Praktische Informatik 2

Sommer-Semester 2007

Prof. Dr. sc. Hans-Dieter Burkhard

www.ki.informatik.hu-berlin.de

Praktische Informatik 2

Themen

Alternative Programmiermethoden:

Deklaratives (logisches Programmieren): Prolog

Auf den Poolrechnerrn:

/usr/local/praktikum/SWIprolog/bin/pl

Abstrakte Datenstrukturen

(Listen, Graphen, Bäume ...)

Repräsentation von „Wissen“

Lexikon

Datenbank

Briefmarkensammlung

Programme

Landkarten

Bilder

Gesetze

Regeln

. . .

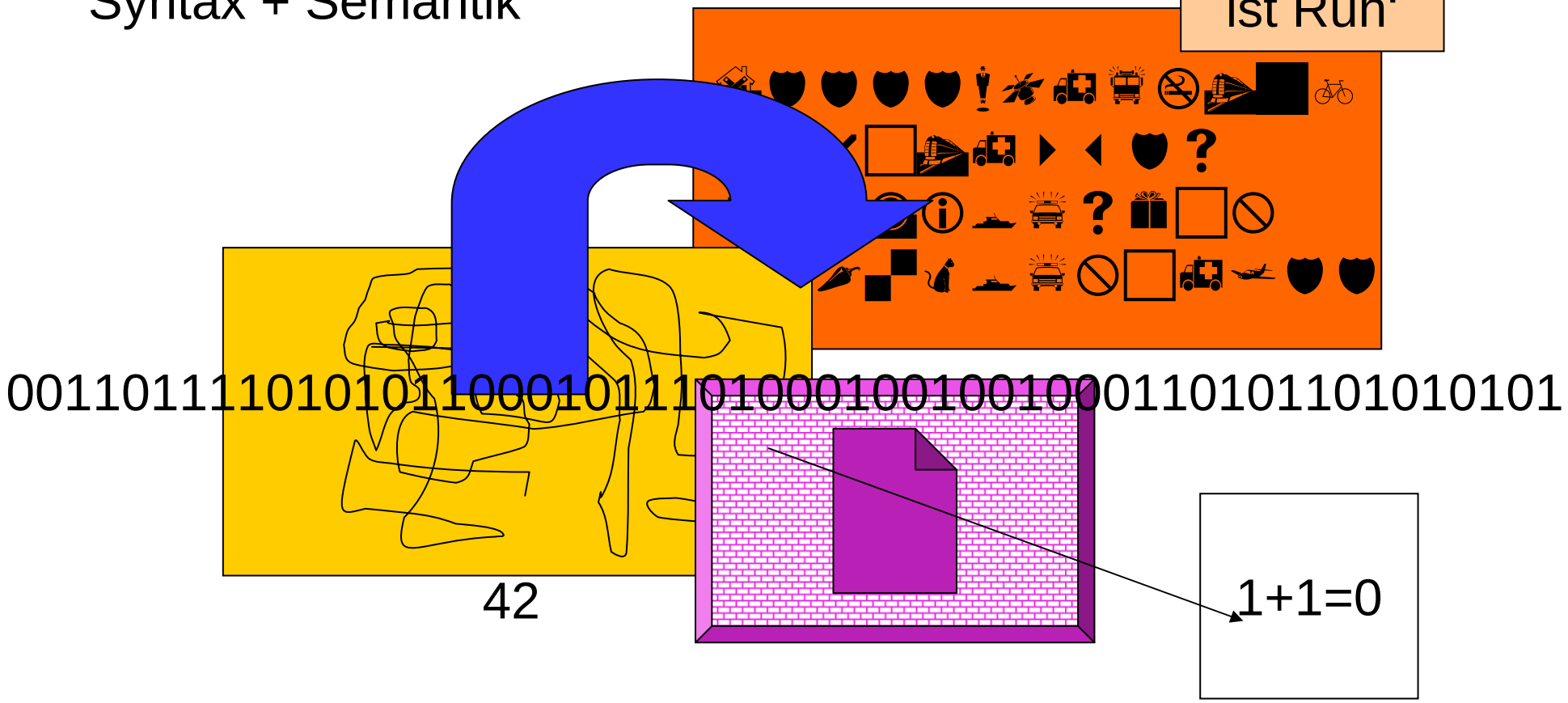
Darstellung und Interpretation

- Menschlicher Nutzer
- Maschinelle Auswertung

Zeichen, Symbole, Signale, ...

Zeichen/Symbole + Interpretation

Syntax + Semantik



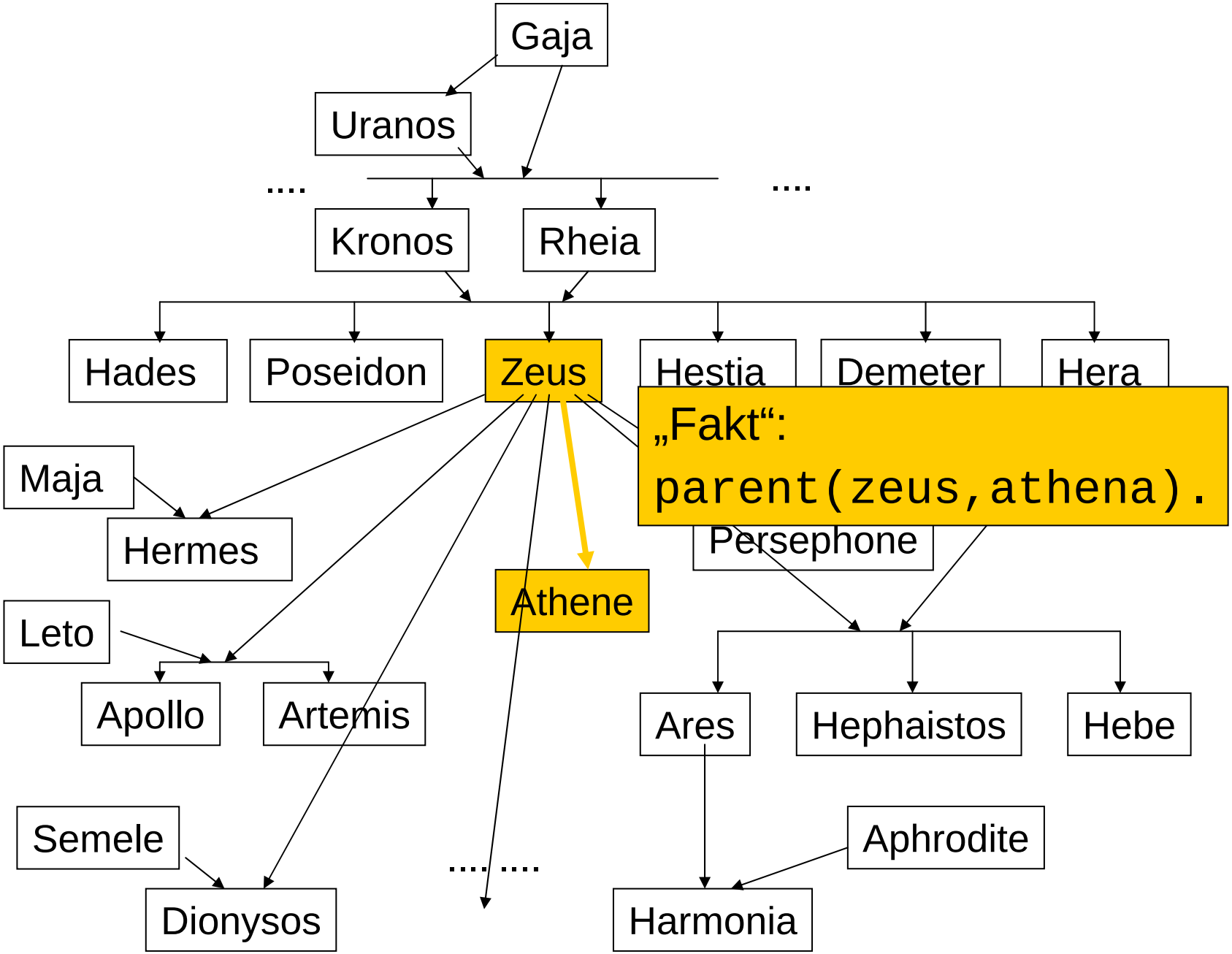
Explizites vs. Implizites Wissen

explizit: z.B. Axiome, Schluss-Regeln

Implizit: Folgerungen

Unterschiedliche Formen für
- Menschlicher Nutzer
- Maschinelle Auswertung

Inferenz: Verfahren zur Herleitung von implizitem Wissen aus explizitem Wissen (z.B. Beweisverfahren)

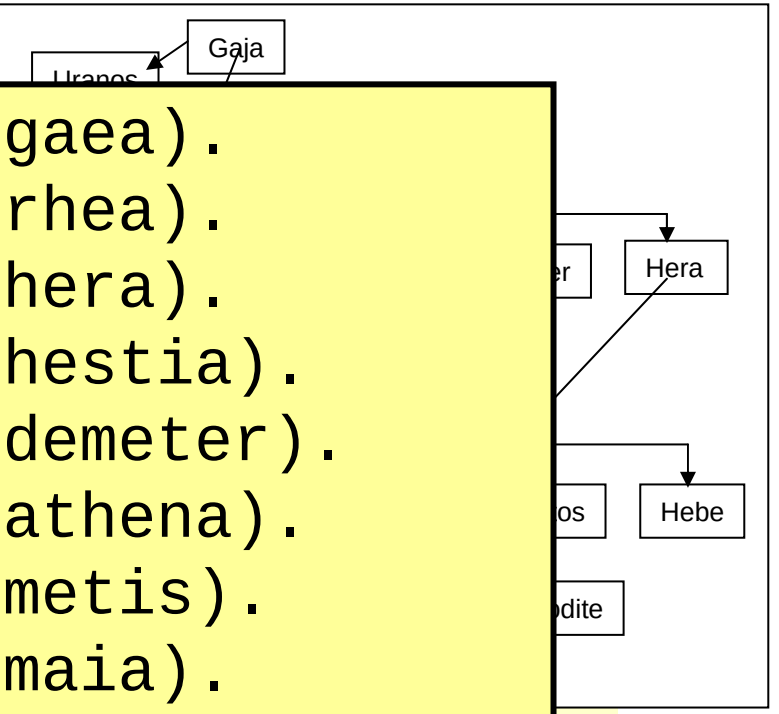

Gaja
Uranos
....
....
Kronos
Rheia
Hades
Poseidon
Zeus
Hestia
Demeter
Hera
„Fakt“:
parent(zeus,athena).
Maja
Hermes
Persephone
Athene
Leto
Apollo
Artemis
Ares
Hephaistos
Hebe
Semele
Aphrodite
....
Dionysos
Harmonia

Griechische Götter: Fakten

```
parent(uranus, cronus).
parent(gaea,   ur
parent(gaea,   cr
pare
pare
pare
pare
pare
pare
pare
...
```

```
male(uranus).
male(cronus).
male(zeus).
male(hades).
male(hermes).
male(apollo).
male(dionysi
male(hephaes
male(poseido
```

```
female(gaea).
female(rhea).
female(hera).
female(hestia).
female(demeter).
female(athena).
female(metis).
female(maia).
female(persephone).
female(aphrodite).
female(artemis).
female(leto).
```




```
parent(ze
parent(he                                    ).
parent(zeus,   hephaestus).
parent(hera,   hephaestus).
...
parent(demeter, persephone).
```

Fakten in Prolog

Ein Fakt hat die Form

```
funktor(argumente).
```

funktor ist der Name einer n-stelligen Relation (Prädikat).
n ist die Anzahl der Argumente.

```
male(zeus).
parent(zeus,athena).
parent(zeus,hera,ares).
plus(3,4,7).
kleiner(6,9).
```

male/1
parent/2
parent/3
plus/3
true/0
fail/0

Definition einer Relation durch Aufzählung

Aufzählung: Die Beziehungen werden explizit aufgezählt (Faktenmenge, Datenbank)

```
parent(uranus, cronus).
parent(gaea,
parent(gaea,
parent(rhea,
parent(cronus,
parent(rhea,
parent(cronus,
parent(cronus,
parent(rhea,
parent(cronus,
parent(rhea,
parent(zeus,   hermes).
parent(maia,   hermes).
...
```

```
male(dionysius).
male(hephaestus).
male(poseidon).
```

```
female(gaea).
female(rhea).
                ).
                ia).
                ter).
                na).
                s).
female(maia).
e(persephone).
e(aphrodite).
e(artemis).
e(leto).
```

Hinzufügen:

```
asserta(male(caesar)).
assertz(male(augustus)).
```

Löschen:

```
retract(male(zeus)).
```

Anfragen

Eine Anfrage hat die Form `?-funktor(argumente).`

funktor ist der Name einer n-stelligen Relation (Prädikat).

```
?- male(zeus).
yes.
?- parent(zeus,athena).
yes.
?-parent(hera,athena).
no.
```

```
parent(uranus, cronus).
parent(gaea,   cronus).
parent(gaea,   rhea).
parent(rhea,   zeus).
parent(cronus, zeus).
parent(rhea,   hera).
parent(cronus, hera).
parent(cronus, hades).
parent(rhea,   hades).
parent(cronus, hestia
parent(rhea,   hestia
parent(zeus,   hermes
parent(maia,   hermes
.
```

```
female(gaea).
female(rhea).
female(hera).
female(hestia).
female(demeter).
female(athena).
female(metis).
female(maia).
female(persephone).
female(aphrodite).
```

```
male(uranus).
male(cronus).
male(zeus).
male(hades).
male(hermes).
male(apollo).
male(dionysius).
male(hephaestus).
male(poseidon).
```

CWA: Closed World Assumption (1a)

Bedeutung der Antwort „no“?

Oder:

Welche Antwort gibt Prolog

für nicht gespeicherte Fakten?

```
?-vater(zeus,athena).
```

„Optimistische“ Variante: `yes.`

„Pessimistische“ Variante: `no.`

CWA: Closed World Assumption (1a)

(Vorläufige) Bedeutung der Antwort „no“:
Der Fakt ist in der Datenbank nicht enthalten.

```
?-parent(hera,athena).
no.
?-parent(zeus,hera,ares).
no.
?-kleiner(3,7).
no.
```

```
parent(uranus, cronus).
parent(gaea,   cronus).
rent(gaea,   rhea).
rent(rhea,   zeus).
rent(cronus, zeus).
rent(rhea,   hera).
rent(cronus, hera).
rent(cronus, hades).
rent(rhea,   hades).
rent(cronus, hestia
rent(rhea,   hestia
rent(zeus,   hermes
rent(maia,   hermes
.
```

```
female(gaea).
female(rhea).
female(hera).
female(hestia).
female(demeter).
female(athena).
female(metis).
female(maia).
female(persephone).
female(aphrodite).
```

```
male(uranus).
male(cronus).
male(zeus).
male(hades).
male(hermes).
male(apollo).
male(dionysius).
male(hephaestus).
male(poseidon).
```

Anfragen nach Existenz

Anfrage enthält Variable `?-funktor(argumente).`

Variablen-Bezeichner beginnen mit Großbuchstaben

```
?- parent(zeus,X).
X=hermes?
;
X=athena?
;
X=ares?
;
no.
```

```
?- parent(zeus,X).
X=hermes?
;
X=athena?
.
yes.
```

```
parent(zeus,   hermes
parent(maia,   hermes
...
```

```
male(zeus).
male(hades).
male(hermes).
male(apollo).
male(dionysius).
male(hephaestus).
```

„ ; “ erwartet weitere Antworten. „ .“ schließt Anfrage ab.

CWA: Closed World Assumption (1b)

Bedeutung der Antwort „no“ ?

(Vorläufige) Bedeutung der Antwort „no“:
Es gibt keinen passenden Fakt in der Datenbank.

```
parent(rhea,    zeus).
parent(cronus,  zeus).
parent(rhea,    hera).
parent(cronus,  hera).
rent(cronus,  hades).
rent(rhea,    hades).
rent(cronus,  hestia
rent(rhea,    hestia
rent(zeus,    hermes
rent(maia,    hermes
.
```

```
female(hestia).
female(demeter).
female(athena).
female(metis).
female(maia).
female(persephone).
female(aphrodite).
```

```
male(uranus).
male(cronus).
male(zeus).
male(hades).
male(hermes).
male(apollo).
male(dionysius).
male(hephaestus).
male(poseidon).
```

```
?-father(X,Y).
no.
?-kleiner(M,N).
no.
?-parent(X,gaea).
no.
```

Relationen definieren durch Regeln

Eine Regel hat die Form

```
kopf:-körper.
```

```
goal:-subgoals.
```

Mit der intuitiven Bedeutung:

„goal“ gilt (ist beweisbar),
falls alle „subgoals“ gelten (beweisbar sind).

Relationen definieren durch Regeln

Definition der Relation „Vater-Kind“:

```
father(Vater,Kind)
      :-parent(Vater,Kind), male(Vater).
```

```
father(X,Y):-parent(X,Y),male(X).
mother(X,Y):-parent(X,Y),female(X).
```

```
parent(X,Y,Z) :-father(X,Z),mother(Y,Z).
```

```
son(Sohn,Elternteil):-
parent(Elternteil,Sohn),male(Sohn).
```

```
grandfather(X,Z):-father(X,Y),parent(Y,Z).
grandmother(X,Z):-mother(X,Y),parent(Y,Z).
```

```
grandchild(X,Y):-grandfather(Y,X).
grandchild(X,Y):-grandmother(Y,Z).
```

Regel als logische Formel

Die Variablen in Regeln sind universell quantifiziert.
Die Relationen der rechten Seite sind konjunktiv verknüpft.

$goal(X_1,...,X_n)$:- $subgoal_1(X_1,...,X_n)$,..., $subgoal_m(X_1,...,X_n)$.

Wird aufgefasst als

$\forall X_1 ... \forall X_n$

$[\ subgoal_1(X_1,...,X_n) \wedge ... \wedge subgoal_m(X_1,...,X_n) \rightarrow goal(X_1,...,X_n)\ ]$

oder

$\forall X_1 ... \forall X_n$

$[\neg subgoal_1(X_1,..,X_n) \vee ... \vee \neg subgoal_m(X_1,..,X_n) \vee goal(X_1,..,X_n)\ ]$

Regel als logische Formel

Hornklausel:

Alternative mit maximal

einem nicht-negierten Literal

$$\forall X_1 ... \forall X_n$$
$$[\neg subgoal_1(X_1,..,X_n) \vee ... \vee \neg subgoal_m(X_1,..,X_n) \vee goal(X_1,..,X_n)\ ]$$

Regel als logische Formel

Variable, die nur im Regelkörper auftreten, können als existentiell quantifizierte Variable **innerhalb des Regelkörpers** (!) betrachtet werden:

$$\forall X_1 ... \forall X_n \forall Y_1 ... \forall Y_k \; [\; subgoal_1(X_1,...,X_n,Y_1,...,Y_k) \wedge ... \wedge subgoal_m(X_1,...,X_n,Y_1,...,Y_k) \rightarrow goal(X_1,...,X_n) \;]$$

ist logisch äquivalent zu

$$\forall X_1 ... \forall X_n \; [\; \exists Y_1 .. \exists Y_k [subgoal_1(X_1,..,X_n,Y_1,..,Y_k) \wedge ... \wedge subgoal_m(X_1,..,X_n,Y_1,..,Y_k)] \rightarrow goal(X_1,...,X_n) \;]$$

```
grandfather(X,Z):-father(X,Y),father(Y,Z).
```

Regel als logische Formel

Intuitive Bedeutung:

„goal“ gilt (ist beweisbar)
falls alle „subgoals“ gelten (beweisbar sind).

entspricht anschaulich der Abtrennungsregel
(modus ponens)

$$\frac{H_1 \rightarrow H_2 \quad , H_1}{H_2}$$

$goal(X_1,...,X_n)$:- $subgoal_1(X_1,...,X_n)$,..., $subgoal_m(X_1,...,X_n)$.

Unbenannte/anonyme Variable

Unbenannte/anonyme Variable: _ für „beliebig“

```
mother_in_law(X,Y):-
                mother(X,Z),parent(Y,Z,_).
```

```
mother_in_law(X,Y):-
                mother(X,Z),parent(Z,Y,_).
```

Anfrage/Beweis

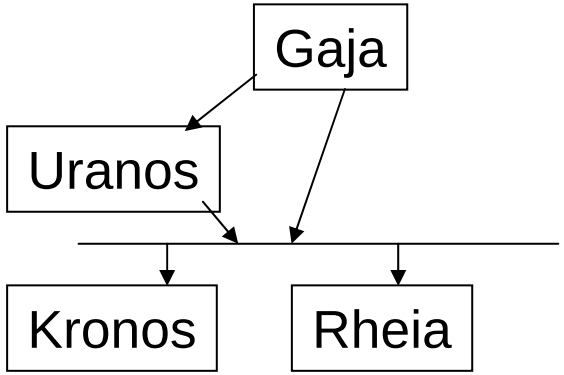
```
?- mother_in_law(gaea,gaea).
```

Anfrage/Beweis

```
?- mother_in_law(gaea,gaea).
```

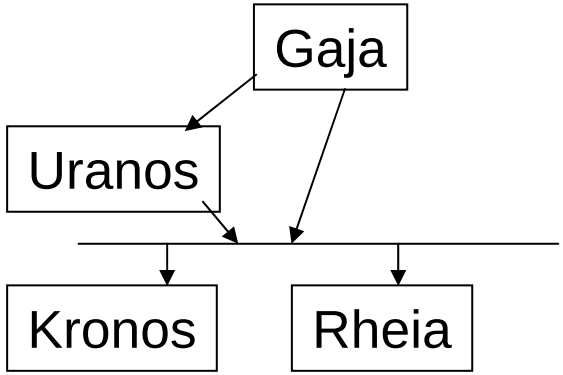
Zu beweisen:

```
mother_in_law(gaea,gaea).
```



Anfrage/Beweis

```
?- mother_in_law(gaea,gaea).
```



Zu beweisen:

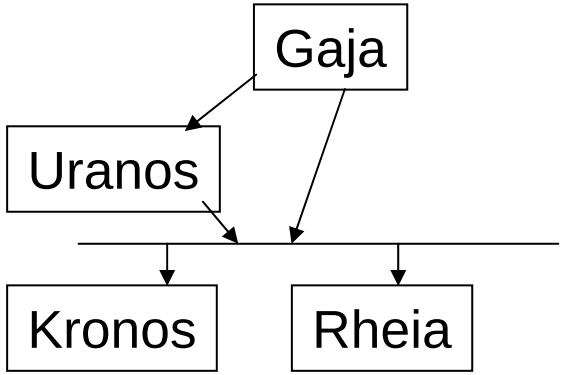
```
mother_in_law(gaea,gaea).
```

verfügbare Klausel:

```
mother_in_law(X,Y):-
                mother(X,Z),parent(Z,Y,_).
```

Anfrage/Beweis

```
?- mother_in_law(gaea,gaea).
```



Zu beweisen:

```
mother_in_law(gaea,gaea).
```

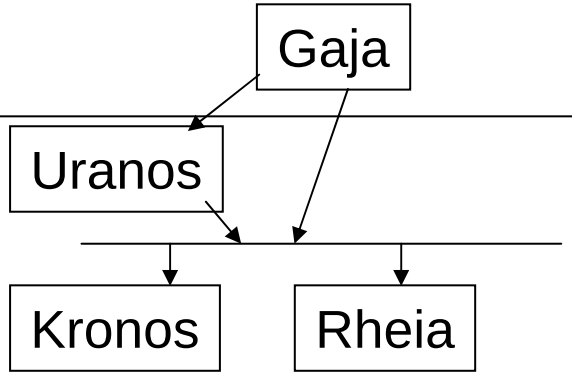
verfügbare Klausel:

```
mother_in_law(X,Y):-
                mother(X,Z),parent(Z,Y,_).
```

Klausel mit Bindung X=gaea, Y=gaea, $Z=Z_{[1]}$

```
mother_in_law(gaea,gaea):-
     mother(gaea,Z[1]),parent(Z[1],gaea,_).
```

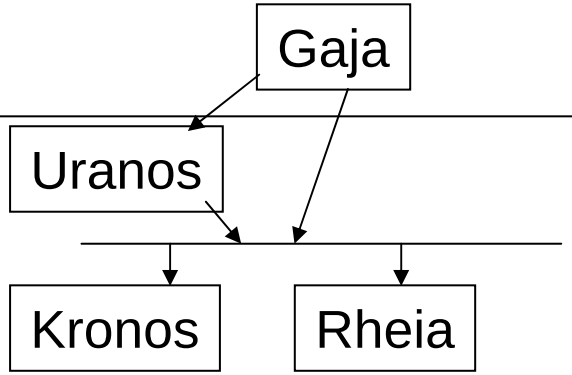
Anfrage/Beweis



zu beweisen:

```
mother_in_law(gaea,gaea):-
     mother(gaea,Z[1]),parent(Z[1],gaea,_).
```

Anfrage/Beweis



zu beweisen:

```
mother_in_law(gaea,gaea):-
     mother(gaea,Z[1]),parent(Z[1],gaea,_).
```

dafür zu beweisen

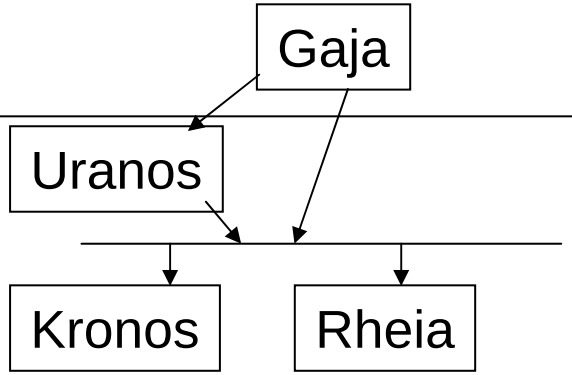
1)

```
mother(gaea,Z[1]).
```

2)

```
parent(Z[1],gaea,_).
```

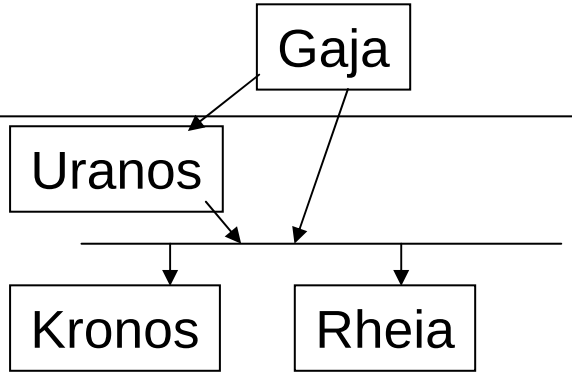
Anfrage/Beweis



zu beweisen:

1) `mother(gaea,Z[1]).`

Anfrage/Beweis



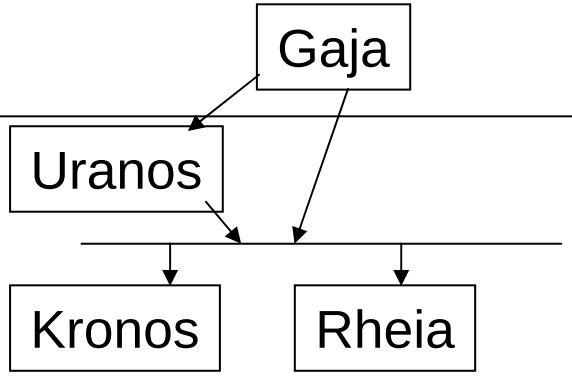
zu beweisen:

1) `mother(gaea,Z[1]).`

Verfügbare Klausel:

```
mother(X,Y):-parent(X,Y),female(X).
```

Anfrage/Beweis



zu beweisen:

1) `mother(gaea,Z[1]).`

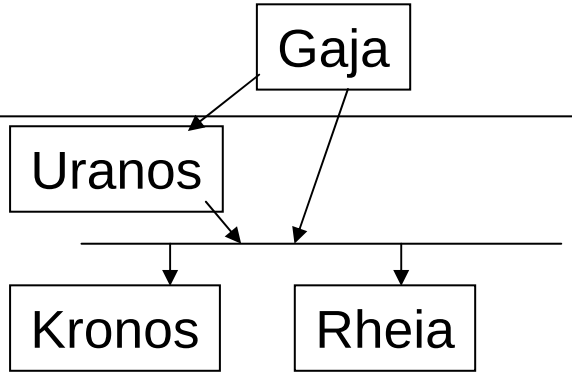
Verfügbare Klausel:

```
mother(X,Y):-parent(X,Y),female(X).
```

Klausel mit Bindung `X=gaea, Y=Z[1]` :

```
mother(gaea,Z[1]):-parent(gaea,Z[1]),female(gaea).
```

Anfrage/Beweis



zu beweisen:

1) `mother(gaea,Z[1]).`

Verfügbare Klausel:

```
mother(X,Y):-parent(X,Y),female(X).
```

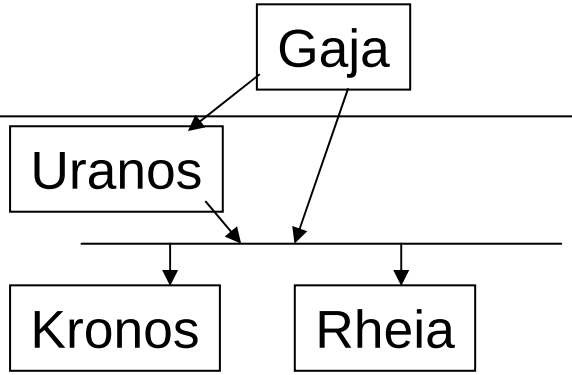
Klausel mit Bindung `X=gaea, Y=Z[1]` :

```
mother(gaea,Z[1]):-parent(gaea,Z[1]),female(gaea).
```

zu beweisen 1.1) `parent(gaea, Z[1] ).`

zu beweisen 1.2) `female(gaea).`

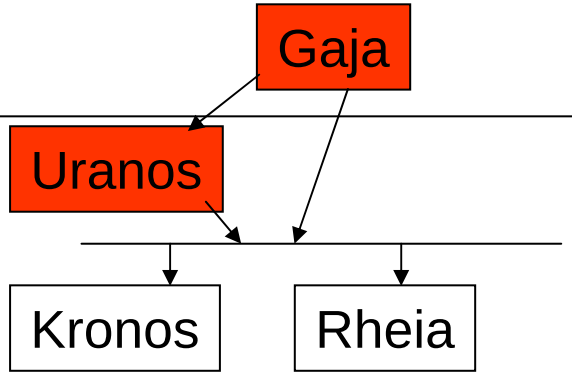
Anfrage/Beweis



zu beweisen:

1.1) `parent(gaea, Z[1] ).`

Anfrage/Beweis



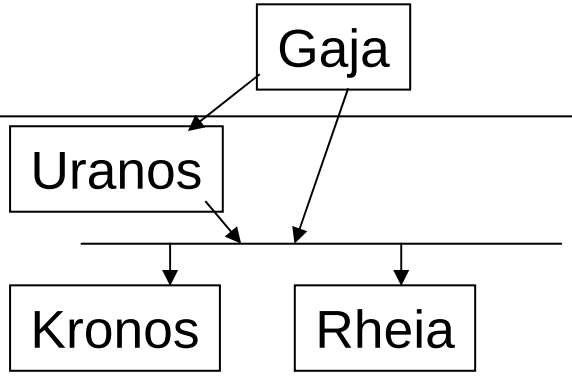
zu beweisen:

1.1) `parent(gaea, Z[1] ).`

verfügbarer Fakt ermöglicht Bindung $Z_{[1]}$ =`uranus` :

`parent(gaea,uranus).`

Anfrage/Beweis



zu beweisen:

1.1) `parent(gaea, Z[1] ).`

verfügbarer Fakt ermöglicht Bindung $Z_{[1]}$ =`uranus` :

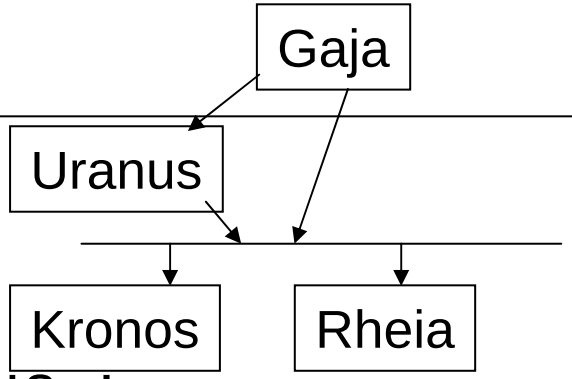
`parent(gaea,uranus).`

1.2) `female(gaea).`

verfügbarer Fakt:

`female(gaea).`

Anfrage/Beweis

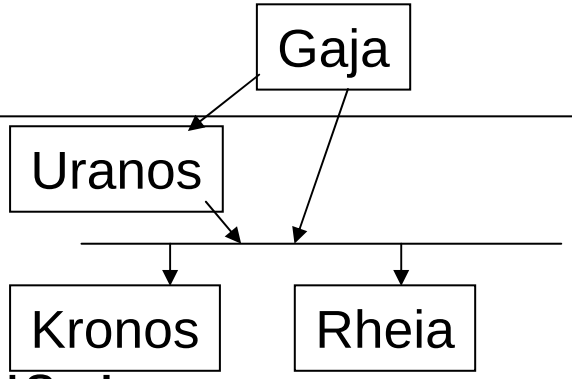


zu beweisen

unter Beachtung der Bindung $Z_{[1]}$=`uranus` :

2) `parent(uranus,gaea,_).`

Anfrage/Beweis



zu beweisen

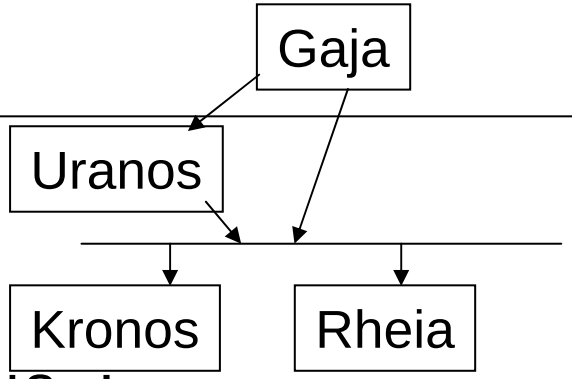
unter Beachtung der Bindung $Z_{[1]}$=`uranus` :

2) `parent(uranus,gaea,_).`

Verfügbare Klausel:

```
parent(X,Y,Z) :-father(X,Z),mother(Y,Z).
```

Anfrage/Beweis



zu beweisen

unter Beachtung der Bindung $Z_{[1]}$=uranus :

2) `parent(uranus,gaea,_).`

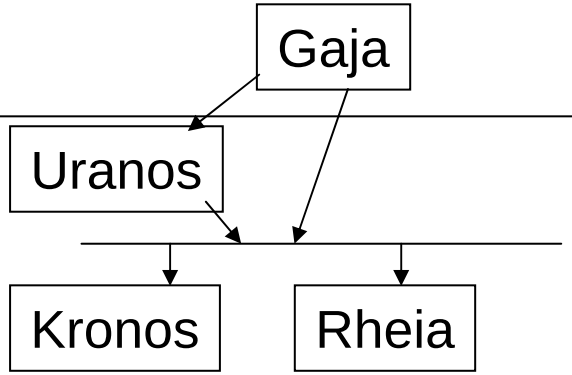
Verfügbare Klausel:

```
parent(X,Y,Z) :-father(X,Z),mother(Y,Z).
```

Klausel mit Bindungen :

```
parent(uranus,gaea,Z[2]) :-
      father(uranus,Z[2]),mother(gaea,Z[2]).
```

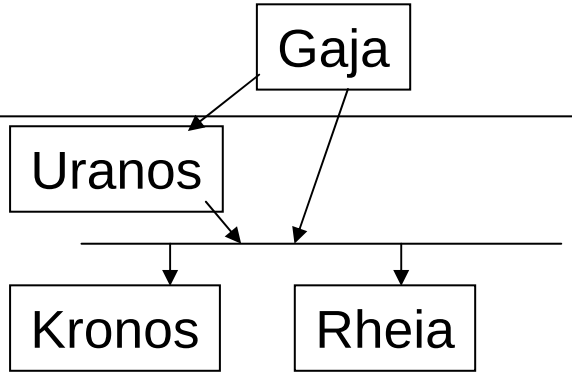
Anfrage/Beweis



zu beweisen

```
parent(uranus,gaea,Z[2]) :-
      father(uranus,Z[2]),mother(gaea,Z[2]).
```

Anfrage/Beweis



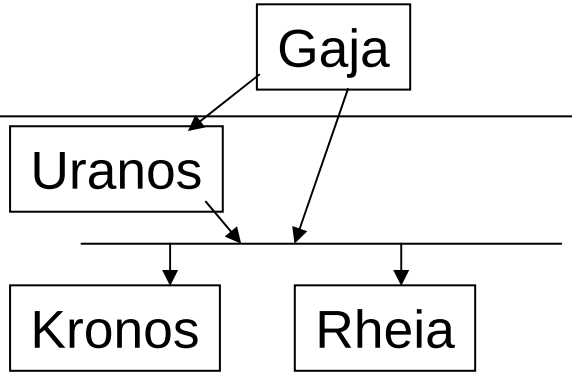
zu beweisen

```
parent(uranus,gaea,Z[2]) :-
      father(uranus,Z[2]),mother(gaea,Z[2]).
```

zu beweisen 2.1) `father(uranus,Z[2]).`

zu beweisen 2.2) `mother(gaea,Z[2]).`

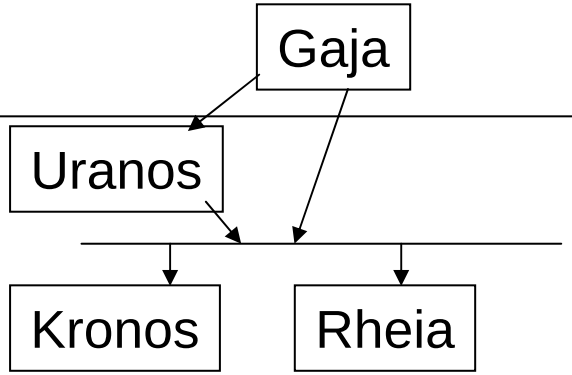
Anfrage/Beweis



zu beweisen:

2.1) `father(uranus,Z[2]).`

Anfrage/Beweis



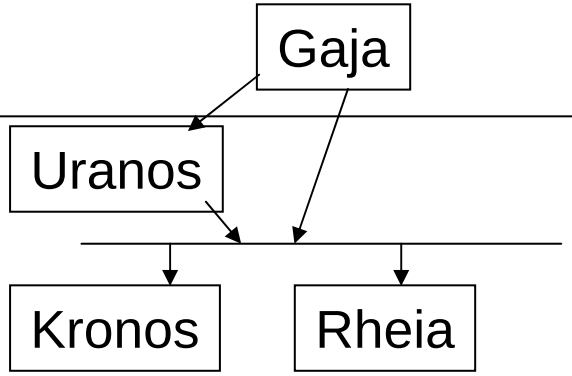
zu beweisen:

2.1) `father(uranus,Z[2]).`

Verfügbare Klausel:

```
father(X,Y):-parent(X,Y),male(X).
```

Anfrage/Beweis



zu beweisen:

2.1) `father(uranus,Z[2]).`

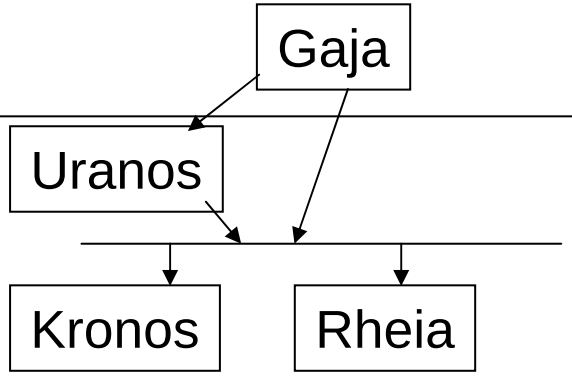
Verfügbare Klausel:

```
father(X,Y):-parent(X,Y),male(X).
```

Klausel mit Bindung `X=uranus, Y=Z[2]` :

```
father(uranus,Z[2]):-
                parent(uranus,Z[2]),male(uranus).
```

Anfrage/Beweis



zu beweisen:

2.1) `father(uranus,Z[2]).`

Verfügbare Klausel:

```
father(X,Y):-parent(X,Y),male(X).
```

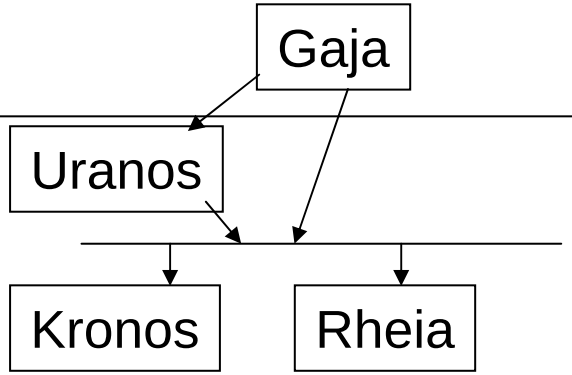
Klausel mit Bindung `X=uranus, Y=Z[2]` :

```
father(uranus,Z[2]):-
                parent(uranus,Z[2]),male(uranus).
```

zu beweisen 2.1.1) `parent(uranus, Z[2]).`

zu beweisen 2.1.2) `male(uranus).`

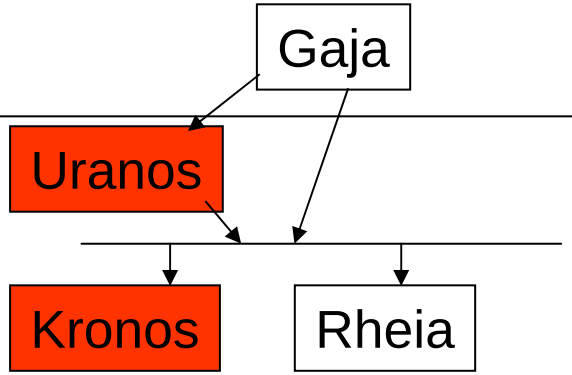
Anfrage/Beweis



zu beweisen:

2.1.1) `parent(uranus,Z[2]).`

Anfrage/Beweis



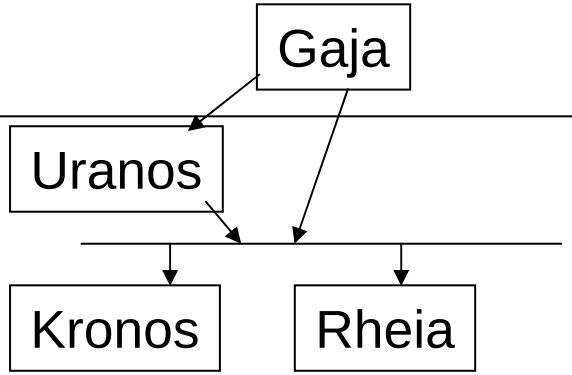
zu beweisen:

2.1.1) `parent(uranus,Z[2]).`

verfügbarer Fakt (z.B.), bewirkt Bindung $Z_{[2]}$=`cronus` :

```
parent(uranus,cronus).
```

Anfrage/Beweis



zu beweisen:

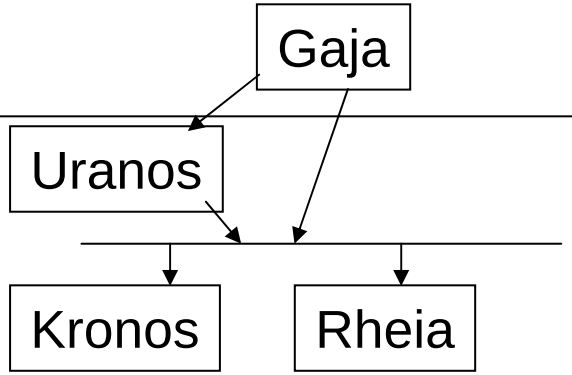
2.1.1) `parent(uranus,Z[2]).`

verfügbarer Fakt (z.B.), bewirkt Bindung $Z_{[2]}$=`cronus` :

```
parent(uranus,cronus).
```

2.1.2) `male(uranus).`

Anfrage/Beweis



zu beweisen:

2.1.1) `parent(uranus,Z[2]).`

verfügbarer Fakt (z.B.), bewirkt Bindung $Z_{[2]}$=`cronus` :

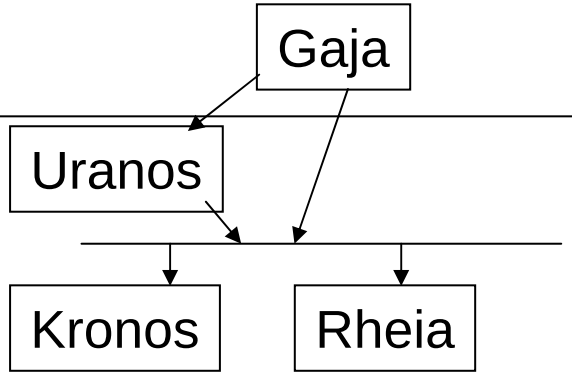
```
parent(uranus,cronus).
```

2.1.2) `male(uranus).`

verfügbarer Fakt:

```
male(uranus).
```

Anfrage/Beweis



zu beweisen

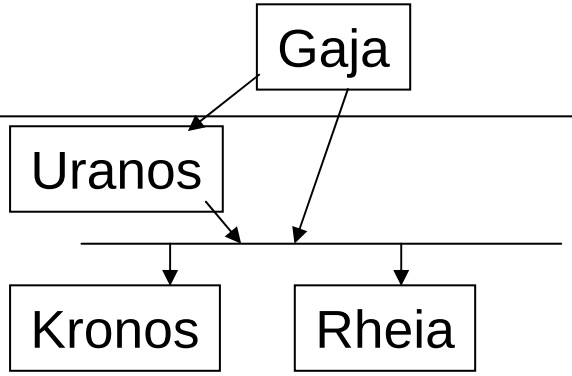
mit erfolgter Bindung $Z_{[2]}$=`cronus`

2.2) `mother(gaea,cronus).`

Verfügbare Klausel:

```
mother(X,Y):-parent(X,Y),female(X).
```

Anfrage/Beweis



zu beweisen

mit erfolgter Bindung $Z_{[2]}$=`cronus`

2.2) `mother(gaea,cronus).`

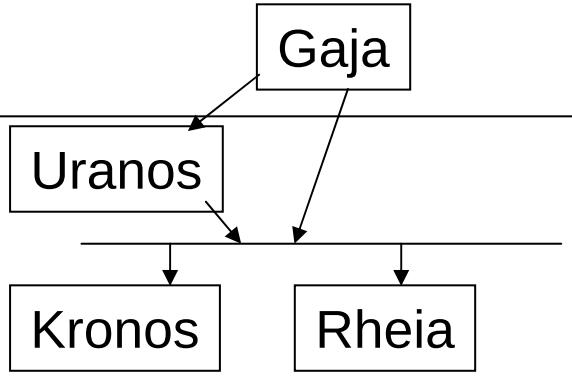
Verfügbare Klausel:

```
mother(X,Y):-parent(X,Y),female(X).
```

Klausel mit Bindung `X=gaea, Y=cronus` :

```
mother(gaea,cronus):-
                parent(gaea,cronus),female(gaea).
```

Anfrage/Beweis



zu beweisen

mit erfolgter Bindung $Z_{[2]}$=`cronus`

2.2) `mother(gaea,cronus).`

Verfügbare Klausel:

```
mother(X,Y):-parent(X,Y),female(X).
```

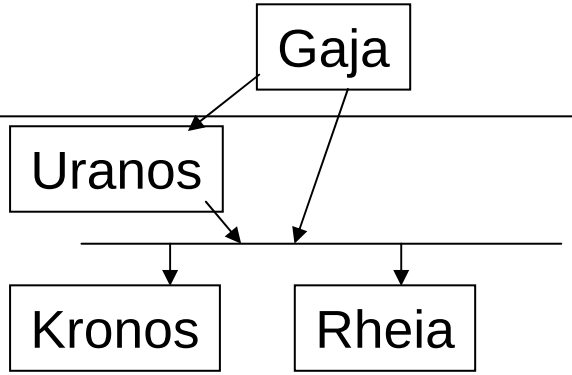
Klausel mit Bindung `X=gaea, Y=cronus` :

```
mother(gaea,cronus):-
                parent(gaea,cronus),female(gaea).
```

zu beweisen 2.2.1) `parent(gaea,cronus),`

zu beweisen 2.2.2) `female(gaea).`

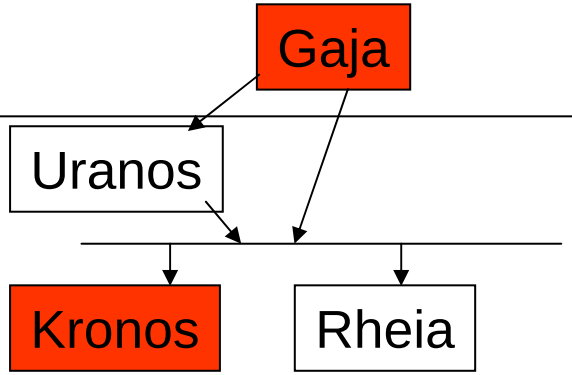
Anfrage/Beweis



zu beweisen:

2.2.1) `parent(gaea,cronus).`

Anfrage/Beweis



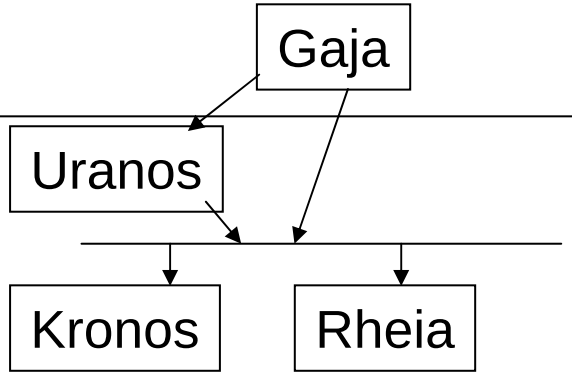
zu beweisen:

2.2.1) `parent(gaea,cronus).`

verfügbarer Fakt:

`parent(gaea,cronus).`

Anfrage/Beweis



zu beweisen:

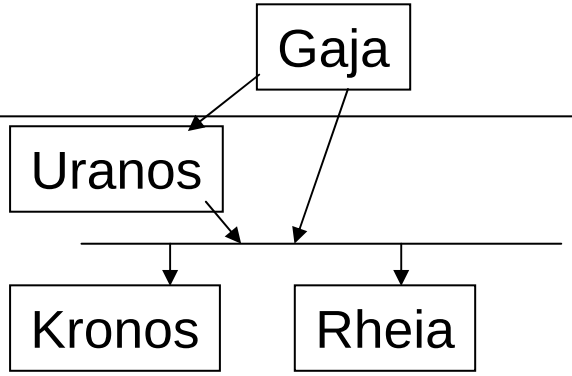
2.2.1) `parent(gaea,cronus).`

verfügbarer Fakt:

`parent(gaea,cronus).`

2.2.2) `female(gaea).`

Anfrage/Beweis



zu beweisen:

2.2.1) `parent(gaea,cronus).`

verfügbarer Fakt:

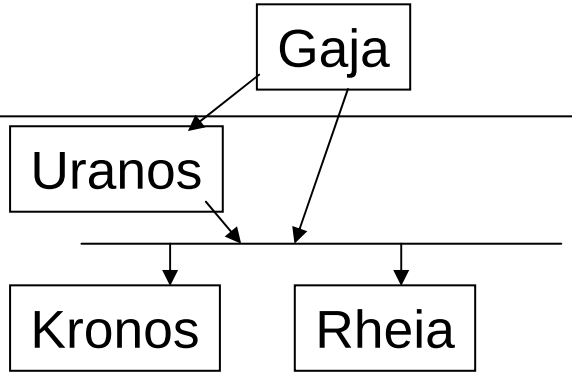
`parent(gaea,cronus).`

2.2.2) `female(gaea).`

verfügbarer Fakt:

`female(gaea).`

Anfrage/Beweis



zu beweisen:

2.2.1) `parent(gaea,cronus).`

verfügbarer Fakt:

`parent(gaea,cronus).`

2.2.2) `female(gaea).`

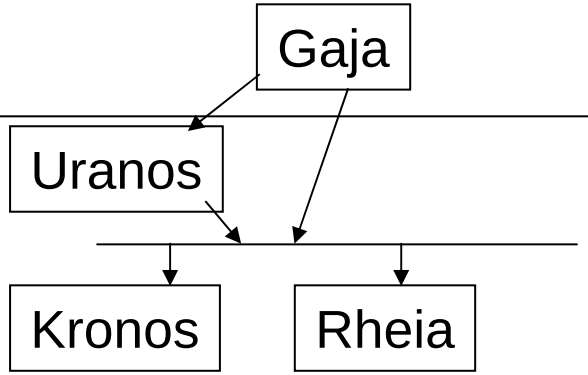
verfügbarer Fakt:

`female(gaea).`

Fertig

Anfrage/Beweis-**Baum**

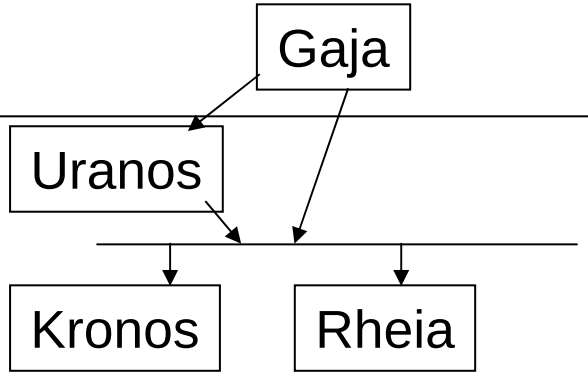
```
?- mother_in_law(gaea,gaea).
Yes.
```

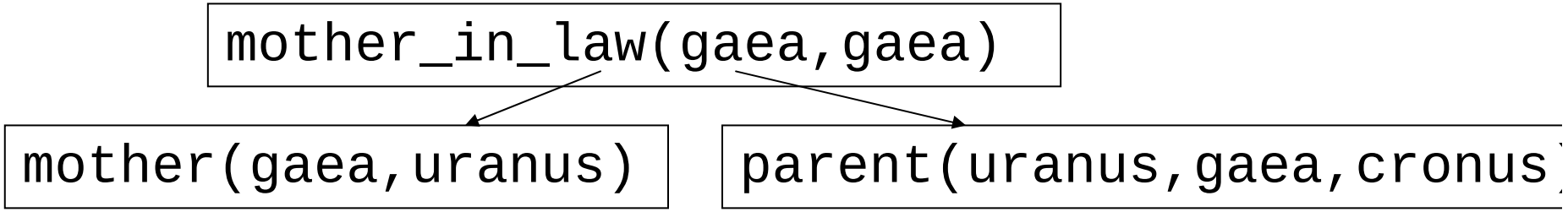



```
mother_in_law(gaea,gaea)
```

Anfrage/Beweis-**Baum**

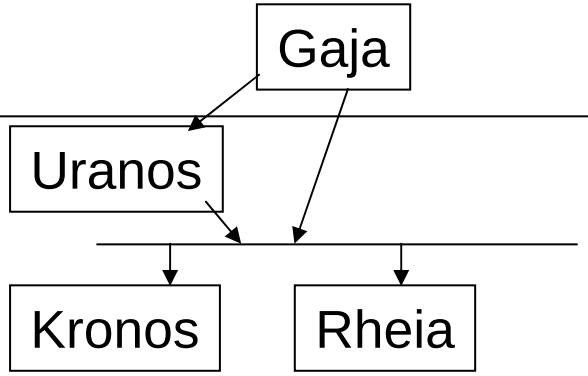
```
?- mother_in_law(gaea,gaea).
Yes.
```

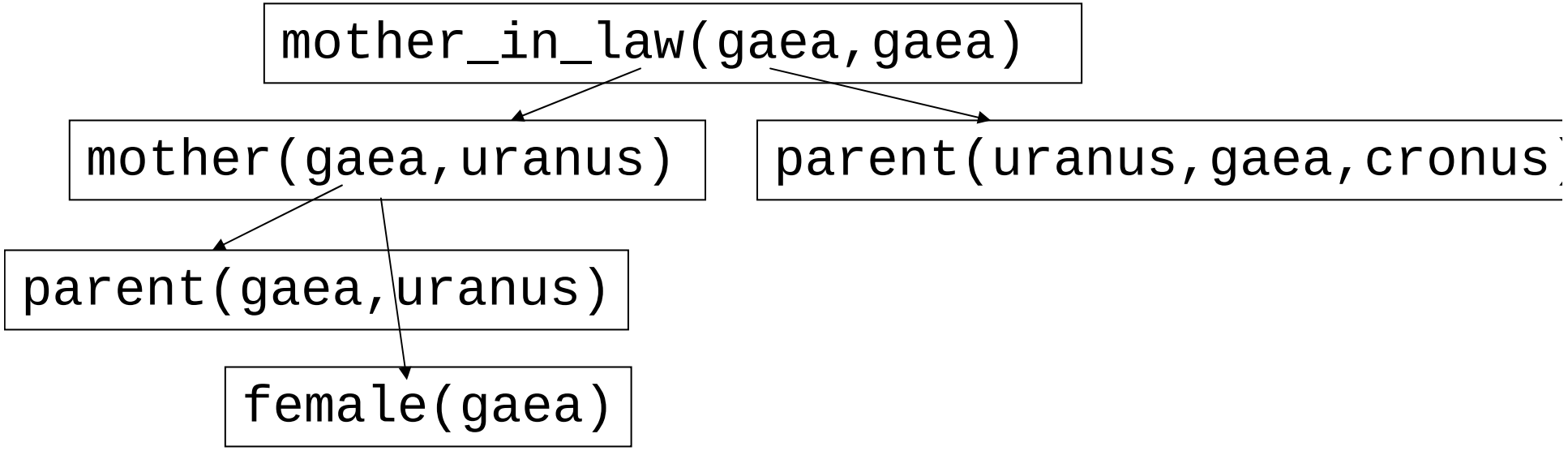




Anfrage/Beweis-**Baum**

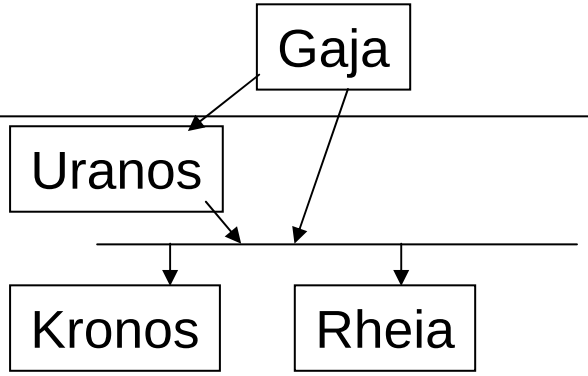
```
?- mother_in_law(gaea,gaea).
Yes.
```

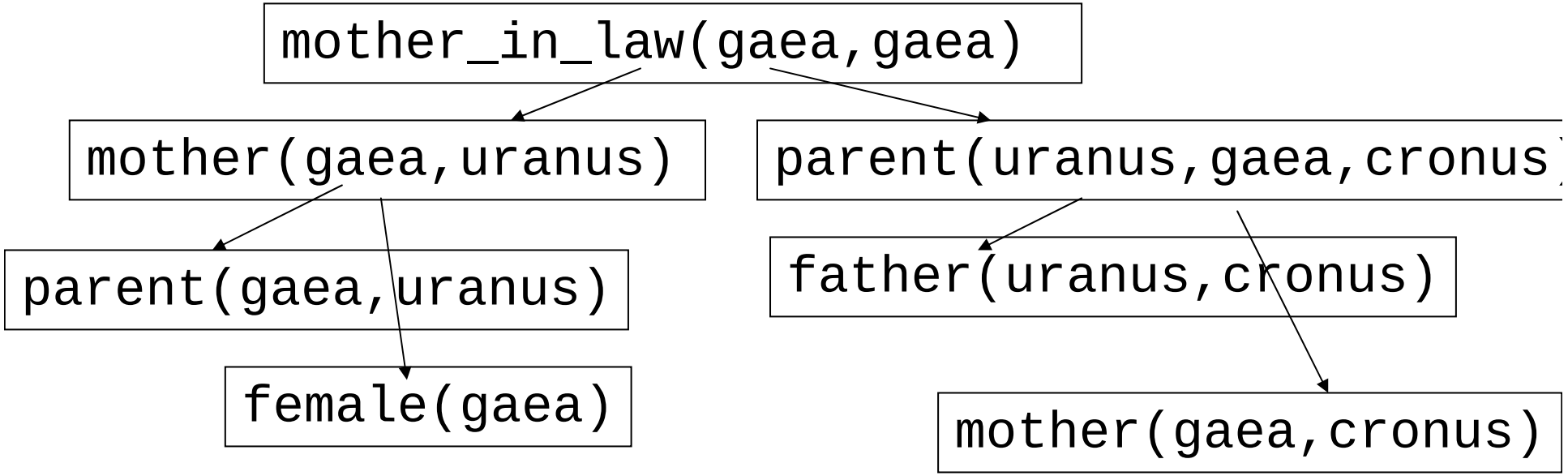




Anfrage/Beweis-**Baum**

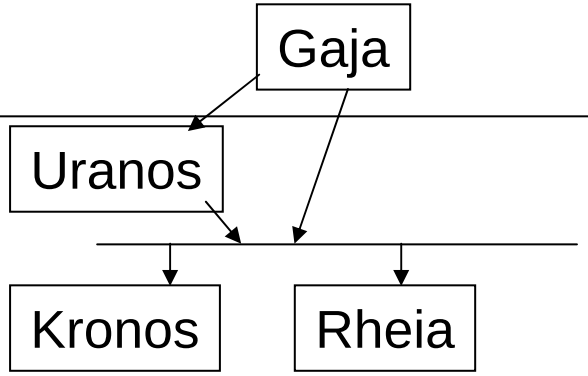
```
?- mother_in_law(gaea,gaea).
Yes.
```

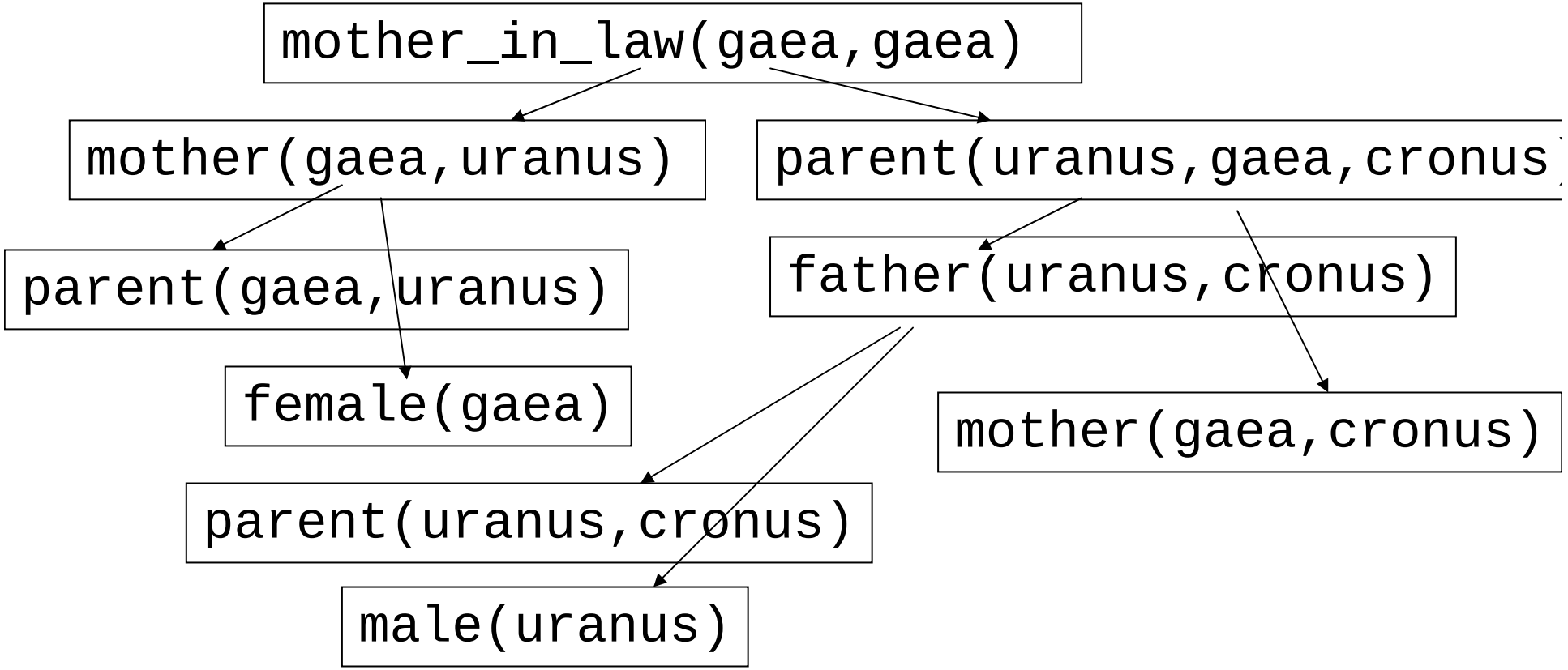




Anfrage/Beweis-**Baum**

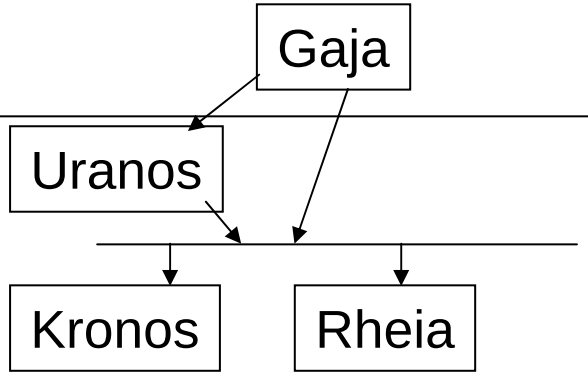
```
?- mother_in_law(gaea,gaea).
Yes.
```

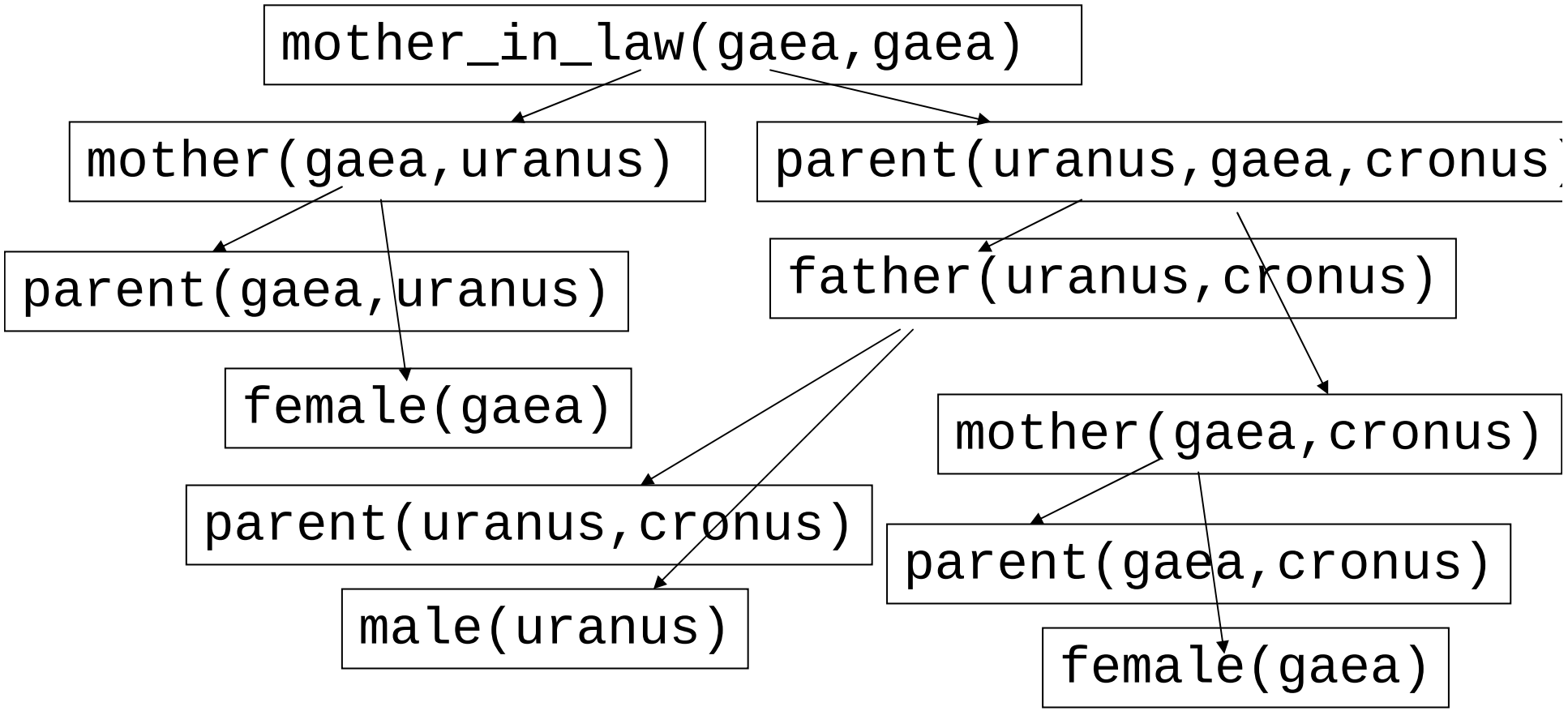




Anfrage/Beweis-**Baum**

```
?- mother_in_law(gaea,gaea).
Yes.
```





(existenzielle) Anfragen mit Regeln

```
?- father(zeus,athena).
yes.
?- father(zeus,X).
X=hermes?
;
X=athena?
;
X=ares?
.
yes.
```

```
rent(uranus, cronus).
rent(gaea,   cronus).
rent(gaea,   rhea).
rent(rhea,   zeus).
rent(cronus, zeus).
rent(rhea,   hera).
rent(cronus, hera).
rent(cronus, hades).
rent(rhea,   hades).
rent(cronus, hestia
rent(rhea,   hestia
rent(zeus,   hermes
rent(maia,   hermes
...
```

```
female(gaea).
female(rhea).
female(hera).
female(hestia).
female(demeter).
female(athena).
female(metis).
female(maia).
female(persephone).
female(aphrodite).
```

```
male(uranus).
male(cronus).
male(zeus).
male(hades).
male(hermes).
```

```
father(Vater,Kind)
      :-parent(Vater,Kind), male(Vater).
```

Prolog-Programm

Programme bestehen aus Klauseln.

```
father(X,Y):-parent(X,Y),male(X).
mother(X,Y):-parent(X,Y),female(X).
```

```
parent(X,Y,Z) :-father(X,Z),mother(Y,Z).
```

```
son(X,Y):-parent(X,Y),male(Y).
```

```
grandfather(X,Z):-father(X,Y),parent(Y,Z).
grandmother(X,Z):-mother(X,Y),parent(Y,Z).
```

```
grandchild(X,Y):-grandfather(Y,X).
grandchild(X,Y):-grandmother(Y,Z).
```

```
parent(uranus, cronus).
parent(gaea,   cronus).
parent(gaea,   rhea).
parent(rhea,   zeus).
parent(cronus, zeus).
parent(rhea,   hera).
parent(cronus, hera).
parent(cronus, hades).
parent(rhea,   hades).
parent(cronus, hestia
parent(rhea,   hestia
parent(zeus,   hermes
parent(maia,   hermes
...
```

```
female(gaea).
female(rhea).
female(hera).
female(hestia).
female(demeter).
female(athena).
female(metis).
female(maia).
female(persephone).
female(aphrodite).
```

```
male(uranus).
male(cronus).
male(zeus).
male(hades).
male(hermes).
male(apollo).
male(dionysius).
male(hephaestus).
male(poseidon).
```

Klauseln sind Fakten oder Regeln.

Fakt als Regel: `fakt:-true`

mit Prädikat true/0

Prolog-Programm

Klauseln definieren Relationen (Prädikate)

```
father(Vater,Kind)
        :-parent(Vater,Kind), male(Vater).
```

Intuitive Bedeutung:

Linke Seite gilt, wenn alle Prädikate der rechten Seite bei entsprechender Belegung/Bindung der Variablen gelten.

Es gilt:

```
father(zeus,athena).
```

weil gilt:

```
parent(zeus,athena).
```

```
male(zeus).
```

Prolog-Programm: Prozeduren

Klauseln mit gleichem Kopf-Funktor (Name,Stelligkeit) bilden eine **Prozedur**

Die Klauseln einer Prozedur bieten Alternativen für die Prolog-Beweise

```
grandfather(X,Z):-father(X,Y),father(Y,Z).
grandfather(X,Z):-father(X,Y),mother(Y,Z).
```

```
parent(uranus, cronus).
parent(gaea,   cronus).
parent(gaea,   rhea).
parent(rhea,   zeus).
...
```

Prolog-Programm: Prozeduren

Klauseln mit gleichem Kopf-Funktor (Name,Stelligkeit) bilden eine **Prozedur**

Redundanzen sind **logisch** unproblemastisch

```
grandfather(X,Z):-father(X,Y),father(Y,Z).
grandfather(X,Z):-father(X,Y),mother(Y,Z).
grandfather(X,Z):-father(X,Y),parent(Y,Z).
grandfather(cronus, ares).
grandfather(cronus, athena).
```

Redundanzen bieten zusätzliche Beweisvarianten

PROLOG: evtl. unerwünschte Auswirkungen

Softwaretechnologie: Minimalitätsprinzip

Prolog-Programm

Anfragen an Programme haben die Form

$$? - goal(X_1,...,X_n).$$

im Sinne von „gilt ...?“ („ist ... beweisbar?“):

$$\exists X_1..\exists X_k\ [\ goal(X_1,...,X_n)\ ]$$

Oder allgemeiner

$$? - goal_1(X_1,...,X_n)\ ,...,\ goal_m(X_1,...,X_n).$$

im Sinne von „gilt ...?“ („ist ... beweisbar?“):

$$\exists X_1..\exists X_k\ [goal_1(X_1,...,X_n) \wedge ... \wedge goal_m(X_1,...,X_n)\ ]$$

Prolog-Interpreter

Laufzeit-System für Prolog,

das Antworten auf Anfragen an ein Programm gibt,

d.h. Beweise sucht und ausführt (Theorem-Beweiser).

Vorstellung:

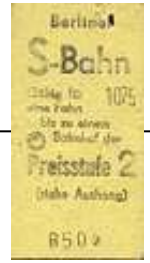
Man gibt Fakten und Zusammenhänge als Programm ein

(z.B. Fahrplantabelle)

und erhält Antworten bzgl. aller Folgerungen

(z.B. billigste Verbindungen von A nach B) .

Prolog-Interpreter



Fahrplan-Fakten

```
s_bahn(alexanderplatz,jannowitzbrücke,6:09,6:11, 103).
s_bahn(jannowitzbrücke,ostbahnhof,    6:11,6:13, 103).
...
```

Suchprogramm

```
erreichbar(Start,Ziel,Zeit)
:- s_bahn(Start,Zwischenziel,Abfahrt,Ankunft,_),
   erreichbar(Zwischenziel,Ziel,Zeit1),
   berechneZeit(Zeit1,Ankunft,Abfahrt,Zeit).
...
```

Fahrplanauskunft

Anfrage:

Nächste Verbindung von Stadtmitte nach Adlershof

Fahrplanauskunft

Problem:
Was genau möchte der Kunde?

Nächste Verbindung?

Kürzeste Verbindung?

Billigste Verbindung?

Wenig Umsteigen?

Prolog und Logik

Prolog-Programm P

= Axiome

Anfrage Q

= Frage nach Beweisbarkeit von Q aus P

Antwort

= Ergebnis eines Beweises bzw. Fehlschlag

Trace

= Verlauf des Beweisversuchs

Prolog-Interpreter

Ziel:

Prolog-Interpreter als universelles Verfahren im PK1

Insbesondere

Vollständigkeit:

Interpreter liefert alle Folgerungen aus dem Programm

Korrektheit:

Interpreter liefert nur Folgerungen aus dem Programm

Situation im PK1

Q folgt aus Formelmenge $\{P_1,..,P_n\}$

gdw. Q ist aus Formelmenge $\{P_1,..,P_n\}$ syntaktisch ableitbar

gdw. $P_1 \wedge ... \wedge P_n \rightarrow Q$ ist allgemeingültig

Allgemeingültigkeit ist <u>axiomatisierbar/aufzählbar</u>:

Falls ein Ausdruck H allgemeingültig ist,
so ist das in endlich vielen Schritten feststellbar.

Genauer: Es gibt dafür ein universelles Verfahren.

Algorithmus/Programm

„Theorembeweiser“

Situation im PK1

Q folgt aus Formelmenge $\{P_1,..,P_n\}$

gdw. Q ist aus Formelmenge $\{P_1,..,P_n\}$ syntaktisch ableitbar

gdw. $P_1 \wedge ... \wedge P_n \rightarrow Q$ ist allgemeingültig

Allgemeingültigkeit ist nicht entscheidbar:

Es gibt **kein universelles** Verfahren, das für beliebige H **entscheidet**, ob H allgemeingültig ist.

Falls ein Ausdruck H **nicht allgemeingültig** ist,
so ist das eventuell nicht feststellbar
(Verfahren kommt evtl. nicht zum Abbruch).

Genauer: Es gibt dafür **kein** universelles Verfahren.

Situation im PK1

Q folgt aus Formelmenge $\{P_1,..,P_n\}$

gdw. Q ist aus Formelmenge $\{P_1,..,P_n\}$ syntaktisch ableitbar

gdw. $P_1 \wedge ... \wedge P_n \rightarrow Q$ ist allgemeingültig

Falls ein Ausdruck H **allgemeingültig** ist,
so ist das in endlich vielen Schritten feststellbar.

Genauer: Es gibt dafür ein **universelles Verfahren**.

Falls ein Ausdruck H **nicht allgemeingültig** ist,
so ist das nicht allgemein feststellbar.

Genauer: Es gibt dafür **kein universelles Verfahren**.

Konsequenzen aus Situation im PK1

Kein Entscheidungsprogramm im PK1 möglich.

Spezielle Situation für Prolog-Interpreter:

- eingeschränkt durch Horn-Klauseln
- eingeschränkt durch Beweisstrategie im Interpreter
  (Frage nach Vollständigkeit/Korrektheit!)

Entscheidungsverfahren existieren prinzipiell für

- aussagenlogische Programme oder
- Programme über endlichen Relationen

Nichtdeterministische Suche nach Beweisbaum

Ausgangspunkt und Zwischenzustände:

Menge von „offenen“ (zu beweisenden) Teilzielen:

$subgoals = \{ subgoal_1(...) , ..., subgoal_m(...) \}$

Die Teilziele $subgoal_1(...)$ haben die Form $funktor(t_1,...,t_n)$.

Dabei bezeichnet funktor ein n-stelliges Prädikat,

dessen Argumente jeweils an Terme t_i gebunden sind.

Terme (Strukturen) sind Variablen, Konstante oder komplexere Strukturen, die wiederum Terme enthalten können.

Nichtdeterministische Suche nach Beweisbaum

Ziel (Ende des Verfahrens):

subgoals ={} , d.h. alle Teilziele sind bewiesen

dabei Antwort „yes“ bzw.

Angabe der Terme,

an die die Variablen der Anfrage gebunden wurden.

oder:

kein weiterer Beweisversuch möglich, dabei Antwort „no“

Nichtdeterministische Suche nach Beweisbaum

Zwischenschritte:

Wähle ein zu beweisendes Teilziel:

$funktor(t_1,...,t_n) \in subgoals$

Wähle eine passende Klausel der zugehörigen Prozedur:

$funktor(x_1,..,x_n)$:-$funktor^1(x^1_1,..,x^1_{n1}),...,funktor^m(x^m_1,..,x^m_{nm})$

Unifikation

des Kopfes $funktor(x_1,...,x_n)$ mit Teilziel $funktor(t_1,...,t_n)$
ergibt eine Variablensubstitution σ
(Ersetzung von Variablen durch Terme)

Neuer Zwischenzustand:

$subgoals := \sigma(\quad (subgoals - \{ funktor(t_1,...,t_n) \} \;)$
$\cup \{ funktor^1(t^1_1,..,t^1_{n1}),...,funktor^m(t^m_1,..,t^m_{nm}) \})$

Nichtdeterministische Suche nach Beweisbaum

Die Suche ist erfolgreich, wenn

- in jedem Zwischenschritt eine passende Klausel gewählt wird, bei der die Unifikation gelingt,
- am Ende subgoals={} gilt.

In jedem Schritt i erfolgen Substitutionen σ_i von (allen) Variablen.

Die Substitutionen ergeben in ihrer Gesamtheit die Terme, an die die Variablen X der Anfrage gebunden wurden:

„Antwort-Substitution“:

$$\sigma(X) = \sigma_k\ (\sigma_{k-1}\ (\ldots\ \sigma_1\ (X)\ \ldots)\ )$$

Interpreter für Standard-Prolog

Idee:

Systematische Suche nach Beweismöglichkeiten

(Reihenfolge für „wähle Teilziel/Klausel“)

Reihenfolge innerhalb einer Prozedur

(Alternativen für Beweis)

oben vor unten

Reihenfolge innerhalb einer Klausel

(alle subgoals müssen erfüllt werden)

links vor rechts

Interpreter für Standard-Prolog

Backtracking:

Alternativen für den Beweis eines Teilziel werden markiert („Backtrack-Punkte“).

Beim Fehlschlagen eines Beweisversuchs wird am jüngsten Backtrack-Punkt ein alternativer Beweis gestartet („chronologisches Backtracking“).
Dabei werden zwischenzeitliche Variablenbindungen zurückgenommen.

Eingabe von „;“ bei Antworten auf existentielle Anfragen wirkt wie Fehlschlag (löst Backtracking aus) .

```
grandfather(X,Z):-father(X,Y),father(Y,Z).
grandfather(X,Z):-father(X,Y),mother(Y,Z).
```

```
?-grandfather(X,ares).
```

Beweisversuch mit 1. Klausel (ggf. neue Variablennamen!)

```
grandfather(X1,Z):-father(X1,Y),father(Y,Z).
```

Backtrackpunkt für 2. Klausel:

```
grandfather(X,Z):-father(X,Y),mother(Y,Z).
```

Substitution σ (X1) = X σ (Z) = ares

```
grandfather(X,ares):-
                father(X,Y),father(Y,ares).
```

zu beweisen:

```
father(X,Y).
```

```
father(Y,ares).
```

```
father(X,Y):-parent(X,Y),male(X).
```

zu beweisen:

```
father(X,Y).
```

Beweisversuch mit Klausel (neue Variablennamen):

```
father(X2,Y1):-parent(X2,Y1),male(X2).
```

(Keine Alternativen – kein Backtrackpunkt)

Substitution σ (X2) = X σ (Y1) = Y

```
father(X,Y):-parent(X,Y),male(X).
```

zu beweisen:

```
parent(X,Y).
```

```
male(X).
```

```
father(Y,ares).
```

```
parent(uranus, cronus).
parent(gaea,   cronus).
parent(gaea,   rhea).
parent(rhea,   zeus).
parent(cronus, zeus).
parent(rhea,   hera).
parent(cronus, hera).
parent(cronus, hades).
...
```

zu beweisen:

```
parent(X,Y).
```

Beweisversuch mit 1. Fakt

```
parent(uranus, cronus).
```

Backtrackpunkt für Alternativen

```
parent(gaea, cronus).
```

Substitution σ (X) = uranus σ (Y) = cronus

```
parent(uranus,cronus).
```

Ist Fakt, d.h. keine neuen Teilziele.

Zu beweisen:

```
male(uranus).
```

```
father(uranus,ares).
```

```
male(uranus).
male(cronus).
male(zeus).
male(hades).
male(hermes).
male(apollo).
male(dionysius).
male(hephaestus).
male(poseidon).
```

zu beweisen:

```
male(uranus).
```

Beweis mit Fakt

```
male(uranus).
```

gelingt:

```
male(uranus).
```

Keine neuen Substitutionen.
Keine neuen Teilziele.

Zu beweisen:

```
father(uranus,ares).
```

```
father(X,Y):-parent(X,Y),male(X).
```

zu beweisen:

```
father(uranus,ares).
```

Beweisversuch mit Klausel (neue Variablennamen)

```
father(X3,Y2):-parent(X3,Y2),male(X3).
```

(Keine Alternativen – kein Backtrackpunkt)

Substitution σ (X3) = uranus σ (Y1) = ares

```
father(uranus,ares):-
        parent(uranus,ares),male(uranus).
```

zu beweisen:

```
parent(uranus,ares).
```

```
male(uranus).
```

```
parent(uranus, cronus).
parent(gaea,   cronus).
parent(gaea,   rhea).
parent(rhea,   zeus).
parent(cronus, zeus).
parent(rhea,   hera).
parent(cronus, hera).
parent(cronus, hades).
...
```

zu beweisen:

```
parent(uranus,ares).
```

Es gibt keinen solchen Fakt

Beweisversuch fehlgeschlagen.

Rückkehr zum jüngsten Backtrack-Punkt

```
parent(gaea,   cronus).
```

beim Beweis für

```
parent(X,Y).
```

```
male(X).
```

```
father(Y,ares).
```

Weitere Fehlschläge folgen
bei Beweisversuchen mit

```
parent(uranus, cronus).
parent(gaea,   cronus).
parent(gaea,   rhea).
parent(rhea,   zeus).
parent(cronus, zeus).
parent(rhea,   hera).
parent(cronus, hera).
parent(cronus, hades).
...
```

```
parent(uranus, cronus).
parent(gaea,   cronus).
parent(gaea,   rhea).
parent(rhea,   zeus).
parent(cronus, zeus).
parent(rhea,   hera).
parent(cronus, hera).
parent(cronus, hades).
...
```

zu beweisen:

```
parent(X,Y).
```

Beweisversuch mit Fakt

```
parent(cronus, zeus).
```

Backtrackpunkt für Alternativen.

```
parent(rhea, hera).
```

Substitution σ (X) = cronus σ (Y) = zeus

```
parent(cronus,zeus).
```

Ist Fakt, d.h. keine neuen Teilziele.

Zu beweisen:

```
male(cronus).
```

```
father(zeus,ares).
```

```
male(uranus).
male(cronus).
male(zeus).
male(hades).
male(hermes).
male(apollo).
male(dionysius).
male(hephaestus).
male(poseidon).
```

zu beweisen:

```
male(cronus).
```

Beweis mit Fakt

```
male(cronus).
```

gelingt:

```
male(cronus).
```

Keine neuen Substitutionen.

Keine neuen Teilziele.

Zu beweisen:

```
father(zeus,ares).
```

```
father(X,Y):-parent(X,Y),male(X).
```

zu beweisen:

```
father(zeus,ares).
```

Beweisversuch mit Klausel (neue Variablennamen)

```
father(X3,Y2):-parent(X3,Y2),male(X3).
```

(Keine Alternativen – kein Backtrackpunkt)

Substitution σ (X3) = zeus σ (Y1) = ares

```
father(zeus,ares):-
        parent(zeus,ares),male(zeus).
```

zu beweisen:

```
parent(zeus,ares).
```

```
male(zeus).
```

```
parent(uranus, cronus).
parent(gaea,   cronus).
parent(gaea,   rhea).
parent(rhea,   zeus).
parent(cronus, zeus).
parent(rhea,   hera).
parent(cronus, hera).
parent(cronus, hades).
...
```

zu beweisen:

```
parent(zeus,ares).
```

Beweis mit Fakt

```
parent(zeus,ares).
```

gelingt:

```
parent(zeus,ares).
```

Keine neuen Substitutionen.

Keine neuen Teilziele.

Zu beweisen:

```
male(zeus).
```

```
male(uranus).
male(cronus).
male(zeus).
male(hades).
male(hermes).
male(apollo).
male(dionysius).
male(hephaestus).
male(poseidon).
```

zu beweisen:

```
male(zeus).
```

Beweis mit Fakt

```
male(zeus).
```

gelingt:

```
male(zeus).
```

Keine neuen Substitutionen.

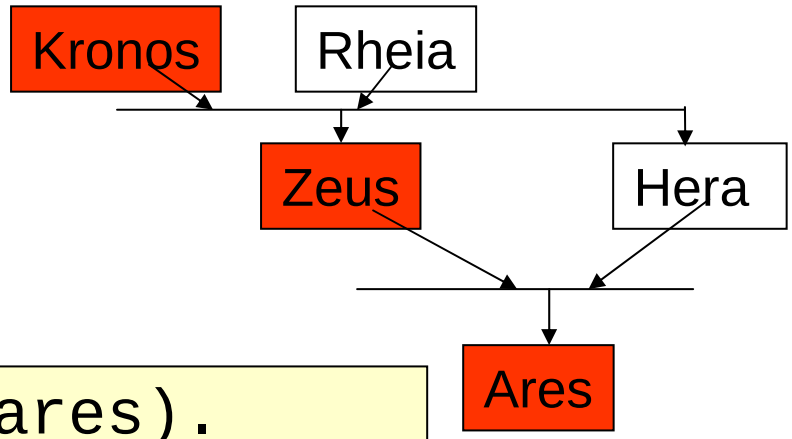
Keine neuen Teilziele.

Beweisversuch gelungen mit Substition: σ (X) = cronus

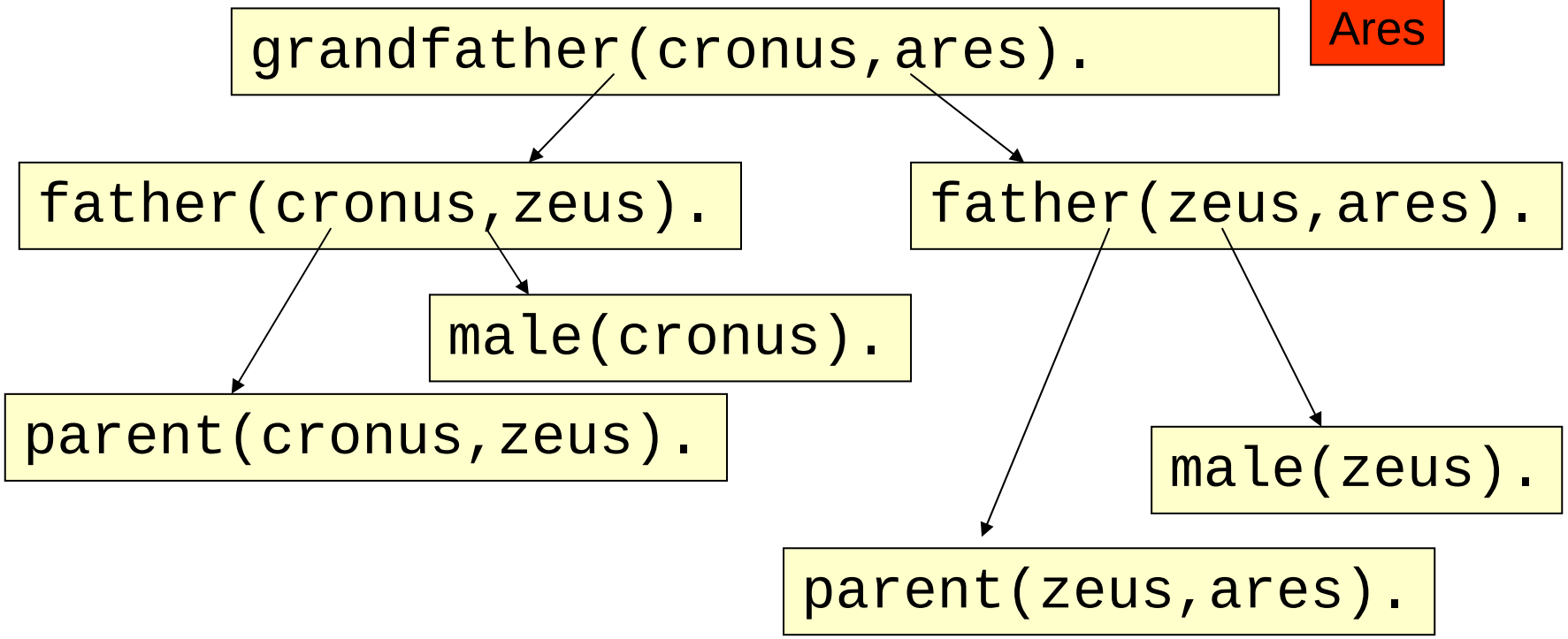
```
?-grandfather(X,ares).
```

Antwort

```
X = cronus?
```

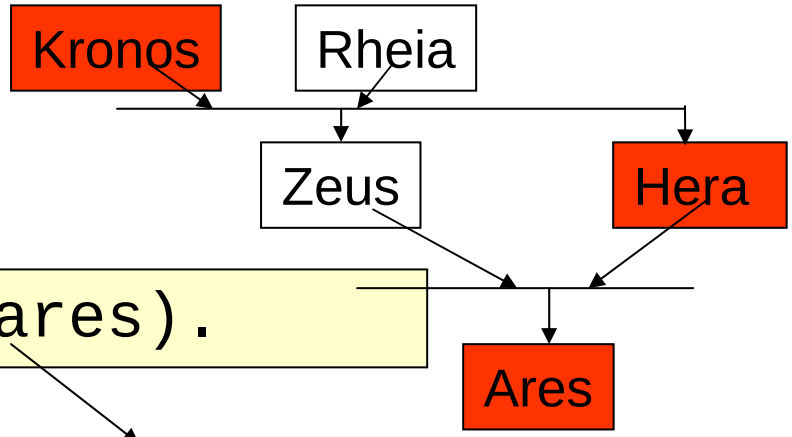


Beweisbaum:

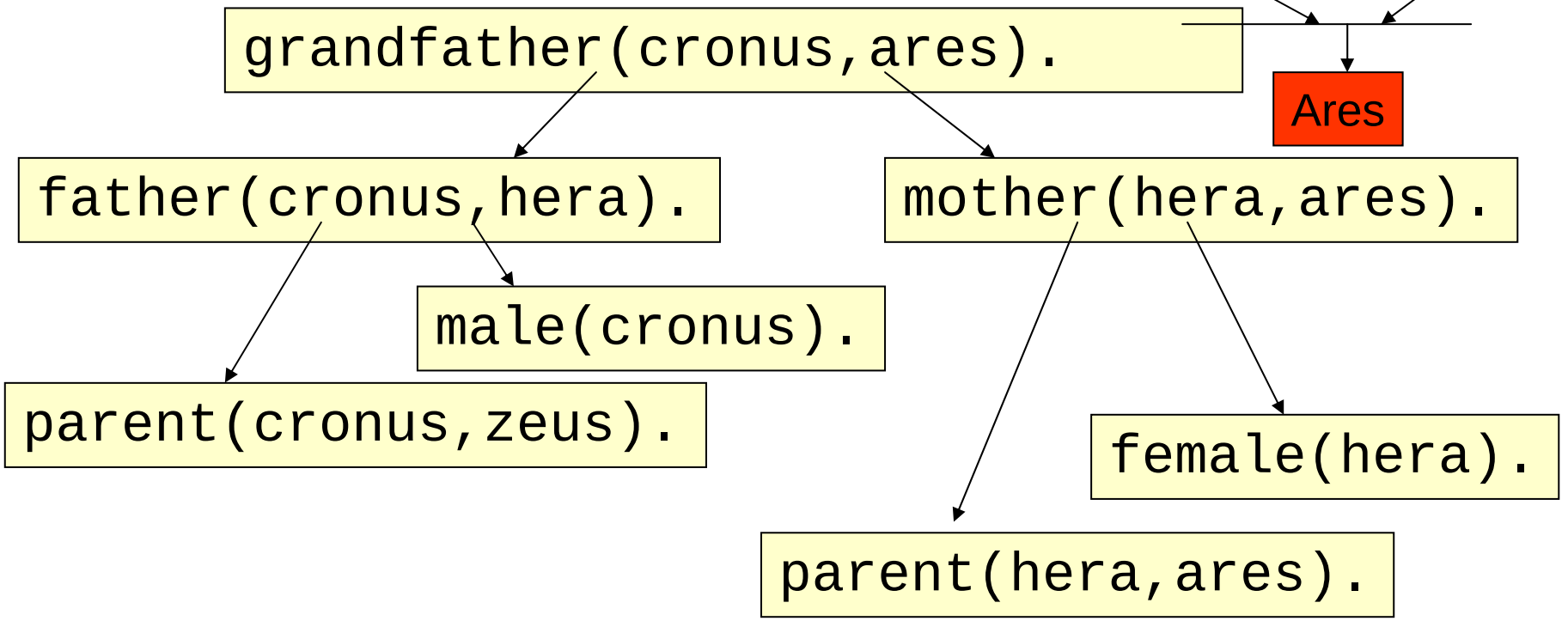


```
?-grandfather(X,ares).
X = cronus?
;
X = cronus?
```

```
grandfather(X,Z):-father(X,Y),mother(Y,Z).
```



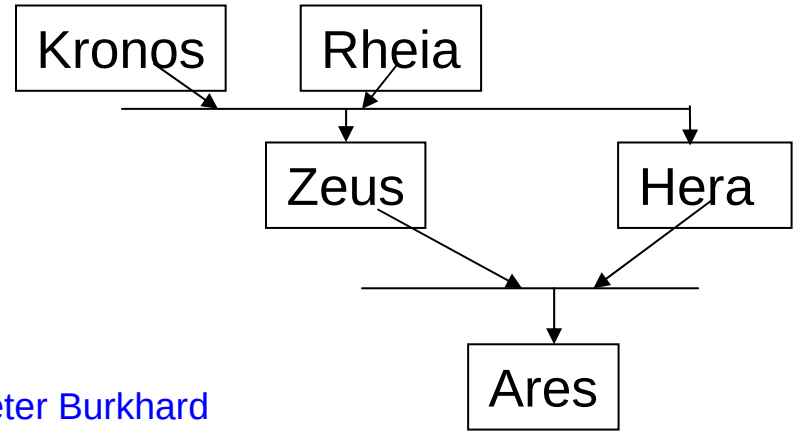
Zweiter Beweisbaum:



Redundanzen ...

... führen wegen der systematischen Durchmusterung aller Beweisversuche zu Wiederholungen von Resultaten

```
?-grandfather(X,ares).
        grandfather(X,Z):-father(X,Y),father(Y,Z).
X = cronus?
;       grandfather(X,Z):-father(X,Y),mother(Y,Z).
X = cronus?
```



Unifikation (matching, instantiation)

Terme unifizieren:

Durch geeigneten **Unifikator** (Variablen-Substitution σ) als Zeichenkette identisch machen.

```
parent(X,ares).
```

```
parent(zeus,Y).
```

σ (X) = zeus σ (Y) = ares

```
parent(zeus,ares).
```

„Instantiation“:

Resultierender Term bei Substitution

Unifikation (matching, instantiation)

in Prolog: Beweisen eines Teilziels erfordert
„Matchen“ von Teilziel und Klauselkopf

```
father(zeus,ares).
father(X3,Y2):-parent(X3,Y2),male(X3).
father(zeus,ares):-
        parent(zeus,ares),male(zeus).
```

Analogie: „Prozedur-Aufruf“

Unifikation (matching, instantiation)

```
father(zeus,ares).
father(X3,Y2):-parent(X3,Y2),male(X3).
father(zeus,ares):-
        parent(zeus,ares),male(zeus).
```

Analogie: „Prozedur-Aufruf“

Parameterübergabe durch Instantiierung:

Bindung von Variablen für gesamte Klausel

$\sigma(X)=zeus$

$\sigma(Y)=ares$

Variable „gehören“ den Klauseln:

Lebensdauer bis zum Backtracking

Sichtbarkeit innerhalb der Klausel

Kein Überschreiben von Werten

Unifikationsregeln

Terme t_1 und t_2 sind unifizierbar, falls

- t_1 und t_2 sind identische Konstanten oder
- t_1 (bzw. t_2) ist eine Variable:
  t_1 (bzw. t_2) wird an t_2 (bzw. t_1) gebunden oder
- $t_1 = \text{funktor}(t_{11},...,t_{1n})$ und $t_2 = \text{funktor}(t_{21},...,t_{2n})$
  sind Strukturen mit identischem funktor (Name, Stelligkeit)
  und die Argumente t_{1i} t_{2i} sind paarweise unifizierbar.

Rekursive Definition,

die Substitution σ ergibt sich schrittweise.

Unifikationsregeln

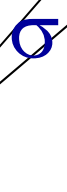
dreieck(P,punkt(X,Y),punkt(1,X)).

dreieck(punkt(2,3),punkt(1,X),punkt(Y,Z)).

Variablenseparierung:

dreieck(P, punkt(X,Y), punkt(1,X)).

dreieck(punkt(2,3),punkt(1,C), punkt(A,B)).



σ(P)= punkt(2,3)

σ(X)=1

σ(Y)=C

σ(A) =1

σ(B) =1

σ(C)=C

dreieck(punkt(2,3),punkt(1,C), punkt(1,1)).

Prolog-Operatoren „=“ und „==“

Das Ziel `term1 = term2` ist erfüllt,

falls `term1` und `term2` unifizierbar sind.

- Variable in `term1` und `term2` werden ggf. instantiiert.

Das Ziel `term1 \= term2` ist erfüllt,
falls `term1` und `term2` nicht unifizierbar sind.

Das Ziel `term1 == term2` ist erfüllt,

falls `term1` und `term2` identisch sind.

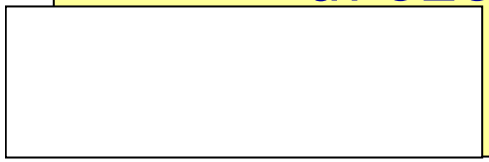
- nicht unifizierte Variable sind nicht identisch

Das Ziel `term1 \== term2` ist erfüllt,
falls `term1` und `term2` nicht identisch sind.

Prolog-Operatoren „=“ und „==“

```
?- dreieck(P,punkt(X,Y),punkt(1,X))
   = dreieck(punkt(2,3),punkt(1,X),punkt(Y,Z)).
```

```
?- dreieck(P,punkt(X,Y),punkt(1,X))
   == dreieck(punkt(2,3),punkt(1,X),punkt(Y,Z)).
```



Occur-Check, $\sigma(X)$ = funktor(..., X, ...)

Beispiel: Programm-Klausel

```
p(X,f (X))
```

Unterschiedliche Reaktionen von Prolog-Systemen auf Anfragen

```
? - p(Y,Y).
```

```
?- p(Y,Y), write(Y).
```

```
?- p(Y,Y), p(Z,Z),Y=Z.
```

aufwendiger Test.

Ignorieren?

Problemzerlegung

Zerlege ein Problem P in einzelne Probleme $P_1,...,P_n$

Löse jedes Problem P_i

Füge die Lösungen zusammen zu P

Beispiele:
Ungarischer Würfel
Kurvendiskussion
Integralrechnung

Problemzerlegung

Klausel als Problemzerlegung

$goal(X_1,...,X_n)$:- $subgoal_1(X_1,...,X_n)$,..., $subgoal_m(X_1,...,X_n)$.

„goal“ gilt (ist beweisbar)
falls alle „subgoals“ gelten (beweisbar sind).

um „goal“ zu beweisen,
beweise alle „subgoals“

Problemzerlegung

$goal(X_1,...,X_n)$:- $subgoal_1(X_1,...,X_n)$,..., $subgoal_m(X_1,...,X_n)$.

um „goal“ zu beweisen,

beweise alle „subgoals“

Problemzerlegung

```
erreichbar(Start,Ziel,Zeit)
:- s_bahn(Start,Zwischenziel,Abfahrt,Ankunft,_),
   erreichbar(Zwischenziel,Ziel,Zeit1),
   berechneZeit(Zeit1,Ankunft,Abfahrt,Zeit).
```

Problemzerlegung

```
berechneZeit(Zeit1,Ankunft,Abfahrt,Zeit).
```

Beabsichtigte Bedeutung:

Zeit = Zeit1 + (Ankunft - Abfahrt)

(Spezielle Notationen für Zeitangaben beachten)

Problemzerlegung:

```
berechneZeit(Zeit1,Ankunft,Abfahrt,Zeit)
:- addiereZeit(Zeit1,Differenz,Zeit),
   subtrahiereZeit(Ankunft,Abfahrt,Differenz).
```

Mit geeignet definierten Prädikaten

`addiereZeit/3, subtrahiereZeit/3`

Unterschied zu prozeduralem Denken

```
berechneZeit(Zeit1,Ankunft,Abfahrt,Zeit)
:- addiereZeit(Zeit1,Differenz,Zeit),
   subtrahiereZeit(Ankunft,Abfahrt,Differenz).
```

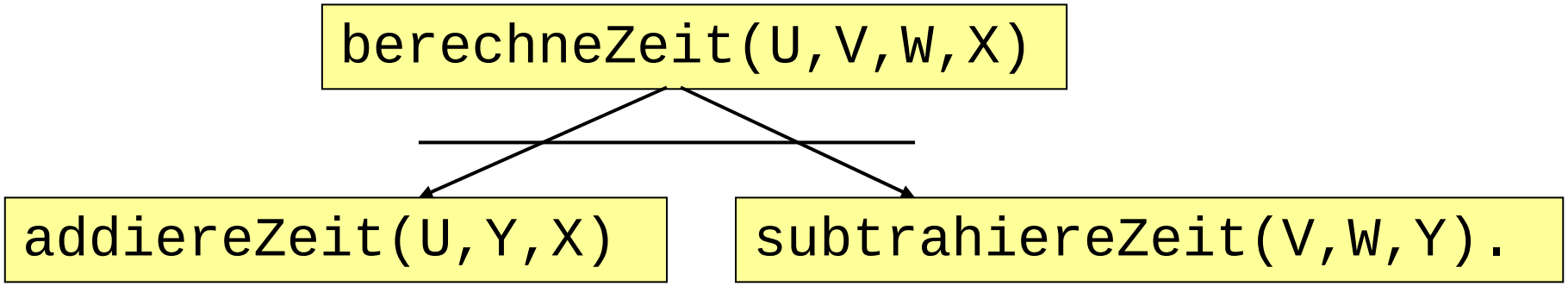
Logische Sicht:

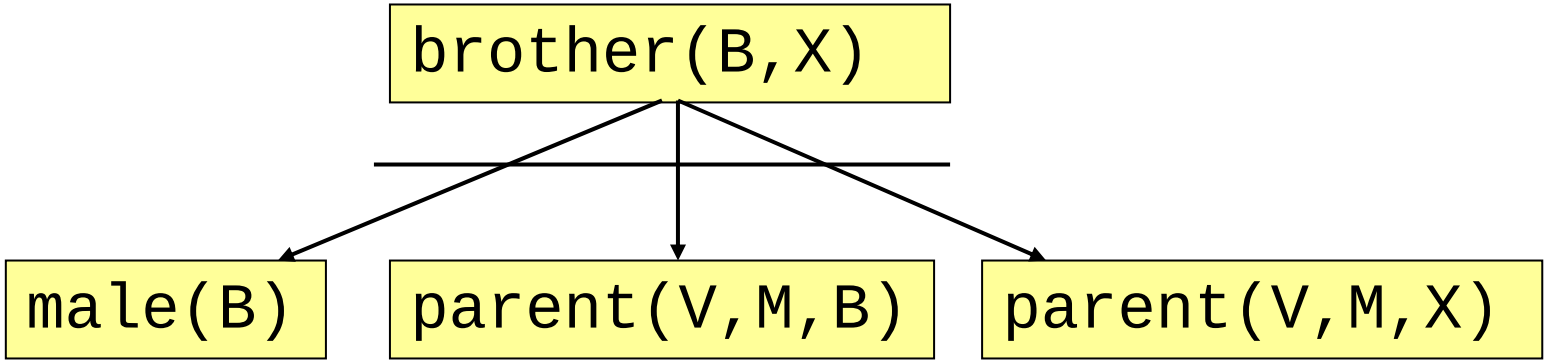
Berechnung von „Differenz“ kann später erfolgen (Bindung statt Wertzuweisung).

Die in Standard-Prolog eingebaute Arithmetik erfordert aus Effizienzgründen allerdings gebundene Eingangsparameter.

Problemzerlegung

Graphische Darstellung





Alternativen für Problemzerlegung

Zerlegung des Problems P in Probleme $P_1,...,P_n$

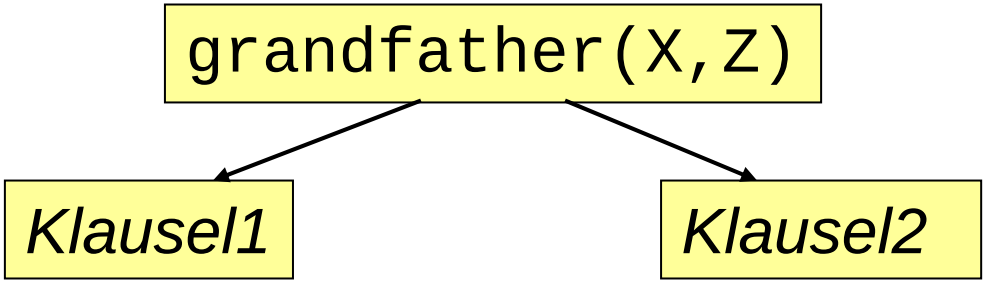
oder in Probleme $P'_1,...,P'_{n'}$

oder ...

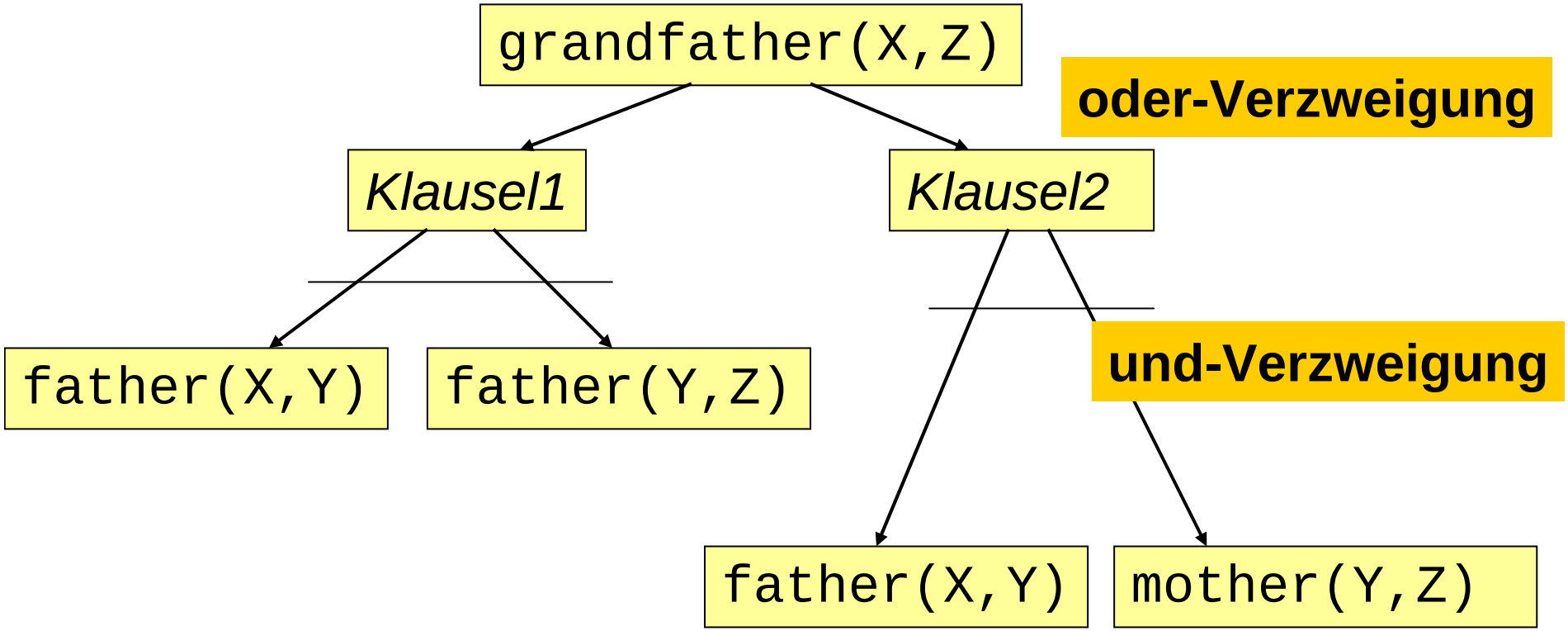
Klauseln einer Prolog-Prozedur bieten Alternativen

```
grandfather(X,Z):-father(X,Y),father(Y,Z).
grandfather(X,Z):-father(X,Y),mother(Y,Z).
```

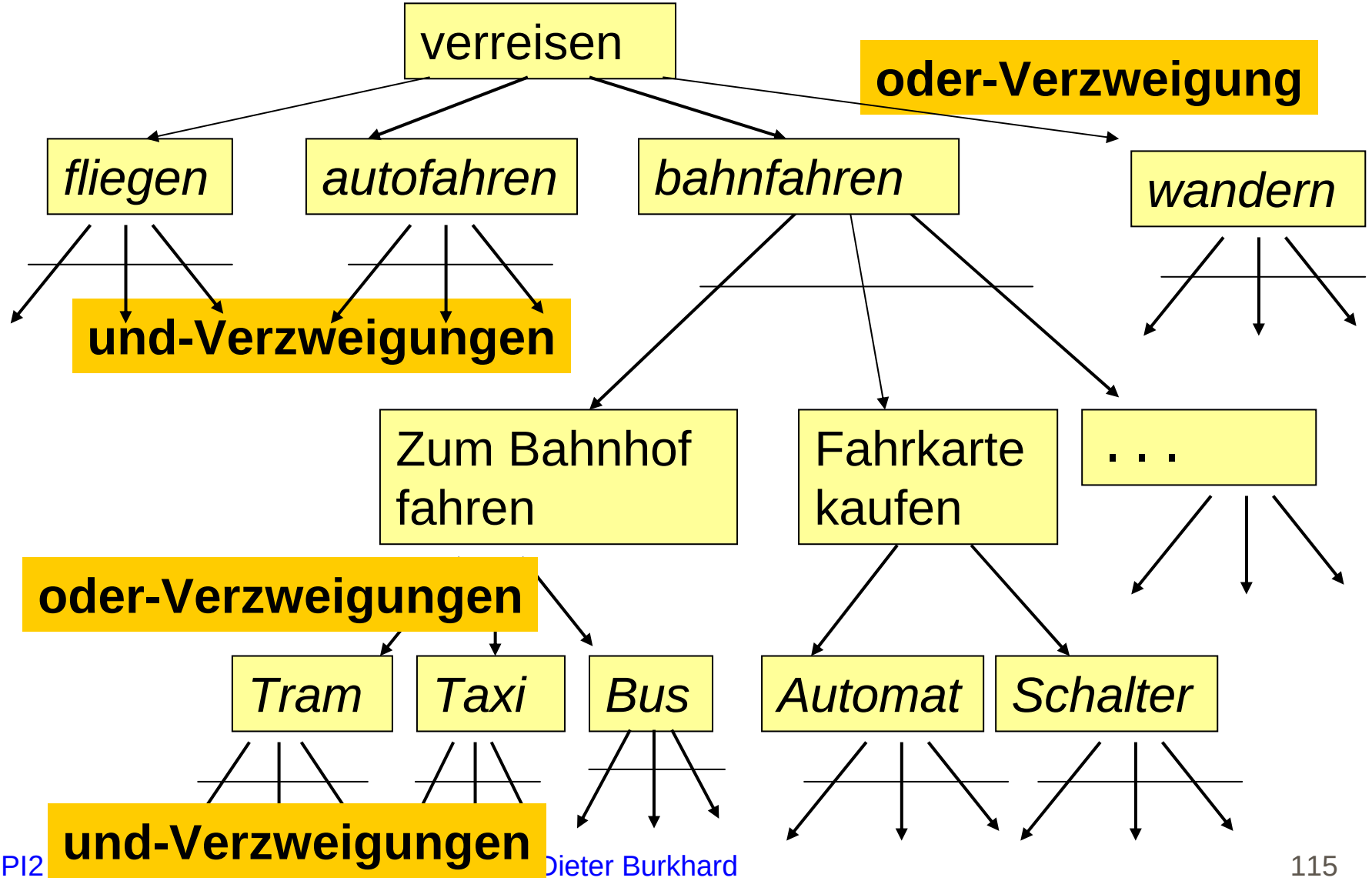
Graphische Darstellung



Alternativen für Problemzerlegung



Kombinierte Verzweigungen



Und-Oder-Baum

Ein und-oder-Baum besteht (abwechselnd) aus

- Knoten mit oder-Verzweigungen und
- Knoten mit und-Verzweigungen

Modell für Problemzerlegungen:

- oder-Verzweigungen für alternative Möglichkeiten zur Problemzerlegung
- und-Verzweigungen für Teilprobleme

Modell für Prolog-Programm:

- oder-Verzweigungen für alternative Klauseln einer Prozedur
- und-Verzweigungen für subgoals einer Klausel

Und-Oder-Baum

Anfrage

Startknoten („**Wurzel**“) modelliert Ausgangsproblem

Knoten ohne Nachfolger („**Blätter**“) sind unterteilt in

- terminale Knoten („primitive Probleme“)
  modellieren unmittelbar lösbare Probleme

Fakt

- nichtterminale Knoten
  modellieren nicht zu lösende Probleme

Unerfüllbares Subgoal

(keine unifizierende Klausel)

Innere Knoten sind unterteilt in

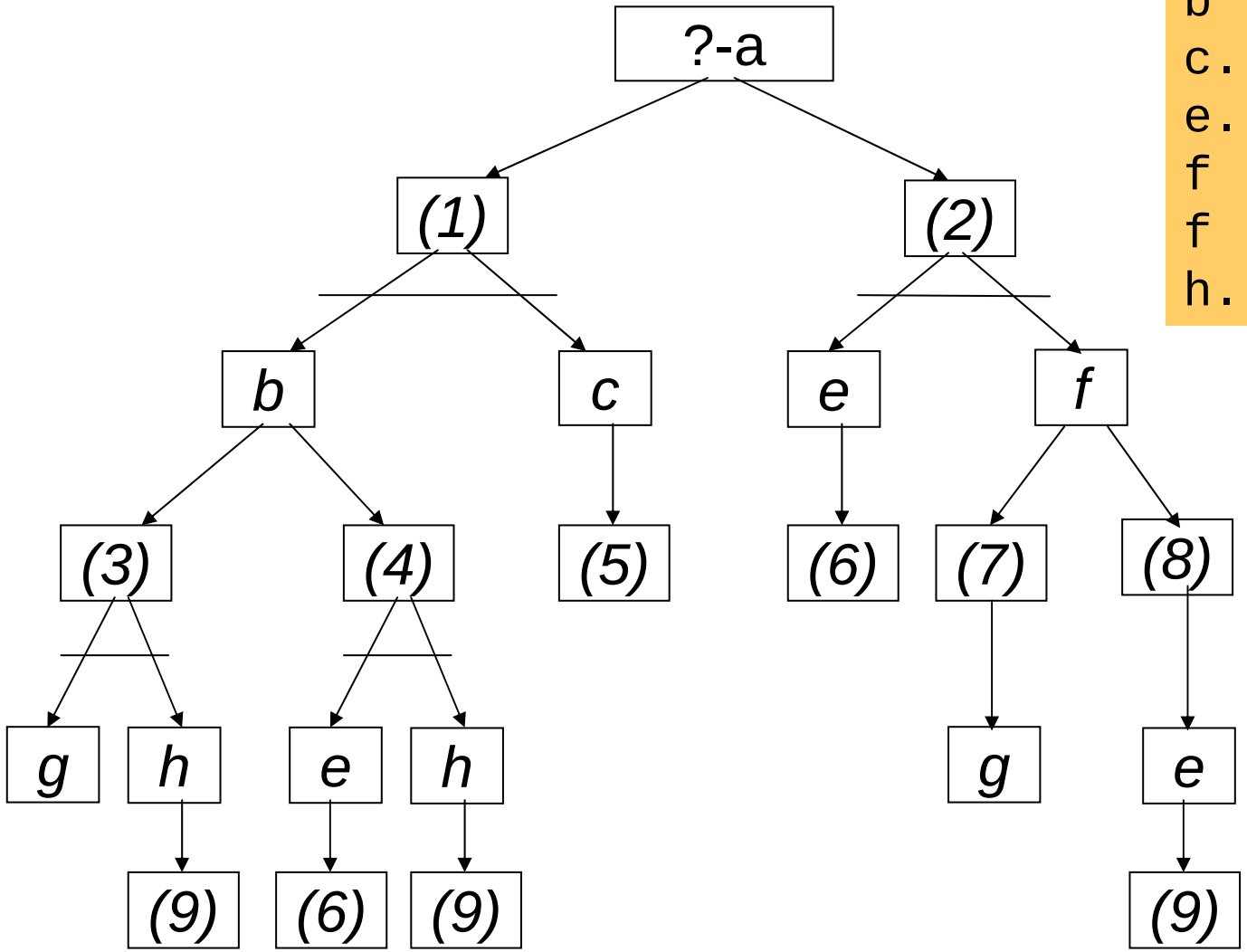
- Knoten mit und-Verzweigung
- Knoten mit oder-Verzeigung

Und-Oder-Baum

```
a :- b,c.          % (1)
a :- e,f.          % (2)
b :- g,h.          % (3)
b :- e,h.          % (4)
c.                 % (5)
e.                 % (6)
f :- g.            % (7)
f :- e             % (8)
h.                 % (9)
```

Und-Oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Lösbare/unlösbare Knoten

Lösbare Knoten:

1. terminale Knoten,
2. Knoten mit Und-Verzweigung:
   falls alle Nachfolger lösbar sind
3. Knoten mit Oder-Verzweigung:
   falls mindestens ein Nachfolger lösbar ist

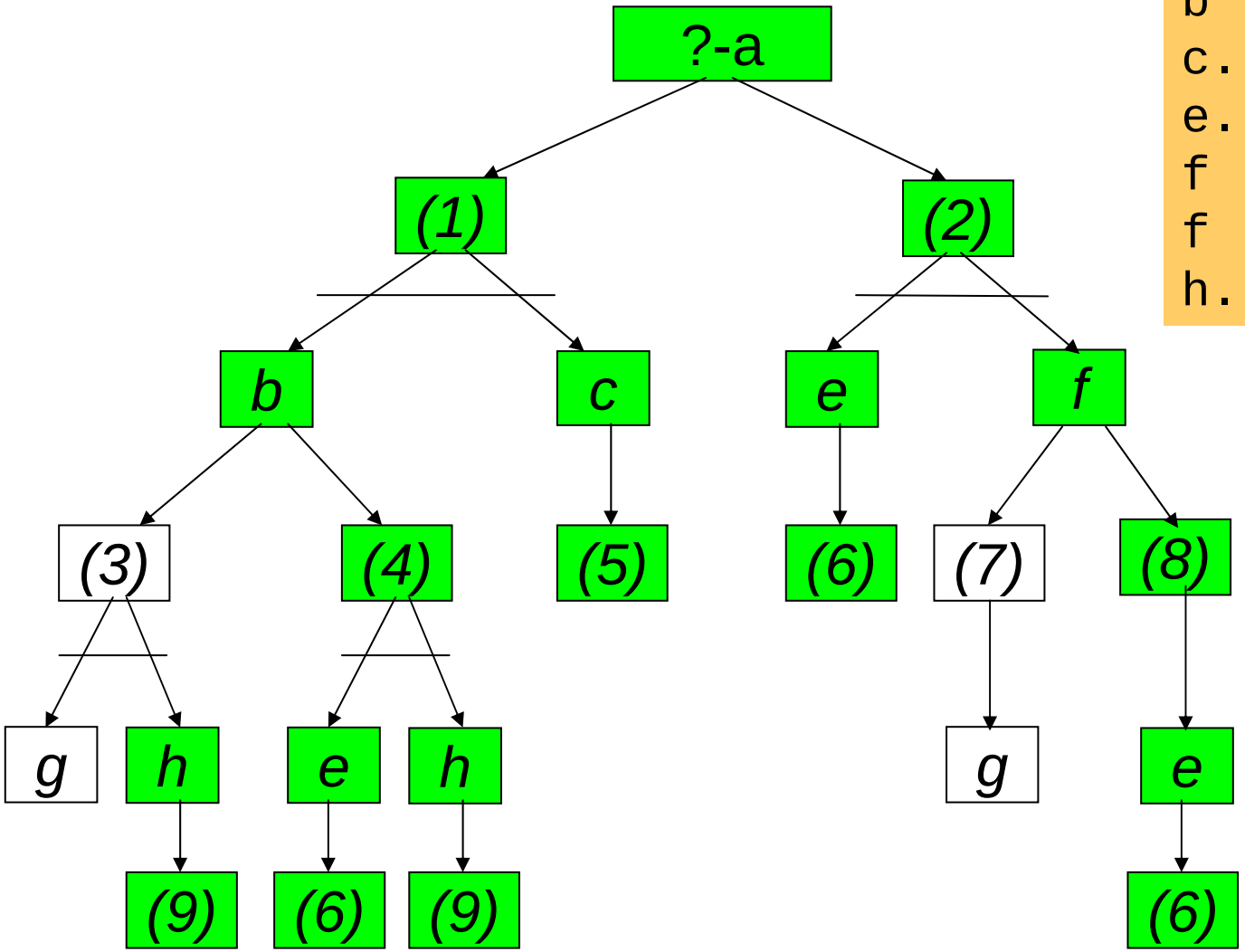
Unlösbare Knoten:

1. nichtterminale Knoten,
2. Knoten mit Und-Verzweigung: falls mindest. ein Nachfolger unlösbar,
3. Knoten mit Oder-Verzweigung: falls alle Nachfolger unlösbar sind

Für endliche Bäume ergibt sich bei Festlegung für die Blätter eine eindeutige Zerlegung in lösbare/unlösbare Knoten (Bedingungen sind jeweils komplementär)

Lösbare/unlösbare Knoten

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Bottom-up-Konstruktionsalgorithmus

Anfang: $M_{LÖSBAR}$:= terminale Knoten

$M_{UNLÖSBAR}$:= nichtterminale Knoten

Zyklus:

Solange nicht alle Knoten untersucht wurden:

Wähle einen Knoten k, dessen Nachfolger alle untersucht wurden.

Falls bei k und-Verzweigung und alle Nachfolger von k in $M_{LÖSBAR}$ oder falls bei k oder-Verzweigung und ein Nachfolger von k in $M_{LÖSBAR}$:

$$M_{LÖSBAR} := M_{LÖSBAR} \cup \{k\} \ ,$$

andernfalls:

$$M_{UNLÖSBAR} := M_{UNLÖSBAR} \cup \{k\} \ .$$

Bottom-up-Konstruktionsalgorithmus

Ergebnis für endliche Und-oder-Bäume:
Zerlegung der Knoten in
lösbare Knoten ($M_{LÖSBAR}$) und
unlösbare Knoten ($M_{UNLÖSBAR}$)

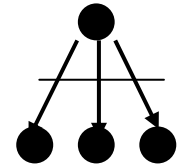
Das Ausgangsproblem
kann genau dann durch Problemzerlegung gelöst werden,
wenn der Startknoten in $M_{LÖSBAR}$ ist.

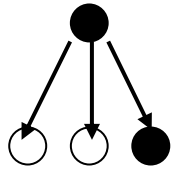
Falls das Ausgangsproblem lösbar ist, kann ein Lösungsbaum konstruiert werden.

Lösungsbaum

Endlicher Teilbaum des Und-oder-Baums mit
folgenden Eigenschaften:

- Enthält nur lösbare Knoten,
- enthält Wurzelknoten,
- bei Und-Verzweigungen sind alle Nachfolger enthalten,
- bei Oder-Verzweigungen ist (genau) ein Nachfolger enthalten

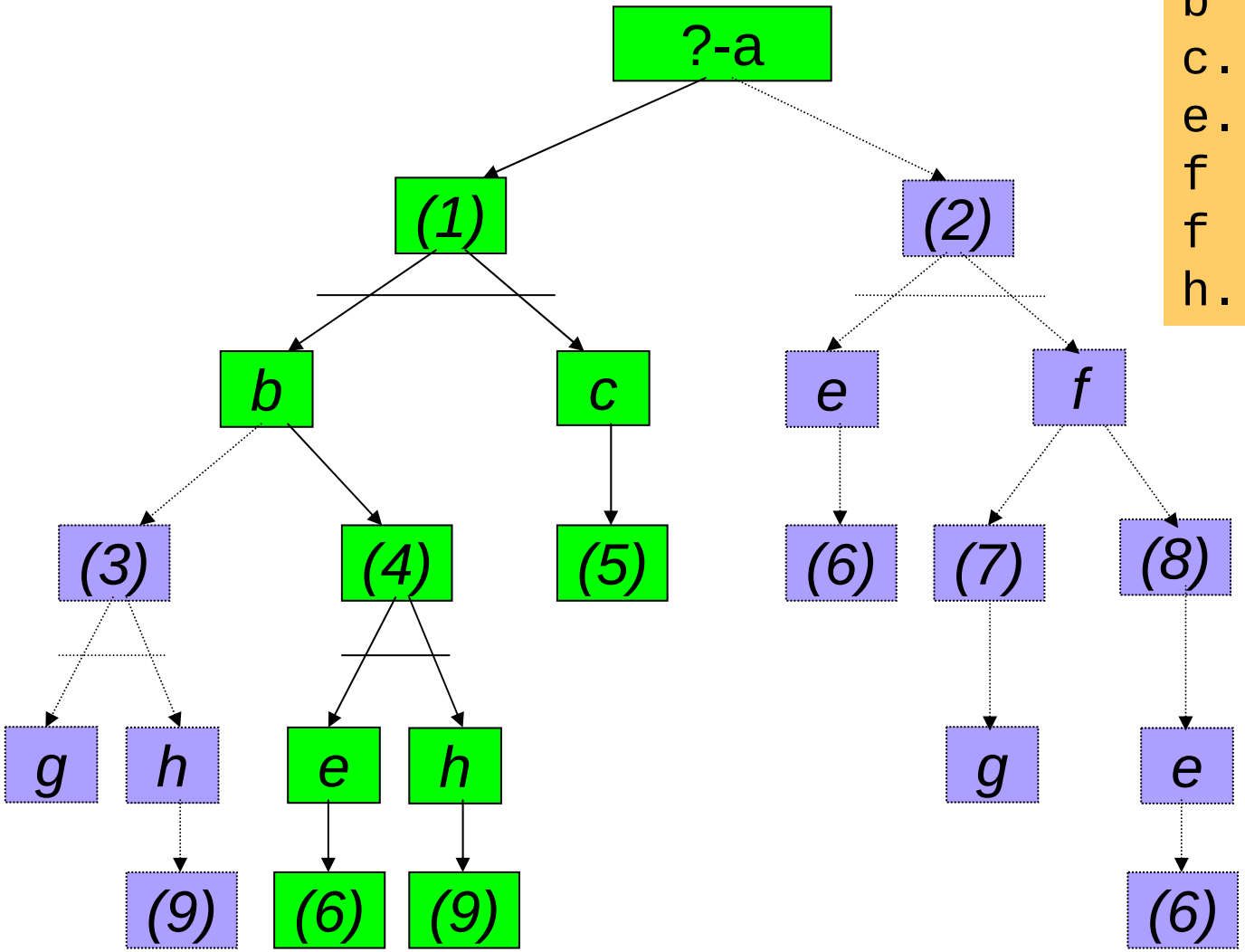




Modell für „Beweisbaum“ in PROLOG

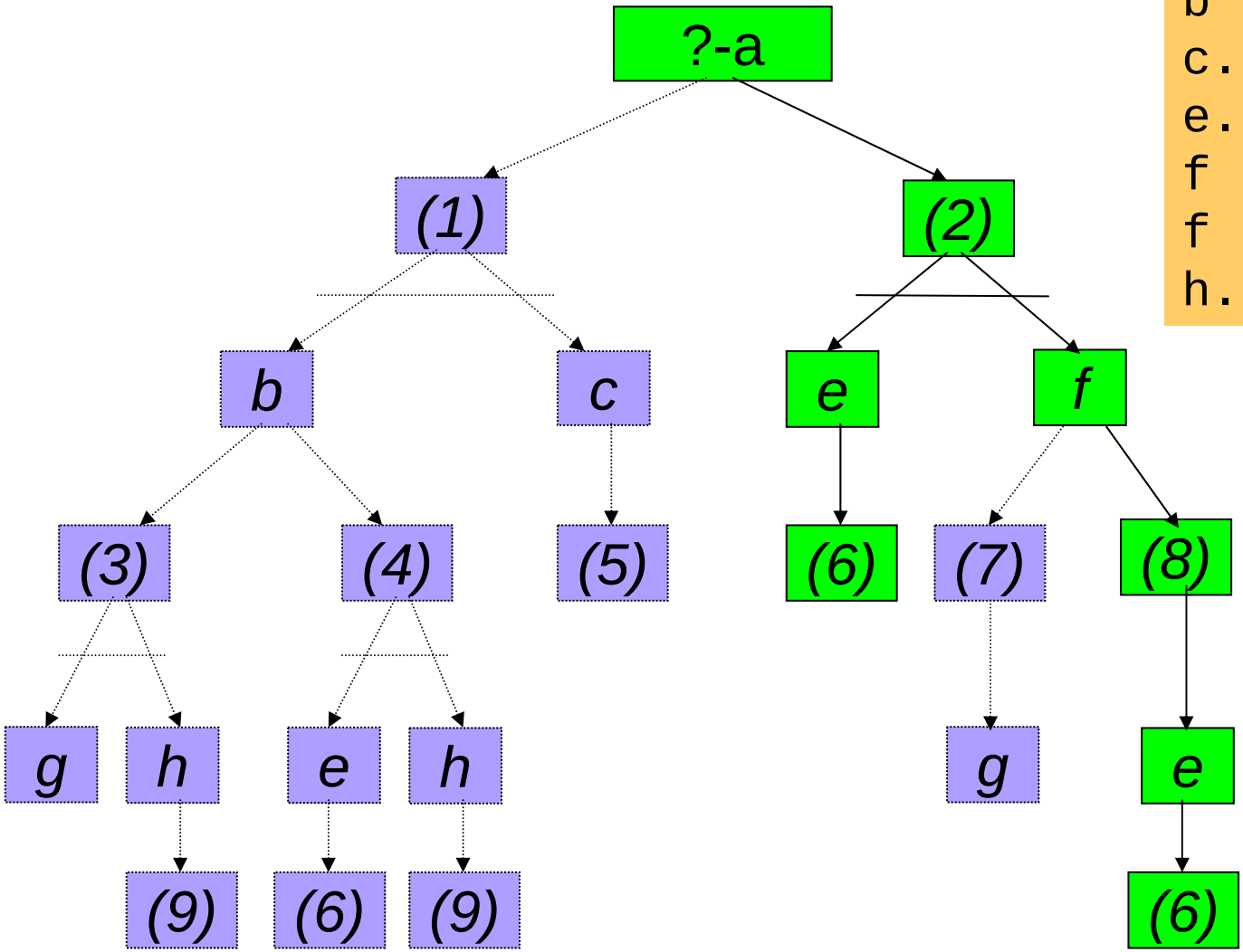
Lösungsbäume

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```

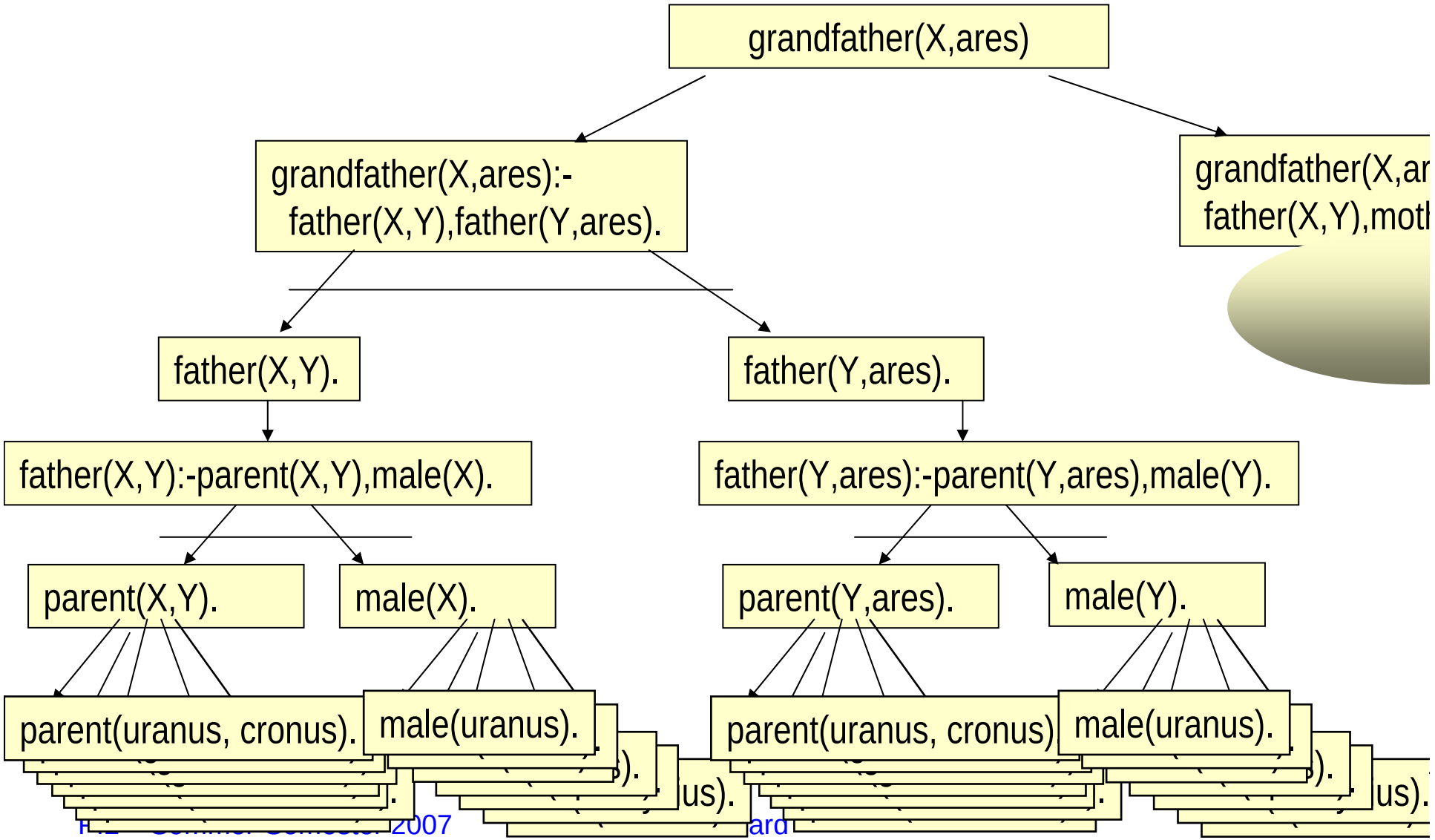


Lösungsbäume

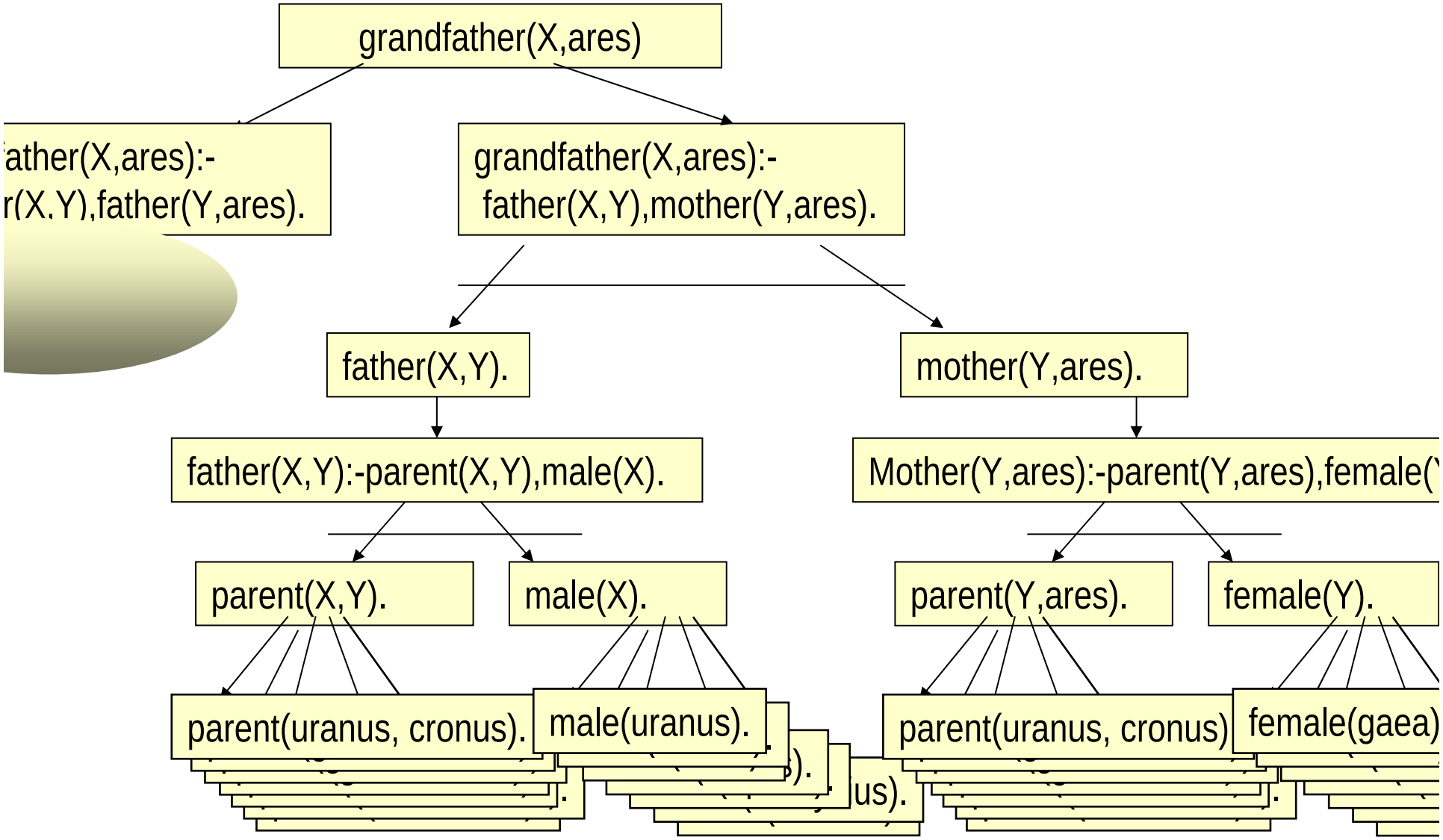
```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



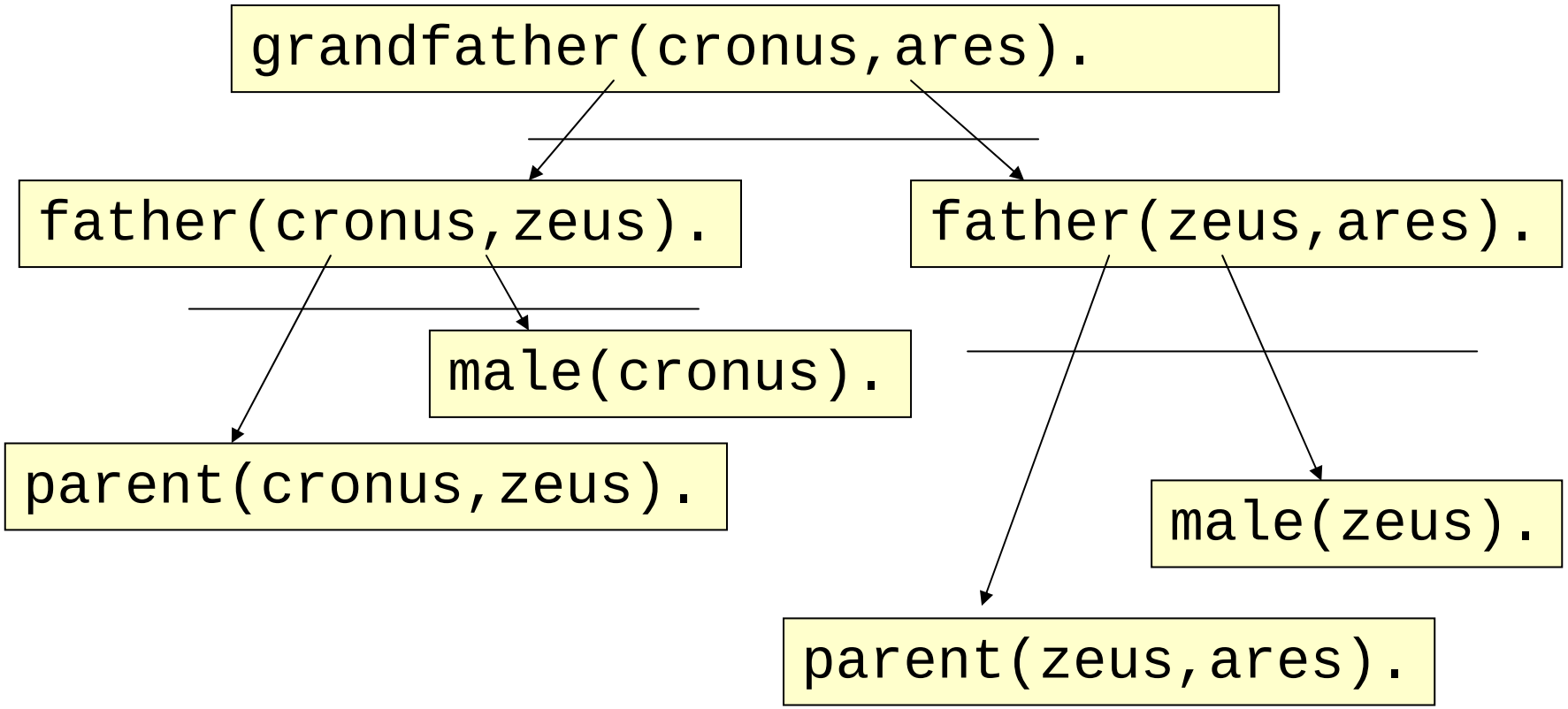
Und-oder-Baum modelliert Beweisversuche



Und-oder-Baum modelliert Beweisversuche



Lösungsbaum (Beweisb.) modelliert Beweis



Richtungen für Suchverfahren

Bottom-up/Backward-Suche:

Vom Ziel zum Anfang

Top-down/Forward-Suche

Vom Anfang zum Ziel

Bottom-up-Suche für Prolog ungeeignet

Modell für nicht-deterministische Suche

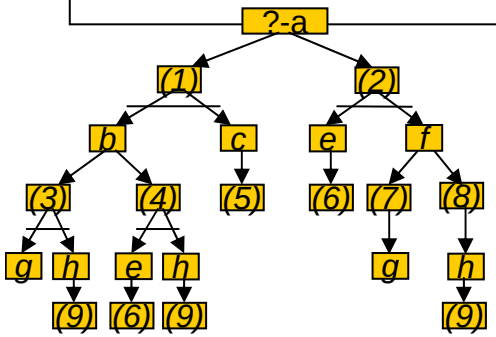
Vereinfachtes Schema (ohne Unifikation: „AK“)

1. subgoals=$\{g_1,...,g_n\}$
2. Falls subgoals = $\varnothing$: Erfolg.
3. Wähle $g \in$ subgoals .
4. Wähle Klausel k: $g :- g'_1,...,g'_m$ der Prozedur für g .
   Falls kein solches k existiert: Mißerfolg (des Versuchs).
5. subgoals := $(\{g_1,...,g_n\} - \{g\}) \cup \{g'_1,...,g'_m\}$.
   Weiter bei 2.

Alle Varianten (in 3 und 4) probieren,
falls keine zum Erfolg führt: „Nicht beweisbar“

Varianten für ND-Suche

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Alle Varianten (**jeden Weg von der Wurzel aus**) probieren, falls keine zum Erfolg führt: „Nicht beweisbar“

Prolog-Interpreter

Einschränkung der Varianten

- Reihenfolge innerhalb einer Klausel (Und-Verzweigung)

  (alle subgoals müssen erfüllt werden)

  links vor rechts
- Reihenfolge innerhalb einer Prozedur (Oder-Verzweigung)

  (Alternativen für Beweis)

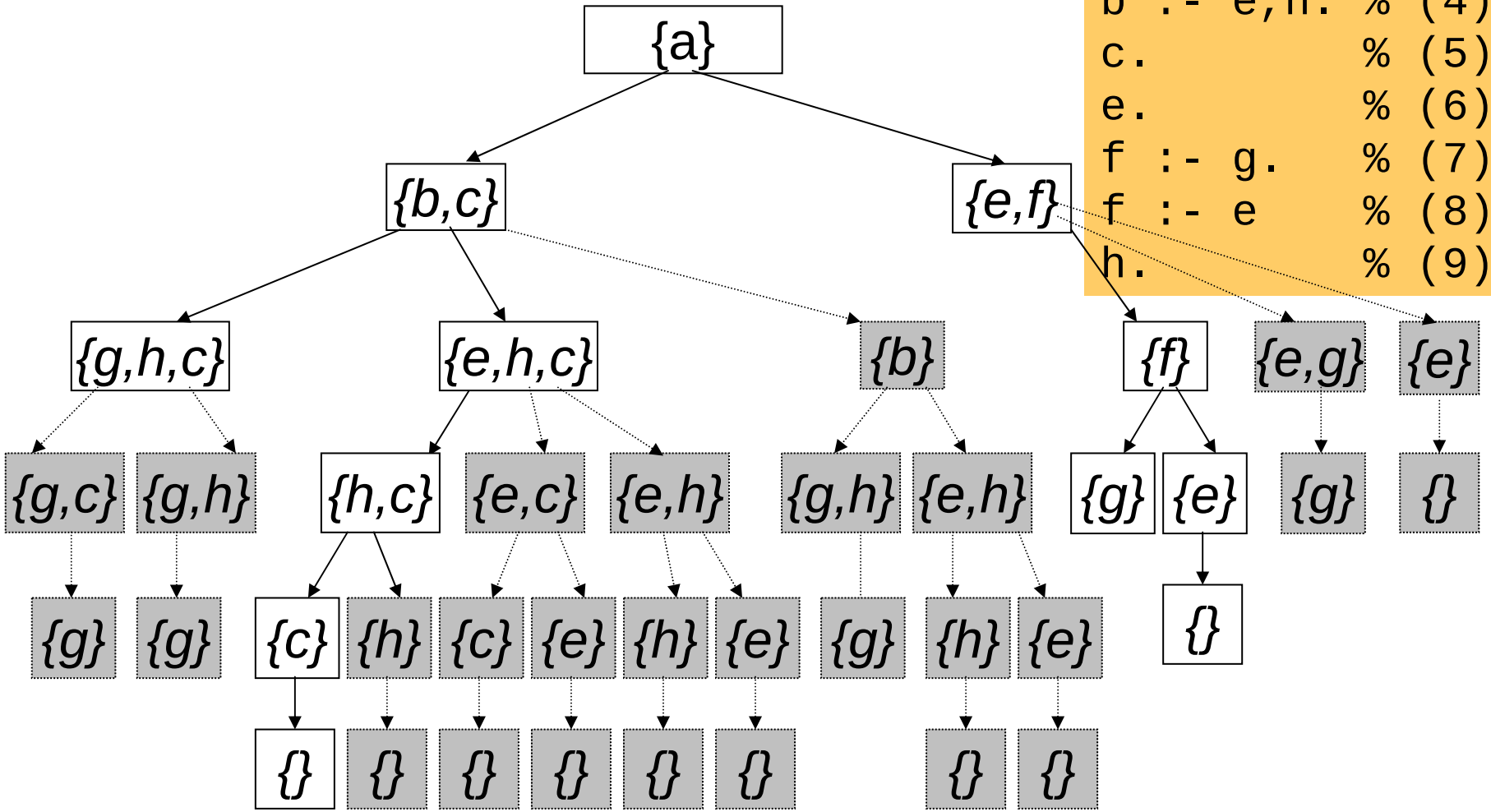
  oben vor unten

Zu zeigen wäre (Vollständigkeit):

Wenn Beweis existiert, dann auch schon hierbei.

Einschränkung der Varianten

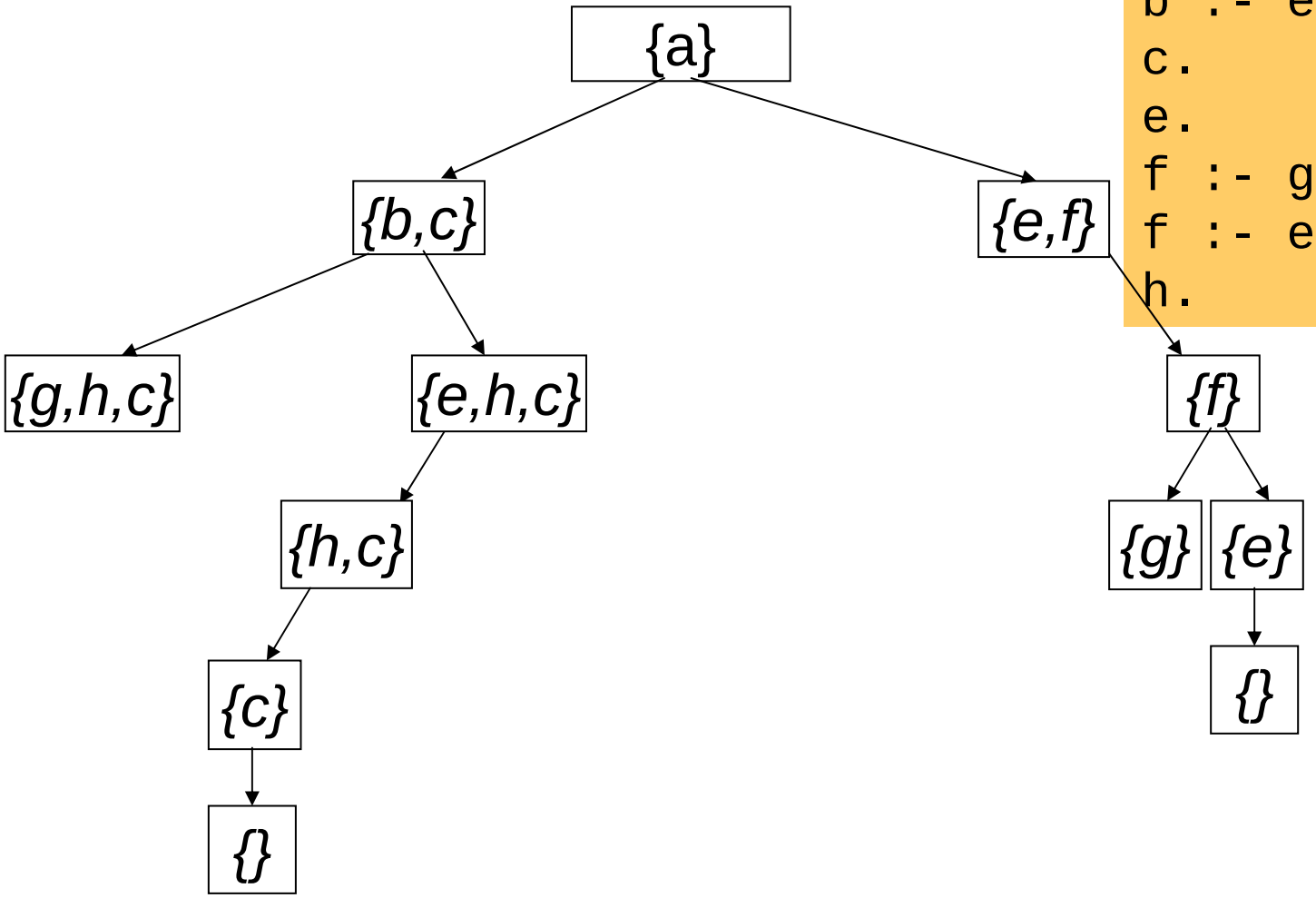
```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Subgoals sind alle zu beweisen,
Reihenfolge links vor rechts

Einschränkung der Varianten

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Subgoals sind alle zu beweisen,
Reihenfolge links vor rechts

Backtracking

Effizienzgewinn

Suchpfade werden nicht vollständig von der Wurzel aus neu probiert, sondern nur stückweise.

Bei Alternativen:

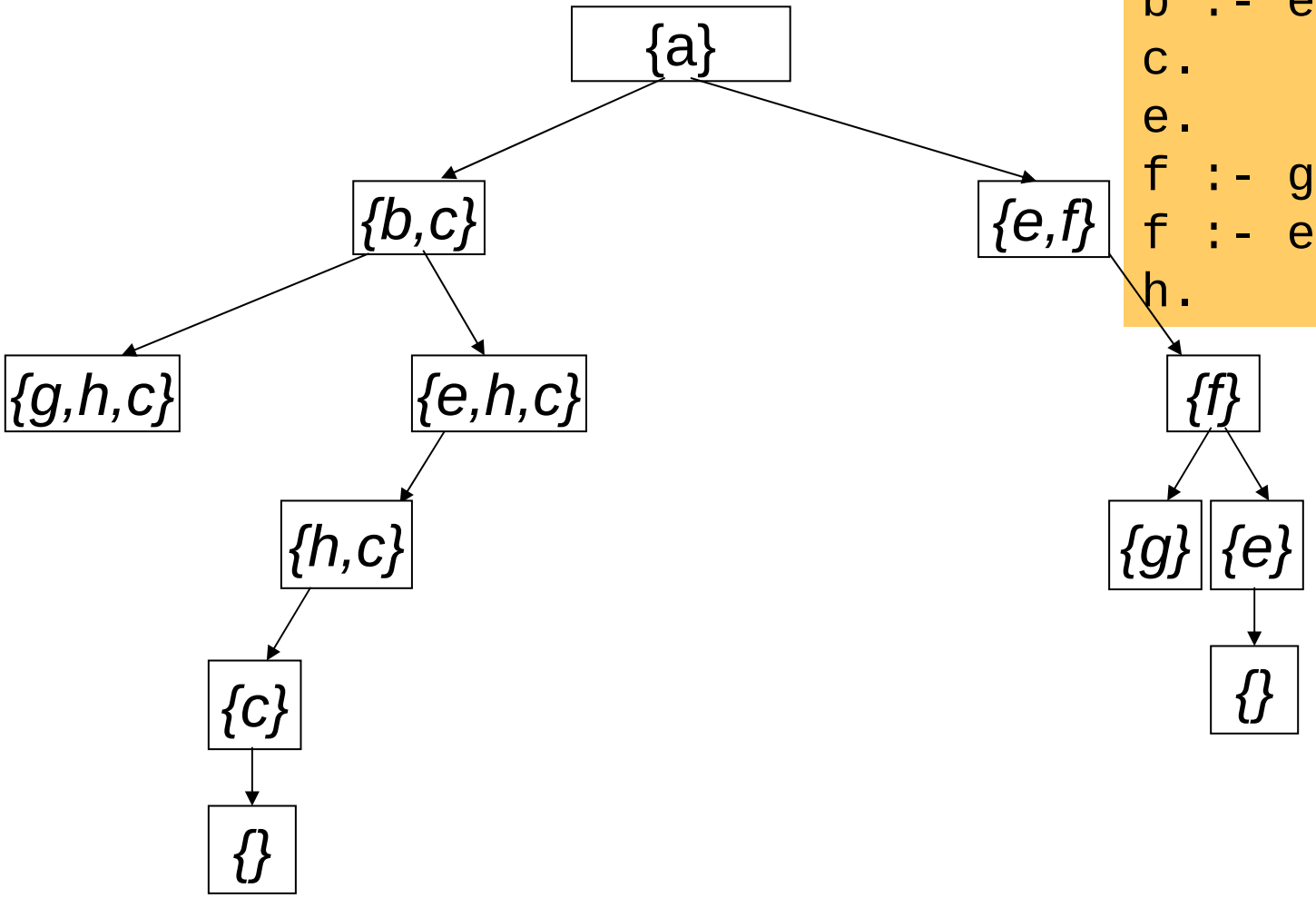
Einfügen eines „Backtrack-Punktes“ („Choicepoint“)

„Backtracking“:

Bei Fehlschlag am jüngsten „Backtrack-Punkt“ andere Alternative verfolgen

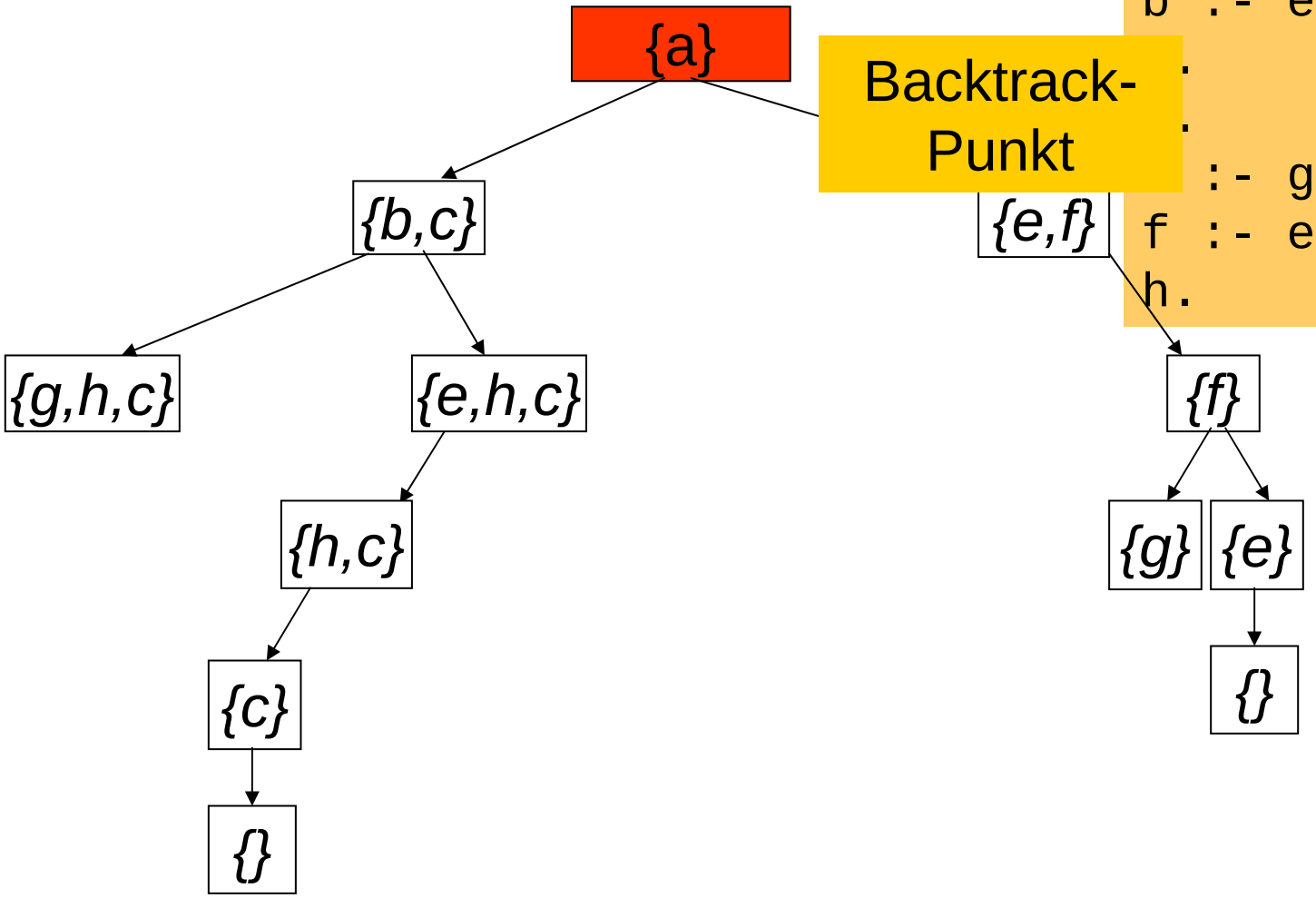
Backtracking

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



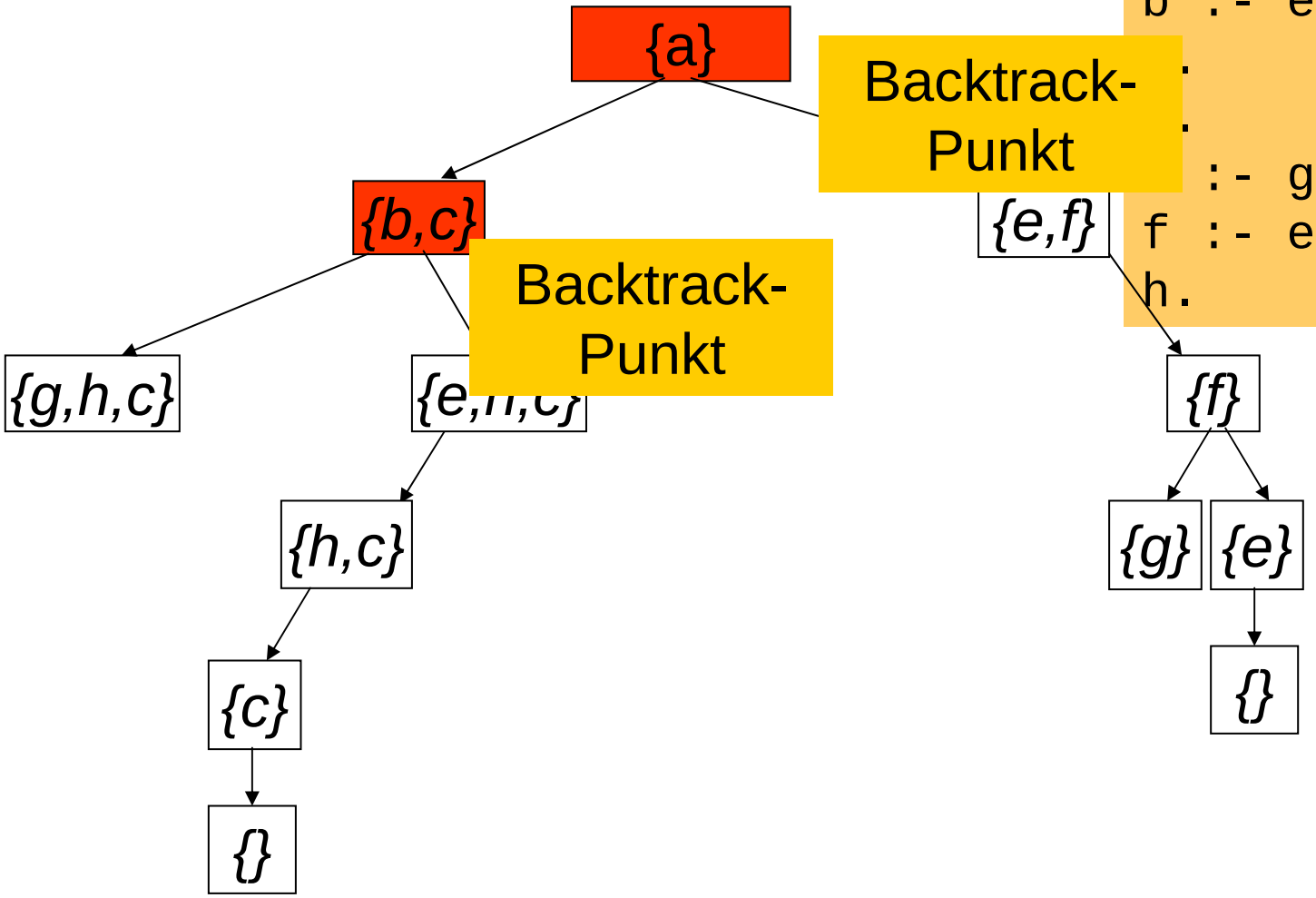
Backtracking

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
.         % (5)
.         % (6)
  :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



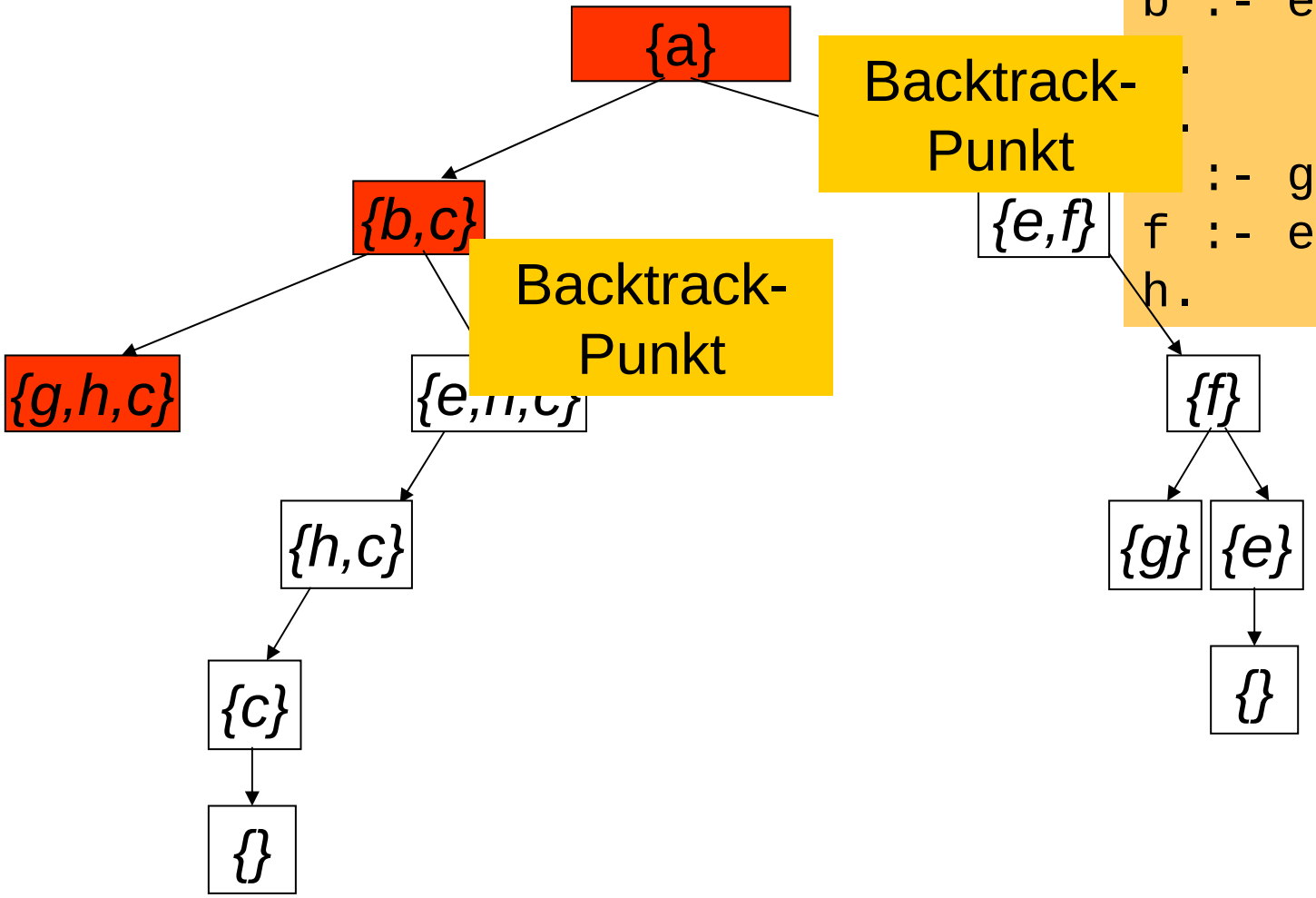
Backtracking

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
.         % (5)
.         % (6)
  :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



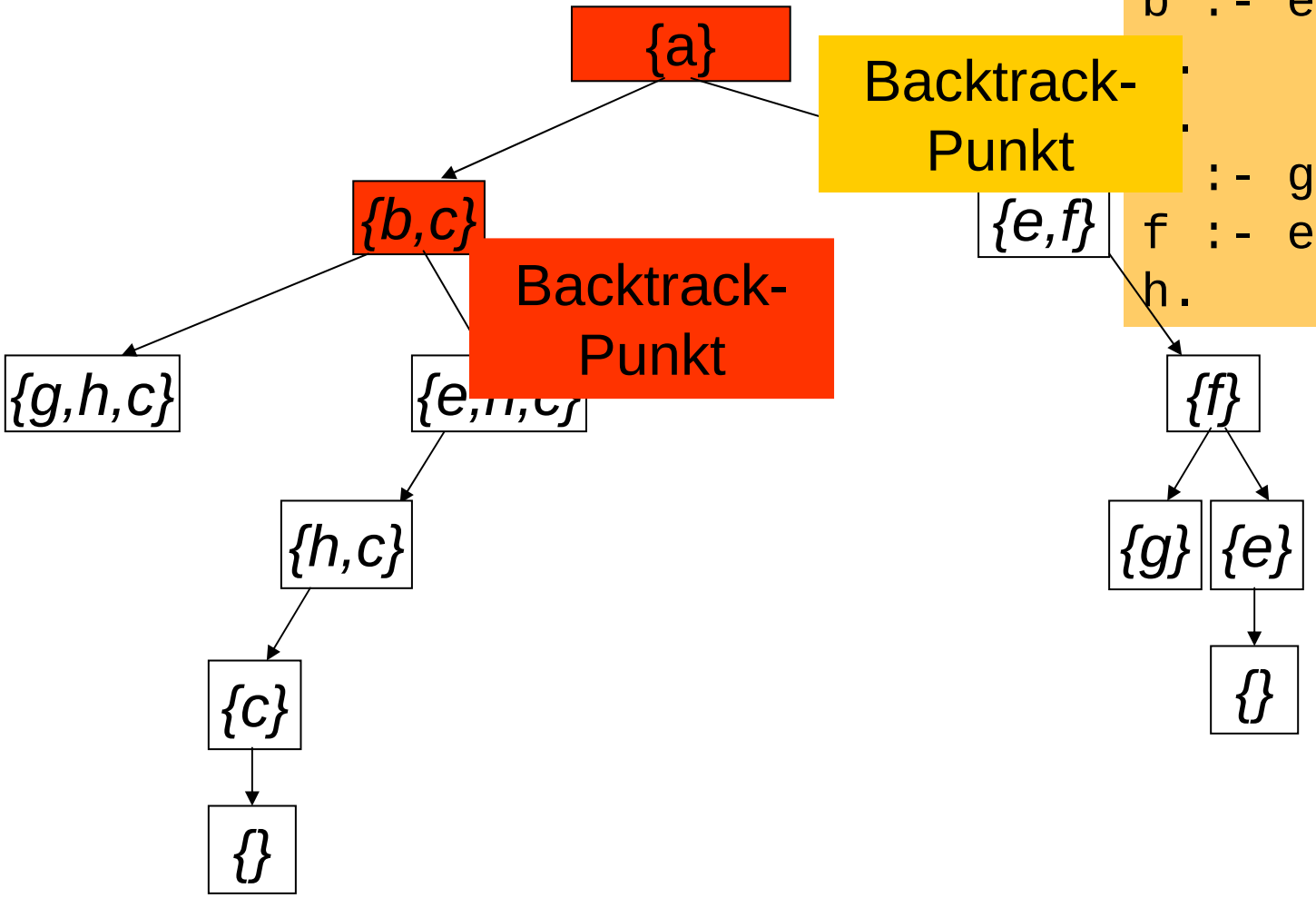
Backtracking

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
.         % (5)
.         % (6)
  :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



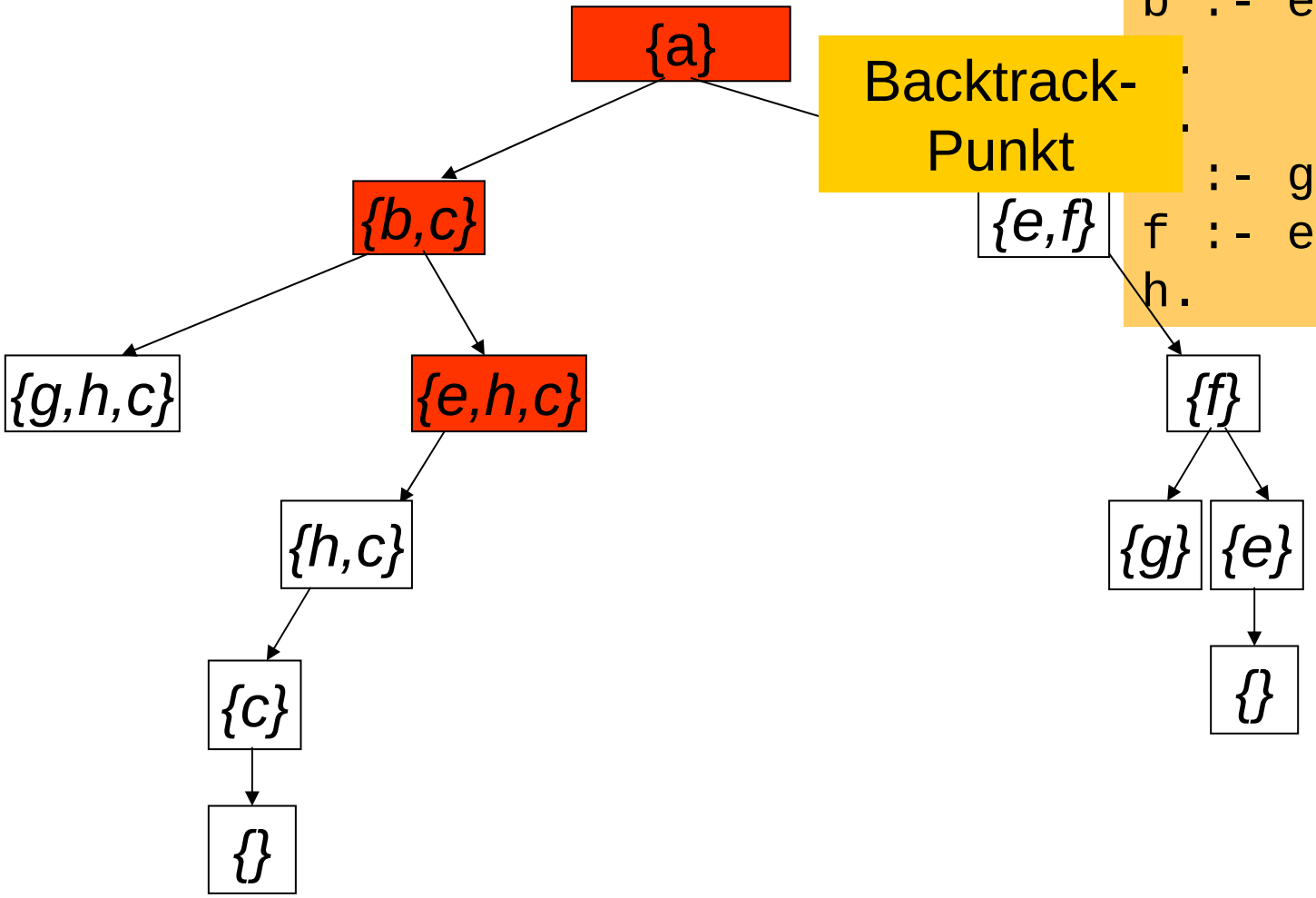
Backtracking

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
.         % (5)
.         % (6)
  :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



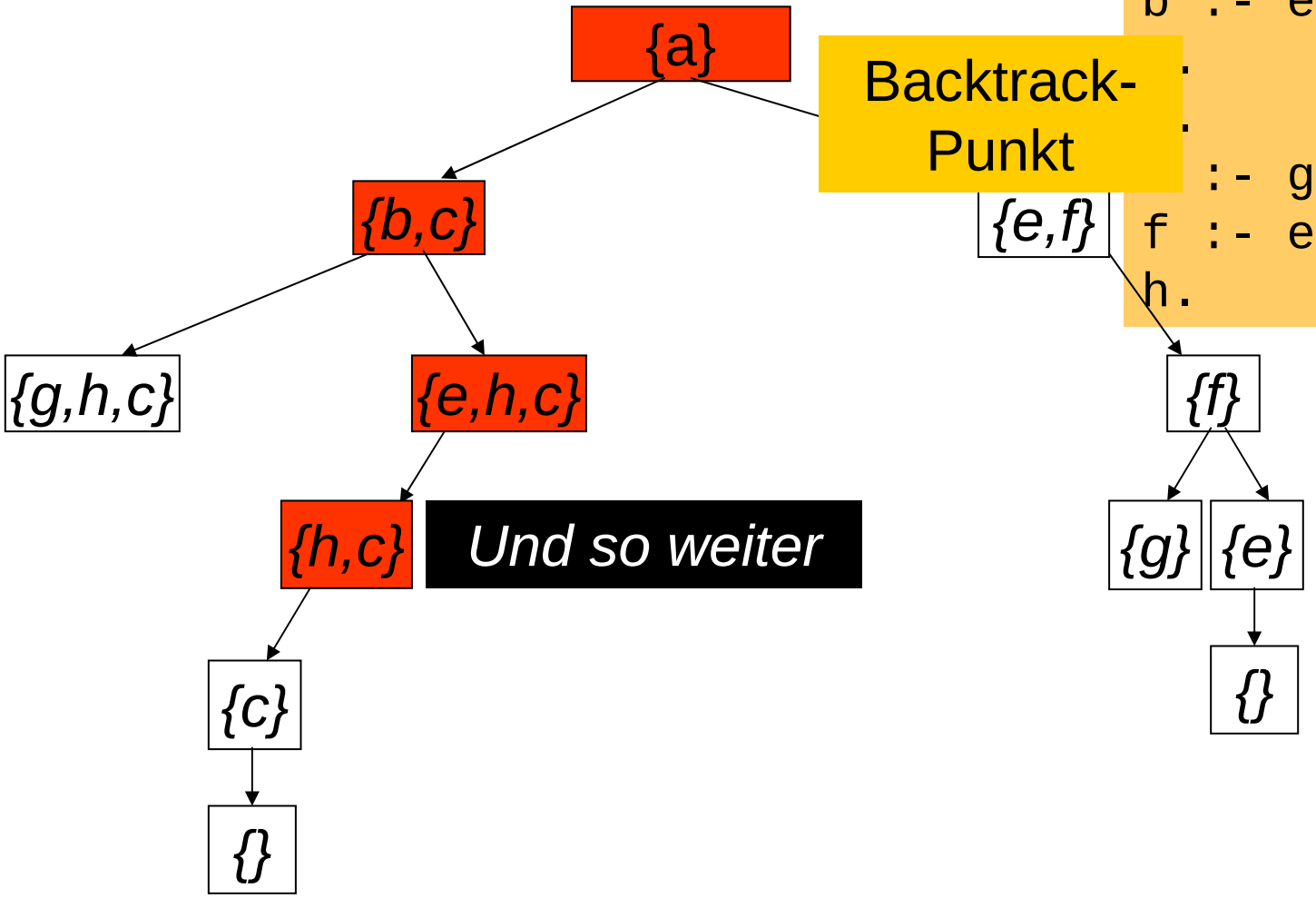
Backtracking

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
.         % (5)
.         % (6)
  :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Backtracking

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
.         % (5)
.         % (6)
  :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Modelle für Prolog-Suche

Vereinfachtes Schema (ohne Unifikation: „AK“)

1. subgoals=[g_1,...,g_n] .
2. Falls subgoals = [ ] : Erfolg .
3. k sei nächste Klausel der Prozedur für g_1 :

   g_1 :- g'_1,...,g'_m .

   Falls kein solches k existiert: Backtracking.

   Falls kein Backtracking möglich: Misserfolg.
4. subgoals := [g'_1,...,g'_m ,g_2,...,$g_{n,}$].

   Weiter bei 2.

Prolog-Interpreter

- Und-Verzweigung: links vor rechts
- Teilziele der Reihe nach vollständig abarbeiten

  Verfolgen eines Zweiges in die Tiefe

- Oder-Verzweigung: oben vor unten

  Linke Zweige zuerst

- Backtracking bei Fehlschlag:

  Rückkehr zu Alternative an oder-Verzweigung

  Nächster Zweig einer Oder-Verzweigung

Modelle für Prolog-Suche

1. subgoals=[g_1,...,g_n] .
2. Falls subgoals = [ ] : Erfolg .
3. k sei nächste Klausel der Prozedur für g_1 :

   g_1 :- g'_1,...,g'_m .

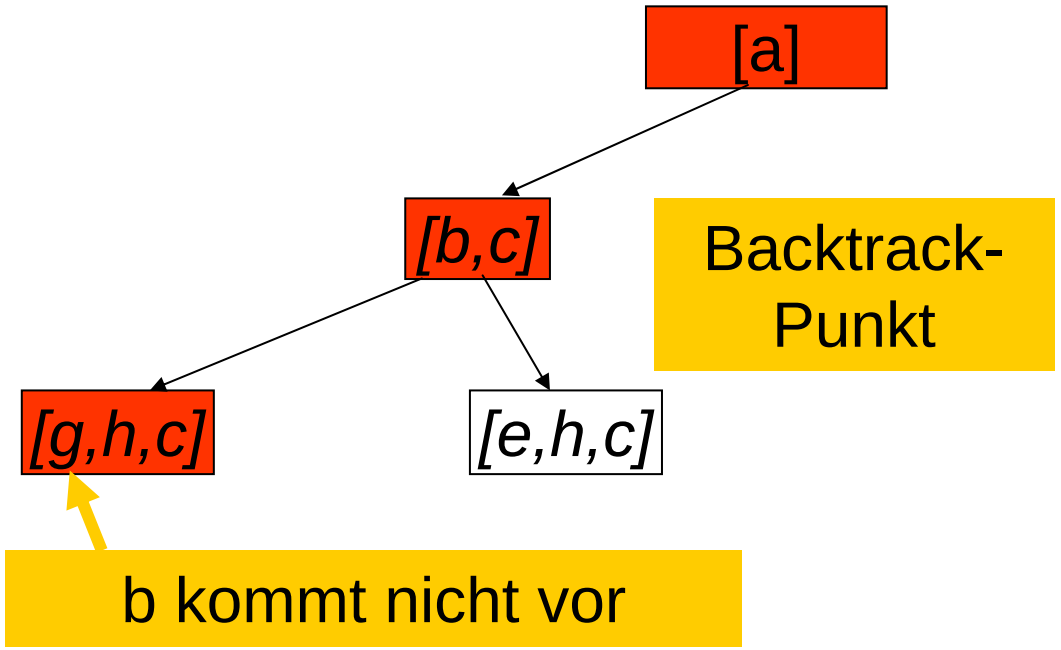
   Falls kein solches k existiert: Backtracking.

   Falls kein Backtracking möglich: Misserfolg.
4. subgoals := [g'_1,...,g'_m ,g_2,...,$g_{n,}$].

   Weiter bei 2.

Problem: Backtrack-Punkt in subgoal –Liste nicht repräsentiert

Modelle für Prolog-Suche

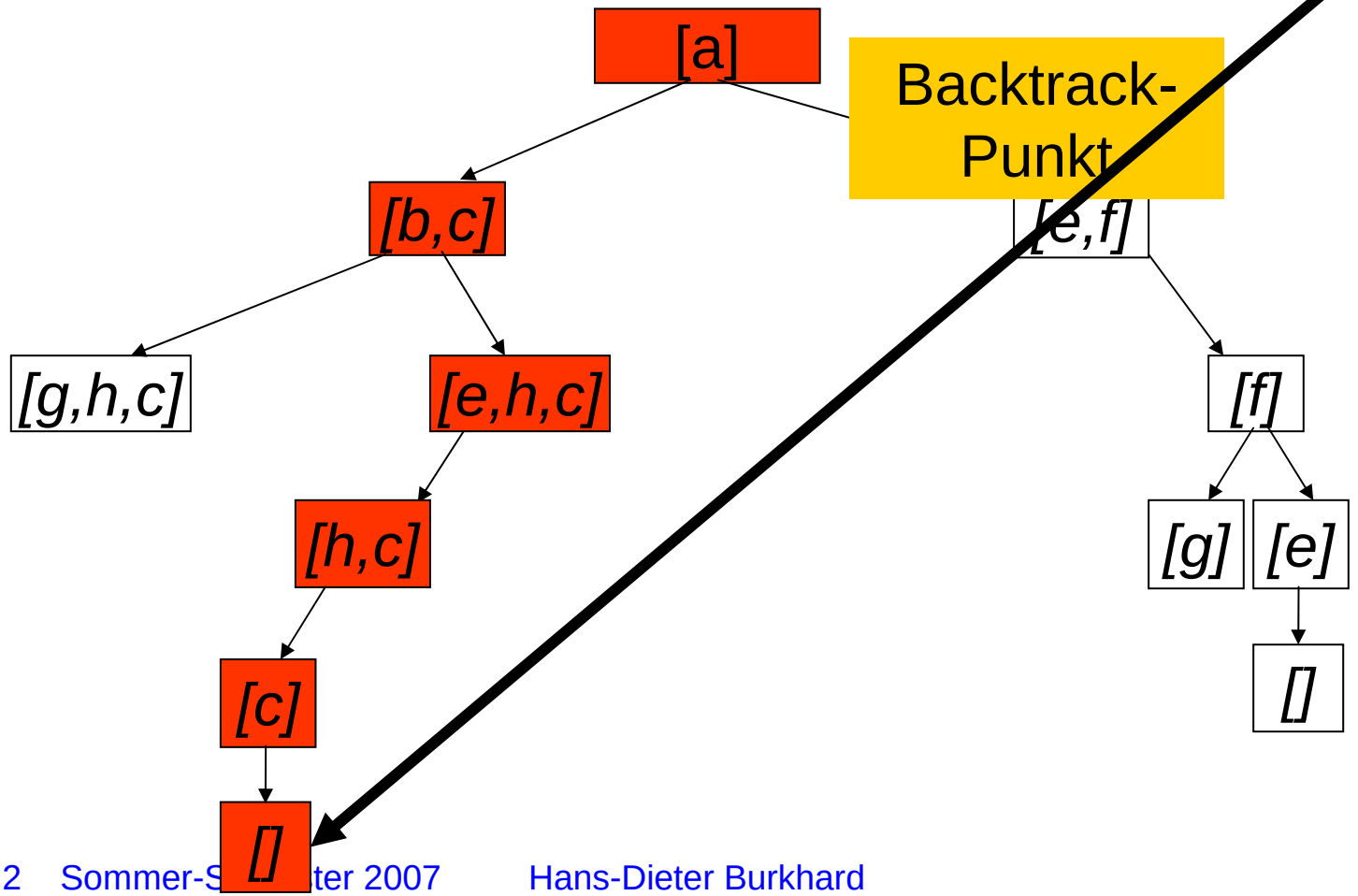


Problem: Backtrack-Punkt in subgoal –Liste nicht repräsentiert

Modelle für Prolog-Suche

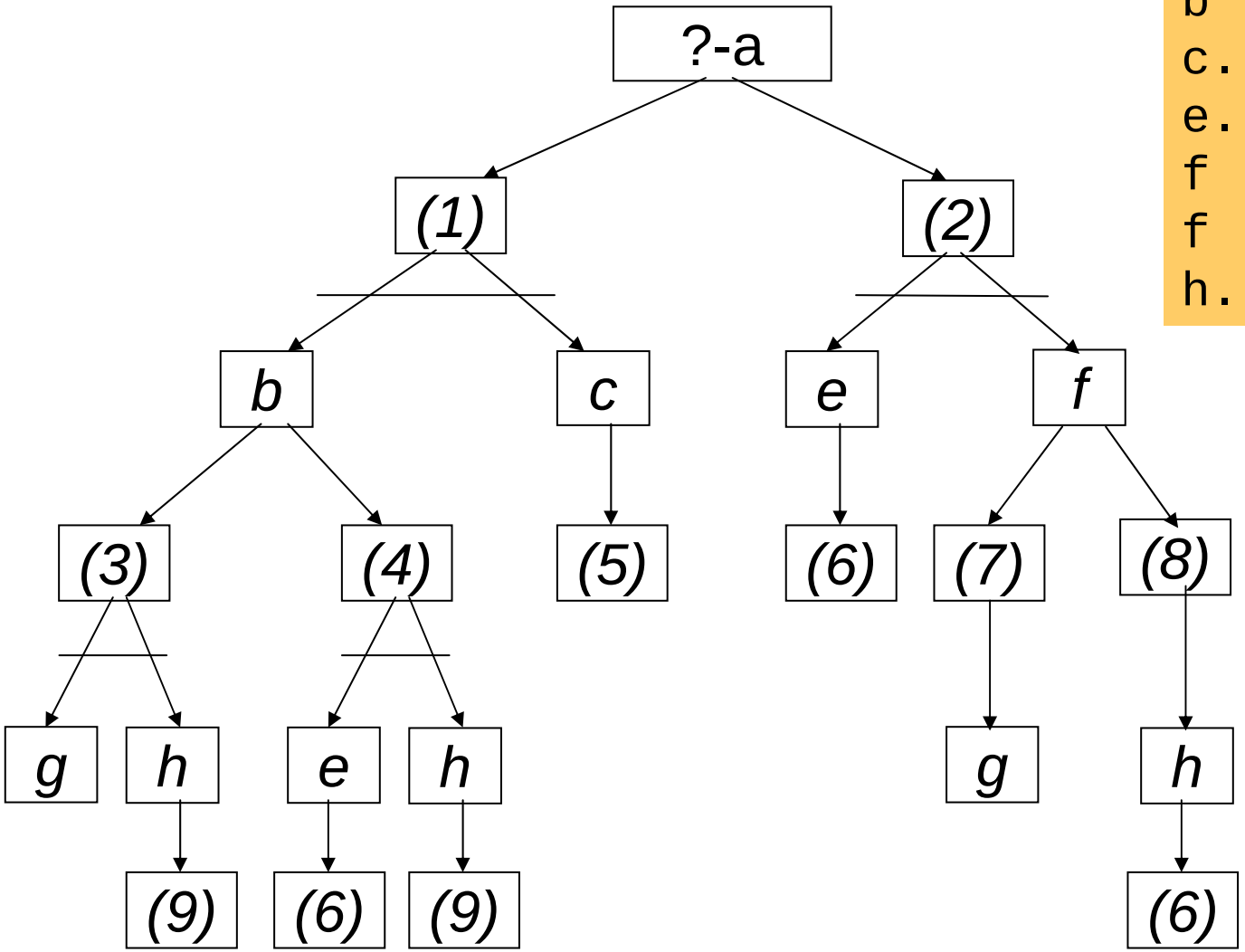
Quelle für Missverständnisse: *Warum?*

Bei erfolgreichem ersten Beweis ist subgoal-Liste leer



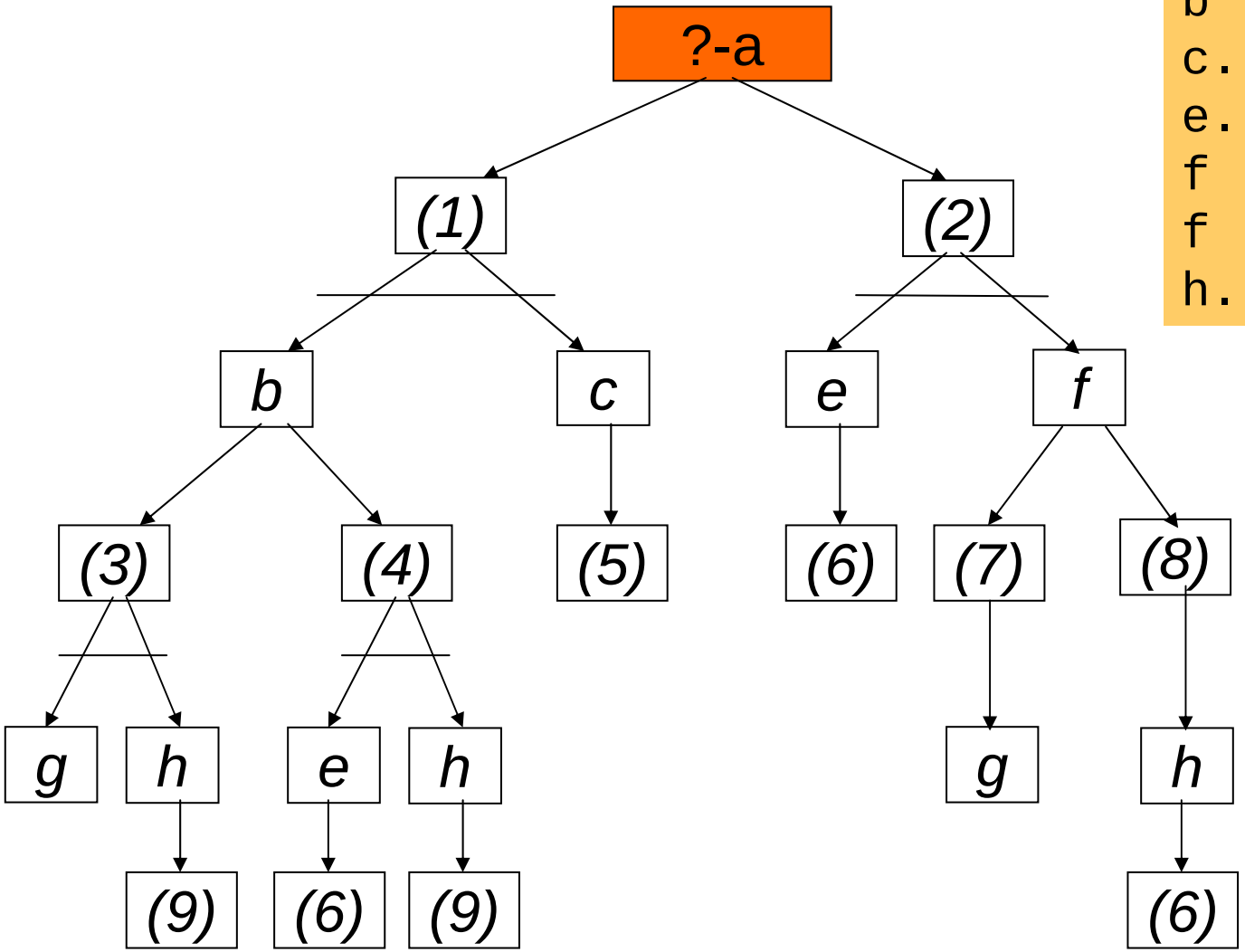
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



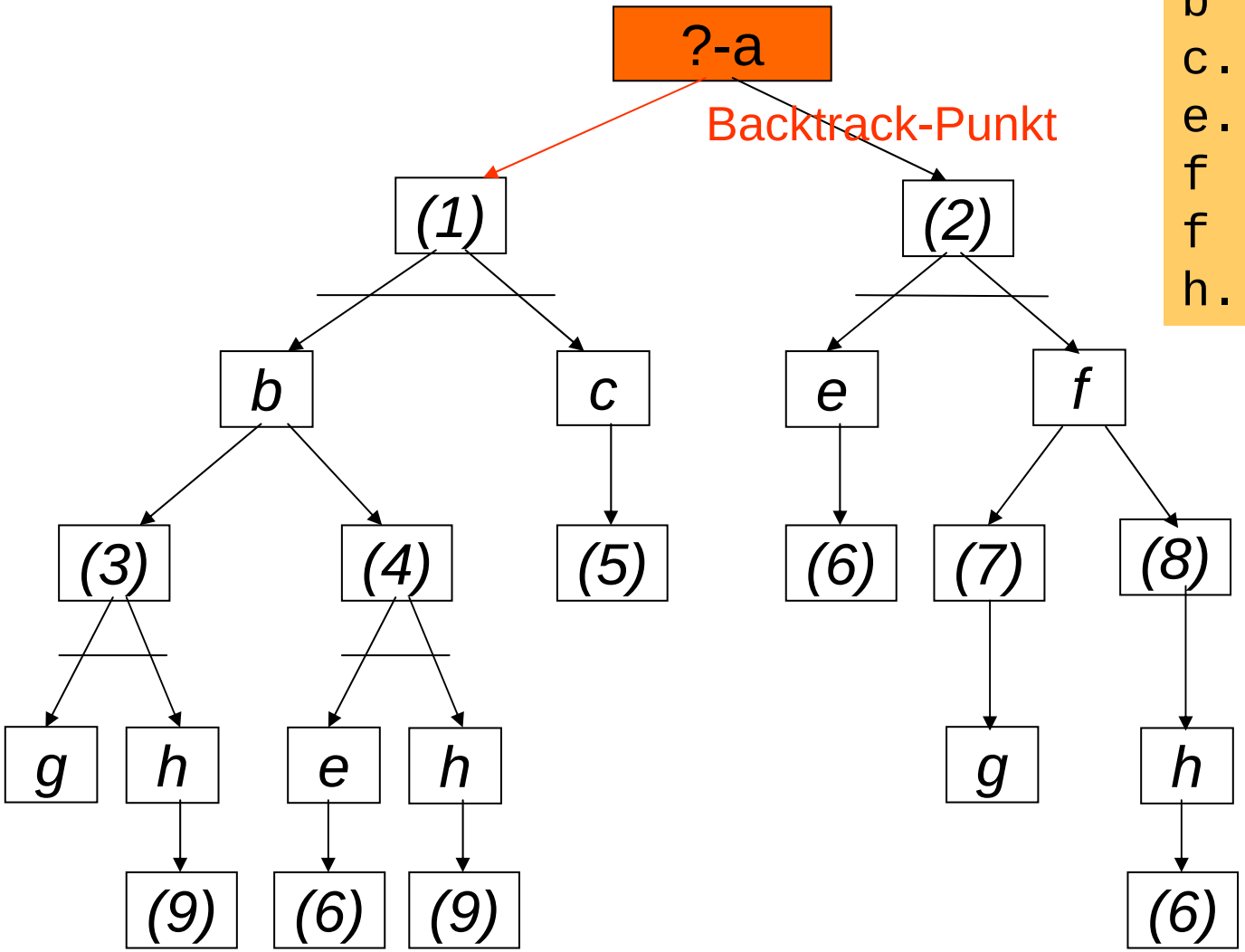
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



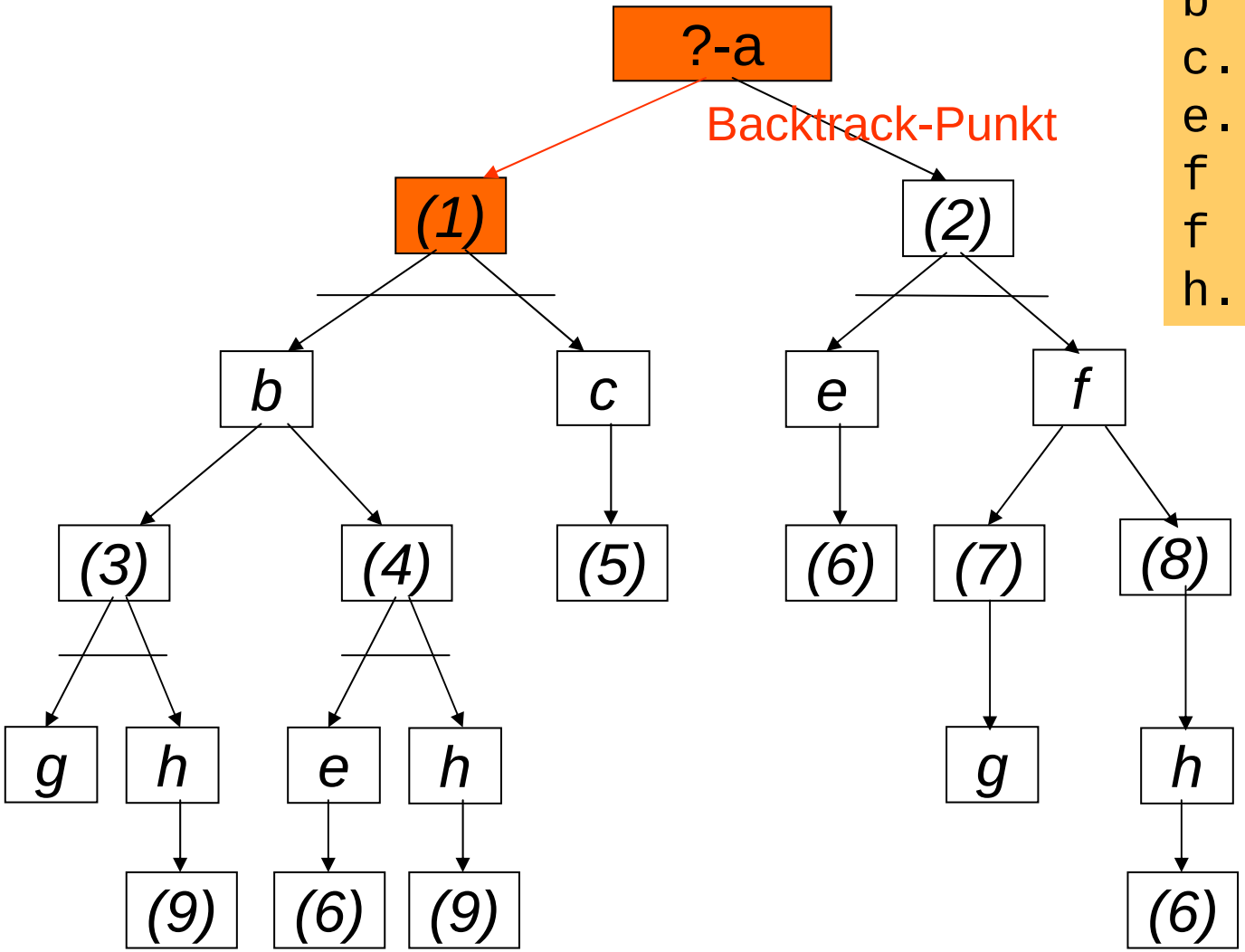
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



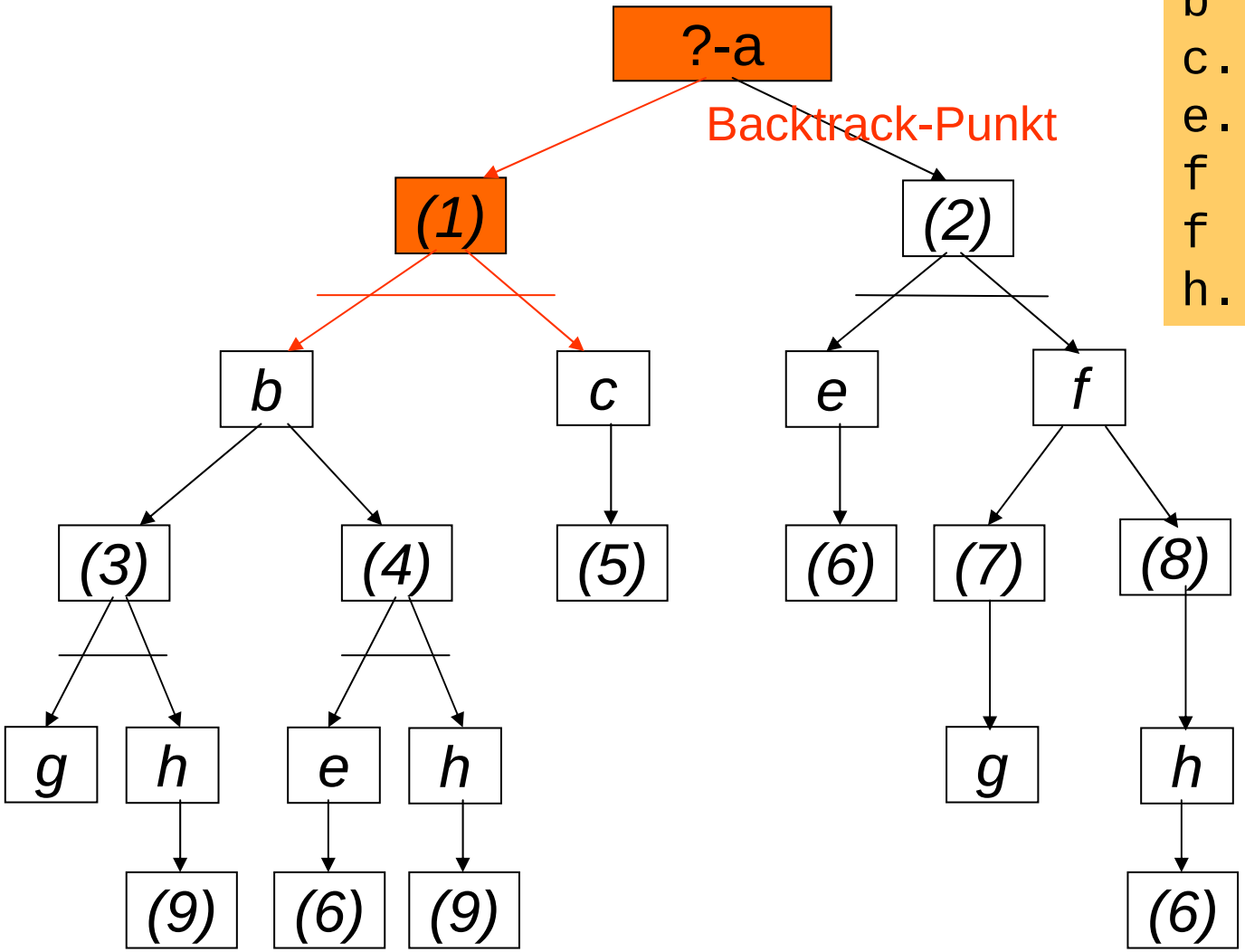
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



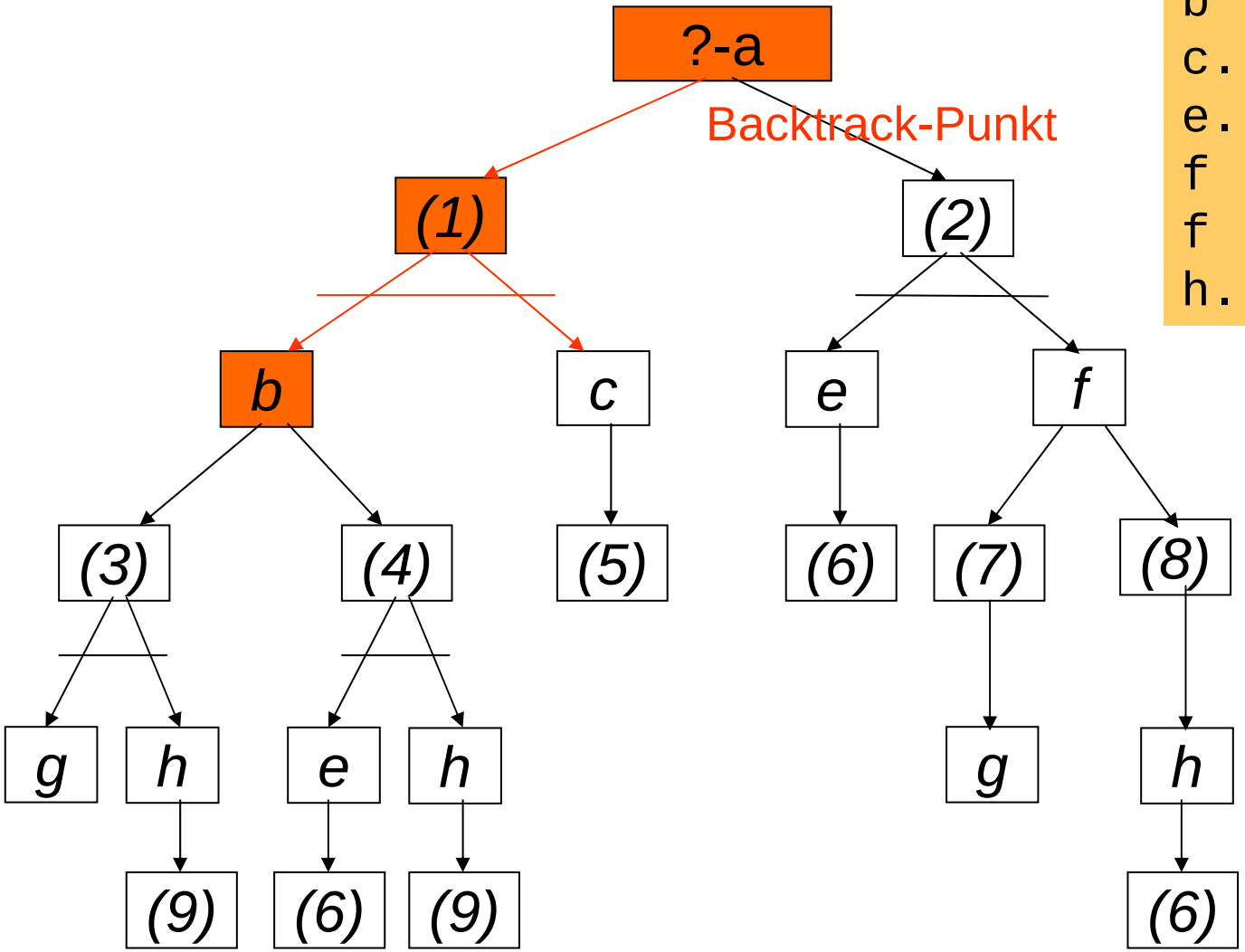
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



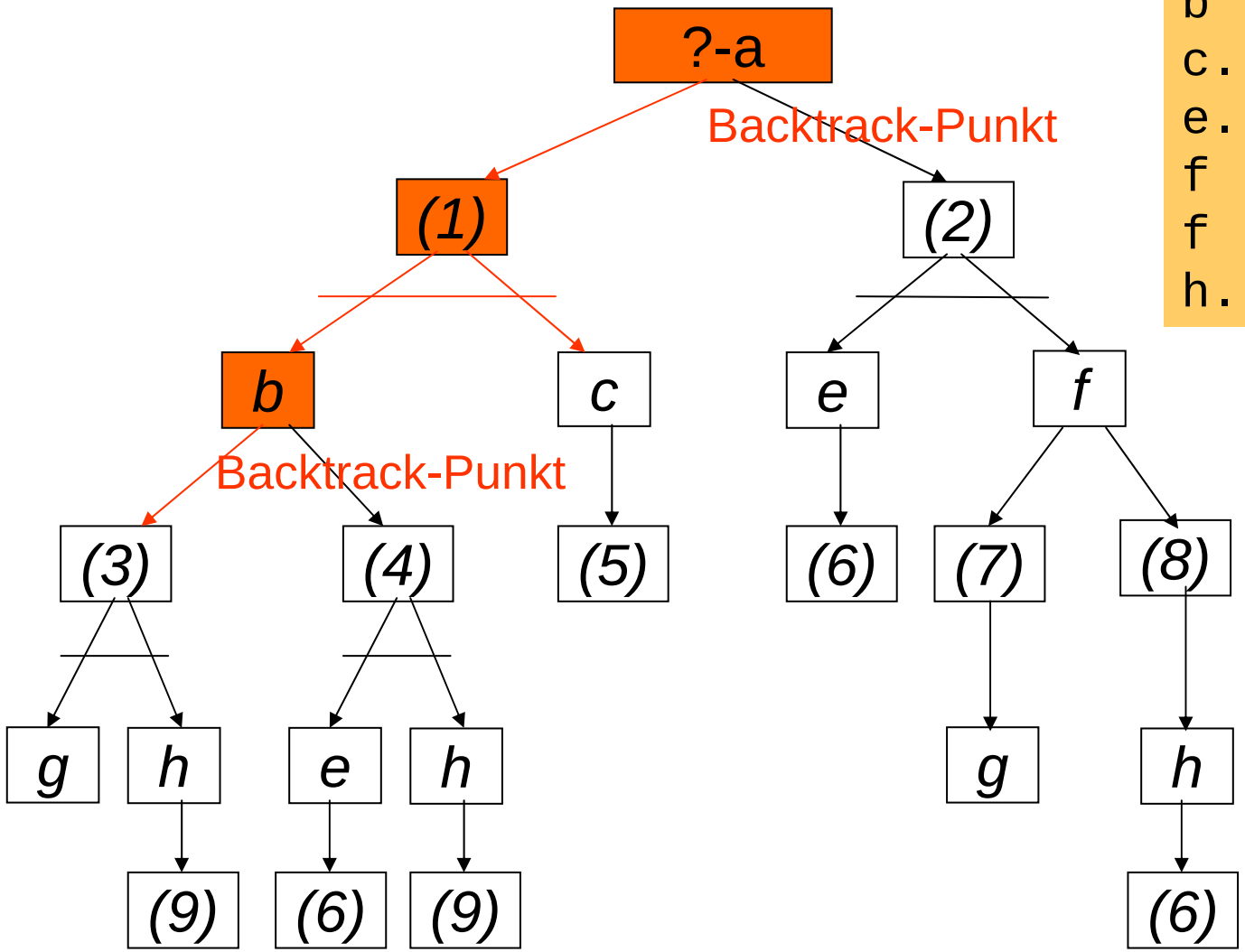
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



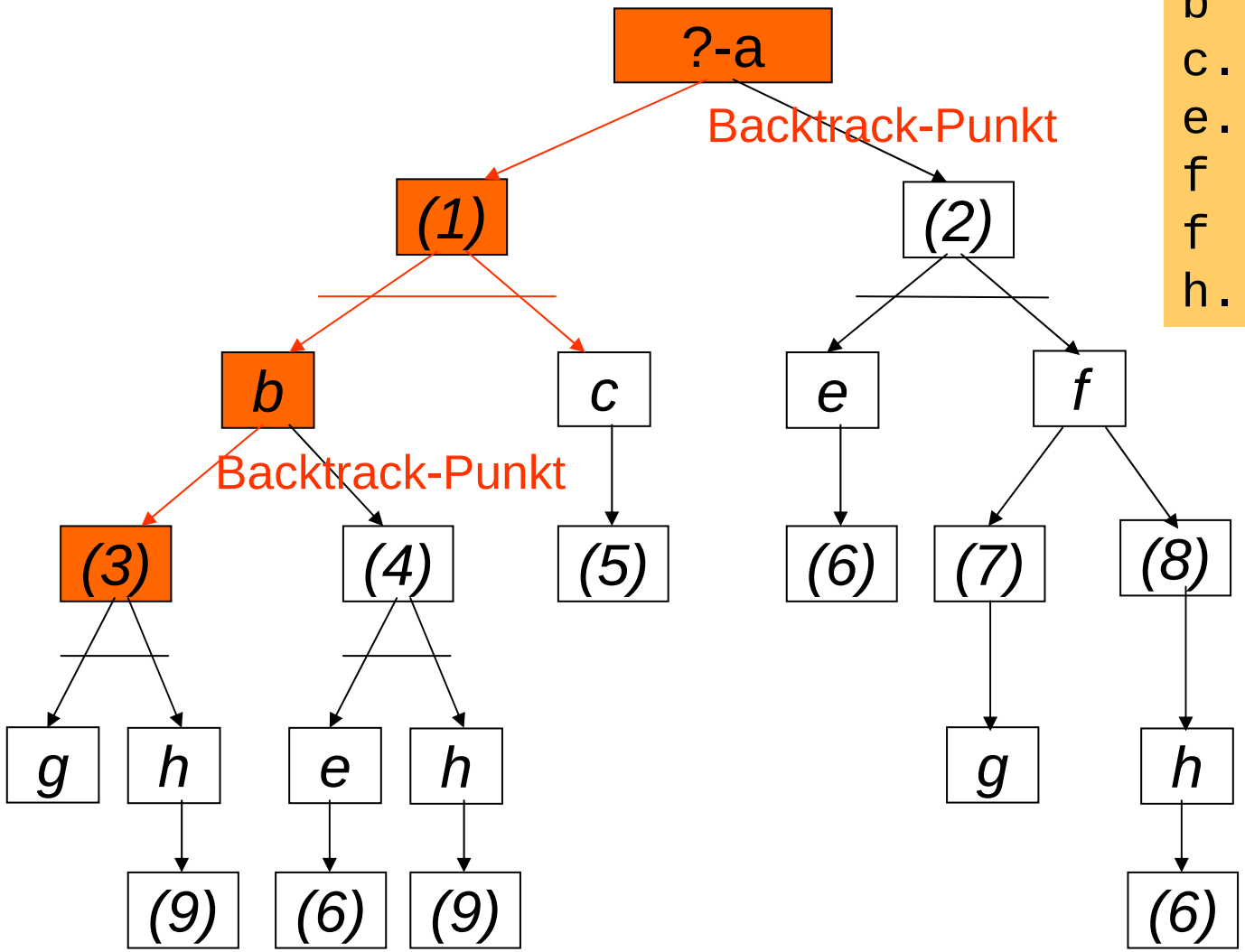
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



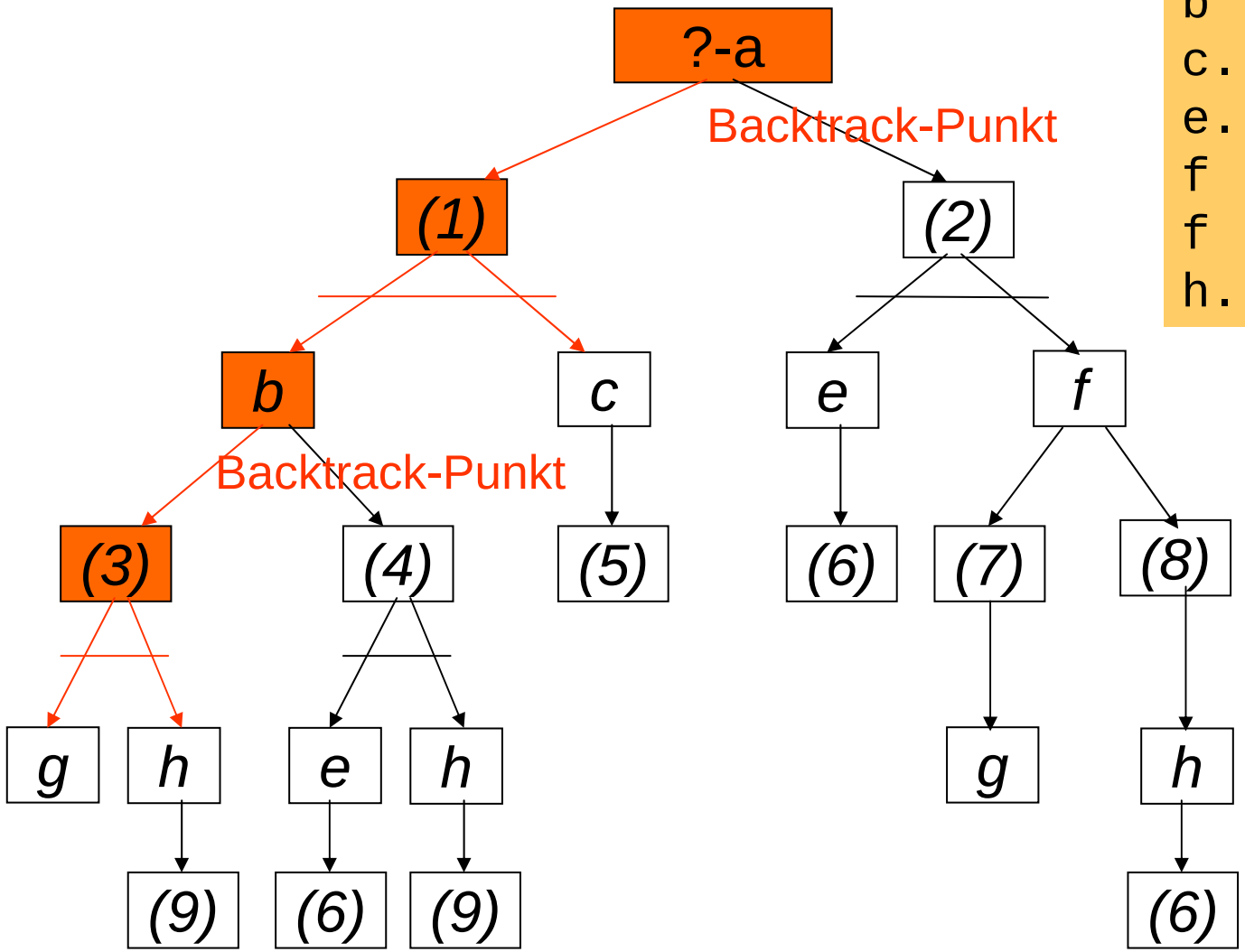
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



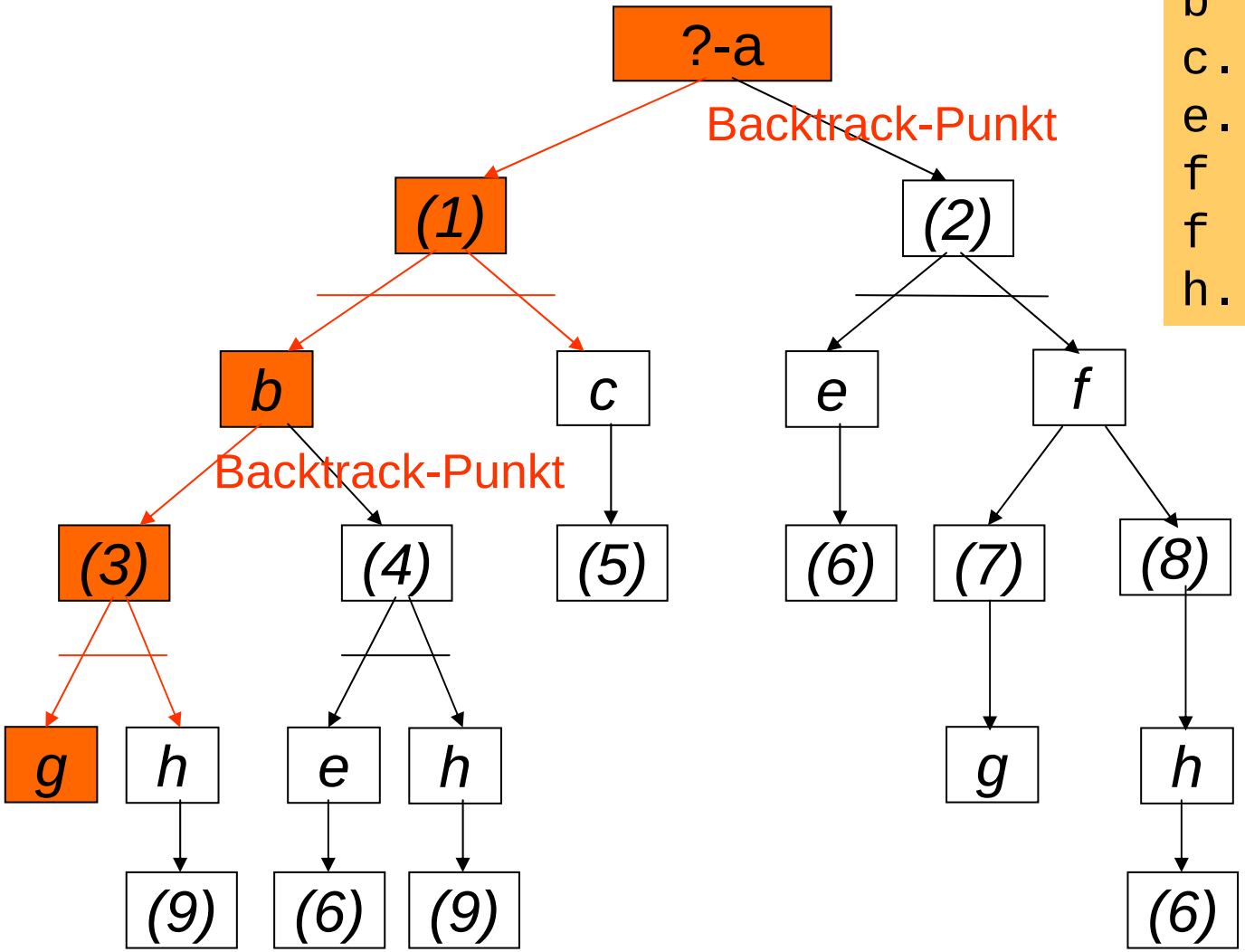
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



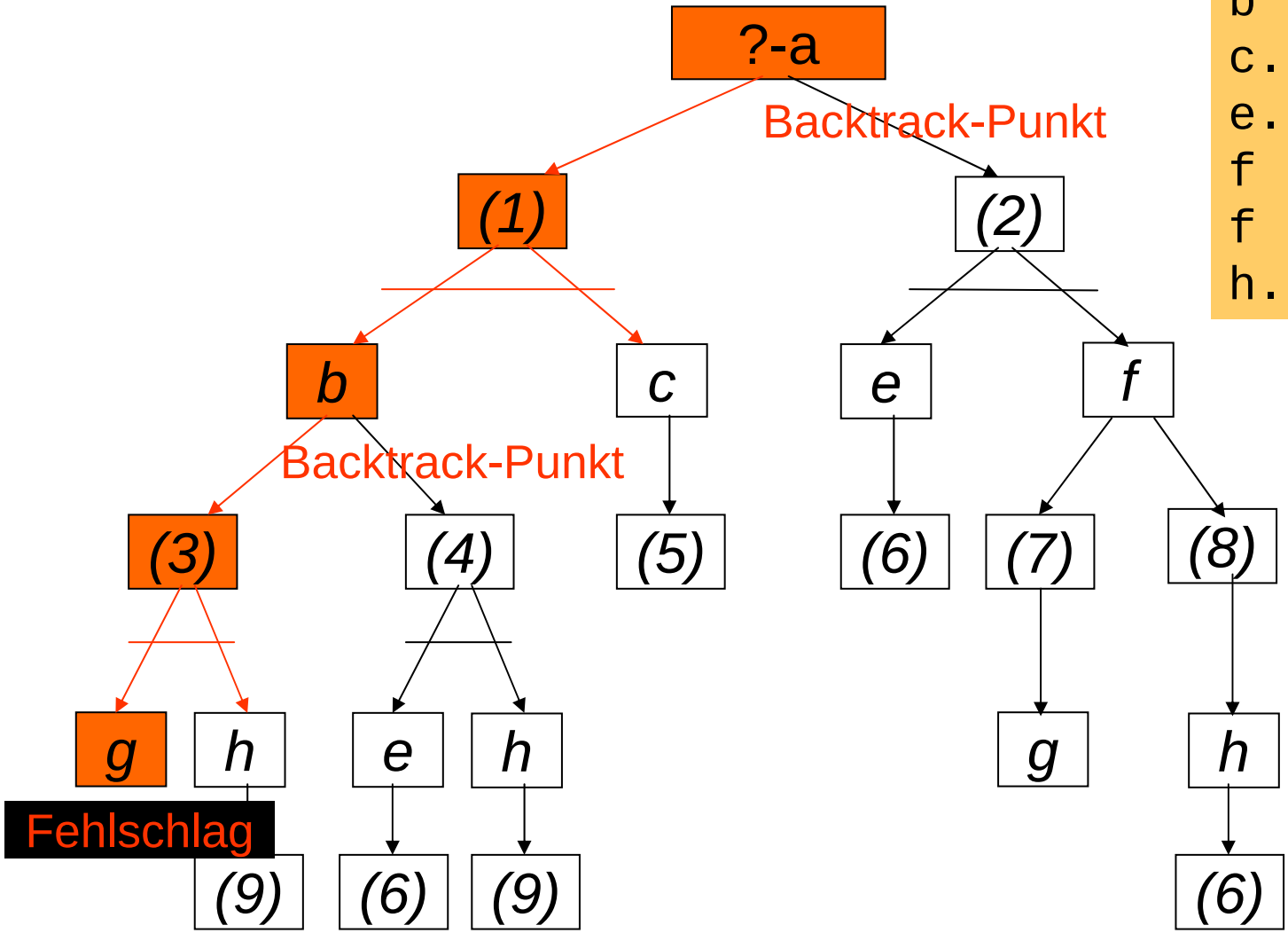
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



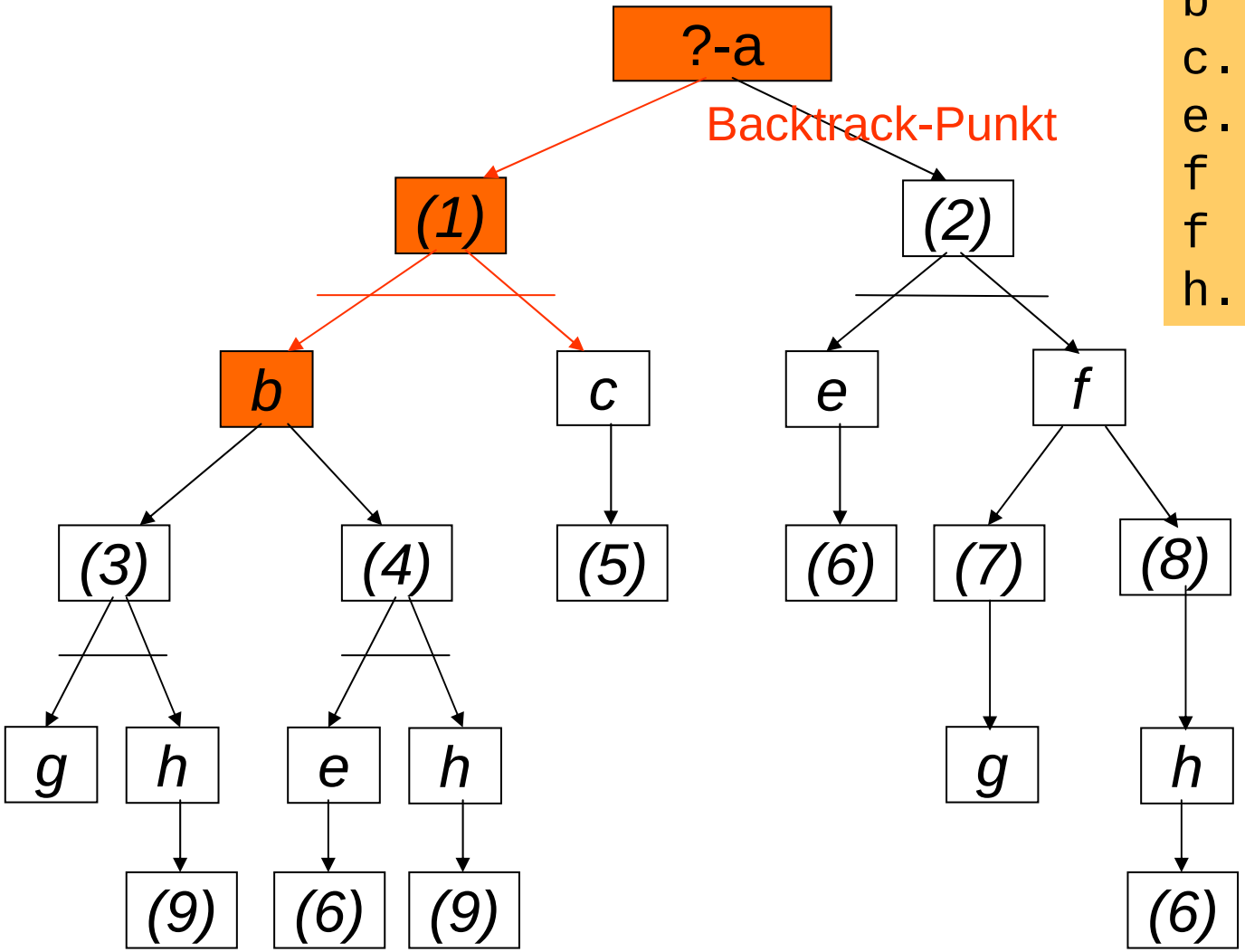
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



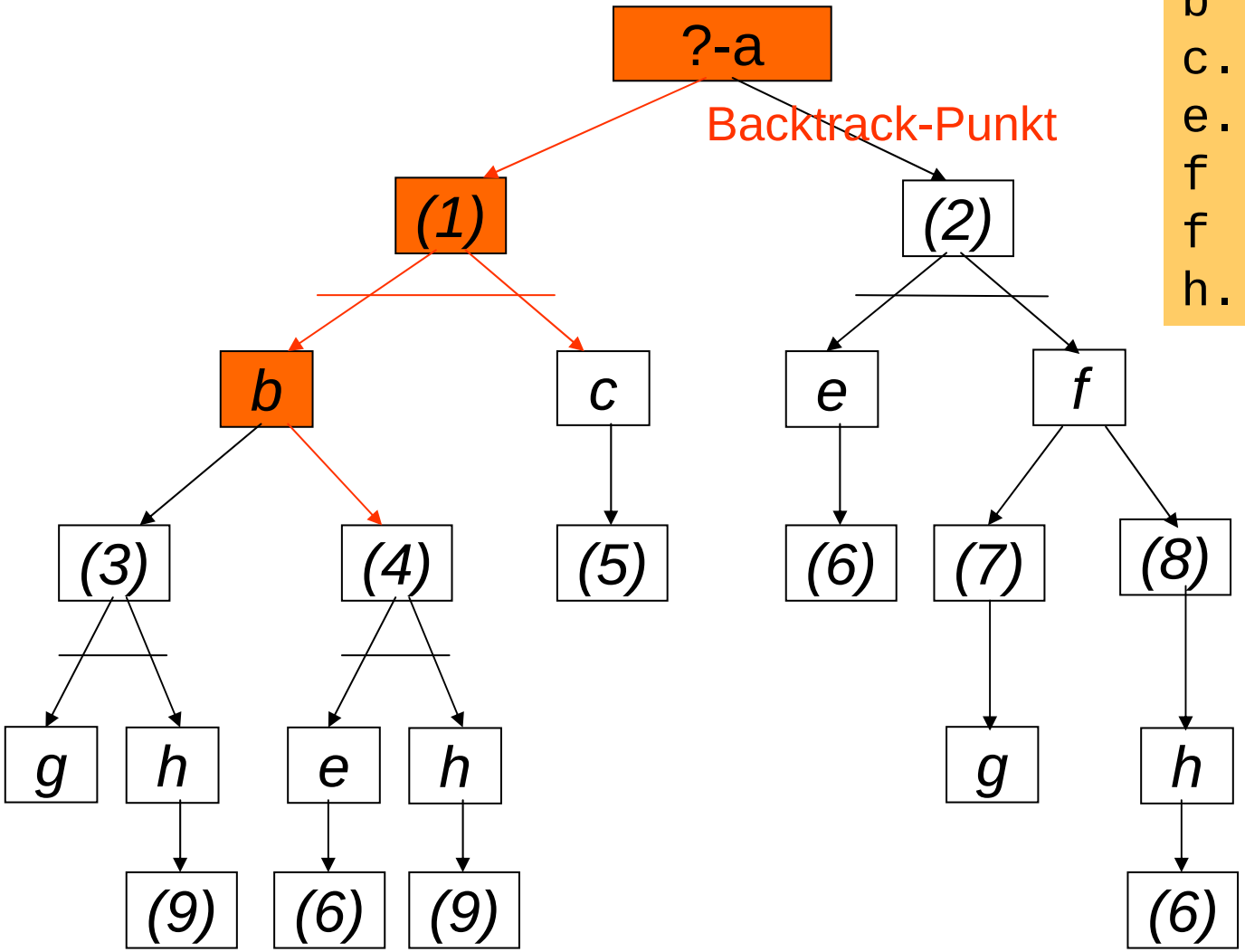
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



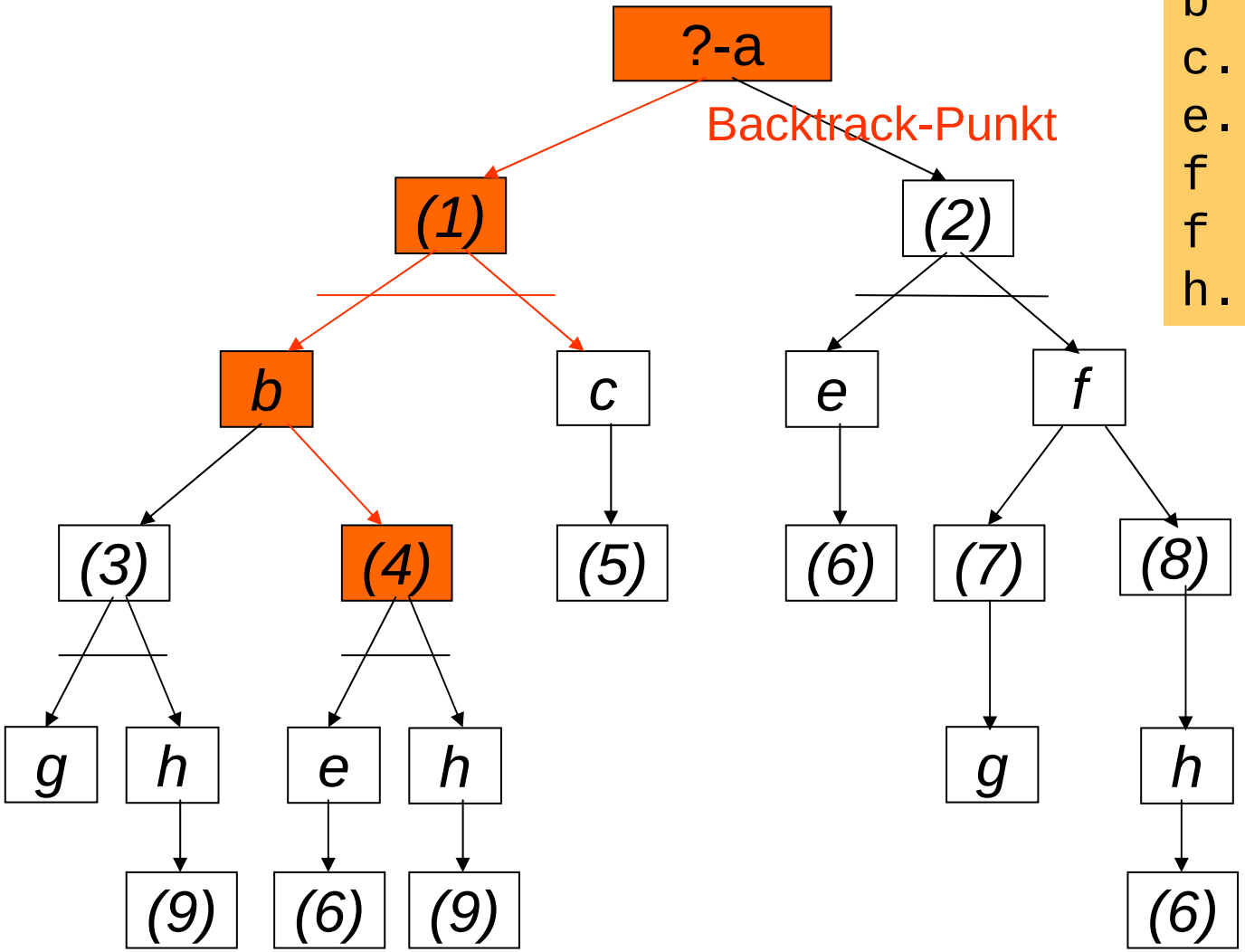
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



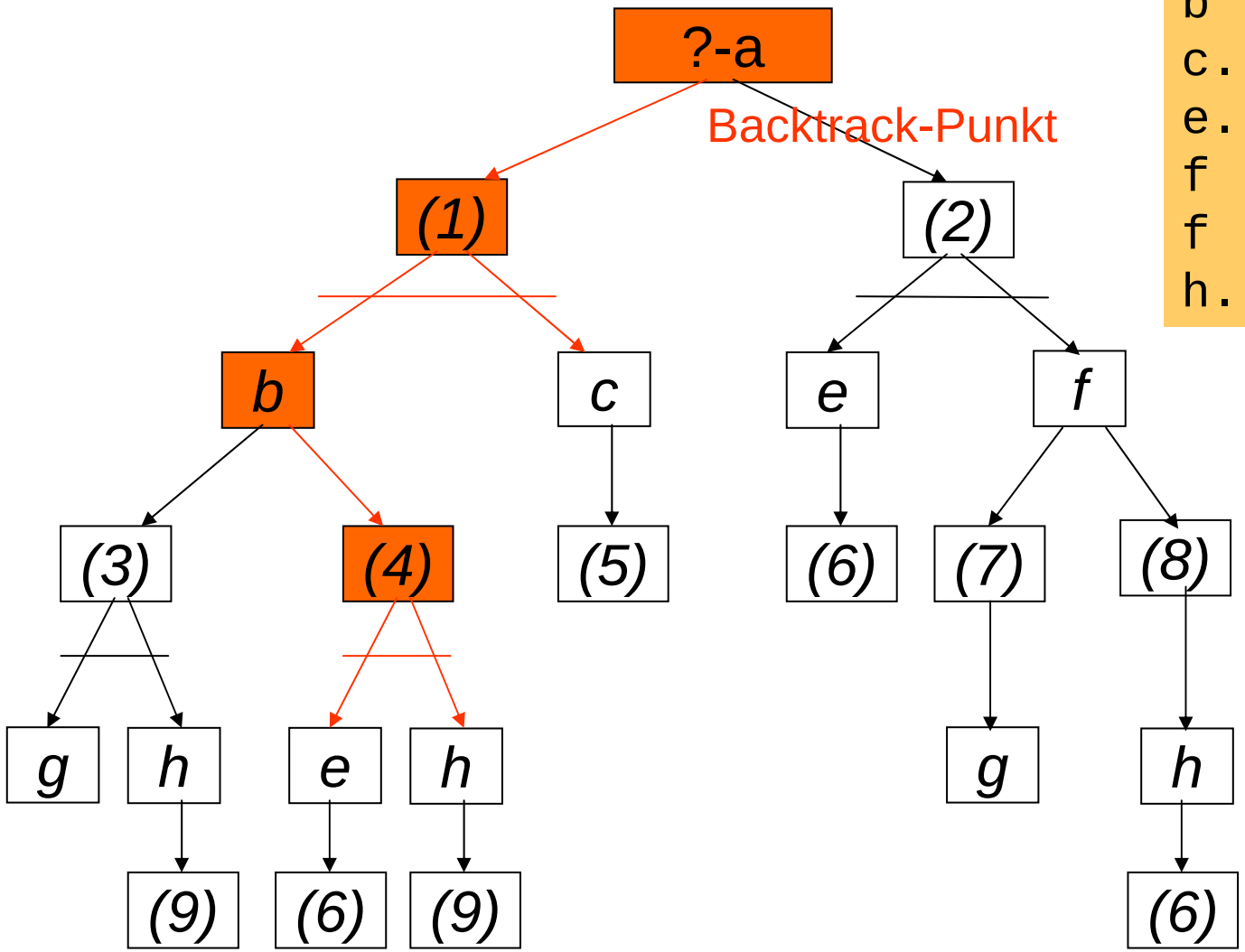
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



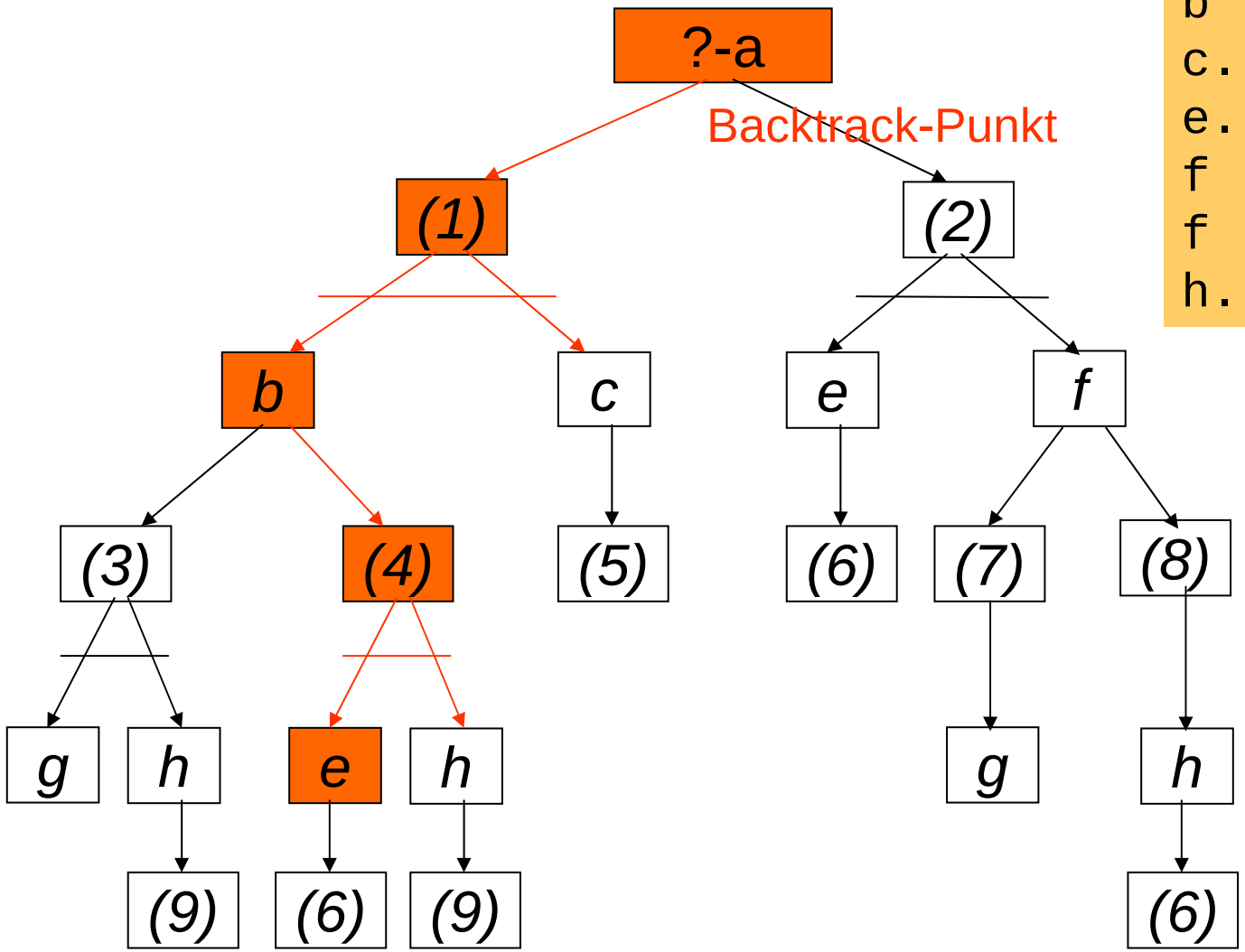
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



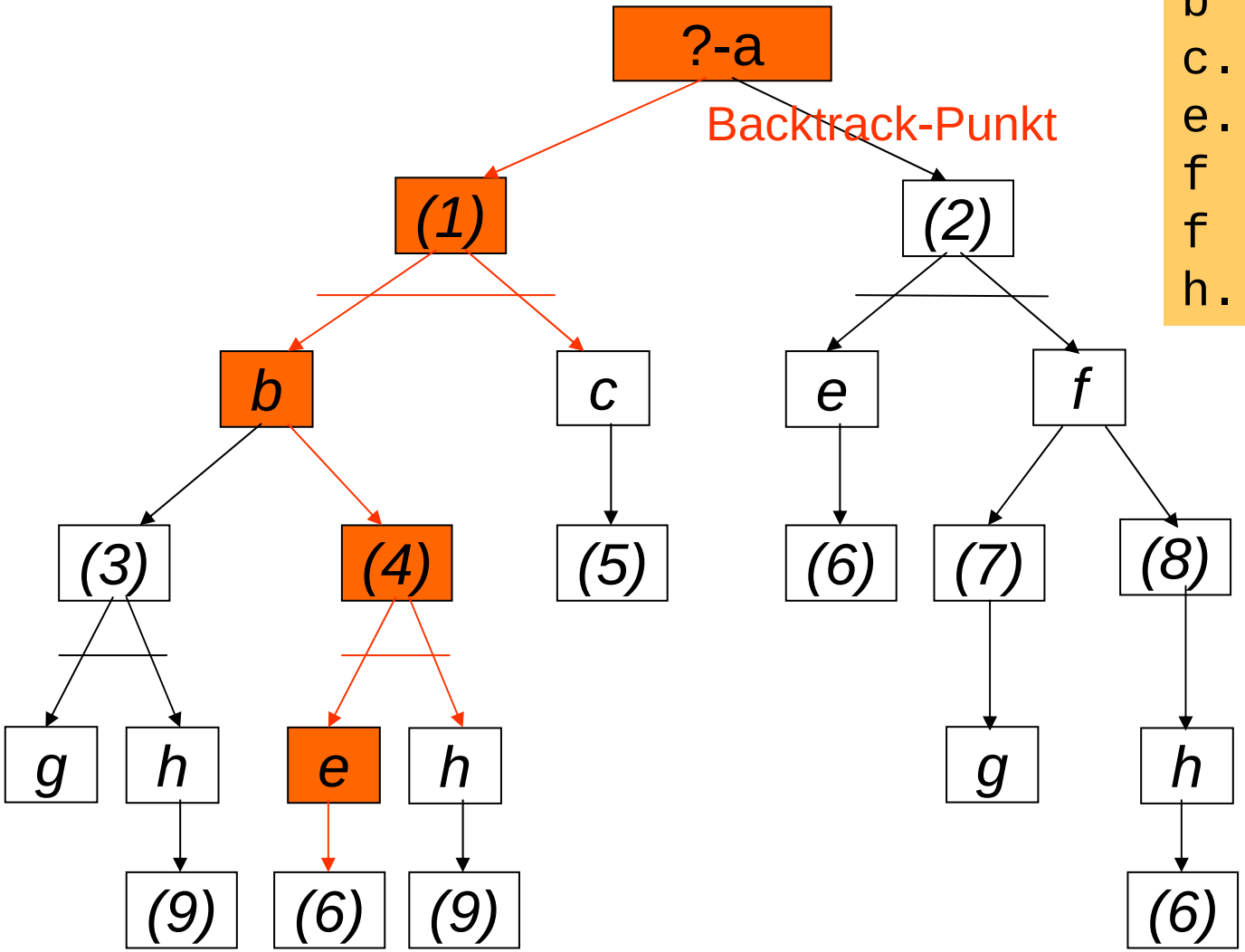
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



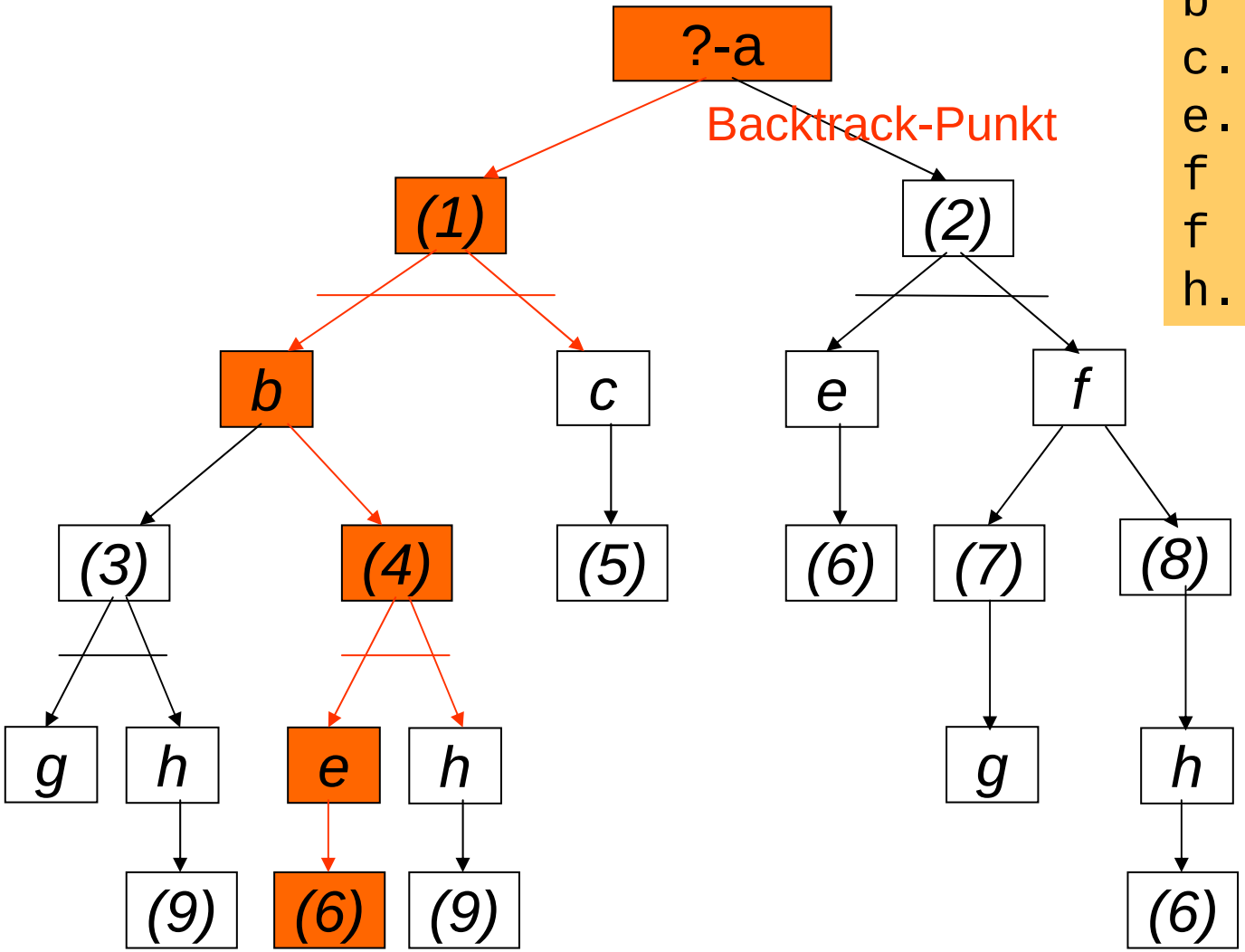
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



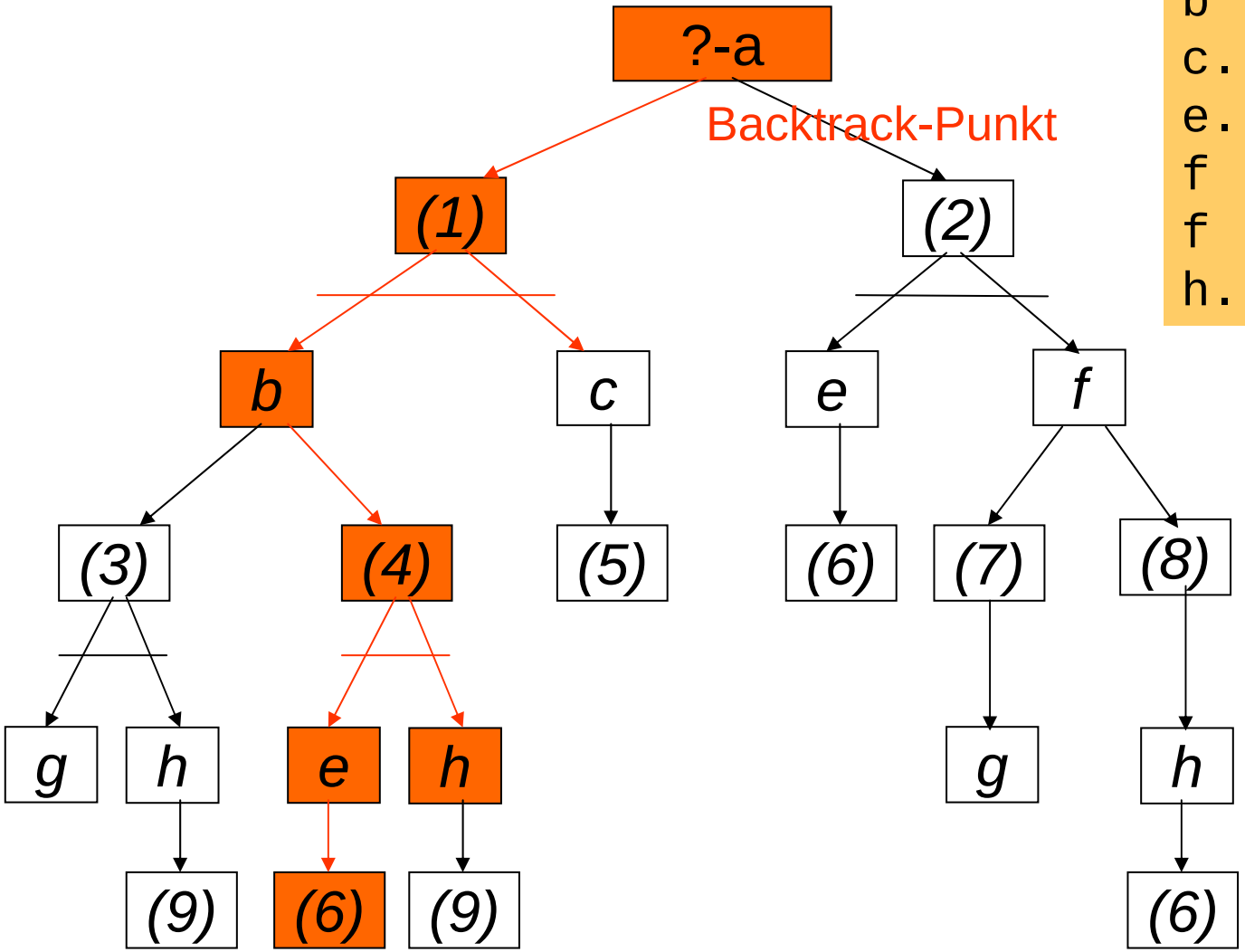
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



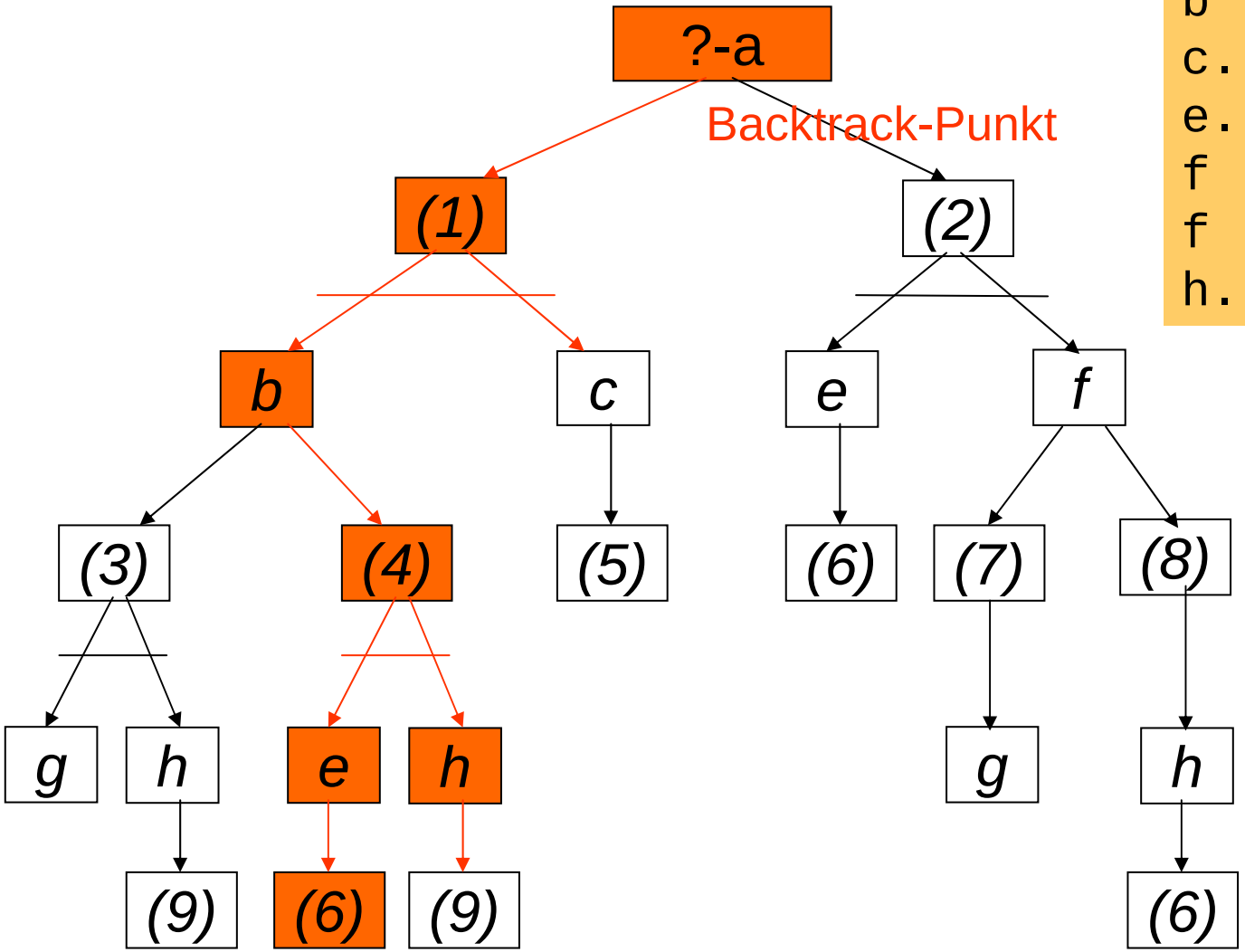
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



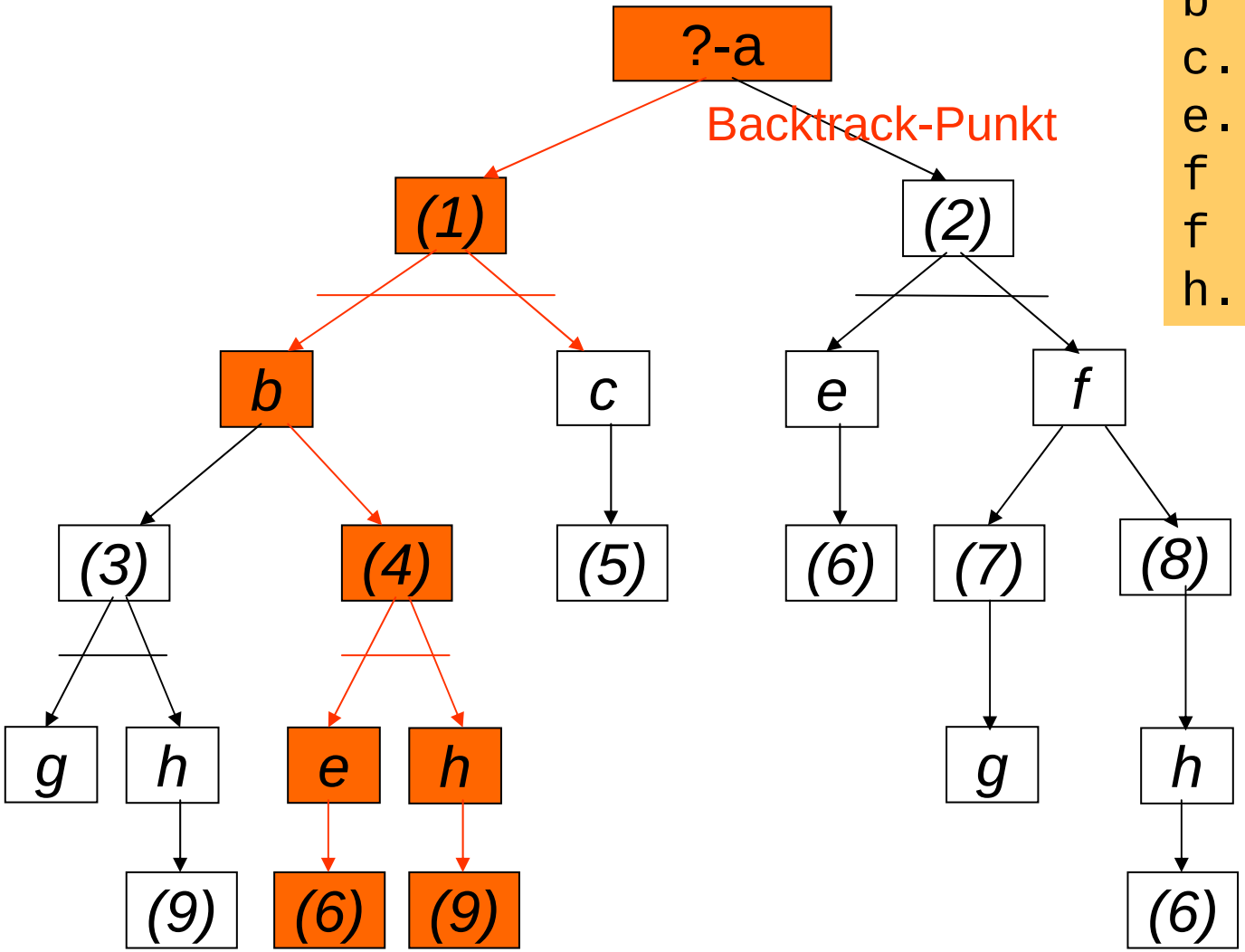
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



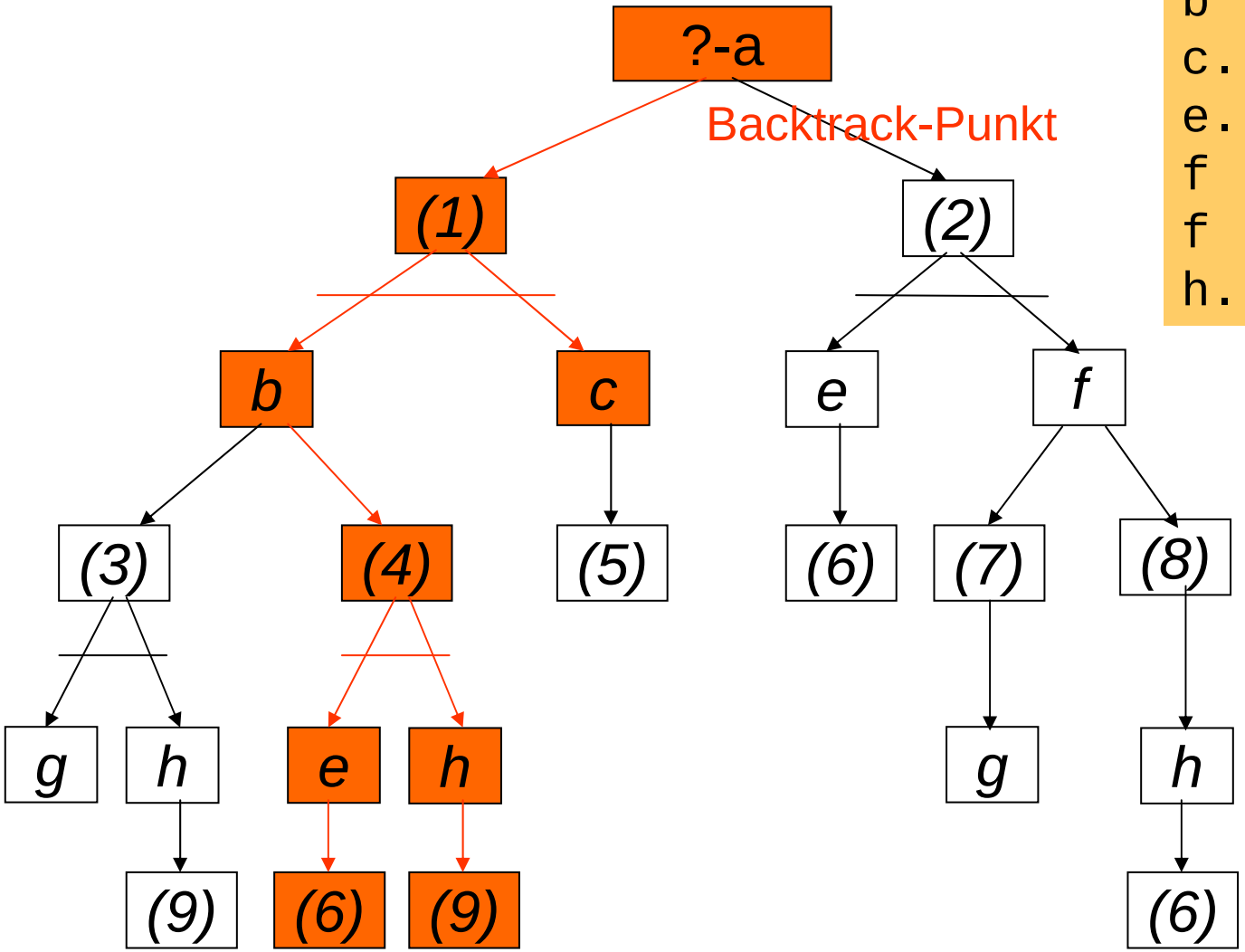
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



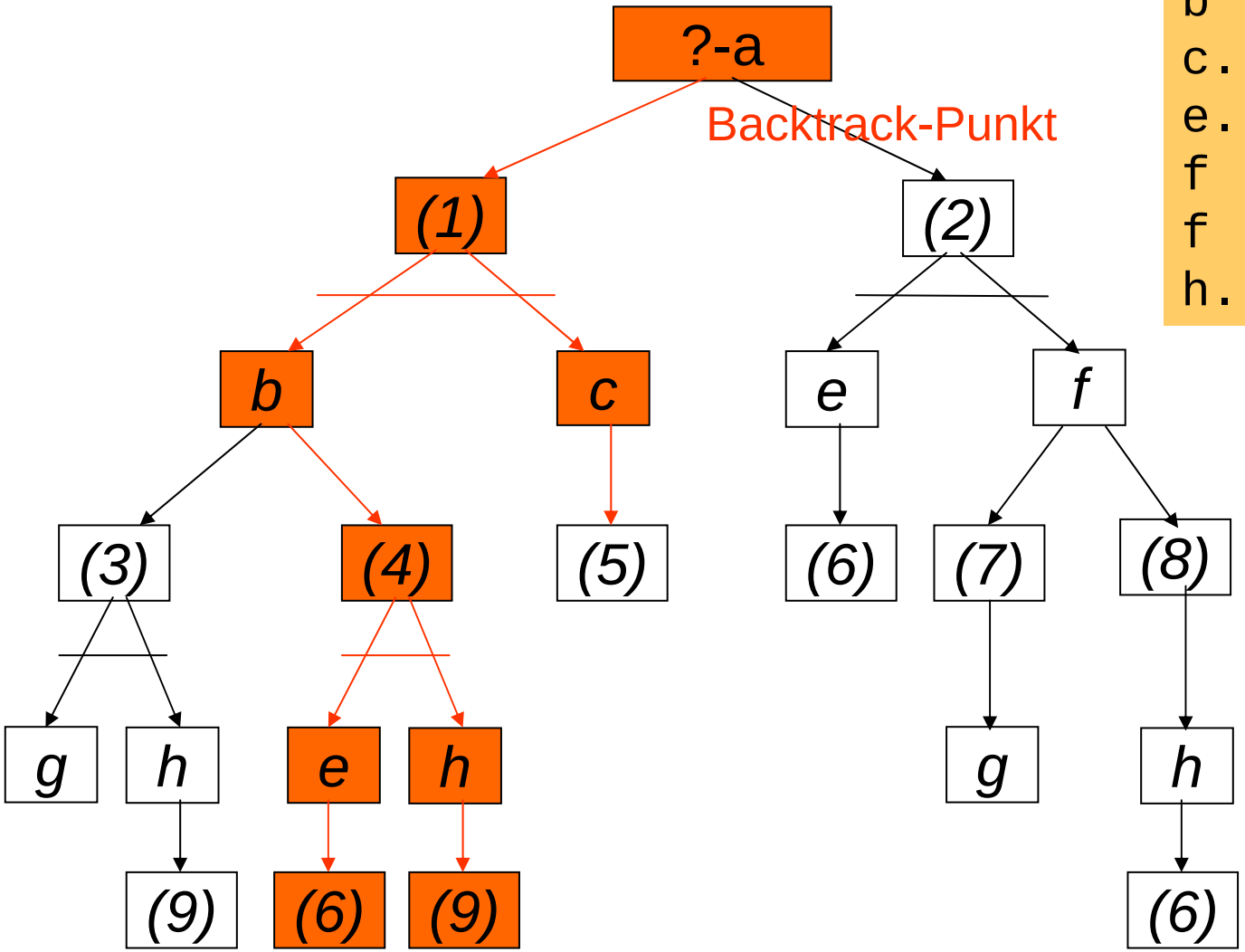
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



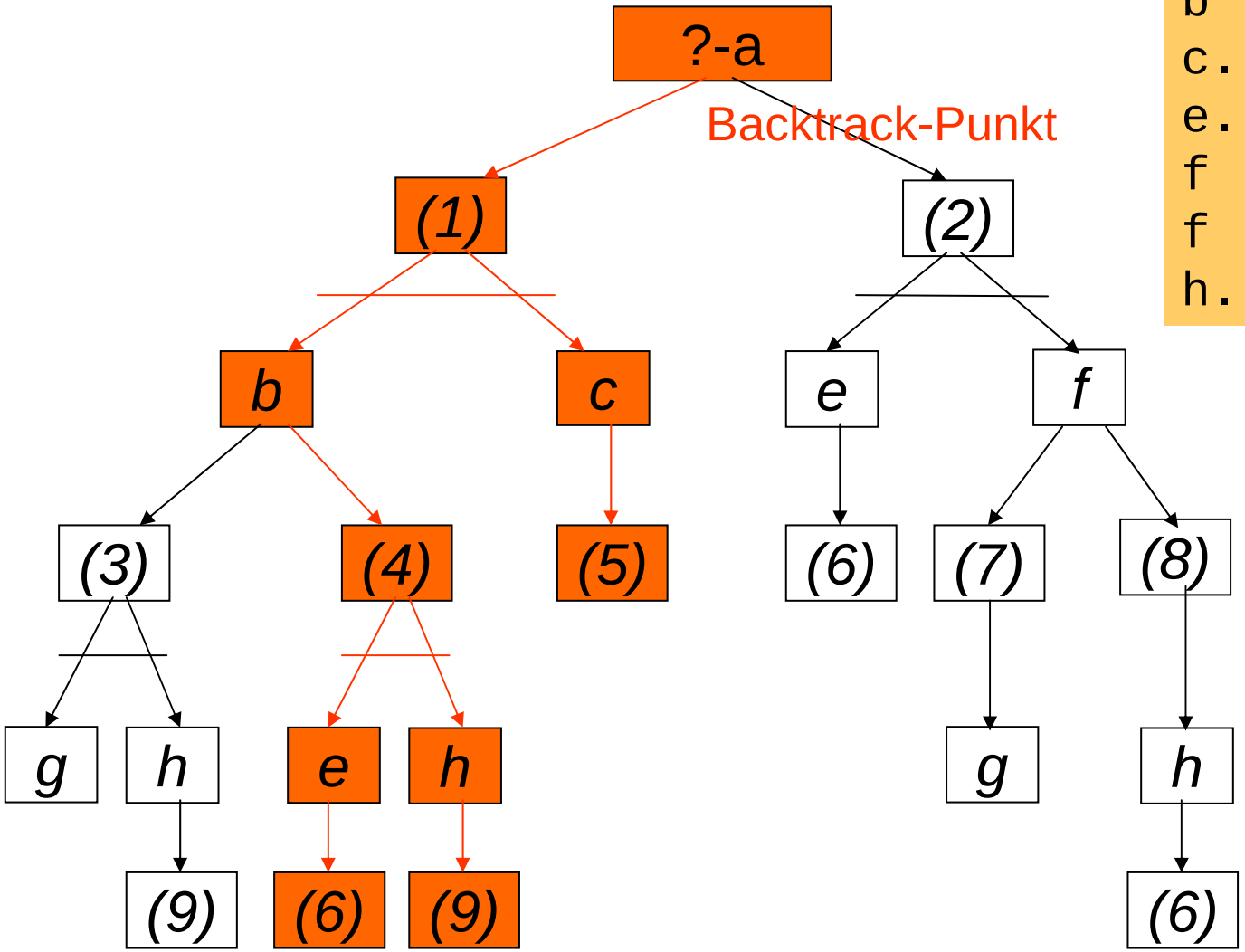
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



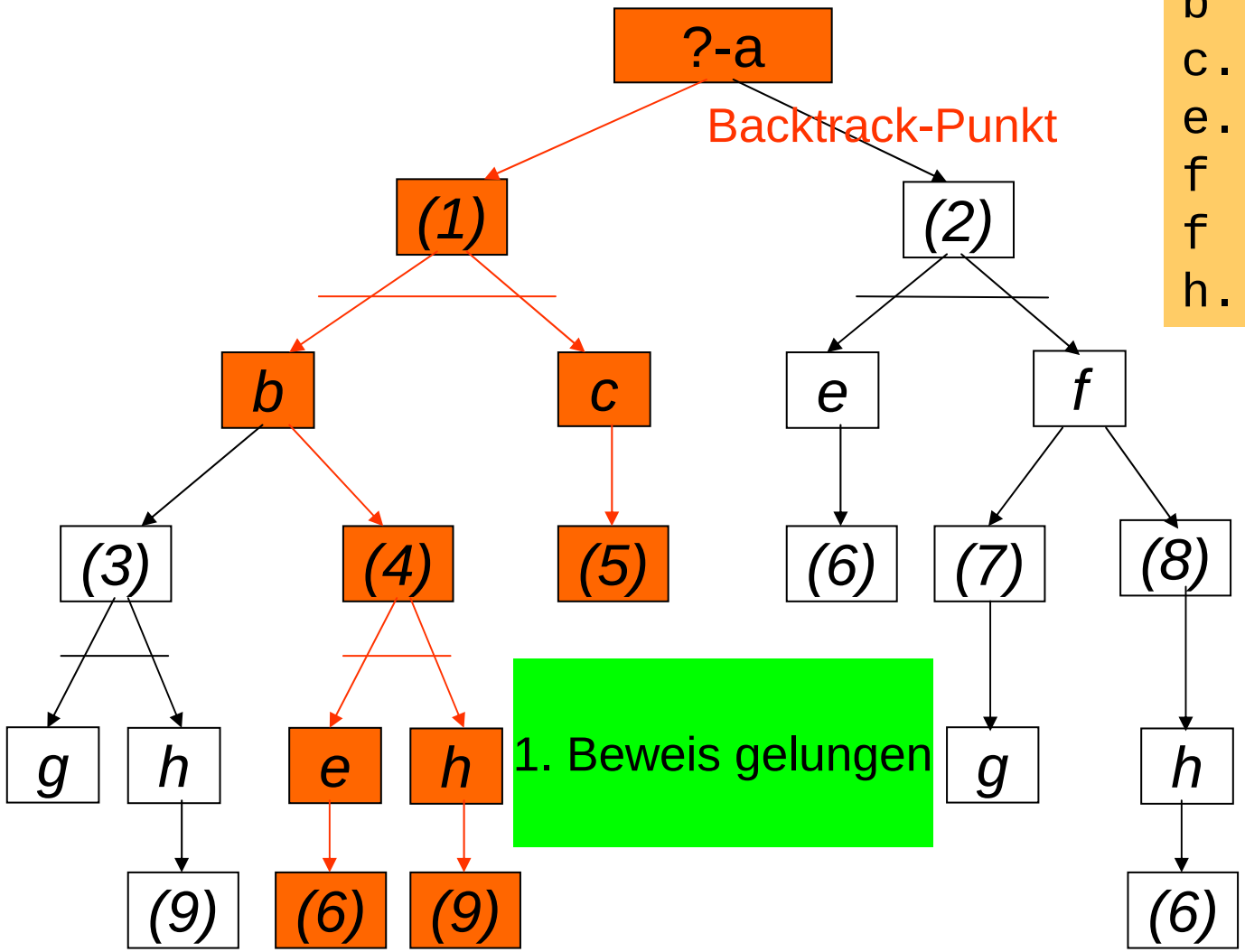
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```

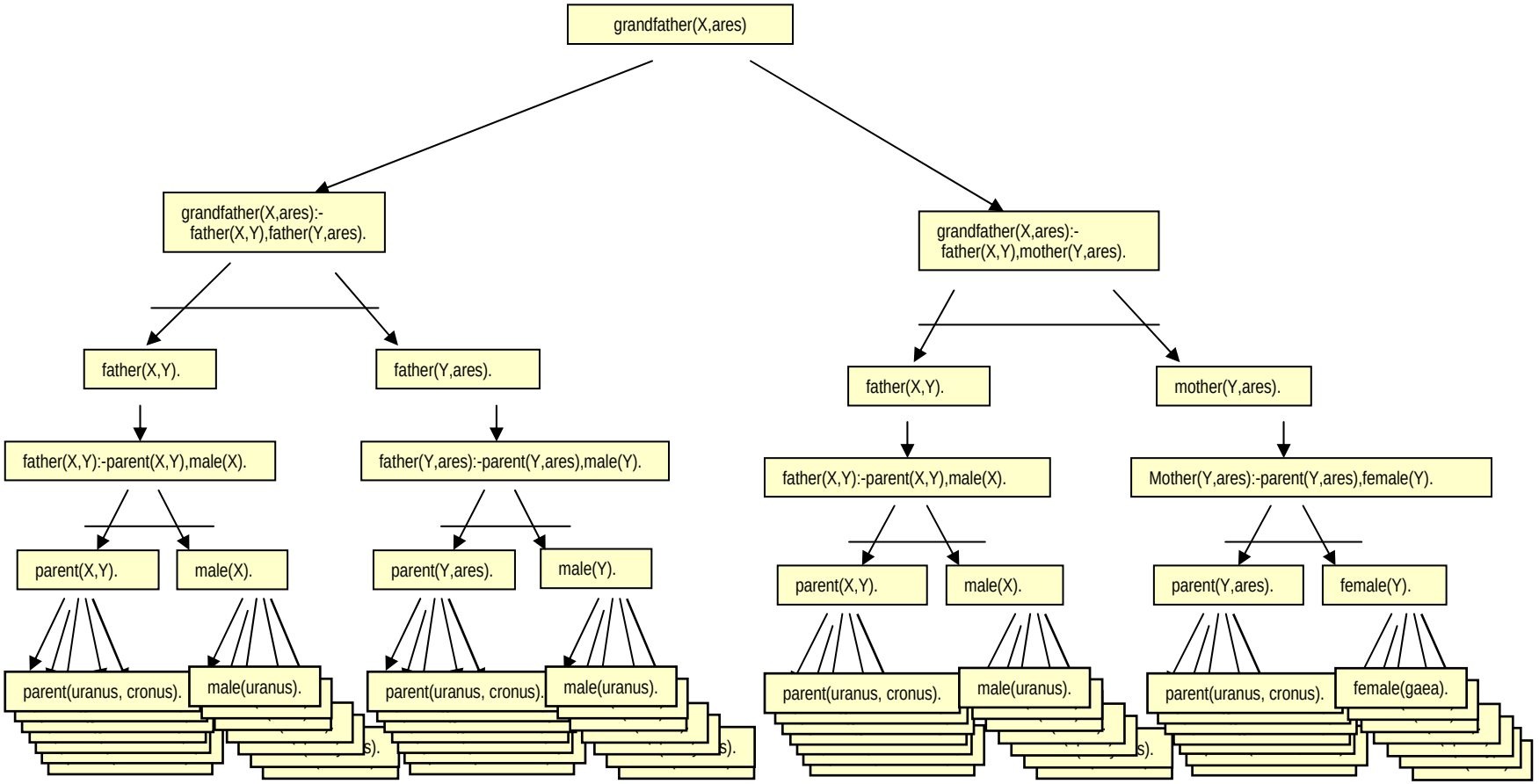


Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Algorithmus für systematische Suche



Algorithmus für systematische Suche

PROCEDURE solve(unsolved_goals: GOALLIST);

goals = $[g_1,\ldots,g_n]$ Bezeichnet Liste von goals g_i

top(goals) = g_1 Bezeichnet erstes Element

tail(goals) = $[g_2,\ldots,g_n]$ Bezeichnet Rest-Liste

NIL = [ ] Bezeichnet leere Liste

concatenate($[g_1,\ldots,g_n]$, $[g'_1,\ldots,g'_m]$) = [$g_1,\ldots,g_n$, $g'_1,\ldots,g'_m$]

Bezeichnet Verkettung von Listen

Algorithmus für systematische Suche

PROCEDURE solve(unsolved_goals: GOALLIST);

unsolved_goals — Liste ungelöster subgoals

klauseln(g) — Klauseln der Prozedur für g

```
ackermann(o,N,s(N)).
ackermann(s(M),o,V):- ackermann(M,s(o),V).
ackermann(s(M),s(N),V):- ackermann(s(M),N,V1), ackermann(M,V1,V).
```

subgoals(k) — Subgoals der Klausel k

```
ackermann(s(M),s(N),V):- ackermann(s(M),N,V1), ackermann(M,V1,V).
```

Algorithmus für systematische Suche

```
PROCEDURE solve(unsolved_goals: GOALLIST);
                    VAR k:KLAUSEL, g: GOAL;
BEGIN
  IF unsolved_goals = NIL THEN HALT(yes)
  ELSE g:= top(unsolved_goals);
    FORALL k ∈ klauseln(g) DO
                    (* Klauseln für g nacheinander rekursiv probieren*)
       solve(concatenate(subgoals(k),tail(unsolved_goals)))
                    (* subgoals der Klausel k weiter verfolgen *)
    END (*FORALL*)
  END (*IF*)
END solve;
```

Warum?

Backtrack-Punkte

Reihenfolge: oben vor unten

Reihenfolge: links vor rechts

Aufruf mit solve([goal]); HALT(no).

Algorithmus für systematische Suche

Rekursive Prozedur-Aufrufe mit Subgoal-Listen.

Bei leerer Subgoalliste:

- Resultat „yes“
- Abbruch der Prozedur-Kette.

Nach vollständiger Abarbeitung einer Aufruf-Kette Rückkehr zur jüngsten Möglichkeit gemäß FORALL ... (Backtracking).

Wenn alle (FORALL-)Varianten erfolglos versucht:

- Resultat „no“
- Prozedur-Ketten vollständig abgearbeitet.

Algorithmus für systematische Suche

Algorithmus verwendet eigentlich zwei Listen
(gemäß LIFO-Prinzip: Keller/stacks)

- Liste unsolved_goals (offene Teilziele)

- Liste der offenen Prozedur-Aufrufe (Prozedurkeller) mit Alternativen für 'Backtracking'

Betrachtung als verschachtelte Listen

$\mathcal{L} = [\ L_1,...,L_m\ ]$

$= [\ [g_{11}, ..., g_{1n}]\ , ..., [g_{i1}, ..., g_{i,ni}], ...\ , [g_{m1}, ..., g_{m,nm}]\ ]$

Algorithmus für systematische Suche

(0) (Start) $\mathcal{L}$: = [[Ausgangsproblem(e)]] .

(1) Falls $\mathcal{L}$ = [[ ], ...,[ ]] : EXIT(yes) .

(2) Sei L_i erste nicht-leere Subgoal-Liste aus $\mathcal{L}$ = [L_1,...,L_m] .

Sei g_{i1} erstes Element aus L_i = [g_{i1}, ..., $g_{i,ni}$] :

- g_{i1} aus L_i entfernen: L'_i := [g_{i2}, ..., $g_{i,ni}$] .
- Falls keine Klauseln für g_{i1} existieren: weiter bei **(4)** .

(3) Sei k die nächste abzuarbeitende Klausel für g_{i1} .

Falls k Fakt: weiter bei **(1)** .

Falls k Regel: g_{i1} :- g_1, ..., g_n :

$\mathcal{L}$:= [[g_1, ..., g_n], L_1,.., L'_i,..., L_m] .

Falls weitere Klauseln für g_{i1} existieren:

Backtrack-Punkt setzen. Weiter bei **(2) .**

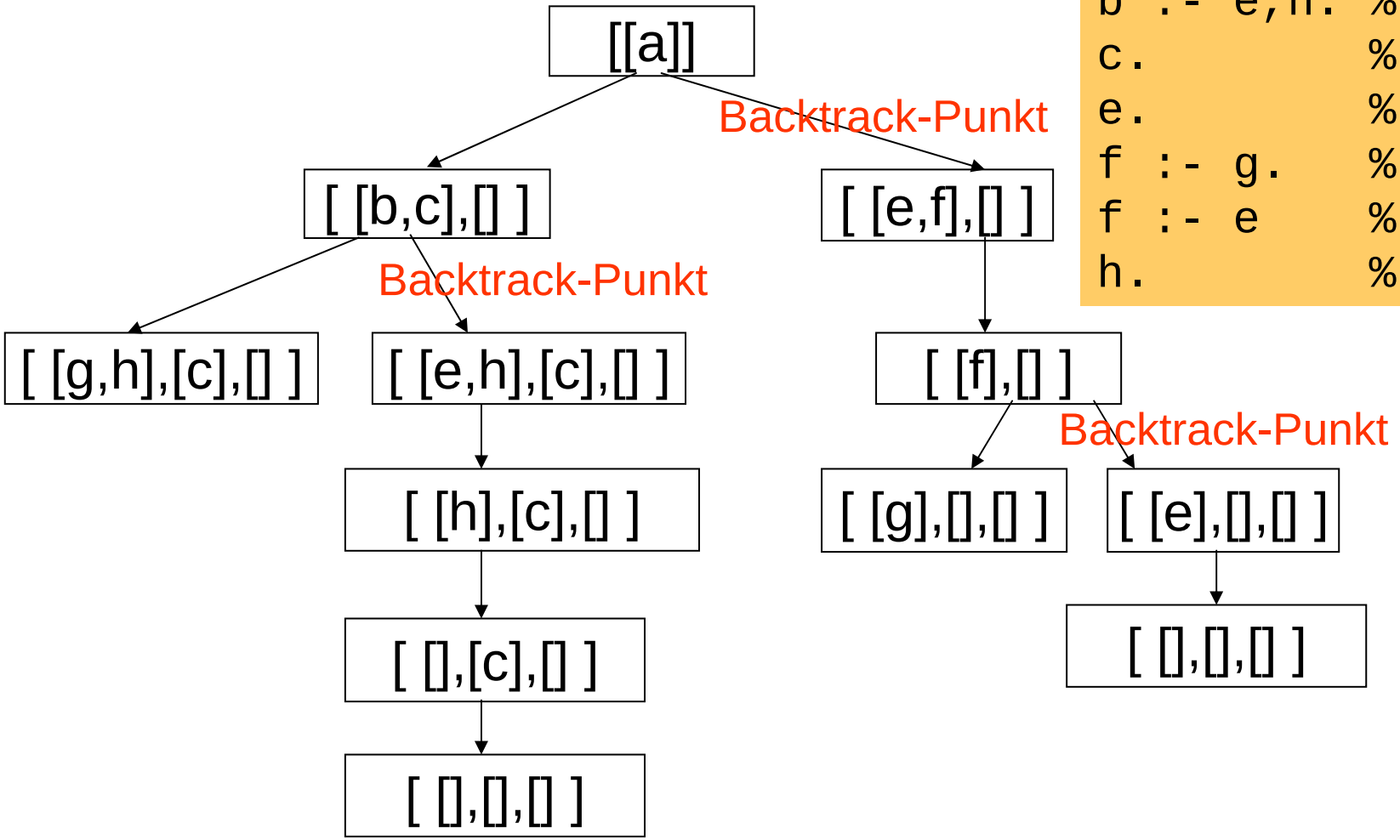
(4) Backtracking: Rücksetzen zum jüngsten *Backtrack-Punkt* :

$\mathcal{L}$ zurücksetzen auf Stand vor *Backtrack-Punkt*, weiter bei **(3)** .

Falls kein *Backtrack-Punkt* existiert: $\mathcal{L}$ =[ ] , EXIT(no).

Algorithmus für systematische Suche

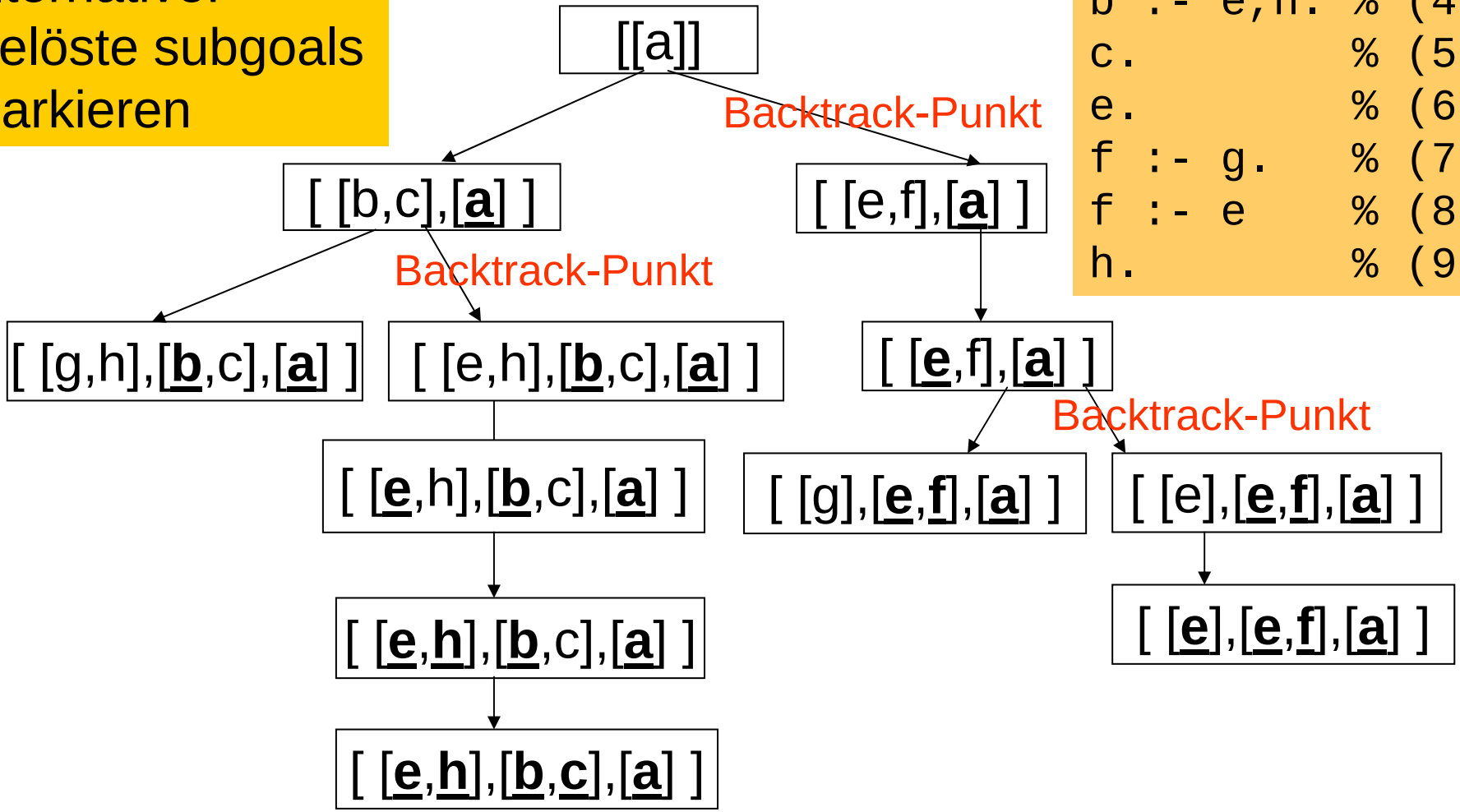
```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Algorithmus für systematische Suche

Alternative:
Gelöste subgoals markieren

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Effizientere Implementation

Bekannt sind:

- Reihenfolge von Klauseln in Prozeduren
- Reihenfolge von subgoals in Klauseln

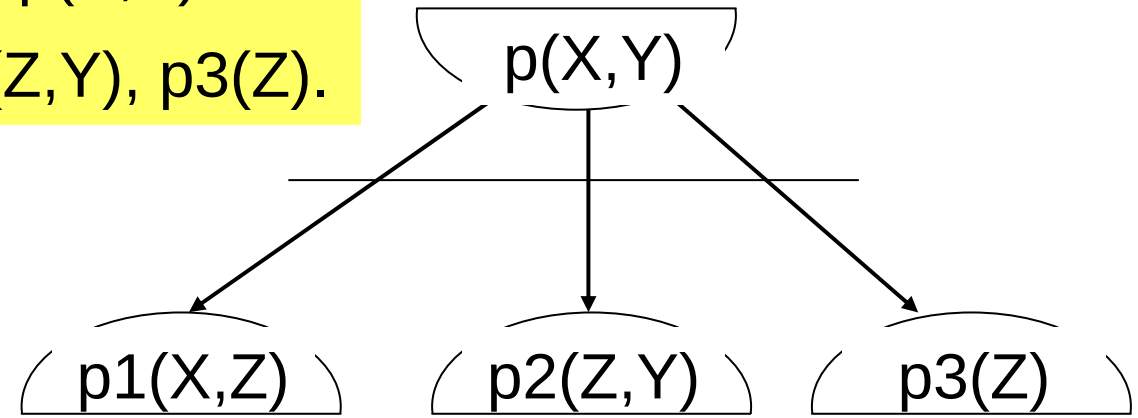
Zur Laufzeit nicht gesamte Listen speichern,
sondern nur Referenz auf jeweils nächsten Eintrag

g_i statt $[g_i, ..., g_n]$,

Modell für Relationen (mit Variablen)

Problemzerlegung für p(X,Y):

p(X,Y) :- p1(X,Z), p2(Z,Y), p3(Z).

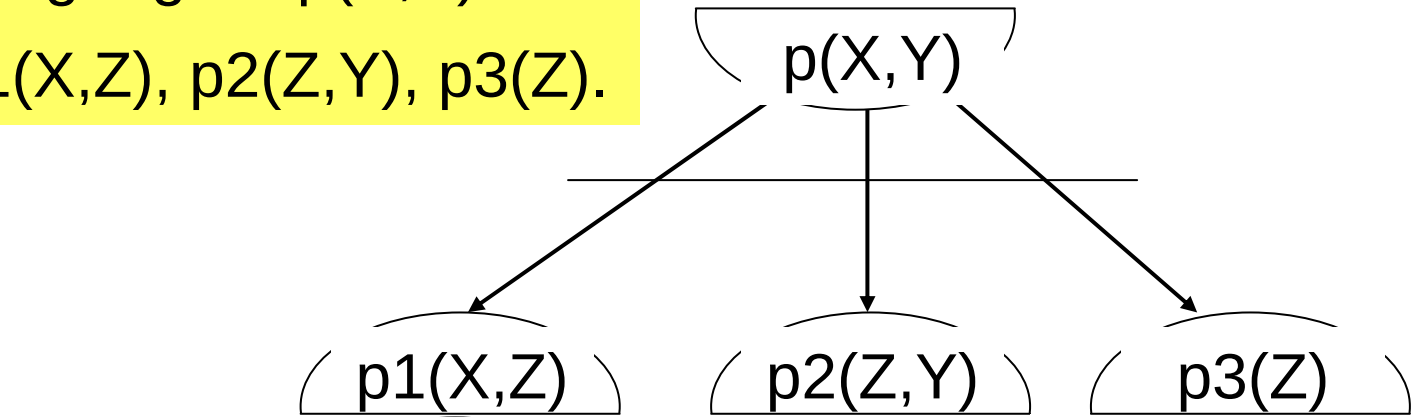


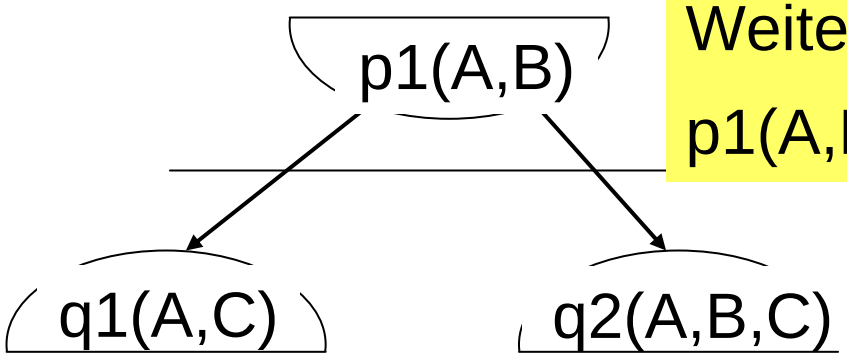
„Ferguson-Diagramm“

Modell für Relationen (mit Variablen)

Problemzerlegung für p(X,Y):

p(X,Y) :- p1(X,Z), p2(Z,Y), p3(Z).





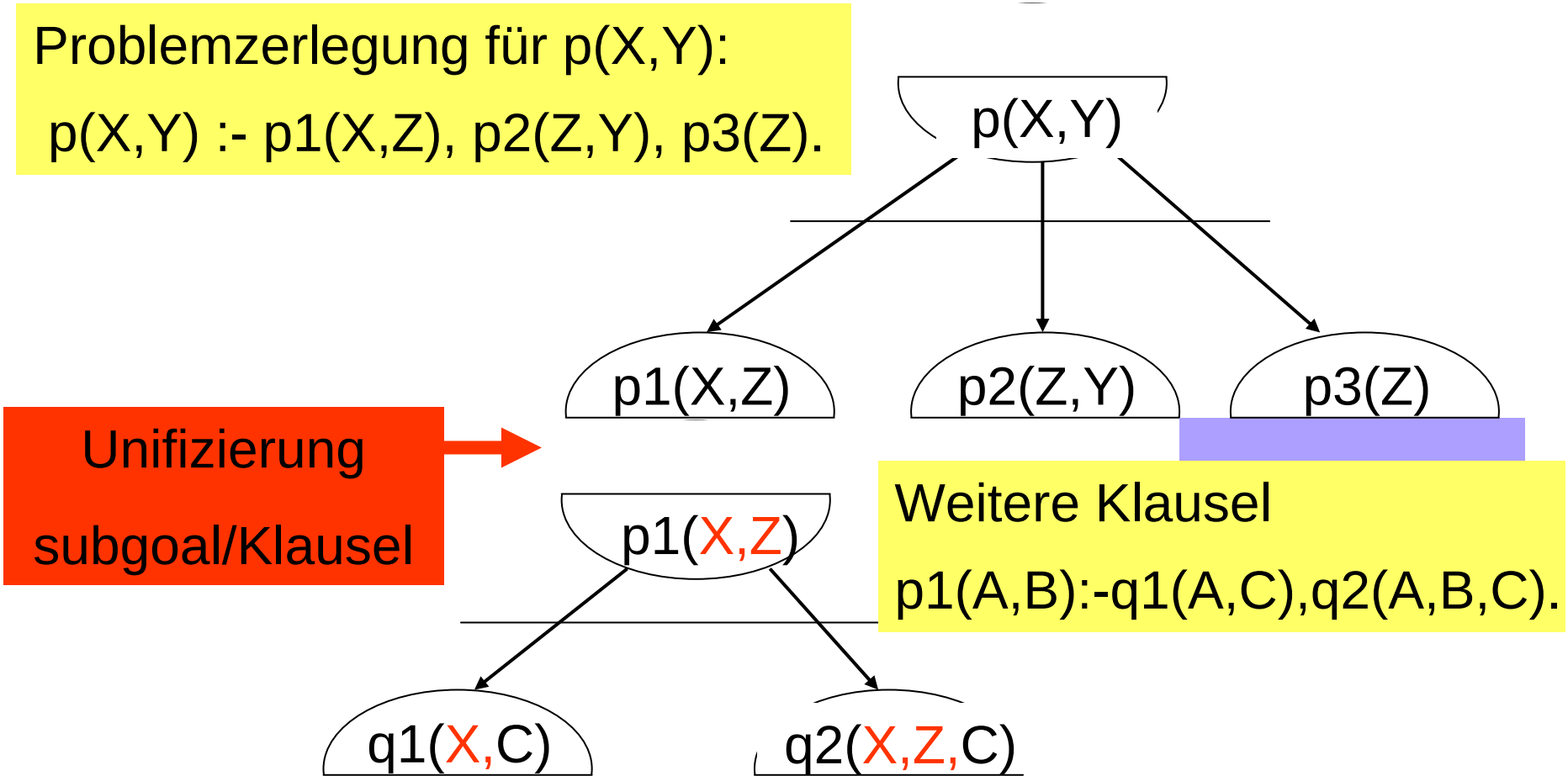
Weitere Klausel

p1(A,B):-q1(A,C),q2(A,B,C).

Modell für Relationen (mit Variablen)

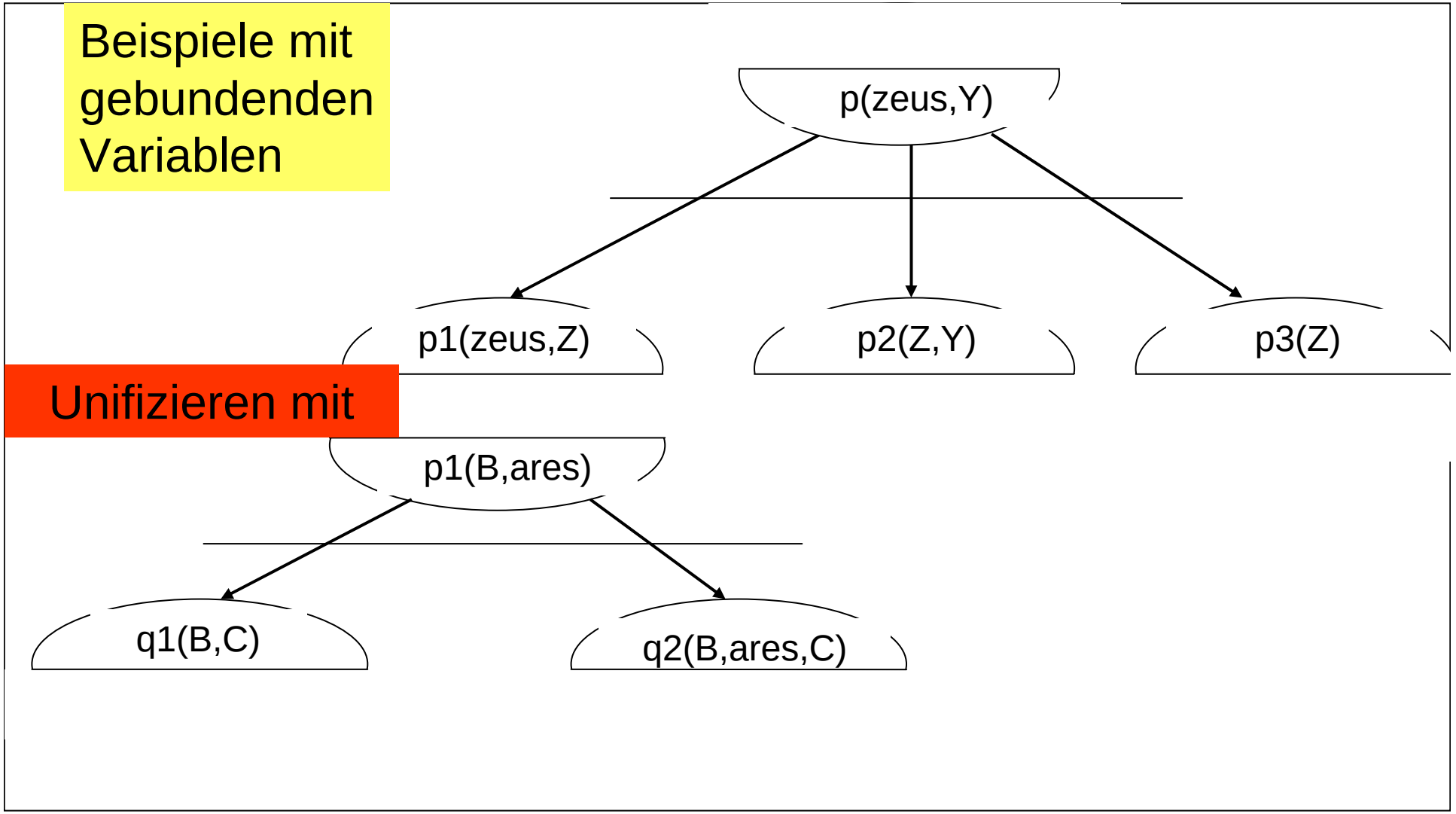
Problemzerlegung für p(X,Y):

p(X,Y) :- p1(X,Z), p2(Z,Y), p3(Z).



Modell für Relationen (mit Variablen)

Beispiele mit gebundenden Variablen

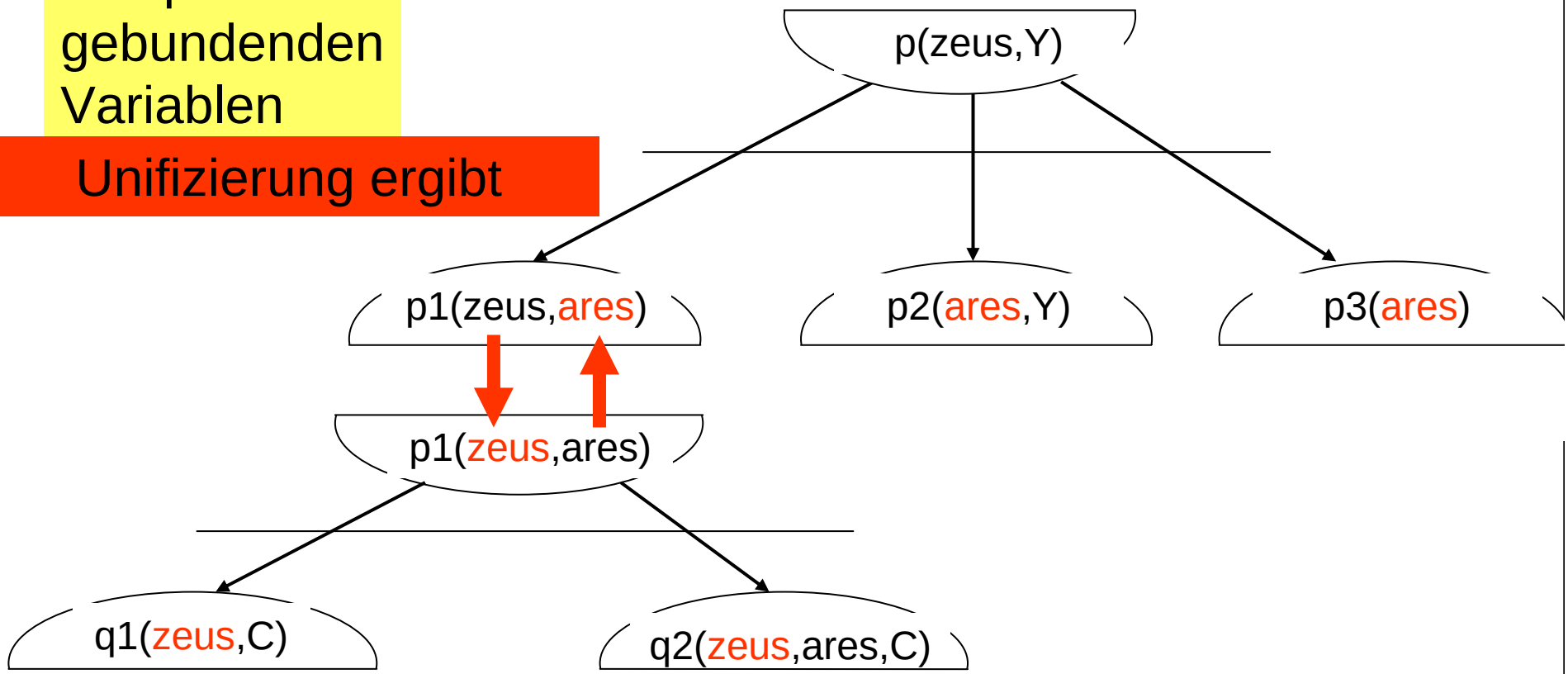


Unifizieren mit

Modell für Relationen (mit Variablen)

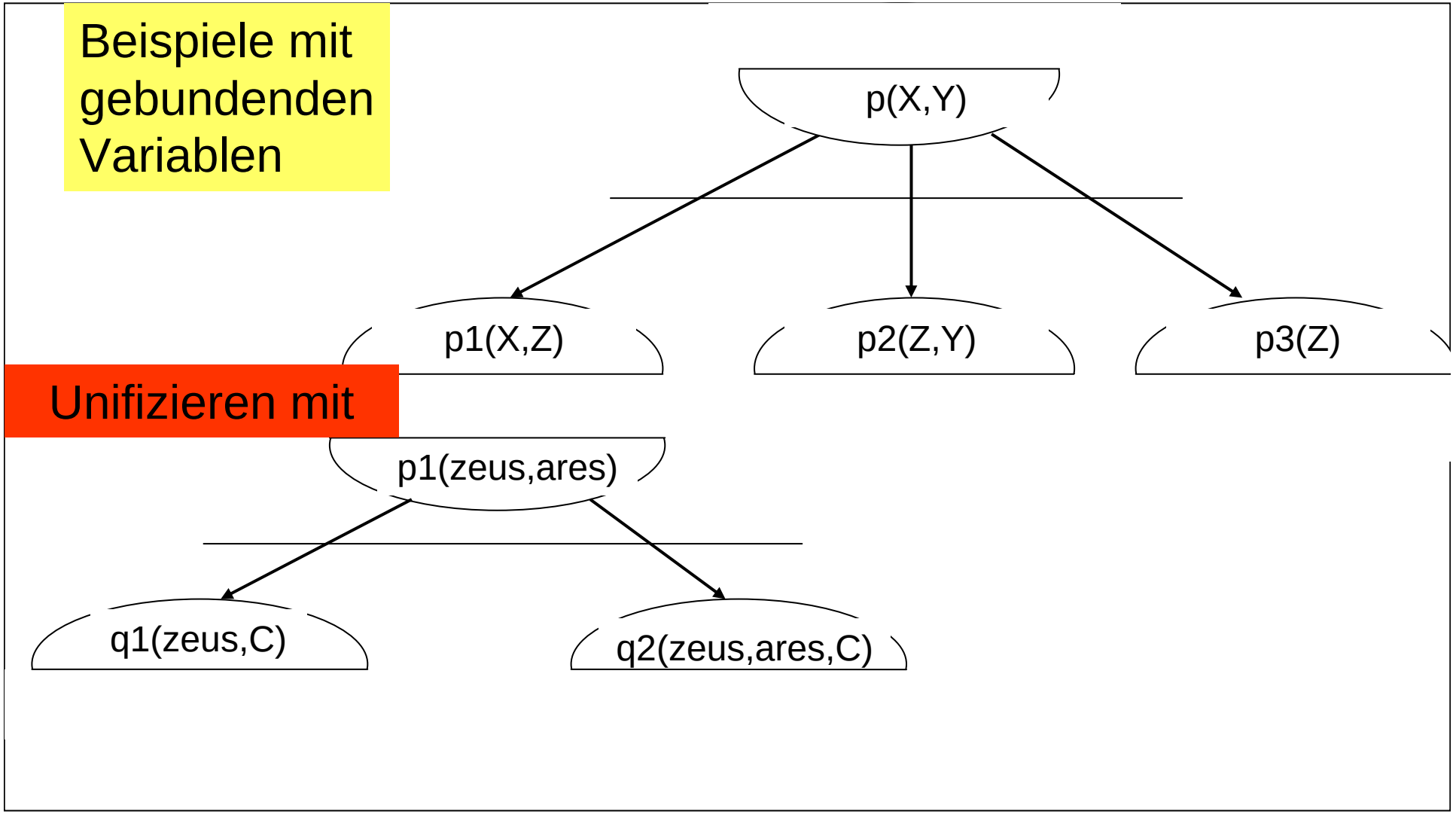
Beispiele mit gebundenden Variablen

Unifizierung ergibt

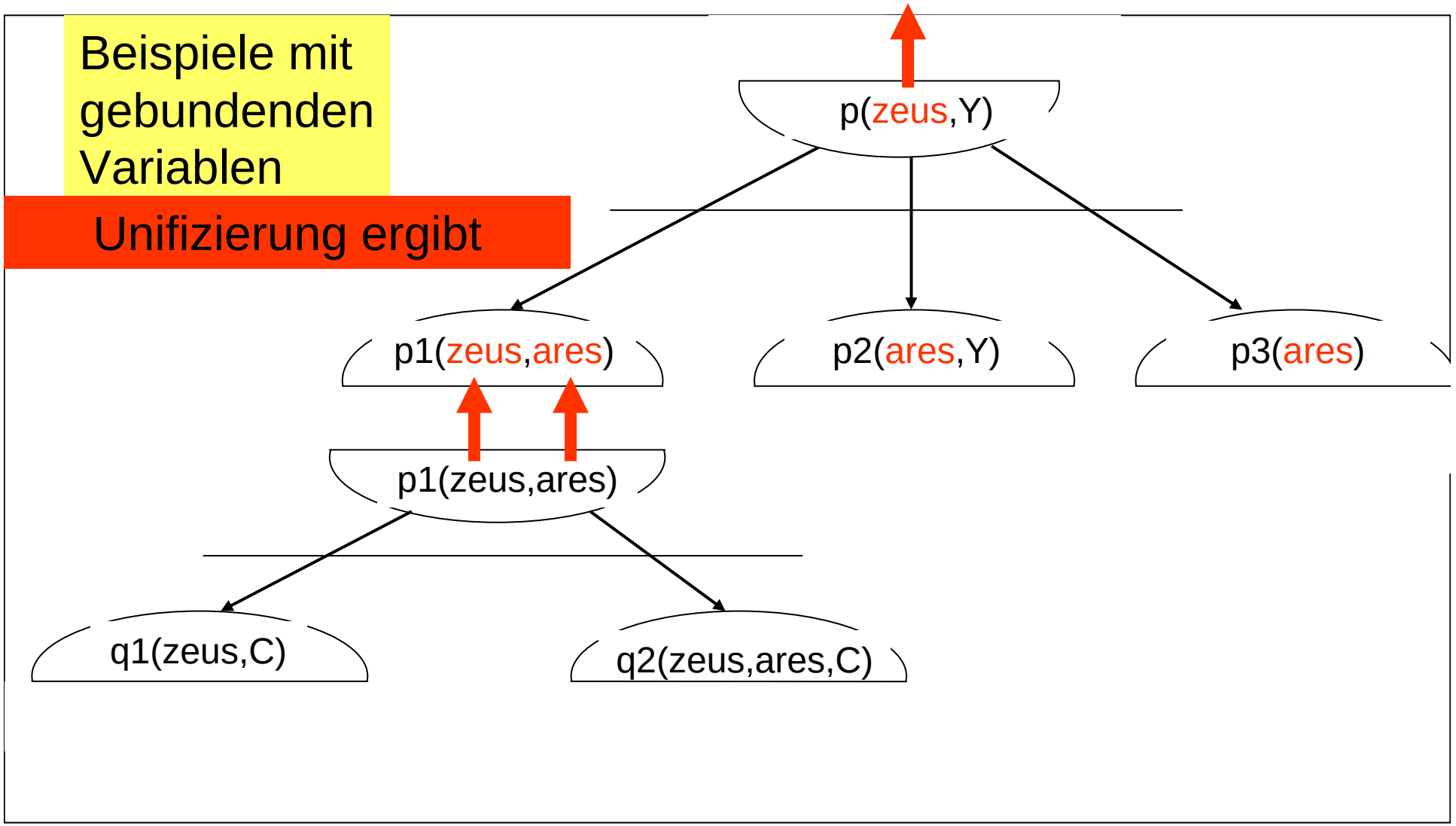


Modell für Relationen (mit Variablen)

Beispiele mit gebundenden Variablen



Modell für Relationen (mit Variablen)



Algorithmus mit Variablen

```
PROCEDURE solve(unsolved_goals: GOALLIST);
                    VAR k:KLAUSEL, g: GOAL;
BEGIN
  IF unsolved_goals = NIL THEN HALT(Result)
 ELSE g:= top(unsolved_goals);
    FORALL k ∈ klauseln(g) DO
                    (* Klauseln für g nacheinander rekursiv probieren*)
       IF unify (g , head(k), σ) THEN
          solve(concatenate( σ(subgoals(k)), σ(tail(unsolved_goals))  ))
                    (* subgoals der Klausel k weiter verfolgen *)
       END
    END (*FORALL*)
  END (*IF*)
END solve;
```

(für mehrfache Antworten: später)

σ ist die bei der Unifikation verwendete Substitution

Algorithmus mit Variablen

unify(g , head(k), σ) bewirkt

Prüfung auf Unifizierbarkeit von g und head(k)

Bindung von Variablen gemäß σ

durch Aktionen im Laufzeitsystem:

- Für die Klausel k wird Speicherplatz (frame) angelegt,
- darin Speicherplatz für Argumente (environment) mit Verweisen auf Bindungen.
- Beim Backtracking löschen der frames, die jünger als jüngster Backtrack-Punkt sind.

Algorithmus mit Variablen

Problem:

Effiziente Methode zur Bindung von Variablen beim Klauselaufruf und zum Auflösen von Bindungen beim Backtracking

Implementierungsidee:

Gegenseitige Referenzen der Variablen in Baumform

Anbindung an konstante/komplexe Strukturen an der Wurzel des Baumes

Dereferenzierung: Verfolgen einer Kette von Referenzen

Implementation: frames

Für jeden Klauselaufruf wird im Laufzeitsystem ein Speicherbereich reserviert: „frame“

Er enthält:

- Klausel einschließlich der Argumente.

  Effiziente Variante: „structure sharing“, in frame nur
  - Verweis auf template der Klausel
  - Argumente der Klausel („environment“).
- Verweis auf nächstes subgoal der aktuellen Klausel.
- Verweis auf Vater-Klausel der aktuellen Klausel.
- (Verweis auf jüngsten Backtrack-Punkt.)

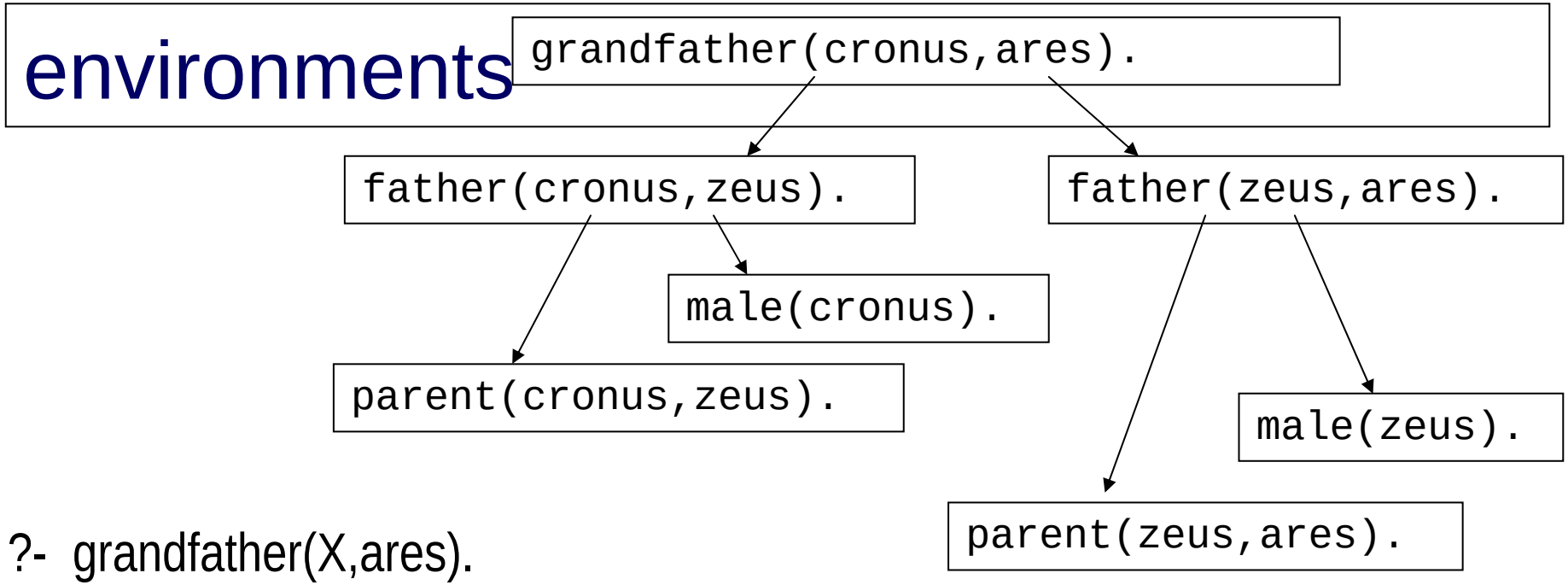
Implementation: frames

Frame enthält zusätzlich für Backtrack-Punkt :

- Nächste Klausel beim Backtracking.
- Davor liegender Backtrack-Punkt.
- Trail-Information (Protokoll der zu löschenden Variablenbindungen)

Dafür Reorganisation bei jedem Backtracking.

environments



?- grandfather(X,ares).

grandfather(X,ares):-father(X,Y),father(Y,ares).

father(X,Y):-parent(X,Y),male(X).

parent(cronus,zeus).

male(cronus).

father(zeus,ares):-parent(zeus,ares),male(zeus).

parent(zeus,ares).

male(zeus).

environments

?- grandfather(X,ares).

grandfather(X,ares):-father(X,Y),father(Y,ares).

father(X,Y):-parent(X,Y),male(X).

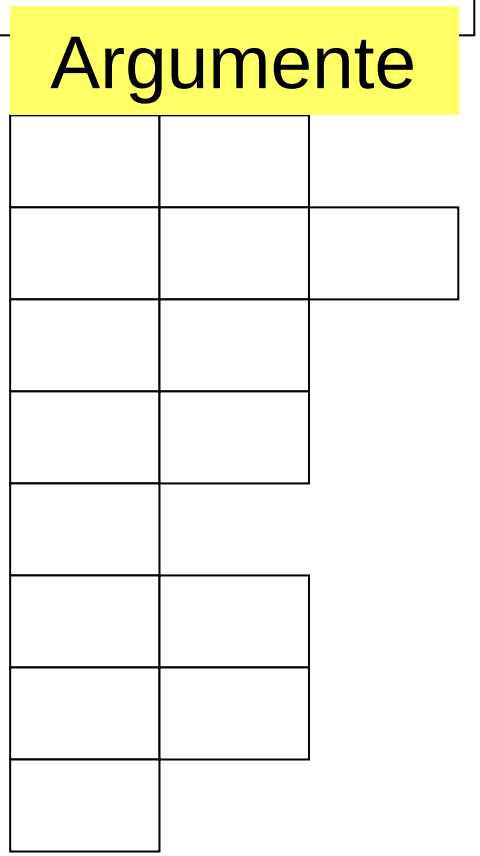
parent(cronus,zeus).

male(cronus).

father(zeus,ares):-parent(zeus,ares),male(zeus).

parent(zeus,ares).

male(zeus).



environments

?- grandfather(X,ares).

grandfather(X,ares):-father(X,Y),father(Y,ares).

father(X,Y):-parent(X,Y),male(X).

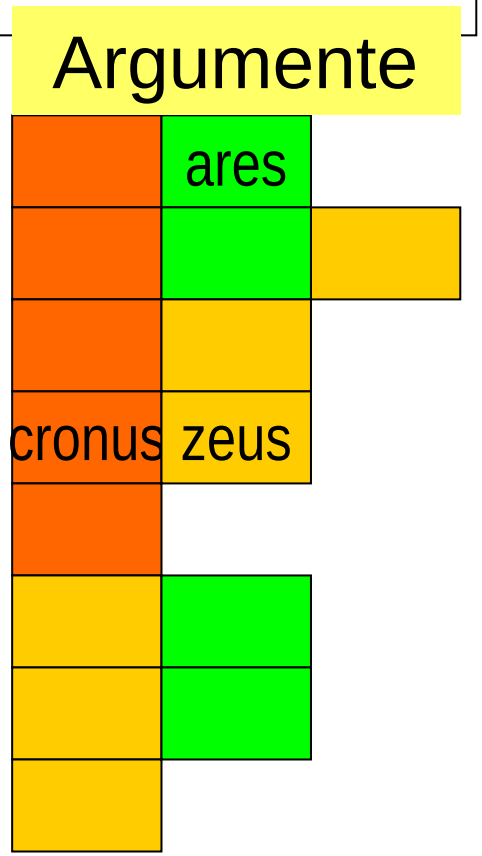
parent(cronus,zeus).

male(cronus).

father(zeus,ares):-parent(zeus,ares),male(zeus).

parent(zeus,ares).

male(zeus).



Aneinandergebundene Variable in einer Klasse

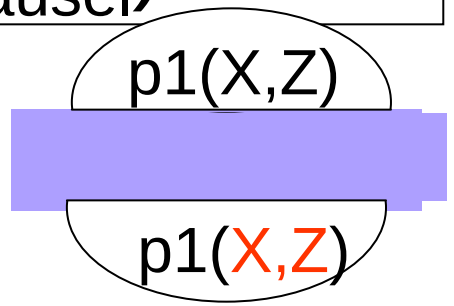
Implementation als Baum von Referenzen

Endpunkte: referierter Term (z.B. in Konstanten-Tabelle)

Unify($Argument^i_{goal}$,$Argument^i_{klausel}$)

T1 := Dereferenziere($Argument^i_{goal}$)

T2 := Dereferenziere($Argument^i_{klausel}$)



Unifikation erfolgreich, falls T1 und T2 unifizierbar.

Als „Seiteneffekte“ dabei Variablen binden:

Falls T1 und T2 Variable: T1 und T2 aneinander binden.

Prinzip der Rückwärtsreferenz:

Jüngere Variable an ältere Variable binden

Sonst: Variable Ti an den nicht-variablen Term binden

Falls jüngere Variable älter als jüngster Backtrack-Punkt:

Bindung im „trail“ protokollieren.

Backtracking

Rücksetzen der frames bis zum letzten Backtrack-Punkt.

Dabei entfallen Bindungen der Variablen aus gelöschten Frames automatisch

Einige Bindungen für ältere Variablen müssen evtl. explizit gelöst werden: gemäß Protokoll im „trail“.

Fortsetzung mit der im Backtrack-Punkt als Alternative vorgemerkten nächsten Klausel

Backtracking

?- grandfather(X,ares).

grandfather(X,ares):-father(X,Y),father(Y,ares).

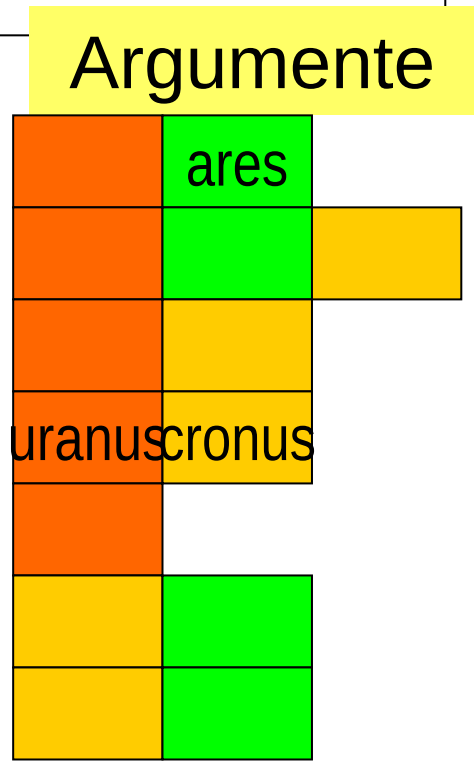
father(X,Y):-parent(X,Y),male(X).

parent(uranus,cronus).

male(uranus).

father(cronus,ares):-parent(cronus,ares),male(cronus).

parent(cronus,ares). Fehlschlag



Backtracking

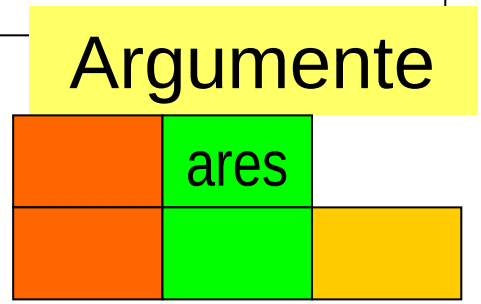
?- grandfather(X,ares).

Argumente

Backtracking

?- grandfather(X,ares).

grandfather(X,ares):-father(X,Y),father(Y,ares).

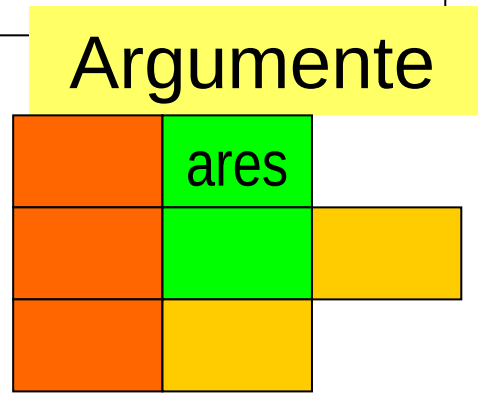


Backtracking

?- grandfather(X,ares).

grandfather(X,ares):-father(X,Y),father(Y,ares).

father(X,Y):-parent(X,Y),male(X).



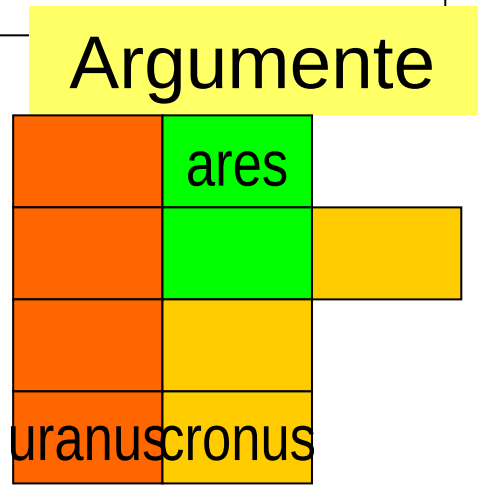
Backtracking

?- grandfather(X,ares).

grandfather(X,ares):-father(X,Y),father(Y,ares).

father(X,Y):-parent(X,Y),male(X).

parent(uranus,cronus).



Backtracking

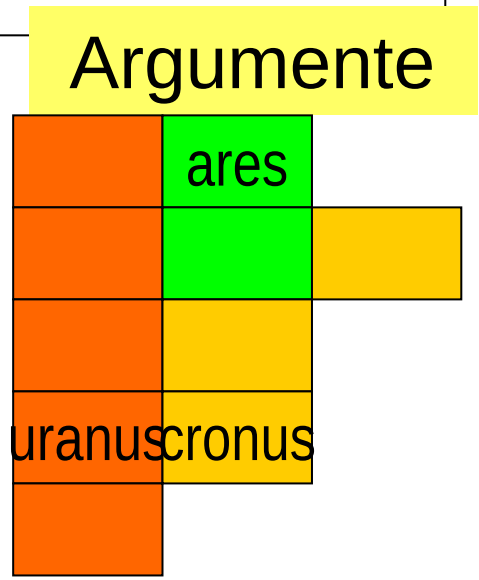
?- grandfather(X,ares).

grandfather(X,ares):-father(X,Y),father(Y,ares).

father(X,Y):-parent(X,Y),male(X).

parent(uranus,cronus).

male(uranus).



Backtracking

?- grandfather(X,ares).

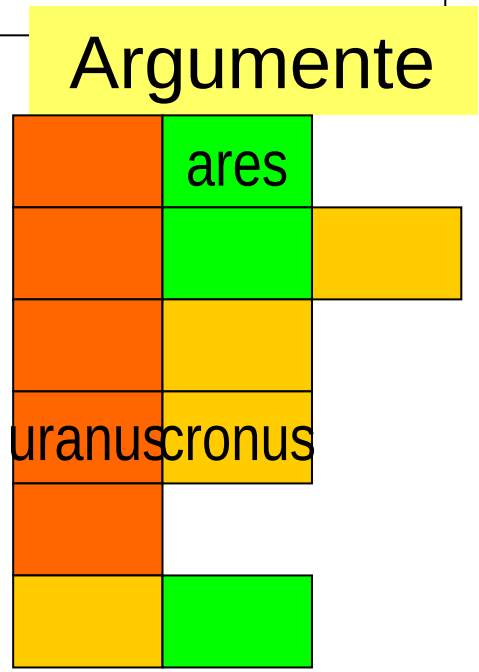
grandfather(X,ares):-father(X,Y),father(Y,ares).

father(X,Y):-parent(X,Y),male(X).

parent(uranus,cronus).

male(uranus).

father(cronus,ares):-parent(cronus,ares),male(cronus).



Backtracking

?- grandfather(X,ares).

grandfather(X,ares):-father(X,Y),father(Y,ares).

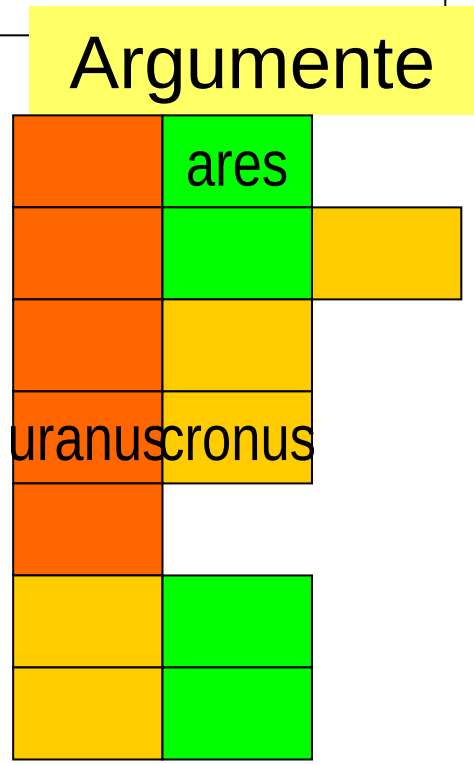
father(X,Y):-parent(X,Y),male(X).

parent(uranus,cronus).

male(uranus).

father(cronus,ares):-parent(cronus,ares),male(cronus).

parent(cronus,ares). Fehlschlag

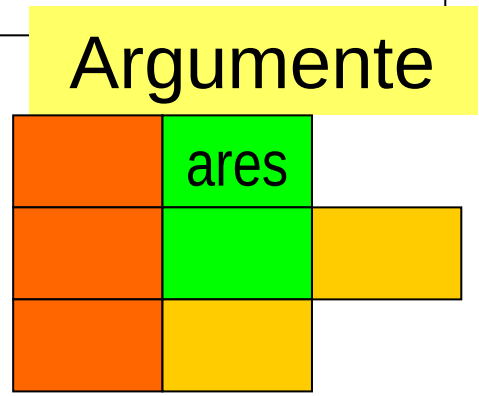


Backtracking

?- grandfather(X,ares).

grandfather(X,ares):-father(X,Y),father(Y,ares).

father(X,Y):-parent(X,Y),male(X).



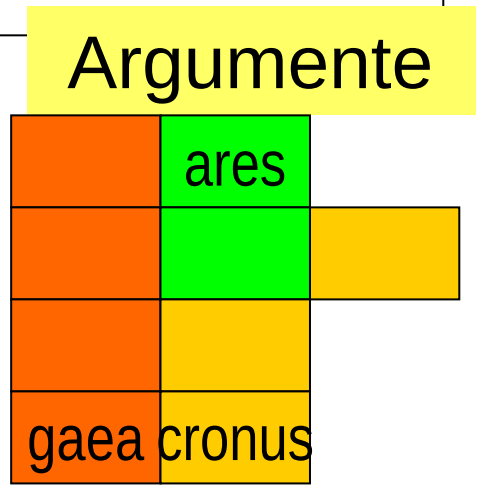
Backtracking

?- grandfather(X,ares).

grandfather(X,ares):-father(X,Y),father(Y,ares).

father(X,Y):-parent(X,Y),male(X).

parent(gaea,cronus).



Backtracking

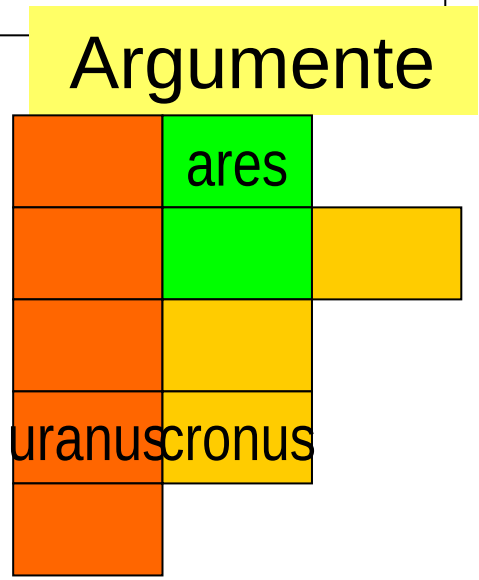
?- grandfather(X,ares).

grandfather(X,ares):-father(X,Y),father(Y,ares).

father(X,Y):-parent(X,Y),male(X).

parent(gaea,cronus).

male(gaea). Fehlschlag

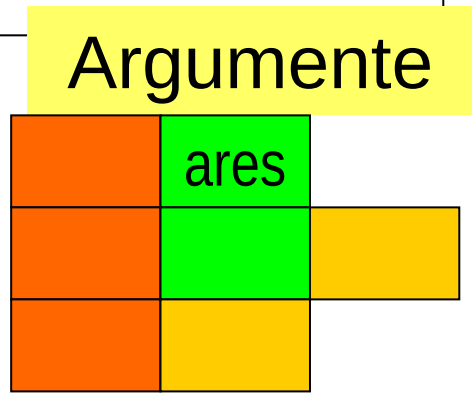


Backtracking

?- grandfather(X,ares).

grandfather(X,ares):-father(X,Y),father(Y,ares).

father(X,Y):-parent(X,Y),male(X).



Weitere Versuche mit Fakten für das parent-Prädikat …

erfolgreicher Beweis mit parent(cronus,zeus).

```
parent(uranus, cronus).
parent(gaea,   cronus).
parent(gaea,   rhea).
parent(rhea,   zeus).
parent(cronus, zeus).
parent(rhea,   hera).
parent(cronus, hera).
parent(cronus, hades).
parent(rhea,   hades).
parent(cronus, hestia).
parent(rhea,   hestia).
parent(zeus,   hermes).
parent(maia,   hermes).
...
```

Backtracking

?- grandfather(X,ares).

grandfather(X,ares):-father(X,Y),father(Y,ares).

father(X,Y):-parent(X,Y),male(X).

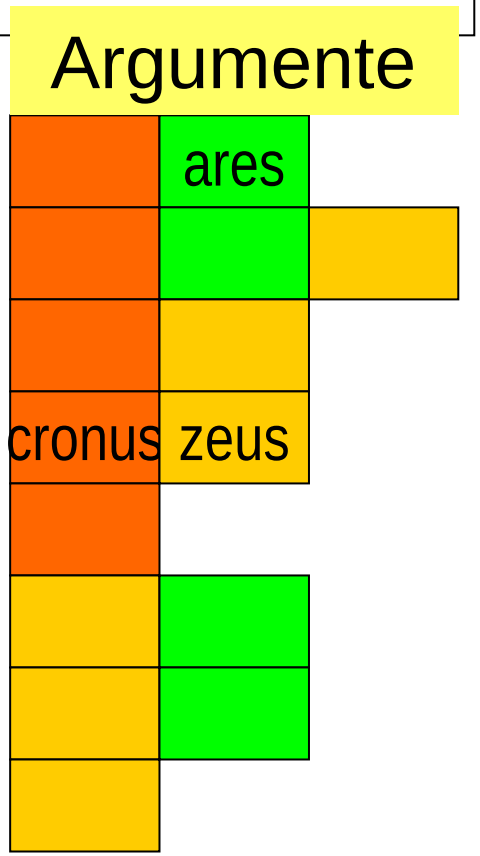
parent(cronus,zeus).

male(cronus).

father(zeus,ares):-parent(zeus,ares),male(zeus).

parent(zeus,ares).

male(zeus).



Algorithmus mit Variablen und mehrfachen Antworten

(0) (Start) $\mathcal{L}$: = [[Ausgangsproblem(e)]] .

(1) Falls $\mathcal{L}$ = [[ ], ...,[ ]] : REPORT(Result), weiter bei **(4)** .

(2) Sei L_i erste nicht-leere Subgoal-Liste aus $\mathcal{L}$ = [L_1,...,L_m] .
Sei g_{i1} erstes Element aus L_i = [g_{i1}, ..., $g_{i,ni}$] :

- g_{i1} aus L_i entfernen: L'_i := [g_{i2}, ..., $g_{i,ni}$] .
- Falls keine unifizierbaren Klauseln für g_{i1}: weiter bei **(4)** .

(3) Sei k die nächste unifizierbare Klausel für g_{i1} mit Unifikator: σ .
Falls k Fakt: weiter bei **(1)** .
Falls k Regel: g_{i1} :- g_1, ..., g_n :

$$\mathcal{L} := \sigma([[g_1, ..., g_n], L_1,.., L'_i,..., L_m]) .$$

Falls weitere Klauseln für g_{i1} existieren: *Backtrack-Punkt* setzer
Weiter bei **(2)** .

(4) Backtracking: Rücksetzen zum jüngsten *Backtrack-Punkt* :
$\mathcal{L}$ zurücksetzen auf Stand vor *Backtrack-Punkt* , weiter bei **(3)** .
Falls kein *Backtrack-Punkt* existiert: EXIT(no).

Weitere Implementationsprobleme

Für Vorwärtsreferenzen bei komplexeren Strukturen.

Für Eingriffe in Beweisablauf (cut).

Effiziensteigerung (vorzeitige Speicherfreigabe), z.B.

- last call Optimierung („lco“)
- deterministische Klauseln („dco“)

Später mehr dazu

Prolog-Compiler:
Übersetzung in optimiertes Programm
(WAM = Warren abstract machine)

Darstellung von Funktionen

Eine n-stellige Funktion f(x1,...,xn) ist als
(n+1)-stellige Relation r(x1,...,xn, f(x1,...,xn)) darstellbar

```
addiere(Summand1,Summand2,Summe)
nachfolger(Zahl,Nachfolger)
```

Der Funktionswert muss nicht an der letzten Stelle stehen.

```
multipliziere(Produkt,Faktor1,Faktor2)
```

Relationen und Funktionen

Modellierung von Aufgabenbereichen erfordert

Strukturierung

Beschreibung von Objekten

Beschreibung von Zusammenhängen

Beschreibung von Verfahren

Formale Modelle sind entscheidend für die

- Entwicklung
- Beschreibung
- Bewertung
- Validierung

von Konzepten und Verfahren

Modellierung (z.B. Sprachverarbeitung)

Strukturen z.B.:

- Wörter
- Satzstruktur (Syntax, Grammatik)
- Bedeutungsstruktur (Akteure, Handlung, ...)

Hans kauft das rote Fahrrad von Fritz.
Fritz verkauft sein rotes Fahrrad an Hans.
Fritz schenkt Hans sein rotes Fahrrad.
Hans klaut das rote Fahrrad von Fritz.

Verfahren z.B.

- Wörter erkennen
- Sätze erkennen
- Bedeutung erfassen

Modellierung

Sprache kann beschreiben

- Objekte (z.B. Substantive: Fahrrad)
- Eigenschaften (z.B. Adjektive: rot)
- Beziehungen (z.B. Verben: besitzt)

Hans besitzt ein rotes Fahrrad.

Prädikatenlogisch:

- Prädikate/Relationen beschreiben Eigenschaften und Beziehungen
- Terme beschreiben Objekte

Modellierung: Eigenschaften

Das rote Fahrrad ist drei Jahre alt, leicht reparaturbedürftig und in sehr gutem Zustand. Es kostet 100 € .

Attribut-Werte-Paare:

{ [Farbe,rot], [Alter, 3Jahre], [Zustand, sehr gut], [Funktionsfähigkeit, leicht reparaturbedürftig], [Preis, 100 €] }

Vektor von Attribut-Werten:

[rot, 3, sehr gut, leicht reparaturbedürftig, 100]

$\in \; W_{Farbe} \times W_{Alter} \times W_{zustand} \times W_{Funktionsfähigkeit} \times W_{Preis}$

Modellierung: Relation (Eigenschaften)

Fahrrad-Angebote

$$\subseteq W_{Farbe} \times W_{Alter} \times W_{zustand} \times W_{Funktionsfähigkeit} \times W_{Preis}$$

Farbe	Alter	Zustand	Fkts-fähig.	Preis
rot	3	sehr gut	Leicht rep.	100
rot	2	mittel	Ersatzteile	20
gelb	2	gut	ja	100

[blau, 1, sehr gut, sehr gut, 10] $\notin$ Fahrrad-Angebote

Fahrrad-Angebote(blau, 1, sehr gut, sehr gut, 10) = falsch

Modellierung: Relation (Beziehungen)

Hans besitzt das Fahrrad.

besitzen $\subseteq$ Gegenstände x Menschen

Gegenstände	Menschen
Fahrrad	Hans
Kleingeld	Hans
Villa	Fritz

[Villa, Hans] $\notin$ besitzen

besitzen(Villa, Hans) = falsch

Relation

Eine n-stellige Relationen ist definierbar als

- Teilmenge $R \subseteq W_1 \times ... \times W_n$ bzw.
- Prädikat $W_1 \times ... \times W_n \rightarrow$ {wahr,falsch}

Endliche Relationen können als Tabellen dargestellt werden (Datenbank), z.B. als Faktenmenge in Prolog.

Relationen können z.B.

- Eigenschaften von Objekten (z.B. Datenbank) oder
- Beziehungen zwischen Objekten oder
- Beschränkungen (Constraints)

beschreiben

Modellierung: Relation (Beschränkungen)

Falls das Fahrrad nicht funktionsfähig ist, soll der Zustand höchstens „schlecht“ sein.

Durch Beschränkungen (Constraints)

$$C \subseteq W_1 x ... x W_n$$

kann verlangt werden, dass nur bestimmte Wertekombinationen zulässig sind.

Beispiele:

- Stundenplanung
- 8-Damenproblem

Constraint-Satisfaction-Problem (CSP).

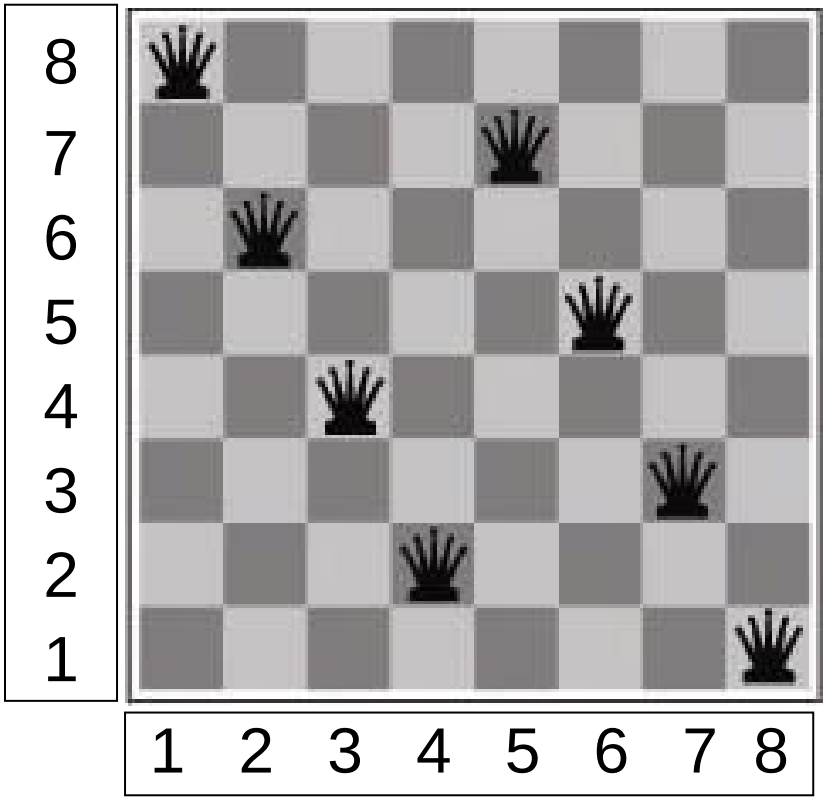
Bestimme n-Tupel $[w_1, ..., w_n] \in W_1 \text{ x } ... \text{x } W_n$,

das gegebenen Beschränkungen $C_1,\ldots,C_k \subseteq W_1 \text{ x} ... \text{x } W_n$

genügt: $[w_1, ..., w_n] \in C_1$, ..., $[w_1, ..., w_n] \in C_k$

Fallstudie: 8-Damen-Problem

8 Damen auf dem Schachfeld so platzieren,
dass keine eine andere angreifen kann
(im Beispiel nicht erfüllt)



Fallstudie: 8-Damen-Problem

Modellierung:

Position p_i der i-ten Dame (i = 1,...,8) beschrieben durch

$p_i = x_i|y_i$ mit Spalte $x_i \in \{1,...,8\}$, Zeile $y_i \in \{1,...,8\}$

Menge der möglichen Positionen

$$P = \{ \; x|y \;\; | \;\; x \in \{1,...,8\} , y \in \{1,...,8\} \;\; \}$$

Lösungen haben Form

$$l = [\; x_1|y_1, \; ... \; , \; x_8|y_8 \;] \in P \times P \times P \times P \times P \times P \times P \times P$$

Lösungsmenge $L \subseteq P \times P \times P \times P \times P \times P \times P \times P$

Fallstudie: 8-Damen-Problem

Lösungen müssen Constraints erfüllen:

Keine Dame darf eine andere angreifen.

Formulierung:

Dame d_i auf $x_i|y_i$ greift Dame d_k auf $x_k|y_k$ an,

falls $x_i = x_k$ oder $y_i = y_k$ oder $x_i-y_i = x_k-y_k$ oder $x_i+y_i = x_k+y_k$

Constraint C_{ik} bezüglich Dame d_i und d_k (mit $i \neq k$)

(Constraint beschreibt **zulässige** Werte)

$C_{ik} = \{ [x_1|y_1,...,x_8|y_8] \mid x_i \neq x_k \wedge y_i \neq y_k \wedge x_i-y_i \neq x_k-y_k \wedge x_i+y_i \neq x_k+y_k \}$

Lösungen: $L = \cap \{ C_{ik} \mid 1 \leq i < k \leq 8\}$

Fallstudie: 8-Damen-Problem

Andere Formulierung:

Dame d_i steht in Spalte i, Lösungen haben Form $l = [\, y_1,..., y_8 \,]$

Dame d_i auf $i|y_i$ greift Dame d_k auf $k|y_k$ an,

falls $y_i = y_k$ oder $i-y_i = k-y_k$ oder $i+y_i = k+y_k$

Constraint C_{ik} bezüglich Dame d_i und d_k

(Constraint beschreibt zulässige Werte)

$C_{ik} = \{ [y_1,...,y_8] \mid y_i \neq y_k \wedge i-y_i \neq k-y_k \wedge i_i+y_i \neq k+y_k \}$

Lösungen: $L = \cap \{ C_{ik} \mid 1 \leq i < k \leq 8 \}$

Vorteil:
Suchraum
ist einfacher

Fallstudie: 8-Damen-Problem

Vergleich der beiden Formulierungen:

Möglichkeiten bei 1. Formulierung:

Dame jeweils auf ein freies Feld stellen

$64 \bullet 63 \bullet 62 \bullet 61 \bullet 60 \bullet 59 \bullet 58 \bullet 57 \approx 3 \bullet 10^{14}$ Möglichkeiten

Möglichkeiten bei 2. Formulierung:

Dame jeweils auf ein Feld in ihrer Spalte stellen

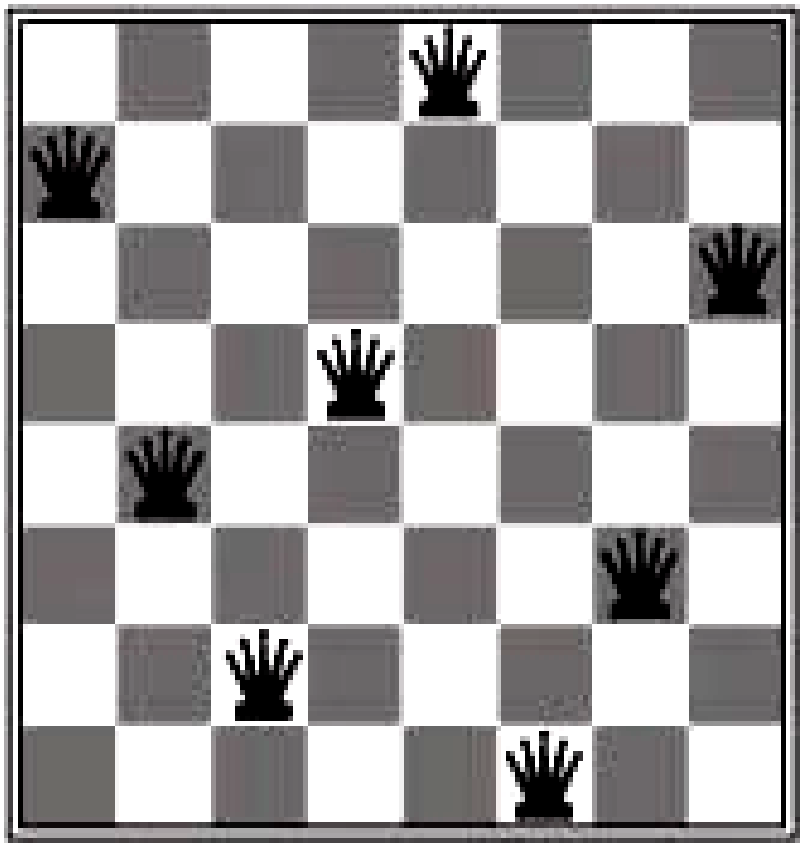
$8 \bullet 8 \bullet 8 \bullet 8 \bullet 8 \bullet 8 \bullet 8 \bullet 8 = 8^8$ Möglichkeiten

Zusätzliche Verbesserung:

Dame jeweils auf bisher nicht angegriffenes Feld stellen

2057 Möglichkeiten

Fallstudie: 8-Damen-Problem



Eine Lösung

Definition von Relationen

Relationen können kombiniert werden z.B.
Mengentheoretisch
(Durchschnitt, Komplement, Projektion...)
Logisch
(Konjunktion,Quantifizierung, ...)

Prolog-Prozeduren definieren neue Relationen (Kopf) aus gegebenen Relationen (Körper). Variable, die im Kopf nicht vorkommen, sind im Körper existentiell quantifiziert.

```
grandfather1(X,Z):-father(X,Y),father(Y,Z).
grandfather1(X,Z):-father(X,Y),mother(Y,Z).
```

Funktionen

Funktionen beschreiben z.B. Abhängigkeiten oder eindeutige Zuordnungen. Im PK1 werden Funktionen syntaktisch durch Terme dargestellt.

Vater(Ares) = Zeus

Kinder(Hera,Zeus) = {Ares, Hephaistos, Hebe}

Vater: Götter $\rightarrow$ Götter

Kinder: Götter x Götter $\rightarrow 2^{\text{Götter}}$

$2^M = \{ N \mid N \subseteq M \}$ bezeichnet die Potenzmenge

Funktionen können Argumente von Relationen oder von Funktionen sein.

Vater(Vater(Ares)) = Kronos

Verheiratet(Vater(Ares),Mutter(Ares))

Funktion als Relation

Eine n-stellige Funktion $W_1 \times ... \times W_n \rightarrow W$

kann aufgefasst werden als

n+1 stellige Relation $R \subseteq W_1 \times ... \times W_n \times W$

mit $[w_1, ..., w_n, w] \in R \leftrightarrow f(w_1, ..., w_n) = w$

Hans besitzt das Fahrrad.

besitzen $\subseteq$ Gegenstände x Menschen

Besitzer(Fahrrad) = Hans

Besitzer: Gegenstände $\rightarrow$ Menschen

Typen, Signaturen

Im Gegensatz zu anderen Programmiersprachen verlangt Prolog i.a. keine Deklaration von Typen für die Argumente (Wertebereiche W_i) von Relationen/Funktionen. Der Typ eines konkreten Arguments ergibt sich aus der Schreibweise.

Allerdings werden bei einigen Prädikaten bestimmte Anforderungen an die Argumente (z.B. Bindung an eine Zahl) verlangt.

Einschränkungen kann es auch geben bzgl.

- Eingabeparametern (müssen instantiiert sein)
- Ausgabeparametern (werden durch Prädikat instantiiert)

Darstellung von Funktionen in Prolog

Ideales Prinzip: Es kann nach Funktionswert
und/oder nach Argumentwerten gefragt werden

```
?- addiere(a,b,Summe).
?- addiere(Summand,b,s).
?- addiere(a,Summand,s).
?- addiere(Summand1,Summand2,s).
...
?- addiere(Summand1,Summand2,Summe).
```

Darstellung von Funktionen in Prolog

Umkehrfunktionen unmittelbar definierbar

```
subtrahiere(Minuend,Subtrahend,Differenz)
    :=addiere(Differenz,Subtrahend,Minuend).
```

Primitiv-rekursive Funktionen

```
zahl(o).
zahl(s(X)):-zahl(X).
```

Verwenden o als spezielle Konstante für die kleinste Zahl

```
kleinergleich(o,X):- zahl(X).
kleinergleich(s(X),s(Y)):-kleinergleich(X,Y).
```

Prolog unterscheidet keine Typen.

Damit X explizit auf Zahlen beschränkt ist:

```
kleinergleich(o,X):-zahl(X).
```

Primitiv-rekursive Funktionen

```
identitaet-i(X1,...Xi,...,Xn,Xi).
constante-c(X1,...,Xn,c).
nachfolger(X,s(X)).
```

```
substitution(X1,...,Xn,F)
   :-f1(X1,...,Xn,F1),..., fm(X1,...,Xn,Fm),
     g(F1,...,Fm,F).
```

```
rekursion(X1,...,Xn,o,F):-g(X1,...,Xn,F).
rekursion(X1,...,Xn,s(X),F)
:-rekursion(X1,...,Xn,X,R),h(X1,...,Xn,X,R,F).
```

Primitiv-rekursive Funktionen

```
add(o,X,X).
add(s(X),Y,s(Z)):- add(X,Y,Z).
```

```
mult(o,X,o).
mult(s(X),Y,Z):- mult(X,Y,W),add(W,Y,Z).
```

```
exp(s(o),o,o).
exp(o,s(X),s(o)).
exp(s(N),X,Y) :- exp(N,X,Z),mult(Z,X,Y).
```

Es kann bei nicht gebundenen Argumenten Probleme geben, deshalb besser expliziter Bezug auf Zahlen:

```
add(o,X,X):- zahl(X).
```

und analog für die anderen Funktionen.

Primitiv-rekursive Funktionen

```
factorial(o,s(o)).
factorial(s(N),F):-
        factorial(N,F1),mult(s(N),F1,F).
```

```
minimum(X,Y,X):-kleinergleich(X,Y).
minimum(X,Y,Y):-kleinergleich(Y,X).
```

```
mod(X,Y,Z):-
 kleiner(Z,Y),mult(Y,Q,W),add(W,Z,X).
```

Alternativ:

```
mod(X,Y,X):-kleiner(X,Y).
mod(X,Y,Z):-add(X1,Y,X),mod(X1,Y,Z).
```

Ackermann-Funktion (Péter-Funktion)

```
ackermann(o,N,s(N)).
ackermann(s(M),o,V):- ackermann(M,s(o),V).
ackermann(s(M),s(N),V):-
     ackermann(s(M),N,V1), ackermann(M,V1,V).
```

Prolog-Arithmetik

```
?- X = 1 + 2 * 3 .
X = 1 + 2 * 3
```

- Behandlung als Term

```
?- X is 1 + 2 * 3 .
X = 7
```

- Behandlung als auszuwertender Arithmetischer Ausdruck

`is/2`

zweistelliges Prädikat in infix-Schreibweise
Links: ungebundene Variable oder Zahl
Rechts: auswertbarer arithmetischer Ausdruck

Prolog-Arithmetik

`value` **`is`** `expression`

Abarbeitung:

- `expression` wird vom Arithmetik-Evaluierer als arithmetischer Ausdruck ausgewertet
- Resultat wird mit `value` unifiziert

Überschreiben nicht möglich

```
?- X is 1 + 2 * 3 .
X = 7
```

```
?- 7 is 1 + 2 * 3 .
yes
```

```
?- 3 + 4 is 1 + 2 * 3 .
no
```

Prolog-Arithmetik

Verfügbare arithmetische Operationen („reelle“ Zahlen):

`+`, `-` , `*`, /, //, mod und ggf. weitere

Verfügbare arithmetische Vergleichsoperationen:

`>` , `<` , `>=`, `=<`, `=:=` , `=/=` (und ggf. weitere)

Vergleiche für numerisch auswertbare Ausdrücke auf beiden Seiten

Unterschied:

`is` erlaubt Ausdruck nur rechts,
ggf. Unifizierung mit Variabler auf linker Seite

`=:=` überprüft Identität zweier Ausdrücke

Prolog-Arithmetik

Ursprüngliche Definition

```
factorial(o,s(o)).
factorial(s(N),F) :- factorial(N,F1), mult(s(N),F1,F).
```

erlaubt Anfrage

```
?- factorial(X,Y).
```

Bei Prolog-Arithmetik ist diese Anfrage nicht möglich

```
factorial(0,1).
factorial(N,F) :- N> 0, N1 is N-1, factorial(N1,F1), F is N * F1.
```

- Laufzeitfehler, wenn Ausdrücke nicht auswertbar sind

Speziell bei ungebundenen Variablen:
Zwang zu „prozeduraler“ Reihenfolge der subgoals

Prolog-Arithmetik

Ursprüngliche Definition

```
add(Summand1,Summand2,Summe)
```

erlaubt Definition

```
minus(Minuend,Subtrahend,Differenz) :- add(Subtrahend,Differenz,Minuend)
```

Bei Verwendung der Prolog-Arithmetik ist das nicht möglich.

```
addiere(X,Y,S) :- S is X + Y
```

entspricht nicht früherem `add(X,Y,S)`

Operatoren

Standard-Schreibweise in Prolog:

Struktur/Term: funktor(Argumente)

Alternativ: Operator-Schreibweise für bessere Lesbarkeit

		Funktor
Infix-Operator	7 + 9 für Struktur + (7,9)	+/2
Prefix-Operator	- 9 für Struktur - (9)	-/1
Postfix-Operator	9! (Fakultät)	!/1

Zu klären:

Priorität: $7 + 9 * 2 = 7 + (9 * 2)$

Assoziativität: $7 - 9 - 2 = (7 - 9) - 2$

Operatoren

Operatoren deklarieren mittels

```
op(Priorität, Typ, Name)
```

```
op( 500, yfx, '-' ).
op( 500, yfx, '+' ).
op( 400, yfx, '*' ).
```

Vergabe von Prioritäten: 0,...,1200 (Maximum)

Priorität eines Terms: Priorität des Hauptfunktors

Priorität 0 haben:

Atome (außer Operatoren), Zahlen, Variable, Zeichenketten,

in Klammern eingeschlossene Terme

Operatoren

Festlegung der Assoziativität durch Typen:

- Infixoperatoren `xfx, xfy, yfx`,
- Präfixoperatoren `fx, fy`,
- Postfixoperatoren `xf, yf`.

```
op( 500, yfx, '-' ).
op( 500, yfx, '+' ).
op( 400, yfx, '*' ).
```

`f`: Funktor

`x`: Term mit geringerer Priorität als `op`

`y`: Term mit maximal gleicher Priorität wie `op`
(andernfalls Klammern notwendig)

Beispiele für Operatoren

Standardmäßig im Prolog-Interpreter

(built-in-Operatoren)

1200 xfx	:-	
1200 fx	?-	
1100 xfy	;	
1000 xfy	,	
900 fy	not	
700 xfx	=, \=	(Unifikation)
	==, \==	(Identität für Terme)
	<, =:=, >, =<, >=, =\=, is	(Arithmetik)
500 yfx	+, -	
500 fx	+, -,	
400 yfx	*, /, //,mod	

Rekursive Definitionen

```
zahl(0).
zahl(s(X)):-zahl(X).
```

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

```
erreichbar(X,X).
erreichbar(X,Y)
          :- benachbart(X,Z),erreichbar(Z,Y).
```

Transitiver Abschluss von Relationen

Logisch äquivalente rekursive Definitionen:

```
erreichbar(X,Y):- benachbart(X,Z), erreichbar(Z,Y).
erreichbar(X,X).
```

```
erreichbar(X,X).
erreichbar(X,Y):- benachbart(X,Z), erreichbar(Z,Y).
```

```
erreichbar(X,X).
erreichbar(X,Y):- erreichbar(Z,Y), benachbart(X,Z).
```

```
erreichbar(X,Y):- erreichbar(Z,Y), benachbart(X,Z).
erreichbar(X,X).
```

Inhaltlich äquivalent z.B. auch:

```
erreichbar(X,Y) :- erreichbar(X,Z), benachbart(Z,Y).
erreichbar(X,X).
```

Unendliche Beweisversuche

```
erreichbar(X,Y):-erreichbar(X,Z),benachbart(Z,Y).
```

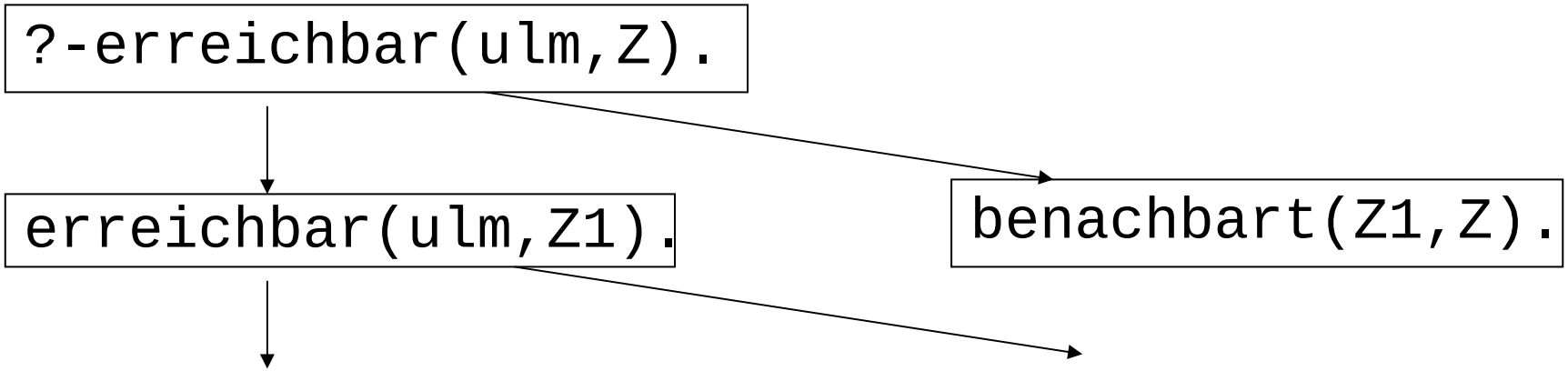
Unendliche Beweisversuche

```
erreichbar(X,Y):-erreichbar(X,Z),benachbart(Z,Y).
```

```
?-erreichbar(ulm,Z).
```

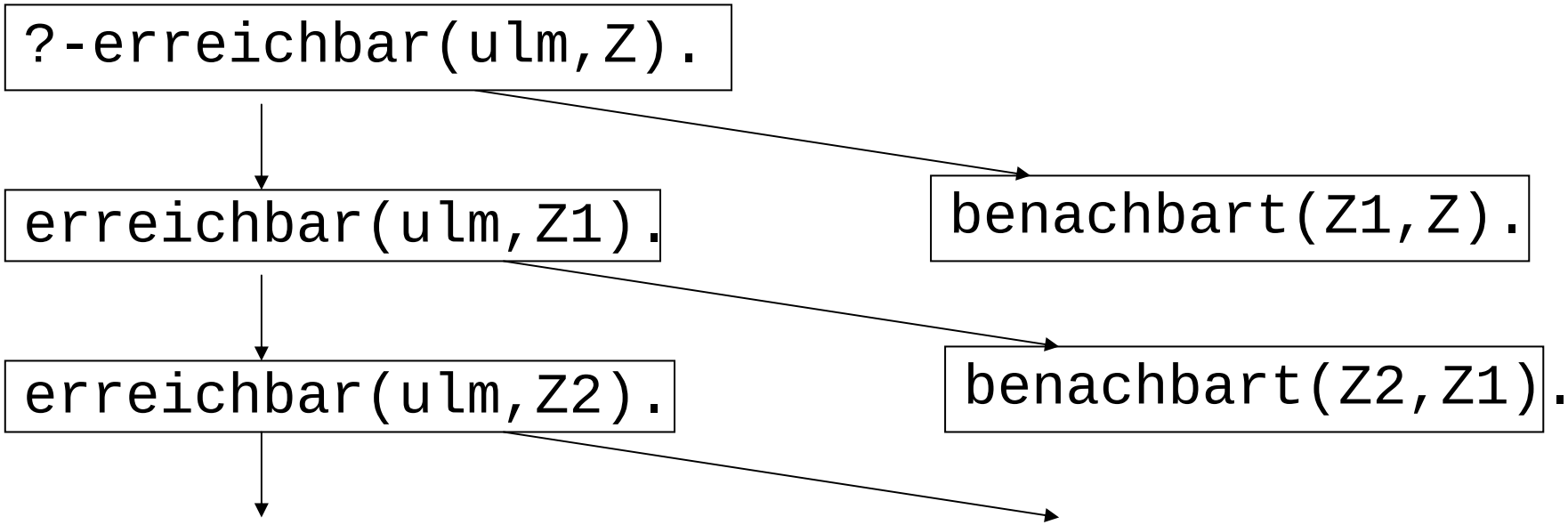
Unendliche Beweisversuche

```
erreichbar(X,Y):-erreichbar(X,Z),benachbart(Z,Y).
```



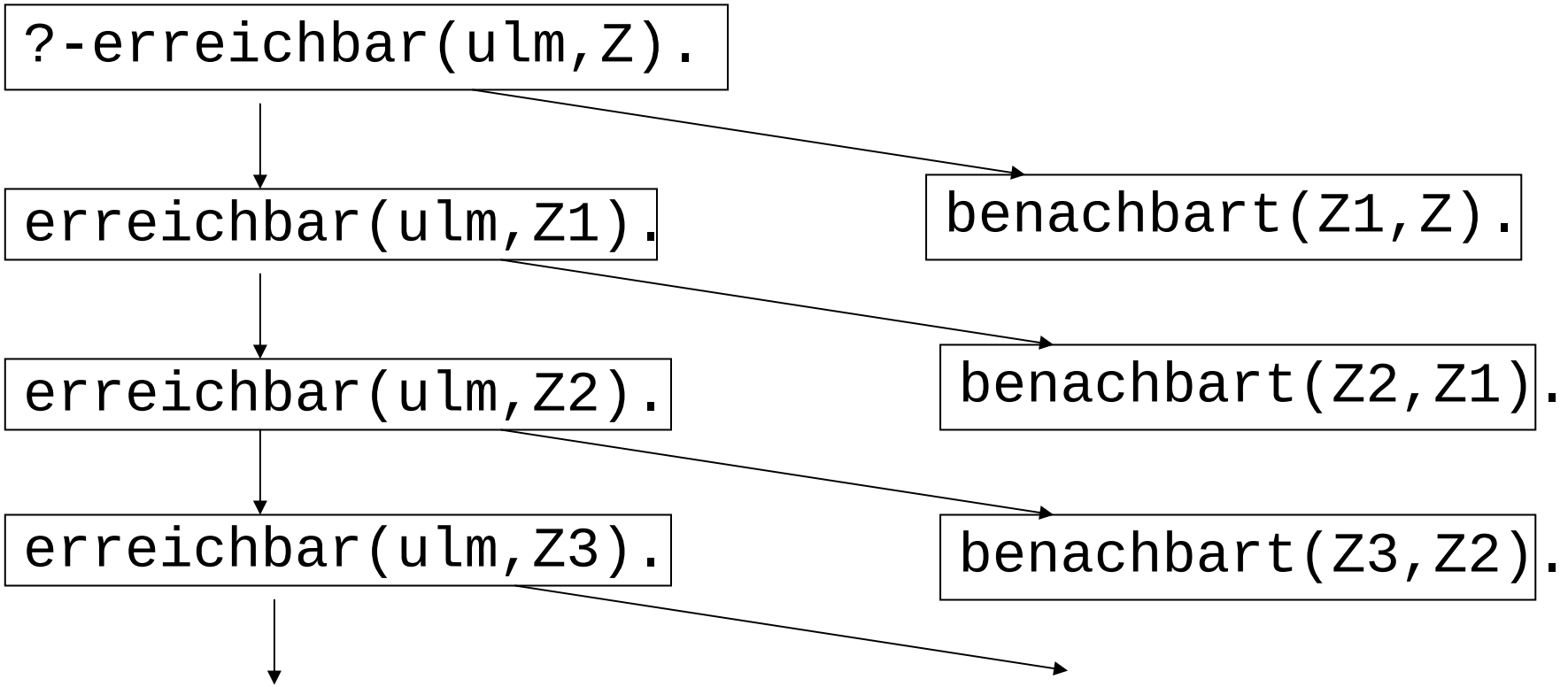
Unendliche Beweisversuche

```
erreichbar(X,Y):-erreichbar(X,Z),benachbart(Z,Y).
```



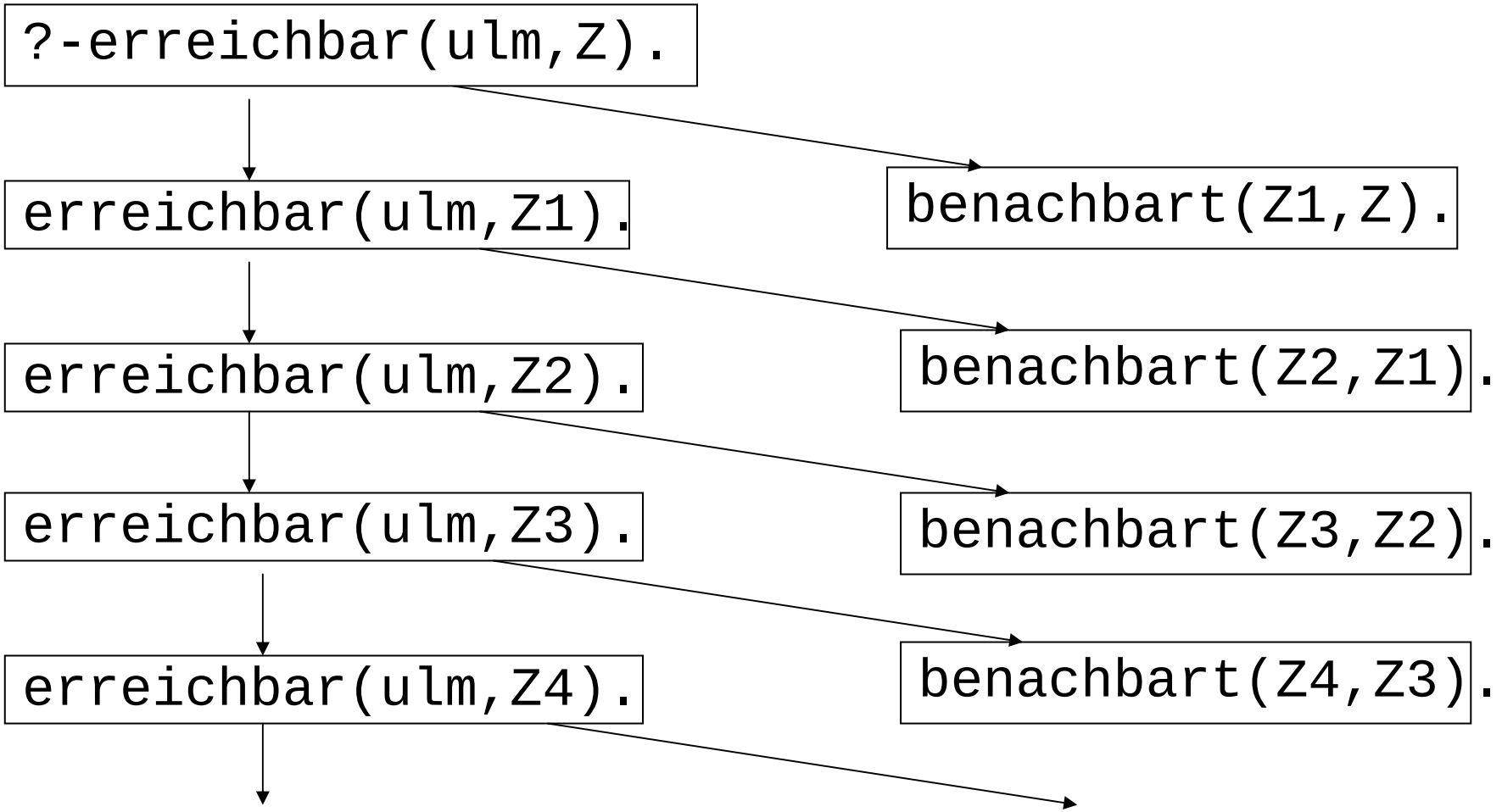
Unendliche Beweisversuche

```
erreichbar(X,Y):-erreichbar(X,Z),benachbart(Z,Y).
```



Unendliche Beweisversuche

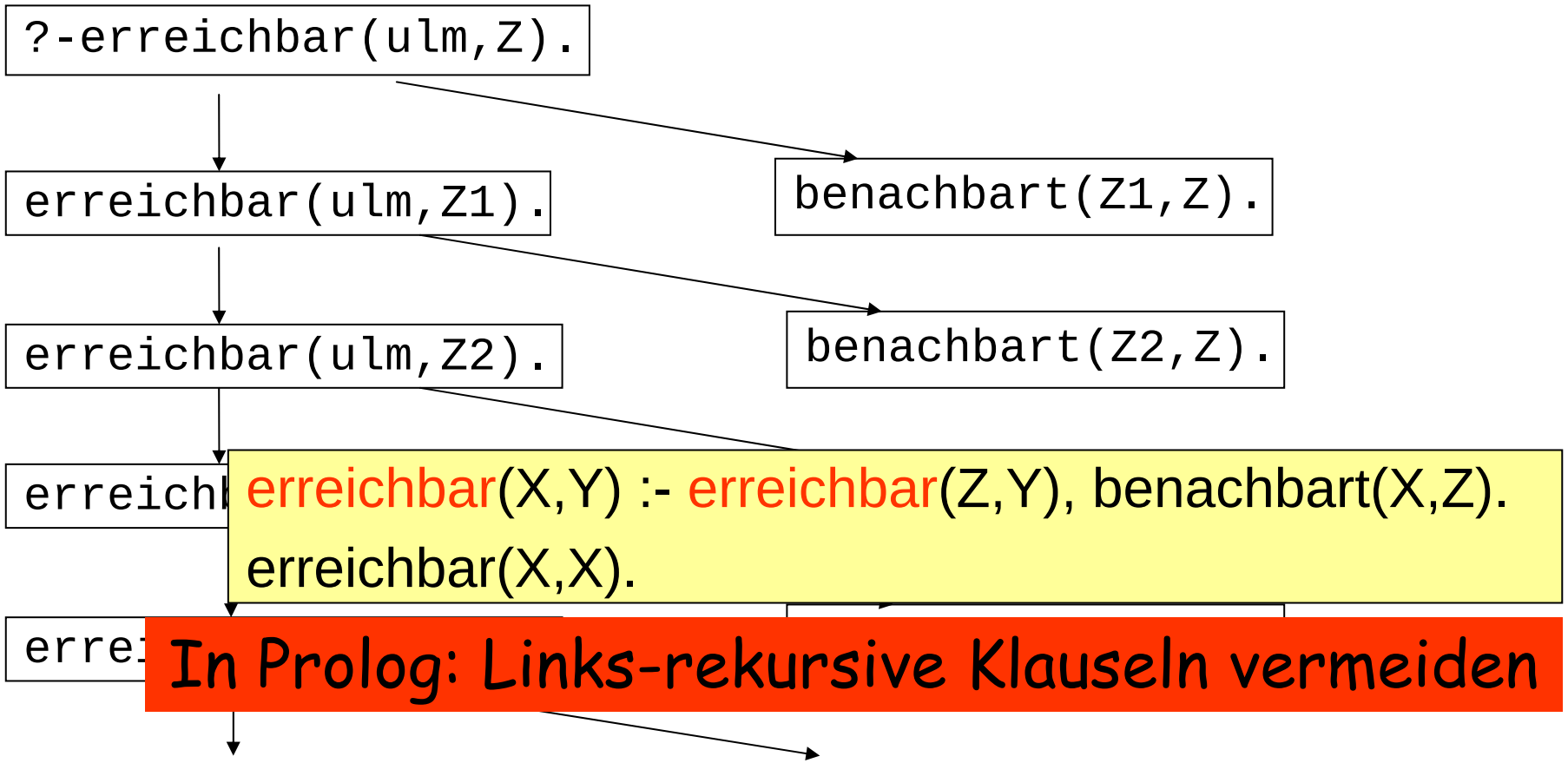
```
erreichbar(X,Y):-erreichbar(X,Z),benachbart(Z,Y).
```



Deklarative vs.prozedurale Semantik

Unterschiedliche Resultate bei

deklarativer und prozeduraler Semantik



Deklarative vs.prozedurale Semantik

Januskopf von Prolog:
Unterschiedliche Resultate
bei deklarativer und prozeduraler Semantik

Reihenfolge der Beweisversuche
    (Auswirkungen z.B. auf rekursive Klauseln, Negation)
(zusätzliche) Arithmetik
Abhängigkeit von „Seiteneffekten“,
        z.B. Eingabe-/Ausgabe-Abhängigkeit
Eingriffe in Beweisversuche (cut)
Meta-logische Prädikate
Programm-Modifikation

Eingriff in die Abarbeitung: Cut

Verändern der Suchstrategie: Prädikat **`!/0`** (*Cut*)

`!/0` gelingt stets und löscht Choice-Points für

- aktuelle Klausel
- subgoals im Klauselkörper, die vor dem Cut stehen
- subgoals dieser subgoals usw.

Folge:

Gefundene Lösung wird ,,eingefroren“

Alternativen für Backtracking entfallen

Eingriff in die Abarbeitung: Cut

```
prinz(X):-grandchild(X,cronus),male(X).
```

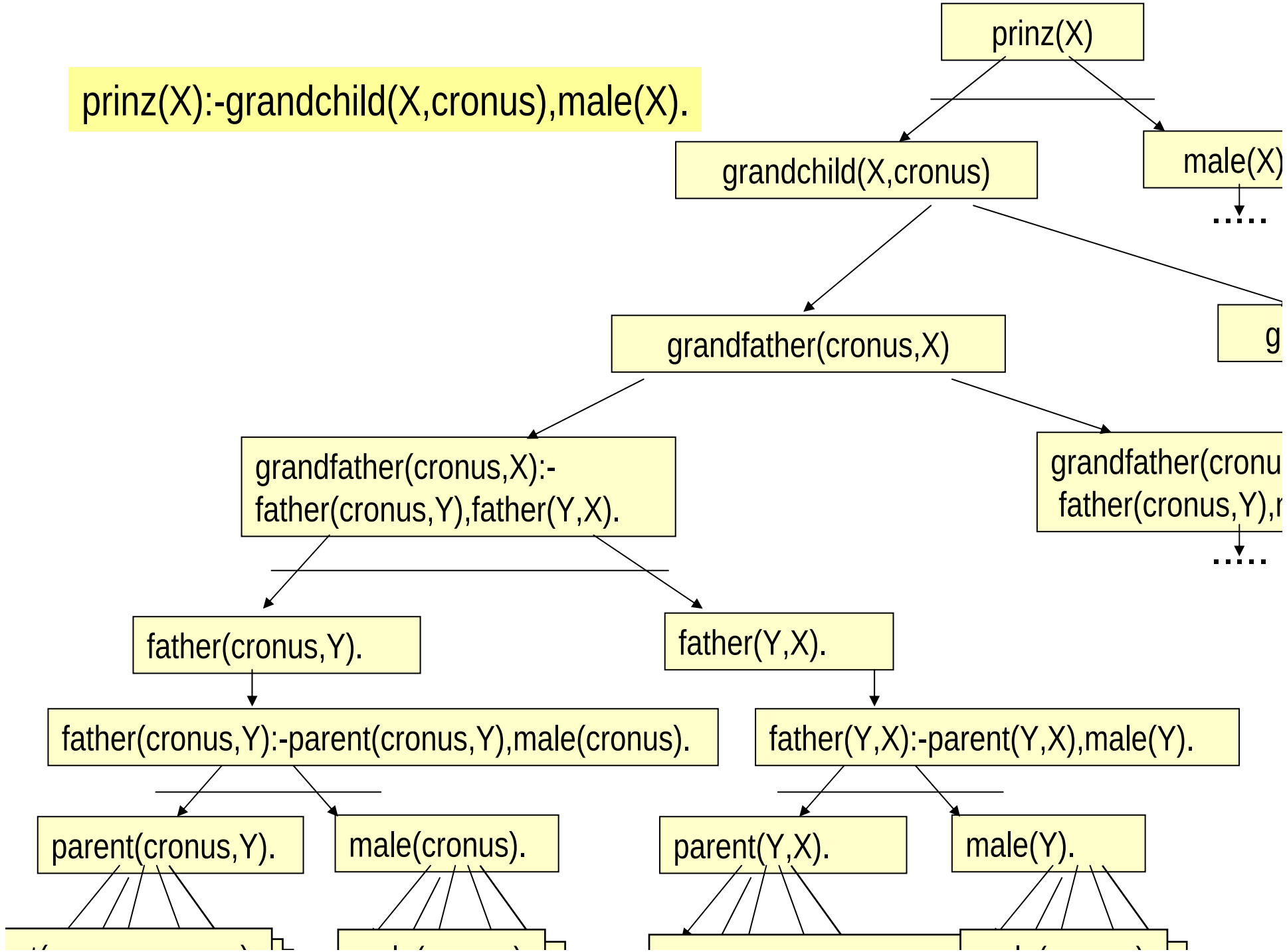
```
?- prinz(X).
X = hermes ;
X = hephaestus ;
X = apollo ;
X = hephaestus ;
no
```

```
kronprinz(X):-grandchild(X,cronus),male(X),!.
```

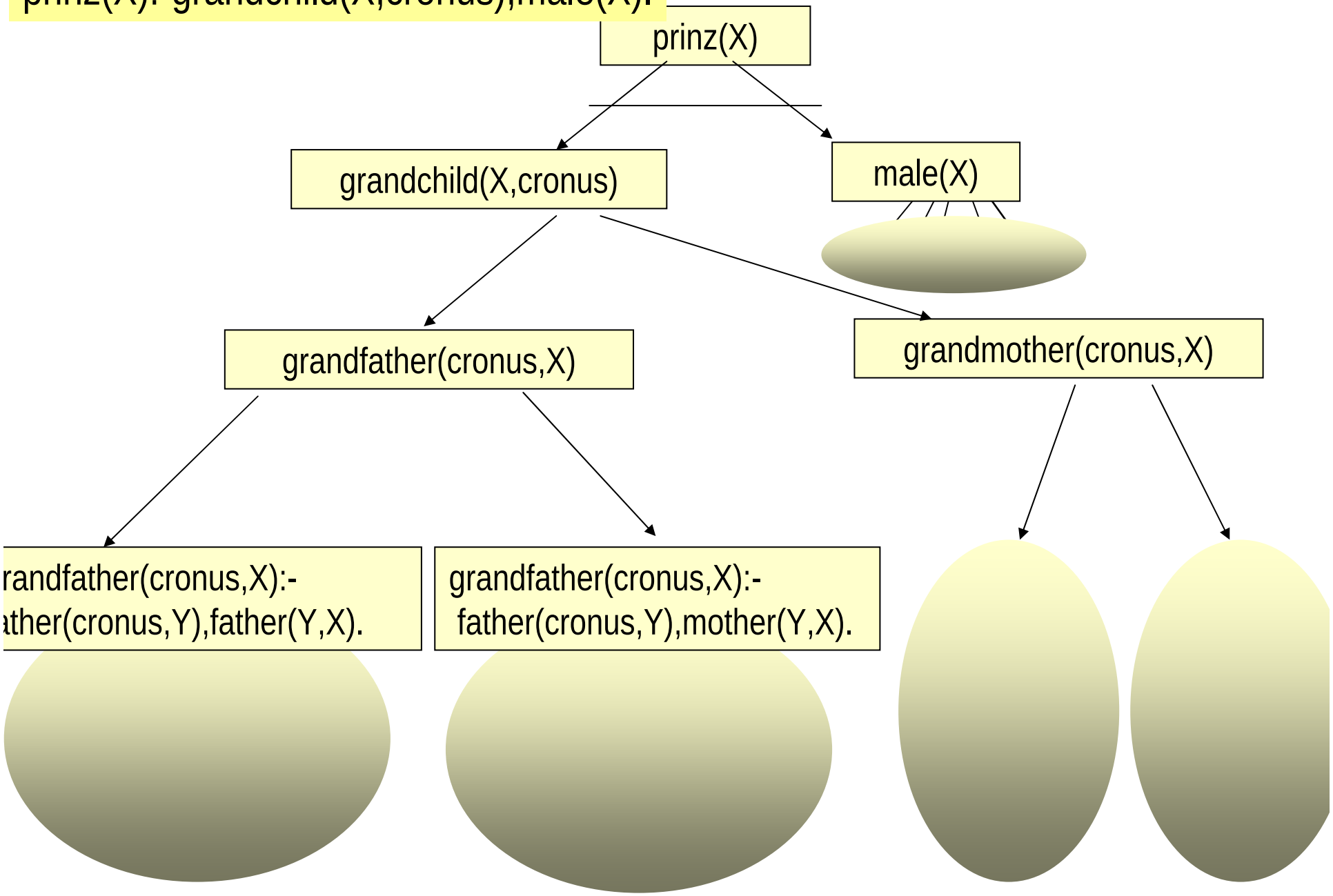
```
?- kronprinz(X).
X = hermes ;
no
```

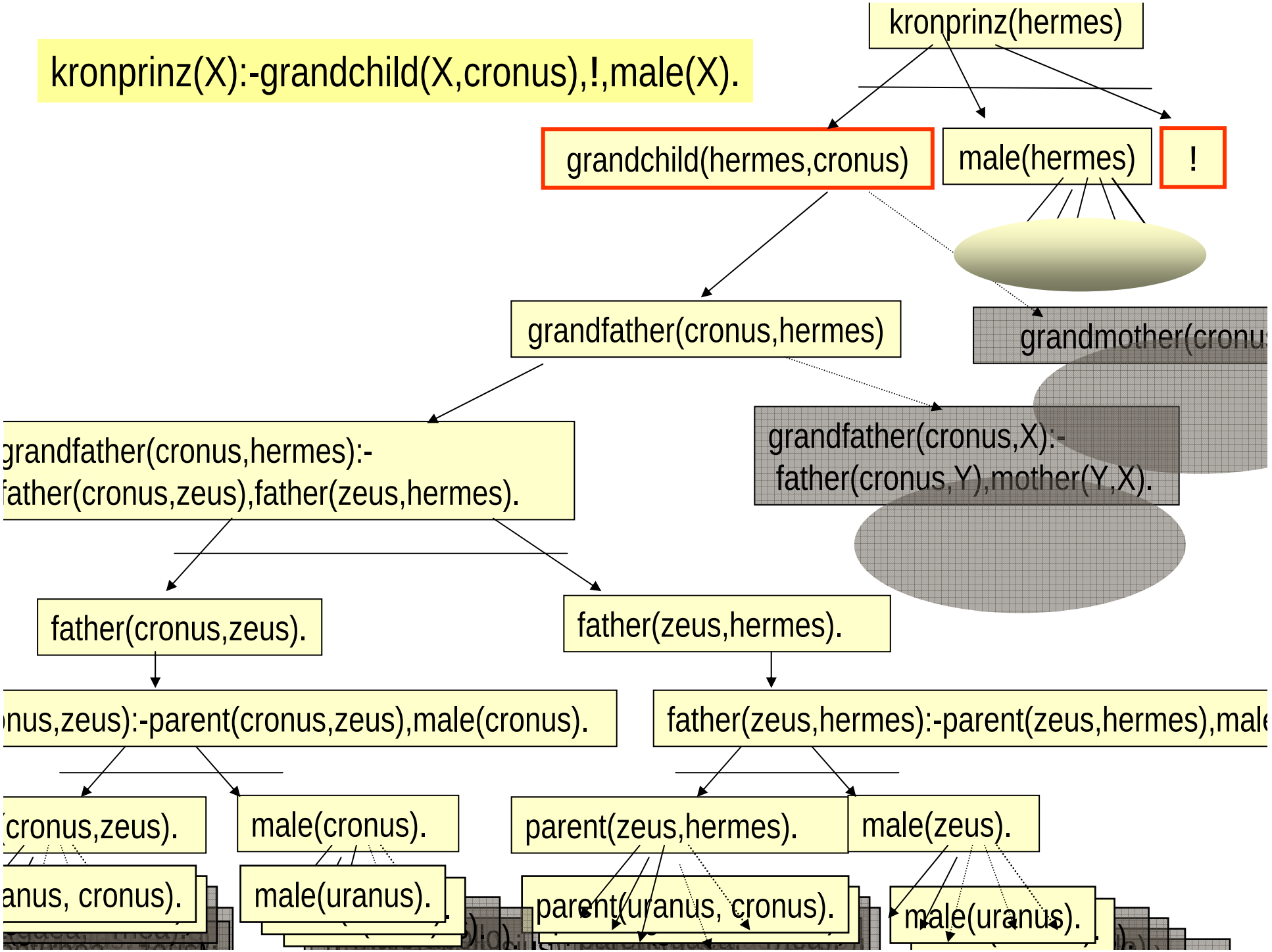
Worin besteht der inhaltliche Unterschied:

```
kronprinz(X):-grandchild(X,cronus),!,male(X) .
```


prinz(X):-grandchild(X,cronus),male(X).
prinz(X)
grandchild(X,cronus)
male(X)
grandfather(cronus,X)
grandfather(cronus,X):-
father(cronus,Y),father(Y,X).
father(cronus,Y).
father(Y,X).
father(cronus,Y):-parent(cronus,Y),male(cronus).
father(Y,X):-parent(Y,X),male(Y).
parent(cronus,Y).
male(cronus).
parent(Y,X).
male(Y).

prinz(X):-grandchild(X,cronus),male(X).




kronprinz(X):-grandchild(X,cronus),!,male(X).
kronprinz(hermes)
grandchild(hermes,cronus)
male(hermes)
!
grandfather(cronus,hermes)
grandmother(cronu
grandfather(cronus,hermes):-
father(cronus,zeus),father(zeus,hermes).
grandfather(cronus,X):-
father(cronus,Y),mother(Y,X).
father(cronus,zeus).
father(zeus,hermes).
nus,zeus):-parent(cronus,zeus),male(cronus).
father(zeus,hermes):-parent(zeus,hermes),male
cronus,zeus).
male(cronus).
parent(zeus,hermes).
male(zeus).
anus, cronus).
male(uranus).
parent(uranus, cronus).
male(uranus).

Cut zur Beschleunigung

(unnötige) Alternativen vermeiden

```
maximum1(X,Y,X) :-  X >= Y, !.
maximum1(X,Y,Y) :-  X <  Y .
```

Grüner cut: gleiches Resultat ohne Cut

```
maximum2(X,Y,X) :- X >= Y, !.
maximum2(X,Y,Y).
```

Roter cut: anderes Resultat ohne Cut

Fallunterscheidung

```
case(...)  :-  condition-1(...),declaration-1(...).
case(...)  :-  condition-2(...),declaration-2(...).
...
case(...)  :-  condition-n(...),declaration-n(...).
```

- Mit Backtracking, ggf. weitere Regel bearbeiten

```
case(Geld,Essen) :- Geld>500,adlon(Essen).
case(Geld,Essen) :- Geld>50,steakhouse(Essen).
case(Geld,Essen) :- Geld>5,doener(Essen).
case(Geld,Essen) :- selberkochen(Essen)
```

Fallunterscheidung mit cut

```
case(...) :- condition-1(...), !, declaration-1(...).
case(...) :- condition-2(...), !, declaration-2(...).
...
case(...) :- condition-n(...), !, declaration-n(...).
```

- Ohne Backtracking, höchstens eine Regel bearbeiten

```
case(Geld,Einladung) :- Geld>500, !, adlon(Einladung).
case(Geld,Einladung) :- Geld>50,  !, steakhouse(Einladung).
case(Geld,Einladung) :- Geld>5,   !, doener(Einladung).
case(Geld,Einladung) :-           !, selberkochen(Einladung)
```

„If then else"

CWA (2)

Wann soll Interpreter Antwort „no“ auf Anfrage Q liefern?

Logische Varianten:
Wenn ¬Q bewiesen wurde.
Wenn Q nachweisbar nicht bewiesen werden kann.
Wenn alle Beweisversuche für Q fehlgeschlagen sind.
Dabei jeweils implizite Annahmen bzgl. Negation.

Nicht-logische Varianten:
Wenn die verfügbaren Argumente für „nicht Q“ sprechen.
Wenn die verfügbaren Argumente gegen Q sprechen.
Im Zweifelsfall für den Angeklagten ...

CWA (2)

In Prolog:

Variante 3 **„Negation by (finite) failure“**

Wenn alle Beweisversuche für Q fehlgeschlagen sind.

Bedeutung der Antwort „no“:
Alle Beweisversuche sind fehlgeschlagen.

Probleme:

In PK1: Kein allgemeines Verfahren

(Nicht-Allgemeingültigkeit ist nicht aufzählbar)

Unterschiede zur logischen Negation

Negation by failure

Was würde bedeuten:

Wenn H nicht aus Programm P beweisbar ist,
wird angenommen, dass ¬H beweisbar ist.

Unterstellung von

„Genau eins von beiden, H oder ¬H , ist aus P beweisbar.“

bedeutet in der Logik

Vollständigkeit (im Sinne der klassischen Logik!)

– für alle Aussagen H gilt: H oder ¬H ist aus P beweisbar

Korrektheit

– für alle Aussagen H gilt:
H und ¬H sind nicht beide aus P beweisbar

Negation by failure

„Genau eins von beiden, H oder ¬H , ist aus P beweisbar.“

Trifft in Prolog i.a. nicht zu:

Weder male(fritz)

noch ¬male(fritz)

folgen logisch aus

unserem Programm

```
parent(gaea,
parent(gaea,
parent(rhea,
parent(cronus
parent(rhea,
parent (cronu
parent (cronu
parent(rhea,
parent(cro
parent(rhe
```

```
female(gaea).
female(rhea).
female(hera).
female(hestia).
female(demeter).
female(athena).
female(metis).
```

```
male(uranus).
male(cronus).
male(zeus).
male(hades).
male(hermes).
male(apollo).
male(dionysius).
male(hephaestus).
male(poseidon).
```

Inhaltlich ist Verfahren von Prolog oft sinnvoll

Operator **not** : Negation by failure

Ein subgoal not(p1(X1...,Xn)) gelingt in Prolog,

falls p1(X1...,Xn) nicht bewiesen werden kann.

```
?- not(male(fritz)).
    yes
```

```
male(X):- not(female(X)).
```

```
?- male(rhea).
   no
```

Nicht erlaubt: negierter Klauselkopf not (p) :- ...

not mittels cut implementierbar

Kombination von cut und fail:

```
verschieden(X,Y) :- X=Y, !, fail.
verschieden(X,Y) .

male(X) :- female(X), !, fail.
male(X) .
```

not/1 wäre implementierbar durch

```
not(P)  :-  P,  ! , fail.
not(P) .
```

Negation by failure: Diskussion

Vollständiger Kalkül müsste negative Prädikate in gleicher Weise wie positive Prädikate behandeln können:

- In Klausel g :- $g_1,...,g_n$ dürften als subgoals g_i
  positive Literale $p(X1,...,Xn)$ oder
  negative Literale $\neg p(X1,...,Xn)$ auftreten.
- Beide Formen mit der Interpretation dass Belegungen der Variablen gesucht werden, die g_i erfüllen

Tatsächlich aber nur für positive Literale realisiert.

not $(p(X1,...,Xn))$ bedeutet dagegen,
dass es keine Belegung gibt, die $p(X1,...,Xn)$ erfüllt.

Negation by failure: Diskussion

Unterschied:

¬Q(X1,...,Xn) beweisen

Q(X1,...,Xn) nicht beweisen können

Insbesondere auch:

Misserfolg von „not(Q)“ ohne Backtracking in Q .

Negation by failure: Diskussion

Veränderte Semantik:

```
a:-not(b).
b.
```

```
?-not (a).
yes
```

Aber $\neg$ a folgt nicht aus $\{\neg b \rightarrow a,\ b\}$

```
a:-not(b).
```

```
?-a.
yes
```

Aber a folgt nicht aus $\{\neg b \rightarrow a\}$

Negation by failure: Diskussion

Man kann positive/negative Fakten separat benennen:
Beispiel: `female` und `male` (= nicht `female`)

Aber kein logischer Zusammenhang zwischen beiden:

Anfragen

```
?- female(fritz).
?- male(fritz).
```

führen beide zu „`no`“ , falls `fritz` nicht in Datenbasis.

Anfrage

```
?- not(female(fritz)).
?- not(male(fritz)).
```

führen beide zu „`yes`“ , falls `fritz` nicht in Datenbasis

Negation by failure: Diskussion

Falls `fritz` nicht in Datenbasis:

```
?- male(fritz).
 no
?- not(male(fritz)).
yes
```

Wäre gleichzeitig definiert

```
male(X) :- not(female(X)).
```

wären alle nicht bekannten Objekte „männlich“:

```
? - male(fritz).
 yes
```

„not“ führt keine Bindungen aus.

Keine Variablenbindung durch not .

Nicht im Sinne existentieller Anfragen verwendbar.

Problem bei Verwendung für nicht gebundene Variable

```
male(X):-not(female(X)).
?-male(X).
no.
```

Solange `female(X)` beweisbar.

Es würde funktionieren bei vorheriger Bindung, z.B.

```
..., human(X),male(X),...
```

Negation by failure: Diskussion

```
male(X) :- not(female(X)).
```

Abarbeitung von `?-male(X)` als $\forall X\,(\neg female(X))$

d.h. $\neg\,\exists X\,(female(X))$

statt $\exists X\,(male(X))$

Unterschiedliche Semantik bei logischer Äquivalenz :

```
r1:-male(X),human(X).
r2:-human(X), male(X).
female(anna).
human(fritz).
```

r1 und r2 sind logisch äquivalent.

?-r1.	?-r2.
no	yes

CWA (2): Negation by **finite** Failure

not(Q) gelingt (Antwort „yes“) , falls Q misslingt (Antwort „no“)

not(Q) misslingt, (Antwort „no“), falls Q gelingt (Antwort „yes“)

Wann kann Interpreter Antwort „not(Q)“ bestätigen?

Gemäß *Variante 3*:

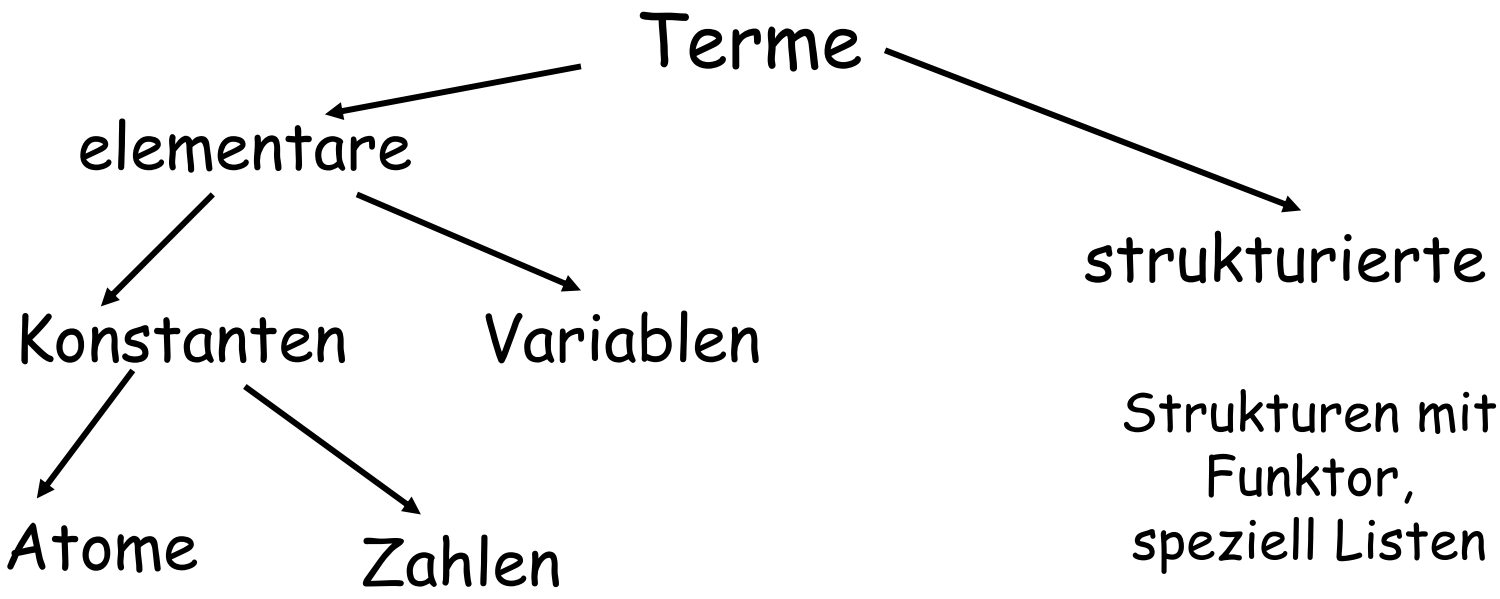
Wenn alle Beweisversuche für Q fehlgeschlagen sind.

Alle Beweisversuche müssen nach endlicher Zeit geprüft sein. Nicht möglich bei unendlichem Und-oder-Baum.

Finite failure

Bedeutung der Antwort „no“:
Alle Beweisversuche sind fehlgeschlagen.

Prolog Syntax: Terme



Prolog Syntax: Terme

Atome: beginnen mit kleinen Buchstaben

```
thomas, lisa, robert
```

Zeichenketten: in Hochkommata eingeschlossen

```
`Thomas`, `Lisa`, `Robert`
```

Zahlen: natürliche und reelle Zahlen

```
26,  5.7E-3, -1.25
```

Variable:

beginnen mit großem Buchstaben oder Unterstrich

```
X, Thomas, _21,
_   (anonyme Variable)
```

Prolog Syntax: Terme

zusammengesetzte Terme (Strukturen):

kleingeschriebener Funktor, Argumente

– *Funktor* charakterisiert durch Name + Stelligkeit:
`name/i`
Unterschied: `punkt(3,4)`, `punkt(12,5,7)`

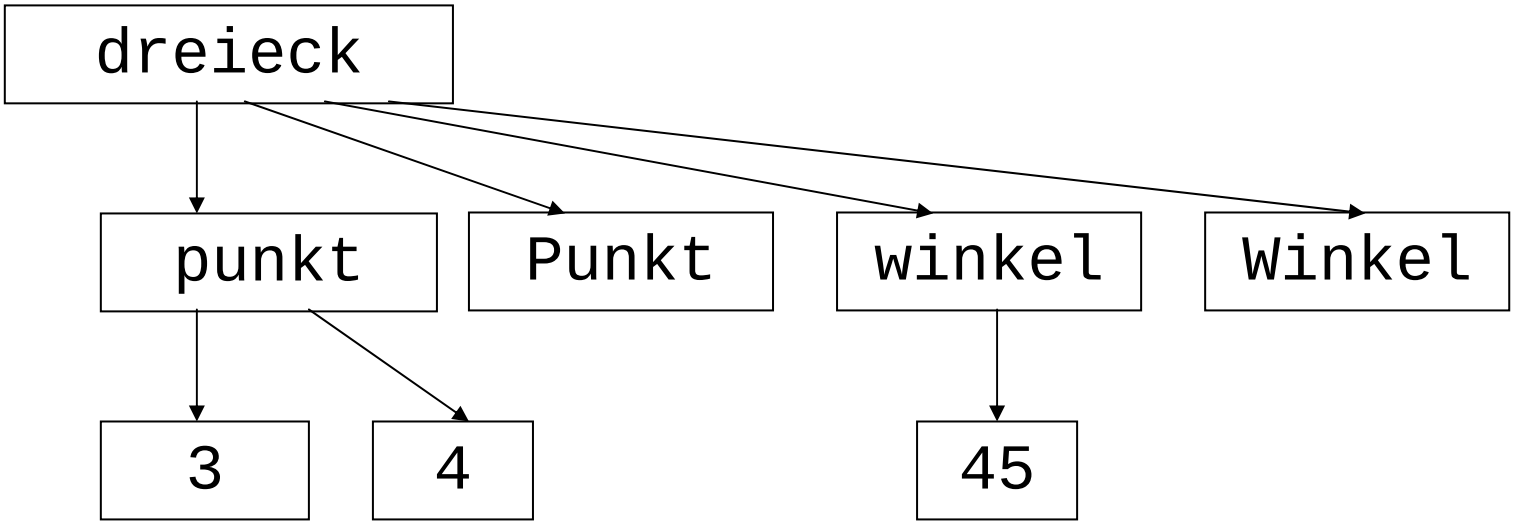
– Argumente sind Terme (rekursiv!)

`punkt(3,4)`, `dreieck(X,Y,Z)`,
`dreieck(punkt(3,4),Punkt,winkel(45),Winkel)`

Prolog Syntax: Terme

Darstellung von Strukturen als Bäume.

```
dreieck(punkt(3,4),Punkt,winkel(45),Winkel)
```

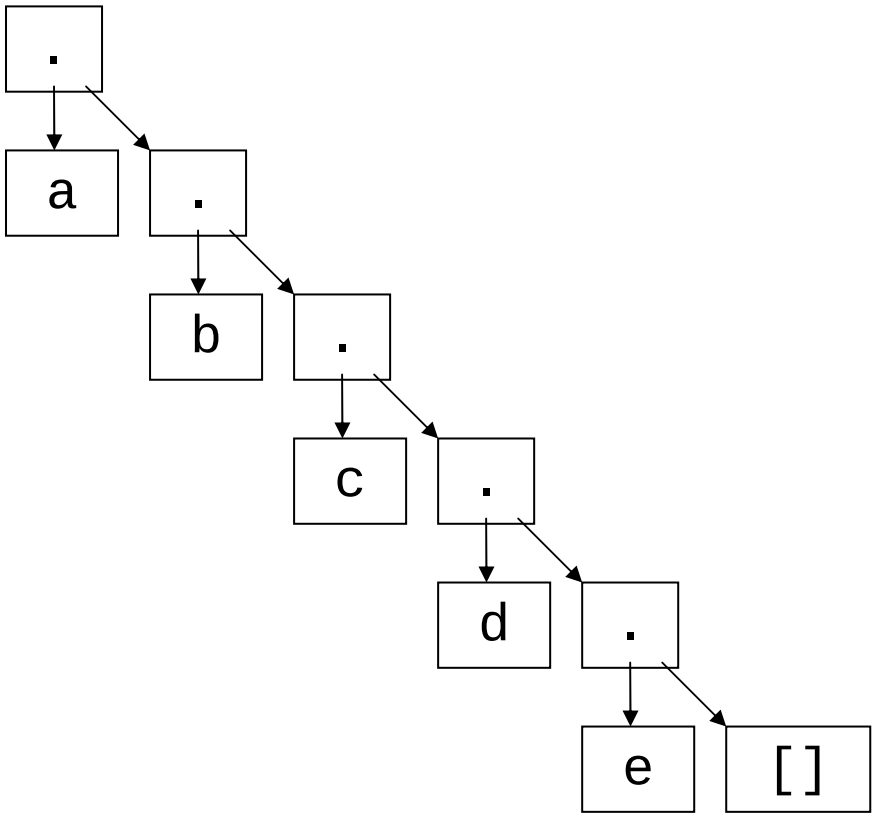


Prolog Syntax: Terme

Darstellung von Strukturen als Bäume.

[a,b,c,d,e] = [a,[b,[c,[d,[e,[]]]]]] = .(a, .(b, .(c, .(d, .(e, [])))))

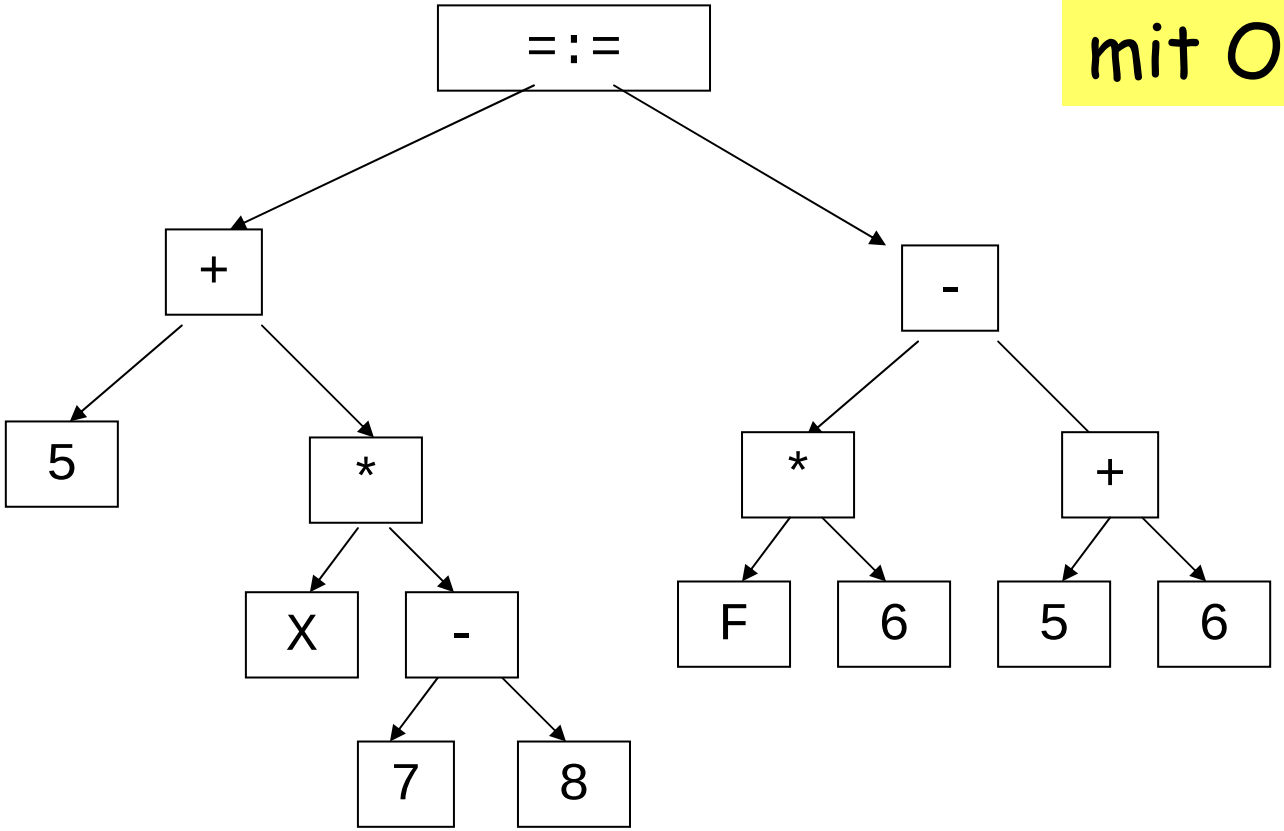
Spezielle Notationen für Listen



Prolog Syntax: Terme

5 + X * (7 – 8) =:= F * 6 - (5 + 6)

Spezielle Notationen mit Operatoren



=:=(+(5, *(X, -(7,8))), -(*(F,6), +(5,6)))

Prolog-Syntax: Programm

Programm: Menge von Prozeduren

Prozedur: Folge von Klauseln mit gleichem Kopf-Funktor

Klauseln: Fakten oder Regeln

Fakt: Prädikat

Regel: Kopf :- Körper

Kopf: Prädikat

Körper: Folge von Teilzielen

Teilziel: Prädikat

Prädikat: Funktor-Name und Liste von Argumenten

Argumente: Terme

Parameter unterscheiden in
- Eingabe-Parameter
- Ausgabe-Parameter
- beliebig

Fakten und Regeln sind auch Strukturen

Prolog-Syntax: Programm

Die Antwort auf alle Fragen erhält man mit

```
? – X .
```

Listen

Endliche geordnete Menge von Elementen
(endliche Folge, Sequenz, Wort)

Schreibweise: [a,b,c,d,e]
[*2, 34, 7, 13, 4, 2, 3, 13*]
[*S*, *t*, *u*, *d*, *e*, *n*, *t*, *i*, *n*, *n*, *e*, *n*]

Mathematisch definierbar als
Abbildung L von {1,...,n } in Elemente-Menge

L = [*L*(1), *L*(2), ..., *L*(*n*)]

oder als ineinander verschachtelte Mengen:
{ *L*(1), {*L*(2), { ..., {*L*(*n*), { } } ... }}}

Listen

Operationen (Methoden) mit Listen:

- Verketten von Listen
- Zugriff auf Elemente
  - einfügen
  - suchen
  - löschen
- Reorganisation (z.B. Sortieren)
- Anzahl der Elemente

Listen

Implementation z.B. als

- Feld von Elementen
- Einfach verkettete Liste: Referenz auf Nachfolger
- Doppelt verkettete Liste: vorwärts-/rückwärts-Referenz

in JAVA z.B.:

- Klasse `String` für Zeichenketten:
  Liste implementiert als Feld von Zeichen: `char[]`
- Klasse `Vector` für Listen allgemein

Listen

Rekursive Definition

- Die leere Liste (NIL oder []) ist eine Liste.
- L ist eine Liste, wenn sie ein erstes Element (**head**) besitzt und der Rest (**tail**) wieder eine Liste ist.

liste([ ]).

liste(L) falls liste(tail(L)).

head([a,b,c,d,e]) = a tail([a,b,c,d,e]) = [b,c,d,e]

[a,b,c,d,e] = [a,[b,c,d,e]] = [a,[b,[c,d,e]]] = [a,[b,[c,[d,e]]]]

= [a,[b,[c,[d,[e]]]]] = [a,[b,[c,[d,[e,[]]]]]]

Listen in Prolog: Funktor „ ./2 “

Struktur **.** (Element, Liste)

dient zur rekursiven Beschreibung von Listen:

. (Kopf, Restliste) beschreibt die Liste [Kopf, ...(Rest)...]

.(a,.(b, .(c, .(d, .(e, []))))) = [a,b,c,d,e]

Prolog erlaubt weitere Schreibweisen:

[Kopf | Restliste]

[Element_1, ..., Element_n]

[Element_1, ..., Element_i | Restliste_ab_i+1]

[a,b,c,d,e]

= [a | [b,c,d,e]]

=[a,b | [c,d,e]]

= ...

=[a,b,c,d,e | []]

member-Prädikat

```
member(X, [X | T] ).
member(X, [H | T] ) :- member(X, T) .
```

?- member(a, liste). Ist a Element von liste?

?- member(X, liste). Welche Elemente hat liste?

?- member(a, L). Welche Listen besitzen a als Element?

Variante für einmalige Antwort:

```
member(X, [X | T] ) :- ! .
member(X, [H | T] ) :- member(X, T) .
```

append-Prädikat

```
append([ ], L, L ).
append([X | L1], L2, [X | L3] ) :- append(L1, L2, L3).
```

?- append(liste1, liste2, L3).

?- append(liste1, L2, liste3).

?- append(L1, liste2, liste3).

?- append(L1, L2, liste3).

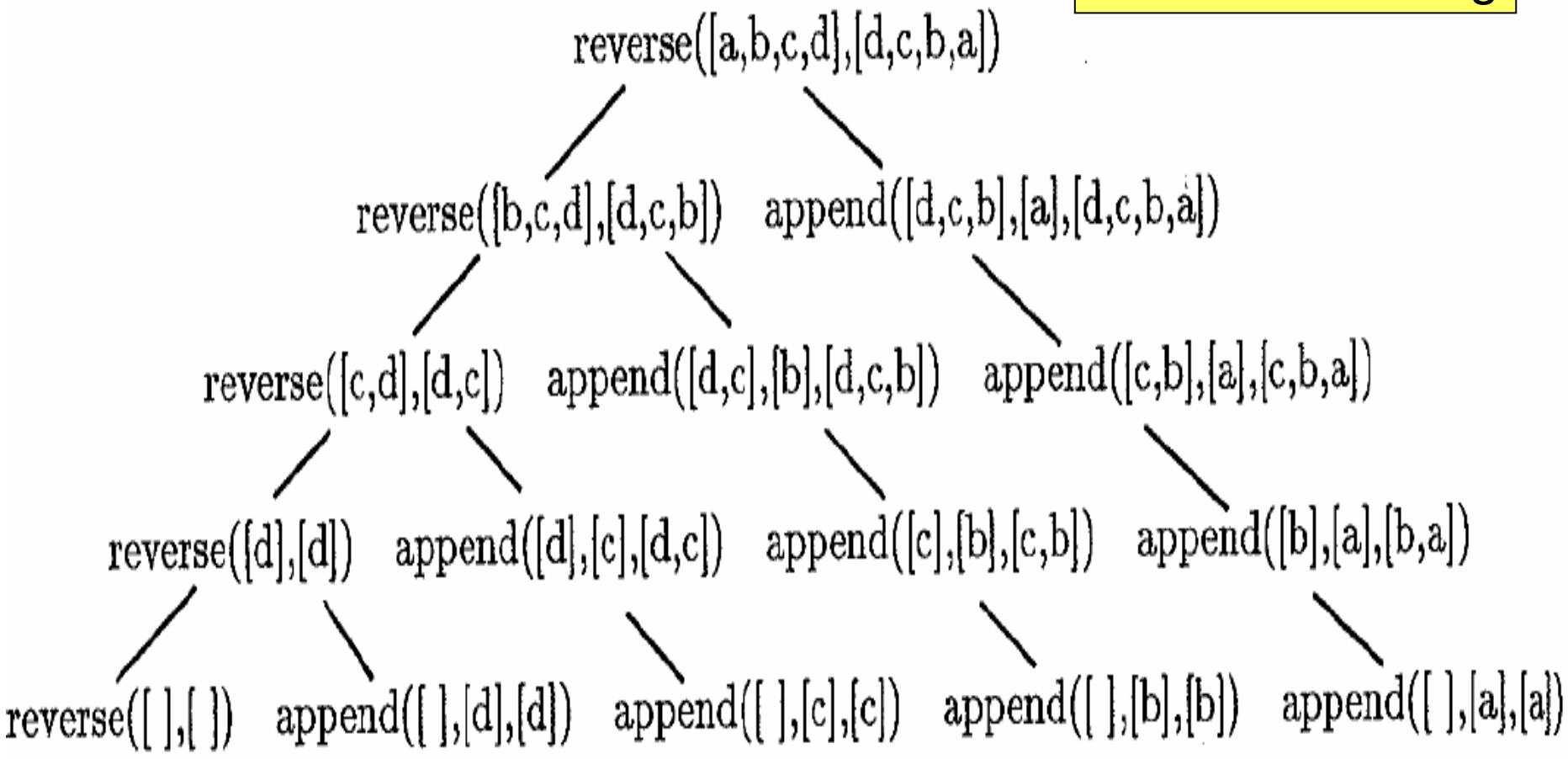
Anwendungen für append-Prädikat

```
prefix(P, L )  :-  append(P,L2,L).
suffix(S, L )   :-  append(L1,S,L).
sublist(SL,L ) :- prefix(P,L), suffix(SL,P).

member(X,L) :- append(P,  [X | S], L).

naive_reverse([ ],[ ]) .
naive_reverse( [H | T], R) :- naive_reverse(T,R1),
                                append(R1,[H], R).
```

Naive Reverse

Aus Shapiro:

The Art of Prolog



Anwendungen für append-Prädikat

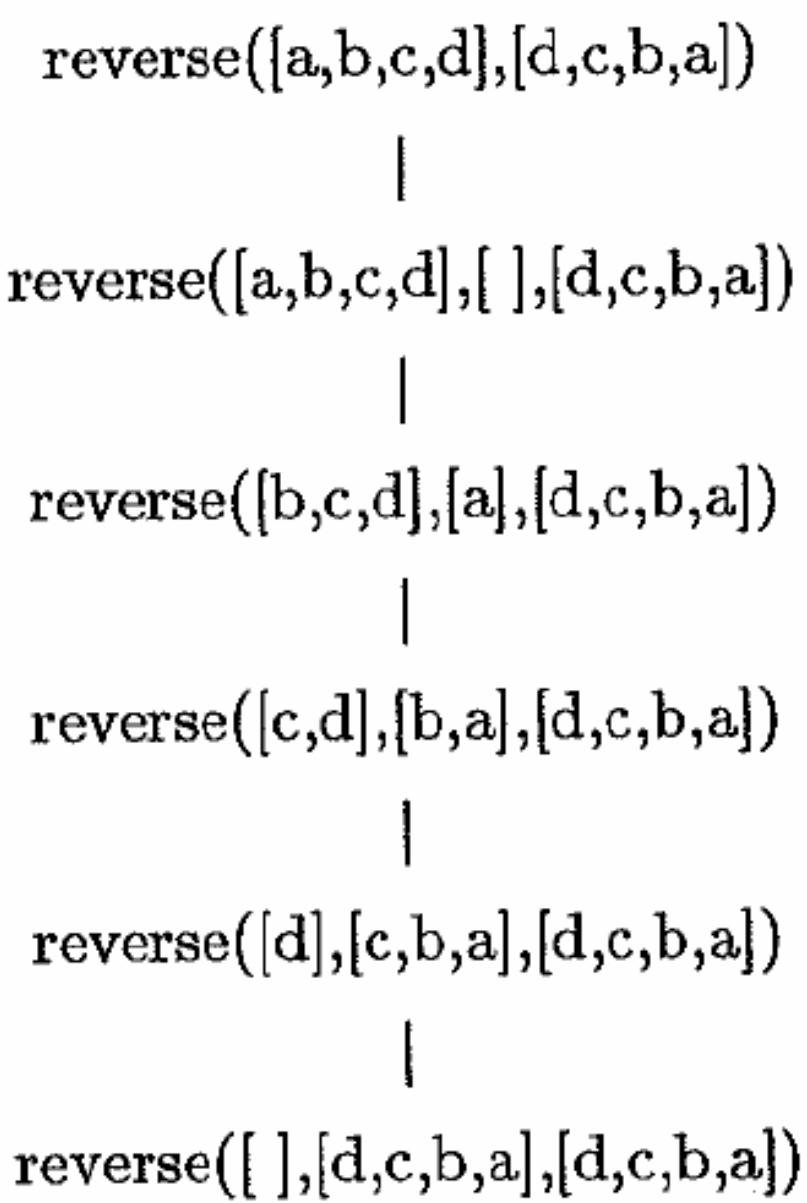
Effizientere Implementation für reverse

Mit Hilfe eines „Akkumulators“ (Zwischenspeicher):

```
reverse( L,R ) :- reverse_acc( L, [ ], R ).

reverse_acc( [ H| T], Acc, R ) :- reverse_acc( T, [H | Acc] , R ) .
reverse_acc( [ ], R, R ).
```

Reverse mit Akkumulator

Aus Shapiro:
The Art of Prolog

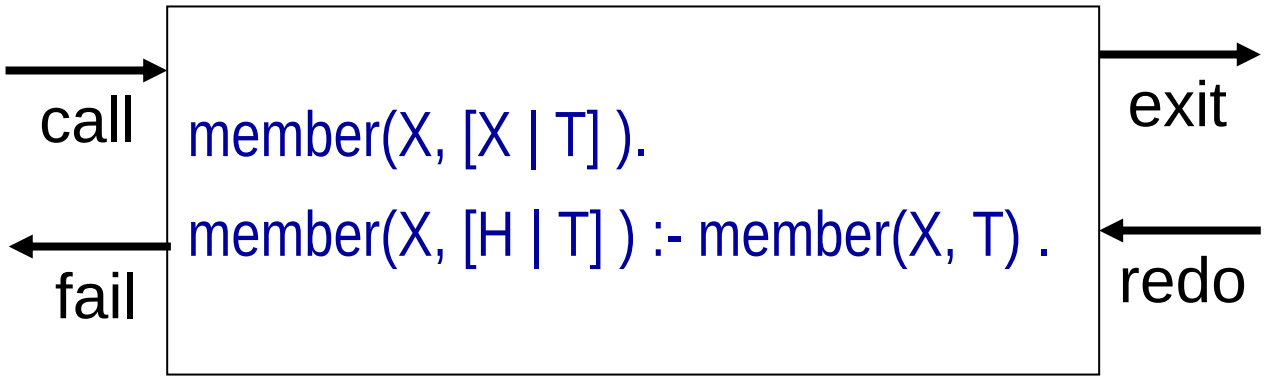


Box-Modell, trace/0

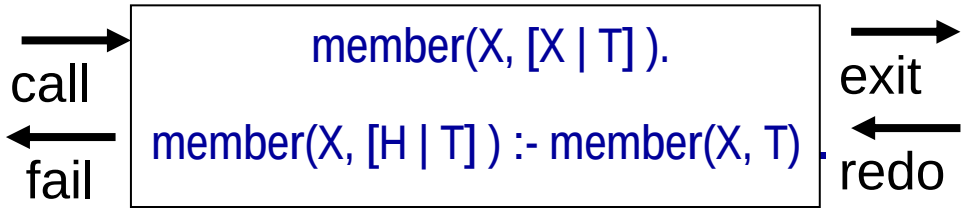
trace/0

Verfolgen des nächsten Beweisversuchs
in der Darstellung des Boxmodells.

Box modelliert Bearbeitung einer Prozedur
durch 4 Ports (Ereignisse) für Betreten/Verlassen



Box-Modell, trace/0



Call: Start des Beweises einer Prozedur
(erster Beweisversuch: erste Klausel).

Exit: Erfolgreicher Beweis.

Redo: Alternativer Beweisversuch (Backtracking).

- Backtracking im Beweis der Klausel
- bzw. bei deren endgültigem Fehlschlag:
  Beweisversuch mit nächster alternativer Klausel.

Fail: Keine weiteren Beweismöglichkeiten für Prozedur.

Box-Modell, trace/0

Reihenfolge der Boxen graphisch:

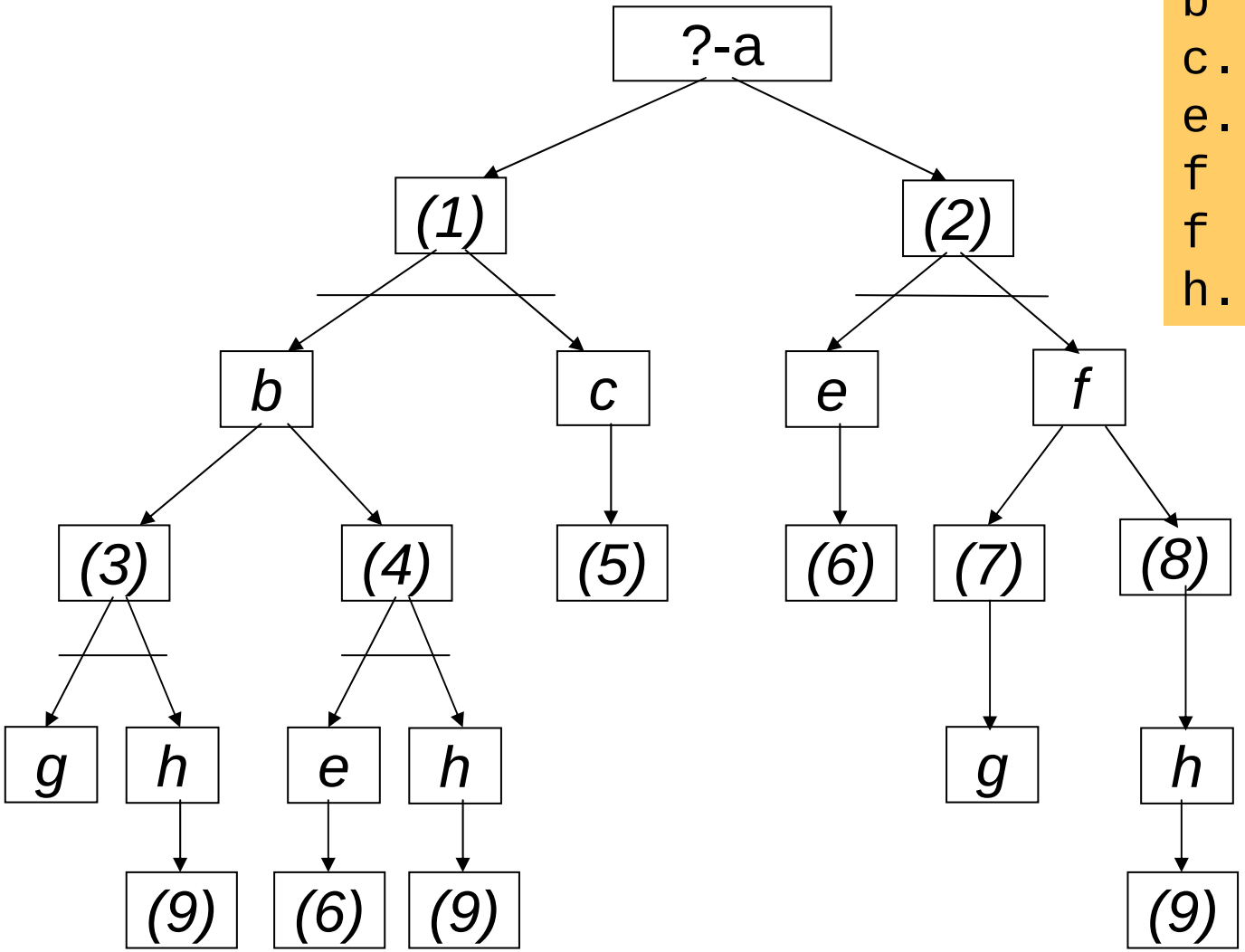
- Subgoals einer Klausel sequentiell
- Klauselaufrufe: ineinander verschachtelt

Reihenfolge der Boxen im trace:

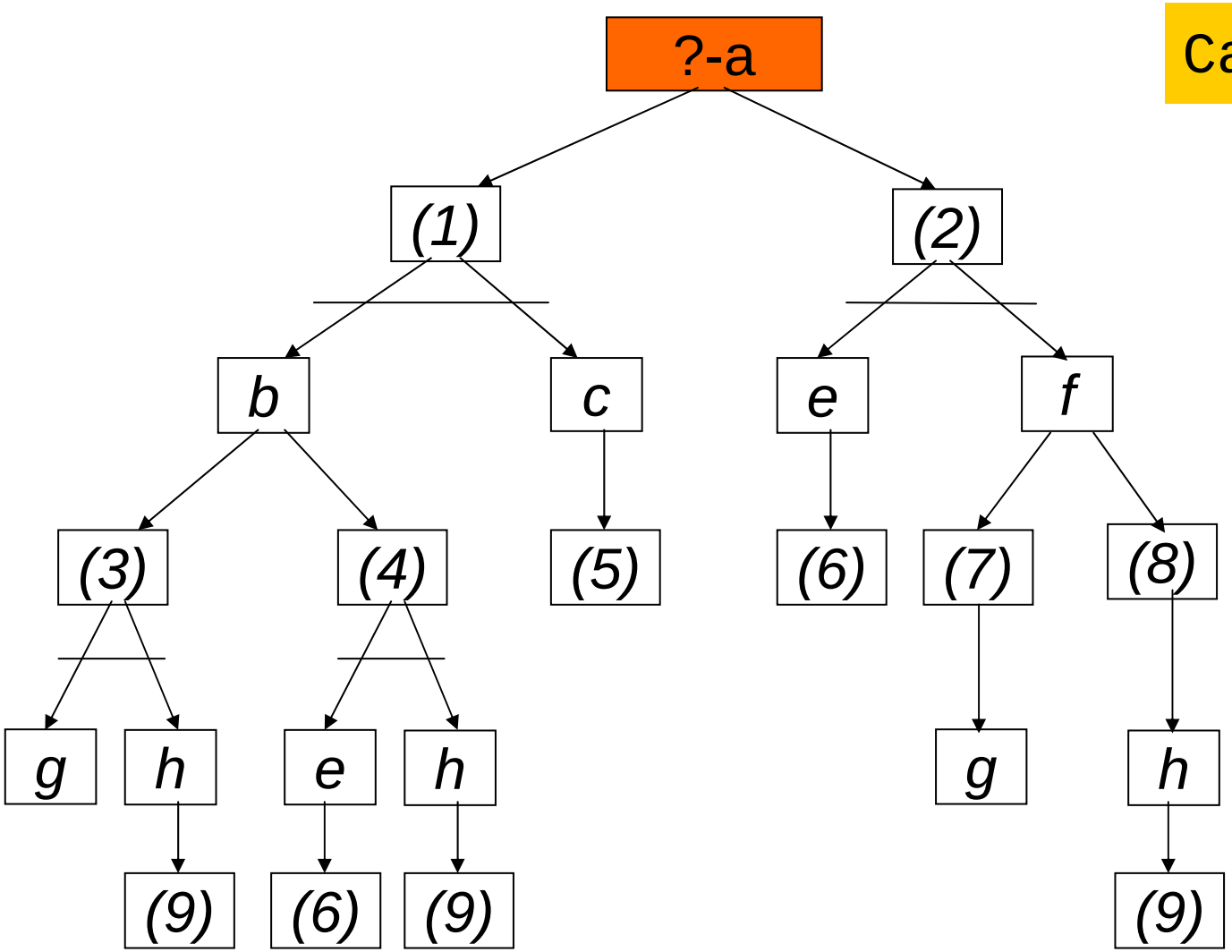
- sequentiell gemäß Abarbeitung im Und-oder-Baum
- mit Nummerierung gemäß Ebenen im Und-oder-Baum

trace

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```

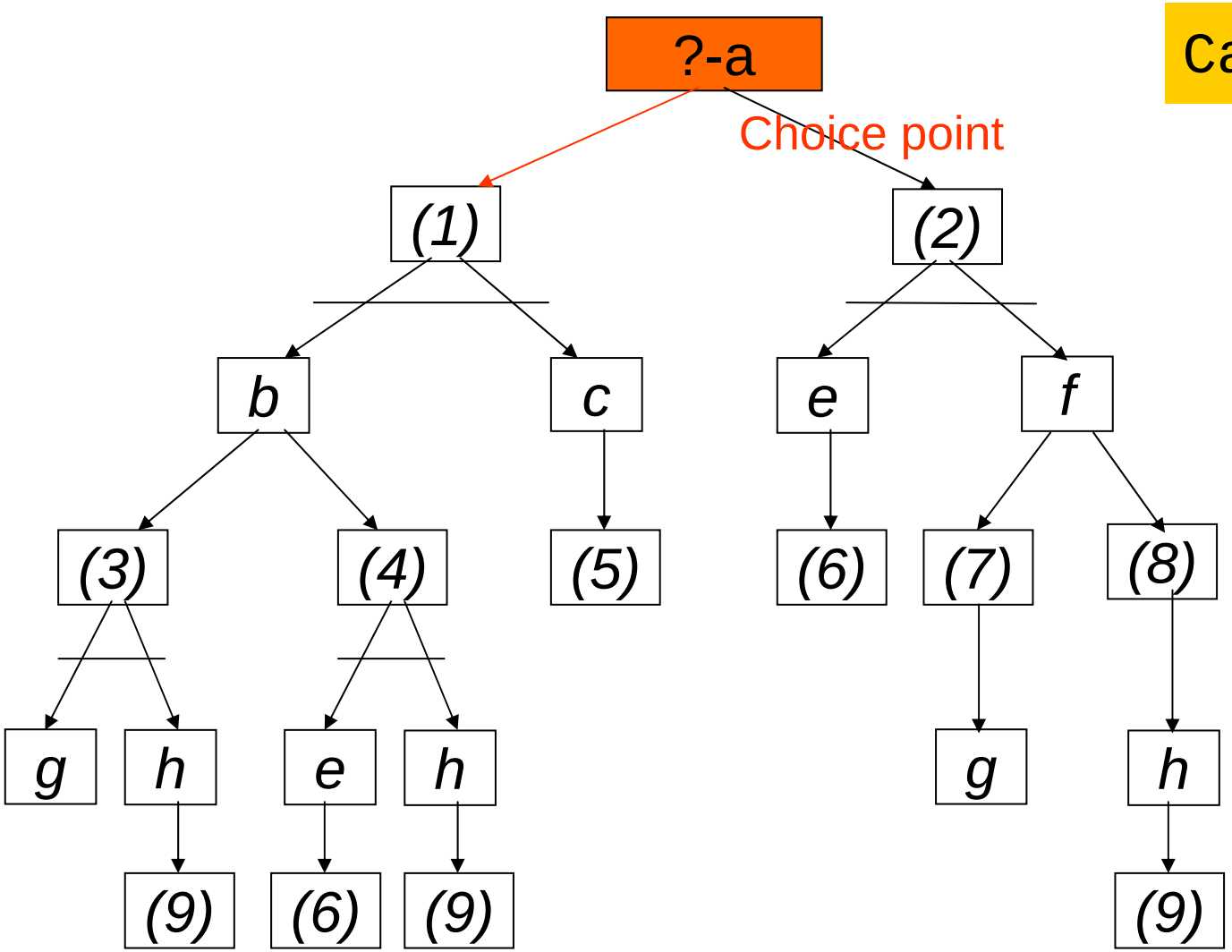


trace



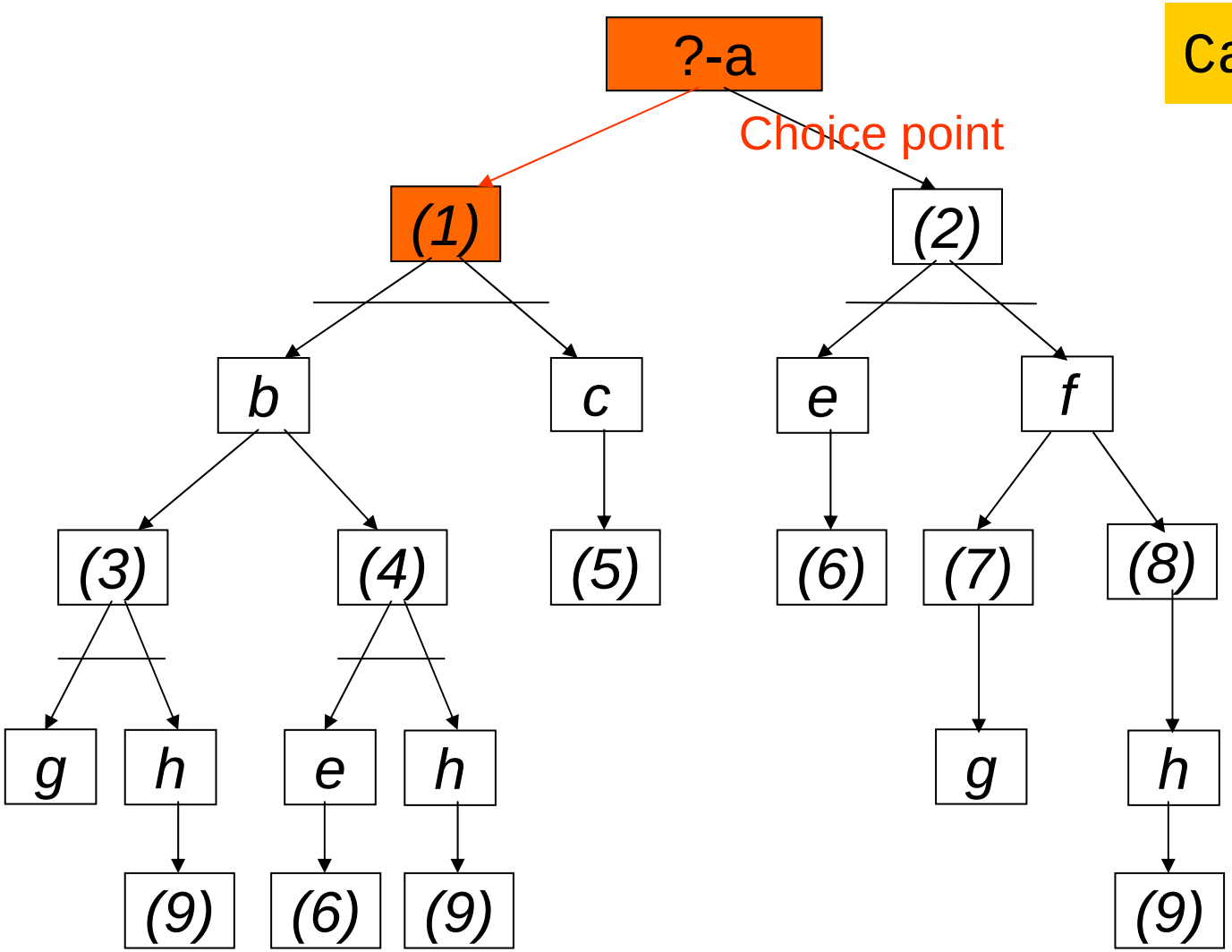

```
Call: (8) a
```

trace



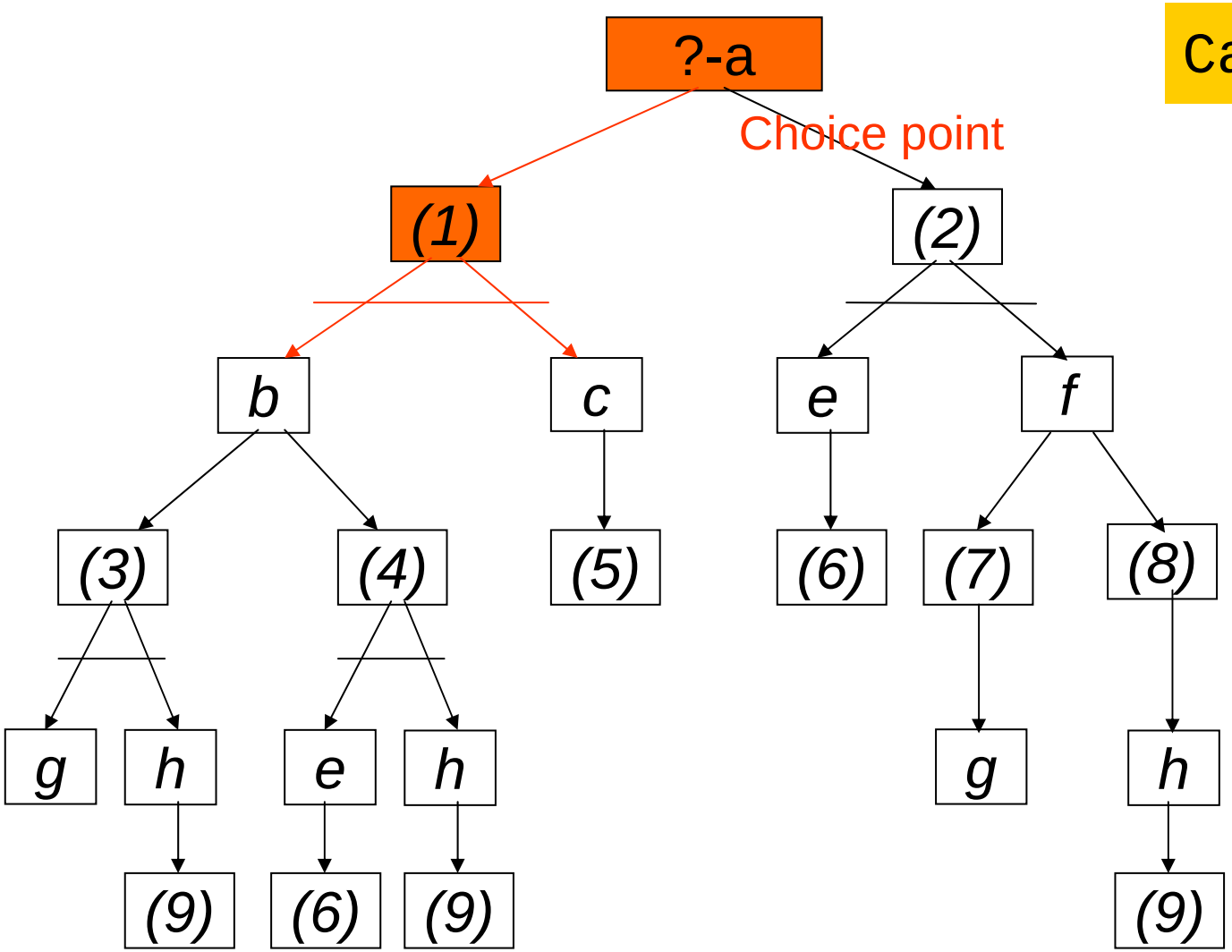

```
Call: (8) a
```

trace



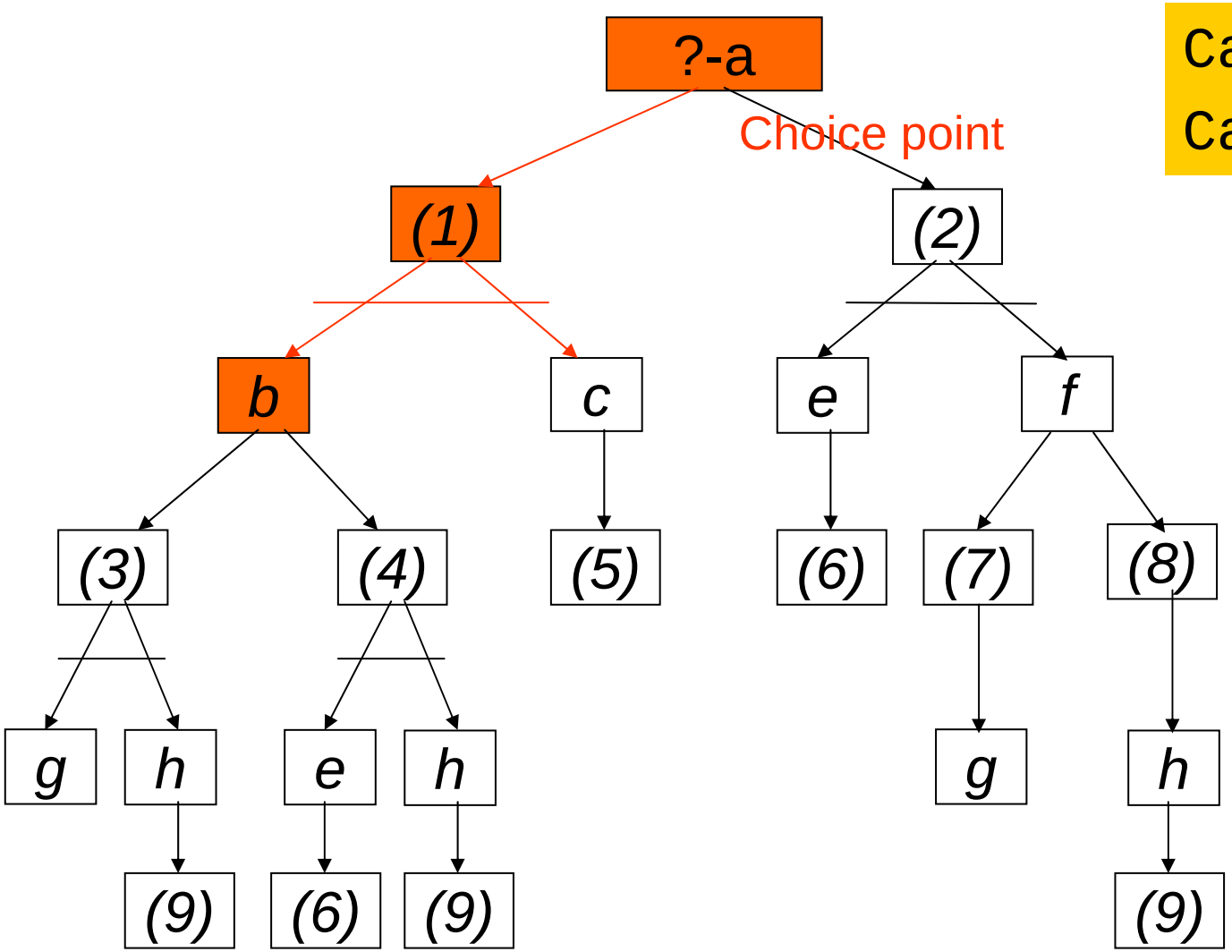

```
Call: (8) a
```

trace



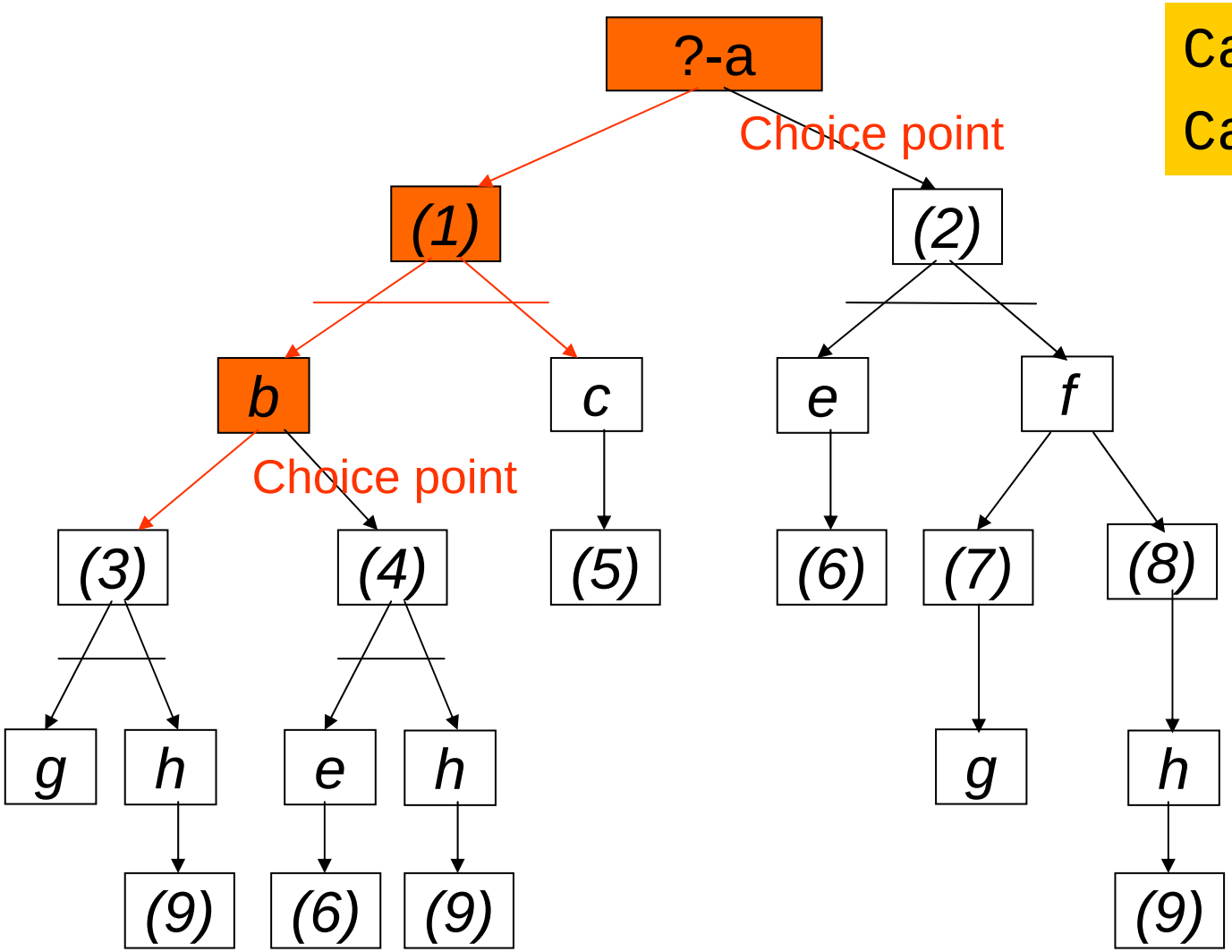

```
Call: (8) a
```

trace



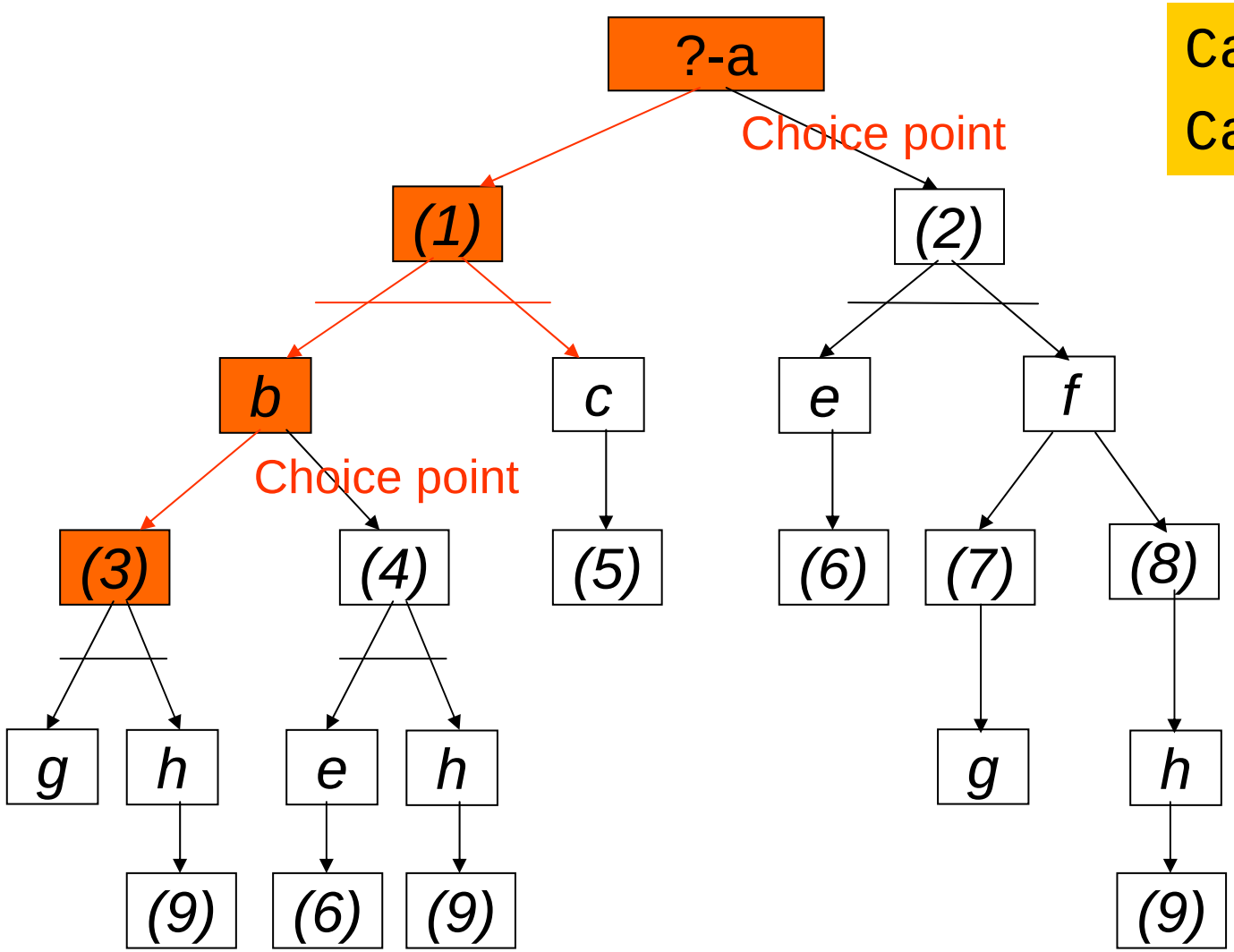

```
Call: (8) a
Call: (9) b
```

trace



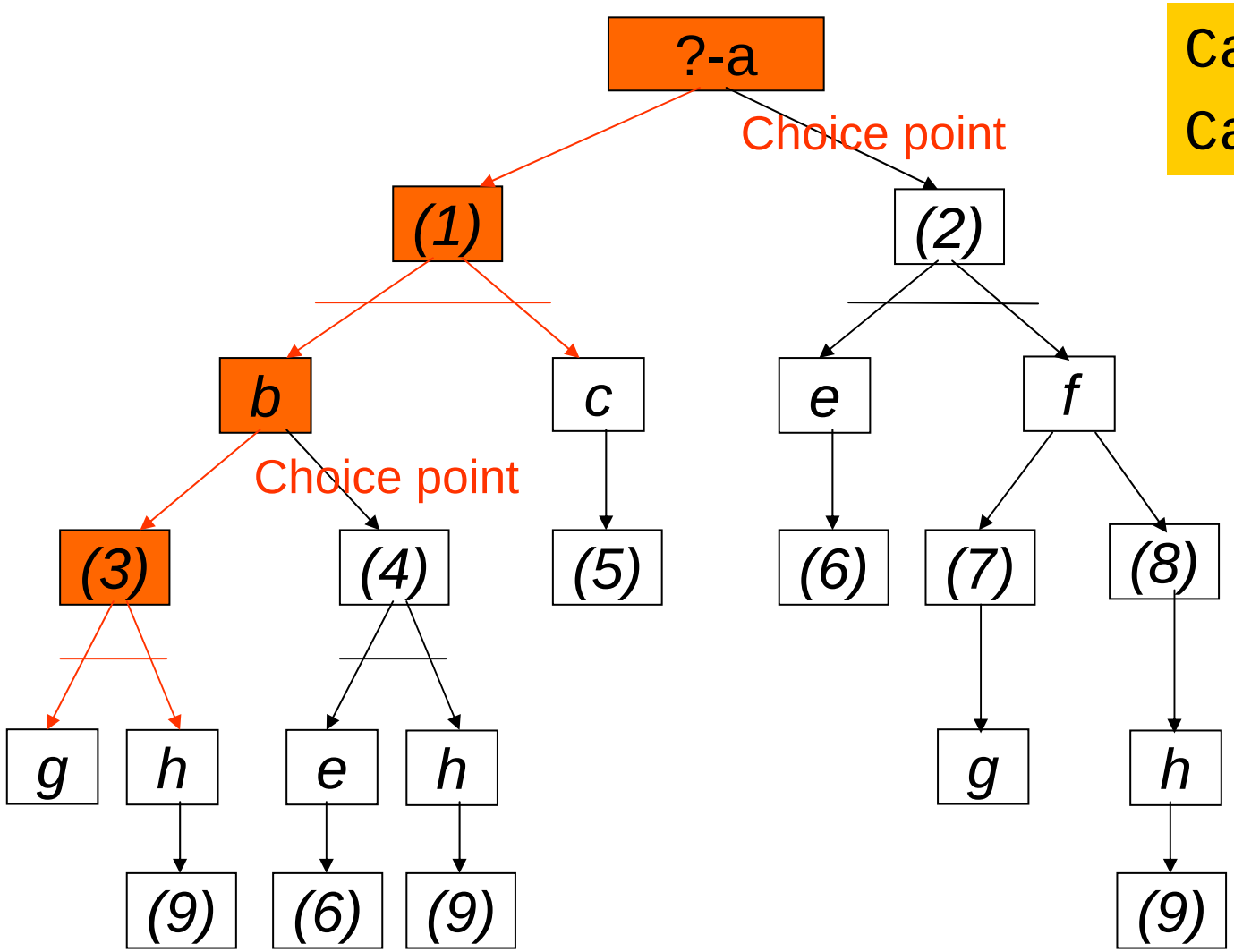

```
Call: (8) a
Call: (9) b
```

trace



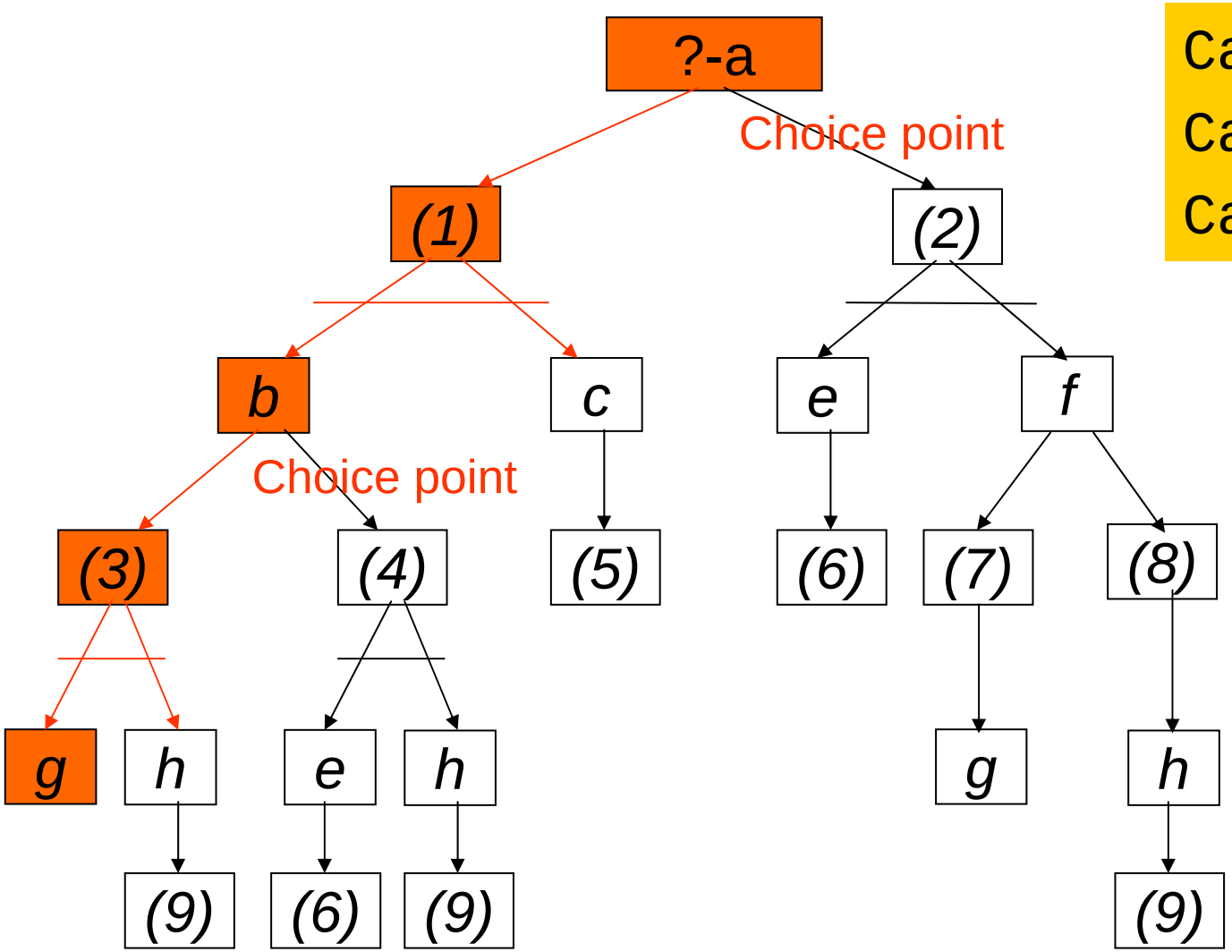

```
Call: (8) a
Call: (9) b
```

trace



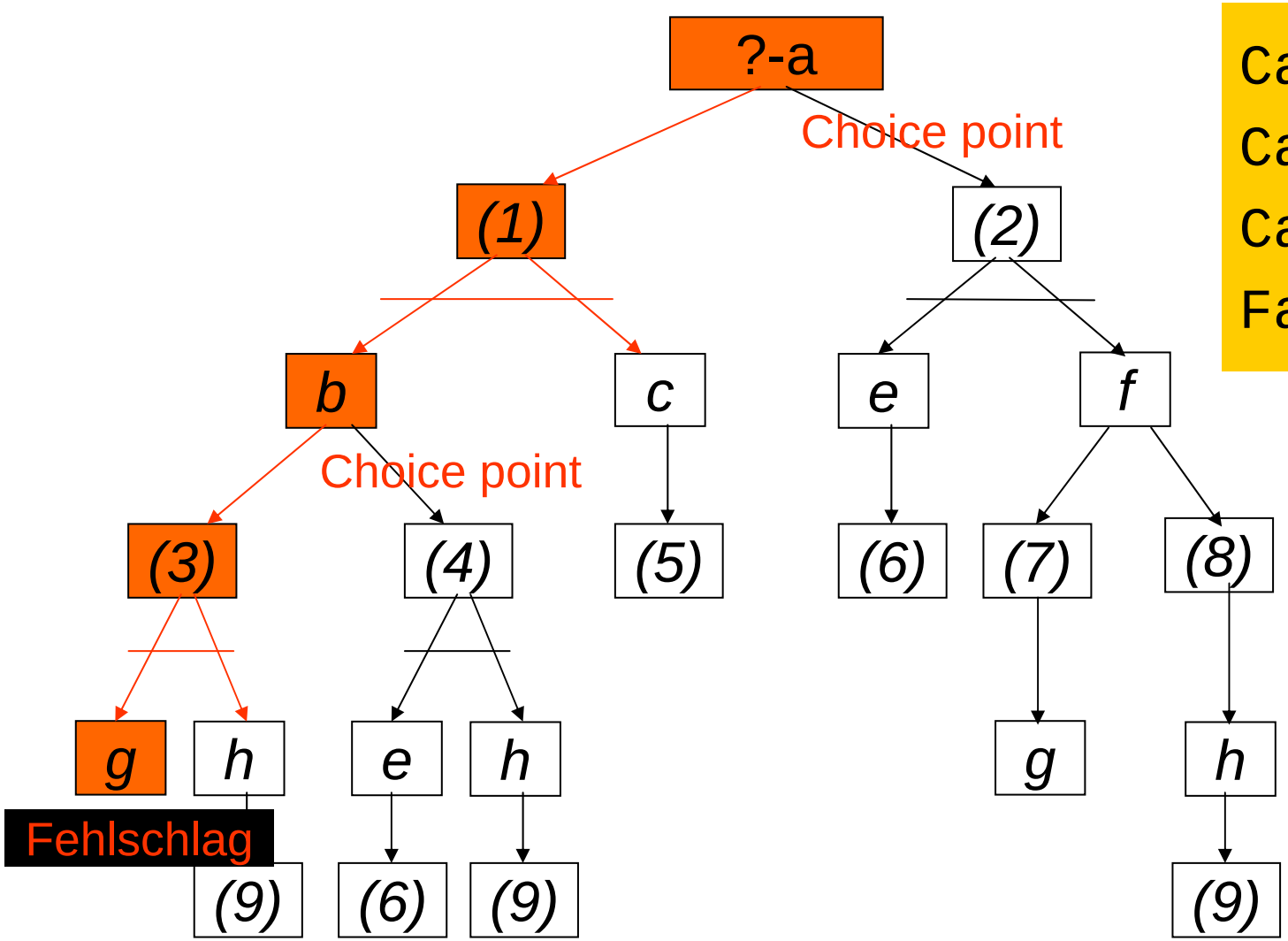

```
Call: (8) a
Call: (9) b
```

trace



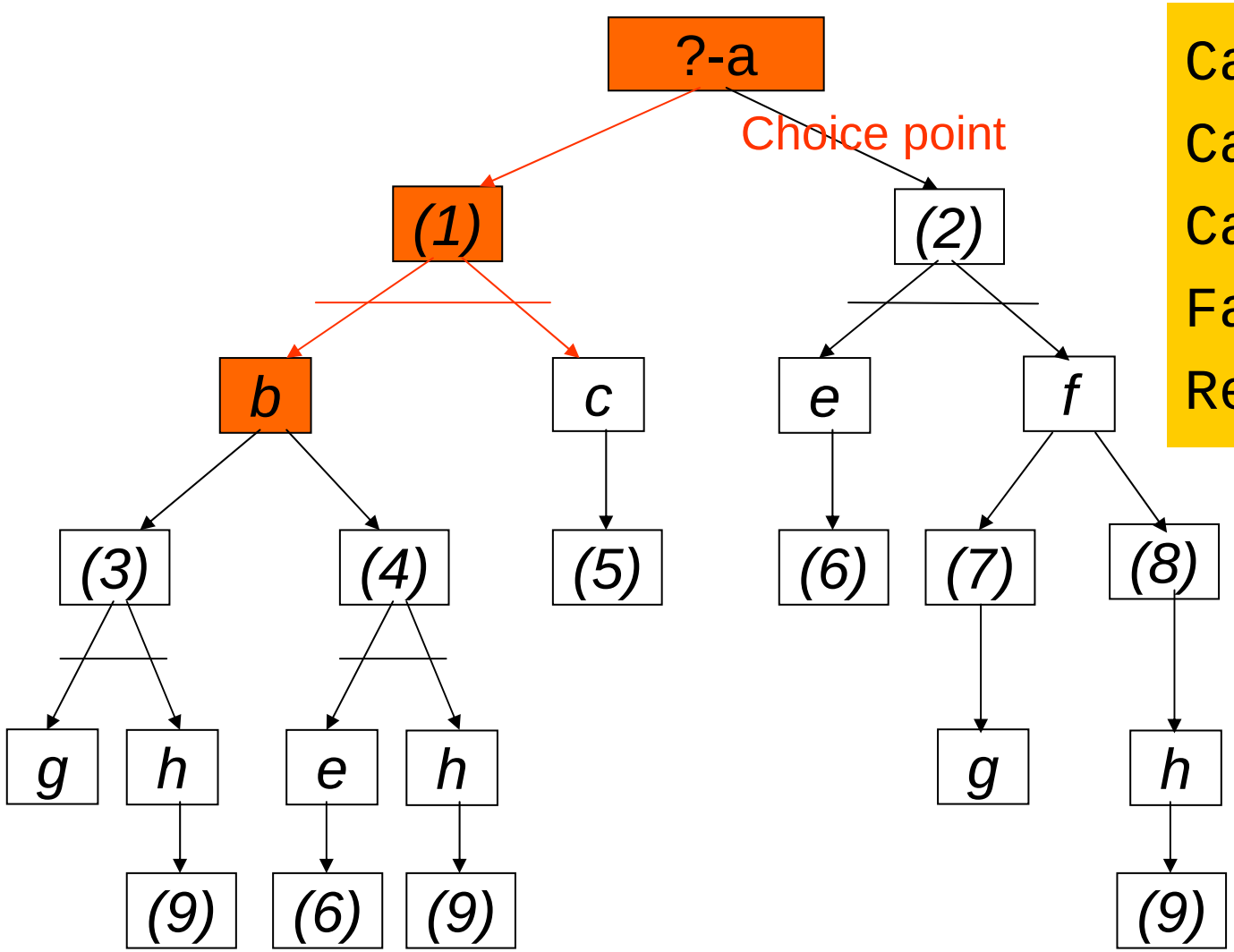

```
Call: (8) a
Call: (9) b
Call:(10) g
```

trace



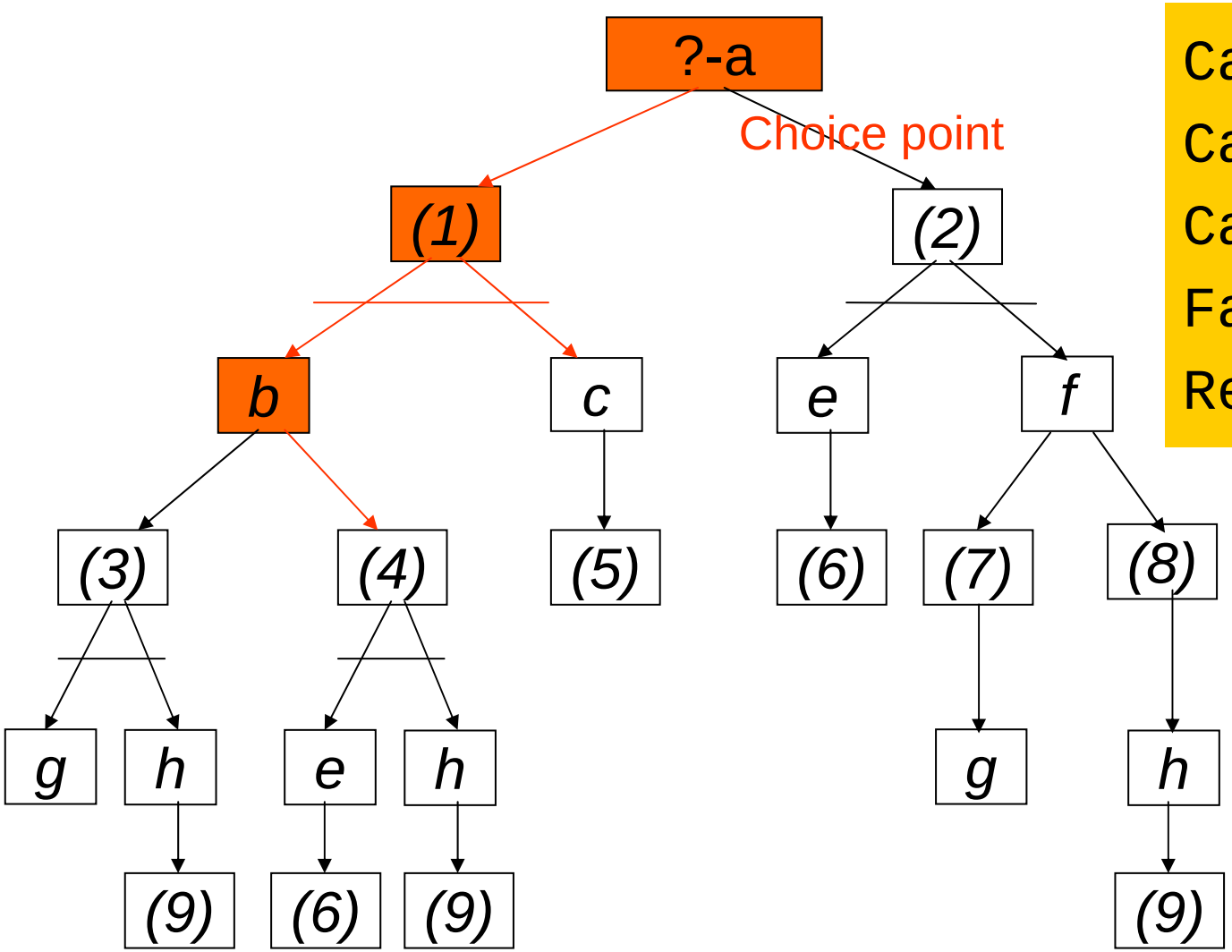

```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
```

trace



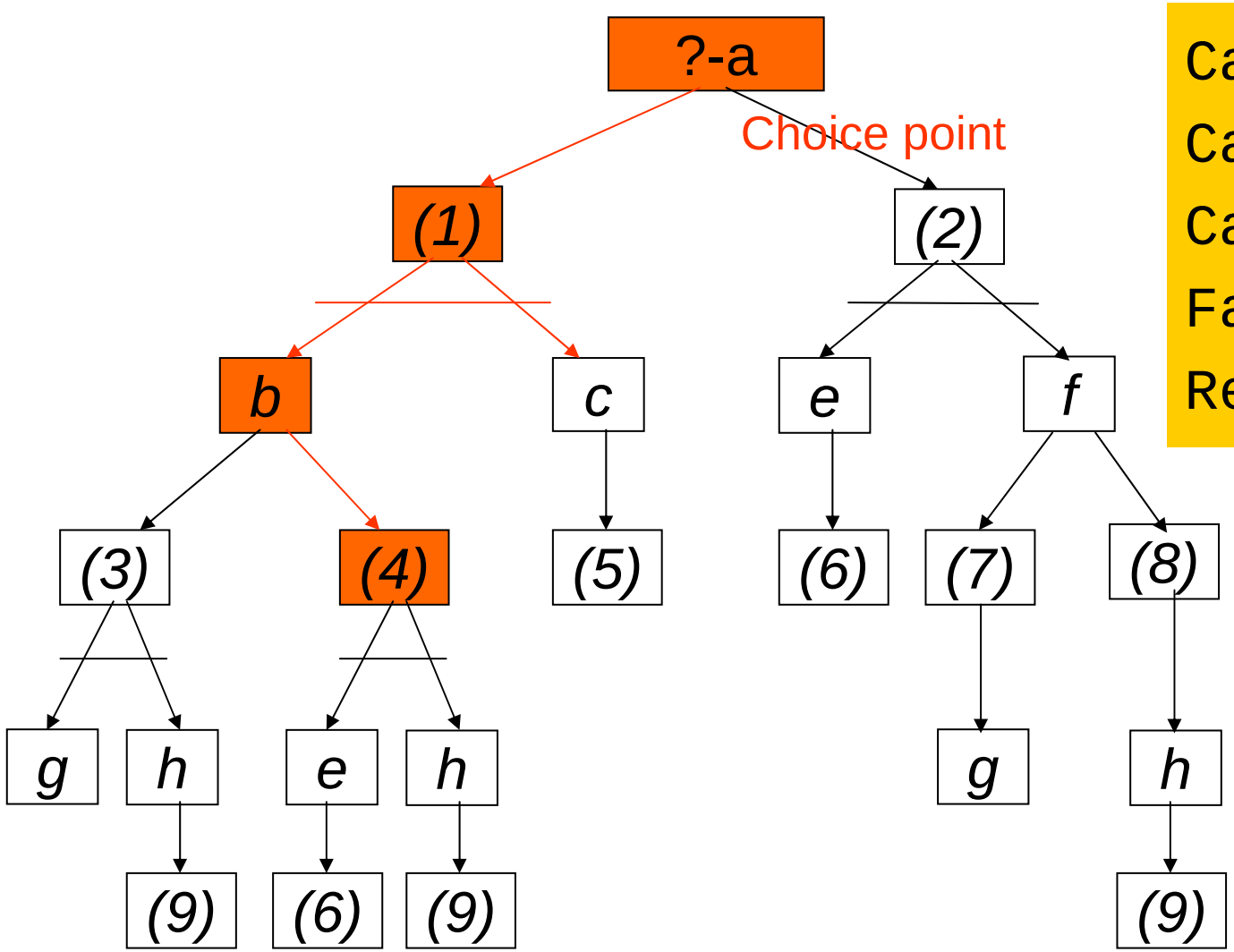

```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
Redo: (9) b
```

trace



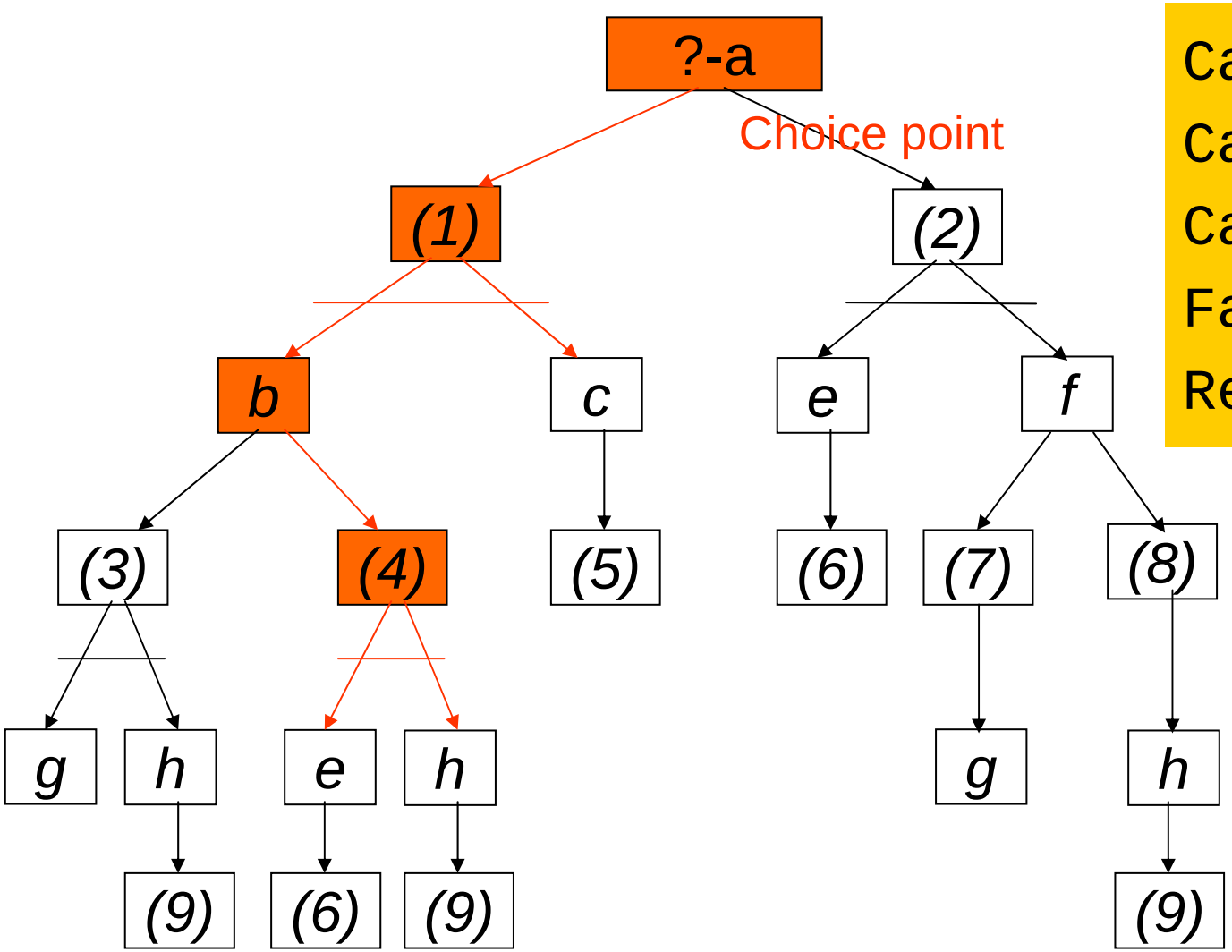

```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
Redo: (9) b
```

trace



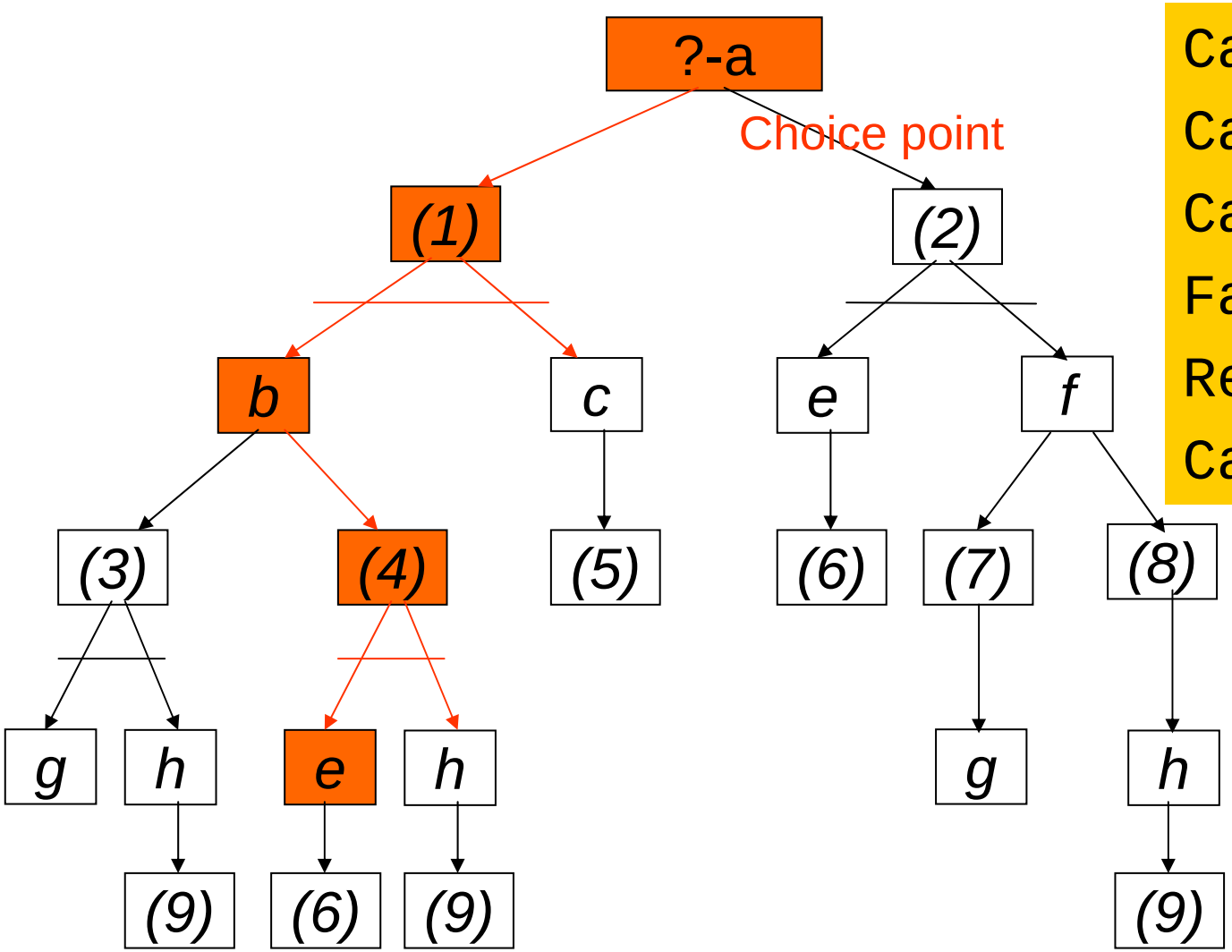

```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
Redo: (9) b
```

trace



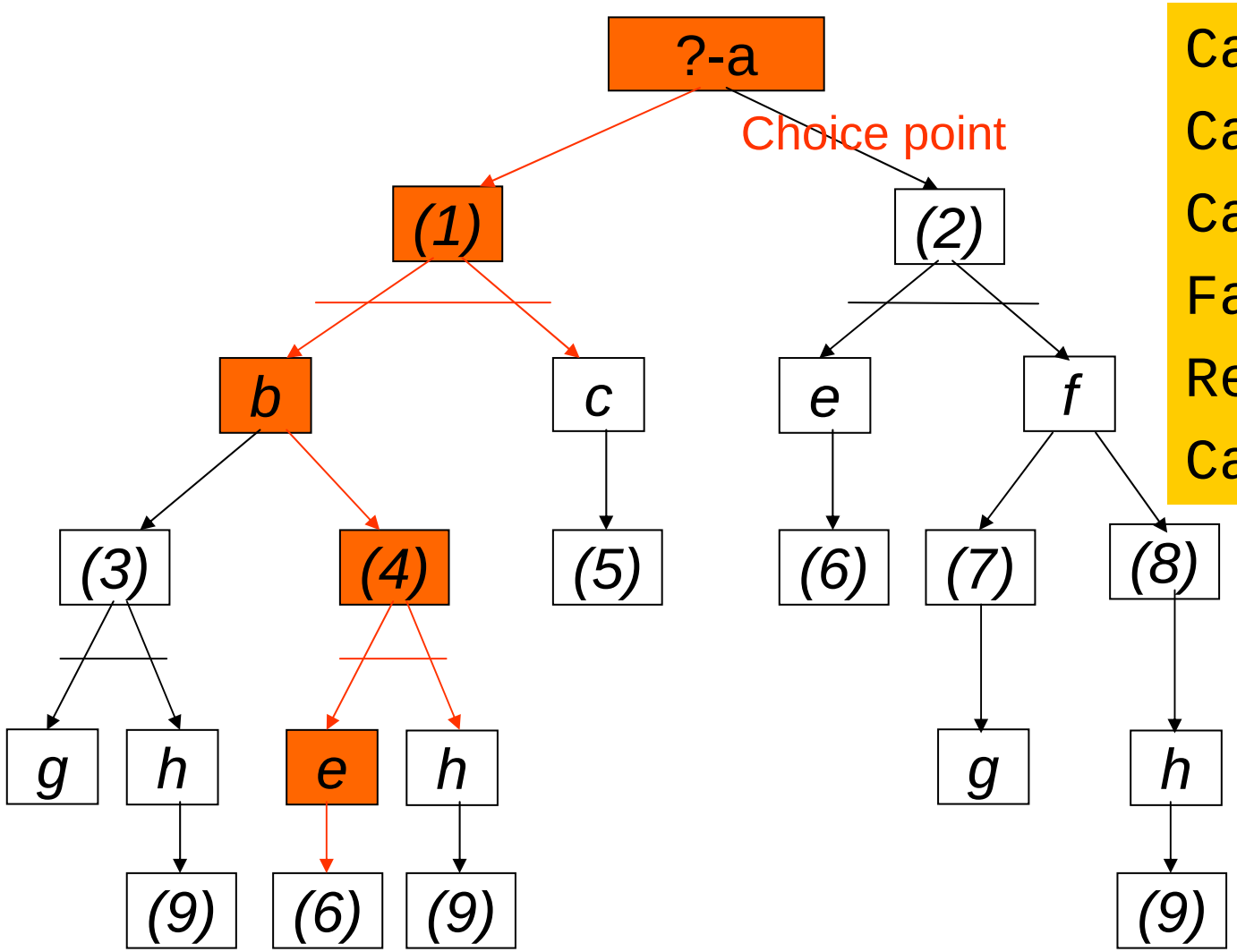

```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
Redo: (9) b
```

trace



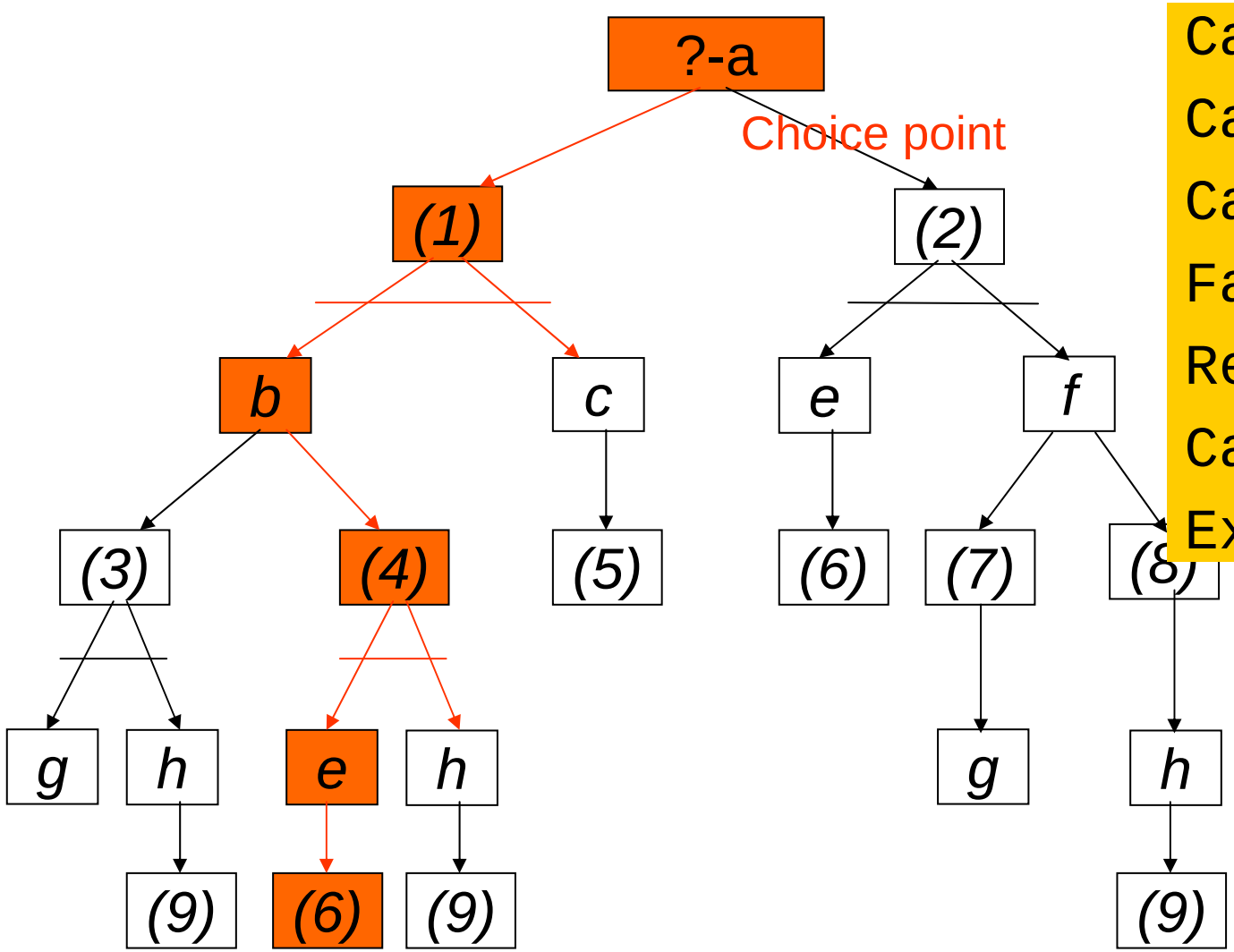

```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
Redo: (9) b
Call:(10) e
```

trace



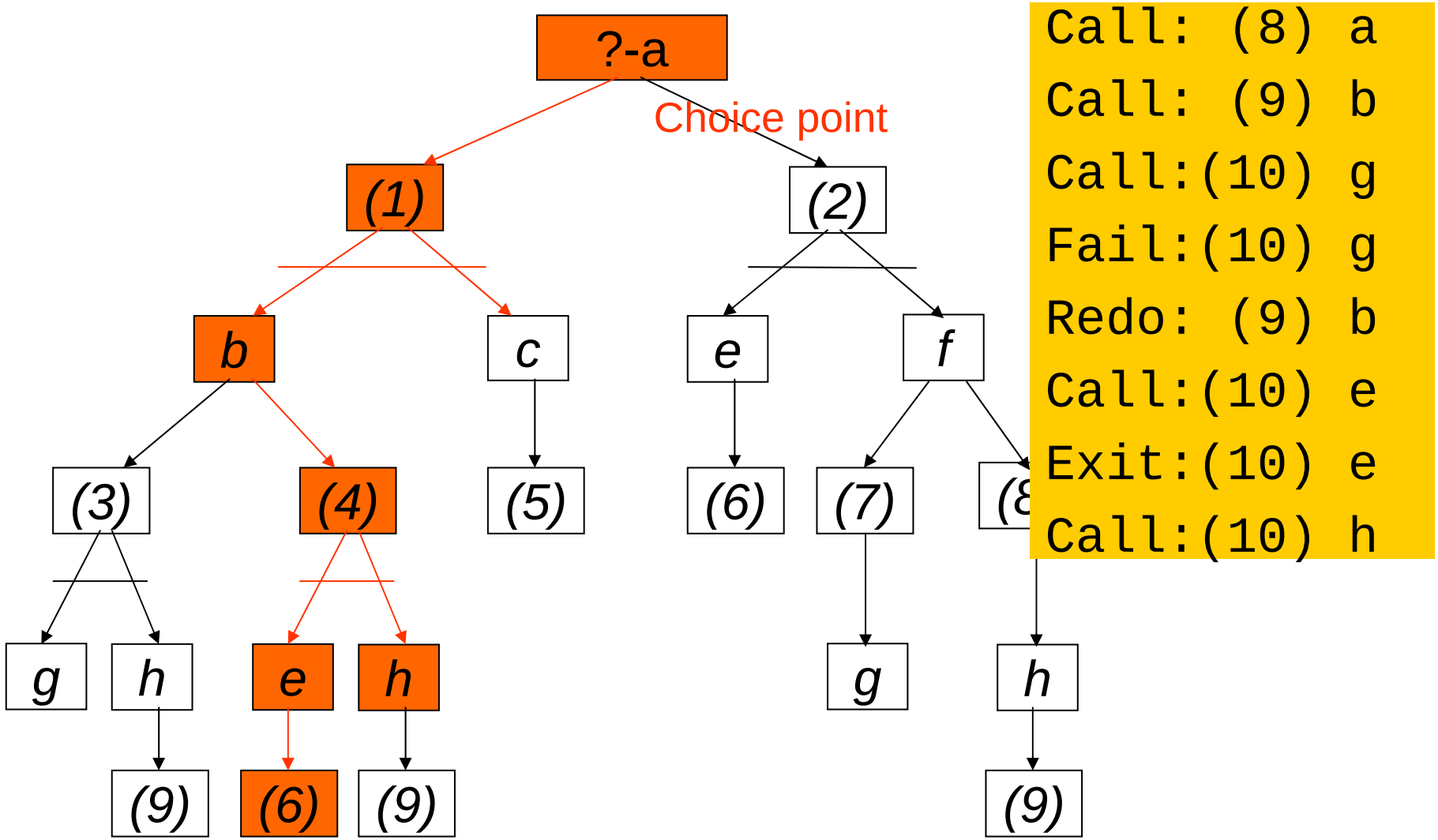

```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
Redo: (9) b
Call:(10) e
```

trace



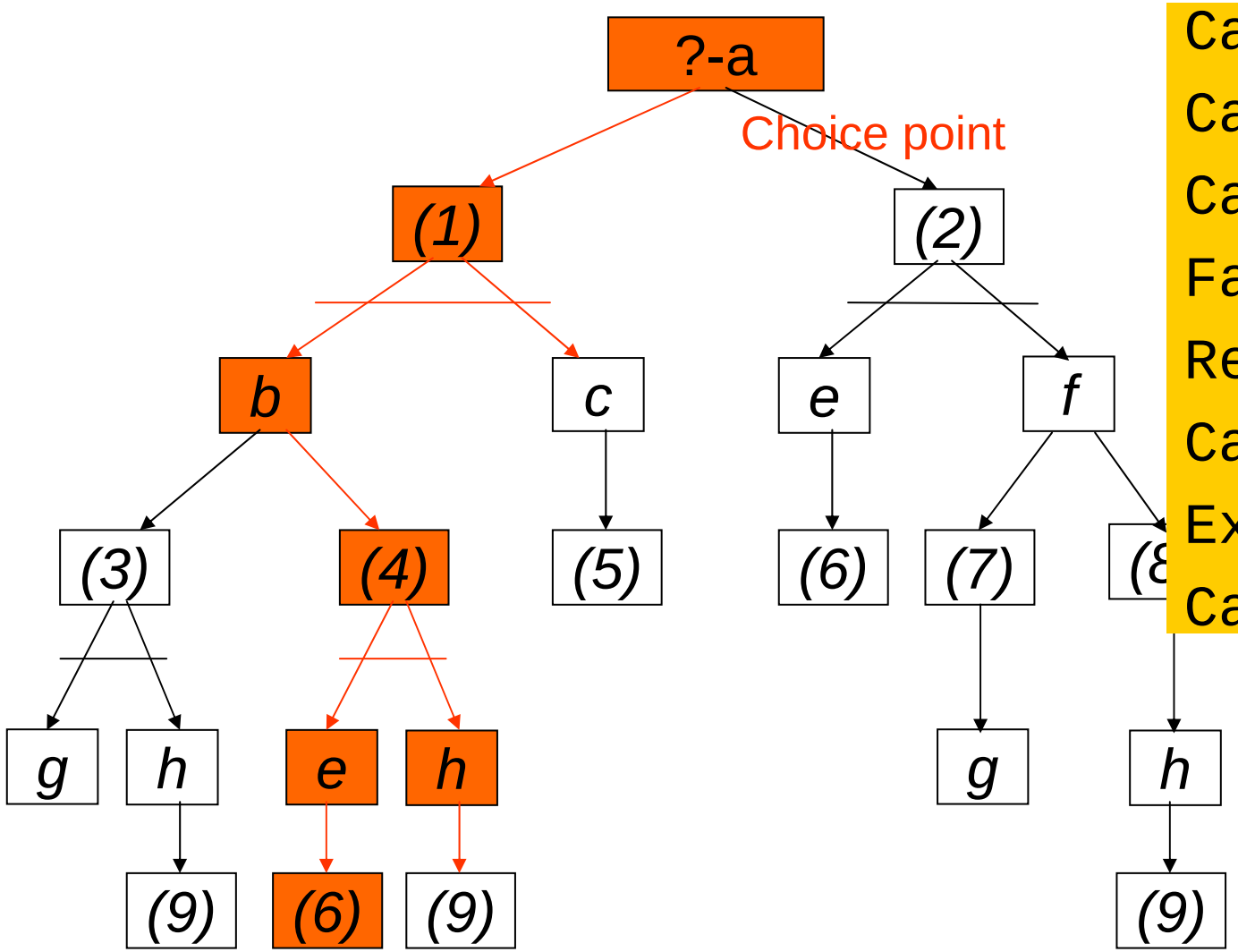

```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
Redo: (9) b
Call:(10) e
Exit:(10) e
```

trace



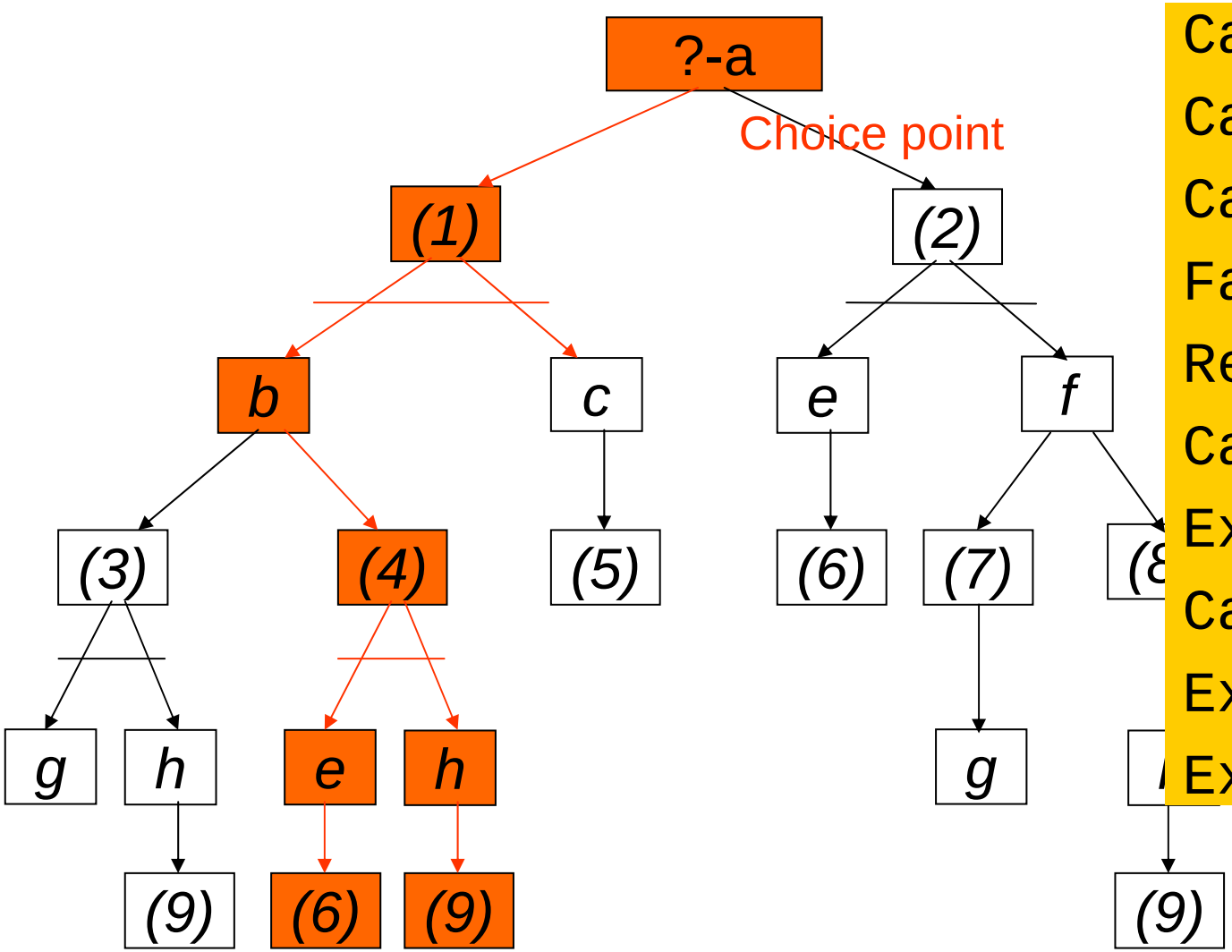

```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
Redo: (9) b
Call:(10) e
Exit:(10) e
Call:(10) h
```

trace



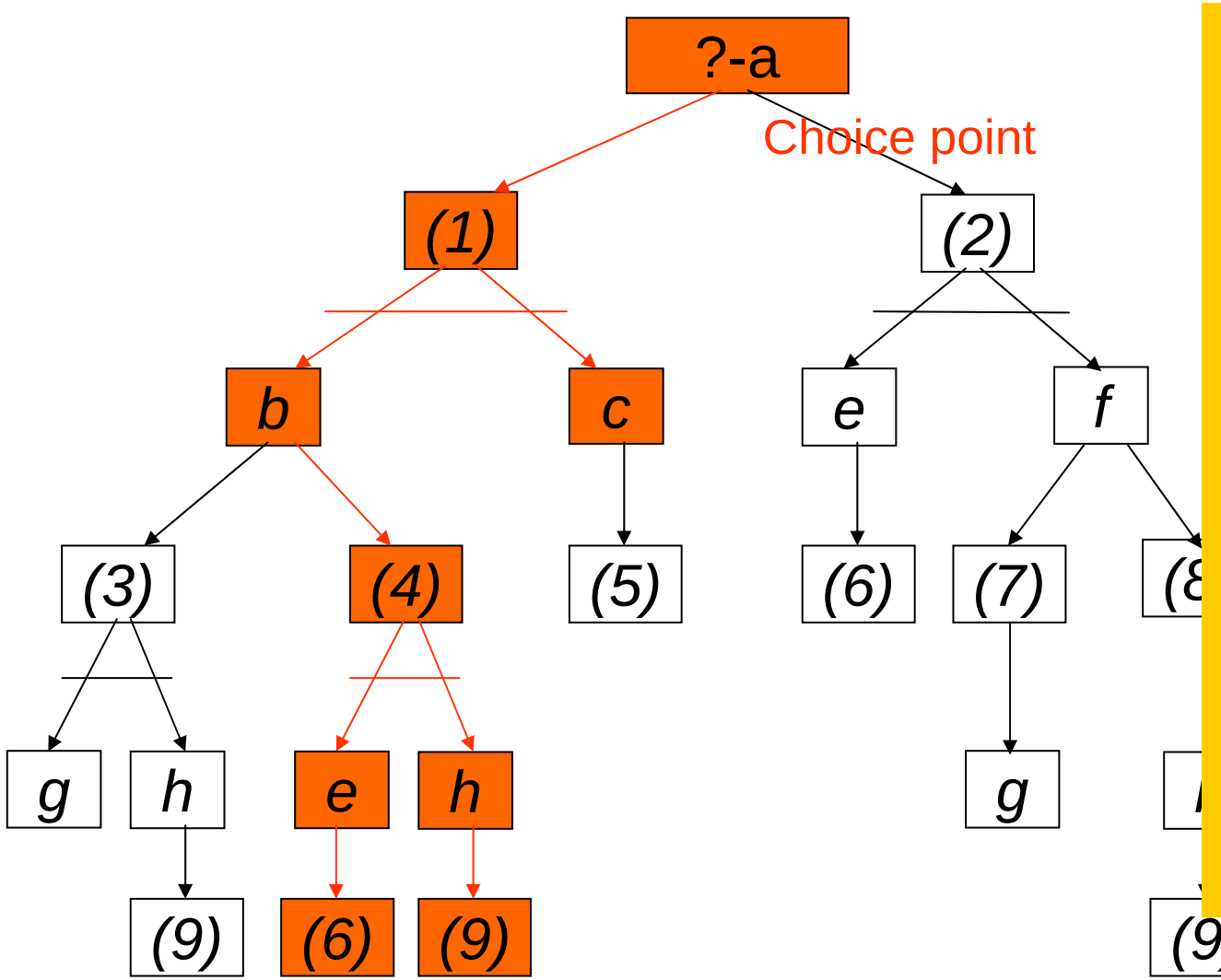

```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
Redo: (9) b
Call:(10) e
Exit:(10) e
Call:(10) h
```

trace



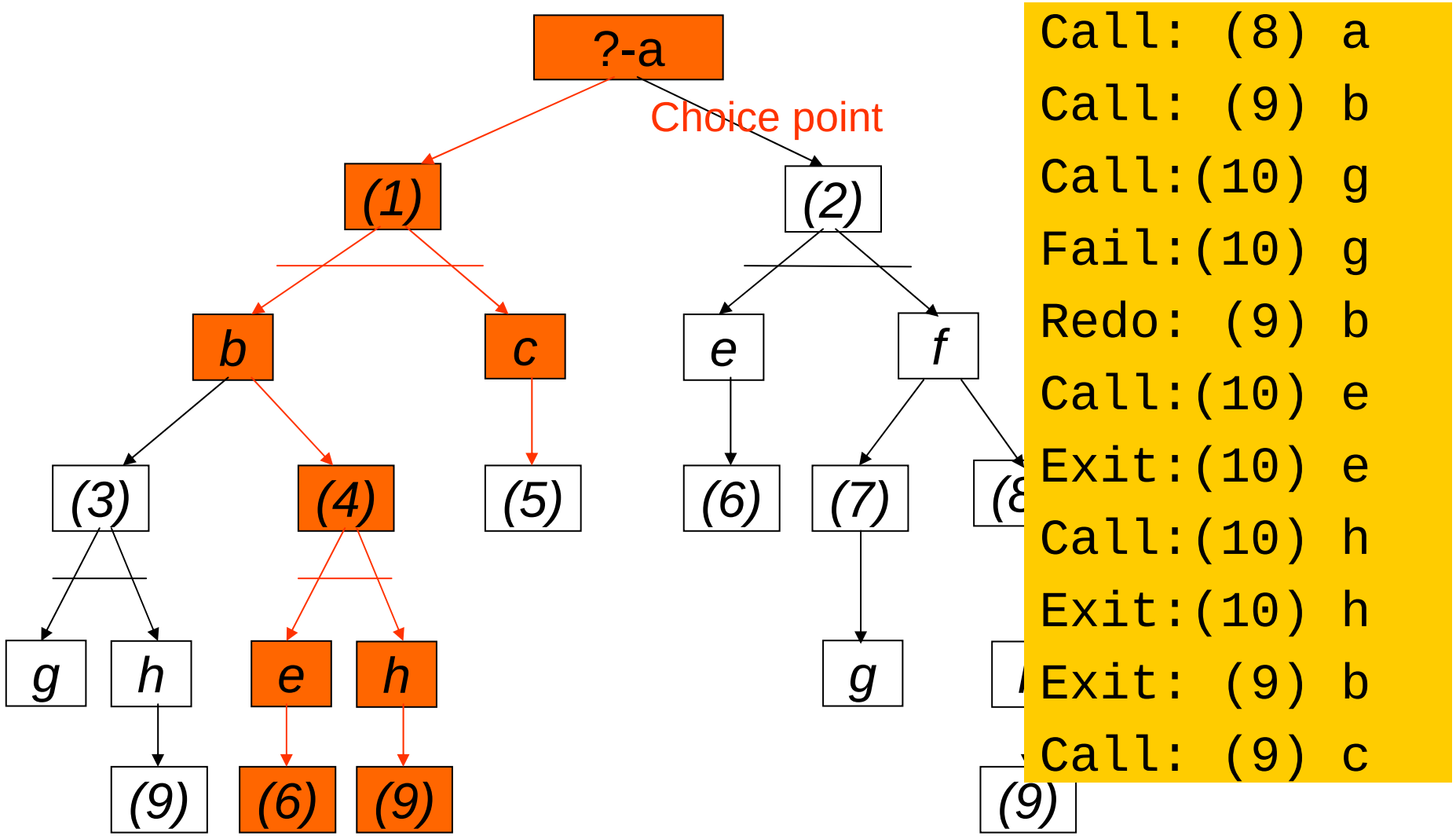

```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
Redo: (9) b
Call:(10) e
Exit:(10) e
Call:(10) h
Exit:(10) h
Exit: (9) b
```

trace



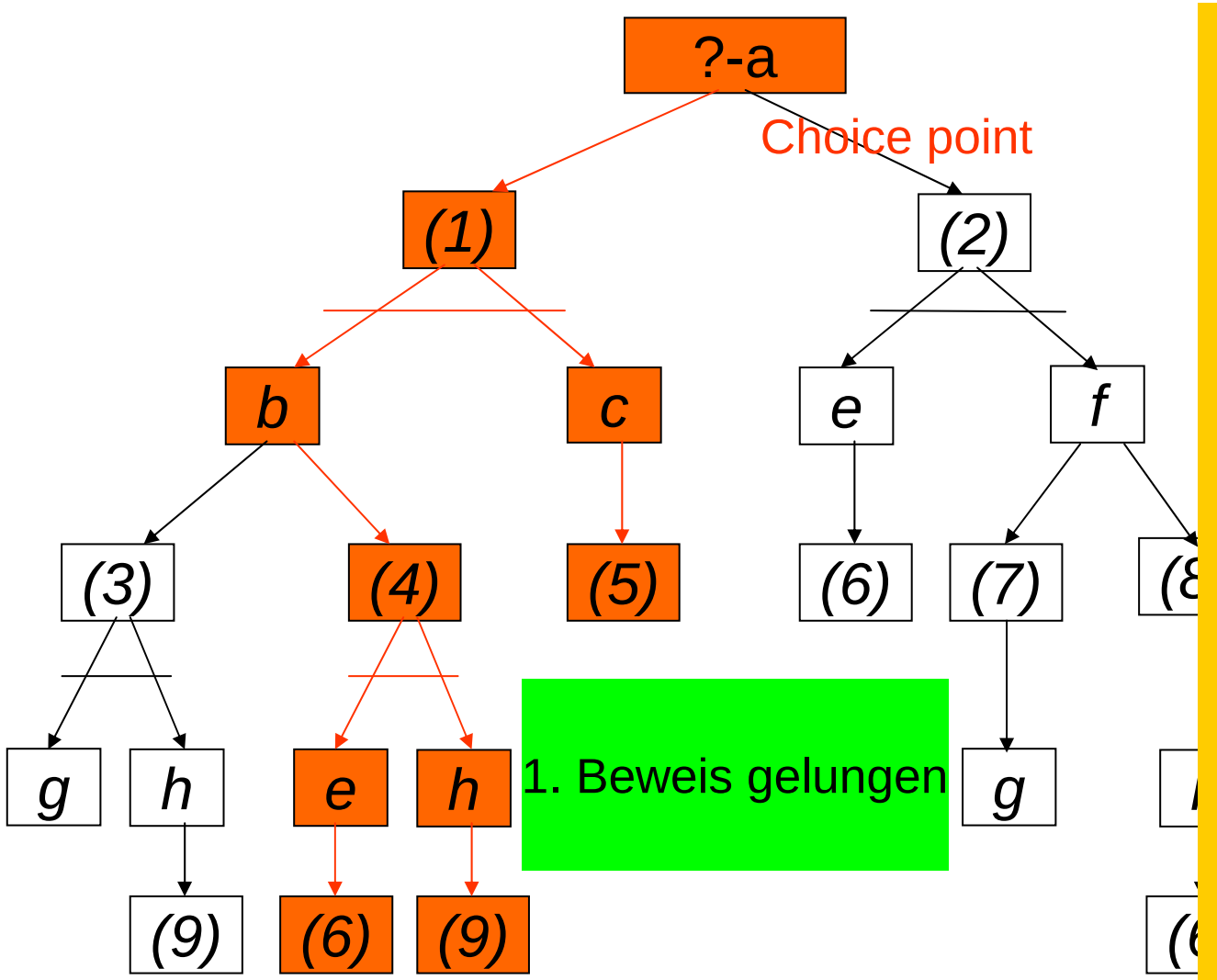

```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
Redo: (9) b
Call:(10) e
Exit:(10) e
Call:(10) h
Exit:(10) h
Exit: (9) b
Call: (9) c
```

trace




```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
Redo: (9) b
Call:(10) e
Exit:(10) e
Call:(10) h
Exit:(10) h
Exit: (9) b
Call: (9) c
```

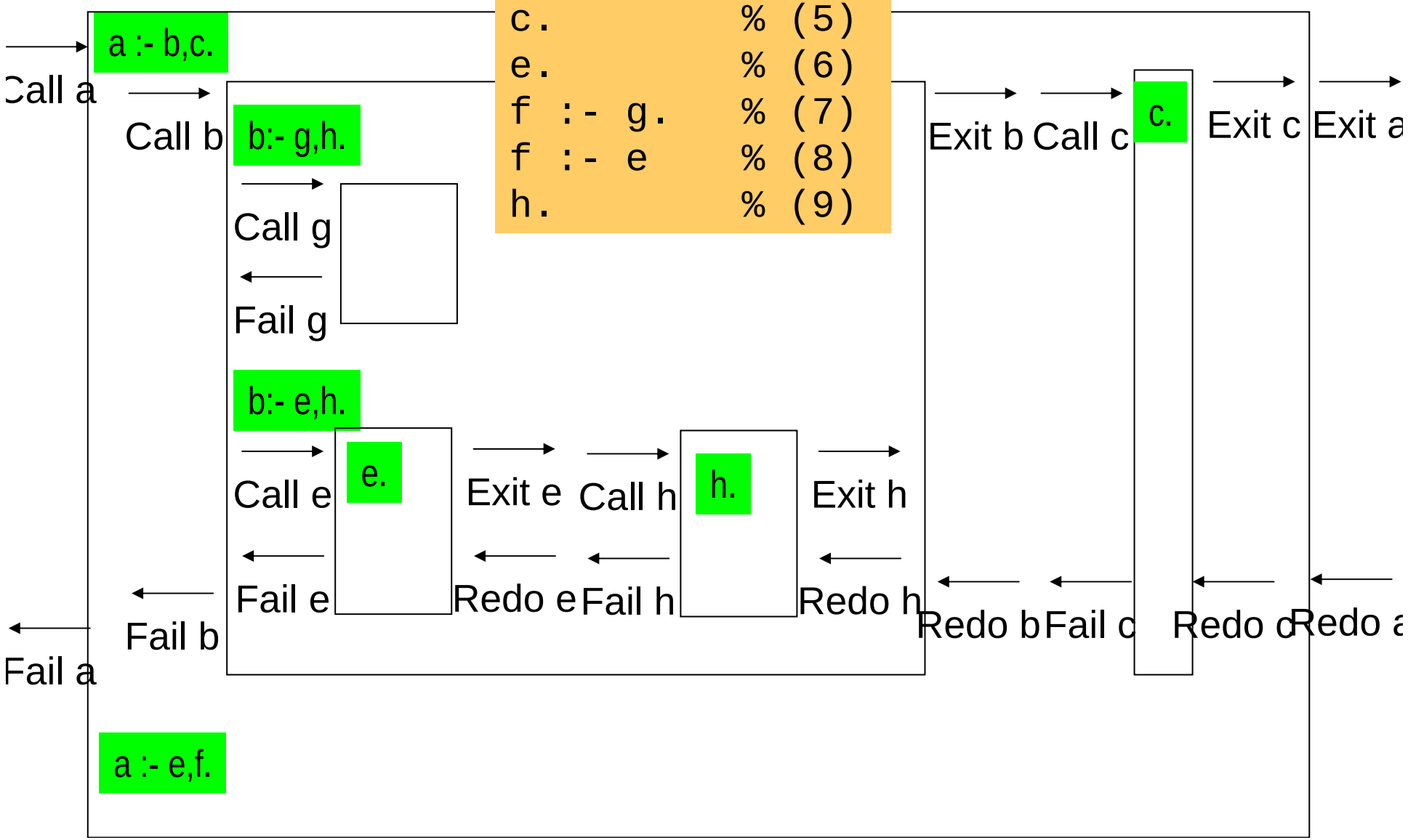
trace




```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
Redo: (9) b
Call:(10) e
Exit:(10) e
Call:(10) h
Exit:(10) h
Exit: (9) b
Call: (9) c
Exit: (9) c
Exit: (8) a
```

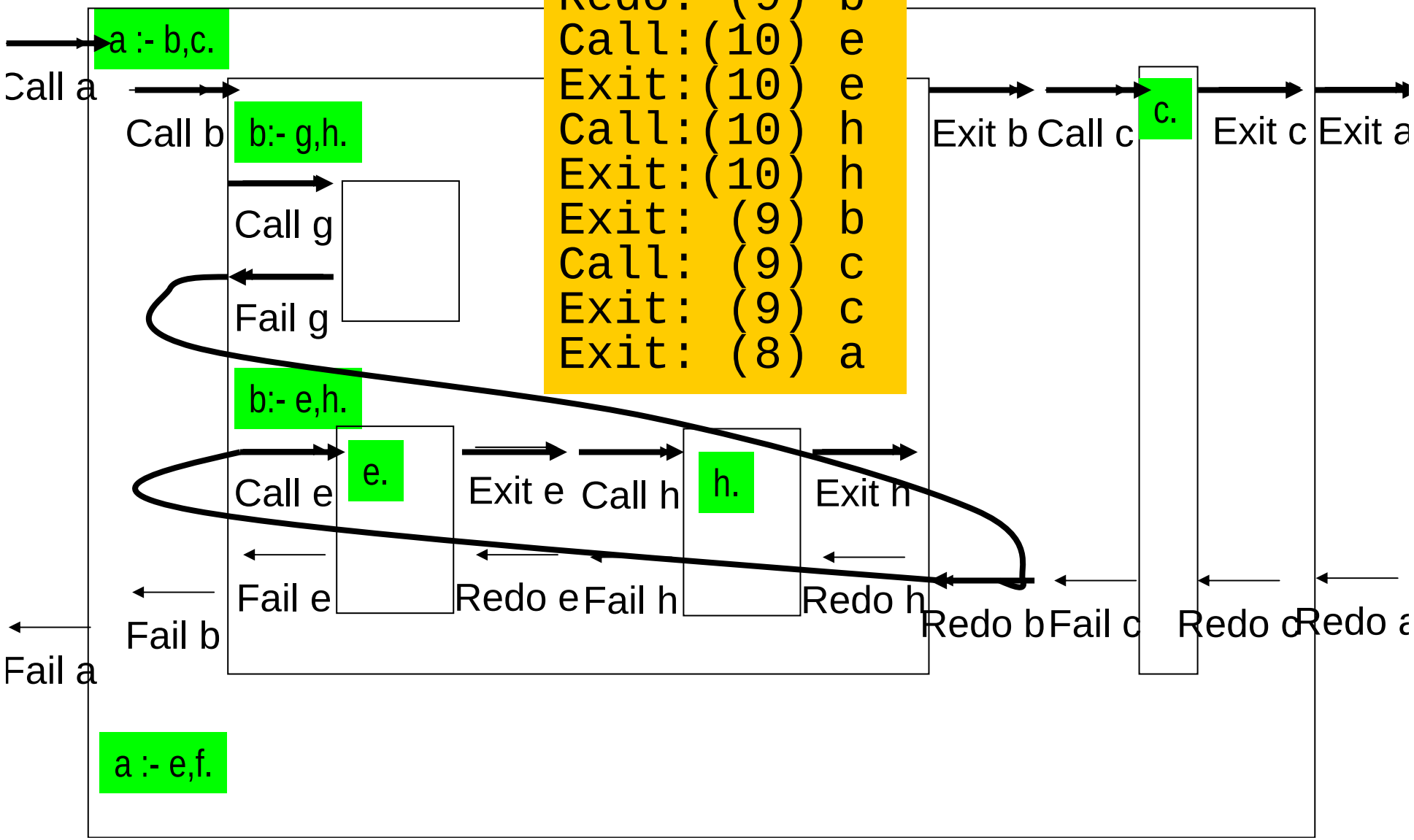
trace

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



trace

```
Call: (8) a
Call: (9) b
Call:(10) g
Fail:(10) g
Redo: (9) b
Call:(10) e
Exit:(10) e
Call:(10) h
Exit:(10) h
Exit: (9) b
Call: (9) c
Exit: (9) c
Exit: (8) a
```



Debugger

Mit debug/0 wird debug-Modus gestartet:

- Anzeigen von Trace-Punkten
- Unterbrechung an Spy-Punkten
  mit Möglichkeit zur interaktiven Arbeit
- Setzt ggf. Optimierungen außer Kraft
- Impliziter Aufruf durch spy bzw. trace
- Abschalten mit nodebug/0

Debugger

trace/2 setzt/löscht Trace-Punkte

- für Prädikate (1.Argument)
- an angebenen Ports (2.Argument):

  +portname bzw. –portname bzw. Liste

spy/1 bzw. nospy/1 setzt/löscht Spy-Punkte

- für angegebene Prädikate

trace/0 setzt überall Trace-Punkte für folgenden Aufruf

debugging/0 zeigt Status

Variable/Parameter in Prolog

Imperative Programmiersprachen

- Überschreiben von Variablen
- Parameterübergabe bei Prozedur-Aufruf über
  - Wert (value-Parameter)
  - Referenz (reference-Parameter)

Prolog

- Dauerhafte Bindung (Instantiierung) von Variablen
- Parameter-Übergabe durch Unifikation
  - Referenzierung an Variable bzw. Wert

Variable/Parameter: Verzögerte Berechnung

```
[trace]  ?- append1([a,b,c],[1,2],L).
   Call: (6) append1([a, b, c], [1, 2], _G294)
   Call: (7) append1([b, c], [1, 2], _G366)
   Call: (8) append1([c], [1, 2], _G369)
   Call: (9) append1([], [1, 2], _G372)
   Exit: (9) append1([], [1, 2], [1, 2])
   Exit: (8) append1([c], [1, 2], [c, 1, 2])
   Exit: (7) append1([b, c], [1, 2], [b, c, 1, 2])
   Exit: (6) append1([a, b, c], [1, 2], [a, b, c, 1, 2])

L = [a, b, c, 1, 2]
```

Variable/Parameter: Verzögerte Berechnung

```
[trace]  ?- naive_reverse([a,b,c],L).
   Call: (6) naive_reverse([a, b, c], _G287)
   Call: (7) naive_reverse([b, c], _G356)
   Call: (8) naive_reverse([c], _G356)
   Call: (9) naive_reverse([], _G356)
   Exit: (9) naive_reverse([], [])
   Call: (9) append1([], [c], _G360)
   Exit: (9) append1([], [c], [c])
   Exit: (8) naive_reverse([c], [c])
   Call: (8) append1([c], [b], _G363)
   Call: (9) append1([], [b], _G358)
   Exit: (9) append1([], [b], [b])
   Exit: (8) append1([c], [b], [c, b])
   Exit: (7) naive_reverse([b, c], [c, b])
   Call: (7) append1([c, b], [a], _G287)
   Call: (8) append1([b], [a], _G364)
   Call: (9) append1([], [a], _G367)
   Exit: (9) append1([], [a], [a])
   Exit: (8) append1([b], [a], [b, a])
   Exit: (7) append1([c, b], [a], [c, b, a])
   Exit: (6) naive_reverse([a, b, c], [c, b, a])
L = [c, b, a]
```

Variable/Parameter: Verzögerte Berechnung

```
[trace]  ?- reverse1([a,b,c],L).
   Call: (6) reverse1([a, b, c], _G287)
   Call: (7) reverse_acc([a, b, c], [], _G287)
   Call: (8) reverse_acc([b, c], [a], _G287)
   Call: (9) reverse_acc([c], [b, a], _G287)
   Call: (10) reverse_acc([], [c, b, a], _G287)
   Exit: (10) reverse_acc([], [c, b, a], [c, b, a])
   Exit: (9) reverse_acc([c], [b, a], [c, b, a])
   Exit: (8) reverse_acc([b, c], [a], [c, b, a])
   Exit: (7) reverse_acc([a, b, c], [], [c, b, a])
   Exit: (6) reverse1([a, b, c], [c, b, a])

L = [c, b, a]
```

Prädikate

Prädikate (Relationen) für „gültige Sachverhalte“

Beweise für Folgerungen/Ableitungen

Prädikate werden vom Nutzer definiert

Prolog-System realisiert (spezielles) Beweisverfahren

Prolog-System besitzt eingebaute (built-in) Prädikate für

- Programmierumgebung
- Eingabe/Ausgabe
- Steuerung der Abarbeitung
- Programm-Manipulation
- Term-Operationen
- Häufig benutzte Prädikate
- Arithmetik

Eingabe/Ausgabe

Eröffnen von files:

tell/1 see/1

Schließen von files:

told/0 seen/0

see(datei1),lese_von_datei(X),see(user), ...

tell(datei3),schreibe_auf_datei(Z),tell(user),...

Erfragen des offenen files (Terminal: user):

telling/1 seeing/1

Eingabe/Ausgabe von Termen:

read/1 write/1

Eingabe/Ausgabe von Zeichen:

get/1 put/1

und weitere ...

read(-Term) („Eingabeparameter“)

write(+Term) („Ausgabeparameter“)

Eingabe/Ausgabe

```
kubik :- write('Nächste Zahl bitte: '),  read( X ), bearbeite( X ).

bearbeite( stop ):- !.
bearbeite( N )  :- K is N*N*N,
                   write('Dritte Potenz von '), write( N ),
                   write(' ist '), write( K ), nl,
                   kubik.
```

Steuerung von Programmen

`true, fail, !, not, ;, ,, call, repeat`

- Aufruf: call/1

```
call(Ziel) :- Ziel .
```

- Endlose Schleifen: repeat/0

```
repeat.
repeat:-repeat.
```

Als System-Prädikat repeat/0 beliebig oft backtrackbar.

```
bearbeite_datei(D) :- see(D),
                      repeat, read(Term),
                      ( Term = end_of_file ; write(Term), fail ),  ! , seen.
```

end_of_file: CTRL d

Programmierumgebung

Laden eines PROLOG-Programms:

consult/1 oder [filename]

consult(user) bzw. [user] : Standard-Eingabe bis CTRL D

Beenden des PROLOG-Systems: halt/0

Debugging: trace/0, notrace usw.

Hilfesystem: help/0, help/1, apropos/1

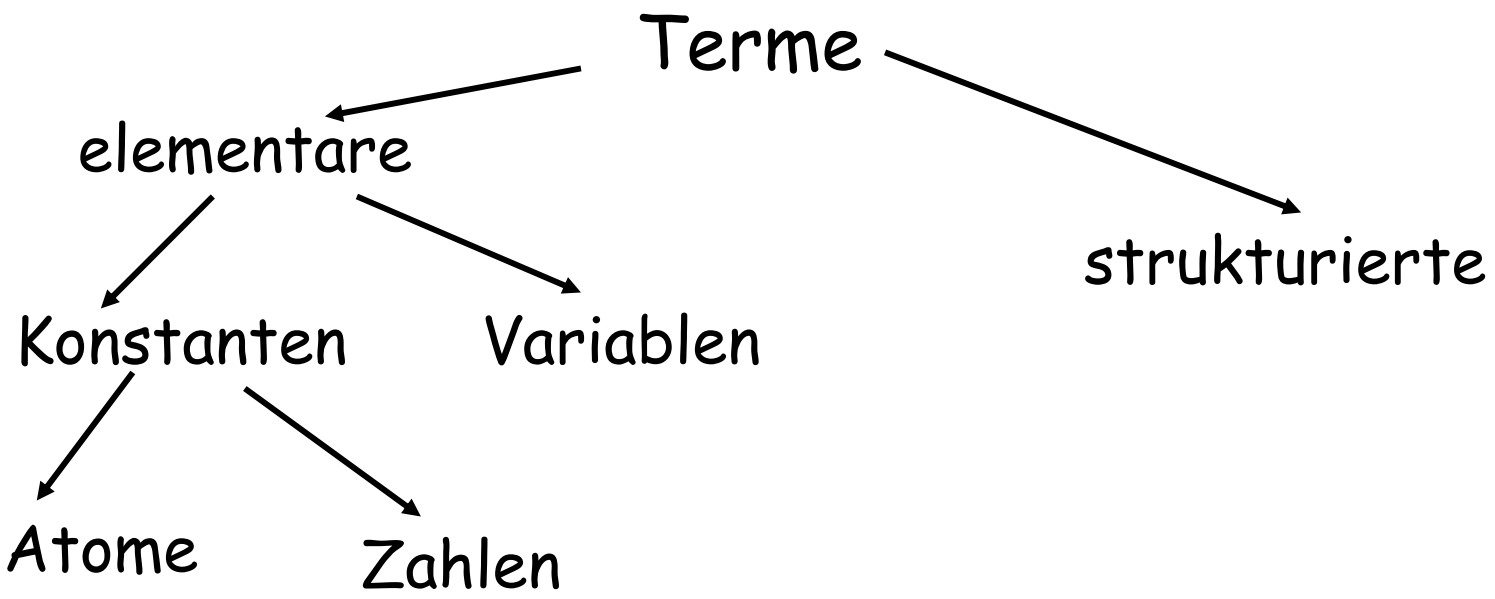
History-System: set_prolog_flag(history, N).

Auflisten von Programm-Klauseln: listing/1, listing/0

Analyse und Konstruktion von Termen

atom/1, integer/1, var/1, nonvar/1, ...

compound/1, ground/1, ...



Analyse und Konstruktion von Termen

- =.. *(„univ“)* , funktor/3, arg/3 , ...

```
?- funktor_name(arg1,arg2) =.. L .
L = [funktor_name,arg1,arg2]
```

```
..., Goal=..[Funktor|ArgListe], call(Goal), ...
```

```
functor(parent(X,zeus),parent,2)
```

```
arg(2, parent(X,zeus),zeus)
```

- Termvergleich = , == , \= , \==

Programmoperationen

Hinzufügen von Programm-Klauseln

`assert/1`, `asserta/1`, `assertz/1`

Löschen von Programm-Klauseln

`retract/1`, `retractall/1` ,

`abolish/1` , `abolish/2`

Suche nach Programm-Klauseln

`clause/2`

Beschränkung auf „dynamische" Klauseln

`dynamic/1`

z.B.

`dynamic(male/1)`

- Selbstmodifikation von Prolog-Programmen
- Selbstanalyse von Prolog-Programmen

Verwendung von assert

Doppelberechnungen vermeiden.

z.B. Wiederverwenden von Zwischenergebnissen:

```
:-dynamic(fibo/2).
fibo( 0, 0 ).
fibo( 1, 1 ).
fibo( N, M ):-  N1 is N-1, N2 is N-2,
                fibo( N1, M1 ),
                fibo( N2, M2 ),
                M is M1 + M2,
                asserta( fibo( N, M )).
```

Kritik am Programm: Kein Test auf negative Zahlen!

Verwendung von assert

Vorverarbeitung, z.B. Multiplikationstabelle:

```
berechne_tabelle   :-
L=[0,1,2,3,4,5,6,7,8,9],
      member(X,L),member(Y,L),
      Z is X * Y,
      assert(produkt(X,Y,Z)),
      fail;
      true.
```

Willkürliches Abbrechen von Beweisketten

Ersatz von

```
problemloesen
   :- problemloesen_1(ARGUMENTE,Z),
      problemloesen_2(Z).
```

durch

```
problemloesen
   :- problemloesen_1(ARGUMENTE) ,
      assert(zwischenergebnis(Z)), fail.
problemloesen
   :- retract(zwischenergebnis(Z))
      problemloesen_2(Z).
```

Idee:

Der durch problemloesen_1 belegte Laufzeitspeicher wird am Ende der 1.Klausel freigegeben.

Bei Bedarf Cut in problemloesen_1 um Backtracking zu vermeiden.