Algorithmisches Denken und imperative Programmierung

Janis Voigtländer

Universität Bonn

Wintersemester 2011/12

Zur Erinnerung

Wir suchen eine Möglichkeit der Syntaxbeschreibung,
um etwa systematisch sinnvolle C-Programmteile wie:

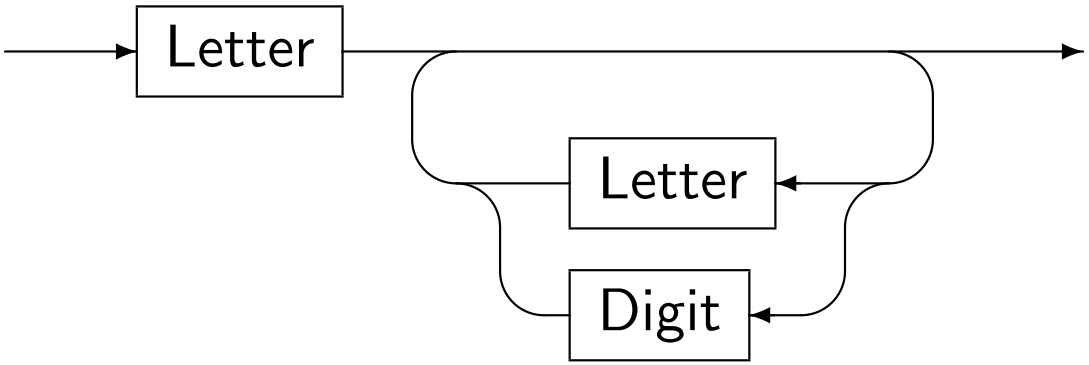
```
int fib(int n){
  int x=1, y=1, i, z;
  if (n>=2)
    for (i=1; i<n; i=i+1) {z=x; x=y; y=z+y;}
  return y;}
```

zu unterscheiden von „irgendwelchen“ Zeichenketten wie:

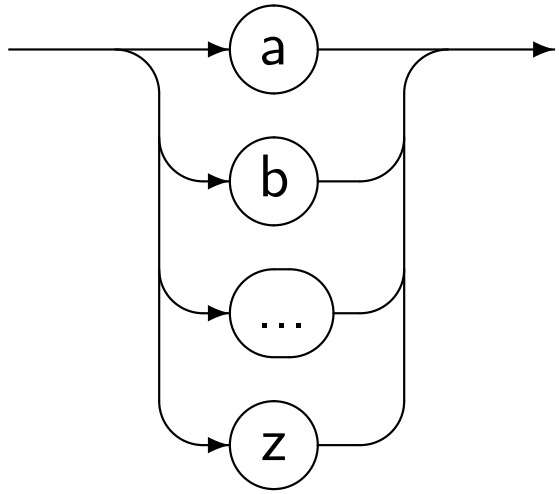
```
int fib(int n){
  int x=1, y=1, i, z;
  if (n>=2; i=i+1)
    for (i=1; i<n) {z=x; x=y; y=z+y;}
  return y;}
```

Syntaxdiagramme — einfaches Beispiel

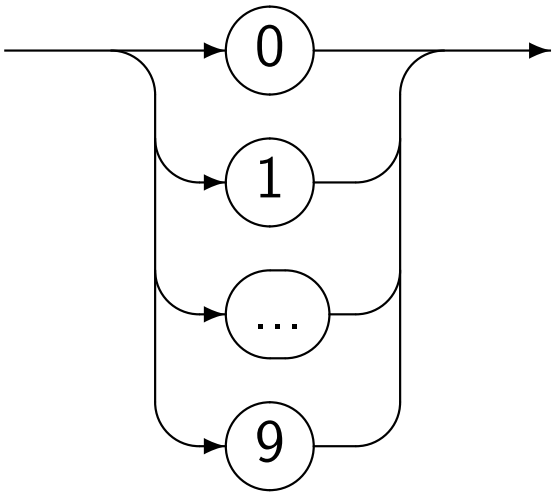
Ident



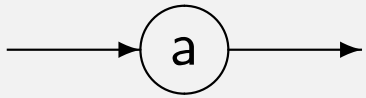
Letter



Digit



Zutaten:

Ident	...	Name
	...	Rechteck
	...	Oval
	...	Verbindung

Klärung einiger Begriffe / Definitionen

Alphabet: eine nichtleere, endliche Menge Σ;
Elemente heißen Symbole

Wort: eine endliche Folge von Symbolen aus Σ

- leeres Wort: Folge der Länge 0; Notation: ε

Σ^*: Menge aller Worte (über Alphabet Σ)

- $\varepsilon \in \Sigma^*$
- wenn $w \in \Sigma^*$ und $\sigma \in \Sigma$, dann $w\sigma \in \Sigma^*$
- keine weiteren Worte in Σ^*

Konkatenation: Operation, die zwei Worte aneinanderhängt

Sprache: Teilmenge von Σ^*

Metasprache: Sprache zur Beschreibung einer anderen Sprache;
in der Regel über verschiedenen Alphabeten

Objektsprache: durch eine andere Sprache beschriebene Sprache

Klärung einiger Begriffe / Beispiele

Alphabet: $\Sigma = \{a, b, c\}$

Wort: $abcca$

Σ^*: $\{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, \dots\}$

Konkatenation: $ab \cdot cca = abcca$

Sprache: $\{a^n bc^n \mid n \geq 0\}$

Metasprache: ???

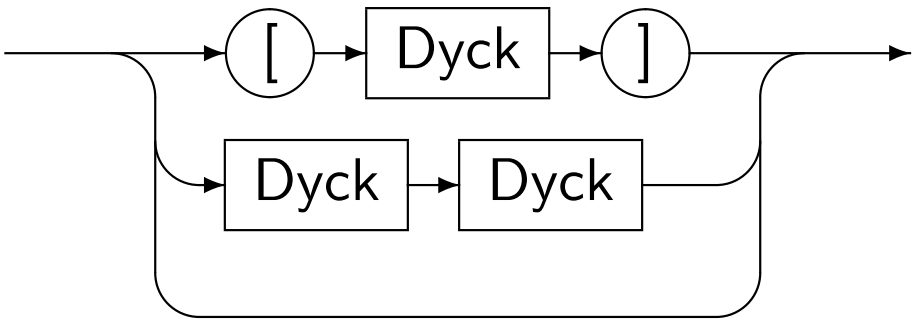
Mengenschreibweise als Metasprache

- Es liegt nahe, aus der Mathematik bekannte Mengennotationen zu verwenden.
- Funktioniert problemlos für endliche Sprachen:
  - $\emptyset$, $\{\}$
  - $\{\varepsilon\}$
  - $\{b, abc, aabcc, aaabccc\}$
- Geht auch noch gut für bestimmte unendliche Sprachen:
  - $\{a^n bc^n \mid n \geq 0\}$, wobei $a^n = \underbrace{a \cdots a}_{n\times}$
  - $\{(ab)^n c^m b^n c^k \mid n, m \geq 0, k \geq 1\}$, wobei $(ab)^n$ durch n-fache Konkatenation definiert
- Schon problematisch etwa für die Sprache aller „wohlgeklammerten Ausdrücke“ über $\Sigma = \{\,[\,,\,]\,\}$:
  - $\{\varepsilon, [\,], [\,][\,], [[\,]], [\,][\,][\,], [[\,]][\,], [\,][[\,]], [[[\,]]], \ldots\}$

... daher Syntaxdiagramme

Wohlgeklammerte Ausdrücke einfach beschreibbar:

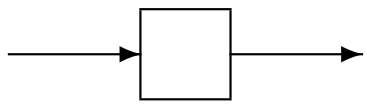
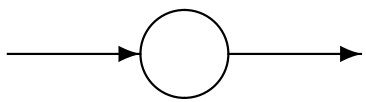
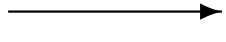
Dyck



Intuition: In jedem nichtleeren wohlgeklammerten Ausdruck gibt es zur (notwendigerweise) am Wortanfang stehenden öffnenden Klammer (genau) eine passende schließende Klammer:

- entweder ganz am Ende, dann steht dazwischen auch ein wohlgeklammerter Ausdruck;
- oder irgendwo in der Mitte, dann wird bis dahin selbst ein wohlgeklammerter Ausdruck gebildet und der Rest muss auch wohlgeklammert sein.

Syntaxdiagramme — Details zum Aufbau

- Wir betrachten immer ein System (eine Menge) von Syntaxdiagrammen.
- Darin hat jedes Syntaxdiagramm einen eindeutigen Namen; dies sind die sogenannten „syntaktischen Variablen“.
- Genau eine syntaktische Variable ist als Start festgelegt.
- Rechtecke  enthalten syntaktische Variablen.
- Ovale  enthalten Symbole aus Σ.
- Verbindungen  dürfen sich nicht überschneiden, aber verzweigen/zusammenführen.
- Jedes Syntaxdiagramm hat genau eine eingehende und eine ausgehende Verbindung.

Syntaxdiagramme — Semantik

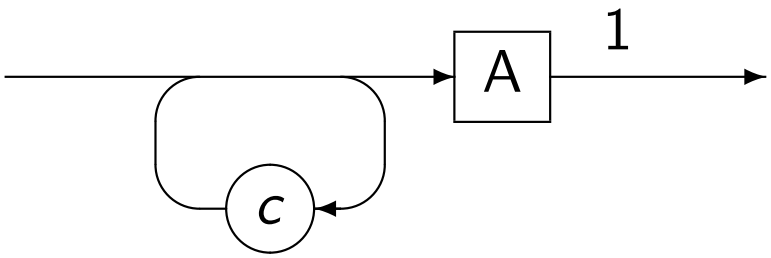
Rücksprungalgorithmus

1. Versehe den Ausgang jedes Rechtecks mit einer eindeutigen Marke, genannt *Rücksprungadresse*.
2. Starte mit einem leer initialisierten *Kellerspeicher*.
3. Beginne am Eingang des festgelegten Startdiagramms.
4. Folge den Verbindungen auf einem legalen Weg.
   - falls Ausgang eines Syntaxdiagramms erreicht, weiter mit 5.
   - falls Oval erreicht, notiere enthaltenes Symbol, weiter mit 4.
   - falls Rechteck erreicht (enthält syntaktische Variable):
     4.1 lege Kopie der Rücksprungadresse oben auf Kellerspeicher
     4.2 weiter mit 4., am Eingang des bezeichneten Syntaxdiagramms
5. - falls Kellerspeicher nicht leer:
     5.1 nimm oberste Rücksprungadresse *adr* vom Kellerspeicher
     5.2 weiter mit 4., an entsprechender Stelle im System
   - falls Kellerspeicher leer (und notwendigerweise Ausgang des Startdiagramms erreicht), Erzeugung beendet

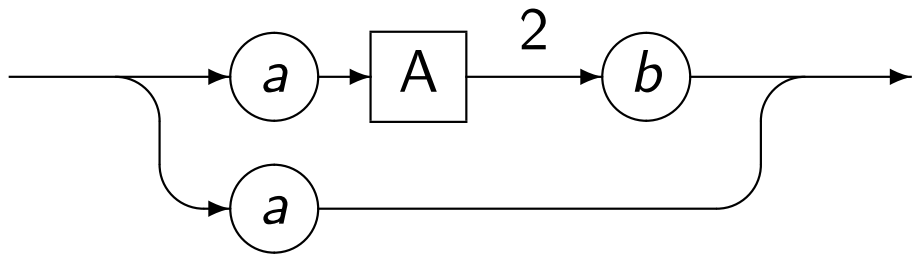
Protokollieren des Algorithmus

Startdiagramm: **S**

S



A



Wort	Kellerspeicher
cc	1
cca	21
ccaa	221
ccaaa	21
ccaaab	1
ccaaabb	–
ccaaabb	–

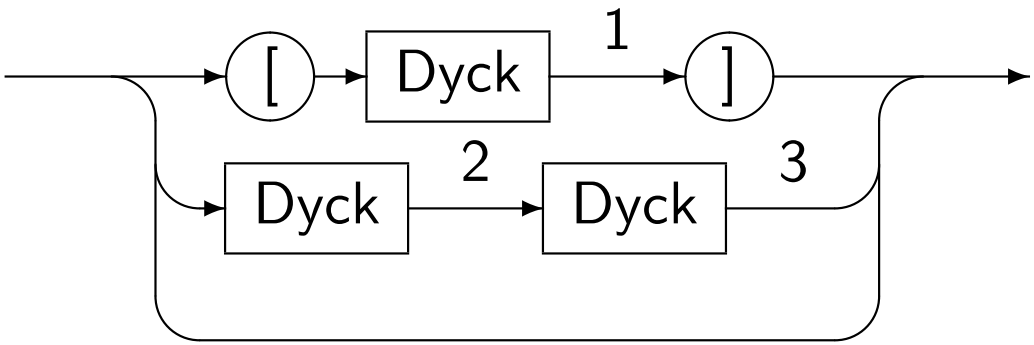
Protokollerstellung:

- aktuelles Wort und Kellerinhalt notieren vor Einsprung in neues Syntaxdiagramm
- aktuelles Wort und Kellerinhalt notieren nach Verlassen eines Syntaxdiagramms

Protokollieren des Algorithmus — anderes Beispiel

Startdiagramm: **Dyck**

Dyck



Wort	Kellerspeicher
ε	2
[	12
[	212
[	12
[	312
[	12
[	2
[]	–
[]	3
[]	–
[]	–

Beobachtungen:

- Das ist sicher nicht die kürzeste Ableitung für [].
- Allein Wort + Kellerspeicher bestimmen den aktuellen Zustand des Algorithmus nicht eindeutig.

Angesichts der gemachten Beobachtungen . . .

Handelt es sich bei:

Rücksprungalgorithmus

1. Versehe den Ausgang jedes Rechtecks mit einer eindeutigen Marke, genannt *Rücksprungadresse*.
2. Starte mit einem leer initialisierten *Kellerspeicher*.
3. Beginne am Eingang des festgelegten Startdiagramms.
4. Folge den Verbindungen auf einem legalen Weg.
   - falls Ausgang eines Syntaxdiagramms erreicht, weiter mit 5.
   - falls Oval erreicht, notiere enthaltenes Symbol, weiter mit 4.
   - falls Rechteck erreicht (enthält syntaktische Variable):
     - 4.1 lege Kopie der Rücksprungadresse oben auf Kellerspeicher
     - 4.2 weiter mit 4., am Eingang des bezeichneten Syntaxdiagramms
5. - falls Kellerspeicher nicht leer:
     - 5.1 nimm oberste Rücksprungadresse *adr* vom Kellerspeicher
     - 5.2 weiter mit 4., an entsprechender Stelle im System
   - falls Kellerspeicher leer (und notwendigerweise Ausgang des Startdiagramms erreicht), Erzeugung beendet

denn überhaupt um einen „Algorithmus“?

Kriterien:

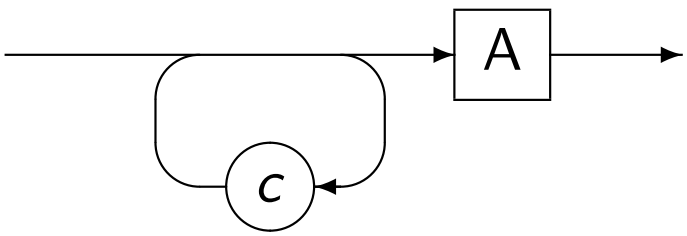
Finitheit	✓
Effektivität	✓
Determiniertheit	↯
Terminierung	↯

$\Rightarrow$ nicht-deterministischer Algorithmus, der nicht immer terminiert

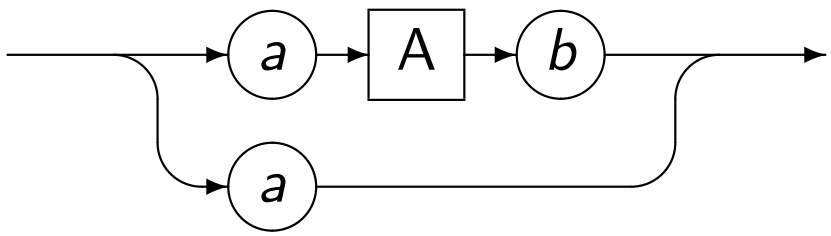
Erzeugung vs. Erkennung

Startdiagramm: **S**

S



A

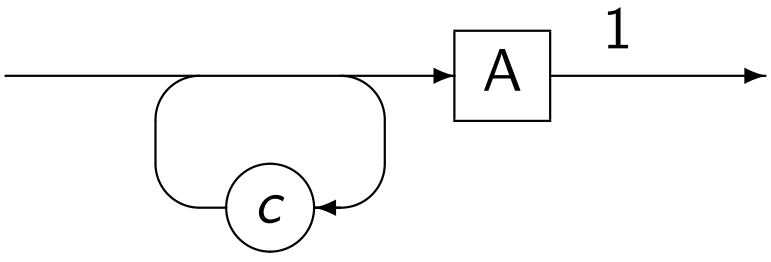


- einige erzeugte Worte: *a*, *ca*, *cca*, *aab*, *ccca*, *caab*, *cccca*, *ccaab*, *aaabb*, *ccccca*, *cccaab*, *caaabb*, *cccccca*, *ccccaab*, ...
- gesamte erzeugte Sprache in Mengenschreibweise: $\{c^n a^{m+1} b^m \mid n, m \geq 0\}$
- anderes, aber verwandtes Problem: entscheiden/erkennen, ob ein gegebenes Wort erzeugt werden kann

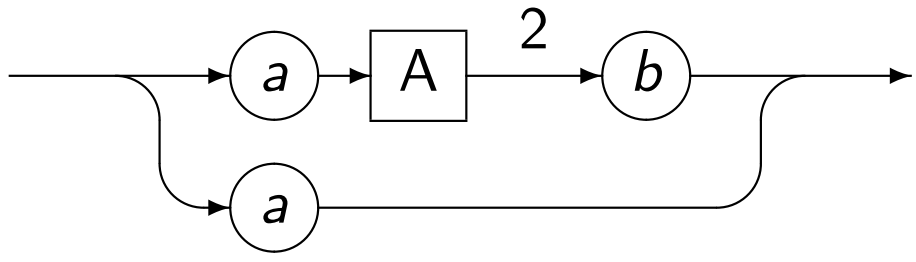
Erzeugung vs. Erkennung

Startdiagramm: **S**

S



A



Für *caab* zwei erfolglose „Durchläufe“, aber auch ein erfolgreicher:

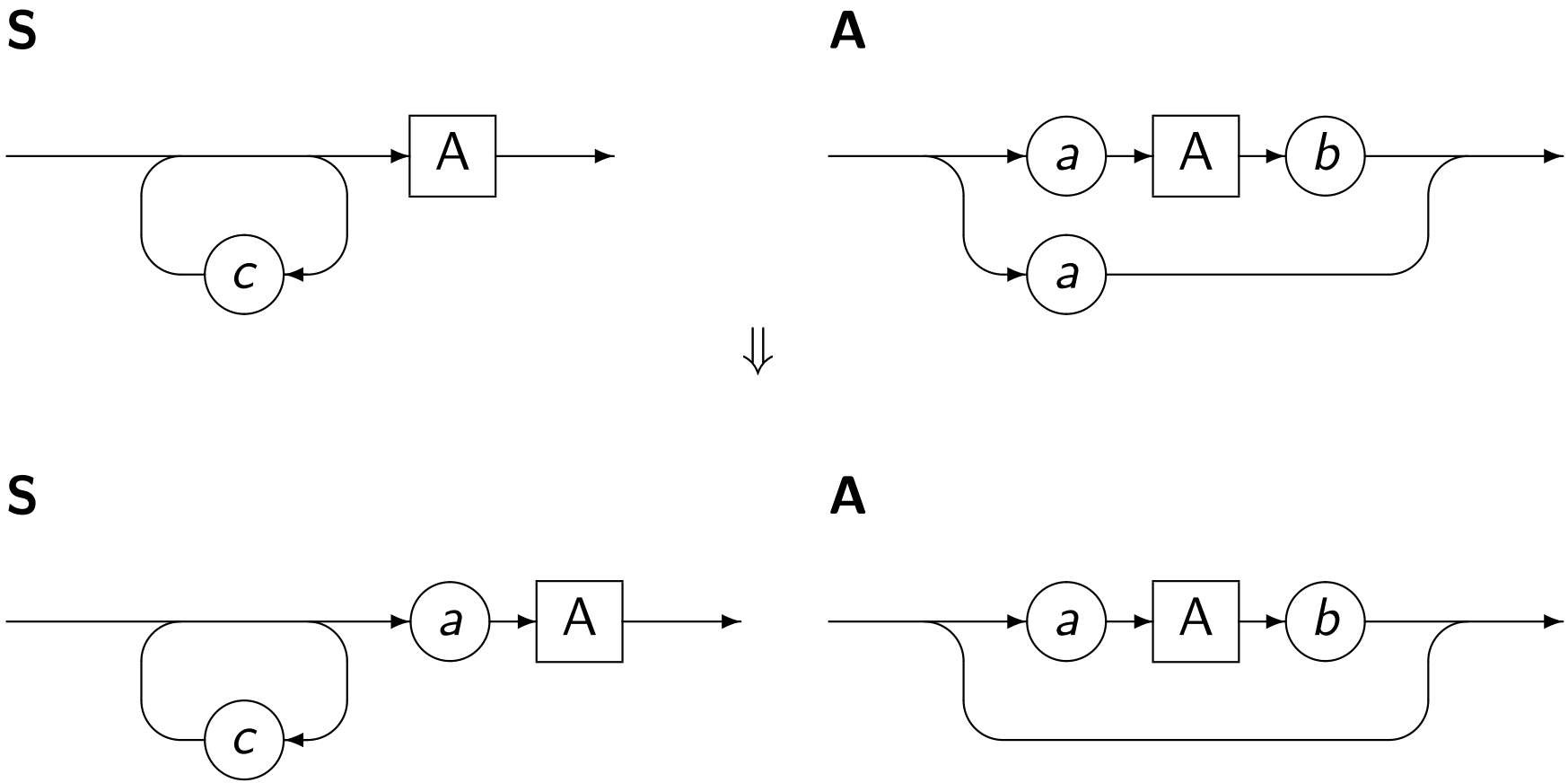
Wort	Kellerspeicher
c	1
ca	–

Wort	Kellerspeicher
c	1
ca	21
caa	221

Wort	Kellerspeicher
c	1
ca	21
caa	1
caab	–
caab	–

Die ersten beiden Fälle entsprechen sogenannter „abnormaler Terminierung“.

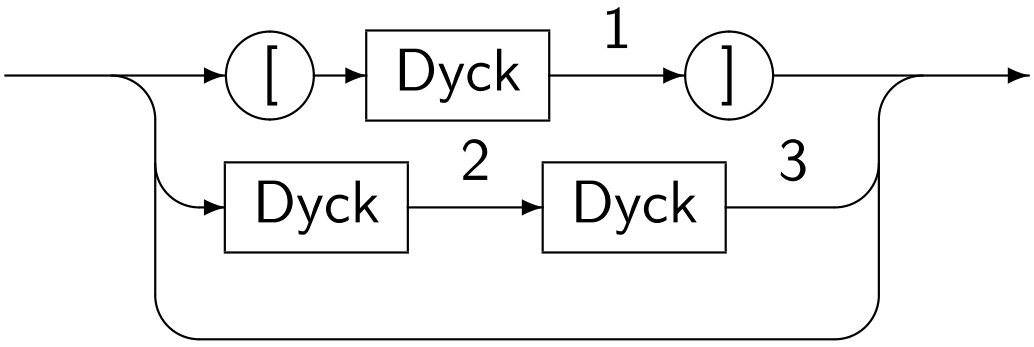
Optimierungen zum Zwecke einfacherer Erkennung



- Erzeugen jeweils die gleiche Sprache $\{c^n a^{m+1} b^m \mid n, m \geq 0\}$.
- Das zweite System erlaubt deterministische Erkennung mit jeweils nur einem Symbol „look-ahead“.

„Analog“

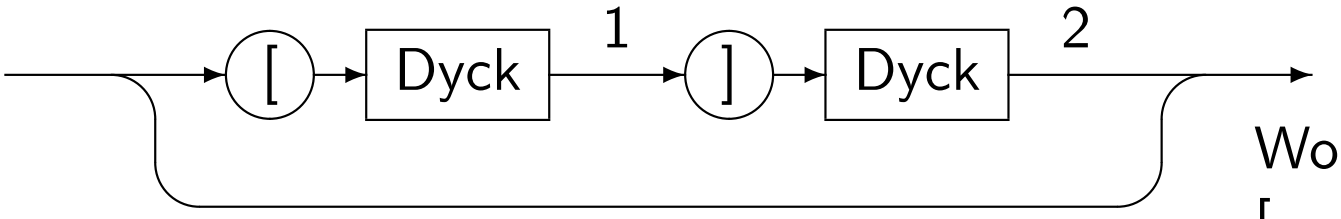
Dyck



Wort	Kellerspeicher
ε	2
[	12
[	212
[	12
[	312
[	12
[	2
[]	–
[]	3
[]	–
[]	–

$\Downarrow$

Dyck

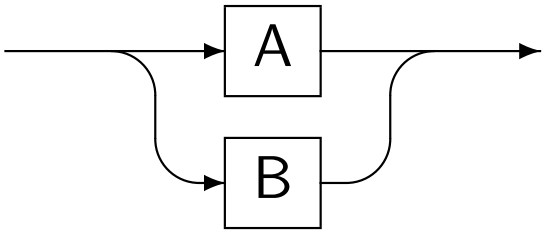


Wort	Kellerspeicher
[	1
[	–
[]	2
[]	–
[]	–

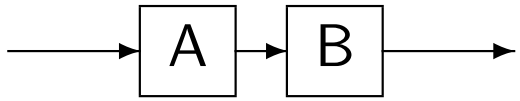
Es lässt sich beweisen, dass beide Diagramme zur gleichen Sprache führen.

Typische Muster — auch für Übungsaufgaben

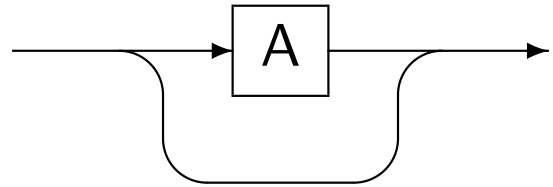
Alternative:



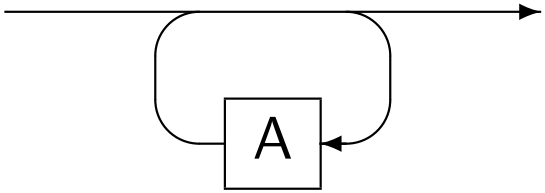
Konkatenation:



Option:

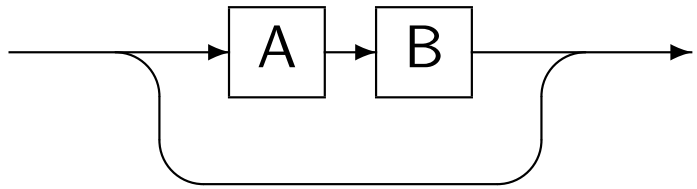


Iteration direkt:



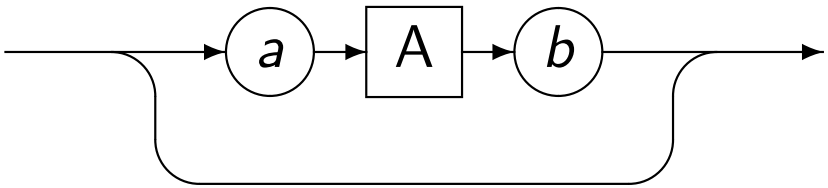
Iteration per Rekursion:

B



„Pumpen“ von Symbolen (oder ganzen Blöcken):

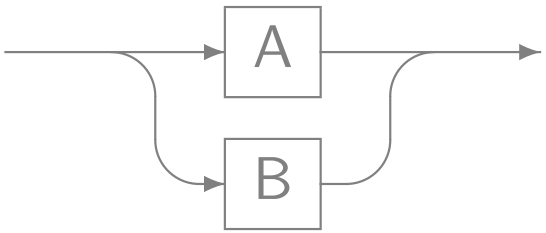
A



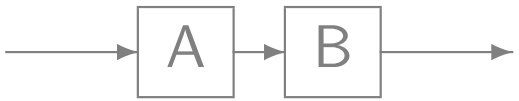
. . . sowie Varianten

Typische Muster — auch für Übungsaufgaben

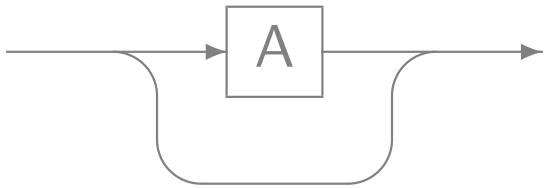
Alternative:



Konkatenation:



Option:



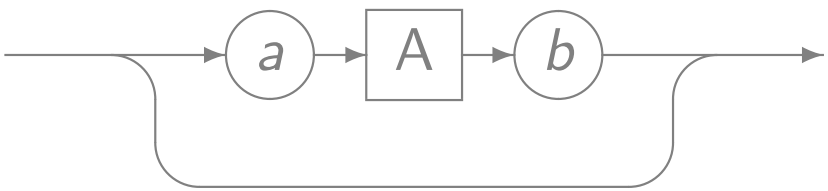
Ite

Einige Dinge, die nicht gehen:

- mehr als doppelte Abhängigkeiten: $\{a^n b^n c^n \mid n \geq 0\}$
- „verschränkte Abhängigkeiten“: $\{a^n b^m c^n d^m \mid n, m \geq 0\}$
- komplizierte Arithmetik: $\{a^n b^m c^{n*m} \mid n, m \geq 0\}$

„Pumpen“ von Symbolen (oder ganzen Blöcken):

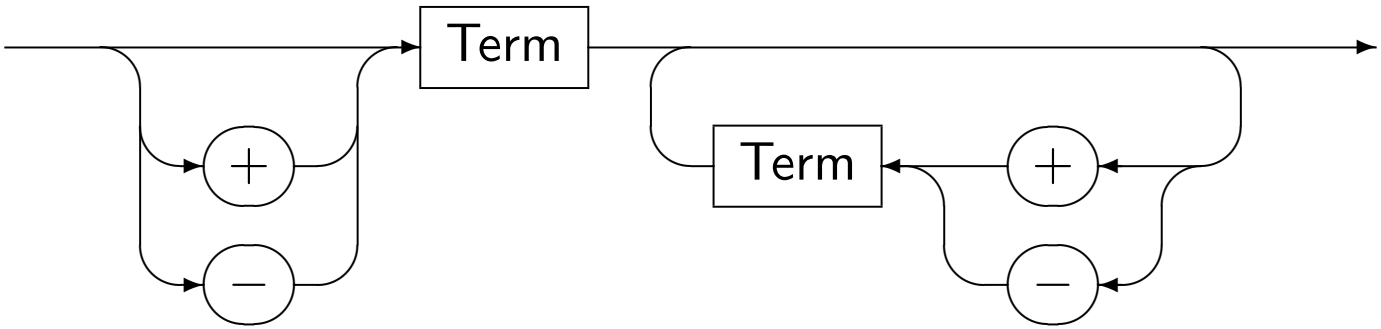
A



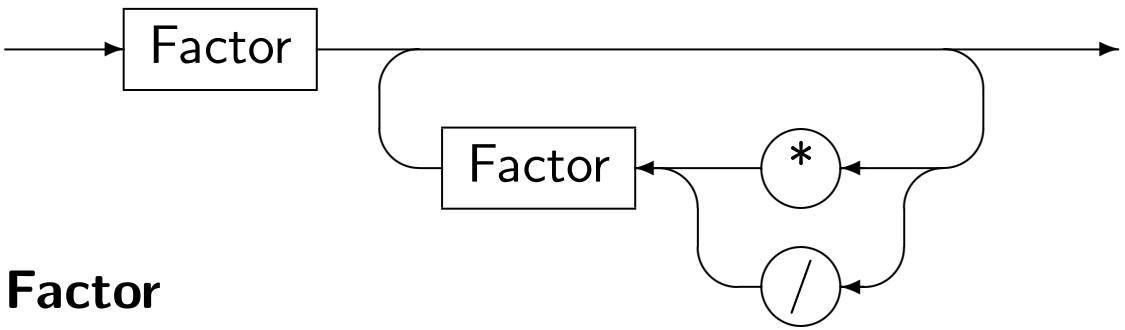
. . . sowie Varianten

Hin zu C

Expression



Term



Factor