

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

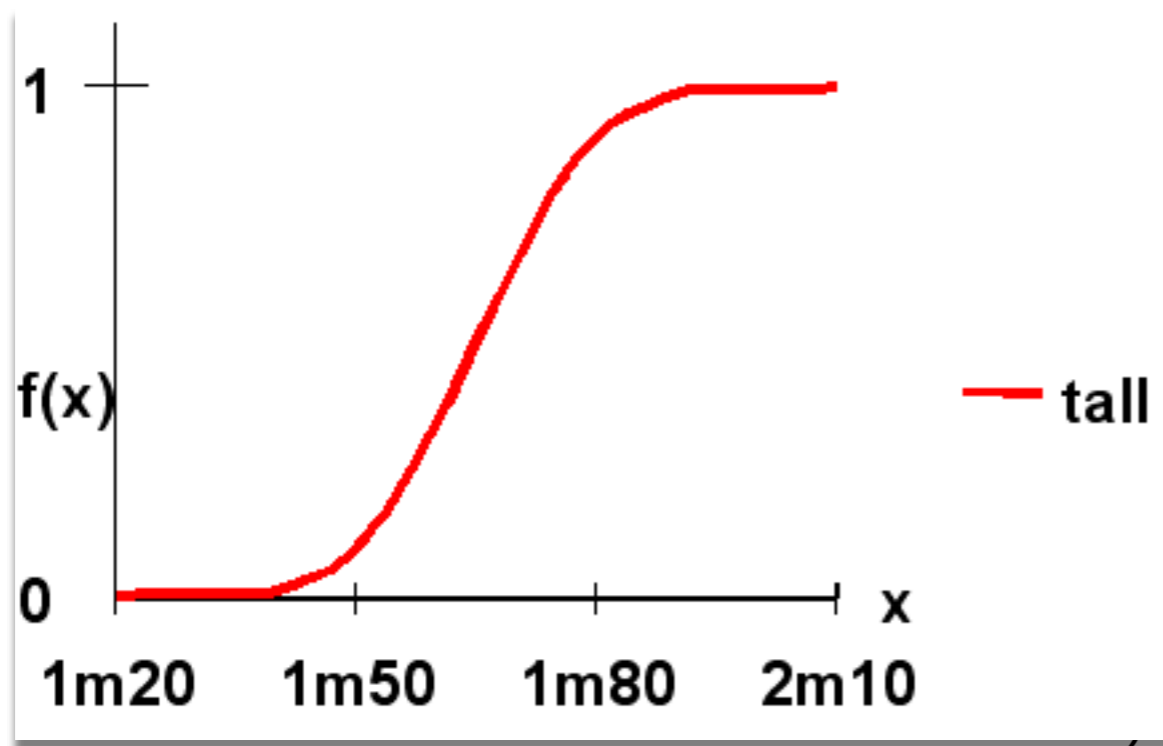
Logic programming with quantified truth: *reasoning with vague (rather than incomplete) information*

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'
<hr/>		
consequence		Z is C'

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

Fuzzy Logic Rice Cooker Reviews

Reader Google

http://www.rice-cooker-guide.com/fuzzy-logic-rice-cooker.html

WSNThesis vakantie cultuur spaans geeky reference nieuws artsy vub e-life remember

Rice Cooker Guide.com

- Performance Reports
- Pros & Cons
- Visitor Reviews

Best Fuzzy Logic Rice Cooker Brands

To help categorize, we have added this Fuzzy Logic rice cooker reviews page to help folks narrow down a specific brand/model. Fuzzy Logic rice has better flavor, great texture, and always comes out better than older basic cookers and remain the best rice cooker choice on the market.

(list subject to change as updates & new units become available)

Zojirushi Fuzzy Logic Rice Cookers

Being the most elite in the industry, Zojirushi rice cookers make a fine line of fuzzy logic cookers and offer some of the best models around.

Home

About This Site

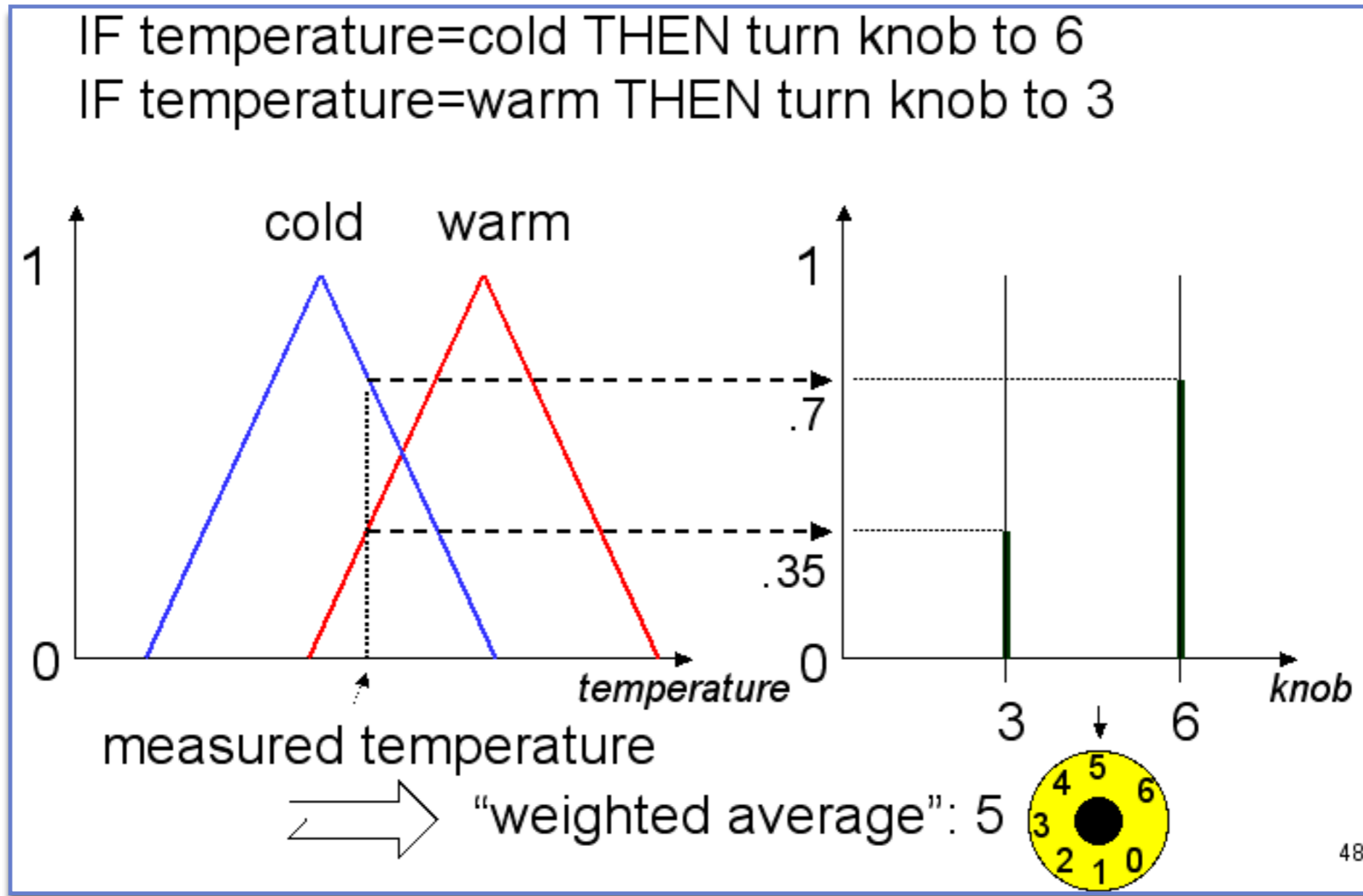
Popular Brands

- Sanyo Cookers
- Tiger Cookers
- Zojirushi Cookers
- Panasonic Cookers
- Aroma Cookers
- Cuisinart Cookers
- Black & Decker
- Rival Cookers

Cup Capacity

- Best 3 Cup Cookers
- Best 4 Cup Cookers

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

confidence
in conclusion q given absolute
truth of q_1, \dots, q_n

LP with quantified truth
weighted logic rules

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

fuzzy resolution procedure

$\tau(q) = c * \min(\tau(q_1), \dots, \tau(q_n))$

similar to
f-Prolog
[1990:liu]

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

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

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

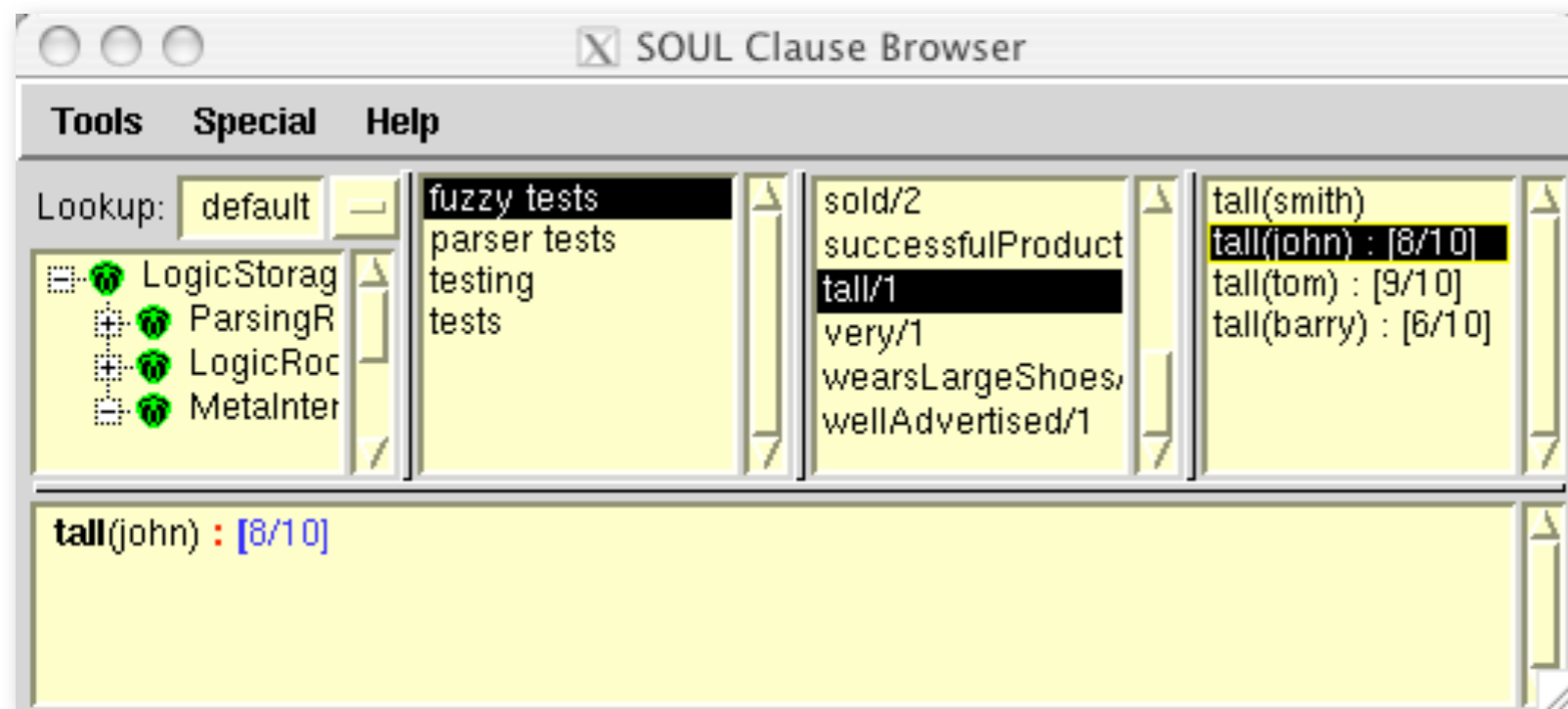
```
popular_product(?product) if  
sold(?product, ?amount),  
?amount > 10.
```

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

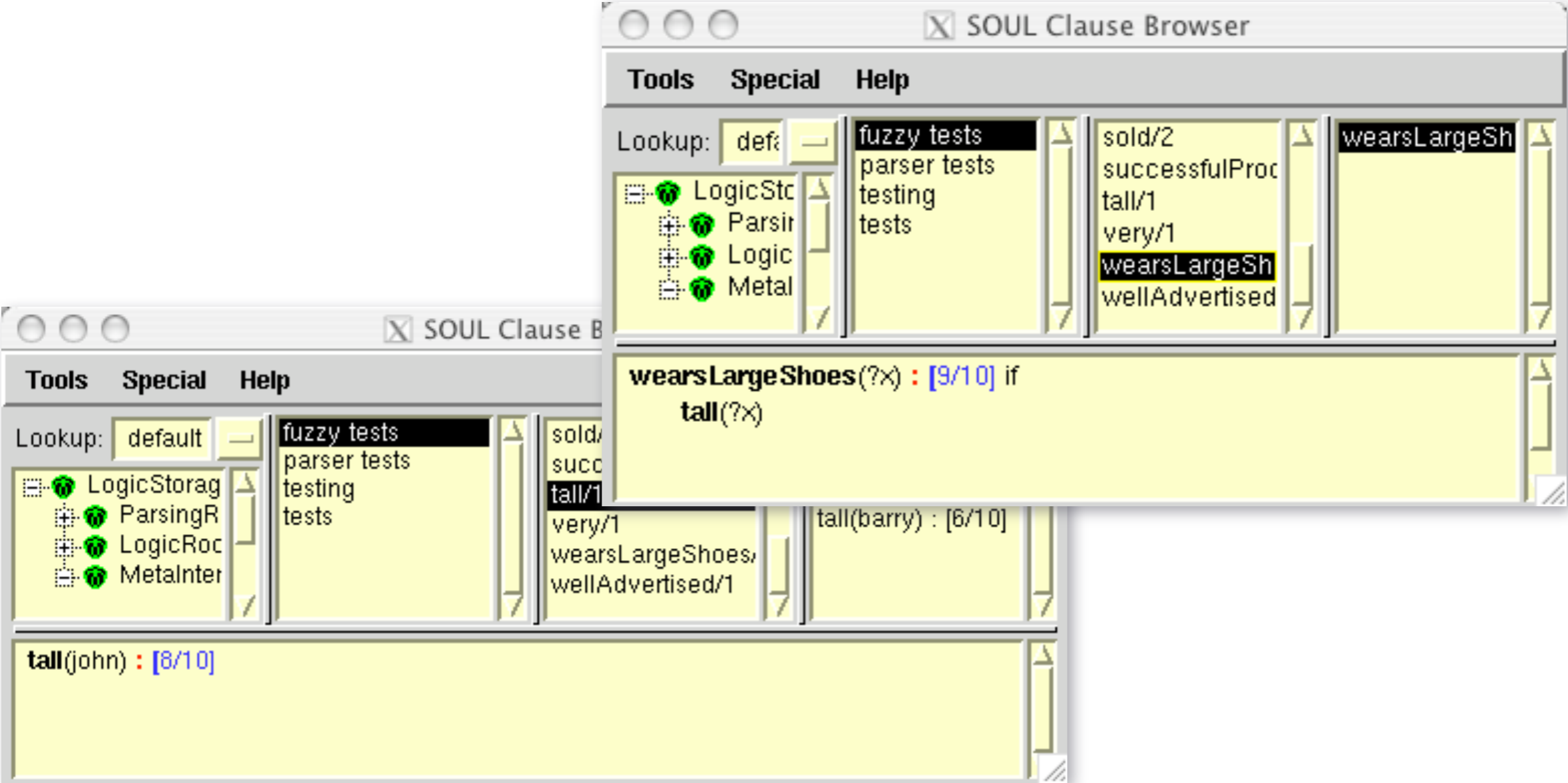
Logic programming with quantified truth:

a meta-interpreter for a fuzzy logic programming language

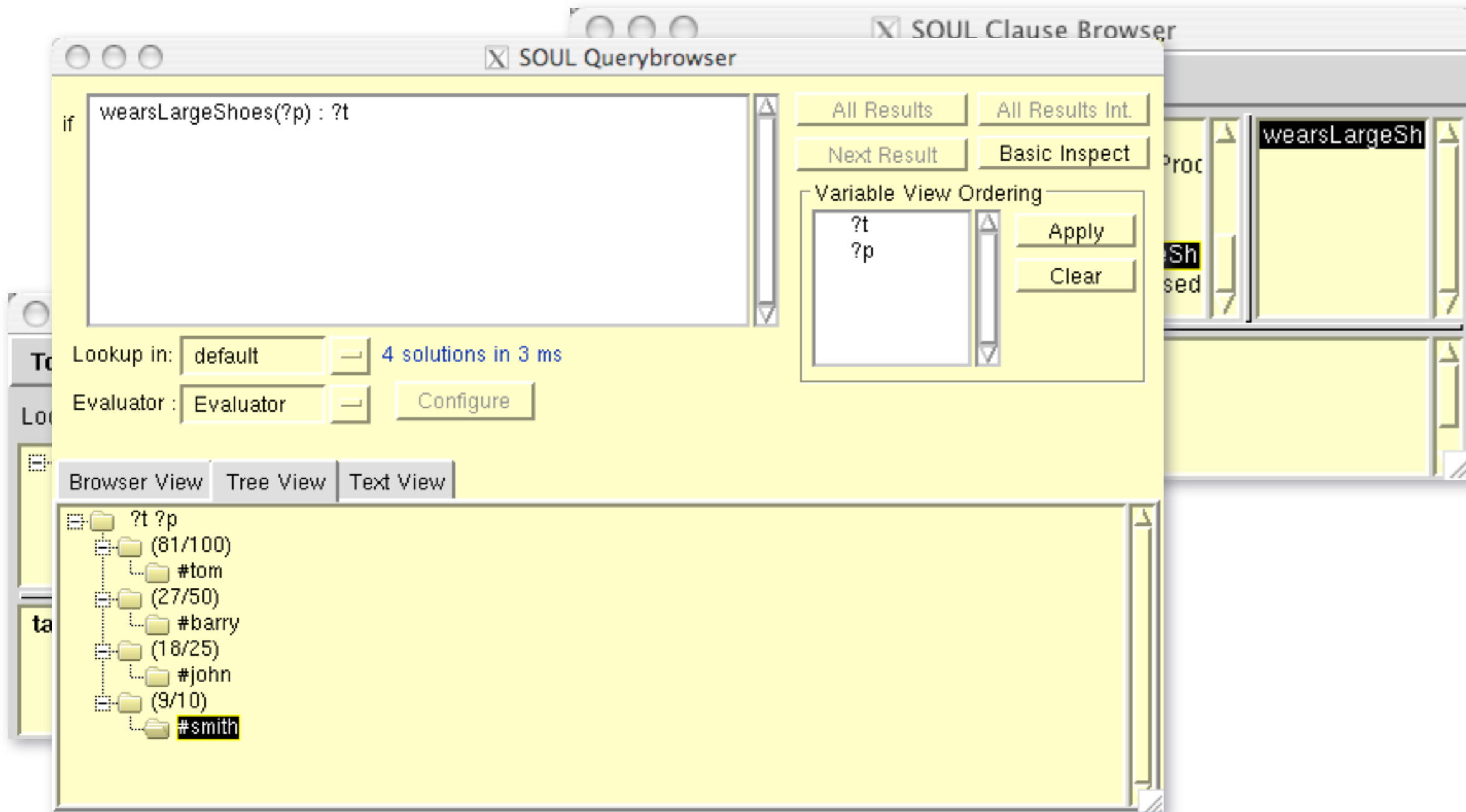
Logic programming with quantified truth: *a meta-interpreter for a fuzzy logic programming language*



Logic programming with quantified truth: *a meta-interpreter for a fuzzy logic programming language*



Logic programming with quantified truth: *a meta-interpreter for a fuzzy logic programming language*



Logic programming with quantified truth: *a meta-interpreter for a fuzzy logic programming language*

The screenshot shows the SOUL Clause Browser application. The window title is "SOUL Clause Browser". The menu bar includes "Tools", "Special", and "Help".

Lookup: default

Tree View:

- LogicPrimitives
- QuotedParseLayer
- TestQueriesLayer
- JavaEclipseReasoning
- SmalltalkReasoning
- IntensionalViewsLayer
- VisualQueryPredicatesForS
- MetaInterpretation
 - VanillaInterpreter
 - FuzzyInterpreter**
- ExampleBased
- OtherJavaTemplateQueries

clause lookup

interpreter
logic

isProvenListOfGoalsToExtent:aboveT
isProvenToExtent:aboveThreshold:/3

<&last> isProvenListOfGoalsToExtent: ?im
<&last> isProvenListOfGoalsToExten
<&gl&r> isProvenListOfGoalsToExter

<&last> isProvenListOfGoalsToExtent: ?degree aboveThreshold: ?threshold runningMin: ?currentMin implicationStrength: &implication if
!,
&last isProvenToExtent: ?d aboveThreshold: ?threshold,
?min equals: [?currentMin min: ?d],
?degree equals: [?min * ?implication],
[?degree >= ?threshold]

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

The screenshot shows the SOUL Querybrowser interface. The query is: `if [19 to: 25] contains: ?x, ?x isEqualToOrGreaterThanButRelativelyCloseTo: 20 : ?t`. The interface includes buttons for 'All Results', 'Debug', 'Next Result', 'Basic Inspect', and 'Next x Results'. There is a 'Variable View Ordering' section with a list containing '?t' and '?x', and buttons for 'Apply' and 'Clear'. Below the query, it shows 'Lookup in: JavaEclipse' and 'Evaluator: FuzzyEvaluator', with '6 solutions in 3 ms' and a 'Configure' button. At the bottom, there are tabs for 'Browser View', 'Tree View', and 'Text View'. The 'Browser View' shows a table of results:

Row	Value
(10 / 11)	25
1	23
(9 / 10)	24
(20 / 21)	

DEMO

Logic programming with quantified truth: quantifying over the elements of a fuzzy set

```

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

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

The screenshot shows the SOUL Querybrowser interface. The query editor contains the following code:

```

if
  ?about20 equals: [Soul.FuzzySet triangularWithPeak: 20 andMin: 10 andMax: 30],
  [8 to: 32] contains: ?e,
  ?about20 contains: ?e : ?t
    
```

Control buttons include: All Results, Debug, Next Result, Basic Inspect, Next x Results, Variable View Ordering (with Apply and Clear buttons), and a Configure button.

Lookup in: JavaEclipse (19 solutions in 2 ms)
Evaluator: FuzzyEvaluato

Views: Browser View, Tree View, Text View

Results (Browser View):

1	14
(3 / 10)	26
(7 / 10)	
(1 / 5)	
(1 / 10)	
(2 / 5)	
(9 / 10)	
(4 / 5)	
(1 / 2)	
(3 / 5)	

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:

- (always)
- ◇ (sometimes)
- (previous)
- (next).

evaluated against an
implicit temporal context:

- ϕ 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).
```

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

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

```
prove(next(A), T) :-  
  NT #= T + 1,  
  prove(A, NT).
```

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

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

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

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

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

#> 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 in integer domain

```
?- X #\= 20.
```

```
X in inf..19\21..sup.
```

X in union of two domains

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

```
X = 5.
```

```
?- X*X #= 144.
```

```
X in -12\12.
```

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

```
?- 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.
```

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+B=C).  
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

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

(b) source code

```
1 int *stack;
2 int top;
3 void init(int size) {
4     top = 0;
5     stack = malloc(size*sizeof(int));
6 }
7 void push(int element) {
8     stack[top++] = element;
9 }
10 #define pop() stack[--top];
```

**Execute
while
intercepting
high-level
events**

verified against

(c) documented behavior

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

5 stackInitialized(S) :- init(S).
6 stackUsed(S) :- push(S).
7 stackUsed(S) :- pop(S).
8 stackOperation(S) :- stackUsed(S).
9 stackOperation(S) :- stackInitialized(S).

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

(d) high-level events specification

```
1 intercept(after, stackPopOperation,
2   event(time, pop(stackTop, stackSize))).
3 intercept(after, stackPushOperation,
4   event(time, push(stackTop, stackSize))).
5 intercept(before, stackInitOperation,
6   event(time, init)).
```

**specific for this
application**

(f) associated run-time values

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

(e) application-specific instances

```
1 stackPushOperation(Construct, Path) :-
2   functionCallHasName(Construct, 'push').
3 stackPopOperation(Construct, Path) :-
4   macroCallHasName(Construct, 'pop').
5 stackInitOperation(Construct, Path) :-
6   functionCallHasName(Construct, 'init').
```

Logic programming with qualified truth: example application: reasoning about execution traces

(a) observed behavior

```

1 ..
2 event(60,cntEntered('ASG',13..1,['ASG','print','exit'])).
3 event(61,cntExited('ASG',13..1,['print','exit'])).
4 ..
    
```

(b) documentation as present in the source code

```

1 /*-----*/
2 /* ASS
3 /* expr-stack: [... .. DCT VAL] ->
4 /*           [... .. VAL]
5 /* cont-stack: [... .. ASS] ->
6 /*           [... .. ]
7 /*-----*/
8 static _NIL_TYPE_ ASG(_NIL_TYPE_)
9 { ... }
    
```

Execute
source code
while
intercepting

verified against

(c) documented behavior

```

1 cntDocumented('ASG',['ASG'|R],R).
2 cntDocumented('REF',['REF'|R],['REF','APL'|R]).
3 ...
4 behavioralModel :-
5   □(when(cntExecuted(Name,Before,After),
6         cntDocumented(Name,Before,After))).
7 cntExecuted(Name,StackBefore,StackAfter) :-
8   cntExited(Name,_,StackAfter),
9   •tcntEntered(Name,_,StackBefore).
    
```

(d) high-level events specification

```

1 intercept(before,continuationEntry,
2   event(time,cntEntered(cntName,cntPtr,cntStack))).
3 intercept(after,continuationExit,
4   event(time,cntExited(cntName,cntPtr,cntStack))).
    
```

specific for this application

(e) application-specific instances

```

1 continuationEntry(Construct,Path) :-
2   inContinuation(Construct,Path),
3   functionEntry(Construct,Path).
4 continuationExit(Construct,Path) :-
5   inContinuation(Construct,Path),
6   functionExit(Construct,Path).
7 continuation(Construct) :-
8   isFunctionDefinition(Construct),
9   expressionIn(Construct,Expression,_),
10  picoStack(Expression).
    
```

(f) associated run-time values

```

1 keyword(cntName,C,P,Expansion) :-
2   continuationName(C,P,Name),
3   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

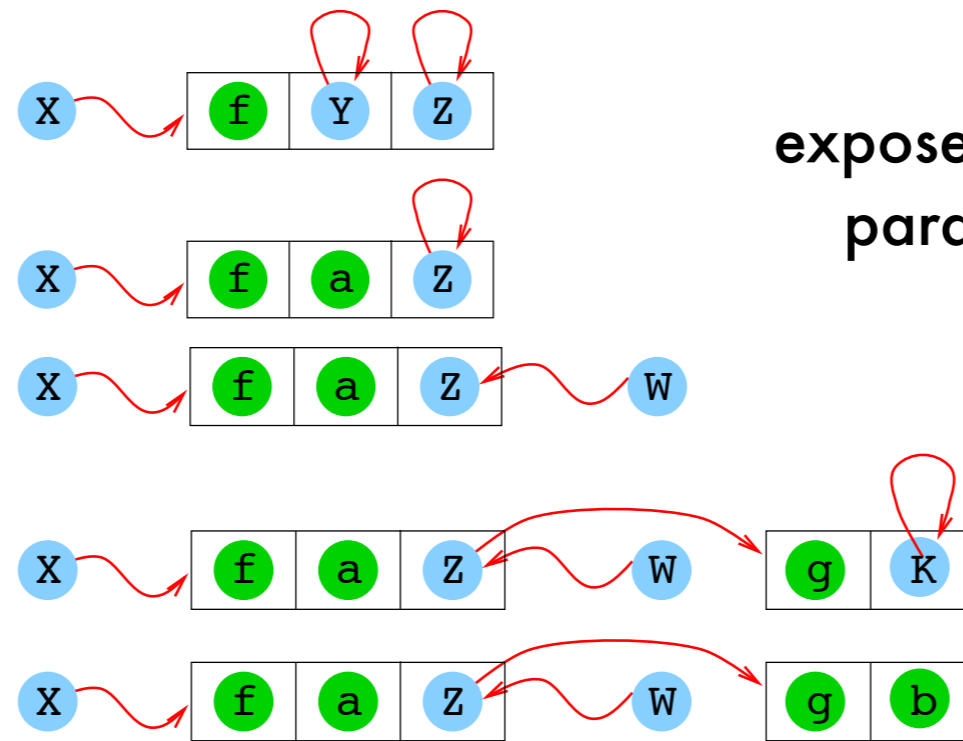
main :- X = f(Y,Z),

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!

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

if (no clauses left) **then** FAIL

Parallelism

else

$\sigma = \text{MostGeneralUnifier}(H,B)$

$\text{Query} = ((\text{Query} \setminus \{B\}) \cup \text{Body})\sigma$

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)

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

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

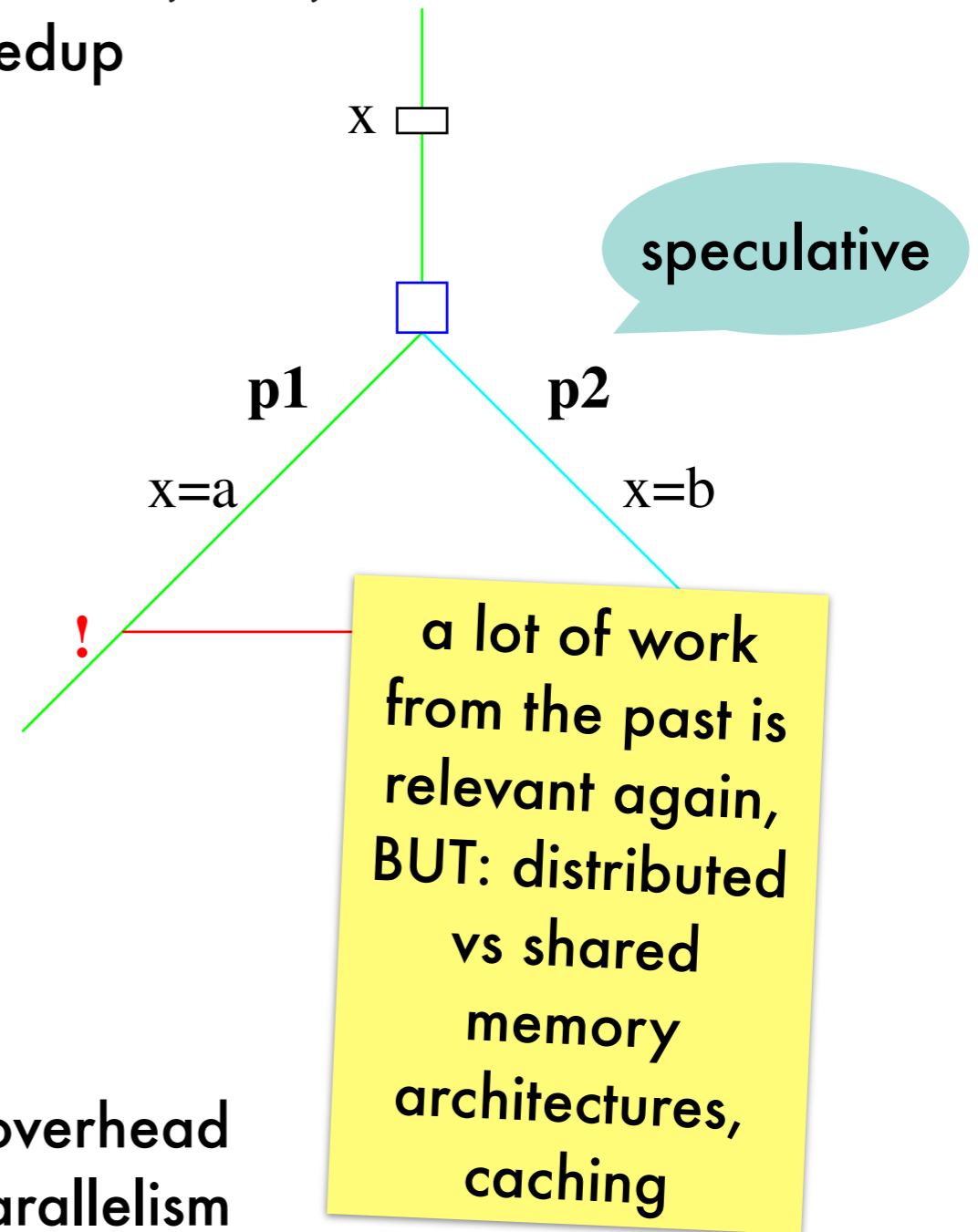
speculative work should be avoided to gain speedup

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

left-based scheduling, immediate killing on cut

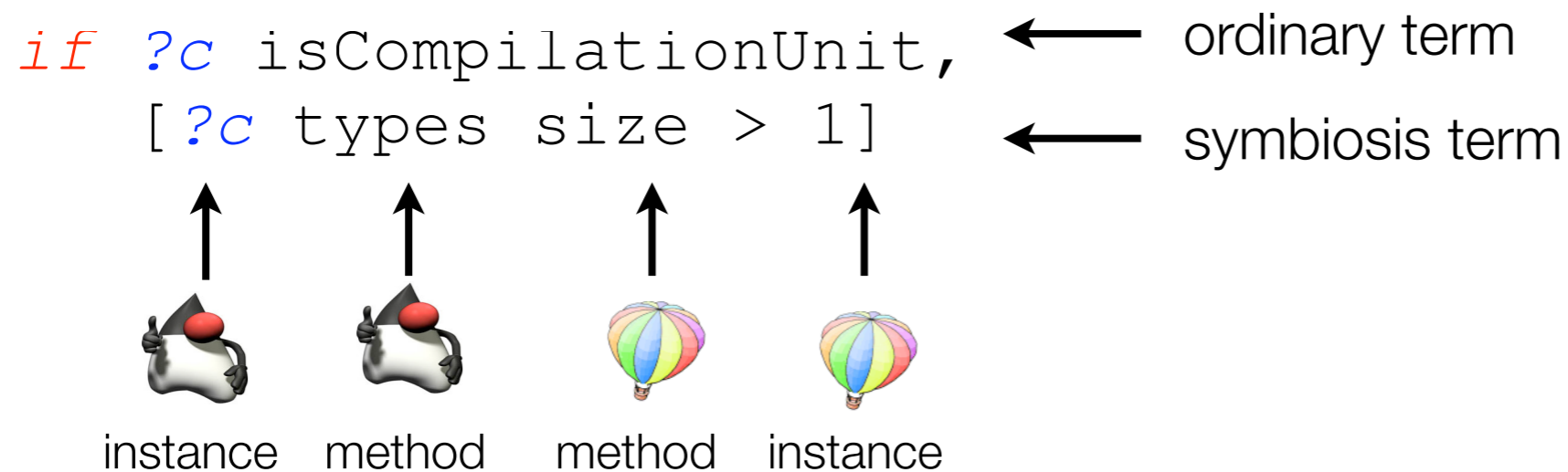
```
main :- l, s.  
  
:- parallel l/0.  
l :- large_work_a.  
l :- large_work_b.  
  
:- parallel s/0.  
s :- small_work_a.  
s :- small_work_b.
```

avoid incurring an overhead
from fine-grained parallelism



Logic programming in software engineering: *SOUL - symbiosis*

symbiosis with base program languages



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*

SOUL Querybrowser

```
if ?c isClassDeclaration,  
[?c getParent] equals: ?parent
```

All Results Debug
Next Result Basic Inspect
Next x Results

Variable View Ordering

2	?parent
1	?c

Apply
Clear

Lookup in: JavaEclipse 72 solutions in 12 ms
Evaluator: Evaluator Configure

Browser View Tree View Text View

MethodCalledFromDifferentSites
Component
MPCompoundBox
Leaf3
Composite
SecondSecondInner
OnlyLoggingLeaf
AbstractBaseClass
MPAugmentedType
NullTest
FirstSecondInner
MPFunctionPointer
Leaf4
IterationTest
MPFunctionObject
MPOutlineSubClass
Composite

Composite.java

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

SOUL Querybrowser

```

if jtClassDeclaration(?c,controlflow) {
  class SumComponentVisitor {
    ?m := [?modList ?type visitLeaf1(?arg) {
      ?s1; ?s2;
    }
  }
}
  
```

Variable View Ordering

?	arg
2	?s1
1	?m
	?type
	?modList
	?c
3	?s2

Lookup in: 153 solutions in 44 ms

Evaluator:

SumComponentVisitor >> public visitLeaf1(Componer

```

l1.value
new Integer(sum.intValue() + l1.value)
System.out
System.out.println("A visitor is visiting a leaf1.")
sum
(Leaf1)c1
c1
sum.intValue() + l1.value
System.out.println("A visitor is visiting a leaf1.");
"A visitor is visiting a leaf1."
sum.intValue()
super.visitLeaf1(c1);
super.visitLeaf1(c1)
Leaf1 l1=(Leaf1)c1;
c1
sum=new Integer(sum.intValue() + l1.value);
sum
sum=new Integer(sum.intValue() + l1.value);
Leaf1 l1=(Leaf1)c1;
System.out.println("A visitor is visiting a leaf1.");
sum
l1.value
sum.intValue() + l1.value
System.out
sum
sum.intValue()
c1
System.out.println("A visitor is visiting a leaf1.")
"A visitor is visiting a leaf1."
sum=new Integer(sum.intValue() + l1.value)
(Leaf1)c1
new Integer(sum.intValue() + l1.value)
  
```


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);
    }
  }
}
```

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:



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



```
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*

The screenshot shows the SOUL Querybrowser application window. The title bar reads "SOUL Querybrowser". The main area contains a query editor with the following code:

```
if jtStatement(?s1) { return ?exp;},  
jtStatement(?s2) { return ?exp;},  
[?s1 ~ ?s2]
```

Below the query editor, there are controls for "Lookup in:" (set to "JavaEclipse") and "Evaluator" (set to "Evaluator"). The status bar indicates "756 solutions in 9549 ms". To the right of the query editor, there are several buttons: "All Results", "Next Result", "Next x Results", "Debug", "Basic Inspect", "Apply", and "Clear". Below these buttons is a "Variable View Ordering" section with a list containing "?s2", "?s1", and "?exp".

At the bottom of the window, there are three tabs: "Browser View", "Tree View", and "Text View". The "Browser View" tab is active, showing a list of solutions. The first solution is highlighted and shows the following code:

```
return this.self().sum;  
return arg1;  
return indirectReturnOfArgument(o,delay - 1);  
return (Integer)indirectReturnOfArgument(sum,1);  
return p1;  
return p;  
return;  
return o.f;  
return arg;  
return p;  
return result;  
return p;  
return;  
return p2;  
return p2;  
return p2;
```

The second solution is also visible:

```
return o;  
return (Integer)retrieved;  
return indirectReturnOfArgument(o,delay - 1);  
return (Integer)indirectReturnOfArgument(sum,1);
```









The third solution is a simple "0".

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

