programming with quantified truth

programming with qualified truth

programming with constraints on integer domains

Declarative Programming

8: interesting loose ends

only to whet your appetite,
will **not** be asked on exam

implicit parallel evaluation

software engineering applications

Logic programming with quantified truth:

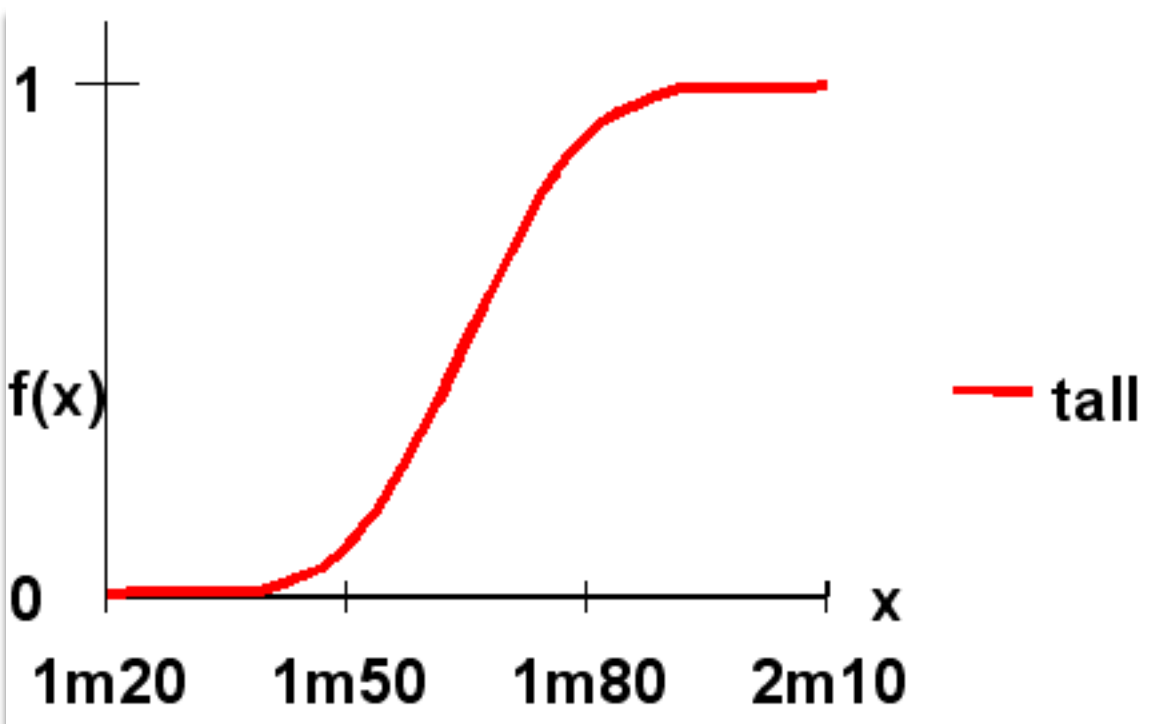
reasoning with vague (rather than incomplete) information

fuzzy set [Zadeh 1965]

characteristic function generalised to allow gradual membership

$$\mu_A : U \rightarrow [0,1]$$

$$\mu_A(x) = \begin{cases} 0 \leftrightarrow x \notin A \\ 1 \leftrightarrow x \in A \\ 0 < \alpha < 1 \leftrightarrow x \in A \text{ to the extent } \alpha \end{cases}$$



Logic programming with quantified truth:
operations on fuzzy sets

classical set-theoretic operations

- Intersection: $\mu_{A\cap B}(x) = min(\mu_A(x), \mu_B(x))$
- Union: $\mu_{A\cup B}(x) = max(\mu_A(x), \mu_B(x))$
- Complement: $\mu_{\bar{A}}(x) = 1 - \mu_A(x)$

original ones by Zadeh, later generalized

linguistic hedges

take a fuzzy set (e.g., set of tall people) and modify its membership function

modelling adverbs: very, somewhat, indeed

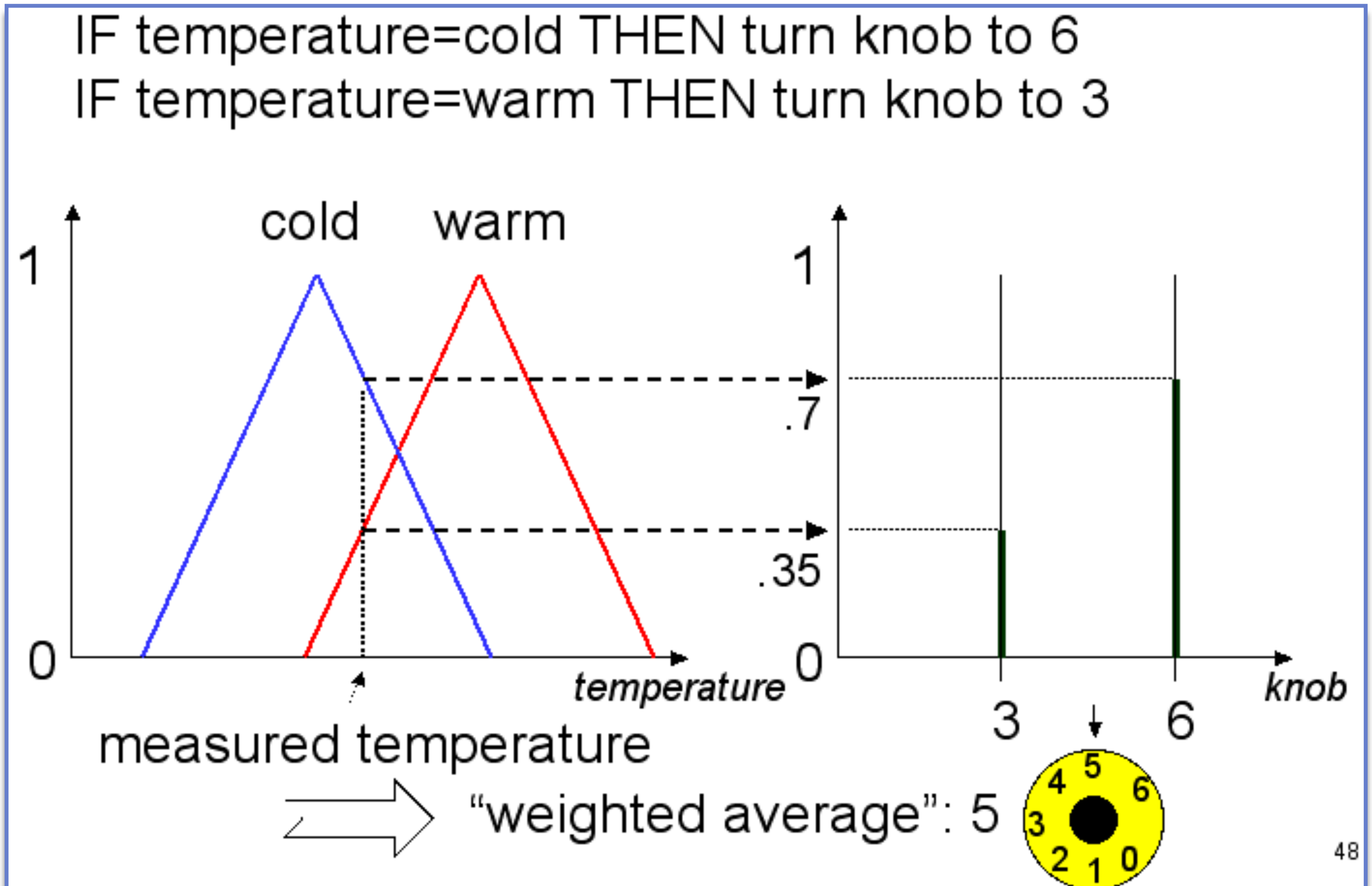
compositional rule of inference

premise	if X is A and Y is B then Z is C
fact	X is A' and Y is B'
consequence	Z is C'

Logic programming with quantified truth:

killer application: fuzzy process control

Logic programming with quantified truth:
killer application: fuzzy process control



easier and smoother operation than classical process control

Logic programming with quantified truth:
killer application: fuzzy process control

$rule_1$	if X is A_1 then Y is B_1
$rule_2$	if X is A_2 then Y is B_2
...	...
fact	X is A
consequence	Y is B

Designing a fuzzy control system generally consists of the following steps:

Fuzzification This is the basic step in which one has to determine appropriate fuzzy membership functions for the input and output fuzzy sets and specify the individual rules regulating the system.

Inference This step comprises the calculation of output values for each rule even when the premises match only partially with the given input.

Composition The output of the individual rules in the rule base can now be combined into a single conclusion.

Defuzzification The fuzzy conclusion obtained through inference and composition often has to be converted to a crisp value suited for driving the motor of an air conditioning system, for example.

Logic programming with quantified truth:

a meta-interpreter for a fuzzy logic programming language

many variations possible

LP with quantified truth

weighted logic rules

$q : c$ **if** $q_1, \ldots, q_n$ where $c \in]0,1]$

fuzzy resolution procedure

$T(q) = c * \min(T(q1), \ldots, T(qn))$

confidence in conclusion **q** given absolute truth of $q_1, \ldots, q_n$

similar to f-Prolog [1990:liu]

```
sold(flowers, 15).
attractive_packaging(chips) : 0.9.
well_advertised(chips) : 0.6.

popular_product(?product) if
  sold(?product, ?amount),
  ?amount > 10.

popular_product(?product) : 0.8 if
  attractive_packaging(?product),
  well_advertised(?product).
```

```
if popular_product(?p) : ?c
```

?p	?c
flowers	1
chips	min(0.9, 0.6)*0.8 = 0.48

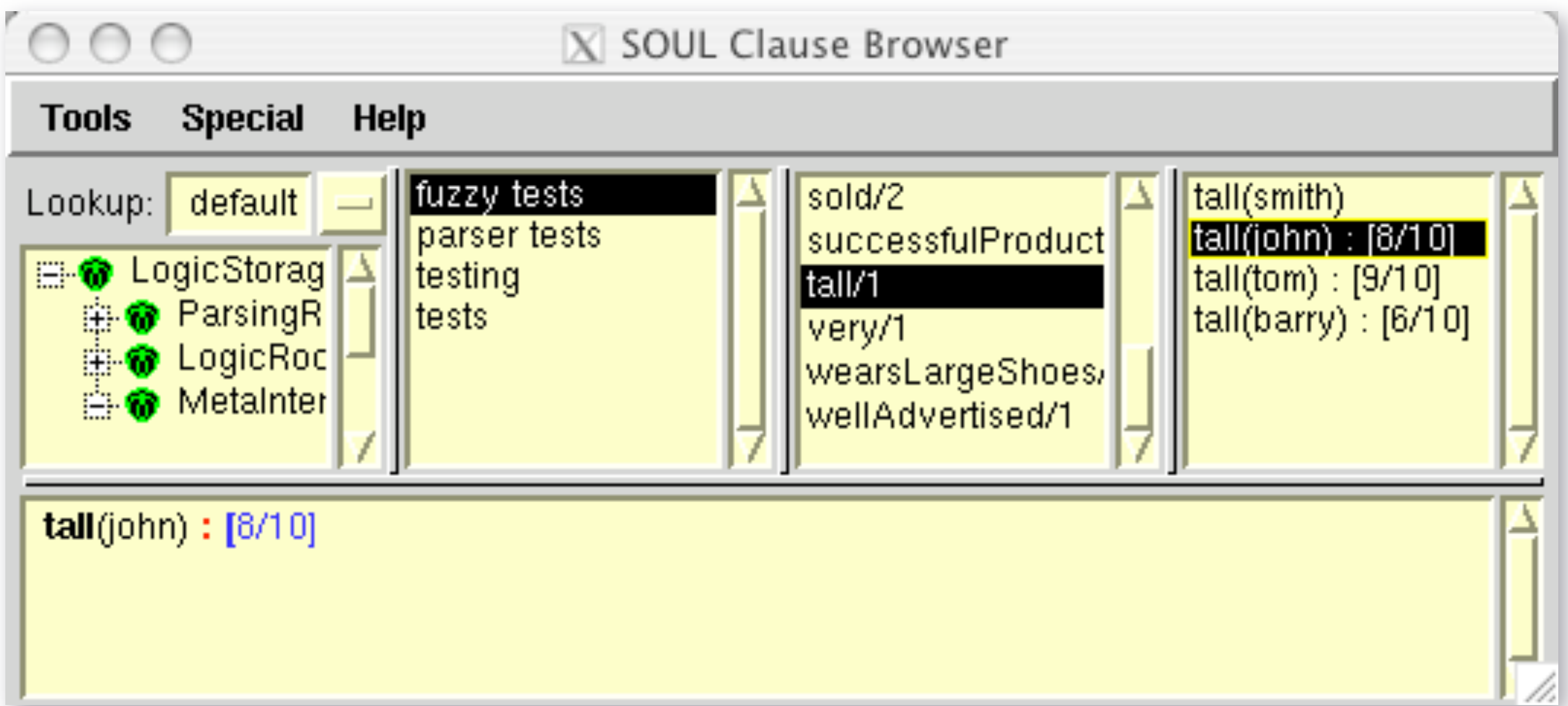
Logic programming with quantified truth:

a meta-interpreter for a fuzzy logic programming language

DEMO

Logic programming with quantified truth:

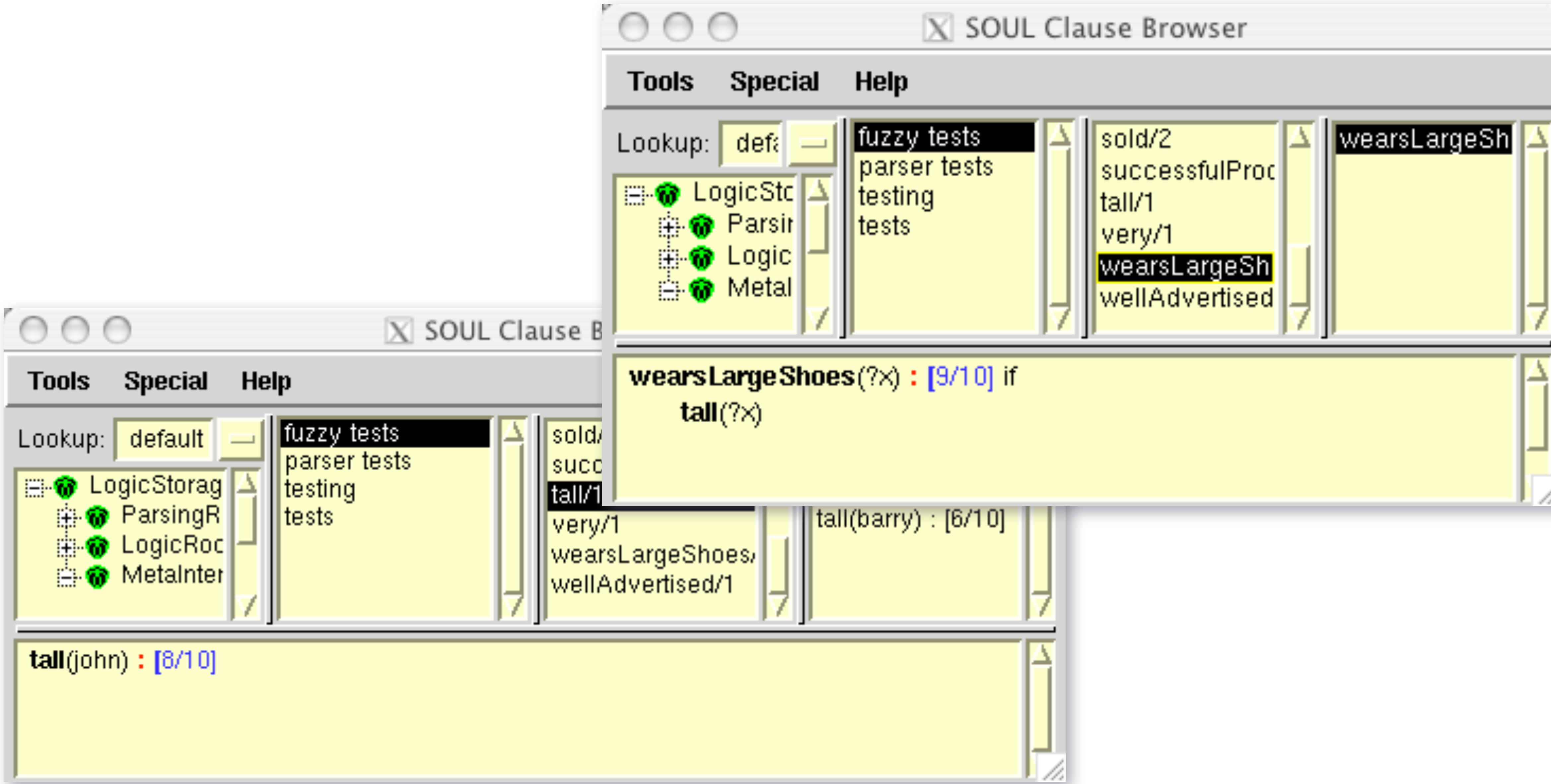
a meta-interpreter for a fuzzy logic programming language



DEMO

Logic programming with quantified truth:

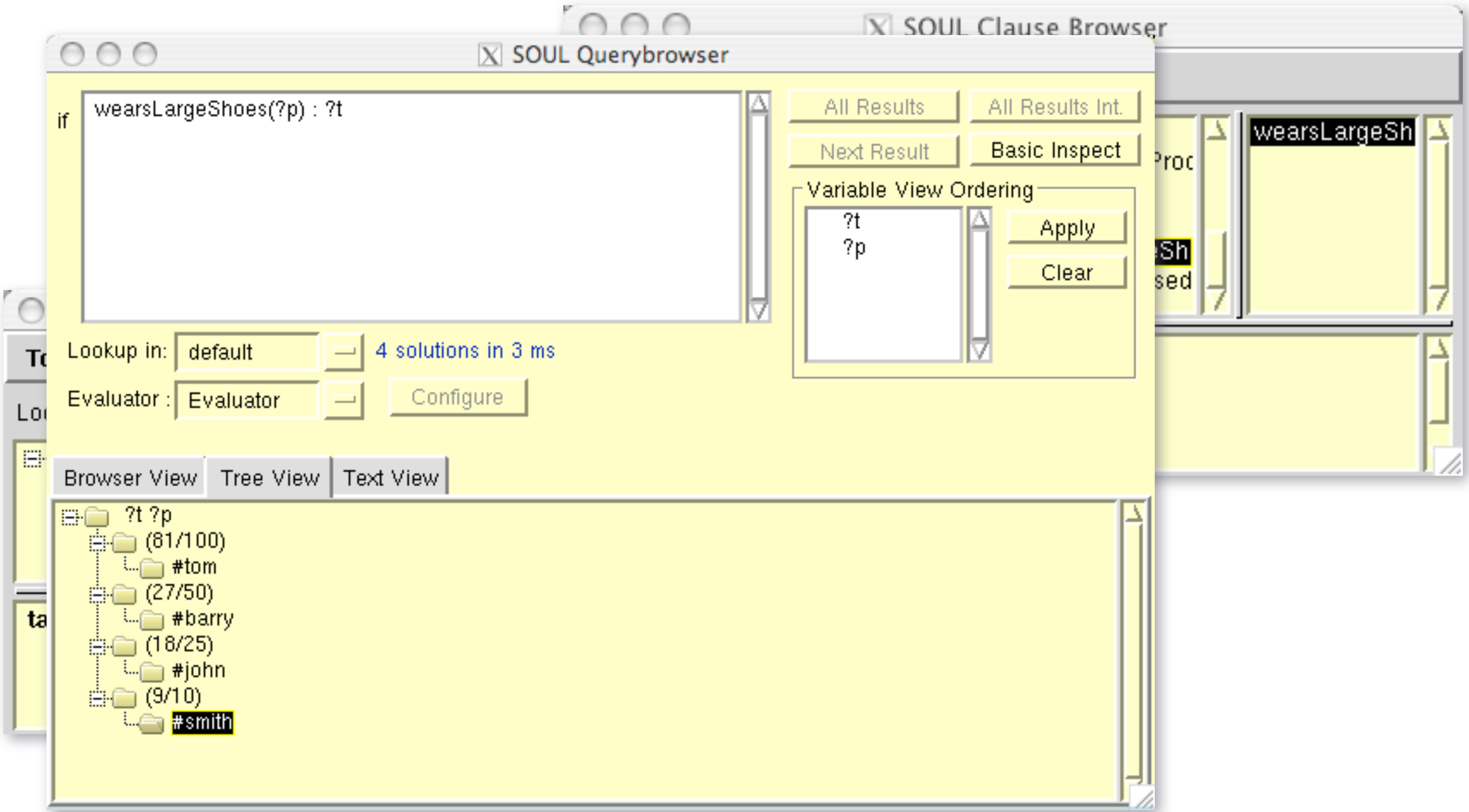
a meta-interpreter for a fuzzy logic programming language



DEMO

Logic programming with quantified truth:

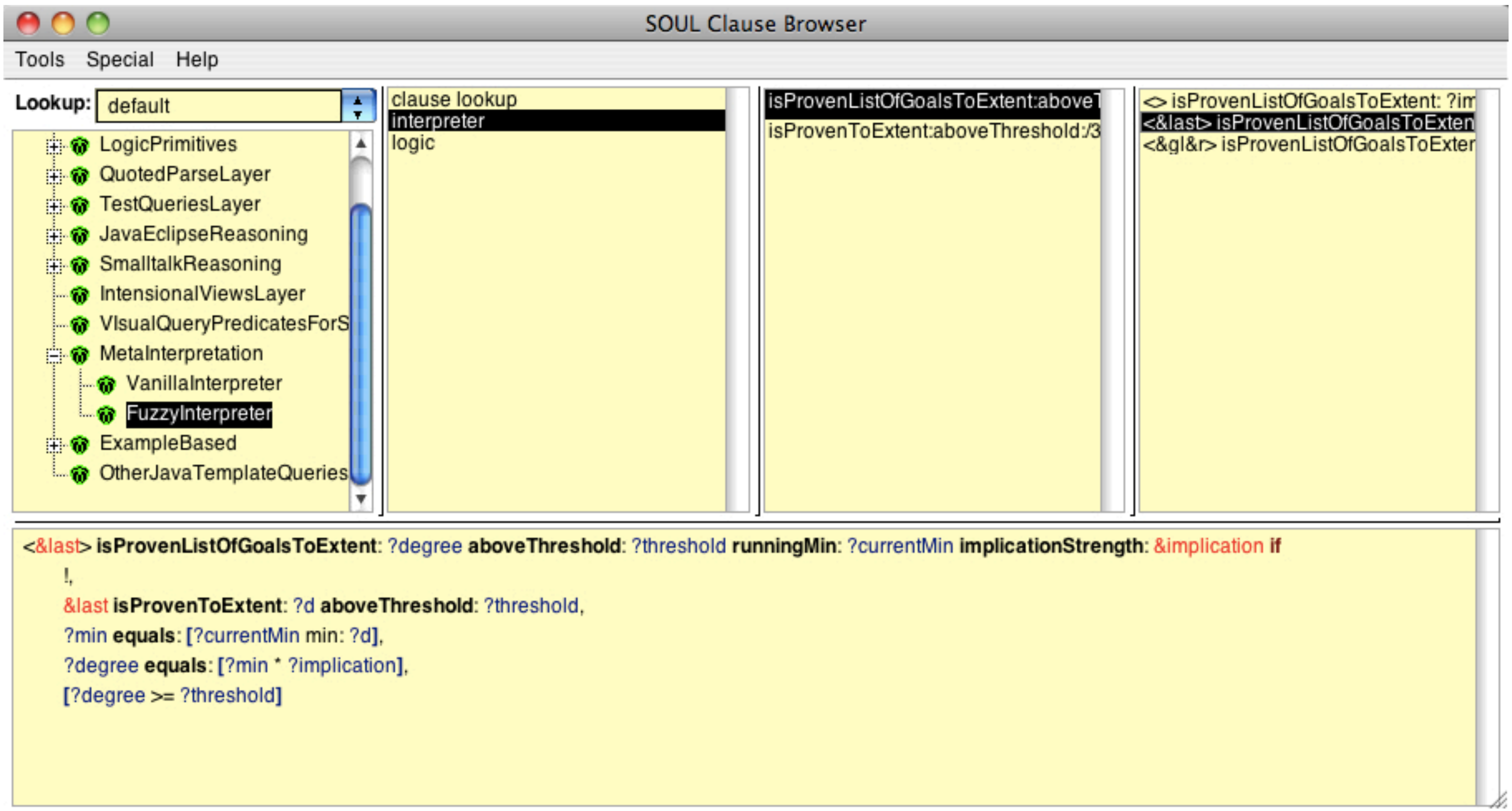
a meta-interpreter for a fuzzy logic programming language



DEMO

Logic programming with quantified truth:

a meta-interpreter for a fuzzy logic programming language



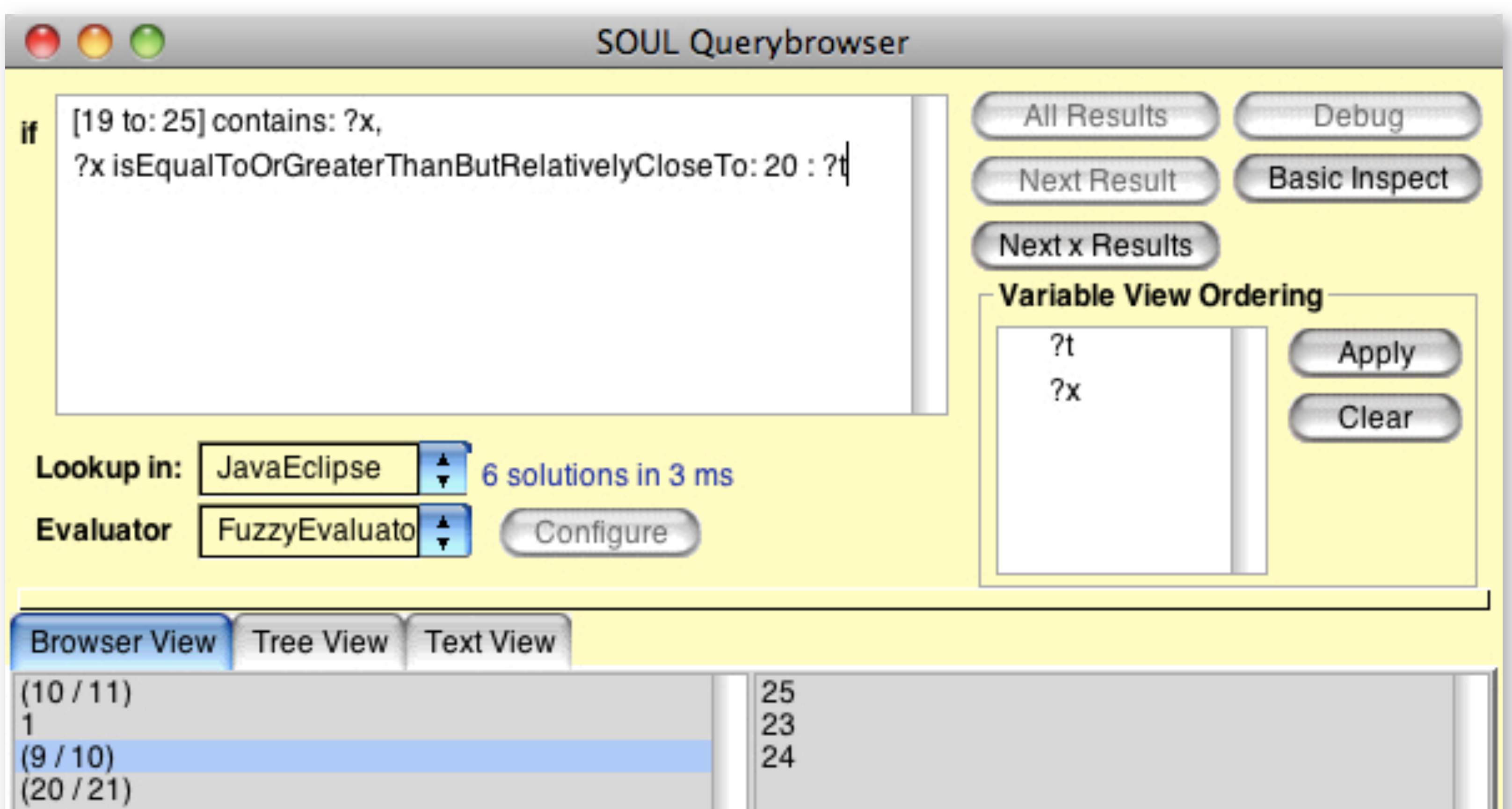
DEMO

Logic programming with quantified truth:

reifying the characteristic function of a fuzzy set

```
+?x isEqualToOrGreaterThanButRelativelyCloseTo: +?x.
+?x isEqualToOrGreaterThanButRelativelyCloseTo: +?y : ?c if
  [?x > ?y],
  ?c equals: [(?y / ?x) max: (9 / 10)]
```

associates a truth degree
[9,1[
with numbers ?x that are
greater than ?y, but do not
deviate more than 10% from ?y



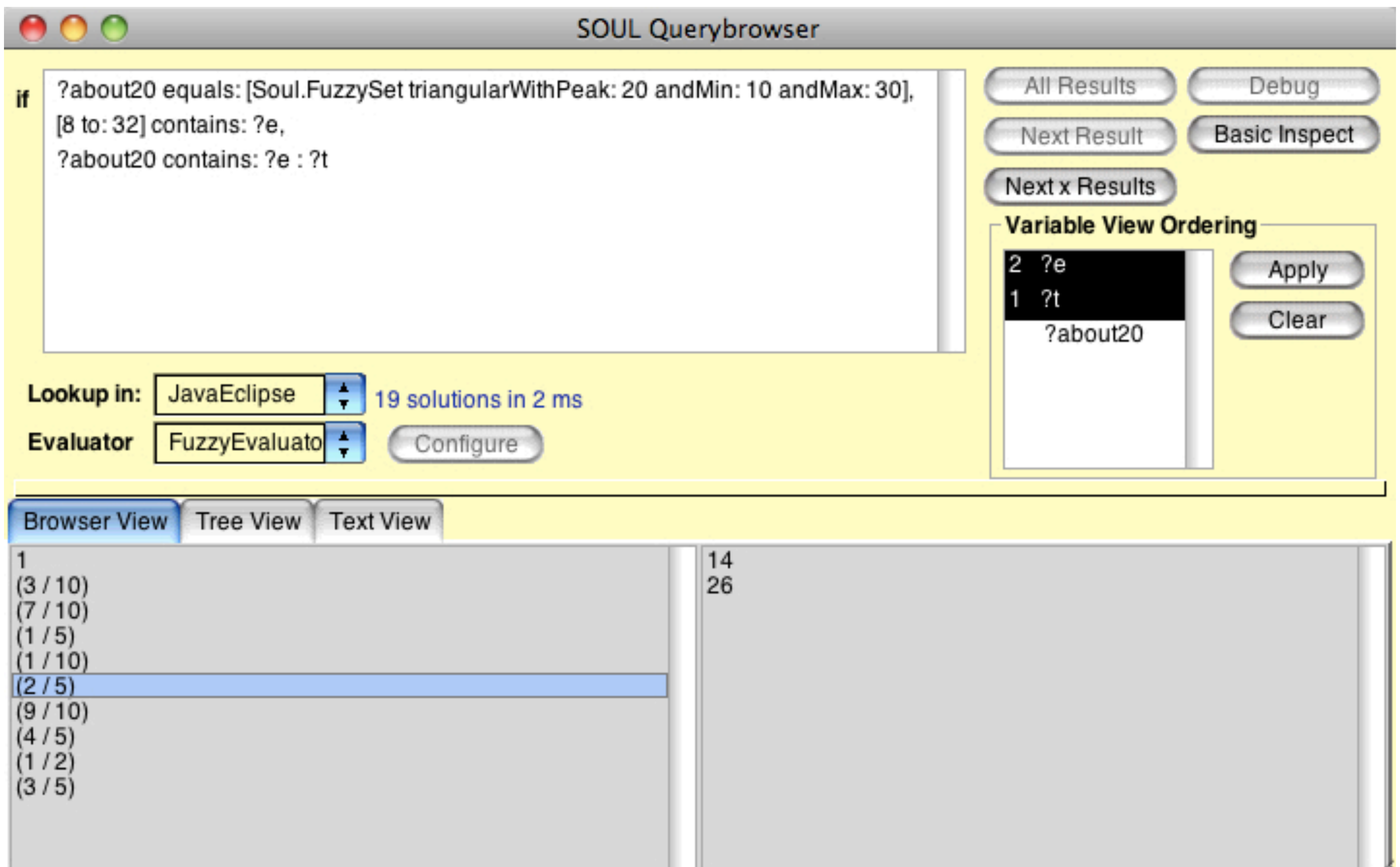
DEMO

Logic programming with quantified truth:

quantifying over the elements of a fuzzy set

```
+?c contains: +?e if
  [?c isKindOf: Soul.FuzzySet],
  [?c membershipDegreeOfElement: ?e]
```

additional contains:/2 clause for fuzzy sets implemented in Smalltalk



linearly models how close an element is to 20

$$\Delta(x,\alpha,\beta,\gamma) = \begin{cases} 0 & x<\alpha \\ (x-\alpha)/(\beta-\alpha) & \alpha\leq x\leq\beta \\ (\gamma-x)/(\gamma-\beta) & \beta\leq x\leq\gamma \\ 0 & x>\gamma \end{cases}$$

Logic programming with qualified truth:

an executable linear temporal logic (informally)

regular logic formulas qualified by temporal operators:

- $\Box$ (always)
- $\Diamond$ (sometimes)
- $\bullet$ (previous)
- $\circ$ (next).

evaluated against an implicit temporal context:

$\Box\phi$ is true if ϕ is true at all moments in time.

we will assume a finite, non-branching timeline for our example application: reasoning about execution traces of a program

Logic programming with qualified truth:

a meta-interpreter for finite linear temporal logic programming

```
solve(A) :-
  prove(A, 0).

prove(not(A), T) :-
  not(prove(A, T)).

prove(next(A), T) :-
  NT #= T + 1,
  prove(A, NT).
prove(next(C, A), T) :-
  C #> 0,
  NT #= T + C,
  prove(A, NT).

prove(previous(A), T) :-
  NT #= T - 1,
  prove(A, NT).
prove(previous(C, A), T) :-
  C #> 0,
  NT #= T - C,
  prove(A, NT).
```

the initial temporal context for all top-level formulas is the beginning of the timeline

next(A) holds if A holds at the next moment in time

next(C,A) holds if A holds C steps into the future (possibly a variable)

#> and friends impose constraints over integer domain: use_module(library(clpfd)).

Intermezzo:

constraint logic programming over integer domains

```
?- X #> 3.
X in 4..sup.

?- X #\= 20.
X in inf..19\/21..sup.

?- 2*X #= 10.
X = 5.

?- X*X #= 144.
X in -12\/12.

?- 4*X + 2*Y #= 24, X + Y #= 9, [X,Y] ins 0..sup.
X = 3,
Y = 6.

?- Vs = [X,Y,Z], Vs ins 1..3, all_different(Vs), X = 1, Y #\= 2.
Vs = [1, 3, 2],
X = 1,
Y = 3,
Z = 2.
```

X in integer domain

X in union of two domains

list of variables on the left is in the domain on the right

ensures elements are assigned different values from domain

Intermezzo:

constraint logic programming over integer domains
SEND + MORE = MONEY

```
puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-
        Vars = [S,E,N,D,M,O,R,Y],
        Vars ins 0..9,
        all_different(Vars),
        S*1000 + E*100 + N*10 + D +
             M*1000 + O*100 + R*10 + E #=
             M*10000 + O*1000 + N*100 + E*10 + Y,
        M #\= 0, S #\= 0.
```

```
?- puzzle(As+Bs=Cs).
As = [9, _G10107, _G10110, _G10113],
Bs = [1, 0, _G10128, _G10107],
Cs = [1, 0, _G10110, _G10107, _G10152],
_G10107 in 4..7,
1000*9+91*_G10107+ -90*_G10110+_G10113+ -9000*1+ -900*0+10*_G10128+ -1*_G10152#=0,
all_different([_G10107, _G10110, _G10113, _G10128, _G10152, 0, 1, 9]),
_G10110 in 5..8,
_G10113 in 2..8,
_G10128 in 2..8,
_G10152 in 2..8.
```

deduced more stringent constraints for variables

Intermezzo:

constraint logic programming over integer domains
SEND + MORE = MONEY

```
puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-
  Vars = [S,E,N,D,M,O,R,Y],
  Vars ins 0..9,
  all_different(Vars),
  S*1000 + E*100 + N*10 + D +
    M*1000 + O*100 + R*10 + E #=
    M*10000 + O*1000 + N*100 + E*10 + Y,
  M #\= 0, S #\= 0.
```

```
?- puzzle(As+Bs=Cs), label(As).
As = [9, 5, 6, 7],
Bs = [1, 0, 8, 5],
Cs = [1, 0, 6, 5, 2] ;
false.
```

labeling a domain variable systematically tries out values for it until it is ground

```
?- puzzle(As+Bs=Cs).
As = [9, _G10107, _G10110, _G10113],
Bs = [1, 0, _G10128, _G10107],
Cs = [1, 0, _G10110, _G10107, _G10152],
_G10107 in 4..7,
1000*9+91*_G10107+ -90*_G10110+_G10113+ -9000*1+ -900*0+10*_G10128+ -1*_G10152#=0,
all_different([_G10107, _G10110, _G10113, _G10128, _G10152, 0, 1, 9]),
_G10110 in 5..8,
_G10113 in 2..8,
_G10128 in 2..8,
_G10152 in 2..8.
```

deduced more stringent constraints for variables

Logic programming with qualified truth:

a meta-interpreter for finite linear temporal logic programming

```
prove(sometime(C, A), T) :-
    C#>=0,
    bot(Bot),
    eot(Tot),
    NT in Bot..Tot,
    NT #>= T,
    NT #=< T+C,
    prove(A, NT).
prove(sometime(C,A), T) :-
    C #=< 0,
    bot(Bot),
    eot(Tot),
    NT in Bot..Tot,
    NT #>= T + C,
    NT #=< T,
    prove(A, NT).
prove(sometime(A), _) :-
    bot(Bot),
    eot(Tot),
    C in Bot..Tot,
    prove(A, C).
```

A holds sometime between now and C steps in the future

A holds sometime between now and C steps in the past

A holds somewhere on the timeline

similar for always

Logic programming with qualified truth:

example application: reasoning about execution traces

(a) observed behavior

```
event(0,init).
event(1,push(10,1)).
event(2,push(20,2)).
event(3,push(30,3)).
event(4,pop(20,2)).
```

Execute while intercepting high-level events

(b) source code

```
int *stack;
int top;
void init(int size) {
  top = 0;
  stack = malloc(size*sizeof(int));
}
void push(int element) {
  stack[top++]=element;
}
#define pop() stack[--top];
```

verified against

(c) documented behavior

```
behavioralModel :-
  until(stackInitialized, ¬stackUsed),
  □(when(push(S) ∧ •stackOperation(S1), S is S1 + 1)),
  □(when(pop(S) ∧ •stackOperation(S1), S is S1 - 1)).

stackInitialized(S) :- init(S).
stackUsed(S) :- push(S).
stackUsed(S) :- pop(S).
stackOperation(S) :- stackUsed(S).
stackOperation(S) :- stackInitialized(S).

push(S) :- event(push(_,S)).
 pop(S) :- event(pop(_,S)).
init(0) :- event(init).
```

(d) high-level events specification

```
intercept(after, stackPopOperation,
  event(time, pop(stackTop, stackSize))).
intercept(after, stackPushOperation,
  event(time, push(stackTop, stackSize))).
intercept(before, stackInitOperation,
  event(time, init)).
```

specific for this application

(e) application-specific instances

```
stackPushOperation(Construct,Path) :-
  functionCallHasName(Construct, 'push').
stackPopOperation(Construct,Path) :-
  macroCallHasName(Construct, 'pop').
stackInitOperation(Construct,Path) :-
  functionCallHasName(Construct, 'init').
```

(f) associated run-time values

```
keyword(stackSize, 'log("%i", top);').
keyword(time, 'log("%i", TIME++);').
keyword(stackTop,'log("%i",stack[top-1]);').
```

Logic programming with qualified truth:

example application: reasoning about execution traces

(a) observed behavior

```
..
event(60,cntEntered('ASG',13..1,['ASG','print', 'exit'])).
event(61,cntExited('ASG',13..1,['print', 'exit'])).
..
```

Execute source code while intercepting

(b) documentation as present in the source code

```
/*--------------------------------------------------------*/
/* ASS                                                    */
/* expr-stack: [... ... ... ... DCT VAL] ->               */
/*             [... ... ... ... ... VAL]                  */
/* cont-stack: [... ... ... ... ... ASS] ->               */
/*             [... ... ... ... ... ...]                  */
/*--------------------------------------------------------*/
static _NIL_TYPE_ ASG(_NIL_TYPE_)
 { ... }
```

verified against

(c) documented behavior

```
cntDocumented('ASG',['ASG'|R],R).
cntDocumented('REF',['REF'|R],['REF','APL'|R]).
...

behavioralModel :-
  □(when(cntExecuted(Name,Before,After),
         cntDocumented(Name,Before,After))).

cntExecuted(Name,StackBefore,StackAfter) :-
  cntExited(Name,_,StackAfter),
  •^t cntEntered(Name,_,StackBefore).
```

(d) high-level events specification

```
intercept(before,continuationEntry,
 event(time,cntEntered(cntName,cntPtr,cntStack))).

intercept(after,continuationExit,
 event(time,cntExited(cntName,cntPtr,cntStack))).
```

specific for this application

(e) application-specific instances

```
continuationEntry(Construct,Path) :-
  inContinuation(Construct,Path),
  functionEntry(Construct,Path).
continuationExit(Construct,Path) :-
  inContinuation(Construct,Path),
  functionExit(Construct,Path).

continuation(Construct) :-
  isFunctionDefinition(Construct),
  expressionIn(Construct,Expression,_),
  picoStack(Expression).
```

(f) associated run-time values

```
keyword(cntName,C,P,Expansion) :-
  continuationName(C,P,Name),
  concat(['log("',Name,'");'],Expansion).
```

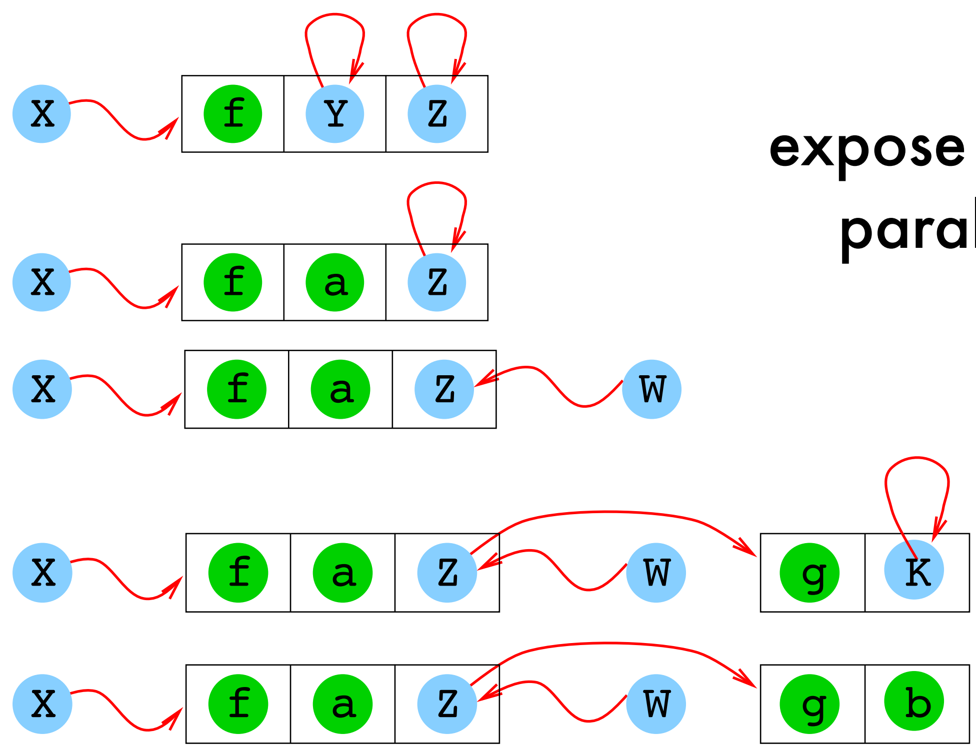
Non-standard evaluation strategies:
a taste of implicit parallel evaluation

multi-core revolution

speed up sequential programs

should be easier for declarative programs

```
main :- X = f(Y,Z),
        Y = a,
        W = Z,
        W = g(K),
        X = f(a,g(b)).
```



expose inherent parallelism

formal foundation

relatively pure

BUT also complex datastructures with pointers ...
imagine executing these goals in parallel!

[http://clip.dia.fi.upm.es/~logalg/slides/PS/A_par.pdf]

Non-standard evaluation strategies:

a taste of implicit parallel evaluation

```
while (Query not empty) do
  select_literal B from Query                         And-Parallelism
  repeat
      select_clause (H :- Body) from Program          Or-Parallelism

  until ( unify(H,B)  or no clauses left)             Unification Parallelism
  if (no clauses left) then FAIL
  else
      σ = MostGeneralUnifier(H,B)
      Query = ((Query \ {B}) ∪ Body)σ
  endif
endwhile
```

not trivial: goals typically depend on each other
(data and control dependency),
workers need to be synchronized

correctness (same solutions as sequential)
efficiency (no slowdown, speedup)

[http://www.cs.nmsu.edu/~ipivkina/Compulog/pontelli.pdf]

Non-standard evaluation strategies:
a taste of implicit parallel evaluation - or-parallelism

```
p(a).
p(b).
?- p(X).
```

there is no dependency between the clauses implementing p/1

execute different branches at choice point simultaneously

relevant for search problems, generate-and-test

much easier to implement than and-parallelism

issue: maintaining a different environment per branch efficiently(e.g., sharing, copying, ...)

typical architecture:

- set of workers, each a full interpreter
- scheduler assigns unexplored branches to idle workers

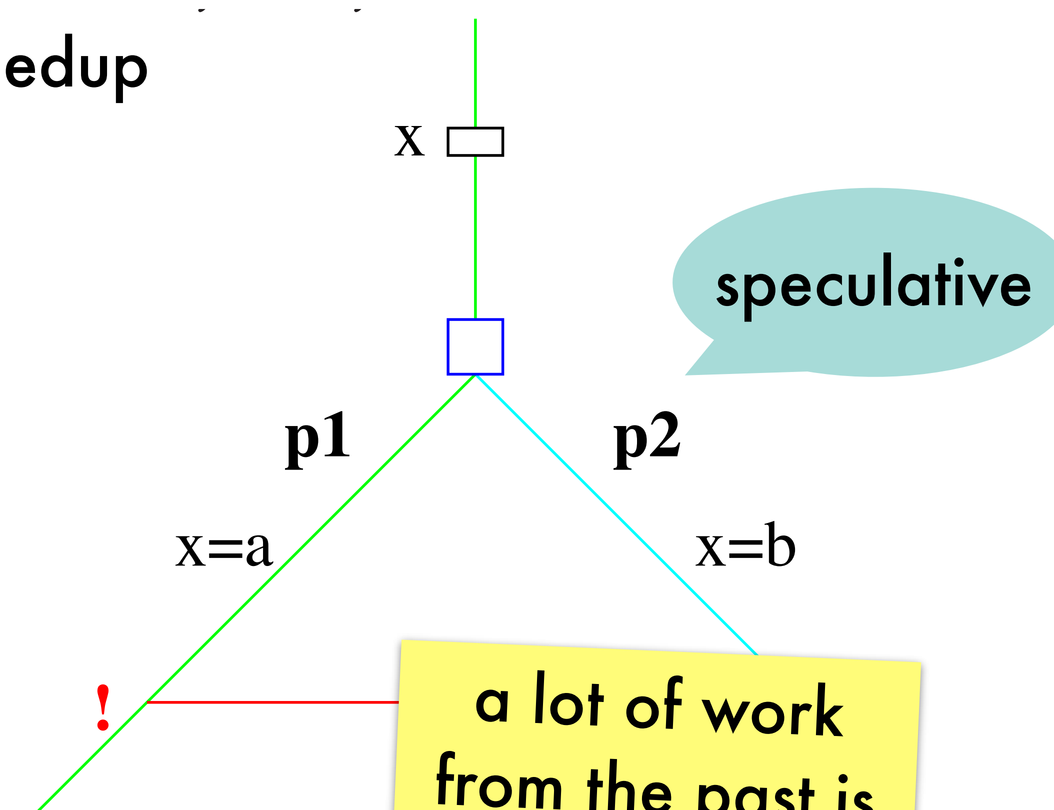
[http://clip.dia.fi.upm.es/~logalg/slides/PS/A_par.pdf]

Non-standard evaluation strategies:

a taste of implicit parallel evaluation - or-parallelism

speculative work should be avoided to gain speedup

```
..., p(X), ...
p(X) :- ..., X=a, ..., !, ...
p(X) :- ..., X=b, ...
```



left-based scheduling, immediate killing on cut

a lot of work from the past is relevant again, BUT: distributed vs shared memory architectures, caching

```
main :- l, s.

:- parallel l/0.
l :- large_work_a.
l :- large_work_b.

:- parallel s/0.
s :- small_work_a.
s :- small_work_b.
```

avoid incurring an overhead from fine-grained parallelism

[http://clip.dia.fi.upm.es/~logalg/slides/PS/A_par.pdf]

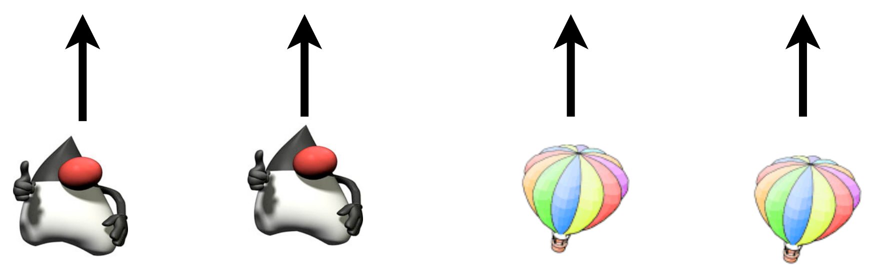
Logic programming in software engineering:
SOUL - symbiosis

symbiosis with base program languages

```
if ?c isCompilationUnit,
   [?c types size > 1]
```

ordinary term

symbiosis term



instance method method instance

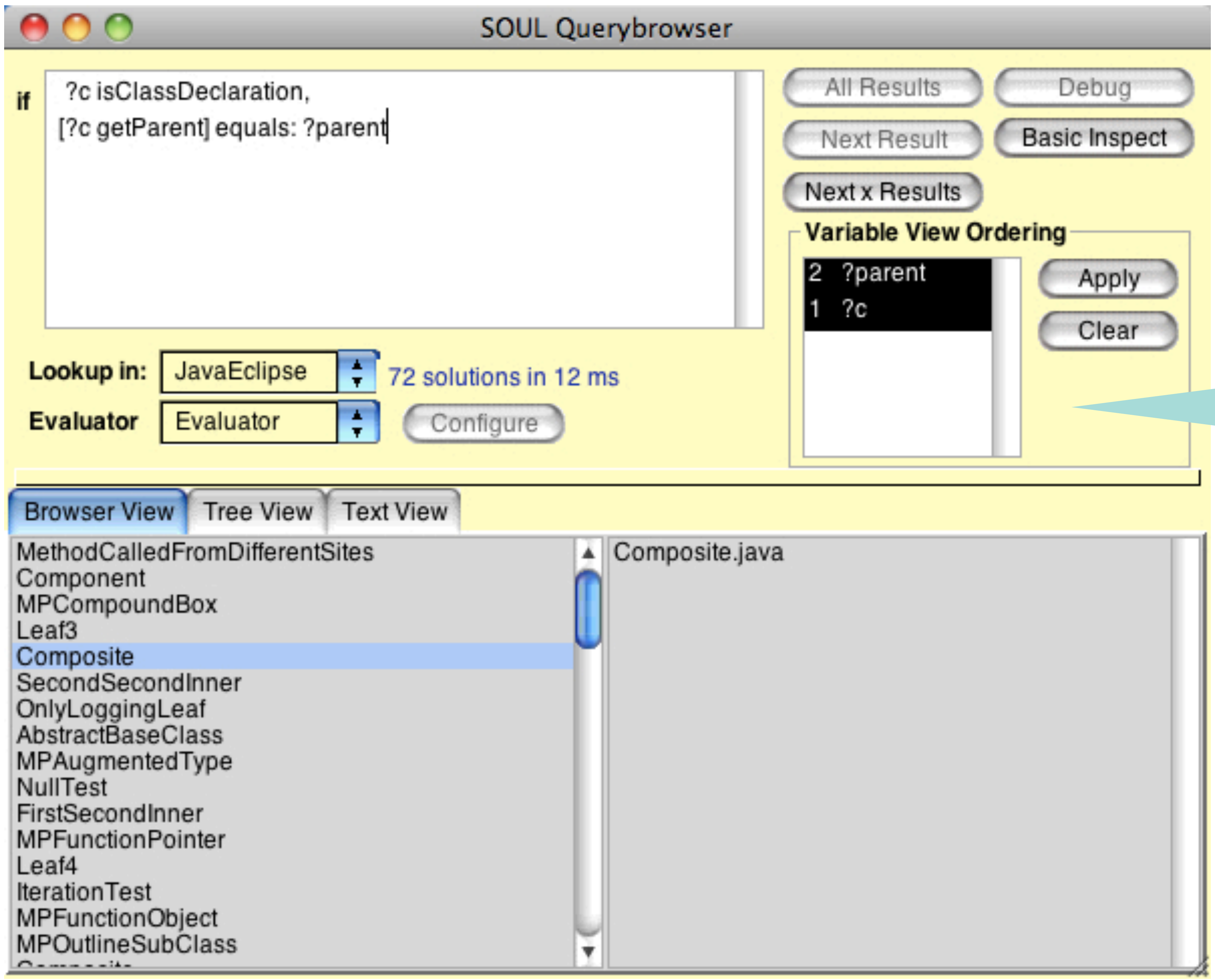
base program not **reified as logic facts**

changes are immediately reflected

query results easily perused by existing IDE’s

Logic programming in software engineering:

SOUL - symbiosis - demo



nice, but true power of logic programming comes not only from backtracking, but also from the ability to unify with a user-provided compound term to quickly select objects one is interested in

hold that thought

hmm .. strange:
the method's name (a Java Object) is unified with a compound term?

```
if ?m methodDeclarationHasName: ?n,
   ?n equals: simpleName(?identifier)
```

```
if ?m methodDeclarationHasName: simpleName(?identifier)
```

Logic programming in software engineering:

SOUL - symbiosis - demo

all subclasses of presentation.Component
should define a method acceptVisitor(ComponentVisitor)
that invokes System.out.println(String) before
double dispatching to the argument

```
public class PrototypicalLeaf extends Component {
   public void acceptVisitor(ComponentVisitor v) {
      System.out.println("Prototypical.");
      v.visitPrototypicalLeaf(this);
   }
}
```

Logic programming in software engineering:

SOUL - symbiosis - demo

```
?type isTypeWithFullyQualifiedName: ['presentation.Component'],
?class inClassHierarchyOfType: ?type,
not(?class classDeclarationHasName: simpleName(['Composite'])),
?class definesMethod: ?m,

?m methodDeclarationHasName: simpleName(['acceptVisitor']),
?m methodDeclarationHasParameters: nodeList(<?p>),
?p singleVariableDeclarationHasName: simpleName(?id),
?m methodDeclarationHasBody: ?body,

?body equals: block(nodeList(<expressionStatement(?log),expressionStatement(?dd)>)),
or(?so equals: qualifiedName(simpleName(['System']),simpleName(['out'])),
   ?so equals: fieldAccess(simpleName(['System']),simpleName(['out']))),
?log equals: methodInvocation(?so,?,simpleName(['println']),nodeList(<?string>)),
?dd equals: methodInvocation(simpleName(?id),?,?,nodeList(<thisExpression([nil])>))
```

yuk .. not as declarative as advertised!

and I have to do this for all implementation variants?

Logic programming in software engineering:

SOUL - code templates

integrate concrete syntax of base program

```
if jtStatement(?s) {
  while(?iterator.hasNext()) {
      ?collection.add(?element);
  }
},
jtExpression(?iterator){?collection.iterator()}
```

resolved by existential queries on control-flow graph

is add(Object) ever invoked in the control-flow of a while-statement?

Logic programming in software engineering:

SOUL - code templates - demo

Logic programming in software engineering:

SOUL - code templates - demo

```
jtClassDeclaration(?class,?interpretation){
    class !Composite extends* presentation.Component {
        ?modList ?type acceptVisitor(?t ?p) {
            System.out.println(?string);
            ?p.?m(this);
        }
    }
}
```

```java
public class SuperLogLeaf extends OnlyLoggingLeaf
{
    public void acceptVisitor(ComponentVisitor v) {
        super.acceptVisitor(v);
        v.visitSuperLogLeaf(this);
    }
}
```

vs

```
?type isTypeWithFullyQualifiedName: ['presentation.Component'],
?class inClassHierarchyOfType: ?type,
not(?class classDeclarationHasName: simpleName(['Composite'])),
?class definesMethod: ?m,

?m methodDeclarationHasName: simpleName(['acceptVisitor']),
?m methodDeclarationHasParameters: nodeList(<?p>),
?p singleVariableDeclarationHasName: simpleName(?id),
?m methodDeclarationHasBody: ?body,

?body equals: block(nodeList(<expressionStatement(?log),expressionStatement(?dd)>)),
or(?so equals: qualifiedName(simpleName(['System']),simpleName(['out'])),
   ?so equals: fieldAccess(simpleName(['System']),simpleName(['out']))),
?log equals: methodInvocation(?so,?,simpleName(['println']),nodeList(<?string>)),
?dd equals: methodInvocation(simpleName(?id),?,?,nodeList(<thisExpression([nil])>))
```

Logic programming in software engineering:

SOUL - code templates - demo

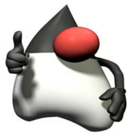
but still not in query results:

```java
public class MustAliasLeaf extends Component {
	public void acceptVisitor(ComponentVisitor v) {
		System.out.println("Must alias.");
		Component temp = this;
		v.visitMustAliasLeaf(temp);
	}
}
```

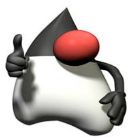
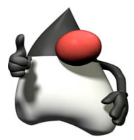
```java
public class MayAliasLeaf extends Component {
	public Object m(Object o) {
		if(getInput() % 2 == 0)
		   return o;
		else
		   return new MayAliasLeaf();
	}

	public void acceptVisitor(ComponentVisitor v) {
		System.out.println("May alias.");
		v.visitMayAliasLeaf((MayAliasLeaf)m(this));
	}
}
```

Logic programming in software engineering:

SOUL - domain-specific unification

 **instance vs compound term**

easily identify elements of interest

 **instance vs**  **instance**

incorporates static analyses: ensures query conciseness & correctness

semantic analysis
correct application of scoping rules, name resolution

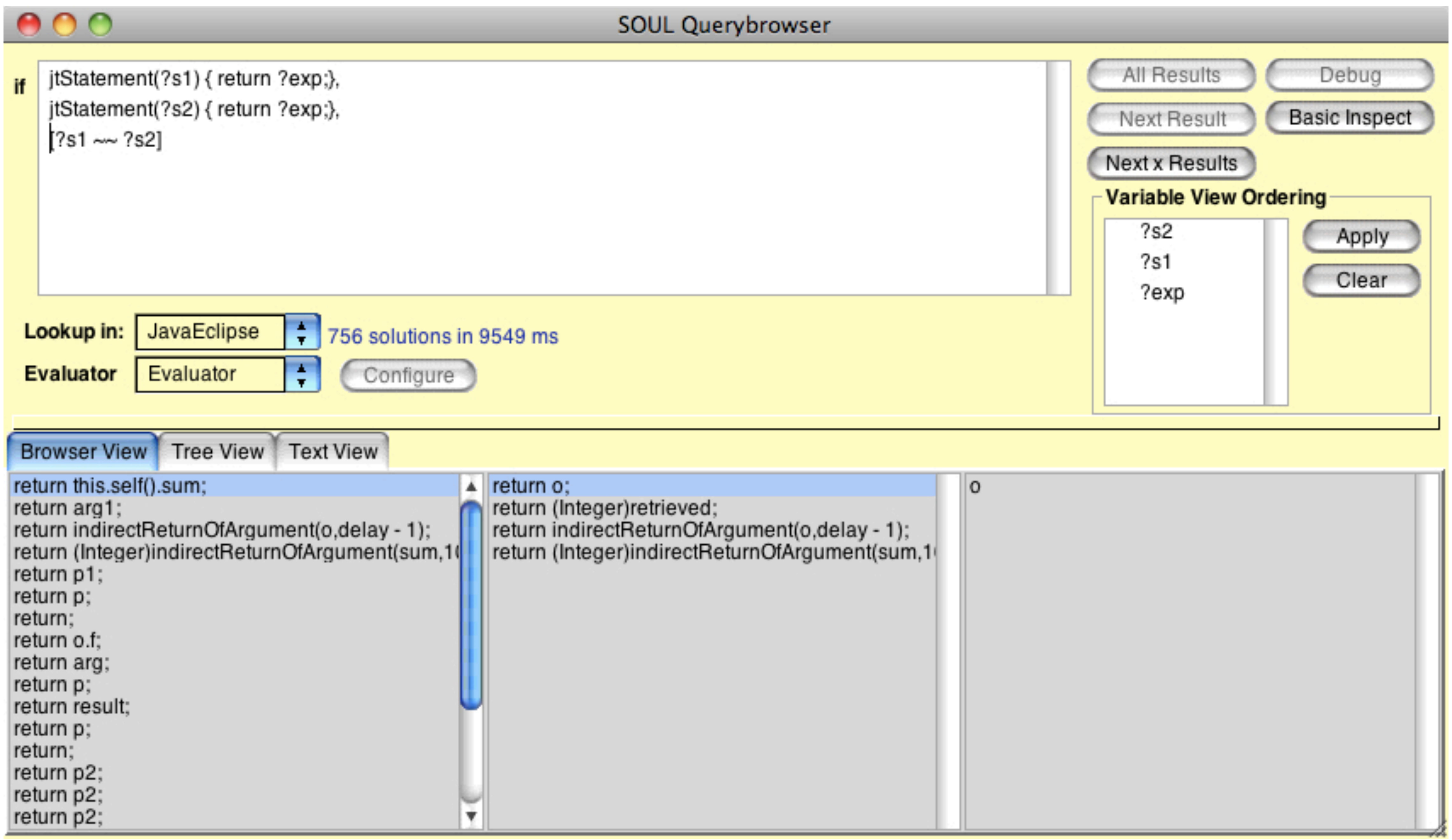
points-to analysis
tolerance for syntactically differing expressions

can the value on which hasNext() is invoked alias the iterator of the collection to which add is invoked?

```
if jtStatement(?s) {
  while(?iterator.hasNext()) {
       ?collection.add(?element);
  }
},
jtExpression(?iterator){?collection.iterator()}
```

never, in at least one or in all possible executions
-> propagate this knowledge using **logic of quantified truth**

Logic programming in software engineering:
SOUL - domain-specific unification - demo



Logic programming in software engineering:
SOUL - domain-specific unification - demo

```
jtClassDeclaration(?class,?interpretation){
    class !Composite extends* presentation.Component {
        ?modList ?type acceptVisitor(?t ?p) {
            System.out.println(?string);
            ?p.?m(this);
        }
    }
}
```

Table View   Text Report

Tuples	1(1680 ms)
class -> PrototypicalLeaf	0.9
class -> MayAliasLeaf	0.36
class -> SuperLogLeaf	0.72
class -> MustAliasLeaf	0.648

0.11