Beispiel (Anfragen mit Variablen)

```
?- father(X,jeff). -> X = bert
```

Eine Anfrage kann mehrere Lösungen liefern:

```
?- sibling(A,B).  ->  A = jeff    B = george
                      A = george  B = jeff
                      A = cindy   B = vic
                      A = vic     B = cindy
```

Beispiel (zusammengesetzte Anfrage)

```
?- grandparent(john,X), parent(Y,X), sibling(Y,jeff).
                              -> X = cindy  Y = george
                                 X = vic    Y = george
```

Formale Differentiation in Prolog

Beispiel

```
d(X, U+V, DU + DV) :- d(X,U,DU), d(X,V,DV).
d(X, U-V, DU - DV) :- d(X,U,DU), d(X,V,DV).
d(X, U*V, DU*V + U*DV) :- d(X,U,DU), d(X,V,DV).
d(X, U/V, (DU*V - U*DV)/V^2) :- d(X,U,DU), d(X,V,DV).
d(X, U^C, C*U^(C-1)*DU) :- d(X,U,DU), atomic(C), C\=X.
d(X,X,1).
d(X,C,0) :- atomic(C), C\=X.
```

Zahlen in Prolog

In Prolog gibt es fast keine Datenstrukturen im üblichen Sinn, sondern Datenstrukturen werden durch ihre Eigenschaften implizit definiert.

Definition (Addition natürlicher Zahlen in Prolog)

```
sum(succ(X), Y, succ(Z)) :- sum(X,Y,Z).
sum(0, X, X).
dif(X,Y,Z) :- sum(Z,Y,X).
```

Bemerkung

`sum` *und* `dif` *legen die Eigenschaften der Summe und der Differenz fest.*
`succ(X)` *steht dabei für den Nachfolger einer Zahl* `X`*;*
`succ` *ist ein Funktionssymbol.*

Zahlen in Prolog

Die Summe $2 + 3 = A$ wird dann so formuliert:

```
?- sum(succ(succ(0)), succ(succ(succ(0))), A).
                    -> A = succ(succ(succ(succ(succ(0)))))
```

Diese Definition kann man auch „anders herum“ benutzen: $2 + B = 5$

```
?- sum(succ(succ(0)), B, succ(succ(succ(succ(succ(0)))))).
                    -> B = succ(succ(succ(0)))
```

oder sogar $A + B = 5$:

```
?- sum(A, B, succ(succ(succ(succ(succ(0)))))).
          -> A = succ(succ(succ(succ(succ(0))))) B = 0
             A = succ(succ(succ(succ(0)))) B = succ(0)
             A = succ(succ(succ(0))) B = succ(succ(0))
             A = succ(succ(0)) B = succ(succ(succ(0)))
             A = succ(0) B = succ(succ(succ(succ(0))))
             A = 0 B = succ(succ(succ(succ(succ(0)))))
```

Listen in Prolog

Definition

```
hd(cons(X, _), X).
tl(cons(_, L), L).

list(nil).
list(cons(_, L)) :- list(L).

null(nil).
```

Das Zeichen „_“ bezeichnet dabei eine neue („anonyme“) Variable, deren Name nicht weiter interessiert.

- `nil` bezeichnet die leere Liste.
- `cons` fügt ein Element vor die Elemente einer existierenden Liste ein.
- `hd` liefert das erste Element einer Liste, die Operation `tl` die Restliste.

Darstellung von Listen in Prolog

Definition („syntaktischer Zucker“)

- Listen werden durch `[a1, ... , an]` bezeichnet.
- Die Notation `[X|L]` entspricht `cons(X, L)`.

Bemerkung

Man könnte sich folgende alternative Definition der Prädikate vorstellen:

```
hd([X|_], X).
tl([_|L], L).
?- hd([[a,b],c,d],X).  ->  X = [a,b]
```

Verkettung zweier Listen in Prolog: append

Definition

```
append([],L,L).
append([X|L],M,[X|N]) :- append(L,M,N).
```

Beispiel (Anfragen)

```
?- append([a,b],[c,d],[a,b,c,d]). -> yes
?- append([a,b],[c,d],Ans). -> Ans = [a,b,c,d]
?- append(L,[c,d],[a,b,c,d]). -> L = [a,b]
?- append(X,Y,[a,b,c,d]).   ->   X = []   Y = [a,b,c,d]
                                 X = [a]    Y = [b,c,d]
                                 X = [a,b]    Y = [c,d]
                                 X = [a,b,c]    Y = [d]
                                 X = [a,b,c,d]   Y = []
```

Das cut-Prädikat „!“

Zur Steuerung des Backtrackings und zur gezielten Effizienzsteigerung gibt es in Prolog das Prädikat cut (Schreibweise „!“), das stets den Wert **true** hat.

Ein cut friert die gemachte Auswahl der Klausel ein, d.h. wird beim Backtracking ein cut erreicht, dann werden weitere Alternativen ausgeschlossen, also der Baum beschnitten!

Durch Abarbeitung des Prädikats „!“ werden alle noch offenen Alternativen für diese Klausel beim Backtracking gelöscht.

Das cut-Prädikat „!“

Beispiel

Ohne cut würde das folgende Programm `amy` **und** `bob` liefern:

```
ancestor(X,Y) :- parent(X,Z), !, ancestor(Z,Y).
ancestor(X,X).
parent(amy,bob).
?- ancestor(A,bob).   ->   A = amy
```

und die folgende Variante liefert:

```
ancestor(X,Y) :- !, parent(X,Z), ancestor(Z,Y).
ancestor(X,X).
parent(amy,bob).
?- ancestor(A,bob).   ->   no
```

Bemerkungen zur Übersetzung von Prolog

Die Übersetzung eines Prolog-Programms kann man sich –ganz grob– stufenweise vorstellen:

- Übersetzung von atomaren Formeln bzw. Literalen (als Prämissen),
- Übersetzung von Horn-Klauseln,
- Übersetzung von Prädikaten.

Ein **Literal** ist eine atomare Formel oder ihre Negation.
Ein Literal $q(t_1, \ldots, t_k)$ kann man wie einen Prozeduraufruf übersetzen:
Es wird ein Kellerrahmen angelegt, die Parameter werden gespeichert und es wird das Prädikat q/k aufgerufen.

Eine **Horn-Klausel** $r \equiv q(X_1, \ldots, X_k) \Leftarrow g_1, \ldots, g_n$
enthalte die Variablen $\{X_1, \ldots, X_m\}$, $m \geq k$.
Der Code für eine Klausel muss zuerst Platz für diese lokalen Variablen der Klausel reservieren. Hier wird die **Unifikation** durchgeführt. Anschließend benötigen wir Code zur Auswertung des Rumpfs. Schließlich sollte der Kellerrahmen freigegeben werden – immer, wenn das möglich ist.

Ein (k-stelliges) **Prädikat** q/k wird definiert durch eine *Folge* von Klauseln $rr \equiv r_1 \ \ldots \ r_f$.
Falls q/k durch mehrere Klauseln definiert ist, muss die erste Alternative angesprungen werden. Schlägt diese fehl, werden der Reihe nach die weiteren Alternativen probiert.

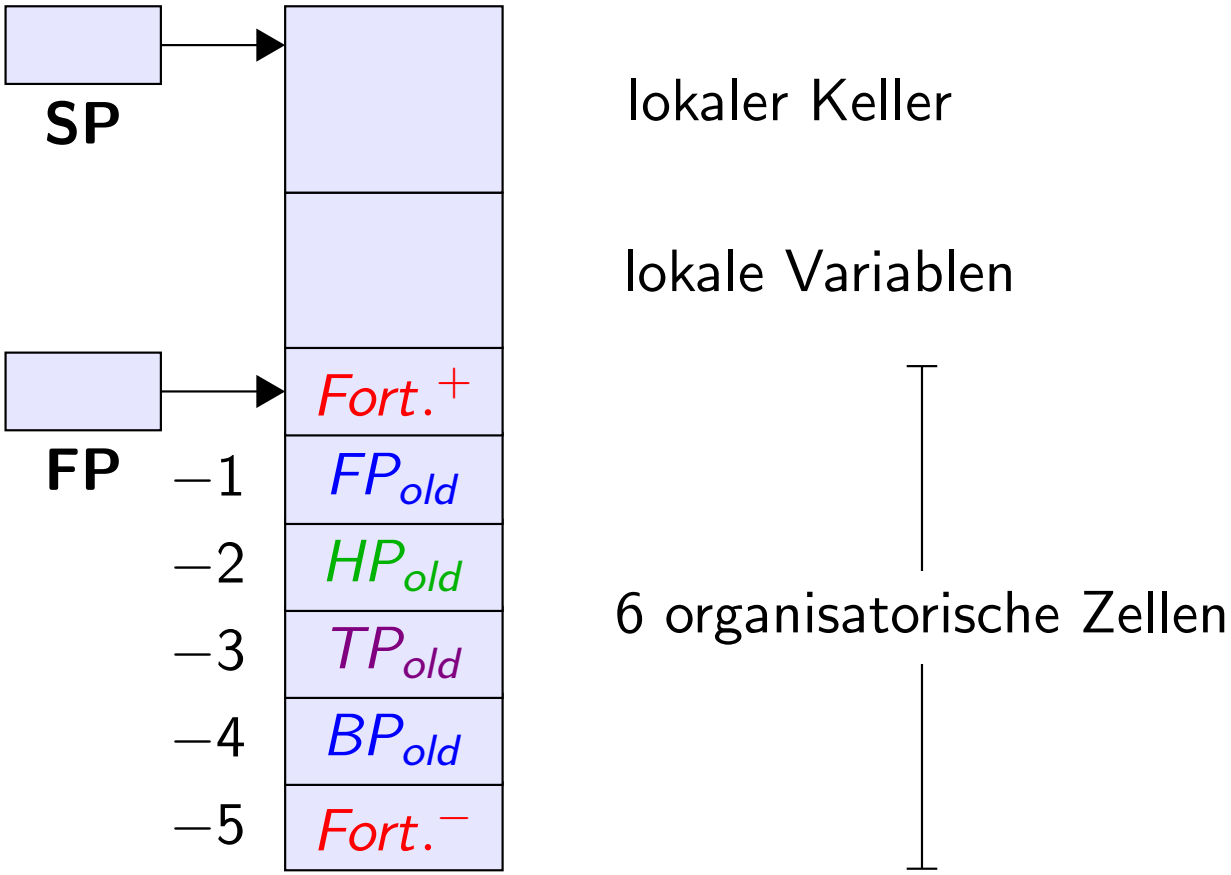
Wir benötigen dazu eine Implementierung, die einzelne Alternativen versuchsweise ausführt und bei Fehlschlag ihren gesamten Effekt wieder rückgängig macht (*backtracking*).

Wird das Backtracking aktiviert, soll die gesamte Berechnung bis zum *dynamisch* letzten Ziel rückgängig gemacht werden, an dem eine alternative Klausel gewählt werden kann. Den zugehörigen Kellerrahmen auf dem Laufzeitkeller nennen wir den aktuellen **Rücksetzpunkt**.
Ein neues Register, der **Rücksetzzeiger** *BP* (*backtrack pointer*), zeigt auf den aktuellen Rücksetzpunkt.

Um zwischenzeitlich eingegangene Variablenbindungen aufzuheben, werden die eingegangenen neuen Bindungen protokolliert, indem die zugehörigen Variable in einer speziellen Datenstruktur, der **Spur** (*trail*) gespeichert werden. Die Spur ist ein weiterer Keller T, auf dem Haldenadressen von Referenzobjekten gemerkt werden, die eine neue Bindung erhalten.
Ein weiteres Register, der **Spurzeiger** *TP* (*trail pointer*), zeigt stets auf die oberste belegte Spurzelle.

Die Organisation eines Kellerrahmes wird dadurch aufwändiger:

- Wie bisher benötigen wir eine organisatorische Zelle zur Rettung des aktuellen Standes des *PC*, der **positiven Fortsetzungsadresse**, d.h. der Stelle im Code, an der bei erfolgreicher Abarbeitung des Literals fortgefahren werden soll. Dies entspricht der Rücksprungadresse.
- Ebenso retten wir wieder den Inhalt des Registers *FP*, den Verweis auf den **dynamischen Vorgänger**.
- Im Kellerrahmen benötigen wir zusätzlich die **Codeadresse für die nächste Alternative**, d.h. die **negative Fortsetzungsadresse** sowie den alten Wert des *BP*.
- Weiterhin retten wir an einem Rücksetzpunkt die alten Werte der Register *TP* und *HP*.



Der Kellerrahmen der WiM.

Funktional-logische Sprachen

Funktionale Sprachen bieten durch Funktionen mit ihren ein-elementigen Ergebnissen deterministische Effizienz.

Geschachtelte Ausdrücke und **Funktionale** ermöglichen knappe, elegante Programme
und die Bedarfsauswertung erlaubt nicht nur **rekursive**, sondern sogar **unendliche Datenstrukturen**.

Ausdrücke einer funktionalen Sprache werden deterministisch reduziert;
am Ende einer Berechnung enthält der Ausdruck nur noch Konstruktoren.

Funktional-logische Sprachen

Ausdrücke in logischen Sprachen sind Suchziele mit Unbekannten, also freien Variablen.
Das Ergebnis einer Berechnung sind **Bindungen** für diese Unbekannten; die Berechnung ist eine Suche nach diesen Bindungen.

Logische Sprachen arbeiten relational (mehrere Ergebnisse sind möglich), also potentiell nicht-deterministisch, was zu Effizienzverlusten durch **Backtracking** führen kann. Aber:

- Die **Unifikation** ist allgemeiner als der funktionale **Mustervergleich**.
- Eine Berechnung funktioniert in logischen Sprachen i.d.R. auch bei **Ausdrücken mit freien Variablen**.
- Die –in logischen Sprachen übliche– **Funktionsinvertierung** spart viele Funktionsdefinitionen ein (Beispiel: `append`).
- Die in logischen Sprachen eingebauten **Suchstrategien** sind bequem (es muss ja nicht immer –wie in Prolog– nur eine Tiefensuche sein).

Kurz: Man hätte gern das Beste aus beiden Welten.

Curry

Die Sprache Curry erweitert Haskell um **freie Variablen** und **Bedingungen** (*constraint*) an diese, um deren Wertebereich einzuschränken, sowie um **nicht-deterministsche „Funktionen“**.

```
x && (y || (not x))   where x,y free
```

Die freien Variablen werden mit Werten belegt, so dass der Ausdruck auswertbar wird:

```
{x=True, y=True} True
{x=True, y=False} False
{x=False, y=y} False
```

Falls wir nicht an allen Lösungen interessiert sind, können wir mit dem Operator =:= eine **Gleichheitsbedingung** formulieren wie

```
(x && (y || (not x))) =:= True   where x,y free
```

mit der Lösung

```
{x=True, y=True} success
```

Curry

Zwischen mengenwertigen Relationen und einwertigen Funktionen vermittelt z.B. die **nicht-deterministsche „Funktion“** `choose`, die eines ihrer Argumente als Ergebnis auswählt und –wie bei logischen Sprachen üblich– weitere Ergebnisse auf Anforderung liefert.

`choose 1 (choose 2 3)` liefert einen der Werte 1, 2 oder 3.

Zwei *Bedingungen* lassen sich mit `&` konjunktiv verknüpfen:

```
x =:= choose 1 (choose 2 3)  &  x+x =:= x*x   where x free
```

liefert

```
{x=2} success
```

Curry: Narrowing

Um freie Variablen in Ausdrücken bearbeiten zu können, gibt es zwei Auswertungsmechanismen:

- Beim **Narrowing** wird der Auswertungsmechanismus funktionaler Sprachen (**Reduktion**) um den der logischen Sprachen (**Resolution**) erweitert. Dabei wird der **Mustervergleich** durch die **Unifikation** der formalen und aktuellen Parameter eines Funktionsaufrufs ersetzt.

  Eine Operation, die durch Narrowing berechnet wird, heißt **flexibel**. Alle benutzer-definierten Funktionen sind flexibel.

```
x =:= 2+2 where x free                    -- Narrowing
{x=4} success
```

Curry: Residuation

Um freie Variablen in Ausdrücken bearbeiten zu können, gibt es zwei Auswertungsmechanismen:

- Die **Residuation** verzögert Funktionsaufrufe mit uninstanziierten Variablen als Parameter solange, bis alle Variablen von parallel laufenden Prozessen gebunden wurden bzw. zumindest eine eindeutige Regelauswahl möglich ist. Leider berechnet die Residuation für manche Terme keine Lösung, selbst wenn die im Term enthaltenen freien Variablen nicht zur Lösung beitragen.

  Eine Operation, die durch Residuation berechnet wird, heißt **starr**. Die meisten vordefinierten Funktionen sind starr (*rigid*), z.B. die arithmetischen Funktionen.

```
x == 2+2 where x free                          -- Residuation
*** Warning: there are suspended constraints
```

Curry: Muster mit flexiblen Funktionen

Das Narrowing ermöglicht **Muster mit Funktionen**;
im Muster dürfen also außer Datenkonstruktoren und Variablen auch *flexible* Funktionen benutzt werden.

Damit kann das letzte Element einer nicht-leeren Liste mit dem flexiblen Konkatenations-Operator ++ definiert werden als:

```
last (_ ++ [e]) = e
```

Beispiel

Bestimme alle Teillisten einer Liste, so dass das letzte Element `z` der Teilliste gerade das Doppelte des ersten Elements `x` der Teilliste ist:

```
dsl (_++[x]++y++[z]++_) | 2*x == z  =  [x] ++ y ++ [z]
```

Wächter (`2*x == z`) — *Ergebnis* (`[x] ++ y ++ [z]`)

`dsl [3,6,2,1,4,5]` liefert also `[3,6]` und `[2,1,4]`.

Teil IX

Syntaktische Analyse – Ein zweiter Blick: Stadiumautomaten und Parser

Kellerautomaten

Kellerautomaten sind das Automatenmodell für kontextfreie Grammatiken.

Definition

Ein **Kellerautomat** ist ein Tupel $K = (Q, T, \Delta, q_0, F)$, wobei

- Q die endliche Menge der **Zustände**,
- T das **Eingabealphabet** und
- $q_0 \in Q$ der **Anfangszustand** und
- $F \subseteq Q$ die Menge der **Endzustände** sind.
- Die **Übergangsrelation** Δ ist eine endliche Relation zwischen $Q^+ \times (T \cup \{\varepsilon\})$ und Q^*.

Δ kann man auch als endliche partielle Funktion von $Q^+ \times (T \cup \{\varepsilon\})$ in die endlichen Teilmengen von Q^* betrachten.

Kellerautomaten

Achtung: Die Menge der Zustände wurde mit der Menge der Kellersymbole identifiziert.
Der Inhalt des Kellers ist also immer eine Folge von Zuständen.
Der oberste Zustand im Keller ist **aktueller Zustand**.
Der Automat soll mehrere Symbole am oberen Kellerrand gleichzeitig berücksichtigen können.

Die Übergangsrelation beschreibt die möglichen Berechnungen des Kellerautomaten.
Sie beschreibt *endlich* viele Übergänge.
Die Anwendung eines Übergangs $(\gamma, x, \bar{\gamma})$ ersetzt den oberen Abschnitt $\gamma \in Q^+$ des Kellers durch eine neue Folge $\bar{\gamma} \in Q^*$ von Zuständen und liest dabei $x \in T \cup \{\varepsilon\}$ in der Eingabe.
Der ersetzte Kellerabschnitt ist nicht leer.
Ein Übergang, bei dem kein Eingabezeichen verarbeitet wird, heißt ε-**Übergang**.

Kofiguration eines Kellerautomaten

Eine Konfiguration umfasst stets alle Komponenten des Automatentyps, die für zukünftige Schritte des Automaten relevant sind.

Für unseren Kellerautomaten: Kellerinhalt γ und die restliche Eingabe w.

Definition

Eine **Konfiguration** des Kellerautomaten K ist ein Paar $(\gamma, w) \in Q^+ \times T^*$.

Die Relation $\vdash_K$ beschreibt die **Konfigurationsübergänge**:

$(\alpha\beta, aw) \vdash_K (\alpha\beta', w) \Leftrightarrow (\beta, a, \beta') \in \Delta$ für $\alpha, \beta' \in Q^*$, $\beta \in Q^+$, $a \in T \cup \{\varepsilon\}$.

(q_0, w) für beliebiges $w \in T^*$ heißt eine **Anfangskonfiguration**,
(q, ε) für $q \in F$ eine **Endkonfiguration**.

Sprache eines Kellerautomaten

Definition

Ein Wort $w \in T^*$ wird vom Kellerautomaten K **akzeptiert**, wenn $(q_0, w) \overset{*}{\underset{K}{\vdash}} (q, \varepsilon)$ für ein $q \in F$.

Die **Sprache** $L(K)$ ist die Menge der vom Kellerautomaten K akzeptierten Wörter:
$L(K) = \{w \in T^* \mid \exists q \in F : (q_0, w) \overset{*}{\underset{K}{\vdash}} (q, \varepsilon)\}$.

Ein Wort w wird von einem Kellerautomaten akzeptiert, wenn *mindestens* eine Berechnung von der Anfangskonfiguration (q_0, w) zu einer Endkonfiguration führt. Eine solche Berechnung heißt **akzeptierend**.

Für ein Wort kann es mehrere akzeptierende Berechnungen geben, aber auch Berechnungen, die nur einen Präfix des Wortes lesen können oder die das ganze Wort lesen können, aber keine Endkonfiguration erreichen.

Da man akzeptierende Berechnungen nicht durch Probieren finden möchte, bevorzugt man **deterministische Kellerautomaten**.

Nichtdeterminismus bei Kellerautomaten

Bei Kellerautomaten gibt es mehrere Quellen für nichtdeterministisches Verhalten:

- Die Übergangsrelation Δ kann zu $(\gamma, a) \in Q^+ \times T$ mehrere Fortsetzungen anbieten.
- Ist $\bar{\gamma} \in Q^+$ ein nichtleerer Suffix von γ, so passt evtl. auch $(\bar{\gamma}, a)$
- und sogar $(\bar{\gamma}, \varepsilon)$ auf die Konfiguration des Automaten.

deterministische Kellerautomaten

Definition: Ein Kellerautomat K heißt **deterministisch**, wenn die Übergangsrelation Δ folgende Eigenschaft erfüllt:

(D) Sind $(\gamma_1, a, \gamma_2), (\bar{\gamma}_1, a', \bar{\gamma}_2)$ zwei verschiedene Übergänge aus Δ und ist $\bar{\gamma}_1$ ein Suffix von γ_1, dann sind $a', a \in T$ und verschieden, d.h. $\varepsilon \neq a \neq a' \neq \varepsilon$.

Bei deterministischen Kellerautomaten gibt es zu jeder Konfiguration höchstens einen Übergang in eine Nachfolgekonfiguration.

Der Parsingprozess

Schauen wir uns den Parsingprozess systematisch an.

Wir beschreiben ein Verfahren, mit dem zu jeder kontextfreien Grammatik ein Kellerautomat konstruiert werden kann, der die von der Grammatik erzeugte Sprache akzeptiert.

Dieser Automat ist nichtdeterministisch und für einen praktischen Einsatz daher weniger geeignet, aber wir können aus ihm effiziente Parser ableiten.

kontextfreie Ableitungsstadien

(kontextfreies Ableitungs-)Stadium (*item*):

Wie weit wurde eine Produktion während der Ableitung bereits bearbeitet?
Wie können wir die shift-Operation des Kellerautomaten sichtbar machen?

Dazu erhält jede Produktion auf der rechten Seite einen Punkt.

Beispiel

Wir können aus der Produktion $S \to bAa$ folgende Stadien bilden:

$$[S \to \cdot\, bAa], \quad [S \to b \cdot Aa], \quad [S \to bA \cdot a] \quad \text{und} \quad [S \to bAa \cdot].$$

Aus $A \to \varepsilon$ entsteht das Stadium $[A \to \cdot\,]$.

Alles was links vom Punkt steht, ist bereits getan, ist „Vergangenheit“;
alles was rechts vom Punkt steht, ist in „Zukunft“ noch zu tun.

Das Stadium $[A \to \alpha \cdot \beta]$ beschreibt die Situation, dass beim Versuch, aus A ein Wort w abzuleiten, aus α bereits ein Präfix von w abgeleitet wurde.

kontextfreie Ableitungsstadien

Definition:
Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik und $A \to \alpha\beta \in P$.

Ein **Stadium** einer Produktion von G ist ein Tripel (A, α, β)
– meist geschrieben als $[A \to \alpha \cdot \beta]$.

Ist $\alpha = \varepsilon$, also $[A \to \cdot\, \beta]$, so heißt das Stadium **initial**.
Ist $\beta = \varepsilon$, also $[A \to \alpha \cdot\,]$, so heißt das Stadium **final** oder **vollständig**.

Die Menge aller Stadien von G bezeichnen wir mit St_G.

Für eine Folge von Stadien
$\rho = [A_1 \to \beta_1 \cdot B_1\gamma_1]\ [A_2 \to \beta_2 \cdot B_2\gamma_2] \ldots [A_n \to \beta_n \cdot B_n\gamma_n] \in St_G^*$ ist
die **Vergangenheit** $past(\rho) = \beta_1\beta_2 \ldots \beta_n$,
die **Zukunft** $fut(\rho) = \gamma_n \ldots \gamma_2\gamma_1$.

Stadiumautomat zu einer kontextfreien Grammatik

Wir wollen einen Kellerautomaten zu den Stadien einer kontextfreien Grammatik konstruieren:

Die Zustände des Stadiumautomaten und damit seine Kellersymbole sind Stadien der Grammatik.

Der aktuelle Zustand ist das Stadium, an dessen rechter Seite der Automat gerade arbeitet.

Im Keller darunter stehen die Stadien, deren Bearbeitung bereits begonnen hat, aber noch nicht beendet wurde.

erweiterte kontextfreie Grammatik

Zunächst erweitern wir die Grammatik G so, dass die **Terminierung** des Stadiumautomaten **am aktuellen Zustand abgelesen** werden kann.

Kandidaten für die Endzustände sind alle finalen Stadien $[S \to \alpha \cdot]$ mit dem Startsymbol S auf der linken Seite.

Tritt das Startsymbol S auch auf der rechten Seite einer Produktion auf, können diese Stadien oben auf dem Keller auftreten, ohne dass der Automat dann terminieren sollte, weil darunter noch zu bearbeitende Stadien liegen.

Darum **erweitern** wir die Grammatik G um ein **neues Startsymbol** Z und fügen die Produktion $Z \to S$ zur Menge der Produktionen von G hinzu. Offensichtlich ändert sich die erzeugte Sprache dadurch nicht.

Das neue Startsymbol Z kommt dann auf keiner rechten Seite vor und wir können $[Z \to S \cdot]$ als Endzustand wählen.

erweiterte kontextfreie Grammatik

Definition: Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik.

Dann ist
$G^+ = (N^+, T, P^+, Z)$ mit $N^+ = N \dot\cup \{Z\}$ und $P^+ = P \cup \{Z \to S\}$
die um Z **erweiterte kontextfreie Grammatik** zu G.

Der Stadiumautomat

Als Anfangszustand des Stadiumautomaten wählen wir das initiale Stadium $[Z \to \cdot\, S]$ und als einzigen Endzustand das finale Stadium $[Z \to S \cdot\,]$.

Definition:
Sei G^+ die um Z erweiterte kontextfreie Grammatik zu $G = (N, T, P, S)$.

Der Kellerautomat $K_G = (St_{G^+}, T, \Delta, [Z \to \cdot\, S], \{[Z \to S \cdot\,]\})$ heißt **Stadiumautomat** zu G.

Die Übergangsrelation Δ besteht aus allen Übergängen der folgenden drei Typen:

- $\Delta([X \to \beta \cdot Y\gamma], \varepsilon) = \{[X \to \beta \cdot Y\gamma]\ [Y \to \cdot\, \alpha] \mid Y \to \alpha \in P\}$
  Expansionsübergänge
- $\Delta([X \to \beta \cdot a\,\gamma], a) = \{[X \to \beta a \cdot \gamma]\}$ **Leseübergänge**
- $\Delta([X \to \beta \cdot Y\gamma]\ [Y \to \alpha \cdot\,], \varepsilon) = \{[X \to \beta Y \cdot \gamma]\}$.
  Reduktionsübergänge