Merkmale imperativer und funktionaler Programmiersprachen

Imperative PS:

- Problemlösung durch Formulierung einer Folge von Anweisungen
- Variablenkonzept: Speicherung von Eingabewerten in Variablen und anschließende Weiterverarbeitung.

Sprachen: ADA, C, Fortran, Modula-2, ...

Funktionale PS:

- Problemlösung durch ein System von Funktionsdefinitionen
- Funktionsdefinition durch Zusammensetzen aus einfacheren Funktionen
- Anwendung von Funktionen auf ihre Parameter

  $\rightarrow$ Programm = Funktionen von Mengen von Eingabewerten in Mengen von Ausgabewerten

Sprachen: LISP, Scheme, ...

Für imperative und funktionale PS gilt:

Formulierung von *"wie geht das"* – Wissen

Merkmale einer deklarativen Programmiersprache

- Programm = Menge von Tatsachen (Fakten) und
  Schlußfolgerungen (Regeln)
- Programmierer formuliert sein Wissen über ein Problem als Menge von wahren Aussagen.
- Problembeschreibung in Form von Objekten und Beziehungen zwischen den Objekten.
- Problemlösung durch Beweisen (bzw. Widerlegen) von Hypothesen (Anfragen) an die Datenbasis mit Hilfe einer Inferenzmaschine

  → *logisches Schließen*

  Die Datenbasis wird daher als *Logikprogramm* bezeichnet.

Für deklarative PS gilt:

Formulierung von *"was ist"* – Wissen

Beispiel – Logikprogramm

Die Datenbasis enthalte folgende Fakten:

```
(istKind maria eva)
(istKind maria paul)
(istKind paul willy)
(istKind heinz eva)
(verheiratet paul eva)
(verheiratet heinz sandra)
```

Beispielanfragen:

```
?- (istKind paul willy)  =>  yes

?- (istKind maria ?x)    =>  ?x = eva;
                             ?x = paul.

?- (istKind ?x eva)      =>  ?x = maria;
                             ?x = heinz.

?- (istKind ?x ?y)     =>  ?x = maria , ?y = eva;
                           ?x = maria , ?y = paul;
                           ?x = paul  , ?y = willy;
                           ?x = heinz , ?y eva.

?- (verheiratet peter maria) =>  no
```

Regeln in Prolog

Die Datenbasis enthalte die Fakten:

```
(istKind maria eva)
(istKind heinz eva)
(verheiratet paul eva)
```

Die Vater-Kind–Beziehung kann durch folgende Regel ausgedrückt werden:

```
((istVater ?va ?ki) <- (istKind ?ki ?mu)
                       (verheiratet ?va ?mu))
```

Beispielanfragen:

```
?- (istVater paul maria)   =>  yes

?- (istVater ?x maria)     =>  ?x = paul.

?- (istVater ?x ?y)    =>  ?x = paul , ?y =maria;
                           ?x = paul , ?y = heinz.

?- (istVater franz sandra) =>  no
```

Unifikation

"Gleichmachen" zweier Muster.

Definition – Unifikator (nach [Schöning95]):

> Eine Substitution sub (also eine Ersetzung von evtl. mehreren Variablen durch Terme) ist ein *Unifikator* einer (endlichen) Menge von Literalen $L = \{L_1, L_2, \ldots, L_k\}$, falls $L_1 sub = L_2 sub = \ldots = L_k sub$. D.h., durch Anwenden der Substitution sub auf jedes Literal in L entsteht ein und dasselbe Literal.

Beispiele:

```
(unify '(a ?x) '(?y b))            =>  ((?x . b) (?y . a))
(unify '?x '?y)                    =>  ((?x . ?y))
(unify '(f ?x) '(f ?y))            =>  ((?x . ?y))
(unify '(?x ?x) '(?y ?y))          =>  ((?x . ?y))
(unify '(?x ?y) '(?y ?x))          =>  ((?x . ?y))
(unify '(?x ?y a) '(?y ?x ?x))     =>  ((?y . a) (?x . ?y))
(unify '?x '(f ?x))                =>  ((?x . (f ?x)))
```

Unifikationsalgorithmus

```
(defun unify (x y &optional (bindings no-bindings))
  "unifiziere x und y unter gegebenen Bindungen"
  (let ((x (value x bindings))       ;; bestimme aktuelle Werte
        (y (value y bindings)))      ;; ueber Bindungen
    (cond ((eq bindings fail) fail)   ;; bereits gescheitert
          ((eql x y) bindings)        ;; schon gleiche Werte
          ;; es entstehen neue Variablenbindungen ->
          ((variable-p x) (extend-bindings x y bindings))
          ((variable-p y) (extend-bindings y x bindings))
          ;; unifiziere Baeume ->
          ((and (consp x) (consp y))
           (unify (rest x) (rest y)
                  (unify (first x) (first y) bindings)))
          (t fail))))    ;; keine Chance auf Unifikation
```

Grundprinzip eines Prolog-Interpreters

Der Beweis einer Anfrage an eine Datenbasis (Logikprogramm) erfolgt — vereinfacht — durch folgende Schritte:

1. Suche eine Regel aus der Datenbasis, deren Kopf sich mit dem zu beweisenden Prädikat (*goal*) unifizieren läßt (Fakten sind Regeln ohne Rumpf).

2. Beweise die Prädikate des Regelrumpfs unter Berücksichtigung der in Schritt 1 hinzugekommenen Variablenbindungen.

Der Beweis der Anfrage ist erfolgreich, wenn keine zu beweisenden *goals* mehr übrig sind.

Voraussetzung: Zurückführen der Prädikate auf Fakten, da sonst durch Schritt 2 die Menge der zu beweisenden *goals* nie leer wird.

Scheitert Schritt 1, so ist das entsprechende *goal* unter den aktuellen Variablenbindungen nicht beweisbar.

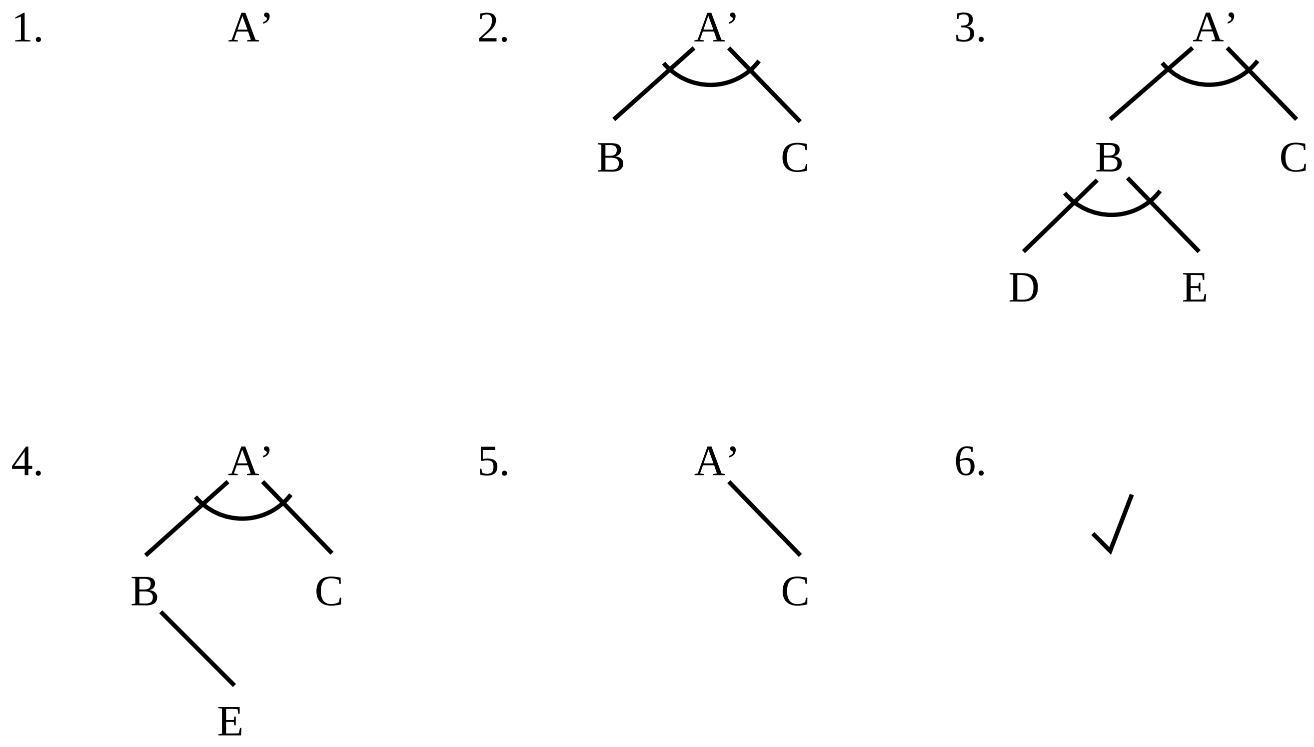
Beispielablauf eines Anfragebeweises

Gegeben seien die Fakten und Regeln:

```
C
D
E
(A <- B C)
(B <- D E)
```

`A`, `B`, `C`, `D` und `E` stehen stellvertretend für Prädikate.

Die Anfrage `A'` führt zu folgendem Beweisablauf:



Konkretes Beispiel eines Anfragebeweises

Gegeben sei:

```
(istKind maria hans)
(istMutter maria klaus)
(istMutter maria eva)
((istVater ?va ?ki) <- (istKind ?ki ?va))
((istGrossvater ?gr ?en) <- (istVater ?gr ?ki)
                            (istMutter ?ki ?en))
```

Anfrage: `(istGrossvater hans ?x)`

(istGrossvater hans ?x)

(istGrossvater ?gr ?en)

(istVater ?gr ?ki)

(istVater ?va ?ki$_1$)

(istKind ?ki$_1$?va)

(istKind maria hans)

(istMutter ?ki ?en)

(istMutter maria klaus)

$\Rightarrow$?x = klaus

Backtracking

Bedeutung:

Versuche ein *goal* erneut zu lösen, mit einer anderen Regel oder einem anderen Fakt aus der Datenbasis.

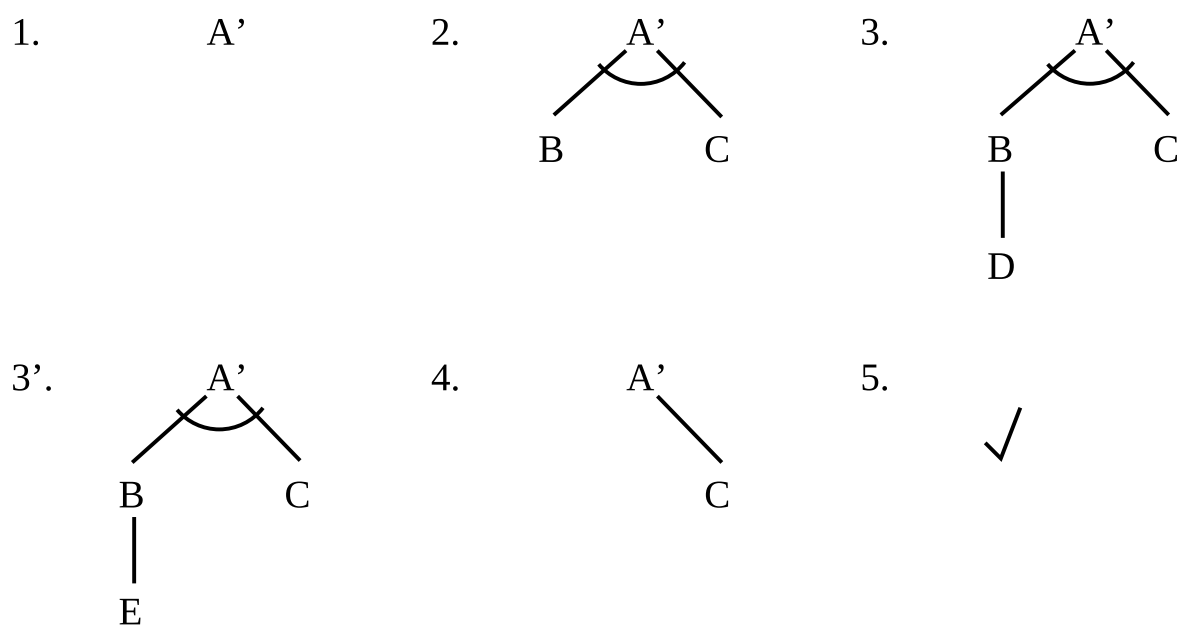
Gründe für Backtracking:

1. Der Beweis des *goals* mit der zuvor gewählten Regel (Fakt) ist gescheitert.
2. Es soll nach weiteren Lösungsmöglichkeiten für eine Anfrage gesucht werden.

Beispiel:

```
(A <- B C)
(B <- D)
(B <- E)
C
E
```

Anfrage: `A'`



Prolog-Interpreter in Lisp — Implementierung I

Konstante:

`fail` zeigt Scheitern der Unifikation an

Hilfsfunktionen:

`(predicate <relation>)` extrahiert das Prädikat aus einer Relation

`(get-clauses <pred>)` sucht zu Prädikat gehörende Regeln und Fakten aus Datenbasis

`(rename-variables <clause>)` benennt in `<clause>` die Variablen um. Sichert dadurch lokalen Gültigkeitsbereich der Regelvariablen.

`(show-result <bindings>)` gibt die Bindungen der Variablen aus der Anfrage aus, sobald eine Lösung gefunden wurde. Löst Backtracking aus, falls der Benutzer weitere Lösungen wünscht (durch Rückgabewert `fail`).

Hauptfunktionen:

`(prove <goal> <bindings> <other-goals>)` sucht eine Lösung für `goal` unter Berücksichtigung aktueller Bindungen und beweist mit ihr `other-goals`.

`(prove-all <goals> <bindings>)` sucht eine Lösungen für eine Konjunktion von `goals` mit gegebenen Bindungen.

Prolog-Interpreter in Lisp — Implementierung II

```
(defun prove-all (goals bindings)
  "Finde eine Loesung fuer die Konjunktion von goals."
  (cond ((eq bindings fail) fail)
        ((null goals) (show-result bindings))
        (t (prove (first goals) bindings (rest goals)))))
```

```
(defun prove (goal bindings other-goals)
  "Finde eine Menge von Loesungsmoeglichkeiten fuer goal."
  (some #'(lambda (clause)
            (let ((new-clause (rename-variables clause)))
              (prove-all
                (append (clause-body new-clause) other-goals)
                (unify goal (clause-head new-clause) bindings))))
        (get-clauses (predicate goal))))
```