3
  - findall/3

# setof(?Template,:Goal,?Set)

**"Set is the set of all instances of Template such that Goal is satisfied, where that set is non-empty."**

- Goal specifies a goal or goals as in call(Goal).
- Set is a set of terms represented as a list of those terms, without duplicates, in the standard order for terms.

- If there are no instances of Template such that Goal is satisfied then the predicate fails.
- The variables appearing in the term Template should not appear anywhere else in the clause except in the term Goal.
- If uninstantiated variables in Goal not also appearing in Template, then a call to this built-in predicate may backtrack, generating alternative values for Set corresponding to different instantiations of the free variables of Goal.

# Setof example

```
likes(bill, cider).
likes(dick, beer).
likes(harry, beer).
likes(jan, cider).
likes(tom, beer).
likes(tom, cider).

| ?- setof(X, likes(X,Y), S).
S = [dick,harry,tom],
Y = beer ? ;
S = [bill,jan,tom],
Y = cider ? ;
no
| ?- setof((Y,S), setof(X, likes(X,Y), S), SS).
SS = [(beer,[dick,harry,tom]),
      (cider,[bill,jan,tom])]
```

# Existential quantifier

- Variables occurring in Goal will not be treated as free if they are explicitly bound within Goal by an existential quantifier.
- An existential quantification is written: **Y^Q** meaning "there exists a Y such that Q is true", where Y is some Prolog variable. Example:

```
| ?- setof(X, Y^(likes(X,Y)), S).
S = [bill,dick,harry,jan,tom];
no
```

- Alternatively:

```
?- assert((likes_something(X) :- once(likes(X,_)).
?- Assert((Once(Goal) :- call(Goal),!)).
setof(X, likes_something(X), S).
S = [bill,dick,harry,jan,tom];
no
```

# bagof(?Template,:Goal,?Bag)

- This is exactly the same as setof/3 except
  - the list (or alternative lists) returned will not be ordered, and
  - may contain duplicates.
- The effect of this relaxation is to save a call to sort/2, which is invoked by setof/3 to return an ordered list.

# findall(?Template,:Goal,?Bag)

"Bag is a list of instances of Template in all proofs of Goal found by Prolog."

- The order of the list corresponds to the order in which the proofs are found.
- The list may be empty.
- All variables are taken as being existentially quantified.

- This means that each invocation of findall/3 succeeds exactly once, and that no variables in Goal get bound.
- Avoiding the management of universally quantified variables can save considerable time and space.

# Implementing findall/3

```
findall(X,G,_) :-
  asserta(found(mark)),
  call(G),
  asserta(found(X)),
  fail.

findall(_,_,L) :-
  collectFound([],M),
  !,
  L=M.
```

```
collectFound(S,L) :-
  getNext(X),
  !,
  collectFound([X|S],L).
collectFound(L,L).

getNext(X) :-
  retract(found(X)),
  !,
  \+(X=mark).
```

# Implementing findall/3

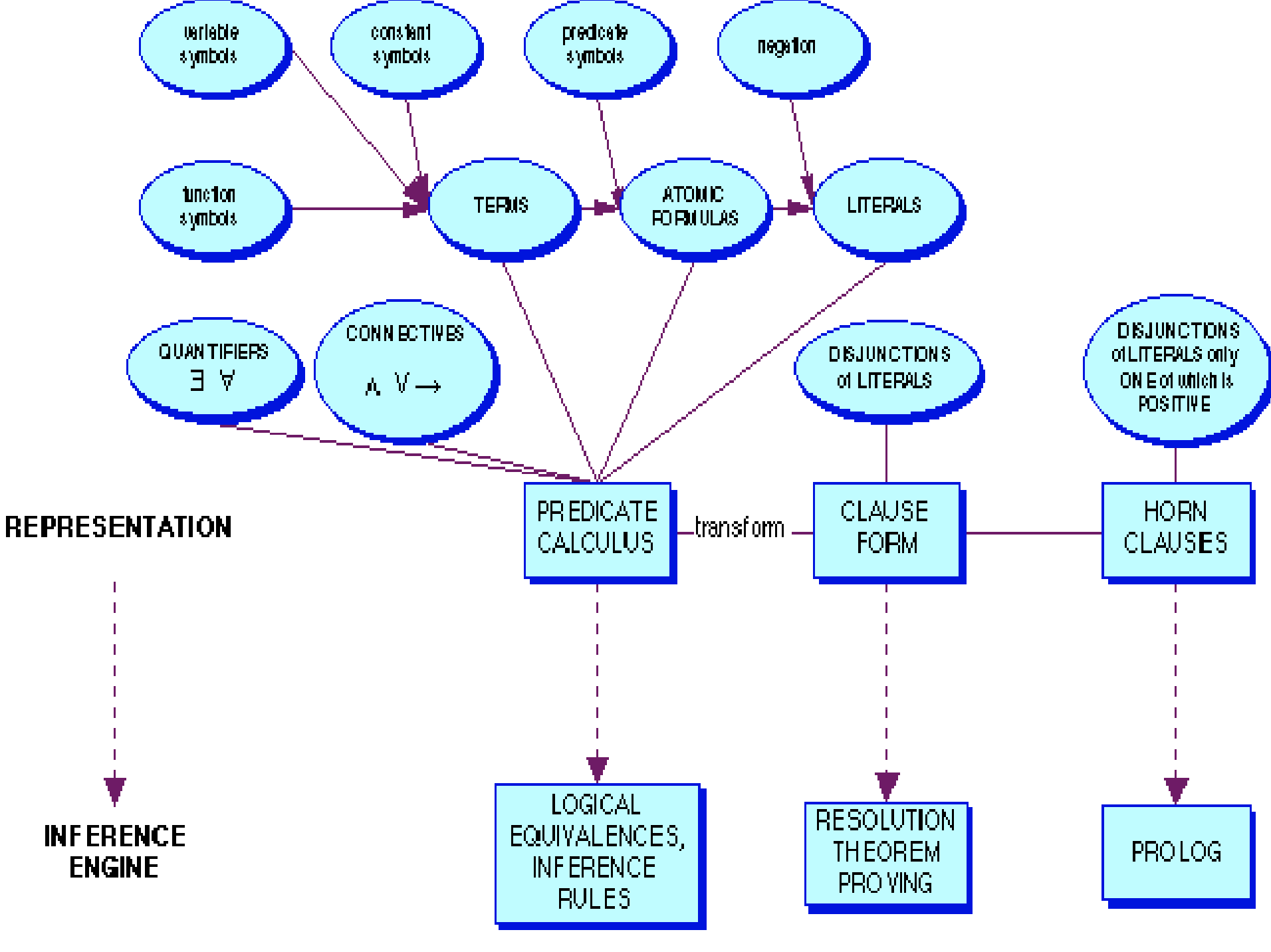
```
findall(X,G,_) :-
  asserta(found(mark)),
  call(G),
  assert(found(X)),
  fail.

findall(_,_,L) :-
  collectFound([],M),
  !,
  L=M.
```

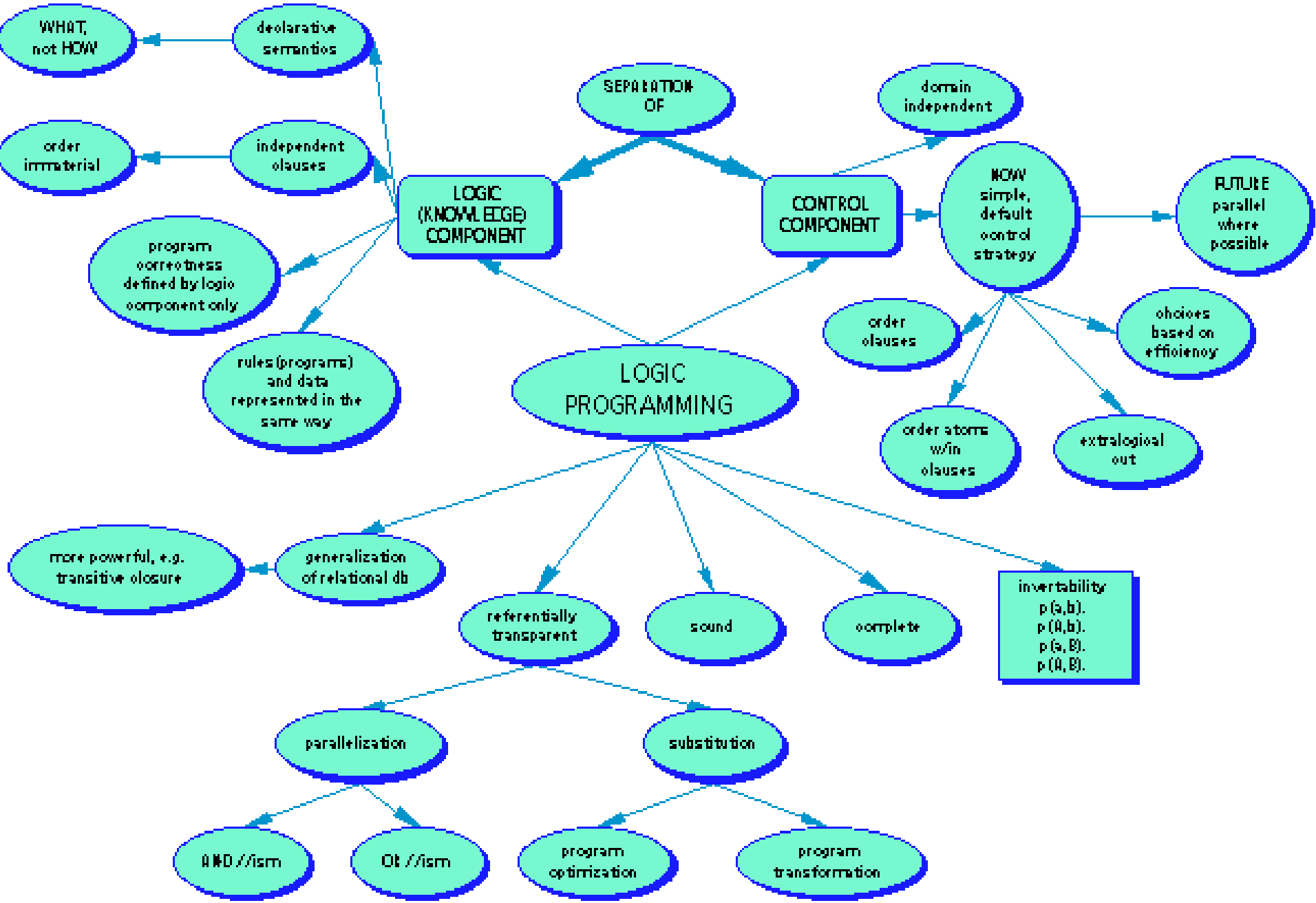
*Why this?*

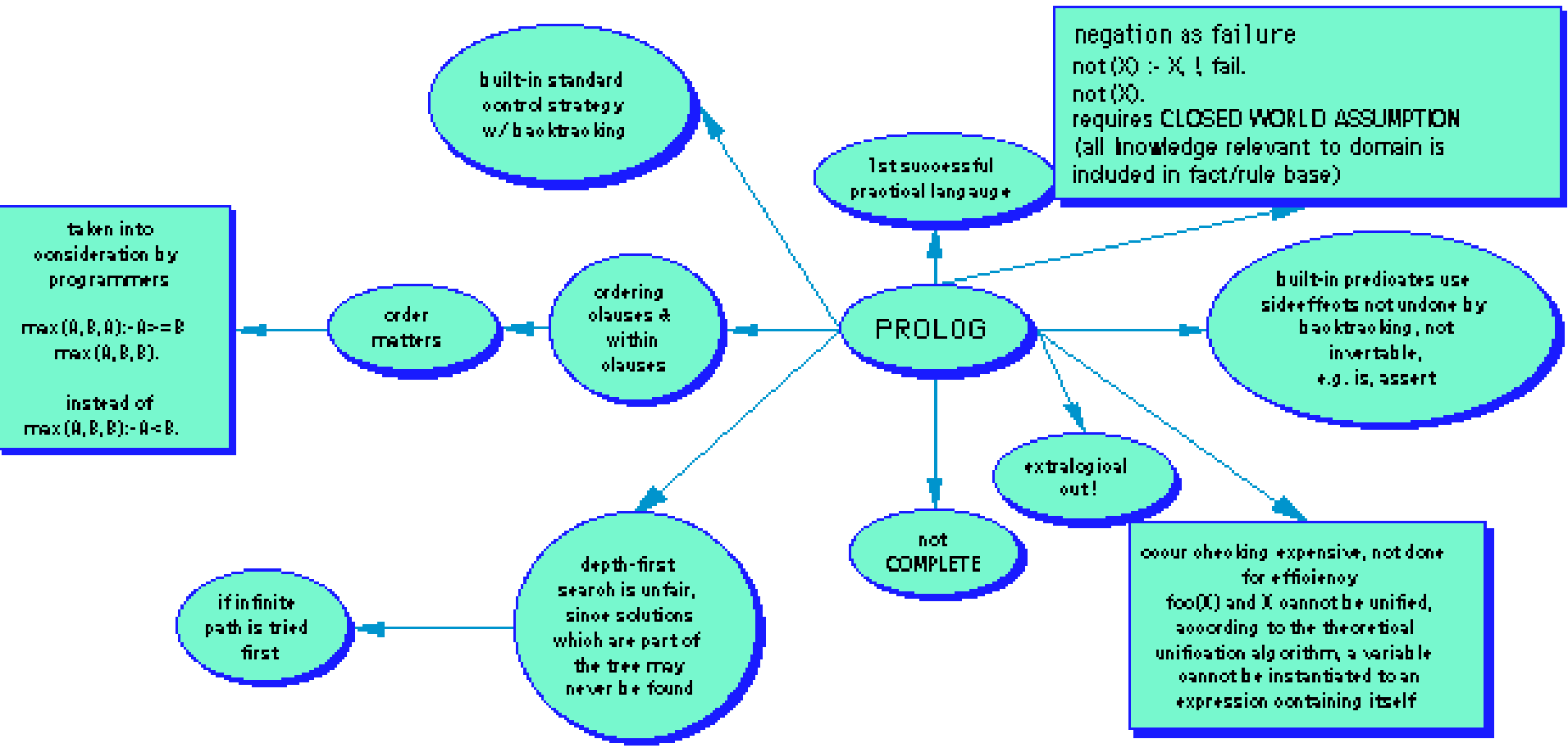
```
collectFound(S,L) :-
  getNext(X),
  !,
  collectFound([X|S],L).
collectFound(L,L).

getNext(X) :-
  retract(found(X)),
  !,
  \+(X=mark).
```

*And this?*

*And this?*


variable symbols
constant symbols
predicate symbols
negation
function symbols
TERMS
ATOMIC FORMULAS
LITERALS
QUANTIFIERS ∃ ∀
CONNECTIVES ∧ ∨ →
DISJUNCTIONS of LITERALS
DISJUNCTIONS of LITERALS only ONE of which is POSITIVE
REPRESENTATION
PREDICATE CALCULUS
transform
CLAUSE FORM
HORN CLAUSES
INFERENCE ENGINE
LOGICAL EQUIVALENCES, INFERENCE RULES
RESOLUTION THEOREM PROVING
PROLOG


LOGIC PROGRAMMING
SEPARATION OF
LOGIC (KNOWLEDGE) COMPONENT
CONTROL COMPONENT
WHAT, not HOW
declarative semantics
order immaterial
independent clauses
program correctness defined by logic component only
rules (programs) and data represented in the same way
domain independent
NOW simple, default control strategy
FUTURE parallel where possible
order clauses
choices based on efficiency
order atoms w/in clauses
extralogical out
more powerful, e.g. transitive closure
generalization of relational db
referentially transparent
sound
complete
invertability
p(a,b).
p(A,b).
p(a,B).
p(A,B).
parallelization
substitution
AND //ism
OR //ism
program optimization
program transformation


built-in standard control strategy w/ backtracking
negation as failure
not(X) :- X, !, fail.
not(X).
requires CLOSED WORLD ASSUMPTION
(all knowledge relevant to domain is included in fact/rule base)
1st successful practical language
taken into consideration by programmers
max(A,B,A):- A>=B
max(A,B,B).
instead of
max(A,B,B):- A<B.
order matters
ordering clauses & within clauses
PROLOG
built-in predicates use sideeffects not undone by backtracking, not invertable, e.g. is, assert
extralogical out!
occur checking expensive, not done for effiency
foo(X) and X cannot be unified, according to the theoretical unification algorithm, a variable cannot be instantiated to an expression containing itself
not COMPLETE
depth-first search is unfair, since solutions which are part of the tree may never be found
if infinite path is tried first