5.6 Prolog

- ... ist die bekannteste Implementierung einer LP-Sprache;
- wurde Anfang der 1970er von Alain Colmerauer (Marseille) und Robert Kowalski (Edinburgh) entwickelt.
- konkretisiert den vorgestellten LP-Kalkül zur Bearbeitung von Zielen durch:
  - ▷ Auflösung des Nichtdeterminismus der Auswahl von Klauseln und Atomen nach einem festen Schema;
  - ▷ Erlauben der Negation durch Scheitern nur für zur Laufzeit unter den aktuellen Substitutionen variablenfreie Ziele.

  $\Longrightarrow$ SLDNF-Resolution (Linear resolution with Selection function for Definite clauses with Negation as Failure)

Auswahl von Klauseln und Atomen in Prolog

- In Prolog wird immer das am weitesten links stehende Literal (Atom oder negiertes Atom) eines Zieles selektiert und ganz entfaltet.
- Klauseln werden in textueller Reihenfolge ausgewählt.
  - ▷ Scheitert eine Ableitung mit einer bestimmten Klausel, versucht man eine neue Ableitung für das Literal mit Hilfe der nächsten Klausel. (Rücksetzen, backtracking)
- Erfolgreiche Ableitungen werden gesucht, indem man immer das zuletzt gewählte Literal, bei dem noch Klauseln zur Auswahl stehen, erneut zu entfalten versucht.
- Die Auswahl der Klauseln bei der Suche durch Rücksetzen wird als don't know Nichtdeterminismus bezeichnet.

Suchbäume

- Ein Suchbaum eines Zieles G_{start} bezüglich eines Programms P ist wie folgt definiert:

  ▷ Knoten sind Ziele.

  ▷ Die Wurzel des Baumes ist G_{start}.

  ▷ Für jede Anwendung der Entfalten-Reduktionsregel:

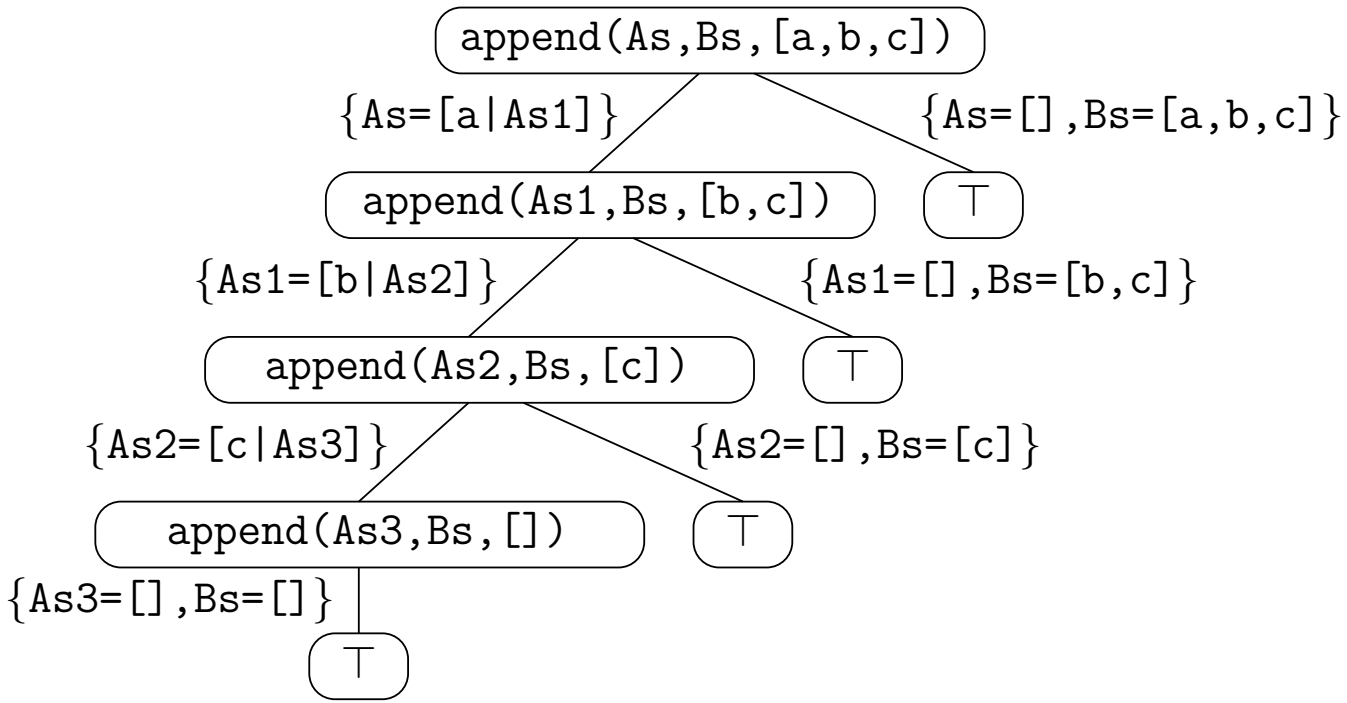
$$\frac{(B \leftarrow H) \in P \qquad \beta \text{ ist allgemeinster Unifikator von } B \text{ und } A\theta}{< A \wedge G, \theta > \mapsto_{Entfalten} < H \wedge G, \theta\beta >}$$

  existiert eine mit β beschrifteter Kante vom Knoten $A \wedge G$ zum Knoten $H \wedge G$.

- Durch die Auswahlstrategie von Prolog entspricht jedem Ziel genau ein Suchbaum.

Suchbäume

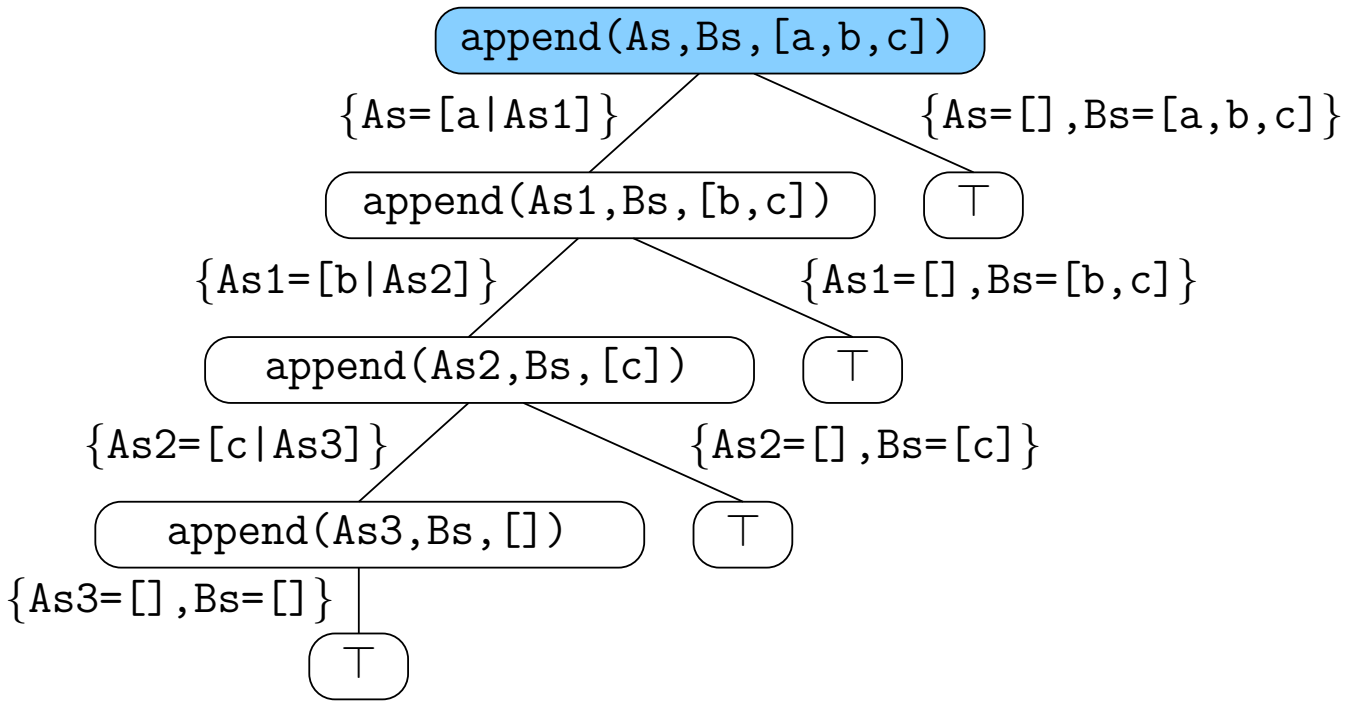
```
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
append([],Ys,Ys).
```



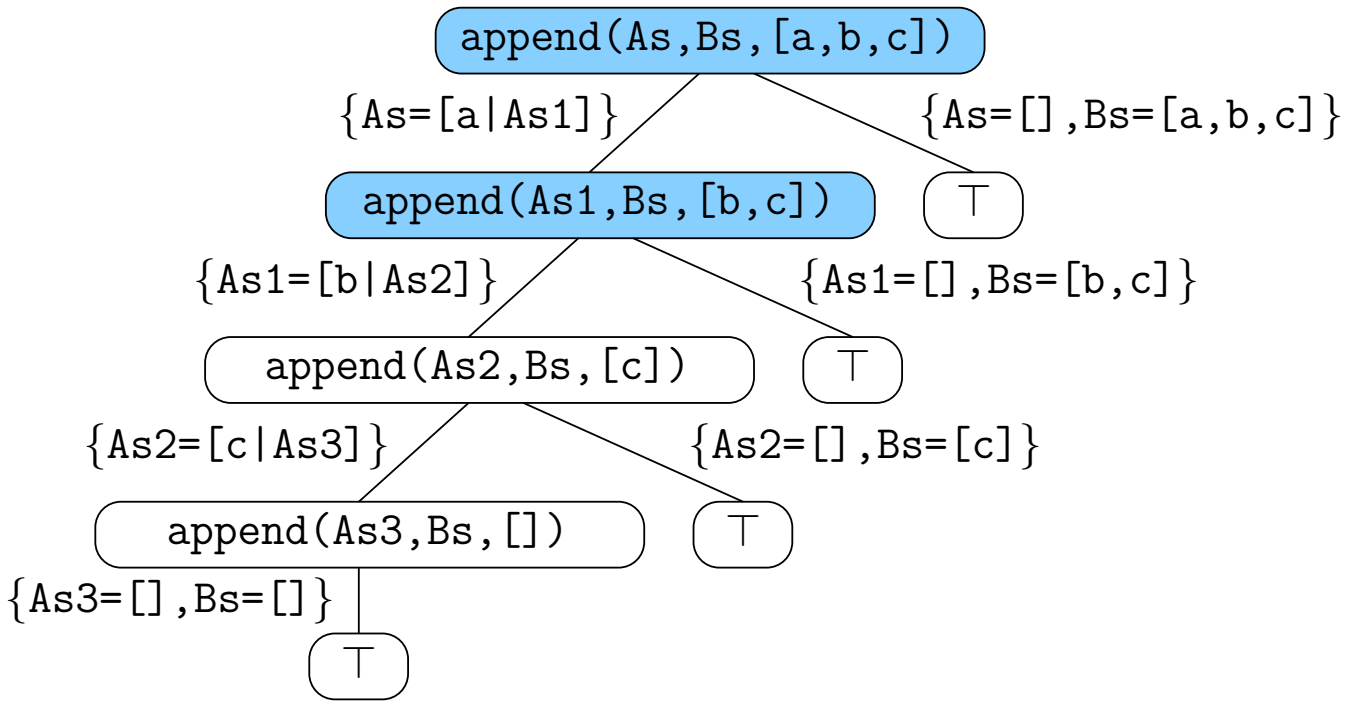
Suchbäume

- Die Blätter eines Suchbaumes, wo das leere Ziel erreicht wird, heißen Erfolgsknoten.
- Der Pfad zu einem Erfolgsknoten entspricht der Berechnung einer Antwort.
- Erfolgsknoten entsprechen einer berechneten Antwort.
- Die übrigen Blätter heißen Scheiternknoten.
- Die Auswertung eines Zieles erfolgt in Prolog durch einen Tiefendurchlauf über den entsprechenden Suchbaum.

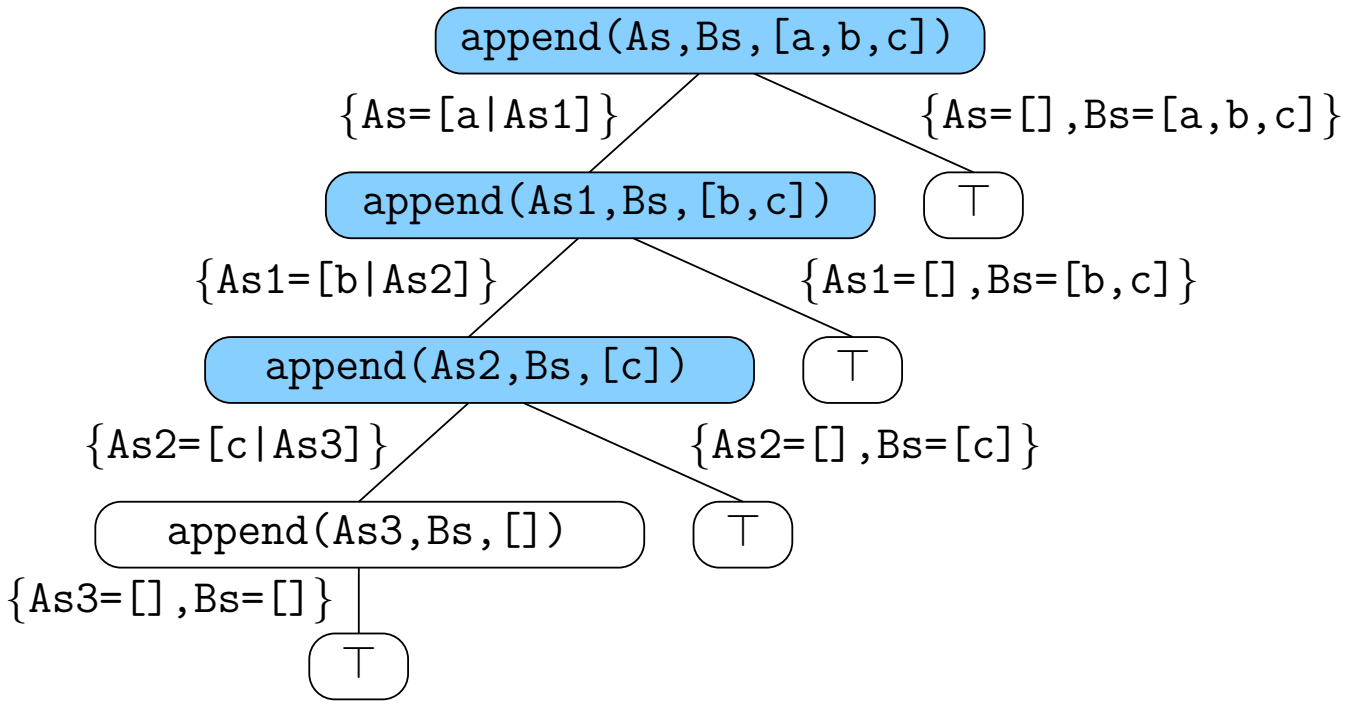
Prolog's Suchstrategie



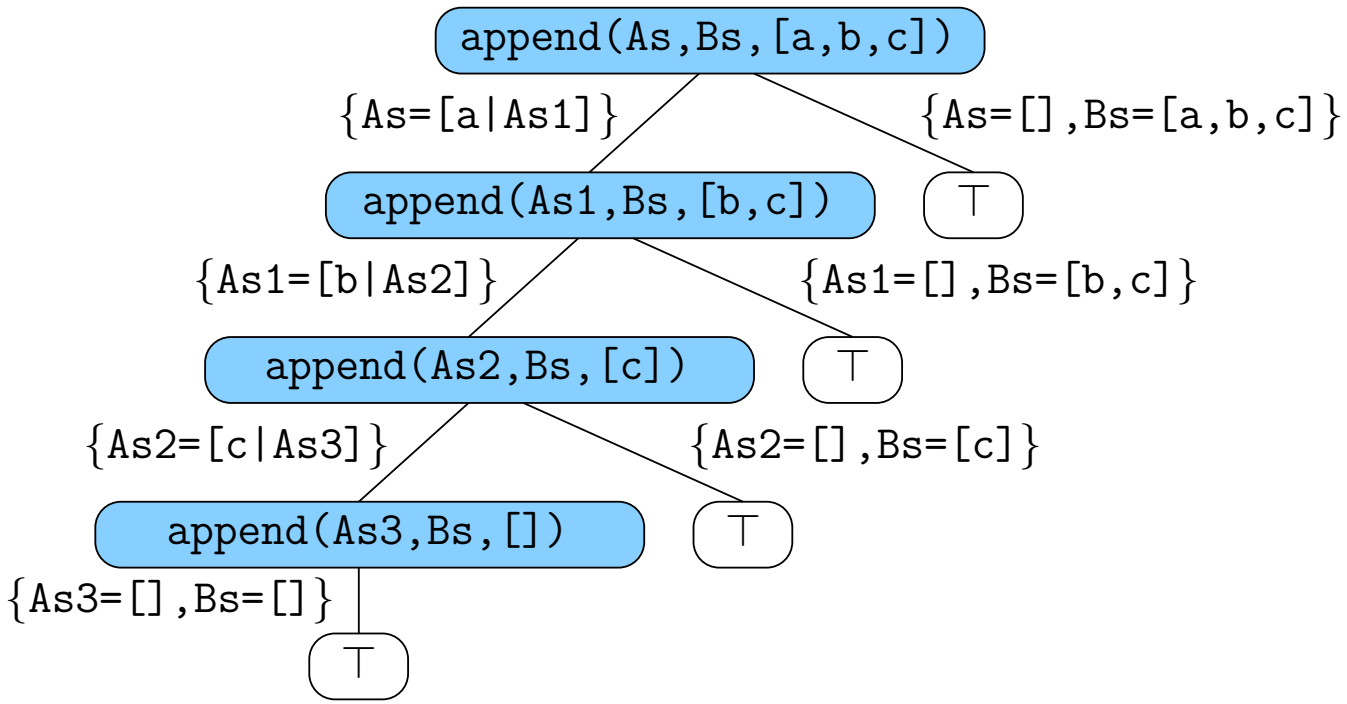
Prolog-Suchbaumdurchläufe



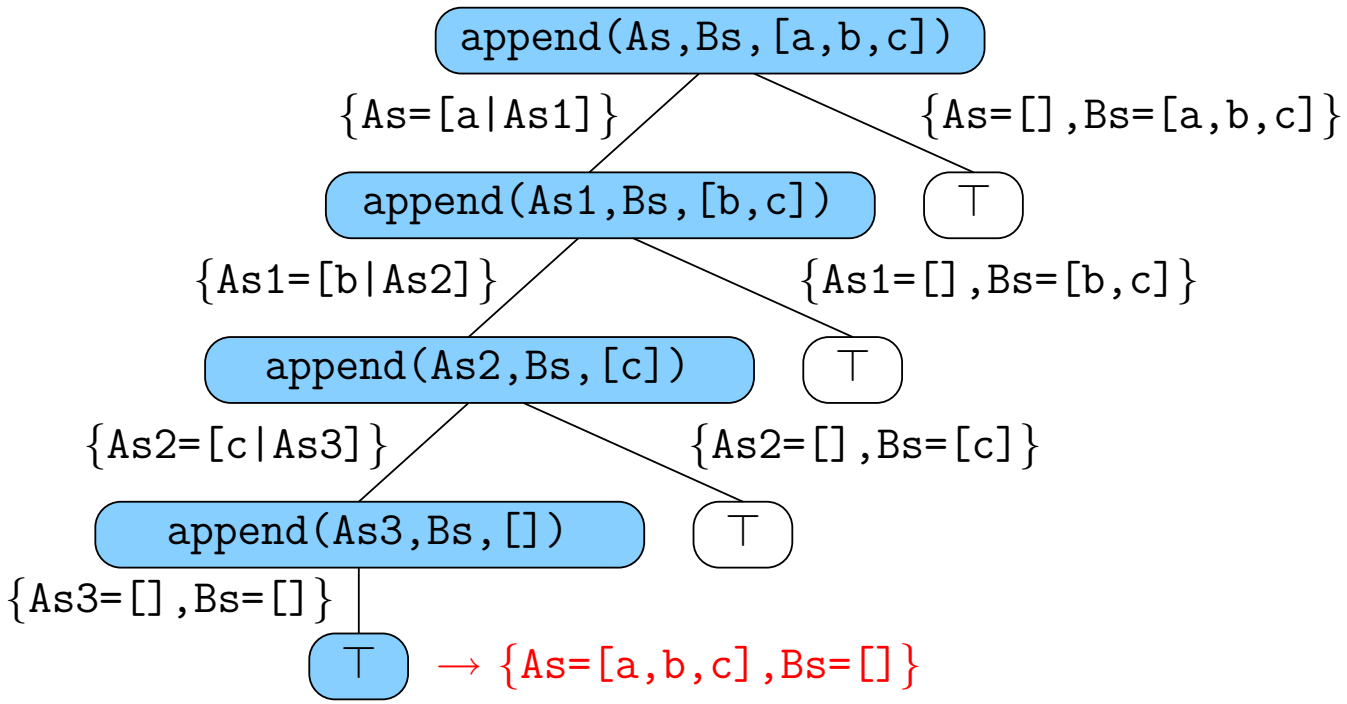
Prolog-Suchbaumdurchläufe



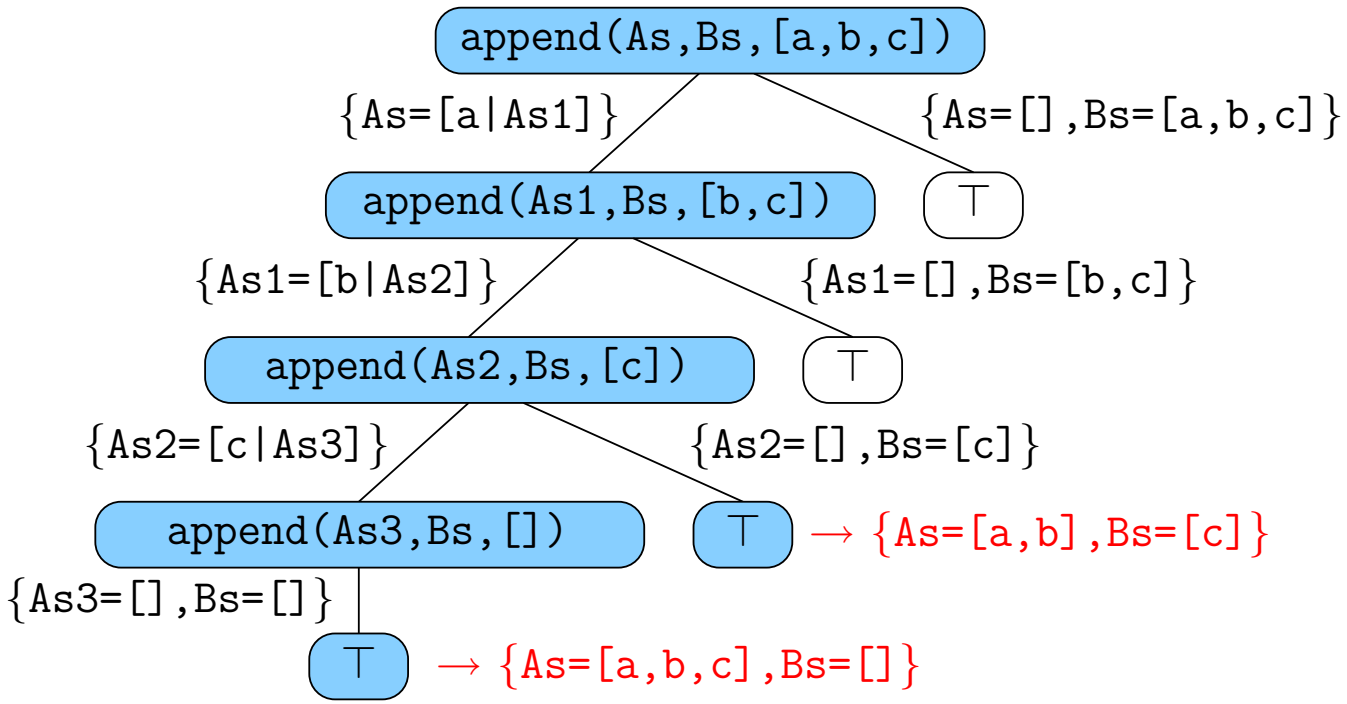
Prolog-Suchbaumdurchläufe



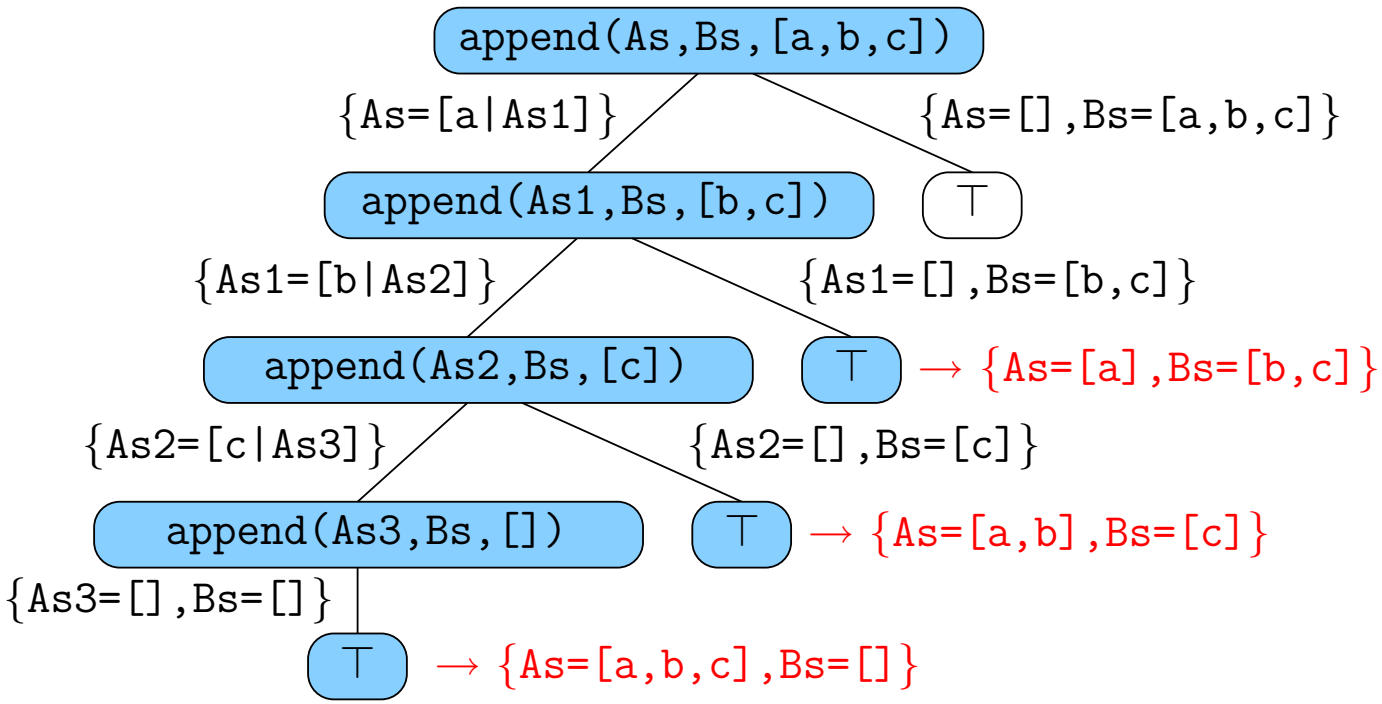
Prolog-Suchbaumdurchläufe



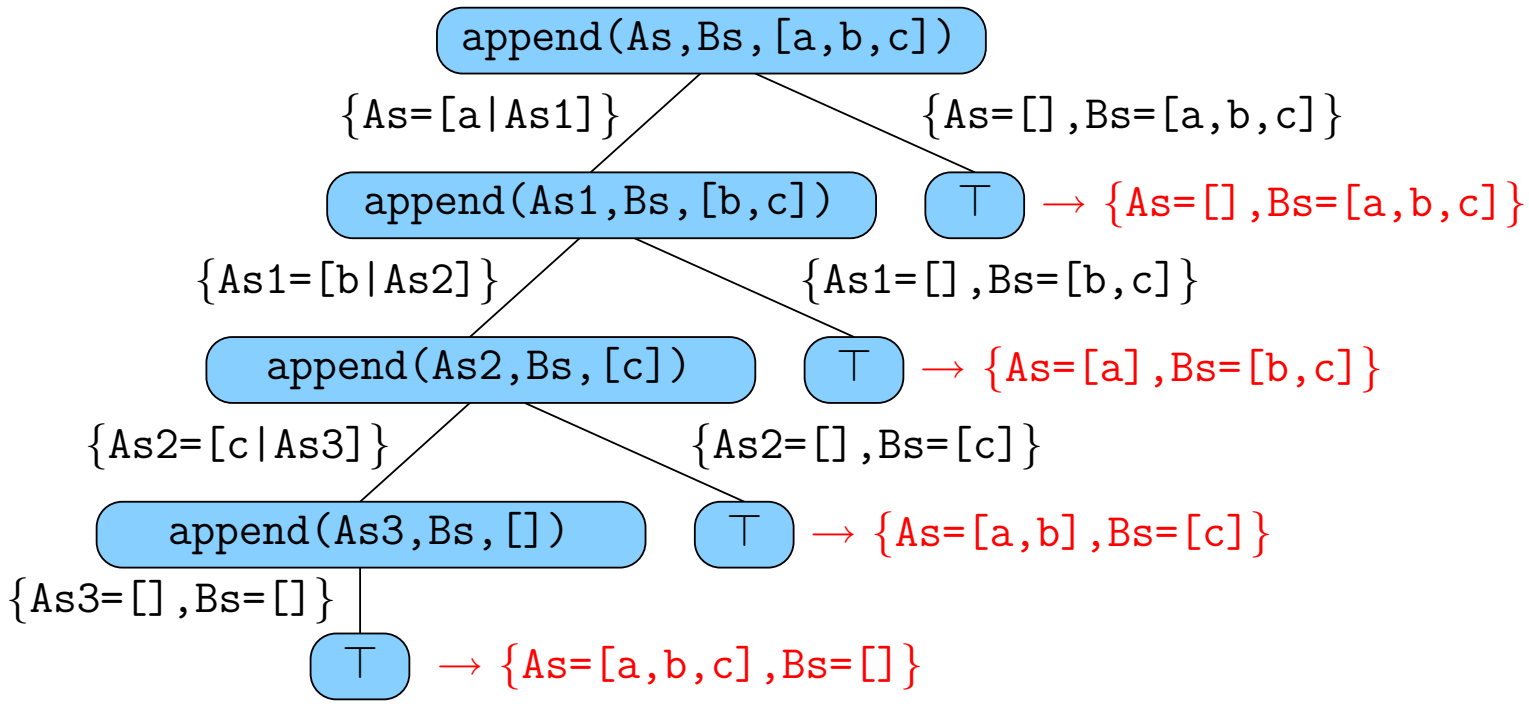
Prolog-Suchbaumdurchläufe



Prolog-Suchbaumdurchläufe



Prolog-Suchbaumdurchläufe



Folgen der Tiefensuche

- Die Tiefensuche ist effizient und einfach zu implementieren, aber...
- Unter Umständen ist der zuerst gefolgte Pfad unendlich, so dass andere eventuell existierende Erfolgsknoten nicht mehr erreicht werden können.
- Durch die Änderung der Reihenfolge der Klauseln und Literale kann erreicht werden, dass unendliche Berechnungen vermieden werden oder später auftretten.

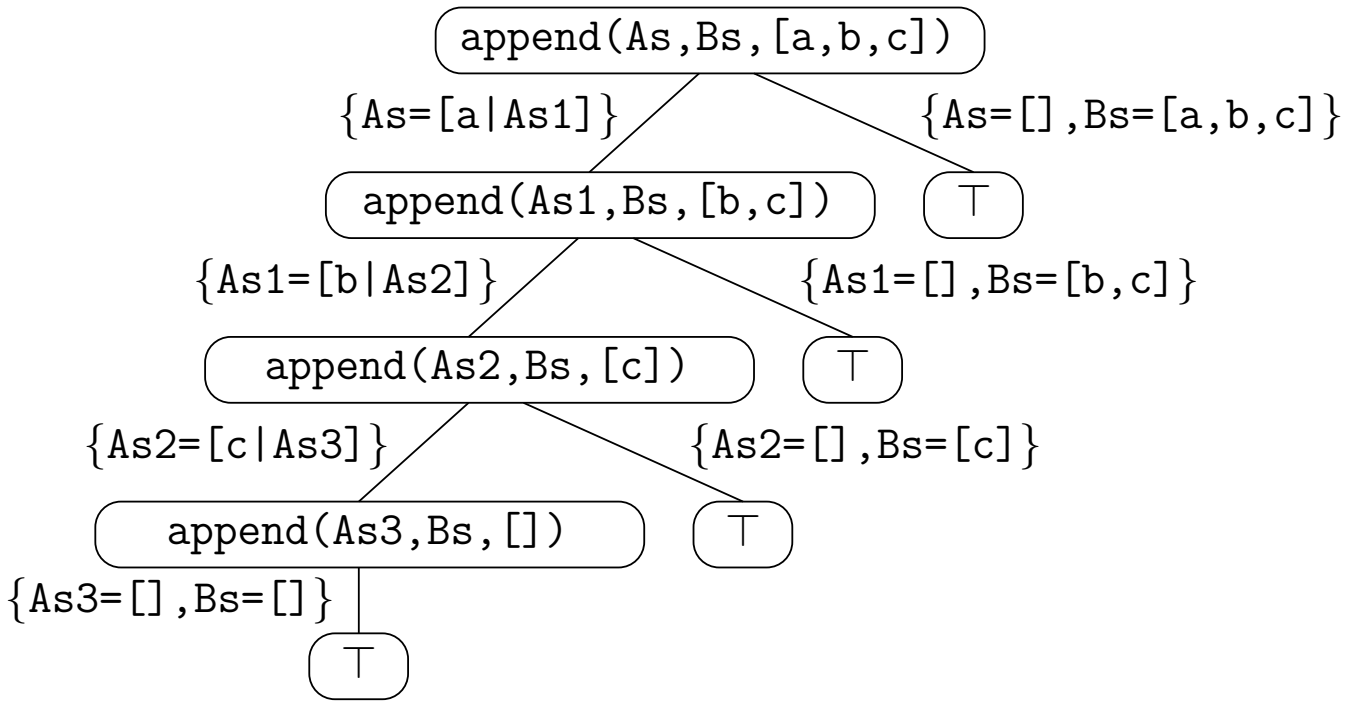
$\Longrightarrow$ Deklarativität der Logikprogrammierung wird (zugunsten der Effizienz) verletzt.

5.6.1 Folgen der Auswahlstrategie in Prolog

- Aus Sicht der LP-Programmierung ist die Reihenfolge der Klauseln und der Ziele irrelevant.
- Die Effizienz der Prolog-Programme allerdings hängt oft maßgeblich von dieser Reihenfolge ab.
- Im Extremfall (oft) terminieren korrekte LP-Programme nicht für eine bestimmte Anordnung der Klauseln und der Ziele.

Reihenfolge in der Lösungen gefunden werden

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



Reihenfolge in der Lösungen gefunden werden

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

```
?- append(As,Bs,[a,b,c]).
As = [a, b, c]
Bs = [] ;

As = [a, b]
Bs = [c] ;

As = [a]
Bs = [b, c] ;

As = []
Bs = [a, b, c] ;
No
```

Reihenfolge in der Lösungen gefunden werden

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

```
?- append(As,Bs,[a,b,c]).
As = []
Bs = [a, b, c] ;

As = [a]
Bs = [b, c] ;

As = [a, b]
Bs = [c] ;

As = [a, b, c]
Bs = [] ;
No
```

Terminierung

Rekursive Klauseln können zu Nichtterminierung führen.

Bsp.: 1. Versuch, eine kommutative Relation zu definieren.

```
married(X,Y) :- married(Y,X).
married(abraham,sarah).
```

```
?- married(abraham,sarah).
```

Nichtterminierende Berechnung

Grund:

```
married(abraham,sarah)
   married(sarah,abraham)
      married(abraham,sarah)
         married(sarah,abraham)
            ⋮
```

Terminierung

Rekursive Klauseln können zu Nichtterminierung führen.

Bsp.: 2. Versuch, eine kommutative Relation zu definieren.

```
married(abraham,sarah).
married(X,Y) :- married(Y,X).
```

```
?- married(abraham,sarah).
Yes
?- married(sarah,abraham).
Yes
?- married(lot,sarah).
Nichtterminierende Berechnung
```

Terminierung

- Kommutative Relationen können mit einem neuen Prädikat definiert werden, das eine Klauseln für jede Permutation der Argumente der Relation hat:

```
are_married(X,Y) :- married(X,Y).
are_married(X,Y) :- married(Y,X).
married(abraham,sarah).
```

```
?- are_married(abraham,sarah).
Yes
?- are_married(sarah,abraham).
Yes
?- are_married(sarah,lot).
No
```

Anordnungen der Literale

- Die Anordnungen der Literale bestimmen den Prolog-Suchbaum.
  (im Unterschied zur Anordnung der Klausel, die nur die
  Reihenfolge ändert, in der Teilbäume besucht werden sollen.)
  - ▷ kann die Effizienz maßgeblich beeinflußen.
  - ▷ kann bestimmen, ob eine Berechnung terminiert oder nicht.

Anordnung der Literale und Effizienz

- Bsp.: Berechnung für `son(X,lot)`
- 1. Möglichkeit: `son(X,Y) :- male(X),parent(Y,X).`
  $\longrightarrow$ man betrachtet die Männer nacheinander und prüft, ob sie Kinder von `lot` sind.
- 2. Möglichkeit: `son(X,Y) :- parent(Y,X),male(X).`
  $\longrightarrow$ man betrachtet die Kinder von `lot` nacheinander und prüft, ob sie Männer sind.

$\Longrightarrow$ Die zweite Anordnung ist günstiger für die Anfrage `son(X,lot)` aber...

- ...die erste ist besser für `son(sarah,X)`

$\Longrightarrow$ Die optimale Anordnung hängt von der beabsichtigten Benutzung ab.

Anordnung der Literale und Effizienz

$\Longrightarrow$ Heuristik: Literale deren Ableitung effizient ist (z.B. arithmetische Teste), sollten möglichst links, vor anderen Literalen (insbesondere vor rekursiven) Atomen stehen.

Beispiel: Eine Prozedur `partition(Liste,Pivot,Kleinere,Groessere)` kann benutzt werden, um eine Liste in zwei Listen der Elemente, die kleiner bzw. größer als ein Pivot sind. ($\longrightarrow$ Quicksort).

- Eine Klausel, die die Prozedur definiert könnte so aussehen:

```
partition([X|Xs],Y,[X|Ks],Gs) :- X<=Y,partition(Xs,Y,Ks,Gs)
```

- Diese führt i.A. zu effizienteren Berechnungen als:

```
partition([X|Xs],Y,[X|Ks],Gs) :- partition(Xs,Y,Ks,Gs),X<=Y
```

Anordnung der Literale und Terminierung

Die Anordnung der Literale kann über Terminierung entscheidend sein.

```
quicksort([X|Xs],Ys) :-
        partition(Xs,X,Kleinere,Groessere),
        quicksort(Kleinere,Ls),
        quicksort(Groessere,Bs),
        append(Ls,[X|Bs],Ys).
```

$\Longrightarrow$ terminiert, weil die rekursive Sortierung auf die kleineren Listen `Kleinere` bzw. `Groessere` angewendet wird.

```
quicksort([X|Xs],Ys) :-
        quicksort(Kleinere,Ls),
        quicksort(Groessere,Bs),
        partition(Xs,X,Kleinere,Groessere),
        append(Ls,[X|Bs],Ys).
```

$\Longrightarrow$ terminiert nicht.

Rednundante Lösungen

Prolog gibt für jede erfolgreiche Ableitung eine Antwort aus, wenn mehrere Klauseln für den selben Fall zuständig sind, z.B.:

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

```
?- append([1],[2,3],X).
X = [1, 2, 3] ;
X = [1, 2, 3] ;
No
```

5.6.2 Arithmetik in Prolog

- Systemprädikate (*builtin Prädikate,bips*) sind Prädikate, die vom implementierenden System direkt unterstützt werden, statt mit Hilfe von Klauseln definiert zu sein.

$\Longrightarrow$ Effizienz, dafür Eingeschränkungen bezüglich ihrer Benutzung.

- Arithmetische Systemprädikate liefern Zugang zur effizienten, maschinenunterstützten arithmetischen Funktionalität.

Auswertung arithmetischer Ausdrücke: das Prädikat `is`

Zur Evaluierung eines arithmetischen Ausdrucks benutzt man das infixierte Prädikat `is(Wert, Ausdruck)` d.h.: `Wert is Ausdruck`

- Prolog-Interpretierung des Zieles:
  1. Wenn `Ausdruck` unter der aktuellen Variablensubstitution zu einem Wert v ausgewertet werden kann, liefert die Unifikation von v und `Wert` das Ergebnis der Ableitung des Zieles.
  2. Sonst schlägt das Ziel fehl.
- Beispiele:

  `X is 1+2` $\longrightarrow$ Antwort: `X=3`.

  `3 is 1+2` $\longrightarrow$ Antwort: `yes`.

  `1+2 is 1+2` $\longrightarrow$ Antwort: `no`. (Grund: `1+2` und `3` unifizieren nicht.)

Auswertung arithmetischer Ausdrücke: das Prädikat `is`

Gründe warum ein `Ausdruck` in `Wert is Ausdruck` nicht auswertbar sein könnte:

- `Ausdruck` ist kein arithmetischer Ausdruck, z.B. `1+x`
  $\Longrightarrow$ Das Ziel scheitert. (*failure*)
- `Ausdruck` benutzt Variablen, die bei der Auswertung (noch) nicht belegt sind, z.B. `1+Y`, wenn noch keine Substitution von `Y` im Laufe der aktuellen Ableitung vorliegt.
  $\Longrightarrow$ Laufzeitfehler (*error condition*)

Das Prädikat `is`

- Vorsicht: `is` dient nicht der Zuweisung eines Wertes an eine Variable.
- X is X+1 schlägt fehl oder führt zu einem Laufzeitfehler – immer.

Arithmetische Vergleiche

- `1+2 <= 6-3`
  - ▷ Linke Seite wird ausgewertet $\longrightarrow$ 3;
  - ▷ Rechte Seite wird ausgewertet $\longrightarrow$ 3;
  - ▷ 3 und 3 unifizieren $\longrightarrow$ Antwort `yes`.
- Andere Vergleichsoperatoren: `>=`,`<`, `>`, `=:=` (Gleichheit), `=/=` (Ungleichheit).

Arithmetik in Prolog: Beispiel

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

```
?- factorial(3,X).
X = 6 ;
No
?- factorial(X,6).
ERROR: Arguments are not sufficiently instantiated
```

Arithmetik in Prolog: Beispiel

Effizientere Implementierung der Fakultätsfunktion:

```
factorial(N,F) :- factorial(0,N,1,F).
factorial(I,N,T,F) :-
        I < N, I1 is I+1, T1 is T*I1, factorial(I1,N,T1,F).
factorial(N,N,F,F).
```

5.6.3 Explizite Kontrolle der Zielauswertung

- Prolog stellt ein Systemprädikat, die das Backtracking bei der Suche von Prolog steuern kann: *cut*, geschrieben !.
- Ziel eines cut ist den Suchaufwand für einer Berechnung zu reduzieren, indem man einen Zweig des Suchbaumes abzuschneidet.
  - ▷ green cuts: schneiden Zweige ab, die keine Lösungen enthalten $\Longrightarrow$ erhöhen die Effizienz;
  - ▷ red cuts: schneiden Zweige ab, die Lösungen enthalten.

Cuts in Prologprogrammen

- Da cuts die Bedeutung eines Programms von der prozeduralen Interpretierung zusätzlich abhängig machen, verletzten sie die strebenswerte Deklarativität.
  - ▷ green cuts: nützlich als Kompromiss zwischen Effizienz und Deklarativität.
  - ▷ red cuts: eher unerwünscht.

Cuts: Bedeutung

- Cut ist ein nullstelliges Prädikat, das immer erfüllt ist. Wird das Cut im Laufe der Ableitung eines aktuellen Zieles A' mit Hilfe einer Klausel

$$A \leftarrow A_1, \ldots, A_k, !, A_{k+1}, \ldots, A_n$$

erfüllt (wobei A und A' unifizieren), so werden Alternative zur Erfüllung von A, A_1, $\ldots$, A_k im Laufe der aktuellen Ableitung von A' ausgeschlossen. D.h.:

- ▷ alternative Klauseln, deren Kopf mit A' unifizieren werden ignoriert;
- ▷ Wenn die Ableitung von A_i mit $i \geq k+1$ im Laufe der weiteren Ableitung von A' fehlschlägt, werden im Laufe des Backtracking alternative Ableitungen nur soweit zurückverfolgt bis zum !.