Tabled Logic Programs: Principles, Practice and Applications

C.R. Ramakrishnan
I.V. Ramakrishnan
Konstantinos Sagonas
Terrance Swift
David S. Warren

- Annotated version of Tutorial Slides presented at Joint International Conference and Symposium on Logic Programming, Bonn, Germany 1996.

Outline

- Motivation: Why is Tabling of General Interest?
- Definite Programs:
  - Algorithms
  - Tabling Applications
- Normal Programs
  - Algorithms
  - Tabling Applications
- Implementation

Motivation

- Solves inadequacies of Prolog (SLDNF)
  - Termination for e.g. Datalog programs

```
ancestor(X,Y):- parent(X,Y).
ancestor(X,Y):- ancestor(X,Z),parent(Z,Y).
```

  - Redundant subcomputations

```
join(X,Y):-
    supplemental(X,X2),rel_3(X2,Y).

supplemental(X,Y):-
    rel_1(X,X1),rel_2(X1,Y).
```

rel_1	rel_2	rel_3
(a,b)	(b,e)	(e,g)
(a,c)	(c,e)	(e,h)
(a,d)	(d,f)	(f,i)

In SLD resolution, 8 join operations are performed on the above example, while if `supplemental` is tabled, there will be only 6 join operations. In extreme cases, redundant subcomputations can lead to exponential data complexity for Prolog. (See the *knapsack* problem below). Tabling has polynomial data complexity for datalog programs with negation.

Motivation

- Allows Logic Programming to be used with disk-resident data.
  - Aditi [112], Validity both have a set-at-a-time interface to disk.
  - A scheduling strategy for tabling is *iteration equivalent* to semi-naive evaluation of a magic program [44].

Motivation

- Tabling is applicable to programs with negation. In fact, tabling can also handle loops through negation. It thus can implement three-valued semantics for negation such as the Well-Founded Semantics [114]
  - Allows logic programs to adequately handle inconsistencies.

    The village barber shaves everyone in the village who does not shave himself [40].

    ```
    shaves(barber,Person):-
             villager(Person),
             not shaves(Person,Person).
    shaves(doctor,doctor).

    villager(barber).         villager(mayor).
    villager(doctor).
    ```
  - Allows logic programming systems to be used to explore Knowledge Representation.
  - Tabling as defined below for WFS has polynomial data completity.

Motivation

- There has been a lot of research into it.
  - Tabling and Related Research
    - Formulation [41], [17], [108], [38], [60], [98], [115], [120], [15], [14], [20], [22], [33], [19], [105], [34], [94], [54], [110], [23], [27]
    - Implementation and Systems [6], [116], [64], [42], [2], [55], [82], [84], [106], [107], [93], [7], [21], [43],[44], [96], [121], [45], [83], [95] [92],
    - Optimizations [30]

Motivation

- Magic Sets and Related Research (e.g. Alexander Method)
  - ∗ Formulation [87], [5], [97],[111], [18], [69], [67], [8], [59], [73], [103], [39], [49], [79], [9], [101], [89] [66]
  - ∗ Implementation and Systems [6], [24], [112], [80], [99], [118], [119], [37], [48], [52], [68]
  - ∗ Optimizations [78], [70],[90], [91],[88], [61], [102], [51], [50], [104], [16]

- Bibliography is incomplete:
  - it considers only formulations of evaluation strategies and not general theories of Datalog, updates, etc.
  - Does not consider some newer areas such as Tabling / Magic Sets and constraints.

Motivation

- Practical and Research Applications
  - Parsing [72], [1], [63], [62]
  - Program Analysis [71], [58], [57], [86], [31]
  - Software Verification [100]
  - Graphics and Data Visualization [46], [26]
  - Diagnosis Systems [32], [28]
  - Other [13], [77]

SQL3 Standard includes recursion.

Definite Programs

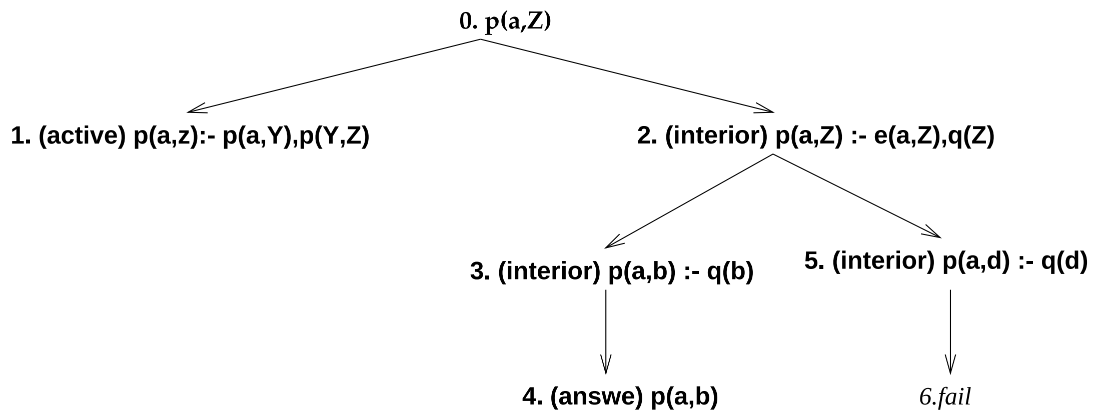
Many formulations of tabling are more or less equivalent for definite programs.

- Earley Deduction (1970)
- Backchain Iteration (1981)
- OLDT (1986)
- Alexander Method (1986)
- SLD-AL (QSQR) (1986-1989)
- Magic Templates (1986-88)
- Extension Tables (1987)
- SLG (1993)
- Logic Push Down Automata (1994)

A Tabled Evaluation can be characterized by

Operations plus a Search Strategy.

Definite Programs


0. p(a,Z)
1. (active) p(a,z):- p(a,Y),p(Y,Z)
2. (interior) p(a,Z) :- e(a,Z),q(Z)
3. (interior) p(a,b) :- q(b)
5. (interior) p(a,d) :- q(d)
4. (answe) p(a,b)
6.fail

Subgoal	Answers
p(a,Z)	p(a,b)

```
:- table p/2.

p(X,Z) :- p(X,Y),p(Y,Z).
p(X,Z) :- e(X,Z),q(Z).


e(a,b).  e(a,d).  e(b,c).

q(a).  q(b).  q(c).
```

Definite Programs

Definite Programs

Definite Programs: Tabling

- The search strategy for that program was a tuple-at-a-time strategy resembling Prolog's [45].
- One derivation path (or tree) may need to consume answers from another derivation path (or tree)
- May be more than one answer for a given tabled subgoal

Definite Programs: Tabling

For finite computations, a tabled evaluation can be seen as a sequence of forests. Given a forest $\mathcal{F}$ a new forest is determined by one of the following operations.

Definition 1

- NEW SUBGOAL. Given a node N with selected tabled literal B, where B is not in $\mathcal{F}$, create a new tree with root B.
- PROGRAM CLAUSE RESOLUTION. Given a node N that is a root node B, or has selected non-tabled literal B, resolve against B a program clause that has not previously been used for resolution against B in N.
- ANSWER RESOLUTION. Given an active node N with selected literal B, resolve an answer against B that has not been previously used for resolution against B in N.
- COMPLETION If S is a set of subgoals that have been *completely evaluated* remove all trees whose root is a subgoal in S.

Non-failure nodes have the form

$$Answer\_Template :- goal\_list$$

The *status* of a node — *active*, *interior* or *answer* — is determined respectively by whether the selected literal of the node is tabled, non-tabled, or if the *goal_list* is empty.

Definite Programs: Tabling

What does *Completely Evaluated* mean?

- A subgoal is completely evaluated iff it has all of its possible answers.
- A subgoal S is completely evaluated when all possible operations have been done on its nodes, and the nodes of trees upon which S depends.
- A ground subgoal is completely evaluated when an answer is derived for it.

Incremental Completion is necessary for efficient evaluation of programs.

Definite Programs: Tabling

Definite Programs: Tabling

Subgoal Dependency Graph

$$\begin{tikzcd}
p(a,Z) \arrow[d] \arrow[loop right] \\
p(b,Z) \arrow[d] \arrow[loop right] \\
p(c,Z) \arrow[loop right]
\end{tikzcd}$$

Incremental Completion can be performed a SCC at a time, or a set of SCCs at a time.

Definite Programs: Magic

Magic provides termination properties similar to tabling, along with goal orientation. How is a program evaluated using Magic?
Consider

```
sg(X,Y):- X=Y.
sg(X,Y):- p(X,Z),sg(Z,Z1),p(Y,Z1).
```

Magic usually assumes a bottom-up evaluation strategy such as semi-naive. This strategy iteratively derives a *delta set* of previously underived answers, and plugs these answers into appropriate places in the program to create a new delta set.

Definite Programs: Magic

- Split program up into EDB and IDB.
- Add a filter to the beginning of each IDB clause to make sure that whenever the clause is activated, it will have the same bindings as in a top-down evaluation.

```
sg(X,Y):- call(sg(X,Y)), X=Y.
sg(X,Y):- call(sg(X,Y)),
          p(X,Z),sg(Z,Z1),p(Y,Z1).
```

- Make sure that the proper calling filters are derived.

```
call(sg(Z,Z1)):-call(sg(X1,X)),p(X,Z).
```

- Add a magic seed to represent the original query.

```
call(sg(1,X)).
```

This is only a simple version of magic.

Definite Programs: Magic

Perform a linear semi-naive rewrite of the magic program:

```
call(sg(1,X)).

call(sg(Z,Z1)):-delta_call(sg(X1,X)),p(X,Z).

sg(X,Y):- delta_call(sg(X,Y)), X=Y.
sg(X,Y):- delta_call(sg(X,Y)),
          p(X,Z),sg_t-2(Z,Z1),p(Y,Z1).
sg(X,Y):- call_t-1(sg(X,Y)),
          p(X,Z),delta_sg(Z,Z1),p(Y,Z1).
```

- Each derived predicate can be thought of as consisting of a sequence of the delta-sets produced at each iteration. Thus, the set of facts available at the *end* of iteration t consists of the first t delta sets for each predicate.
- At time t, the second rule will join calls first produced at time $t-1$ (`delta_call/1`) with appropriate `sg/2` facts from $t-2$.

- At time t, the third rule will join all calls produced by the end of $t-1$ with the `sg/2` facts first produced at time $t-1$.

Definite Programs: Magic

Add a simple EDB to the program:

```
call(sg(1,X)).

call(sg(Z,Z1)):-delta_call(sg(X1,X)),p(X,Z).

sg(X,Y):- delta_call(sg(X,Y)), X=Y.
sg(X,Y):- delta_call(sg(X,Y)),
          p(X,Z),sg_t-1(Z,Z1),p(Y,Z1).
sg(X,Y):- call_t(sg(X,Y)),
          p(X,Z),delta_sg(Z,Z1),p(Y,Z1).

p(1,3).    p(1,4).   p(2,3).   p(2,4).
```

- Iteration 0: $call(sg(1,Y))$ added (magic seed).
- Iteration 1: $sg(1,1)$, $call(sg(3,Y))$, $call(sg(4,Y))$ added.
- Iteration 2: $sg(3,3)$, $sg(4,4)$ added.
- Iteration 3: $sg(1,2)$, $sg(1,1)$ each derived twice, $sg(1,2)$ added.
- Iteration 4: Fixpoint.

Definite Programs: Grammars

Consider the grammar

```
expr --> expr + term
expr --> term
term --> term * factor
term --> factor
factor --> ( expr )
factor --> integer(Int)
```

Definite Programs: Grammars

A translation into Prolog-style DCGs.

```
expr --> term, addterm.
addterm --> [].
addterm --> [+], expr.
term --> factor, multfactor.
multfactor --> [].
multfactor --> [*], term.
factor --> [I], {integer(I)}.
factor --> ['('], expr, [')'].
```

- the programmer has executed left-recursion elimination and left-factoring.
- grammar now has right-associative operators rather than the left-associative operators of the original grammar.

Definite Programs: Grammars

The same grammar using tabling.

```
:- table expr/2, term/2.

expr --> expr, [+], term.
expr --> term.
term --> term, [*], factor.
term --> factor.
factor --> ['('], expr, [')'].
factor --> [Int], {integer(Int)}.
```

- Syntactic variant of original grammar.
- Has no associativity problem

Definite Programs: Grammars

- Applying tabling to a DCG grammar can effectively give Earley Parsing [41]
  - Supplementary Tabling (Supplementary Magic) can convert the grammar to Chomsky Normal Form[1].
- Earley Parsing of grammars in Chomsky Normal Form takes at most $\mathcal{O}(N^3)$ for ambiguous grammars; at most $\mathcal{O}(N^2)$ for unambiguous grammars; and is linear for a large class of grammars.
- Additional optimizations such as Left Factoring can be performed by CRA optimizations as described in [30].

[1]Implementing Earley's Dotted Rules.

Definite Programs: Grammars

- It is efficient to represent sentences in Datalog when using tabling (see below in Implementation).

```
'C'(every,0,1).
'C'(man,1,2).
'C'(loves,2,3).
'C'(a,3,4).
'C'(woman,4,5).
```

- Tabling (and Earley Parsing) offer useful complexity advantages for other grammatical formalisms such as categorial grammars [1].

Definite Programs: Dynamic Programming

The minimum edit distance problem: find the minimum number of insertions, deletions, or replacements to turn one string into another.

```
:- table med/3.

med(0,0,0).
med(0,M,M) :- M > 0.
med(N,0,N) :- N > 0.
med(N,M,C) :- N > 0, M > 0,
        N1 is N-1, M1 is M-1,
        med(N1,M,C1), C1a is C1+1,
        med(N,M1,C2), C2a is C2+1,
        med(N1,M1,C3),
        a(N,A), b(M,B),
        (A==B -> C3a=C3; C3a is C3+1),
        min(C1a,C2a,Cm1), min(Cm1,C3a,C).
```

- c.f. [65] pg. 153–154 for an equivalent imperative solution.

Definite Programs: Dynamic Programming

Solutions to `med/3` recursively create a $M \times N$ array:

Definite Programs: Dynamic Programming

The knap-sack problem:

Given n items, each of integer size k_i $(1 \leq i \leq n)$, and a knap-sack size K.

- determine whether there is a subset of the items that sums to K.
- Find such a subset.

Definite Programs: Dynamic Programming

A Prolog solution to the knapsack problem.

```
ks(0,0).
ks(I,K) :- I>0,
    I1 is I-1, ks(I1,K).
ks(I,K) :- I>0,
    item_size(I,Ki),
    K1 is K-Ki, K1 >= 0,
    I1 is I-1, ks(I1,K1).

item_size(1,2).
item_size(2,3).
item_size(3,5).
item_size(4,6).
```

Worst-case comlexity is 2^I.

Definite Programs: Dynamic Programming

A tabling solution to the knapsack problem.

```
:- table ks/2.

ks(0,0).
ks(I,K) :- I>0,
    I1 is I-1, ks(I1,K).
ks(I,K) :- I>0,
    item_size(I,Ki), K1 is K-Ki,
    K1 >= 0, I1 is I-1,
    ks(I1,K1).

item_size(1,2).
item_size(2,3).
item_size(3,5).
item_size(4,6).
```

Worst-case complexity is I^2.

Definite Programs: Dynamic Programming

But how do you find the subset(s)?

```
ksp(0,0,[]).
ksp(I,K,P) :- I>0,
    I1 is I-1,
    ks(I1,K),
    ksp(I1,K,P).
ksp(I,K,[I|P]) :- I>0,
    item_size(I,Ki),
    K1 is K-Ki, K1 >= 0,
    I1 is I-1,
    ks(I1,K1),
    ksp(I1,K1,P).
```

- Note that `ks/2` does not repeat computations.
- cf. [65] pg. 110 for an equivalent imperative solution.

Definite Programs: Dynamic Programming

- Note that with the goal-orientation of tabling, in certain problems it may not be necessary to build an entire array. One such case occurs when tabling is used in the Unification Factoring compiler optimization [29].
- [48] offers other approaches to dynamic programming using tabling.

Definite Programs: Applications

Program Analysis

- Expoits the ability of tabled evaluation to find minimal models of definite programs
- General Strategy: *Abstract Compilation* (e.g., see [35, 53])
  - From a given source (concrete) program, obtain an *abstract* program.
  - Concrete semantics of abstract program $\equiv$ abstract semantics of concrete program.
  - Evaluate abstract program using some *complete* evaluation strategy.

Applications: Program Analysis

Example: Groundness Analysis (from [25])

```
      append([],Ys,Ys).
      append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

$$\Downarrow$$

```
   g_append(g,Y,Y).
   g_append(X1,Ys,Z1) :- iff(X1,X,Xs), iff(Z1,X,Zs),
                         g_append(Xs,Ys,Zs).

   iff(g,g,g).
   iff(n,n,n).
   iff(n,n,g).
   iff(n,g,n).
```

- Elegance of formulation ("specification-based").
- Ease of implementation.
- Applies to imperative [85], functional [31] and logic [25] program analysis.
- Efficient enough for simple logic and functional program analysis [31] to be put in compilers.

Applications: Program Analysis

Semantic equations can be expressed as metaprograms

Example: While programs

```
interp( S_1 ; S_2 )    --> interp(S_1), interp(S_2).

interp( if E then S_t else S_f ) --> expr_eval(E, Val),
                           Val == true -> interp(S_1)
                                       ;  interp(S_2).

interp( while E do S) --> expr_eval(E, Val),
                           Val == true -> interp(S),
                                          interp( while E do S)
                                       ;  []

interp( X := E )       --> expr_eval(E, Val),
                           store(Val, X).
⋮

expr_eval( (E_1 + E_2) , Val) --> ...
⋮
```

Applications: Program Analysis

Generating abstract programs:

- Express abstract semantics as metaprograms
- Partially evaluate abstract semantic equations wrt. input (concrete) program

Metaprogramming and Tabling

Power of combining metaprogramming with tabling yields ability to express a variety of semantics.

Applications:

- Program analysis
- Model checking (see CCS, below)
- Constraint languages
- Knowledge representation (see Negation, below)

Definite Programs: Applications

CCS as a Horn Program: (Y.S. Ramakrishna)

```
:- table trans/3.

% Prefix:      Act.P Act-> P
trans(pref(Act, P), Act, P).

% Choice:      P = P1 + P2
trans(choice(P, _Q), Act_a, P1) :-trans(P, Act_a, P1).
trans(choice(_P, Q), Act_a, Q1) :-trans(Q, Act_a, Q1).

% Parallel:    P = Q | R
trans(par(P, Q), Act_a, par(P1, Q)) :-trans(P, Act_a, P1).
trans(par(P, Q), Act_a, par(P, Q1)) :-trans(Q, Act_a, Q1).
% Represent Coactions
trans(par(P, Q), tau, par(P1, Q1))  :-
        trans(P, Act_a, P1), trans(Q, Act_b, Q1),
        comp(Act_a, Act_b).
trans(par(P, Q), tau, par(P1, Q1))  :-
        trans(P, Act_a, P1), trans(Q, Act_b, Q1),
        comp(Act_b, Act_a).

% Restriction:  P\L Act-> P1\L
trans(rest(P,L), Act_a, rest(P1,L)) :-
        trans(P, Act_a, P1),legitimate_action(Act_a, L).

% Relabelling: P = Q [f]
trans(relab(P, Hom_f), Act_b, relab(P1, Hom_f)) :-
        trans(P, Act_a, P1),map(Hom_f, Act_a, Act_b).
```

```
% Transitive Redefinition
trans(P, Act_a, Q) :- def(P, R), trans(R, Act_a, Q).
```

Applications: Program Analysis

- The previous meta-interpreter, when combined with a model checking algorithm can be used for verification of concurrent systems.
- Time and space utilization are roughly comparable with special-purpose model checkers

Definite Programs: Topics in Definite Tabling

Recall that a tabled evaluation can be characterized by operations plus scheduling.

- Changes in Scheduling: Local and Breadth-First
- Changes in Operations: Checking for Variance vs. Subsumption

Definite Programs: Scheduling

Tabled evaluations have NEW SUBGOAL operation and ANSWER RESOLUTION operation which must be scheduled along with the PROGRAM CLAUSE RESOLUTION step of SLD.

- Can return answers as soon as they are derived, or postpone their return.
- Can create a new tree as soon as there is a selected literal for it, or postpone this operation.
- Prolog's strategy is approximated by postponing neither the return of answers or the creation of new trees.
- Postponing answer return out of an SCC until an SCC is completely evaluated gives *local* evaluation [45].
- Postponing both operations until the end of an iteration, gives *breadth-first evaluation*, typi-

cal of semi-naive magic evaluations. (Joint work with J. Freire).

Definite Programs: Tabling and Magic

Consider a tuple-at-a-time evaluation of same-generation

Definite Programs: Tabling and Magic

Now consider a set-at-a-time evaluation

In particular, this is how a semi-naive evaluation would evaluate a magic-rewritten same-generation program.

Definite Programs: Tabling and Magic

Relations have been often studied: [111], [18], [103], [15], [106], [36].

- Asymptotically Equivalent: A version of magic templates under naive evaluation is asymptotically equivalent to a version of QSQR tabling [97].
- Iteration Equivalent: A version of magic templates under semi-naive evaluation is asymptotically iteration equivalent to a version of tabling [45]. This means
  - At each iteration a magic fact is added if a new subgoal is called
  - At each iteration a non-magic fact is derived (added) if a new answer is derived (added).

Definite Programs: Tabling and Magic

- Tabling starts from resolution and is a *programmer's* view of combining top-down and bottom-up.
- Magic starts from semi-naive evaluation and is a database query processor's view of combining top-down and bottom-up.
- Tabling captures disk-access methods of magic via Breadth-First Tabling.
- Magic captures the dynamic dependencies of tabling via *Ordered Search* [79]
- Both allow subsumption or variance of calls or answers.

Reflections in a fun-house mirror?

Definite Programs: Operations

The operations mentioned before assumed a variant check for subgoals and answers. Alternately one could use subgoal subsumption or answer subsumption.

Consider the program

```
p(X):- p(f(X)).
p(a).
```

Minimal model is {`p(a)`}.

As previously defined, the tabling evaluation of a query `?- p(X)` would create an infinite number of trees.

p(X),p(f(X)),p(f(f(X))),...

Subgoal subsumption addresses this problem.

Definite Programs: Subgoal Subsumption

Definition 2 • NEW NON-SUBSUMED SUBGOAL. Given a node N with selected tabled literal B, where B is not *subsumed by a subgoal in* $\mathcal{F}$, create a new tree with root B.

Theorem 1 *Let P be a program with a finite model in which every predicate is tabled, and $\mathcal{E}$ be an evaluation consisting of* NEW NON-SUBSUMED SUBGOAL, PROGRAM CLAUSE RESOLUTION, ANSWER RESOLUTION *and* COMPLETION. *Then $\mathcal{E}$ will correctly terminate after a finite number of operations.*

- Originally stated in [108] using OLDT formalism.

Definite Programs: Subgoal Subsumption

- Subgoal Subsumption can also be of use for Datalog programs

Consider a same-generation program.

```
sg(X,X).
sg(X,Y):- sg(Y,X).
sg(X,Y):- up(X,X1),sg(X1,X2),down(X2,Y).
```

Definite Programs: Subgoal Subsumption

For the query $sg(f,f)(X,Y)$ the annotation is

```
sg(f,f)(X,X).
sg(f,f)(X,Y):- sg(f,f)(Y,X).
sg(f,f)(X,Y):-
     up(X,X1),sg(b,f)(X1,X2),down(X2,Y).

sg(b,f)(X,X).
sg(b,f)(X,Y):- sg(f,b)(Y,X).
sg(b,f)(X,Y):-
     up(X,X1),sg(b,f)(X1,X2),down(X2,Y).

sg(f,b)(X,X).
sg(f,b)(X,Y):- sg(b,f)(Y,X).
sg(f,b)(X,Y):-
     up(X,X1),sg(b,f)(X1,X2),down(X2,Y).
```

Definite Programs: Answer Subsumption

Definition 3 • (NON-SUBSUMING) ANSWER RESOLUTION. Given an active node N with selected literal B, resolve an answer A against B if

- A that has not been previously used by N.
- A is not subsumed by any other answer in the tree for B.

This tends to be most useful when subsumption is used on a partial order other than that of terms.

- In the 3-valued information ordering true and false are greater than undefined. Therefore, true answers subsume undefined answers.
- Subsumption can be generalized to implication for both subgoals and answers. In an appropriate constraint domain

$$p(X) : (X > 2) \Rightarrow p(X) : (X > 3)$$

Definite Programs: Answer Subsumption

Answer Subsumption models *min* and *max* aggregate operators.

Example: find the shortest distance between two people in the same generation.

```
sgi(X,X)(0).
sgi(X,Y)(I) :-
     anc(X,Z),
     subsumes(min)(sgi(Z,Z1),I1),
     anc(Y,Z1), I is I1+1.

:- subsumes(min)(sgi(joan,carl),I).
```

Variance vs. Subsumption

- Call Variance + Answer Variance gives Prolog-style observables that are suitable for meta-interpretation.
- Call Variance + Answer Subsumption is useful for non-stratified negation. Call variance in non-floundering programs avoids constructive negation. Answer subsumption is used to handle uncertain answers.
- Call Subsumption is useful for minimal model computations of definite or stratified programs.

Tabling Programs with Negation

- Realistic programs use negation, how is negation combined with tabling?
- Can the greater expressive power of tabling also be used as a basis for a Logic Programming implementations of Non-Monotonic Reasoning?

Intuition: The well-founded semantics (WFS) treats all paths with infinite positive recursion as failed, and all paths with infinite recursion through negation as undefined. Thus the loop-checking features of tabling can be used to evaluate WFS.

Negation

The following progam uses negation in a straightforward way, but may cause Prolog to go into an infinite loop.

```
get_best_choices(Course,Teacher,Final):-
    can_teach(Course,Initial_choice),
    best_choice(Course,Initial_choice,Final_choice).

best_choice(Course,Teacher,Teacher):-
    not better_choice(Course,Teacher).
best_choice(Course,Teacher,Final):-
    just_as_good_choice(Course,Teacher,Other),
    best_choice(Course,Other,Final).

better_choice(Course,Teacher):-
    can_teach(Course,Teacher1),
    not (Tearcher1 = Teacher),
    rates(Course,Teacher1,Rank1),
    rates(Course,Teacher,Rank),
    Rank1 > Rank.

just_as_good_choice(Course,Teacher,Other):-
    can_teach(Course,Other),
    not (Other = Teacher),
    rates(Course,Other,Rank1),
    rates(Course,Teacher,Rank),
Rank1 >= Rank.
```

Negation:
Towards WFS through stratification

Negation: Towards WFS through stratification

Lower stratification classes are computed by

- Determining a dependency graph (DG)
- Determining whether components in the dependency graph contain cycles through negation

Examples:

- Predicate Stratification [4]: single DG for entire program, based on predicate dependencies.
- Local Stratification [75]: single DG for entire (grounded) program, based on atom dependencies.

Negation:
Towards WFS through stratification

Negation: Towards WFS through stratification

The basic idea of forming a dependency graph and checking for loops through negation is performed *iteratively* in the higher stratification classes such as modular stratification [89] and weak stratification [74]. We consider the highest of these, Dynamic Stratification [76, 11] in detail.

- The power of Dynamic stratification can be seen from the following theorem

  Theorem 2 *[76] A program is Dynamically Stratified iff it has a two-valued well-founded model.*

Negation: Dynamic Stratification

Dynamic stratification iteratively finds an interpretation I for a ground program, and *reduces* the rest of the program with respect to I. Start with $I_0 = \emptyset$

$$A_h \text{ :- } A_1, ..., A_m, \mathit{not}\ A_{m+1}, ..., \mathit{not}\ A_n$$

- To determine positive facts find the least fixpoint of the operator
  - $\mathcal{T}_I(T) = \{A :$ there is a clause $B \leftarrow L_1, ..., L_n$ in P and a ground substitution θ such that $A = B\theta$ and for every $1 \leq i \leq n$ either $L_i\theta$ is true in I, or $L_i\theta \in T\}$;
- To determine negative facts, find the *greatest* fixpoint of the operator
  - $\mathcal{F}_I(F) = \{A :$ for every clause $B \leftarrow L_1, ..., L_n$ in P and a ground substitution θ such that $A = B\theta$ and there is some i $(1 \leq i \leq n)$, such that $L_i\theta$ is false in I or $L_i\theta \in F\}$.

Negation

Consider the program:

```
p(b).
p(c) :- not p(a).
p(X) :- t(X,Y,Z), not p(Y), not p(Z).
t(a,b,a).
t(a,a,b).
```

the ground instantiation of this program is:

```
p(b).
p(c):- not p(a).
p(a) :- t(a,a,a), not p(a), not p(a).
p(a) :- t(a,a,b), not p(a), not p(b).
:
p(a) :- t(a,b,a), not p(b), not p(a).
:
p(c) :- t(c,c,c), not p(c), not p(c).

t(a,b,a).
t(a,a,b).
```

Negation

The first partial model, I_0 is $\emptyset$ so the operators effectively work on the program

```
p(b).
p(c):- undef.

p(a) :- t(a,a,a), undef,undef.
p(a) :- t(a,a,b), undef,undef.
:
p(a) :- t(a,b,a), undef,undef.
:
p(c) :- t(c,c,c), undef,undef.

t(a,b,a).
t(a,a,b).
```

Whose fixpoint gives I_1 in which

$$I_1^{true} = \{p(b), t(a,a,b), t(a,b,a)\}$$

are true and

$$I_1^{false} = \{t(a,a,a), t(a,a,c), ...\}$$

are false.

Negation

Thus the second reduction is

```
p(c):- undef.
p(b).
t(a,b,a).
t(a,a,b).
```

Now $I_2^{true} = I_1^{true}$, while $I_2^{false} = I_1^{false} \cup \{p(a)\}$, and the third reduction is

```
p(c).
p(b).
t(a,b,a).
t(a,a,b).
```

Which adds `p(c)` to I_3^{true}. Further iterations will not change I_3, which is, in fact, a model for the program.

- Any undefined literals at the end of this iterative process may be said to be in the *ultimate* stratum.

Negation

The method just shown was pure bottom-up.

- To make it goal-oriented requires a notion of relevance.

  Assuming a left-to-right computation rule:

  - In Prolog, relevant literals for a selected clause belong to a failing prefix.

    p(a) :- t(a,b,a), not p(b), not p(a).
  - To get this dynamic stratification an evaluation cannot view only a prefix.

    p(a) :- t(a,a,b), not p(a), **not p(a)**.

Are relevant literals all those in a body for a selected clause?

Negation

Negation

Negation

Negation

These operations can be expressed in SLG-style terminology as follows:

- NEGATIVE RETURN Given a node N:

  $$Answer\_template \leftarrow Delay\_list | not\, B, Goal\_list$$

  where B is true in $\mathcal{F}$ create a failure node as the child of B. If B is failed, create a unique child

  $$Answer\_template \leftarrow Delay\_list | Goal\_list$$

  with appropriate status.

- DELAY Given a node N:

  $$Answer\_template \leftarrow Delay\_list | not\, B, Goal\_list$$

  delay evaluation of *not* B by creating a child of N:

  $$Answer\_template \leftarrow not\, B, Delay\_list | Goal\_list$$

- SIMPLIFICATION Given an answer A whose delay list contains a literal L, if L is true in F, remove L from the delay list of A. If L is false in F, remove A from the table.

Negation

Is this programming or specification?

Left-to-right dynamic stratification allows only failing *prefixes*

- $\mathcal{F}_{\mathcal{M}_i}(F) = \{A$: for every clause $B \leftarrow L_1, ..., L_n$ in P and a ground substitution θ such that $A = B\theta$ and (1) is some i ($1 \leq i \leq n$), such that $L_i\theta$ is false in $\mathcal{M}_i$ or $L_i\theta \in F$; (2) there exists a *failing prefix*: for all j ($1 \leq j \leq i-1$), $L_j\theta$ is true in $I\}$.

By adjusting the operations of SLG, a tabling strategy SLG_{RD} can be formulated with the following property.

Theorem 3 *[94] Given a ground program P, an SLG_{RD} evaluation will only delay on encountering a literal in the ultimate left-to-right dynamic stratum.*

We conjecture that a similar theorem is possible for other formalisms such as Well-Founded Ordered Search [101].

Meta-Interpreting in WFS

To meta-interpret a program with loops, table the meta-interpreter.

```
:- table demo/1.

demo(true).
demo(',' (A,B)):- !,demo(A),demo(B).
demo(not A):- !,not demo(A).
demo(A):- clause(A,B),demo(B).
```

- Note Use of cuts with tabled predicates

Negation

A meta-interpreter for well-founded semantics with explicit negation (WFSX) [3]

```
demo(_)(true).
demo(X)(',''(A,B)):-!,
        demo(X)(A),demo(X)(B).
demo(t)(not(A)):-!, not(demo(tu)(A)).
demo(tu)(not(A)):-!, not(demo(t)(A)).
demo(t)(A):- rule(A,B), demo(t)(B).
demo(tu)(A):- rule(A,B), demo(tu)(B),
        exchange(A,A_opp),demo(tu)(not(A_opp))

exchange(-B,B):-!.
exchange(B,-B).
```

- Note use of Hilog and Tabling

Negation

Similar meta-interpreters transformations can be performed for

- Head-Cycle Free Disjunctive Logic Programs [10]
- Generalized Horn Programs [12]
- Extended Databases [117, 109]
- Imex Negation [56]
- A restriction to WFS of the action language $\mathcal{A}$ [47].

Tabling can also be used as a preprocessor for stable model computations.

Negation

An Extended Logic Program (C. Damasio)

```
perforation(X) <-
    sudden_pain(X),abd_tenderness(X),
    peritoneal_irritation(X),
    not_believed high_amylase(X).
pancreatitis(X) <-
    sudden_pain(X),abd_tenderness(X),
    peritoneal_irritation(X),
    not_believed jobert(X).

-nourish(X) <- perforation(X).
-nourish(X) <- pancreatitis(X).

h2_antagonist(X) <- pancreatitis(X).
h2_antagonist(X) <- perforation(X).

surgery_indication(X) <- perforation(X).
-surgery_indication(X) <- pancreatitis(X).
```

```
anesthesia(X) <- surgery_indication(X).
```

Negation

Suppose a patient comes in with

```
sudden_pain(patient) <- true.
abd_tenderness(patient) <- true.
peritoneal_irritation(patient) <- true.
```

A paraconsistent model is derived

- Indications are contradictory. The patient has both a perforation and pancreatitis. As a result, there is an indication for surgery and an indication against.
- Nonetheless, the patient should be given h2_antagonists and should not be nourished.

Negation

Suppose an amylase test is performed and comes back high.

Then the belief in the perforation will be withdrawn, as will the surgery indication.

Alternatively, suppose the user did not want to make an epistimological commitment about Jobert's Syndrome. Define:

```
jobert <- unknown.
```

where

```
unknown <- not unknown.
```

- `pancreatitis(patient)` would therefore have truth-value unknown.
- The delayed clause would be
  `pancreatitis(patient):- believed_not jobert.`

There are four truth values to use: true, false, both and neither.

Implementation of Tabling

Some Implementations of Tabling

- Semi-Naive Model: Coral, Aditi, LDL, LogicBase
- WAM Model: Portable SLG, XSB[2],

[81] provides a relatively recent survey.

Currently:

- Systems based on the WAM model are about an order of magnitude faster for in-memory data, and have a tighter integration with Prolog.
- Systems based on the semi-naive model have a tighter integration with disk.

[2]Tabling features in XSB were implemented by J. Freire and P. Rao along with the authors.

Implementation of Tabling

Features necessary for tabling (from a Prolog perspective).

- Mechanism to suspend and resume a computation
- Mechanism to access tables
- Mechanism to detect (incremental) completion
- Mechanism to handle undefined literals in a clause

Implementation of Tabling

Issue: Suspension and Resumption of subgoals.

- Suspension is used to wait for answers, to wait for information about a negative subgoal or to delay the start of a new tree.
- Resuming is necessary to return answers, to return information about a negative goal, or to create a tree for a suspended subgoal.

The various tabling strategies — batched, local, breadth-first — are reflected at the implementation level by suspending and resuming computation paths. Semi-naive can also be seen as a particular way to suspend and resume computations.

- A WAM-based strategy can resume suspended environments by
  - re-executing a computation path; or
  - restoring a computation path using a forward trail.

Implementation of Tabling: Table Access Mechanisms

- Subgoal Check/Insert
  - Tabling: NEW (NON-SUBSUMING) SUBGOAL
  - Magic: Creating a delta set of magic facts
- Answer Check/Insert
  - Tabling: Interning an answer in the table
  - Magic: Creating a delta set of non-magic facts
- Answer Backtracking
  - Tabling: (NON-SUBSUMING) ANSWER RESOLUTION
  - Magic: Joining a delta set of magic or non-magic facts.

Examples of Implementation Structures

- Coral uses hash-consed values for ground terms.
- XSB uses tries (Implemented by P. Rao [84]).

Implementation of Tabling: Table Access

```
rt(b,V,d)                  rt(a,g(b,c),c)
rt(a,f(a,b),a)             rt(a,f(a,V),V)
```

Tries allow check/insert in a single pass and makes the duplicate check nearly costless.

Implementation of Tabling

Issue: How to incrementally complete a table.

- XSB uses a stack-based mechanism [21]
- Vanilla Magic uses a statically defined control strategy.
- Ordered search uses a dynamic control strategy [79]

Issue: How to handle unknown/undefined literals

- One issue involves dynamically changing the computation rule
- A second issue involves representing atoms that are neither true nor false.

- XSB implements delay and simplification [95]
- WFOS [101] uses the Alternating Fixpoint of [113]

Implementation of Tabling: Optimizations

- Tabling is weak for acyclic right-recursive queries

Left:

```
ancestor(X,Y):- parent(X,Y).
ancestor(X,Y):- ancestor(X,Z),parent(Z,Y).
```

Right:

```
ancestor(X,Y):- parent(X,Y).
ancestor(X,Y):- parent(X,Z),ancestor(Z,Y).
```

What if `parent` is a chain of length N?

Then N calls:

```
a(1,X), a(2,X), a(3,X),...,a(n,X)
```

But $\mathcal{O}(N^2)$ answers

```
a(1,2),a(1,3),a(1,4),...,a(1,n)
       a(2,3),a(2,4),...,a(2,n)
              a(3,4),...,a(3,n)
                     :
                        a(n-1,n)
```

Implementation of Tabling: Optimizations

Approaches to right recursion problem

- If the recursion is acyclic and non-repeating, use SLD!
- Use Tail-recursion optimization to only return answers to the original query. Linear in number of answers in this example. [88], [16].
- Transform right recursion into left recursion if possible, using NRSU-factoring [70].
  - This strategy works for right recursion under all query forms, but does not work for instance, for same generation or for the right recursions in the CCS example.

Implementation of Tabling Copy Avoidance

Structural recursion is acyclic for Prolog-style terms

```
append([],L,L).
append([H|T].L,[H|T1]):- append(T.L,T1).
```

which can be seen to have a right recursive form:

```
append([],L,L).
append(Term.L,[H|T1]):- cons(Term,H,T),append(T.L,T1).
```

Consider the query

```
append([a,b,X],[c],Y).
```

The following queries are made

```
append([a,b,X],[c],Y).
append([b,X],[c],Y).
append([X],[c],Y).
append([],[c],Y).
```

Still quadratic *in the size of the first argument* if you must copy from execution area to table. Other possible solutions:

- Intern Ground Structures in Table

- Can, in principle, use structure-sharing techniques for non-ground terms. [33] [104].

Implementation of Tabling

```
join(X,Y):-
    supplemental(X,X2),rel_3(X2,Y).

supplemental(X,X2):-
    rel_1(X,X1),rel_2(X1,X2).
```

Where **join/2** and **supplemental/2** are tabled, may be more efficient than

```
join(X,Y):-
    rel_1(X,X1),rel_2(X1,X2),rel_3(X2,Y).
```

- A simple optimization consists of folding EDB predicates into new tabled predicates. This is called Supplemental Magic Sets [8] or Supplemental Tabling.
- Both rediscover Earley's observation that the complexity of grammar processing is proportional to

the number of non-terminals on the RHS of a production [41].

Summary

- Tabling and Magic are usually different formulations of the same algorithms. Tabling thus provides a potential way to peform disk access efficiently from a logic program.
- Tabling can be tightly coupled with Prolog, so that it is possible to *program* with tabling
- Tabling provides a proper computational basis for certain forms of Non-monotonic reasoning.
- Tabling adds power to logic programming in addressing important application areas such as program verification, execution of program analysis, grammar-processing, and reasoning for intelligent agents.

References

[1] E. Aarts. *Investigations in Logic Language and Computation.* PhD thesis, University of Utrecht, 1995.

[2] J. Alferes, C. Damasio, and L. Pereira. SLX a top-down derivation procedure for programs with explicit negation. In M. Bruynooghe, editor, *Intl Logic Programming Symp*, pages 424–439, 1994.

[3] J. Alferes, C. Damasio, and L. Pereira. A logic programming system for non-monotonic reasoning. *Journal of Automated Reasoning*, 1995.

[4] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, 1988.

[5] F. Banchilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*. ACM, 1986.

[6] F. Banchilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. of SIGMOD 1986 Conf.*, pages 16–52. ACM, 1986.

[7] J. Barklund. Tabulation of functions in logic programs. Technical report, Uppsala University, 1995.

[8] C. Beeri and R. Ramakrishnan. On the power of magic. *J. Logic Programming*, 10(3):255–299, 1991.

[9] C. Beeri, R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Extending the well-founded and valid semantics for aggregateion. In *International Logic Programming Symposium*, 1993.

[10] R. Ben-Eliahu and R. Dechter. Propositional semantics for disjunctive logic programs. In *Joint International Conference and Symposium on Logic Programming*, 1992.

[11] N. Bidoit and C. Froidevaux. Negation by default and unstratifiable logic programs. *Theoretical Computer Science*, 78:85–112, 1991.

[12] H. Blair and V.S. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science*, 68:135–154, 1989.

[13] R. Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16:25–46, 1993.

[14] R. Bol and L. Degerstedt. Tabulated resolution for well-founded semantics. In *Proc. of the Symp. on Logic Programming*, 1993.

[15] R. Bol and L. Degerstedt. The underlying search for magic templates and tabulation. In *Proc. of the Tenth Int'l Conf. on Logic Programming*, pages 793–812, 1993.

[16] S. Brass. SLDMagic — an improved magic set technique. In B. Novikov and J. Schmidt, editors, *Advances in Databases and Information Systems*, 1996.

[17] D.R. Brough and A. Walker. Some practical properties of logic programming interpreters. In H. Aiso, editor, *International Conference on FIfth Generation Computer Systems*, pages 149–158, 1984.

[18] F. Bry. Query evaluation in recursive databases: Bottom-up and top-down reconciled. In *Deductive and Obkect-Oriented Databases*, pages 25–44, 1990.

[19] W. Chen. Query evaluation of alternating fixpoint logic. Technical report, SMU, 1994.

[20] W. Chen and L. Adams. Constructive negation of general logic programs. Technical report, SMU, 1994.

[21] W. Chen, T. Swift, and D.S. Warren. Efficient top-down computation of queries under the well-founded semantics. *J. Logic Programming*, 24(3):161–199, September 1995.

[22] W. Chen and D.S. Warren. Computation of stable models and its integration with logical query evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 1995.

[23] Weidong Chen and David S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[24] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL system prototype. *IEEE Transactions on Knowledge and Data Engineering*, 2:76–89, 1990.

[25] M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a Magic wand. In *ILPS*, pages 114–129. MIT Press, 1993. (To appear in JLP).

[26] M. Consens, A. Mendelzon, and D. Vista. Deductive database support for data visualization. Technical report, U. Toronto, 1994.

[27] C. Damasio. *Paraconsistency and Negation in Logic Programs.* PhD thesis, Univ. Nova de Lisboa, 1995.

[28] C. Damasio, W. Nejdl, L. Pereira, and M. Schroeder. Model-based diagnosis preferences and strategies representation with logic meta-programming. 1995.

[29] S. Dawson, C. R. Ramakrishnan, S. Skiena, and T. Swift. Principles and practice of unification factoring. *ACM Transactions on Programming Languages and Systems*, September 1996.

[30] S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, and T. Swift. Optimizing clause resolution: Beyond unification factoring. In *Proc. of the Int'l Logic Programming Symposium*, 1995.

[31] S. Dawson, C.R. Ramakrishnan, and D.S. Warren. Practical program analysis using general purpose logic programming systems — a case study. In *ACM PLDI*, 1996. To Appear.

[32] I. de Almeida Morá and J. Alferes. Modelling diagnosis systems with logic programming. Technical report, Univ. Nova de Lisboa, 1995.

[33] E. Villemont de la Clergerie. Layer sharing: an improved structure-sharing framework. In *Proc. of the 20th. Symp. on Principles of Programming Languages*, pages 345–359, 1993.

[34] E. Villemont de la Clergerie and B. Lang. Lpda: Another look at tabulation in logic programming. In *International Conference on Logic Programming*, pages 470–488, 1994.

[35] S. Debray and D.S. Warren. Functional computations in logic programs. *ACM TOPLAS*, 11(3):451–481, July 1989.

[36] Lars Degerstedt and Ulf Nilsson. Magic Computation for Well-founded Semantics. In Jürgen Dix, Luis Moniz Pereira, and Teodor C. Przymusinski, editors, *Non-Monotonic Extensions of Logic Programming*, number 927 in LNAI, pages 181–204. Springer-Verlag, June 1994.

[37] M. Derr, S. Morishita, and G. Phipps. Design and implementation of the Glue-Nail database system. In *Proc. of the SIGMOD 1993 Conf.*, pages 147–156. ACM, 1993.

[38] S. Dietrich. *Extension Tables for Recursive Query Evaluation.* PhD thesis, SUNY at Stony Brook, 1987.

[39] F. Dong and L.V. Lakshmanan. Deductive databases with incomplete information. In *Joint International Conference and Symposium on Logic Programming*, 1992.

[40] P. Dung. Negation as hypothesis: An abductive foundation for logic programming. In *International Logic Programming Conference*, pages 1–17, 1991.

[41] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[42] C. Fan and S. Dietrich. Extension table built-ins for prolog. *Software — Practice and Experience*, 22:573–597, 1992.

[43] J. Freire, R. Hu, T. Swift, and D.S. Warren. Parallelizing tabled evaluation. In *7th International PLILP Symposium*, pages 115–132. Springer-Verlag, 1995.

[44] J. Freire, T. Swift, and D.S. Warren. Treating I/O seriously: Resolution reconsidered for disk. Technical report, SUNY at Stony Brook, 1995.

[45] J. Freire, T. Swift, and D.S. Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. In *8th International PLILP Symposium.* Springer-Verlag, 1996.

[46] H. Gao. *Declarative Picture Description and Interpretation in Logic.* PhD thesis, Department of Computer Science, SUNY at Stony Brook, 1993.

[47] M. Gelfond and V. Lifshitz. Representing actions in extended logic programming. In *Joint Int'l Conf and Symp on Logic Programming*, pages 559–573, 1992.

[48] S. Greco, D. Sacca, and C. Zaniolo. Dynamic programming optimization for logic queries with aggregates. In *International Symposium on Logic Programming*, pages 575–589, 1993.

[49] A. Gupta and I. Mumick. Magic sets transformations in non-recursive systems. In *PODS*, pages 354–367, 1992.

[50] J. Han. Compilation and evaluation of linear mutual recursions. *Information Sciences*, 69:157–183, 1993.

[51] J. Han and L. Liu. Efficient evaluation of multiple linear recursions. *IEEE Transactions on Software Engineering*, 17(12):1241–1252, 1991.

[52] J. Han, L. Liu, and Z. Xie. LogicBase: A deductive database system prototype. Technical report, Simon Fraser University, 1994.

[53] M. Hermenegildo, R. Warren, and S.K. Debray. Global flow analysis as a practical compilation tool. *Journal of Logic Programming*, 13(1,2,3 and 4):349–366, 1992.

[54] G. Janssens, M. Bruynooghie, and V. Dumortier. A blueprint for an abstract machine for the abstract interpretation of (constraint) logic programs. In *International Logic Programming Symposium*, pages 336–351, 1995.

[55] B. Jayaraman and K. Moon. Implementation of subset logic progrms. Technical report, Dept. of Computer Science, SUNY Buffalo, 1994.

[56] C. Jonker. *Constraints and Negations in Logic Programs.* PhD thesis, Utrecht University, 1994.

[57] T. Kanamori. Abstract interpretation based on alexander templates. *Journal of Logic Programming*, 15:31–54, 1993.

[58] T. Kanamori and T. Kawamura. Abstract interpretation based on oldt resolution. *Journal of Logic Programming*, 15:1–30, 1993.

[59] D. Kemp and P. Stuckey. Semantics of logic programs with aggregates. In *International Logic Programming Symposium*, pages 387–404, 1991.

[60] D. Kemp and R. Topor. Completeness of a top-down query evaluation procedure for stratified databases. In *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 178–194, 1988.

[61] David B. Kemp, Kotagiri Ramamohanarao, and Zoltan Somogyi. Right-, left- and multi-linear rule transformations that maintain context information. In *Proceedings of the 16th Conference on Very Large Data Bases*, pages 380–391, 1990.

[62] R. Larson, D. S. Warren, J. Freire, and K. Sagonas. *Semantica.* MIT Press, 1995. In preparation.

[63] R. Larson, D. S. Warren, J. Freire, and K. Sagonas. *Syntactica.* MIT Press, 1995.

[64] Alexandre Lefebvre. Recursive aggregates in the eks-v1 system. Technical Report KB34, ECRC, 1991.

[65] U. Manber. *Introduction to Algorithms: A Creative Approach.* Addison-Wesley, 1989.

[66] S. Morishita. An Extension of Van Gelder's Alternating Fixpoint to Magic Programs. *Journal of Computer System Sciences*, 52:506–521, June 1996.

[67] I. Mumick, S. Finklestein, R. Ramakrishnan, and H. Pirahesh. Magic conditions. In *PODS*, 1990.

[68] I. Mumick and H. Pirahesh. Implementation of Magic-sets in a Relational Database System. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 103–114, 1994.

[69] I. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *Proc. of the 16th Int'l Conf. on Very Large Data Bases*, pages 264–277. VLDB End., 1990.

[70] J. Naughton, R. Ramakrishnan, Y. Sagiv, and J. Ullman. Argument reduction through factoring. In *Proc. of the 15th Int'l Conf. on Very Large Data Bases*, pages 173–182. VLDB End., 1989.

[71] Ulf Nilsson. Abstract interpretation: A kind of magic. In *PLILP91*, pages 299–310, 1991.

[72] F.C.N. Pereira and D.H.D. Warren. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, 1983.

[73] G. Phipps, M. Derr, and K. Ross. Glue-Nail: A deductive database system. pages 308–317, 1991.

[74] H. Przymusinska and T. Przymusinski. Weakly perfect model semantics for logic programs. In *5th International Conference and Symposium on Logic Programming*, pages 1106–1123, 1988.

[75] T.C. Przymusinski. On the declarative semantics of deductive databases and logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.

[76] T.C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *PODS*, pages 11–21, 1989.

[77] R. Ramakrishnan, editor. *Applications of Logic Databases.* Kluwer Academic Publishers, 1995.

[78] R. Ramakrishnan, C. Beeri, and R. Krishnamurthi. Optimizing existential datalog queries. In *Proc. of the ACM Symp. on Principles of Database Systems*, pages 89–102. ACM, 1988.

[79] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In *Proc. of the Joint Int'l Conf. and Symp. on Logic Programming*, 1992.

[80] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. Implementation of the CORAL Deductive Database System. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 167–176, 1993.

[81] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1995.

[82] R. Ramesh and W. Chen. A portable method of integrating SLG resolution into Prolog systems. In *Proc. of the Symp. on Logic Programming*, 1994.

[83] P. Rao, C.R. Ramakrishnan, and I.V. Ramakrishnan. A thread in time saves tabling time. In *1996 Joint International Conference and Symposium on Logic Programming*, 1996.

[84] P. Rao, I.V. Ramakrishnan, K. Sagonas, T. Swift, and D.S. Warren. Efficient table access mechanisms for logic programs. In *International Conference on Logic Programming*, 1995. To Appear.

[85] T. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*. Kluwer Academic, 1994.

[86] T. Reps. Shape analysis as a generalized path problem. In *ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 1–11. ACM Press, 1995.

[87] J. Rohmer, R. Lescoeur, and J. Kerisit. The Alexander method: a technique for the processing of recursive atoms in deductive databases. *New Generation Computing*, 4:522–528, 1986.

[88] K. Ross. Modular acyclicity and tail recursion in logic programs. In *Proc. of 10th PODS*, 1991.

[89] K.A. Ross. Modular stratification and magic sets for datalog programs with negation. In *JACM*, pages 1216–1266, 1994.

[90] D. Sacca and C. Zaniolo. The generalized counting method for recursive logic queries. *Theoretical Computer Science*, 62:187–220, 1989.

[91] Y. Sagiv. Is there anything better than magic? In *Proc. of the 1990 North American Conference*, pages 235–254, 1990.

[92] K. Sagonas. *The SLG-WAM: A Search-Efficient Engine for Well-Founded Evaluation of Normal Logic Programs.* PhD thesis, SUNY at Stony Brook, 1996.

[93] K. Sagonas, T. Swift, and D.S. Warren. XSB as an efficient deductive database engine. In *Proc. of SIGMOD 1994 Conf.* ACM, 1994.

[94] K. Sagonas, T. Swift, and D.S. Warren. The limits of fixed-order computation. Technical report, SUNY at Stony Brook, 1995.

[95] K. Sagonas, T. Swift, and D.S. Warren. An abstract machine for computing the well-founded semantics. In *Joint International Conference and Symposium on Logic Programming.*, 1996.

[96] K. Sagonas, T. Swift, and D.S. Warren. An abstract machine for fixed-order stratified programs. In *Proc. of 13th Conference on Automated Deduction.*, 1996.

[97] H. Seki. On the power of Alexandrer templates. In *Proc. of 8th PODS*, pages 150–159. ACM, 1989.

[98] H. Seki and H. Itoh. A query evaluation method for stratified programs under the extended CWA. In *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 195–211, 1988.

[99] O. Shmueli, S. Tsur, and C. Zaniolo. Compilation of set terms in the logic data language (LDL). *Journal of Logic Programming*, 12:89–119, 1992.

[100] S. Smolka, O. Sokolsky, and S. Zhang. Model checking in the modal μ-calculus. In *IEEE LICS*, 1994.

[101] P. Stuckey and S. Sudarshan. Well-founded ordered search. In *13th conference on Foundations of Software Technology and Theoretical Computer Science*, pages 161–172, 1993.

[102] S. Sudarshan. *Optimizing Bottom-up Query Evaluation for Deductive Databases.* PhD thesis, University of Wisconsin, 1992.

[103] S. Sudarshan and R. Ramakrishnan. Aggregation and relevance in deductive databases. In *Proc. of the 17th Int'l Conf. on Very Large Data Bases*, pages 501–511. VLDB End., 1991.

[104] S. Sudarshan and R. Ramakrishnan. Optimizations of bottom-up evaluation with non-ground terms. In *Proc. of the Symp. on Logic Programming*, 1993.

[105] T. Swift. *Efficient Evaluation of Normal Logic Programs.* PhD thesis, SUNY at Stony Brook, 1994.

[106] T. Swift and D. S. Warren. An abstract machine for SLG resolution: definite programs. In *Proceedings of the Symposium on Logic Programming*, pages 633–654, 1994.

[107] T. Swift and D. S. Warren. Analysis of sequential SLG evaluation. In *Proceedings of the Symposium on Logic Programming*, pages 219–238, 1994.

[108] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *Third Int'l Conf. on Logic Programming*, pages 84–98, 1986.

[109] F. Teusink. A proof procedure for extended logic programs. In *International Logic Programming Symposium*, pages 235–249, 1993.

[110] D. Toman. Top-down beats bottom-up for constraint extensions of datalog. In *International Logic Programming Symposium*, pages 98–115, 1995.

[111] J. Ullman. Bottom-up beats top-down for datalog. In *Proc. of 8th PODS*, pages 140–149. ACM, 1989.

[112] J. Vaghani, K. Ramamohanorao, D. Kemp, Z. Somogyi, and P. Stuckey. Design overview of the Aditi deductive database system. In *Seventh Int'l Conf. on Data Engineering*, pages 240–247, 1991.

[113] A. van Gelder. The alternating fixpoint of logic programs with negation. In *Proc. of 8th PODS*, pages 1–10. ACM, 1989.

[114] A. van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *JACM*, 38(3):620–650, 1991.

[115] L. Vieille. Recursive query processing: The power of logic. *Theoretical Computer Science*, 69:1–53, 1989.

[116] L. Vieille, P. Bayer, V. Kuchenhoff, and A. Lefebvre. EKS-V1, a short overview. In *AAAI-90 Workshop on Knowledge Base Management Systems*, 1990.

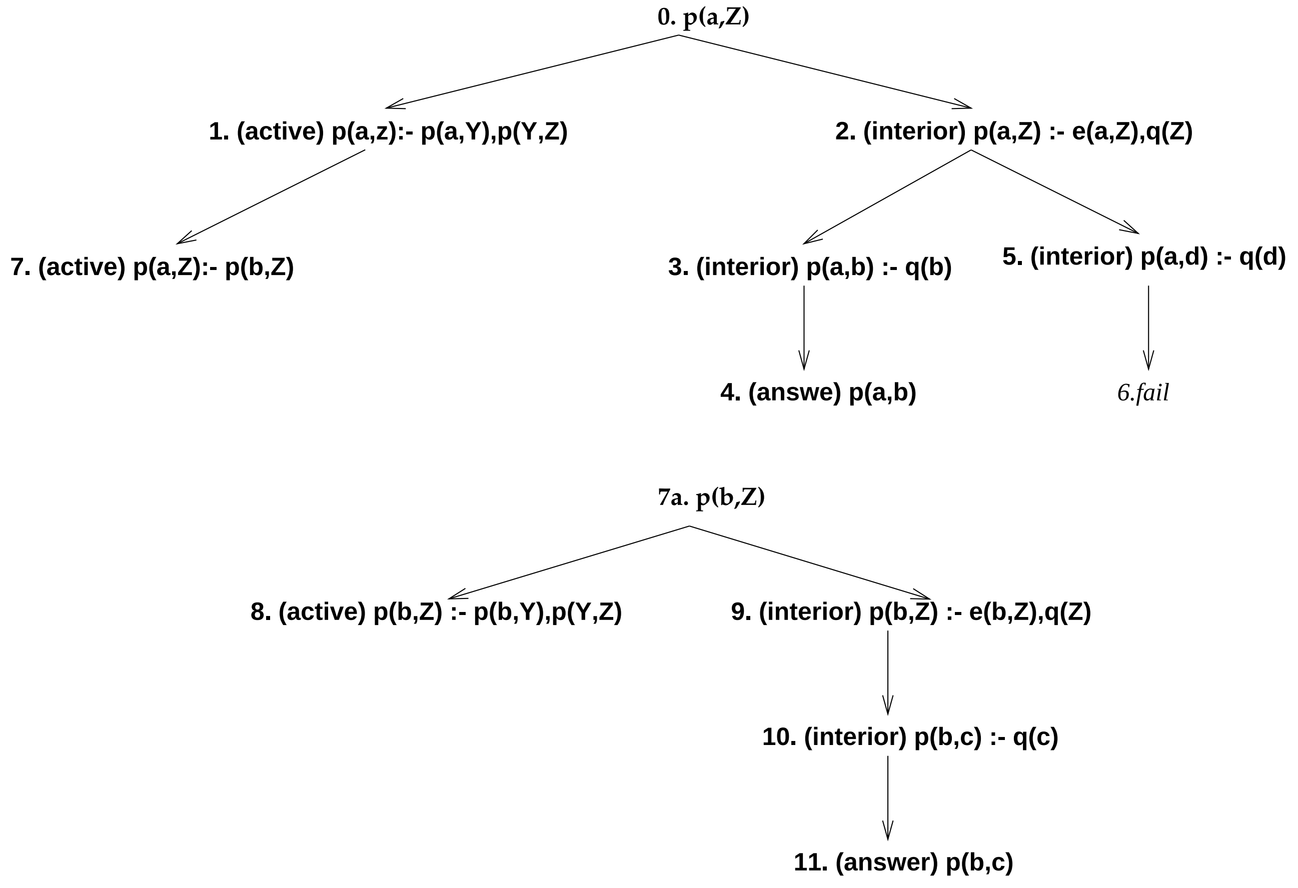
[117] G. Wagner. Reasoning with inconsistency in extended deductive databases. In *International Workshop on Logic Programming and Non-Monotonic Reasoning*, pages 300–315, 1994.

[118] A. Walker. The Syllog expert database system. Technical report, IBM, 1992.

[119] A. Walker. Backchain iteration: Towards a practical inference method that is simple enough to be proved terminating, sound, and complete. *J. Automated Reasoning*, 11(1):1–23, 1993. Originally formulated in New York University TR 34, 1981.

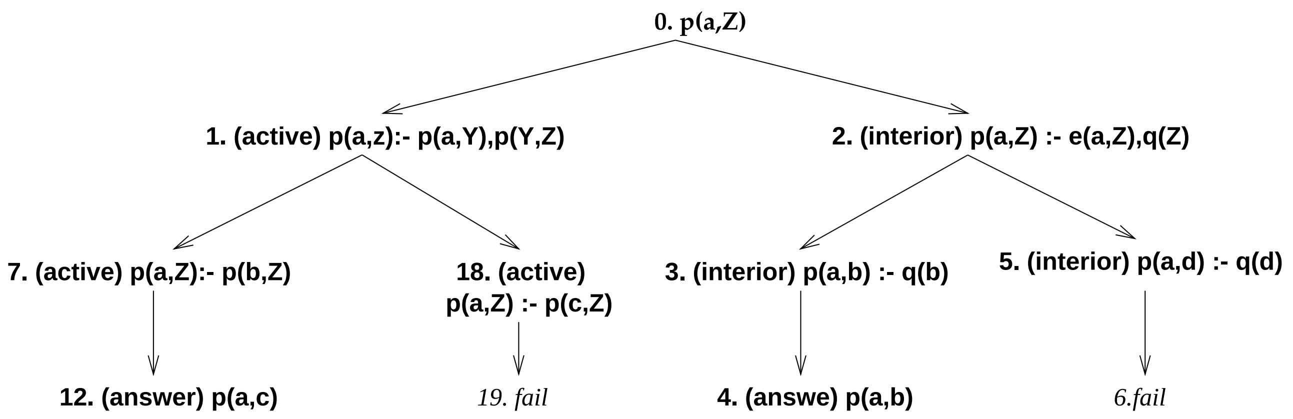
[120] D. S. Warren. Memoing for logic programs with applications to abstract interpretatino and partial deduction. *Communications of the ACM*, 1992.

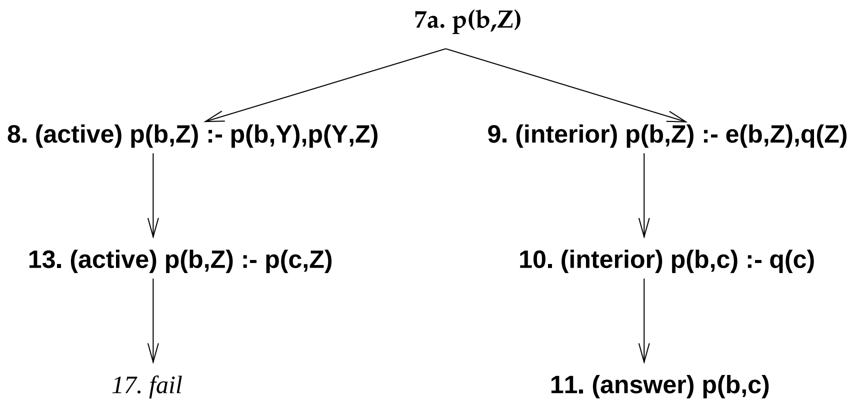
[121] J. Wunderwald. A portable implementation of memoing evaluation. In *LOPSTR 95*, 1995.


0. p(a,Z)
1. (active) p(a,z):- p(a,Y),p(Y,Z)
2. (interior) p(a,Z) :- e(a,Z),q(Z)
7. (active) p(a,Z):- p(b,Z)
3. (interior) p(a,b) :- q(b)
5. (interior) p(a,d) :- q(d)
4. (answe) p(a,b)
6.fail
7a. p(b,Z)
8. (active) p(b,Z) :- p(b,Y),p(Y,Z)
9. (interior) p(b,Z) :- e(b,Z),q(Z)
10. (interior) p(b,c) :- q(c)
11. (answer) p(b,c)

Subgoal	Answers
p(a,Z)	p(a,b)
p(b,Z)	p(b,c)

```
:- table p/2.

p(X,Z) :- p(X,Y),p(Y,Z).
p(X,Z) :- e(X,Z),q(Z).

e(a,b).  e(a,d).  e(b,c).

q(a).  q(b).  q(c).
```

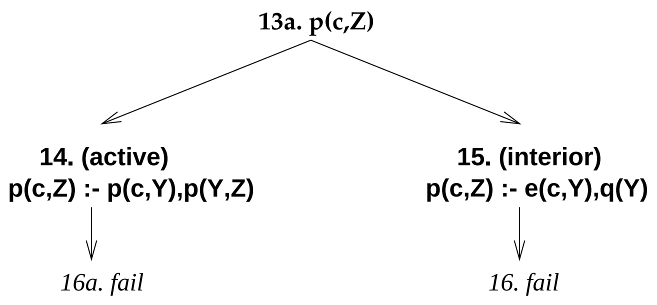

0. p(a,Z)
1. (active) p(a,z):- p(a,Y),p(Y,Z)
2. (interior) p(a,Z) :- e(a,Z),q(Z)
7. (active) p(a,Z):- p(b,Z)
18. (active) p(a,Z) :- p(c,Z)
3. (interior) p(a,b) :- q(b)
5. (interior) p(a,d) :- q(d)
12. (answer) p(a,c)
19. fail
4. (answe) p(a,b)
6.fail


7a. p(b,Z)
8. (active) p(b,Z) :- p(b,Y),p(Y,Z)
9. (interior) p(b,Z) :- e(b,Z),q(Z)
13. (active) p(b,Z) :- p(c,Z)
10. (interior) p(b,c) :- q(c)
17. fail
11. (answer) p(b,c)

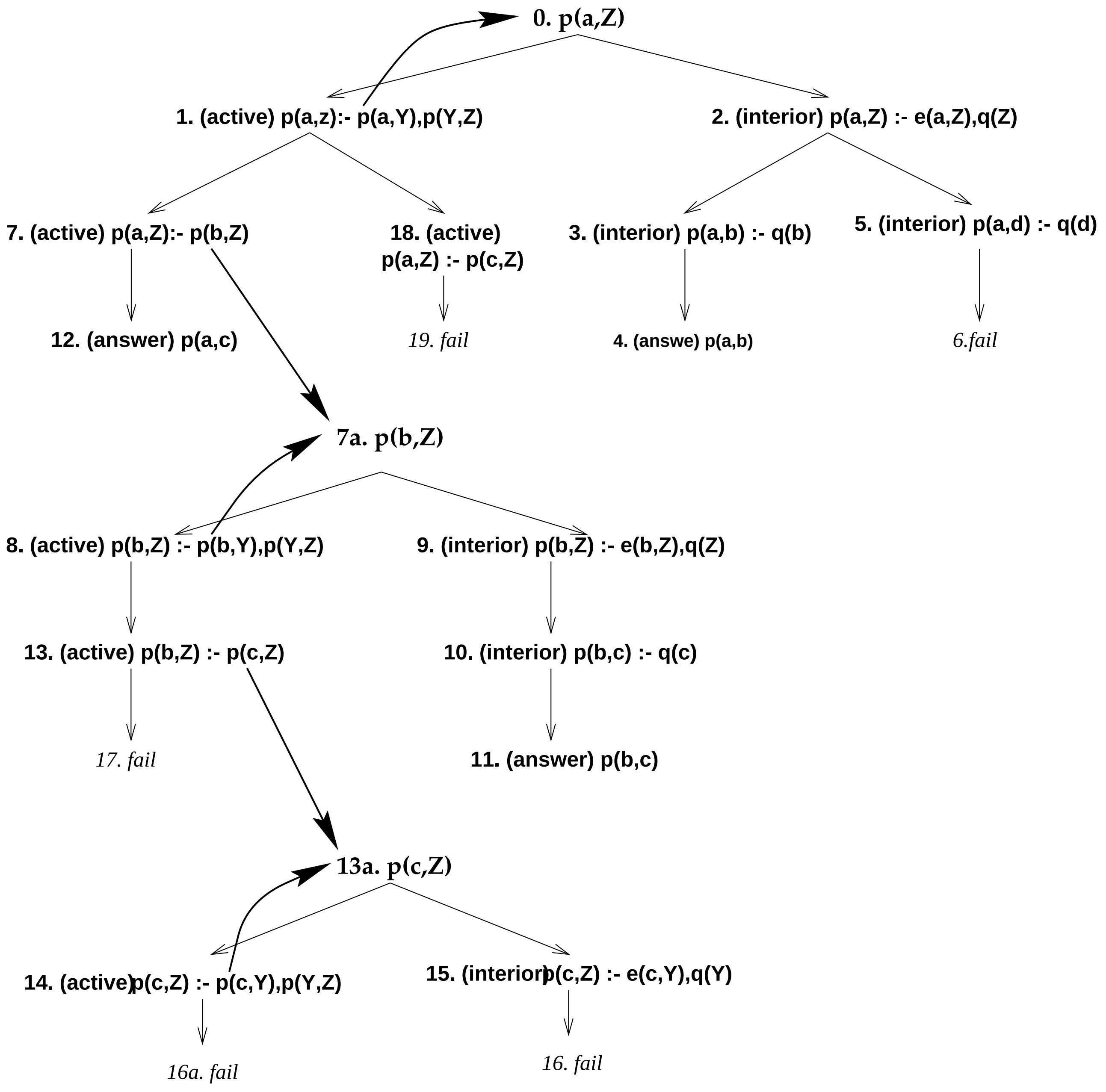
Subgoal	Answers

p(a,Z)	p(a,b)
	p(a,c)

p(b,Z)	p(b,c)
p(c,Z)	

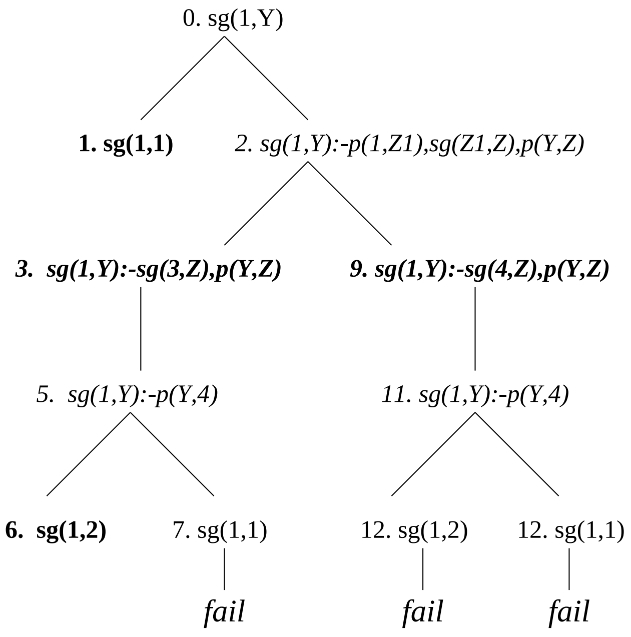

13a. p(c,Z)
14. (active) p(c,Z) :- p(c,Y),p(Y,Z)
15. (interior) p(c,Z) :- e(c,Y),q(Y)
16a. fail
16. fail

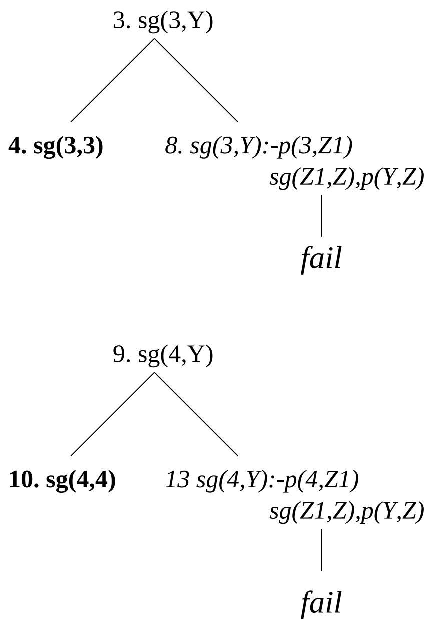
```
:- table p/2.

p(X,Z) :- p(X,Y),p(Y,Z).
p(X,Z) :- e(X,Z),q(Z).

e(a,b).  e(a,d).  e(b,c).

q(a).  q(b).  q(c).
```



0. p(a,Z)
1. (active) p(a,z):- p(a,Y),p(Y,Z)
2. (interior) p(a,Z) :- e(a,Z),q(Z)
7. (active) p(a,Z):- p(b,Z)
18. (active) p(a,Z) :- p(c,Z)
3. (interior) p(a,b) :- q(b)
5. (interior) p(a,d) :- q(d)
12. (answer) p(a,c)
19. fail
4. (answwe) p(a,b)
6.fail
7a. p(b,Z)
8. (active) p(b,Z) :- p(b,Y),p(Y,Z)
9. (interior) p(b,Z) :- e(b,Z),q(Z)
13. (active) p(b,Z) :- p(c,Z)
10. (interior) p(b,c) :- q(c)
17. fail
11. (answer) p(b,c)
13a. p(c,Z)
14. (active) p(c,Z) :- p(c,Y),p(Y,Z)
15. (interior) p(c,Z) :- e(c,Y),q(Y)
16a. fail
16. fail

		a	*b*	*e*	*t*
		1	2	3	4
b	1	1	1	2	3
e	2	2	2	1	3
g	3	3	3	2	2





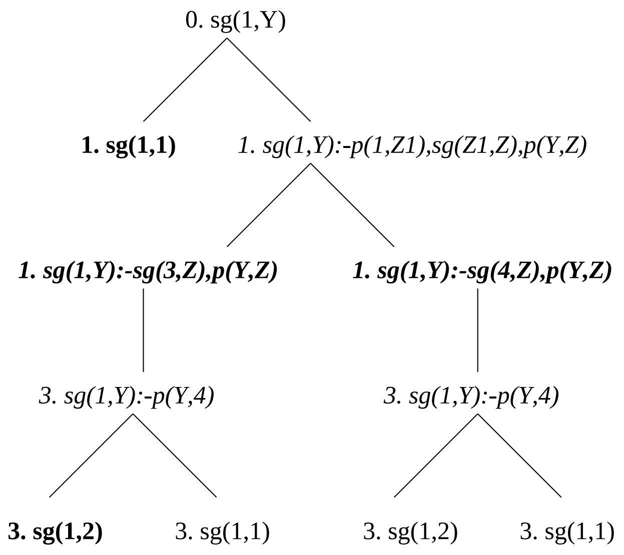
3. sg(3,Y)
4. sg(3,3)
8. sg(3,Y):-p(3,Z1) sg(Z1,Z),p(Y,Z)
fail
9. sg(4,Y)
10. sg(4,4)
13 sg(4,Y):-p(4,Z1) sg(Z1,Z),p(Y,Z)
fail

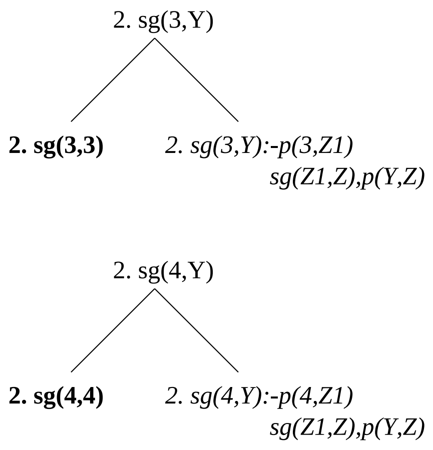
```
sg(X,X).
sg(X,Y):- p(X,Z1),sg(Z1,Z),p(Y,Z).


p(1,3)          p(1,4)

p(2,3).         p(2,4).
```

Subgoals	Answers

sg(1,Y)	sg(1,1)
	sg(1,2)

sg(2,Y)	sg(2,2)
sg(3,Y)	sg(3,3)





Subgoals	Answers

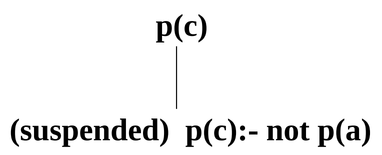
sg(1,Y)	sg(1,1)
	sg(1,2)

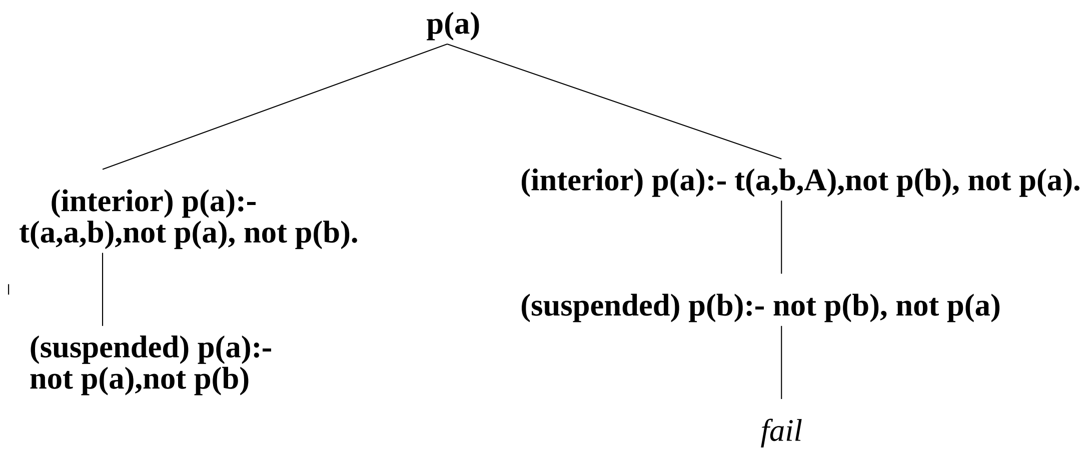
sg(2,Y)	sg(2,2)
sg(3,Y)	sg(3,3)

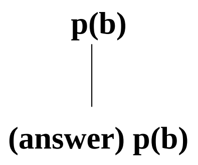
$$\begin{tikzcd}
 & \text{dynamically stratifed} & \\
\text{lr-dynamically stratified} \arrow[ur] & & \text{weakly stratified} \arrow[ul] \\
\text{lr-weakly stratified} \arrow[u] \arrow[urr] & & \\
 & \text{(lr)-modularly stratified} \arrow[ul] \arrow[uur] & \\
 & \text{locally stratifed} \arrow[u] & \\
 & \text{predicate stratified} \arrow[u] &
\end{tikzcd}$$

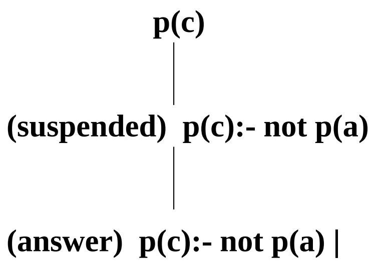
$$\begin{tikzcd}
 & \text{get\_best\_choices/3} \arrow[d] & \\
 & \text{best\_choice/3} \arrow[loop right] \arrow[dl, "\text{not}"'] \arrow[dr] & \\
\text{better\_choice/3} \arrow[d] \arrow[drr] & & \text{just\_as\_good\_choice/3} \arrow[dll] \arrow[d] \\
\text{can\_teach/2} & & \text{rates/2}
\end{tikzcd}$$

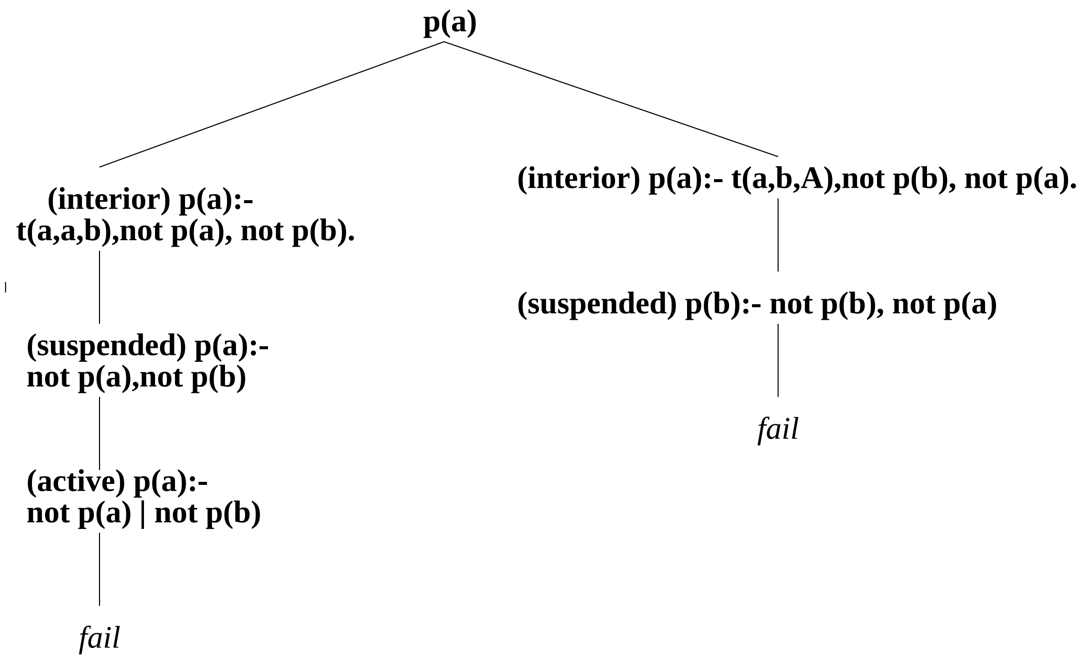
$$\begin{tikzcd}
 & \text{demo(get\_best\_choices/3)} \arrow[d] & \\
 & \text{demo(best\_choice/3)} \arrow[loop right] \arrow[dl, "\text{not}"'] \arrow[dr] & \\
\text{demo(better\_choice/3)} \arrow[d] \arrow[drr] & & \text{demo(ust\_as\_good\_choice/3)} \arrow[dll] \arrow[d] \\
\text{demo(can\_teach/2)} & & \text{demo(rates/2)}
\end{tikzcd}$$

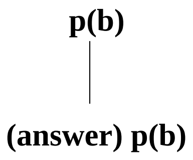

p(c)
(suspended) p(c):- not p(a)

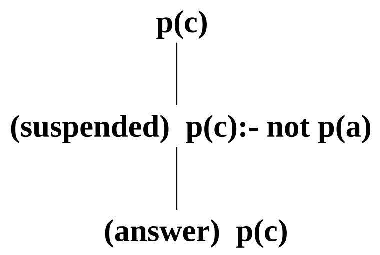

p(a)
(interior) p(a):- t(a,a,b),not p(a), not p(b).
(interior) p(a):- t(a,b,A),not p(b), not p(a).
(suspended) p(a):- not p(a),not p(b)
(suspended) p(b):- not p(b), not p(a)
fail

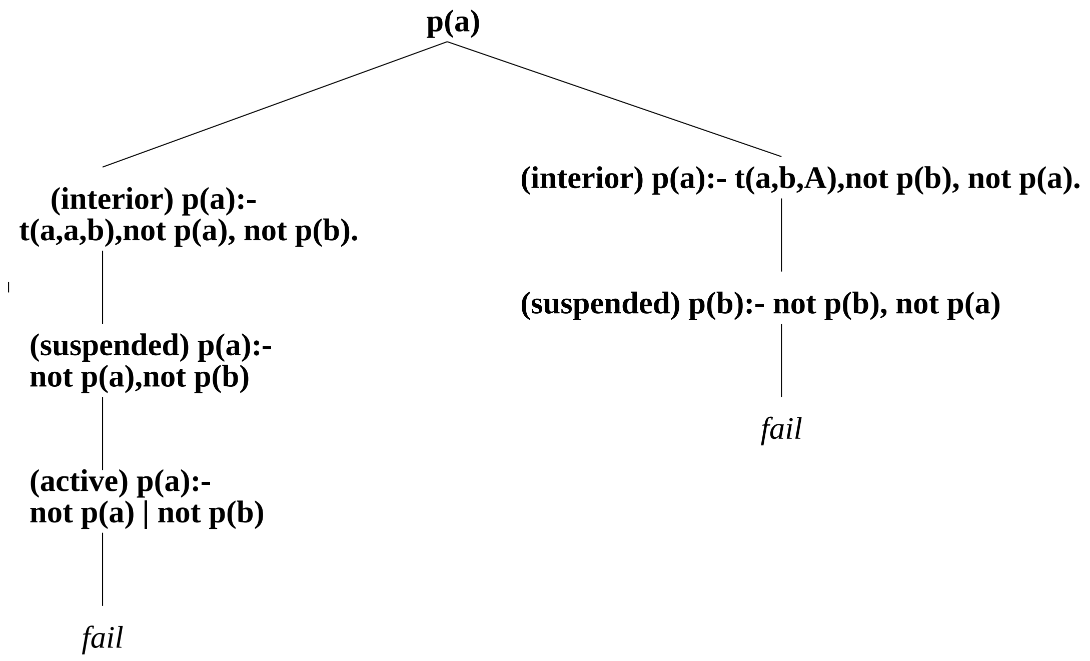

p(b)
(answer) p(b)

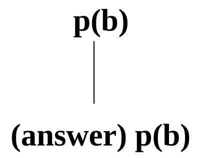

p(c)
(suspended) p(c):- not p(a)
(answer) p(c):- not p(a) |

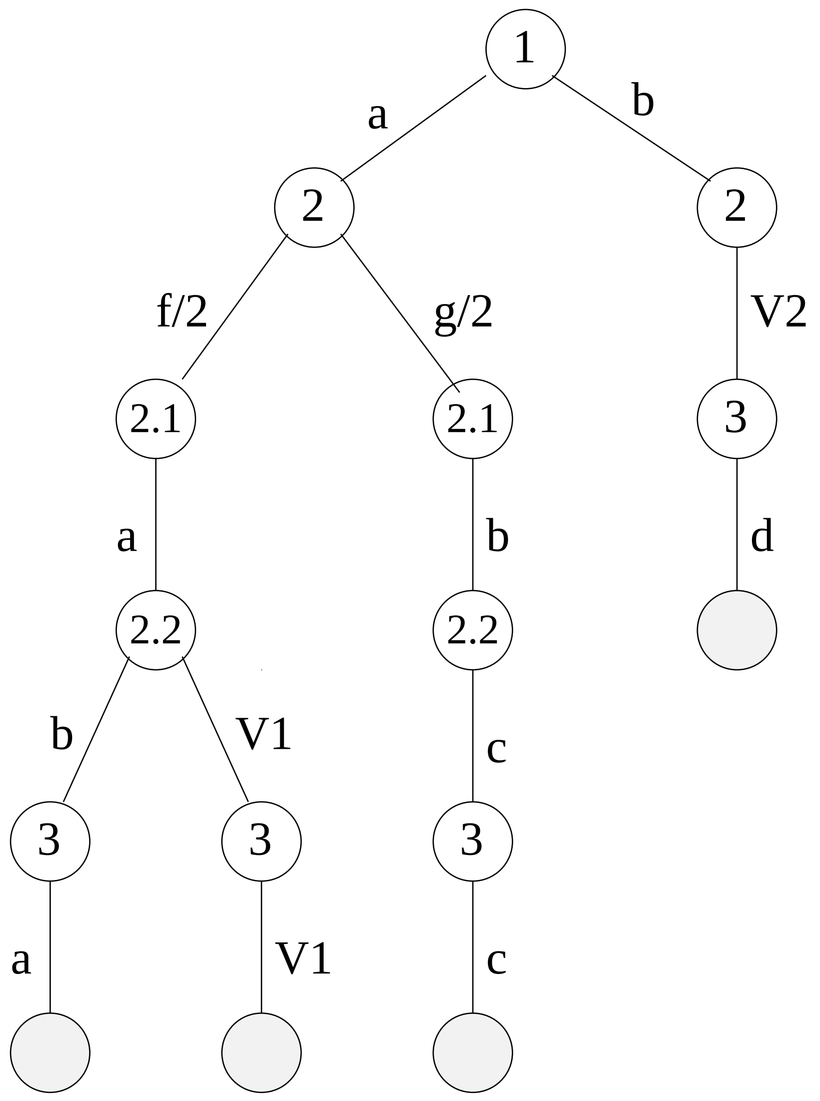

p(a)
(interior) p(a):-
t(a,a,b),not p(a), not p(b).
(suspended) p(a):-
not p(a),not p(b)
(active) p(a):-
not p(a) | not p(b)
fail
(interior) p(a):- t(a,b,A),not p(b), not p(a).
(suspended) p(b):- not p(b), not p(a)
fail


p(b)
(answer) p(b)


p(c)
(suspended) p(c):- not p(a)
(answer) p(c)


p(a)
(interior) p(a):- t(a,a,b),not p(a), not p(b).
(interior) p(a):- t(a,b,A),not p(b), not p(a).
(suspended) p(a):- not p(a),not p(b)
(suspended) p(b):- not p(b), not p(a)
(active) p(a):- not p(a) | not p(b)
fail
fail


p(b)
(answer) p(b)


1
a
b
2
2
f/2
g/2
V2
2.1
2.1
3
a
b
d
2.2
2.2
b
V1
c
3
3
3
a
V1
c