Helmut Seidl

Compilerbau
+
Abstrakte Maschinen

München

Sommersemester 2004

Organisatorisches

Der erste Abschnitt Die Übersetzung von C ist den Vorlesungen Compilerbau und Abstrakte Maschinen gemeinsam :-)

Er findet darum zu beiden Vorlesungsterminen statt :-)

Zeiten:

Vorlesung Compilerbau:	Mo. 12:15-13:45 Uhr
	Mi. 10:15-11:45 Uhr
Vorlesung Abstrakte Maschinen:	Mi. 13:15-14:45 Uhr
Übung Compilerbau:	Di./Do. 12:15-13:45 Uhr
	Di./Fr. 10:15-11:45 Uhr
Übung Abstrakte Maschinen:	Do. 14:15-15:45 Uhr

Einordnung:

Diplom-Studierende:

Compilerbau:	Wahlpflichtveranstaltung
Abstrakte Maschinen:	Vertiefende Vorlesung

Bachelor-Studierende:

Compilerbau:	8 ETCS-Punkte
Abstrakte Maschinen:	nicht anrechenbar

Scheinerwerb:

Diplom-Studierende:

- 50% der Punkte;
- zweimal Vorrechnen :-)

Bachelor-Studierende:

- Klausur
- Erfolgreiches Lösen der Aufgaben wird zu 20% angerechnet :-))

Material:

- Literaturliste (im Netz)
- Aufzeichnung der Vorlesungen
  (Folien + Annotationen + Ton + Bild)
- die Folien selbst :-)
- Tools zur Visualisierung der Abstrakten Maschinen :-))
- Tools, um Komponenten eines Compilers zu generieren ...

Weitere Veranstaltungen:

- Seminar Programmanalyse — Di., 14:00-16:00 Uhr
- Wahlpflicht-Praktika:

SS 2004:	Oberflächengenerierung (Frau Höllerer)
WS 2004/05:	Konstruktion eines Compilers (Frau Höllerer)

0 Einführung

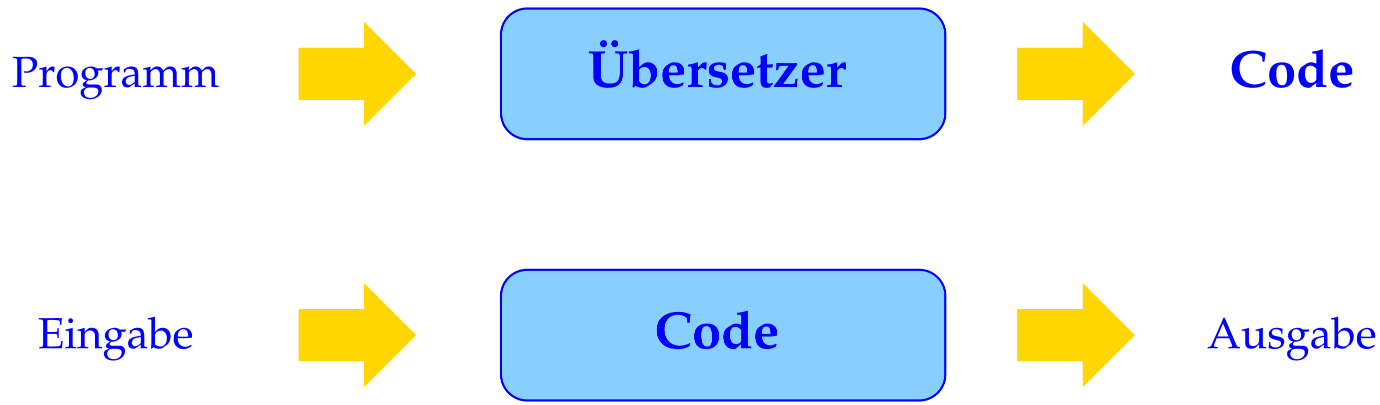
Prinzip eines Interpreters:

Programm + Eingabe $\Rightarrow$ **Interpreter** $\Rightarrow$ Ausgabe

Vorteil: Keine Vorberechnung auf dem Programmtext erforderlich $\Longrightarrow$ keine/geringe Startup-Zeit :-)

Nachteil: Während der Ausführung werden die Programm-Bestandteile immer wieder analysiert $\Longrightarrow$ längere Laufzeit :-(

Prinzip eines Übersetzers:



Zwei Phasen:

- Übersetzung des Programm-Texts in ein Maschinen-Programm;
- Ausführung des Maschinen-Programms auf der Eingabe.

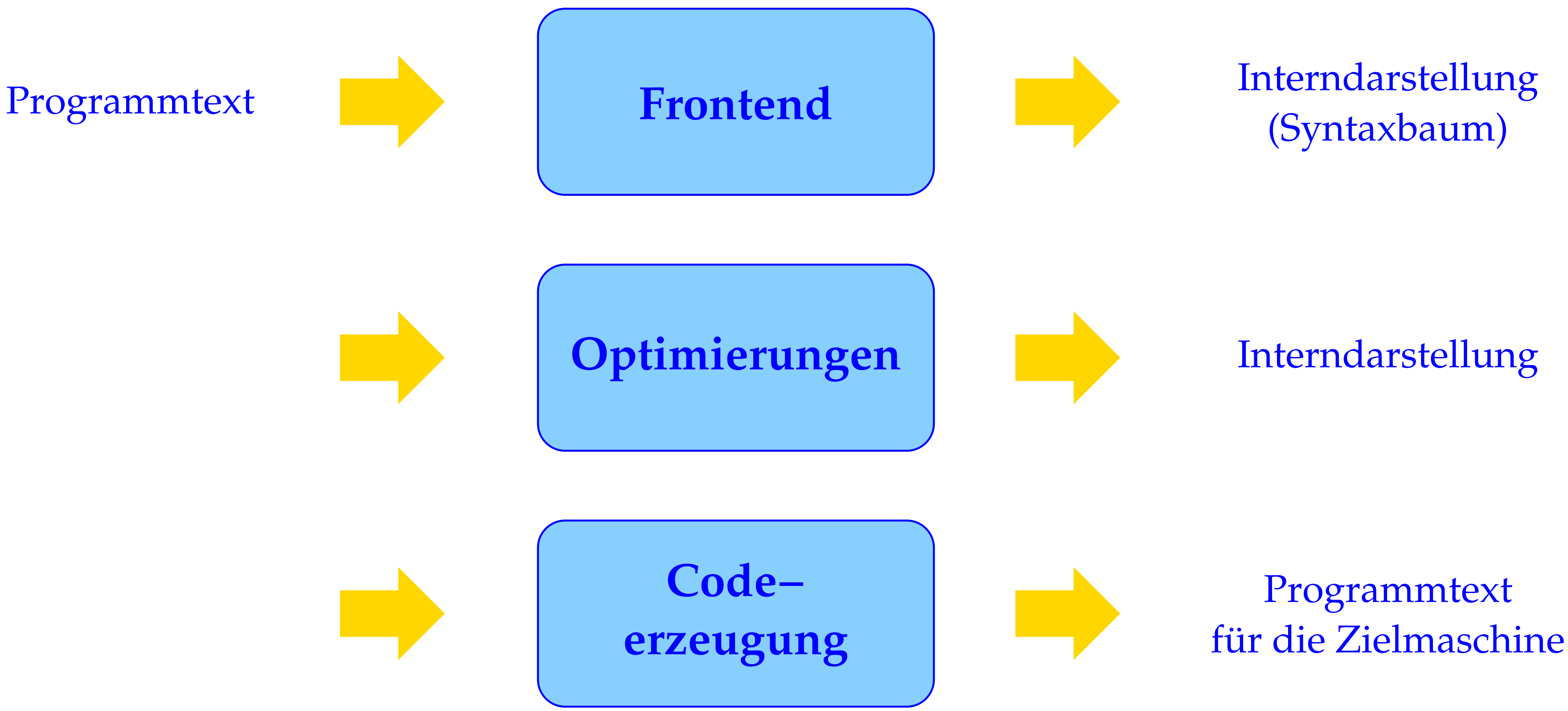
Eine Vorberechnung auf dem Programm gestattet u.a.

- eine geschickte(re) Verwaltung der Variablen;
- Erkennung und Umsetzung globaler Optimierungsmöglichkeiten.

Nachteil: Die Übersetzung selbst dauert einige Zeit :-(

Vorteil: Die Ausführung des Programme wird effizienter $\Longrightarrow$ lohnt sich bei aufwendigen Programmen und solchen, die mehrmals laufen ...

Aufbau eines Übersetzers:



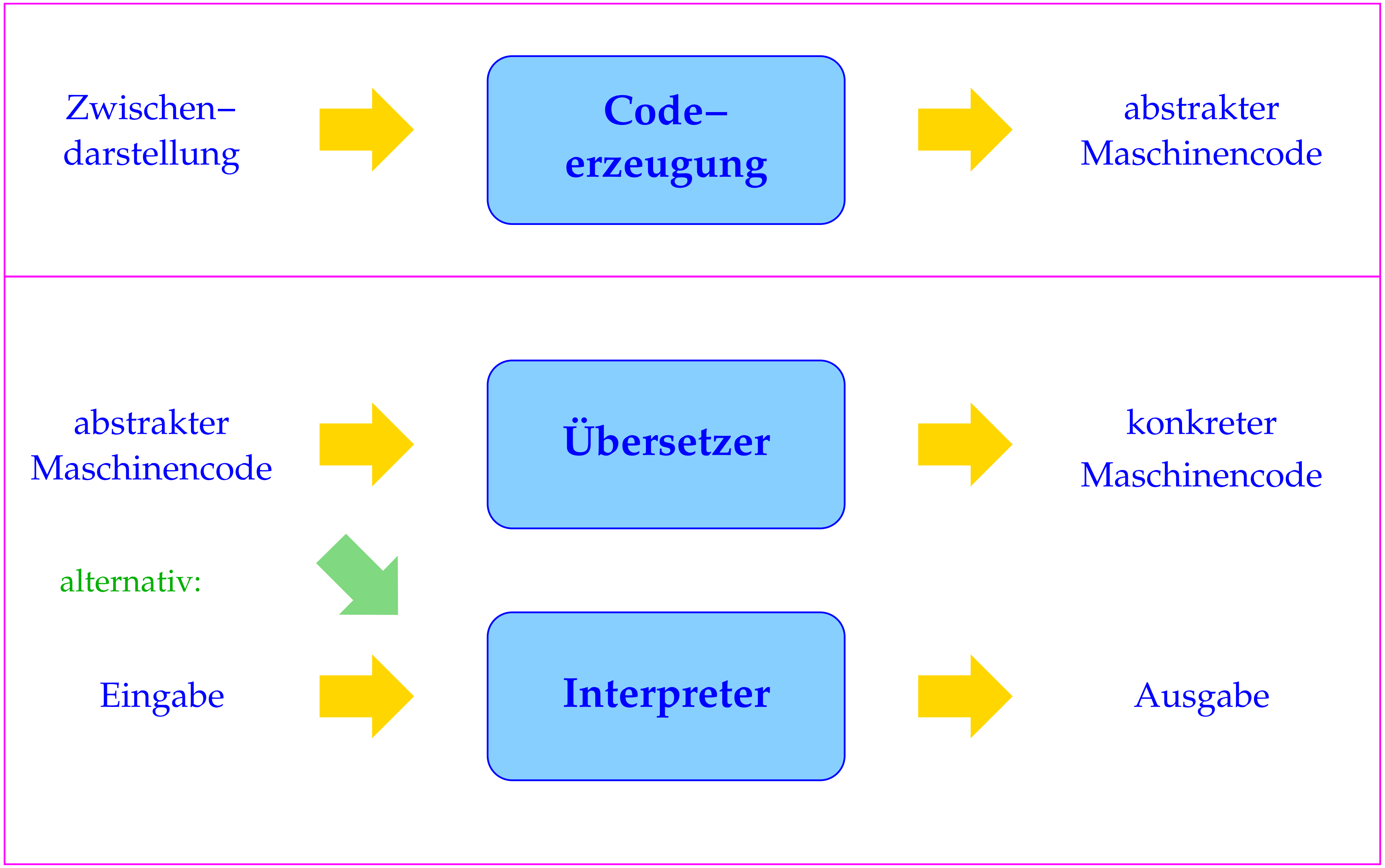
Aufgaben der Code-Erzeugung:

Ziel ist eine geschickte Ausnutzung der Möglichkeiten der Hardware. Das heißt u.a.:

1. Instruction Selection: Auswahl geeigneter Instruktionen;
2. Registerverteilung: optimale Nutzung der vorhandenen (evt. spezialisierten) Register;
3. Instruction Scheduling: Anordnung von Instruktionen (etwa zum Füllen einer Pipeline).

Weitere gegebenenfalls auszunutzende spezielle Hardware-Features können mehrfache Recheneinheiten sein, verschiedene Caches, . . .

Weil konkrete Hardware so vielgestaltig ist, wird die Code-Erzeugung oft erneut in zwei Phasen geteilt:


Zwischen–
darstellung
Code–
erzeugung
abstrakter
Maschinencode
abstrakter
Maschinencode
Übersetzer
konkreter
Maschinencode
alternativ:
Eingabe
Interpreter
Ausgabe

Eine abstrakte Maschine ist eine idealisierte Hardware, für die sich einerseits "leicht" Code erzeugen lässt, die sich andererseits aber auch "leicht" auf realer Hardware implementieren lässt.

Vorteile:

- Die Portierung auf neue Zielarchitekturen vereinfacht sich;
- der Compiler wird flexibler;
- die Realisierung der Programmkonstrukte wird von der Aufgabe entkoppelt, Hardware-Features auszunutzen.

Programmiersprachen, deren Übersetzungen auf abstrakten Maschinen beruhen:

Pascal	$\longrightarrow$	P-Maschine	
Smalltalk	$\longrightarrow$	Bytecode	
Prolog	$\longrightarrow$	WAM	(“Warren Abstract Machine”)
SML, Haskell	$\longrightarrow$	STGM	
Java	$\longrightarrow$	JVM	

Hier werden folgende Sprachen und abstrakte Maschinen betrachtet:

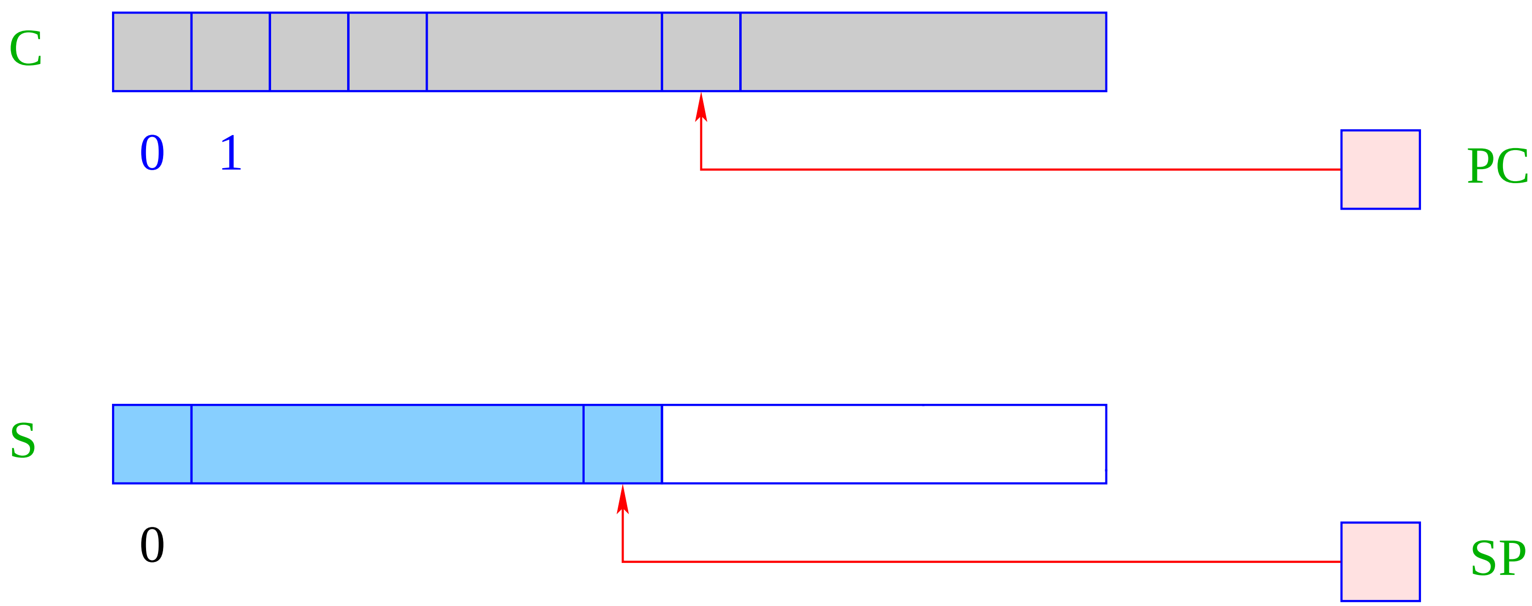
C	$\longrightarrow$	CMa	//	*imperativ*
PuF	$\longrightarrow$	MaMa	//	*funktional*
PuP	$\longrightarrow$	WiM	//	*logikbasiert*

Die Übersetzung von C

1 Die Architektur der CMa

- Jede abstrakte Maschine stellt einen Satz abstrakter Instruktionen zur Verfügung.
- Instruktionen werden auf der abstrakten Hardware ausgeführt.
- Die abstrakte Hardware fassen wir als eine Menge von Datenstrukturen auf, auf die die Instruktionen zugreifen
- ... und die vom Laufzeitsystem verwaltet werden.

Für die CMa benötigen wir:


C
0 1
PC
S
0
SP

- S ist der (Daten-)Speicher, auf dem nach dem LIFO-Prinzip neue Zellen allokiert werden können $\Longrightarrow$ Keller/Stack.
- SP ($\widehat{=}$ Stack Pointer) ist ein Register, das die Adresse der obersten belegten Zelle enthält.

  Vereinfachung: Alle Daten passen jeweils in eine Zelle von S.
- C ist der Code-Speicher, der das Programm enthält.

  Jede Zelle des Felds C kann exakt einen abstrakten Befehl aufnehmen.
- PC ($\widehat{=}$ Program Counter) ist ein Register, das die Adresse des nächsten auszuführenden Befehls enthält.
- Vor Programmausführung enthält der PC die Adresse 0

  $\Longrightarrow$ C[0] enthält den ersten auszuführenden Befehl.

Die Ausführung von Programmen:

- Die Maschine lädt die Instruktion aus C[PC] in ein Instruktions-Register IR und führt sie aus.
- Vor der Ausführung eines Befehls wird der PC um 1 erhöht.

```
while (true) {
    IR = C[PC]; PC++;
    execute (IR);
}
```

- Der PC muss vor der Ausführung der Instruktion erhöht werden, da diese möglicherweise den PC überschreibt :-)
- Die Schleife (der Maschinen-Zyklus) wird durch Ausführung der Instruktion halt verlassen, die die Kontrolle an das Betriebssystem zurückgibt.
- Die weiteren Instruktionen führen wir nach Bedarf ein :-)

2 Einfache Ausdrücke und Wertzuweisungen

Aufgabe: werte den Ausdruck $(1+7)*3$ aus!

Das heißt: erzeuge eine Instruktionsfolge, die

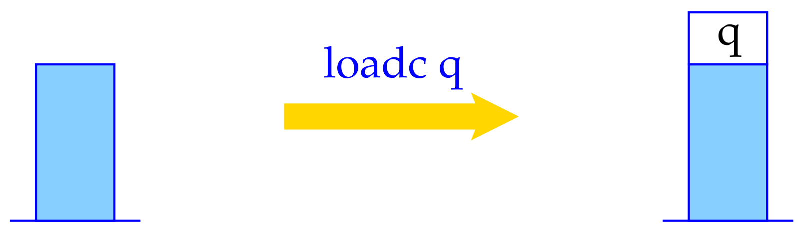
- den Wert des Ausdrucks ermittelt und dann
- oben auf dem Keller ablegt...

Idee:

- berechne erst die Werte für die Teilausdrücke;
- merke diese Zwischenergebnisse oben auf dem Keller;
- wende dann den Operator an!

Generelles Prinzip:

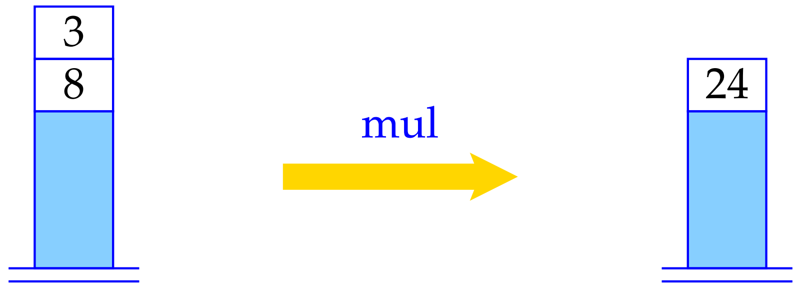
- die Argumente für Instruktionen werden oben auf dem Keller erwartet;
- die Ausführung einer Instruktion konsumiert ihre Argumente;
- möglicherweise berechnete Ergebnisse werden oben auf dem Keller wieder abgelegt.



SP++;

S[SP] = q;

Die Instruktion loadc q benötigt keine Argumente, legt dafür aber als Wert die Konstante q oben auf dem Stack ab.



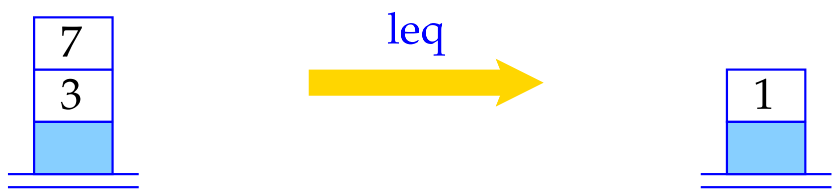
SP--;

S[SP] = S[SP] $*$ S[SP+1];

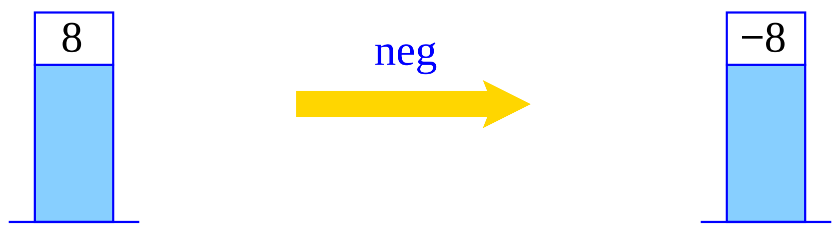
mul erwartet zwei Argumente oben auf dem Stack, konsumiert sie und legt sein Ergebnis oben auf dem Stack ab.

... analog arbeiten auch die übrigen binären arithmetischen und logischen Instruktionen add, sub, div, mod, and, or und xor, wie auch die Vergleiche eq, neq, le, leq, gr und geq.

Beispiel: Der Operator leq



Einstellige Operatoren wie neg und not konsumieren dagegen ein Argument und erzeugen einen Wert:

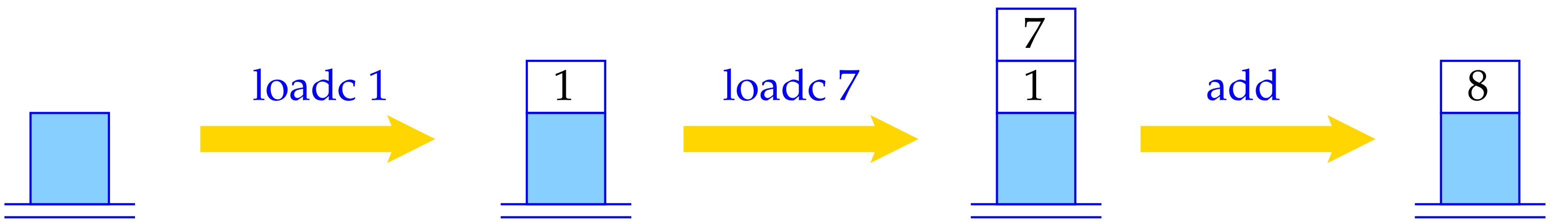


S[SP] = – S[SP];

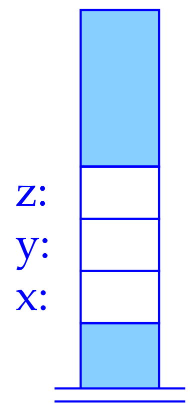
Beispiel: Code für $1 + 7$:

loadc 1 loadc 7 add

Ausführung dieses Codes:



Variablen ordnen wir Speicherzellen in S zu:



Die Übersetzungsfunktionen benötigen als weiteres Argument eine Funktion ρ, die für jede Variable x die (Relativ-)Adresse von x liefert. Die Funktion ρ heißt Adress-Umgebung (Address Environment).

Variablen können auf zwei Weisen verwendet werden.

Beispiel: $x = y + 1$

Für y sind wir am Inhalt der Zelle, für x an der Adresse interessiert.

L-Wert von x = Adresse von x

R-Wert von x = Inhalt von x

$\text{code}_\text{R}\ e\ \rho$	liefert den Code zur Berechnung des R-Werts von e in der Adress-Umgebung ρ
$\text{code}_\text{L}\ e\ \rho$	analog für den L-Wert

Achtung:

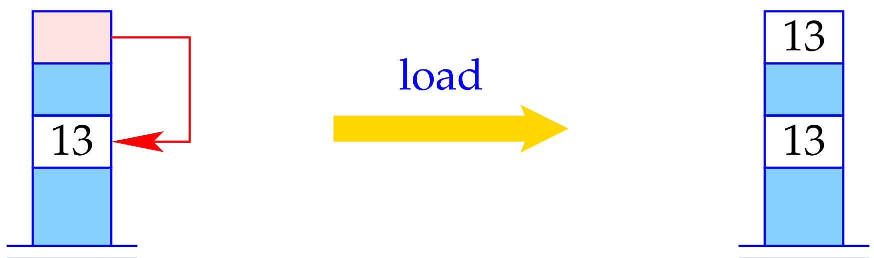
Nicht jeder Ausdruck verfügt über einen L-Wert (Bsp.: $x + 1$).

Wir definieren:

$$
\begin{array}{lcl}
\mathrm{code_R}\ (e_1 + e_2)\ \rho & = & \mathrm{code_R}\ e_1\ \rho \\
 & & \mathrm{code_R}\ e_2\ \rho \\
 & & \mathrm{add} \\
 & & \quad \text{... analog für die anderen binären Operatoren} \\
\mathrm{code_R}\ (-e)\ \rho & = & \mathrm{code_R}\ e\ \rho \\
 & & \mathrm{neg} \\
 & & \quad \text{... analog für andere unäre Operatoren} \\
\mathrm{code_R}\ q\ \rho & = & \mathrm{loadc\ q} \\
\mathrm{code_L}\ x\ \rho & = & \mathrm{loadc}\ (\rho\, x) \\
 & \ldots &
\end{array}
$$

$$\begin{array}{lll} \text{code}_R\; x\; \rho & = & \text{code}_L\; x\; \rho \\ & & \text{load} \end{array}$$

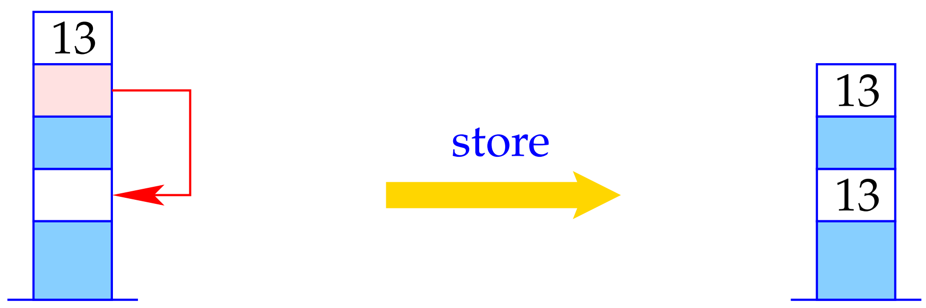
Die Instruktion load lädt den Wert der Speicherzelle, deren Adresse oben auf dem Stack liegt.



S[SP] = S[S[SP]];

$$\text{code}_R\ (x = e)\ \rho \quad = \quad \begin{array}{l} \text{code}_L\ x\ \rho \\ \text{code}_R\ e\ \rho \\ \text{store} \end{array}$$

Die Instruktion store schreibt den Inhalt der obersten Speicherzelle in die Speicherzelle, deren Adresse darunter auf dem Keller steht, lässt den geschriebenen Wert aber oben auf dem Keller liegen :-)



S[S[SP-1]] = S[SP];

S[SP-1] = S[SP]; SP--;

Beispiel: Code für $e \equiv x = y - 1$ mit $\rho = \{x \mapsto 4, y \mapsto 7\}$.
Dann liefert $\text{code}_\text{R}\ e\ \rho$:

$\boxed{\text{loadc 4}}$	loadc 7 load	loadc 1 sub	$\boxed{\text{store}}$

Optimierungen:

Einführung von Spezialbefehlen für häufige Befehlsfolgen, hier etwa:

```
loada q         =   loadc q
                    load
bla; storea q   =   loadc q; bla
                    store
```

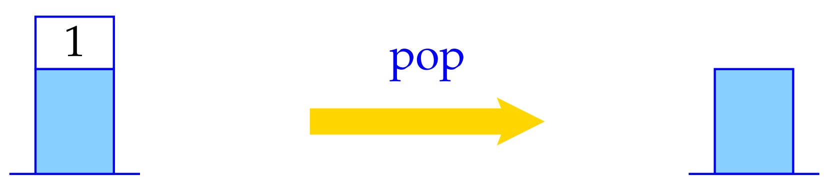
3 Anweisungen und Anweisungsfolgen

Ist e ein Ausdruck, dann ist $e;$ eine Anweisung (Statement).

Anweisungen liefern keinen Wert zurück. Folglich muss der SP vor und nach der Ausführung des erzeugten Codes gleich sein:

$$\begin{array}{lcl} \text{code}\; e;\; \rho & = & \text{code}_{\text{R}}\; e\; \rho \\ & & \text{pop} \end{array}$$

Die Instruktion pop wirft das oberste Element des Kellers weg ...



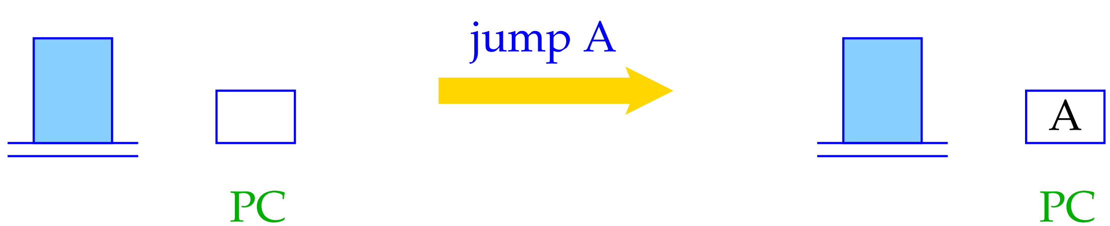
SP--;

Der Code für eine Statement-Folge ist die Konkatenation des Codes for die einzelnen Statements in der Folge:

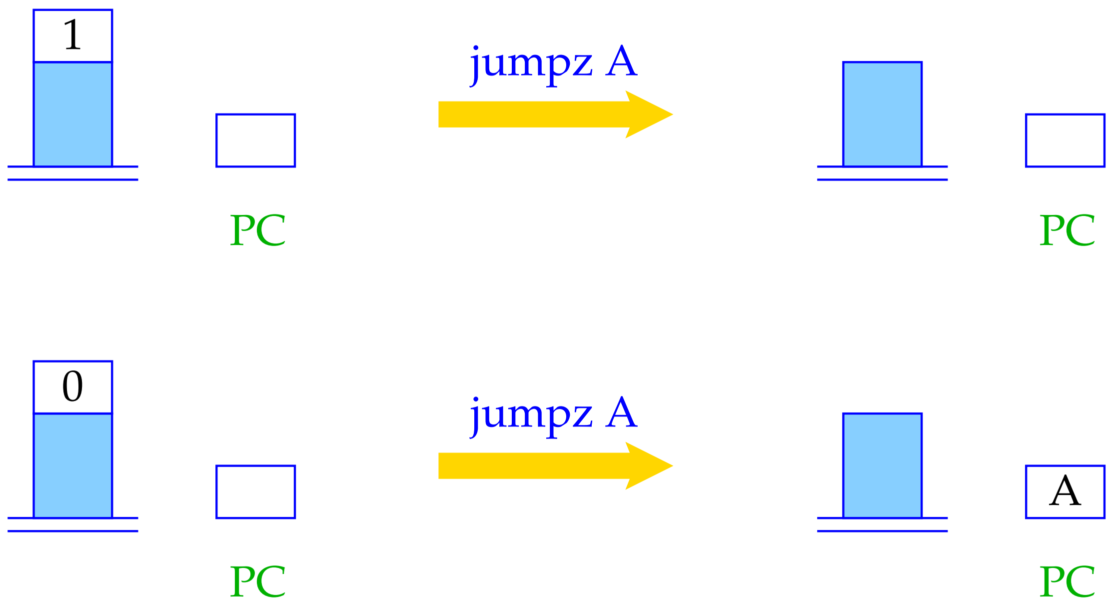
$$
\begin{array}{lcl}
\mathrm{code}\ (s\,ss)\ \rho & = & \mathrm{code}\ s\ \rho \\
 & & \mathrm{code}\ ss\ \rho \\
\mathrm{code}\ \varepsilon\ \rho & = & \qquad // \quad \textit{leere Folge von Befehlen}
\end{array}
$$

4 Bedingte und iterative Anweisungen

Um von linearer Ausführungsreihenfolge abzuweichen, benötigen wir Sprünge:



PC = A;



if (S[SP] == 0) PC = A;

SP--;

Der Übersichtlichkeit halber gestatten wir die Verwendung von symbolischen Sprungzielen. In einem zweiten Pass können diese dann durch absolute Code-Adressen ersetzt werden.

Statt absoluter Code-Adressen könnte man auch relative Adressen benutzen, d. h. Sprungziele relativ zum aktuellen PC angeben.

Vorteile:

- kleinere Adressen reichen aus;
- der Code wird relokierbar, d. h. kann im Speicher unverändert hin und her geschoben werden.

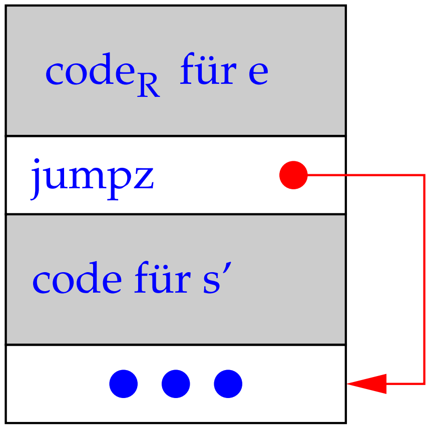
4.1 Bedingte Anweisung, einseitig

Betrachten wir zuerst $s \equiv \textbf{if } (e)\ s'$.

Idee:

- Lege den Code zur Auswertung von e und s' hintereinander in den Code-Speicher;
- Dekoriere mit Sprung-Befehlen so, dass ein korrekter Kontroll-Fluss gewährleistet ist!

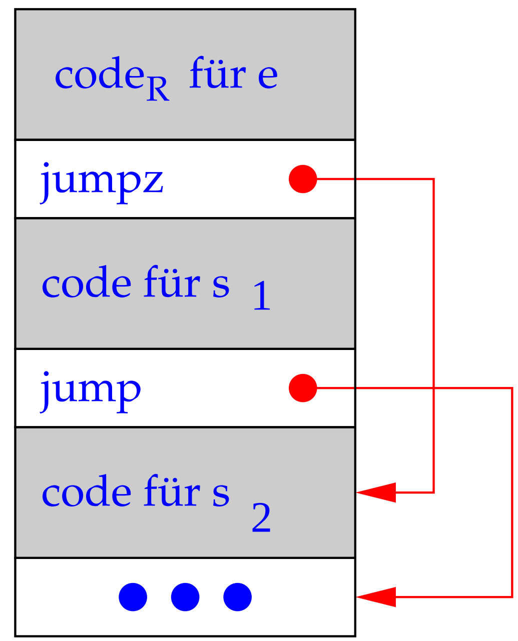
$$
\begin{aligned}
\text{code } s\ \rho \quad &= \quad \text{code}_\text{R}\ e\ \rho \\
& \quad\quad \text{jumpz A} \\
& \quad\quad \text{code } s'\ \rho \\
\text{A}: & \quad\quad \ldots
\end{aligned}
$$



4.2 Zweiseitiges if

Betrachte nun $s \equiv \textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2$. Die gleiche Strategie liefert:

$$
\begin{array}{lcl}
\text{code}\ s\ \rho & = & \text{code}_R\ e\ \rho \\
 & & \text{jumpz A} \\
 & & \text{code}\ s_1\ \rho \\
 & & \text{jump B} \\
 & \text{A}: & \text{code}\ s_2\ \rho \\
 & \text{B}: & \ldots
\end{array}
$$



Beispiel: Sei $\rho = \{x \mapsto 4, y \mapsto 7\}$ und

$$
\begin{array}{rcll}
s & \equiv & \textbf{if}\,(x > y) & (i) \\
 & & \quad x = x - y; & (ii) \\
 & & \textbf{else}\; y = y - x; & (iii)
\end{array}
$$

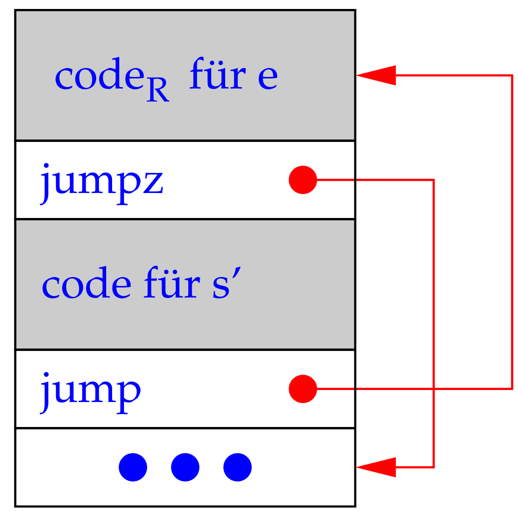
Dann liefert code $s\ \rho$:

loada 4	loada 4	A:	loada 7
loada 7	loada 7		loada 4
gr	sub		sub
jumpz A	storea 4		storea 7
	pop		pop
	jump B	B:	...
(i)	(ii)		(iii)

4.3 while-Schleifen

Betrachte schließlich die Schleife $s \equiv \textbf{while}\ (e)\ s'$. Dafür erzeugen wir:

$$
\begin{array}{lll}
\text{code}\ s\ \rho & = & \\
& \text{A}: & \text{code}_\text{R}\ e\ \rho \\
& & \text{jumpz B} \\
& & \text{code}\ s'\ \rho \\
& & \text{jump A} \\
& \text{B}: & \ldots
\end{array}
$$



Beispiel: Sei $\rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\}$ und s das Statement:

$$\textbf{while}\ (a > 0)\ \{c = c + 1;\ a = a - b;\}$$

Dann liefert code $s\ \rho$ die Folge:

A:	loada 7	loada 9	loada 7	B:	...
	loadc 0	loadc 1	loada 8		
	gr	add	sub		
	jumpz B	storea 9	storea 7		
		pop	pop		
			jump A		

4.4 for-Schleifen

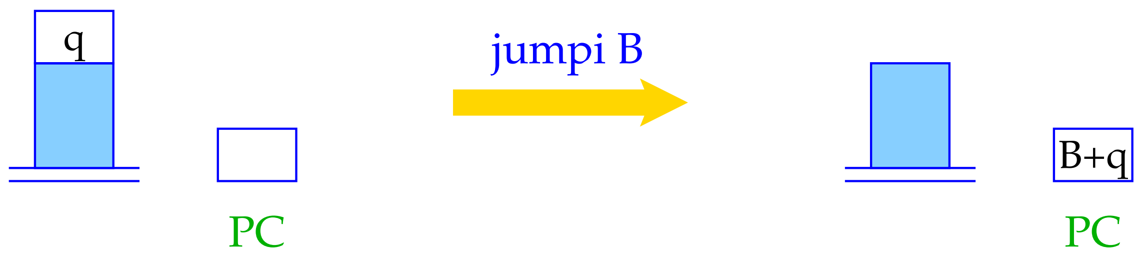
Die **for**-Schleife $s \equiv \textbf{for}\ (e_1; e_2; e_3)\ s'$ ist äquivalent zu der Statementfolge $e_1;\ \textbf{while}\ (e_2)\ \{s'\ e_3;\}$ – sofern s' keine **continue**-Anweisung enthält.
Darum übersetzen wir:

$$
\begin{array}{rcl}
\text{code}\ s\ \rho & = & \text{code}_\text{R}\ e_1 \\
 & & \text{pop} \\
 & \text{A}: & \text{code}_\text{R}\ e_2\ \rho \\
 & & \text{jumpz B} \\
 & & \text{code}\ s'\ \rho \\
 & & \text{code}_\text{R}\ e_3\ \rho \\
 & & \text{pop} \\
 & & \text{jump A} \\
 & \text{B}: & \ldots
\end{array}
$$

4.5 Das switch-Statement

Idee:

- Unterstütze Mehrfachverzweigung in konstanter Zeit!
- Benutze Sprungtabelle, die an der i-ten Stelle den Sprung an den Anfang der i-tem Alternative enthält.
- Eine Möglichkeit zur Realisierung besteht in der Einführung von indizierten Sprüngen.



PC = B + S[SP];

SP--;

Vereinfachung:

Wir betrachten nur **switch**-Statements der folgenden Form:

$$
\begin{array}{lll}
s & \equiv & \textbf{switch}\ (e)\ \{ \\
 & & \qquad \textbf{case}\ 0: \quad ss_0\ \textbf{break;} \\
 & & \qquad \textbf{case}\ 1: \quad ss_1\ \textbf{break;} \\
 & & \qquad\qquad \vdots \\
 & & \qquad \textbf{case}\ k-1: \quad ss_{k-1}\ \textbf{break;} \\
 & & \qquad \textbf{default:} \quad ss_k \\
 & & \}
\end{array}
$$

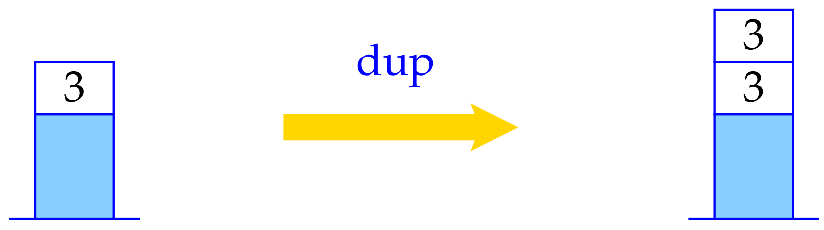
Dann ergibt sich für s die Instruktionsfolge:

$$
\begin{array}{llllllll}
\text{code}\ s\ \rho & = & \text{code}_R\ e\ \rho & C_0: & \text{code}\ ss_0\ \rho & B: & \text{jump}\ C_0 \\
 & & \text{check}\ 0\ k\ B & & \text{jump}\ D & & \ldots \\
 & & & & \ldots & & \text{jump}\ C_k \\
 & & & C_k: & \text{code}\ ss_k\ \rho & D: & \ldots \\
 & & & & \text{jump}\ D & &
\end{array}
$$

- Das Macro check $0\ k$ B überprüft, ob der R-Wert von e im Intervall $[0, k]$ liegt, und führt einen indizierten Sprung in die Tabelle B aus.
- Die Sprungtabelle enthält direkte Sprünge zu den jeweiligen Alternativen.
- Am Ende jeder Alternative steht ein Sprung hinter das **switch**-Statement.

check 0 k B	=	dup	dup		jumpi B
		loadc 0	loadc k	A:	pop
		geq	leq		loadc k
		jumpz A	jumpz A		jumpi B

- Weil der R-Wert von e noch zur Indizierung benötigt wird, muss er vor jedem Vergleich kopiert werden.
- Dazu dient der Befehl dup.
- Ist der R-Wert von e kleiner als 0 oder größer als k, ersetzen wir ihn vor dem indizierten Sprung durch k.



S[SP+1] = S[SP];

SP++;

Achtung:

- Die Sprung-Tabelle könnte genauso gut direkt hinter dem Macro check liegen. Dadurch spart man ein paar unbedingte Sprünge, muss aber evt. das **switch**-Statement zweimal durchsuchen.
- Beginnt die Tabelle mit u statt mit 0, müssen wir den R-Wert von e um u vermindern, bevor wir ihn als Index benutzen.
- Sind sämtliche möglichen Werte von e sicher im Intervall $[0, k]$, können wir auf check verzichten.

5 Speicherbelegung für Variablen

Ziel:

Ordne jeder Variablen x statisch, d. h. zur Übersetzungszeit, eine feste (Relativ-)Adresse $\quad \rho\, x \quad$ zu!

Annahmen:

- Variablen von Basistypen wie **int**, ... erhalten eine Speicherzelle.
- Variablen werden in der Reihenfolge im Speicher abgelegt, wie sie deklariert werden, und zwar ab Adresse 1.

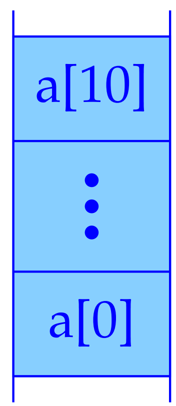
Folglich erhalten wir für die Deklaration $\quad d \equiv t_1\; x_1; \;\ldots\; t_k\; x_k; \quad (t_i$ einfach) die Adress-Umgebung ρ mit

$$\rho\, x_i = i, \quad i = 1, \ldots, k$$

5.1 Felder

Beispiel: **int** [11] a;

Das Feld a enthält 11 Elemente und benötigt darum 11 Zellen.
$\rho\, a$ ist die Adresse des Elements $a[0]$.



Notwendig ist eine Funktion sizeof (hier: $|\cdot|$), die den Platzbedarf eines Typs berechnet:

$$|t| = \begin{cases} 1 & \text{falls } t \text{ einfach} \\ k \cdot |t'| & \text{falls } t \equiv t'[k] \end{cases}$$

Dann ergibt sich für die Deklaration $\quad d \equiv t_1\ x_1;\ \ldots\ t_k\ x_k;$

$$\begin{array}{lll} \rho\, x_1 & = & 1 \\ \rho\, x_i & = & \rho\, x_{i-1} + |t_{i-1}| \qquad\qquad \text{für } i > 1 \end{array}$$

Weil $|\cdot|$ zur Übersetzungszeit berechnet werden kann, kann dann auch ρ zur Übersetzungszeit berechnet werden.

Aufgabe:

Erweitere $\mathrm{code_L}$ und $\mathrm{code_R}$ auf Ausdrücke mit indizierten Feldzugriffen.

Sei $\quad t[c]\ a; \quad$ die Deklaration eines Feldes $\quad a$.

Um die Anfangsadresse der Datenstruktur $\quad a[i] \quad$ zu bestimmen, müssen wir $\rho\, a + |t| * (R\text{-}Wert\ von\ i)$ ausrechnen. Folglich:

$$
\begin{array}{lll}
\mathrm{code_L}\ a[e]\ \rho & = & \mathrm{loadc}\ (\rho\, a) \\
 & & \mathrm{code_R}\ e\ \rho \\
 & & \mathrm{loadc}\ |t| \\
 & & \mathrm{mul} \\
 & & \mathrm{add}
\end{array}
$$

... oder allgemeiner:

$$
\begin{array}{lcl}
\text{code}_L\ e_1[e_2]\ \rho & = & \text{code}_R\ e_1\ \rho \\
 & & \text{code}_R\ e_2\ \rho \\
 & & \text{loadc}\ |t| \\
 & & \text{mul} \\
 & & \text{add}
\end{array}
$$

Bemerkung:

- In C ist ein Feld ein Zeiger. Ein deklariertes Feld a ist eine Zeiger-Konstante, deren R-Wert die Anfangsadresse des Feldes ist.
- Formal setzen wir für ein Feld e: $\text{code}_R\ e\ \rho = \text{code}_L\ e\ \rho$
- In C sind äquivalent (als L-Werte):

$$2[a] \qquad a[2] \qquad a+2$$

5.2 Strukturen

In Modula heißen Strukturen Records.

Vereinfachung:

Komponenten-Namen werden nicht anderweitig verwandt.
Alternativ könnte man zu jedem Struktur-Typ *st* eine separate Komponenten-Umgebung ρ_{st} verwalten :-)

Sei **struct** { **int** a; **int** b; } x; Teil einer Deklarationsliste.

- x erhält die erste freie Zelle des Platzes für die Struktur als Relativ-Adresse.
- Für die Komponenten vergeben wir Adressen relativ zum Anfang der Struktur, hier $a \mapsto 0$, $b \mapsto 1$.

Sei allgemein $t \equiv \textbf{struct}\,\{t_1\ c_1; \ldots t_k\ c_k;\}$. Dann ist

$$
\begin{aligned}
|t| &= \sum_{i=1}^{k} |t_i| \\
\rho\, c_1 &= 0 \quad \text{und} \\
\rho\, c_i &= \rho\, c_{i-1} + |t_{i-1}| \quad \text{für } i > 1
\end{aligned}
$$

Damit erhalten wir:

$$
\begin{aligned}
\mathrm{code}_{\mathrm{L}}\ (e.c)\ \rho \quad = \quad & \mathrm{code}_{\mathrm{L}}\ e\ \rho \\
& \mathrm{loadc}\ (\rho\, c) \\
& \mathrm{add}
\end{aligned}
$$

Beispiel:

Sei **struct** { **int** a; **int** b; } x; mit $\rho = \{x \mapsto 13, a \mapsto 0, b \mapsto 1\}$.

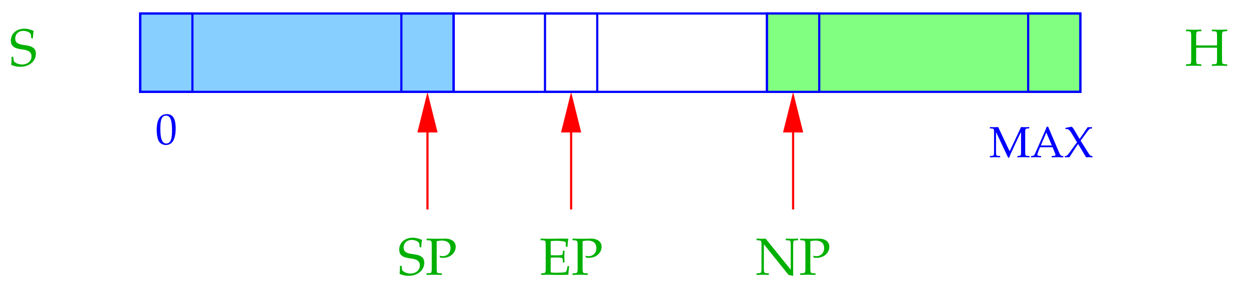
Dann ist

$$\text{code}_\text{L}\ (x.b)\ \rho \quad = \quad \begin{array}{l} \texttt{loadc 13} \\ \texttt{loadc 1} \\ \texttt{add} \end{array}$$

6 Zeiger und dynamische Speicherverwaltung

Zeiger (Pointer) gestatten den Zugriff auf anonyme, dynamisch erzeugte Datenelemente, deren Lebenszeit nicht dem LIFO-Prinzip unterworfen ist.

$\Longrightarrow$ Wir benötigen eine weitere potentiell beliebig große Datenstruktur H – den Heap (bzw. die Halde):



NP $\widehat{=}$ New Pointer; zeigt auf unterste belegte Haldenzelle.

EP $\widehat{=}$ Extreme Pointer; zeigt auf die Zelle, auf die der SP maximal zeigen kann (innerhalb der aktuellen Funktion).

Idee dabei:

- Chaos entsteht, wenn Stack und Heap sich überschneiden (Stack Overflow).
- Eine Überschneidung kann bei jeder Erhöhung von SP, bzw. jeder Erniedrigung des NP eintreten.
- EP erspart uns die Überprüfungen auf Überschneidung bei den Stackoperationen :-)
- Die Überprüfungen bei Heap-Allokationen bleiben erhalten :-(.

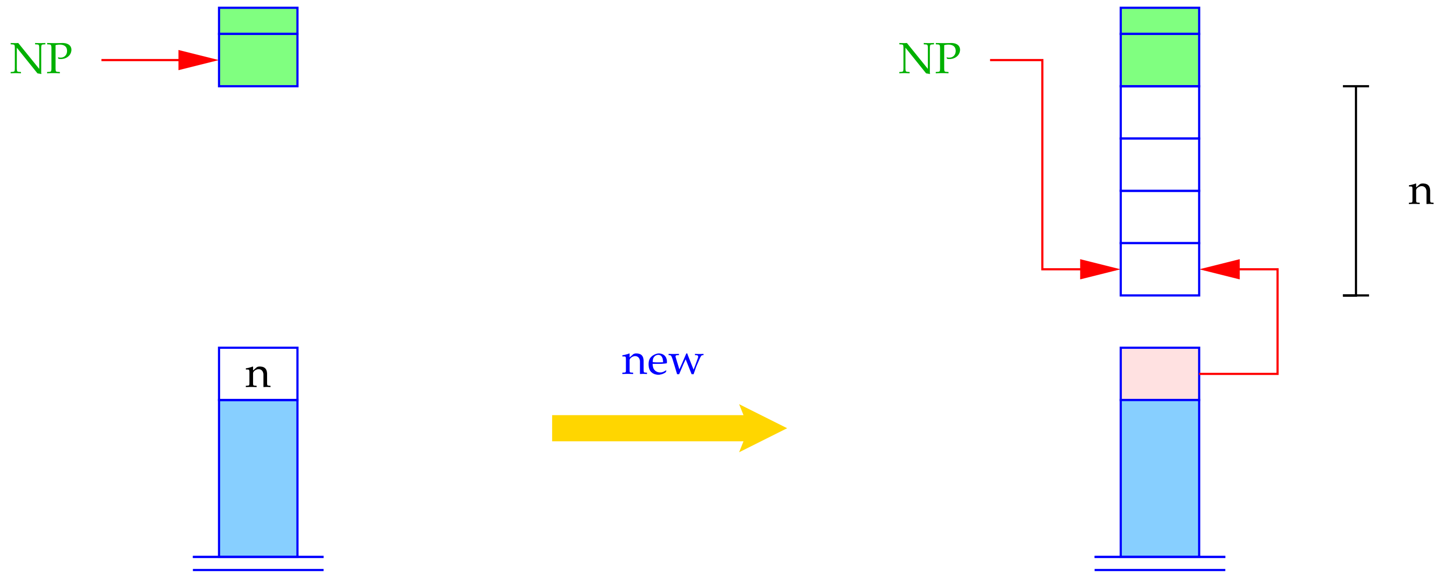
Mit Zeiger (-Werten) rechnen, heißt in der Lage zu sein,

- Zeiger zu erzeugen, d.h. Zeiger auf Speicherzellen zu setzen; sowie
- Zeiger zu dereferenzieren, d. h. durch Zeiger auf die Werte von Speicherzellen zugreifen.

Es gibt zwei Arten, Zeiger zu erzeugen:

(1) Ein Aufruf von **malloc** liefert einen Zeiger auf eine Heap-Zelle:

$$\begin{array}{lll} \text{code}_\text{R}\ \mathbf{malloc}\,(e)\ \rho & = & \text{code}_\text{R}\ e\ \rho \\ & & \text{new} \end{array}$$


NP
n
new
NP
n

```
if (NP - S[SP] ≤ EP)
        S[SP] = NULL;
else {
        NP = NP - S[SP];
        S[SP] = NP;
}
```

- NULL ist eine spezielle Zeigerkonstante (etwa 0 :-)
- Im Falle einer Kollision von Stack und Heap wird der NULL-Zeiger zurückgeliefert.

(2) Die Anwendung des Adressoperators & liefert einen Zeiger auf eine Variable, d. h. deren Adresse ($\widehat{=}$ L-Wert). Deshalb:

$$\text{code}_\text{R}\ (\&e)\ \rho = \text{code}_\text{L}\ e\ \rho$$

Dereferenzieren von Zeigern:

Die Anwendung des Operators $*$ auf den Ausdruck e liefert den Inhalt der Speicherzelle, deren Adresse der R-Wert von e ist:

$$\text{code}_\text{L}\ (*e)\ \rho = \text{code}_\text{R}\ e\ \rho$$

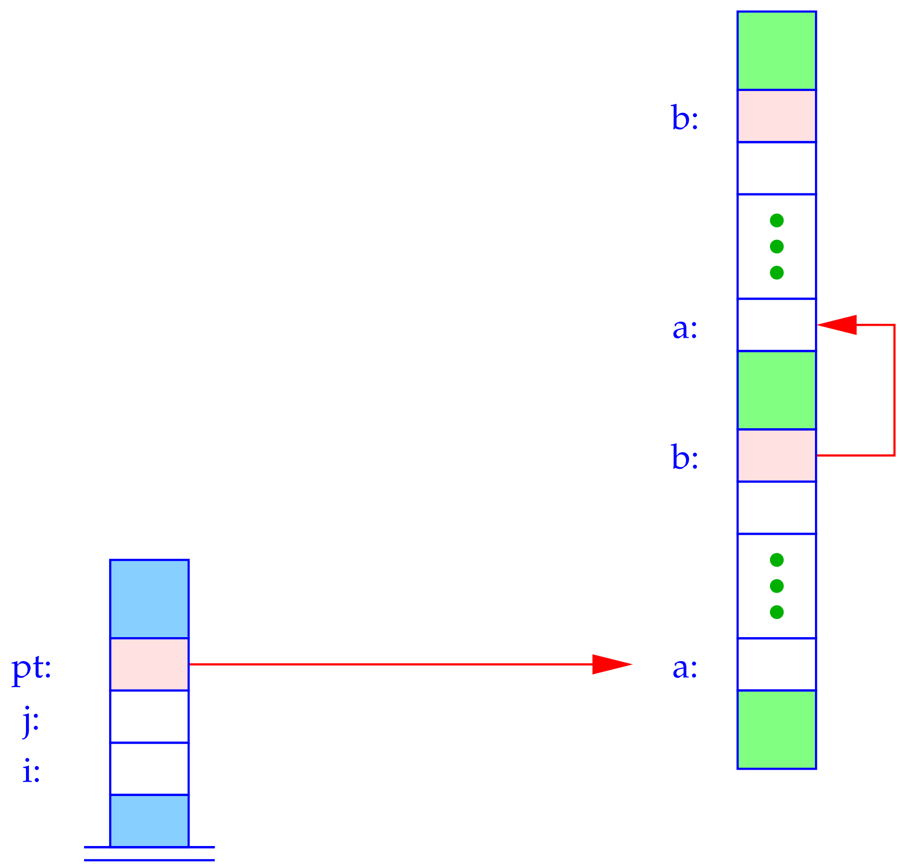
Beispiel: Betrachte für

$$\textbf{struct } t \, \{ \, \textbf{int } a[7]; \textbf{struct } t \, *b; \};$$
$$\textbf{int } i, j;$$
$$\textbf{struct } t \, *pt;$$

den Ausdruck $e \equiv ((pt \to b) \to a)[i+1]$

Wegen $e \to a \equiv (*e).a$ gilt:

$$\begin{aligned} \text{code}_\text{L}\,(e \to a)\;\rho \quad &= \quad \text{code}_\text{R}\; e\;\rho \\ & \qquad \text{loadc}\;(\rho\, a) \\ & \qquad \text{add} \end{aligned}$$


b:
a:
b:
pt:
a:
j:
i:

Sei $\quad \rho = \{i \mapsto 1, j \mapsto 2, pt \mapsto 3, a \mapsto 0, b \mapsto 7\}$. Dann ist:

$$
\begin{array}{lllll}
\text{code}_\text{L}\; e\; \rho & = & \text{code}_\text{R}\; ((pt \to b) \to a)\; \rho & = & \text{code}_\text{R}\; ((pt \to b) \to a)\; \rho \\
 & & \text{code}_\text{R}\; (i+1)\; \rho & & \text{loada } 1 \\
 & & \text{loadc } 1 & & \text{loadc } 1 \\
 & & \text{mul} & & \text{add} \\
 & & \text{add} & & \text{loadc } 1 \\
 & & & & \text{mul} \\
 & & & & \text{add}
\end{array}
$$

Für Felder ist der R-Wert gleich dem L-Wert. Deshalb erhalten wir:

$$\text{code}_\text{R}\ ((pt \to b) \to a)\ \rho \quad = \quad \text{code}_\text{R}\ (pt \to b)\ \rho \quad = \quad \text{loada } 3$$

	loadc 0	loadc 7
	add	add
		load
		loadc 0
		add

Damit ergibt sich insgesamt die Folge:

loada 3	load	loada 1	loadc 1
loadc 7	loadc 0	loadc 1	mul
add	add	add	add

7 Zusammenfassung

Stellen wir noch einmal die Schemata zur Übersetzung von Ausdrücken zusammen.

$$
\begin{array}{lcll}
\mathrm{code_L}\ (e_1[e_2])\ \rho & = & \mathrm{code_R}\ e_1\ \rho & \\
 & & \mathrm{code_R}\ e_2\ \rho & \\
 & & \mathrm{loadc}\ |t| & \\
 & & \mathrm{mul} & \\
 & & \mathrm{add} & \text{sofern } e_1 \text{ Typ } t\,[\,] \text{ hat}
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{code_L}\ (e.a)\ \rho & = & \mathrm{code_L}\ e\ \rho \\
 & & \mathrm{loadc}\ (\rho\, a) \\
 & & \mathrm{add}
\end{array}
$$

$$
\begin{array}{lcll}
\text{code}_L\ (*e)\ \rho & = & \text{code}_R\ e\ \rho & \\[2em]
\text{code}_L\ x\ \rho & = & \text{loadc}\ (\rho\, x) & \\[2em]
\text{code}_R\ (\&e)\ \rho & = & \text{code}_L\ e\ \rho & \\[2em]
\text{code}_R\ (\text{malloc}\,(e))\ \rho & = & \text{code}_R\ e\ \rho & \\
 & & \text{new} & \\[2em]
\text{code}_R\ e\ \rho & = & \text{code}_L\ e\ \rho & \text{falls } e \text{ ein Feld ist} \\[2em]
\text{code}_R\ (e_1 \Box e_2)\ \rho & = & \text{code}_R\ e_1\ \rho & \\
 & & \text{code}_R\ e_2\ \rho & \\
 & & \text{op} & \text{op Befehl zu Operator `}\Box\text{'}
\end{array}
$$

$$\begin{array}{llll}
\mathrm{code_R}\ q\ \rho & = & \mathtt{loadc\ q} & q \text{ Konstante} \\[2ex]
\mathrm{code_R}\ (e_1 = e_2)\ \rho & = & \mathrm{code_L}\ e_1\ \rho & \\
& & \mathrm{code_R}\ e_2\ \rho & \\
& & \mathtt{store} & \\[2ex]
\mathrm{code_R}\ e\ \rho & = & \mathrm{code_L}\ e\ \rho & \\
& & \mathtt{load} & \text{sonst}
\end{array}$$

Beispiel: $\quad$ **int** $a[10],\ *b;$ $\quad$ mit $\rho = \{a \mapsto 7, b \mapsto 17\}$.

Betrachte das Statement: $\quad s_1 \equiv *a = 5;$

Dann ist:

$$\begin{array}{lcl} \mathrm{code_L}\ (*a)\ \rho & = & \mathrm{code_R}\ a\ \rho \quad = \quad \mathrm{code_L}\ a\ \rho \quad = \quad \mathrm{loadc}\ 7 \\ \mathrm{code}\ s_1\ \rho & = & \mathrm{loadc}\ 7 \\ & & \mathrm{loadc}\ 5 \\ & & \mathrm{store} \\ & & \mathrm{pop} \end{array}$$

Zur Übung übersetzen wir auch noch:

$$s_2 \equiv b = (\&a) + 2; \qquad \text{und} \qquad s_3 \equiv *(b + 3) = 5;$$

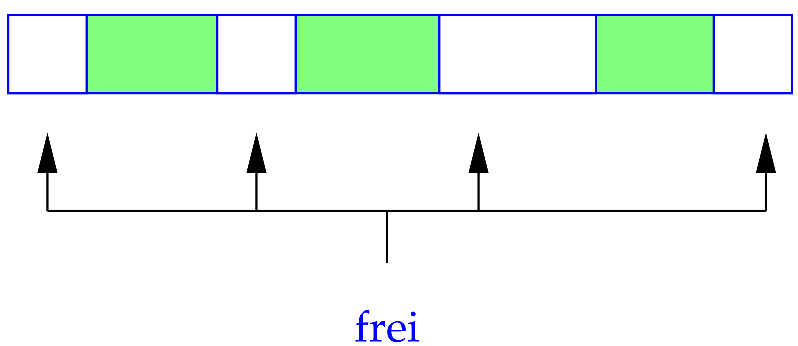
code $(s_2 s_3)\ \rho$ =

loadc 17		loadc 17	
loadc 7		load	
loadc 2		loadc 3	
loadc 1	// *Skalierung*	loadc 1	// *Skalierung*
mul		mul	
add		add	
store		loadc 5	
pop	// *Ende von* s_2	store	
		pop	// *Ende von* s_3

8 Freigabe von Speicherplatz

Probleme:

- Der freigegebene Speicherbereich wird noch von anderen Zeigern referenziert (dangling references).
- Nach einiger Freigabe könnte der Speicher etwa so aussehen (fragmentation):



Mögliche Auswege:

- Nimm an, der Programmierer weiß, was er tut. Verwalte dann die freien Abschnitte (etwa sortiert nach Größe) in einer speziellen Datenstruktur;

  $\Longrightarrow$ **malloc** wird teuer :-(

- Tue nichts, d.h.:

$$\text{code } \mathbf{free}\,(e);\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\text{pop}$$

  $\Longrightarrow$ einfach und (i.a.) effizient :-)

- Benutze eine automatische, evtl. “konservative” Garbage-Collection, die gelegentlich sicher nicht mehr benötigten Heap-Platz einsammelt und dann **malloc** zur Verfügung stellt.

9 Funktionen

Die Definition einer Funktion besteht aus

- einem Namen, mit dem sie aufgerufen werden kann;
- einer Spezifikation der formalen Parameter;
- evtl. einem Ergebnistyp;
- einem Anweisungsteil.

In C gilt:

$$\text{code}_{\text{R}}\ f\ \rho \quad = \quad \text{load c\_}f \quad = \quad \text{Anfangsadresse des Codes für } f$$

$\Longrightarrow$ Auch Funktions-Namen müssen in der Adress-Umgebung verwaltet werden!

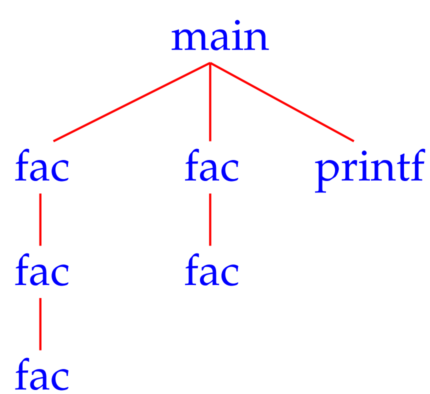
Beispiel:

```
int fac (int x) {
    if (x ≤ 0) return 1;
    else return x ∗ fac(x − 1);
}
```

```
main () {
    int n;
    n = fac(2) + fac(1);
    printf (“%d”, n);
}
```

Zu einem Ausführungszeitpunkt können mehrere Instanzen (Aufrufe) der gleichen Funktion aktiv sein, d. h. begonnen, aber noch nicht beendet sein.

Der Rekursionsbaum im Beispiel:



Wir schließen:

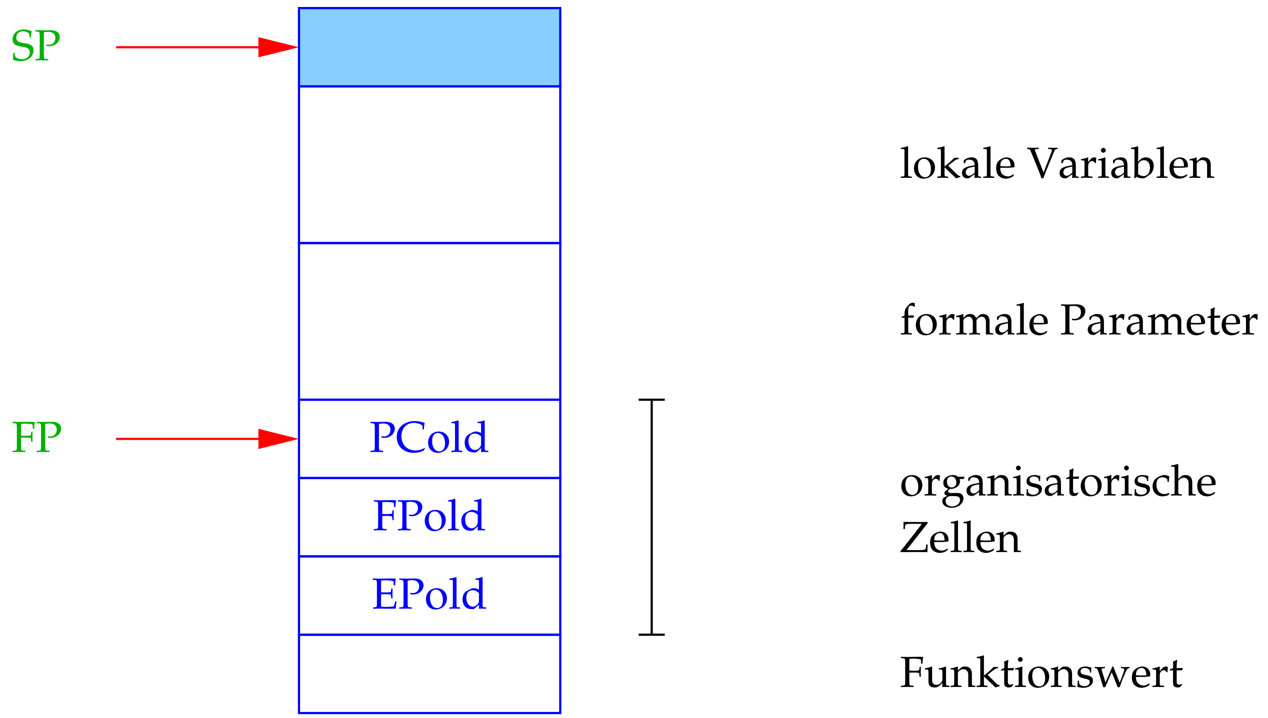
Die formalen Parameter und lokalen Variablen der verschiedenen Aufrufe der selben Funktion (Instanzen) müssen auseinander gehalten werden.

Idee:

Lege einen speziellen Speicherbereich für jeden Aufruf einer Funktion an.

In sequentiellen Programmiersprachen können diese Speicherbereiche auf dem Keller verwaltet werden. Deshalb heißen sie auch Keller-Rahmen (oder Stack Frame).

9.1 Speicherorganisation für Funktionen



FP $\widehat{=}$ Frame Pointer; zeigt auf die letzte organisatorische Zelle und wird zur Adressierung der formalen Parameter und lokalen Variablen benutzt.

- Die lokalen Variablen und formalen Parameter adressieren wir relativ zu FP.
- Bei einem Funktions-Aufruf muss der FP in eine organisatorische Zelle gerettet werden.
- Weiterhin müssen gerettet werden:
  - die Fortsetzungsadresse nach dem Aufruf;
  - der aktuelle EP.

Vereinfachung: Der Rückgabewert passt in eine einzige Zelle.

Unsere Übersetzungsaufgaben für Funktionen:

- Erzeuge Code für den Rumpf!
- Erzeuge Code für Aufrufe!

9.2 Bestimmung der Adress-Umgebung

Wir müssen zwei Arten von Variablen unterscheiden:

1. globale/externe, die außerhalb von Funktionen definiert werden;
2. lokale/interne/automatische (inklusive formale Parameter), die innerhalb von Funktionen definiert werden.

$$\Longrightarrow$$

Die Adress-Umgebung ρ ordnet den Namen Paare $(tag, a) \in \{G, L\} \times \mathbb{N}_0$ zu.

Achtung:

- Tatsächlich gibt es i.a. weitere verfeinerte Abstufungen der Sichtbarkeit von Variablen.
- Bei der Übersetzung eines Programms gibt es i.a. für verschiedene Programmteile verschiedene Adress-Umgebungen!

Beispiel:

$\boxed{0}$

```
int i;
struct list {
    int info;
    struct list * next;
} * l;
```

$\boxed{1}$

```
int ith (struct list * x, int i) {
    if (i ≤ 1) return x →info;
    else return ith (x →next, i − 1);
}
```

$\boxed{2}$

```
main () {
    int k;
    scanf ("%d", &i);
    scanlist (&l);
    printf ("\n\t%d\n", ith (l,i));
}
```

Vorkommende Adress-Umgebungen in dem Programm:

$\boxed{0}$ Außerhalb der Funktions-Definitionen:

$$
\begin{array}{lrcl}
\rho_0: & i & \mapsto & (G, 1) \\
 & l & \mapsto & (G, 2) \\
 & \text{ith} & \mapsto & (G, \text{\_ith}) \\
 & \text{main} & \mapsto & (G, \text{\_main}) \\
 & & \dots &
\end{array}
$$

$\boxed{1}$ Innerhalb von ith:

$$
\begin{array}{lrcl}
\rho_1: & i & \mapsto & (L, 2) \\
 & x & \mapsto & (L, 1) \\
 & l & \mapsto & (G, 2) \\
 & \text{ith} & \mapsto & (G, \text{\_ith}) \\
 & \text{main} & \mapsto & (G, \text{\_main}) \\
 & & \dots &
\end{array}
$$

$\boxed{2}$ Innerhalb von main:

$$\begin{array}{llcl}\rho_2: & i & \mapsto & (G, 1) \\ & l & \mapsto & (G, 2) \\ & k & \mapsto & (L, 1) \\ & \text{ith} & \mapsto & (G, \_\text{ith}) \\ & \text{main} & \mapsto & (G, \_\text{main}) \\ & & \ldots & \end{array}$$

9.3 Betreten und Verlassen von Funktionen

Sei f die aktuelle Funktion, d. h. der Caller, und f rufe die Funktion g auf, d. h. den Callee.

Der Code für den Aufruf muss auf den Caller und den Callee verteilt werden. Die Aufteilung kann nur so erfolgen, dass der Teil, der von Informationen des Callers abhängt, auch dort erzeugt wird (analog für den Callee).

Aktionen beim Betreten von g:

1.	Retten von FP, EP	mark	stehen in f
2.	Bestimmung der aktuellen Parameter		
3.	Bestimmung der Anfangsadresse von g		
4.	Setzen des neuen FP	call	
5.	Retten von PC und Sprung an den Anfang von g		
6.	Setzen des neuen EP	enter	stehen in g
7.	Allokieren der lokalen Variablen	alloc	

Aktionen beim Verlassen von g:

1.	Rücksetzen der Register FP, EP, SP	return
2.	Rücksprung in den Code von f, d. h. Restauration des PC	

Damit erhalten wir für einen Aufruf:

$$
\begin{aligned}
\mathrm{code_R}\ g(e_1, \ldots, e_n)\ \rho \quad = \quad & \mathrm{mark} \\
& \mathrm{code_R}\ e_1\ \rho \\
& \ldots \\
& \mathrm{code_R}\ e_n\ \rho \\
& \mathrm{code_R}\ g\ \rho \\
& \mathrm{call\ m}
\end{aligned}
$$

wobei m der Platz für die aktuellen Parameter ist.

Beachte:

- Von jedem Ausdruck, der als aktueller Parameter auftritt, wird jeweils der R-Wert berechnet $\Longrightarrow$ Call-by-Value-Parameter-Übergabe.
- Die Funktion g kann auch ein Ausdruck sein, dessen R-Wert die Anfangs-Adresse der aufzurufenden Funktion liefert ...

- Ähnlich deklarierten Feldern, werden Funktions-Namen als konstante Zeiger auf Funktionen aufgefasst. Dabei ist der R-Wert dieses Zeigers gleich der Anfangs-Adresse der Funktion.
- Achtung! Für eine Variable **int** $(*g)()$; sind die beiden Aufrufe

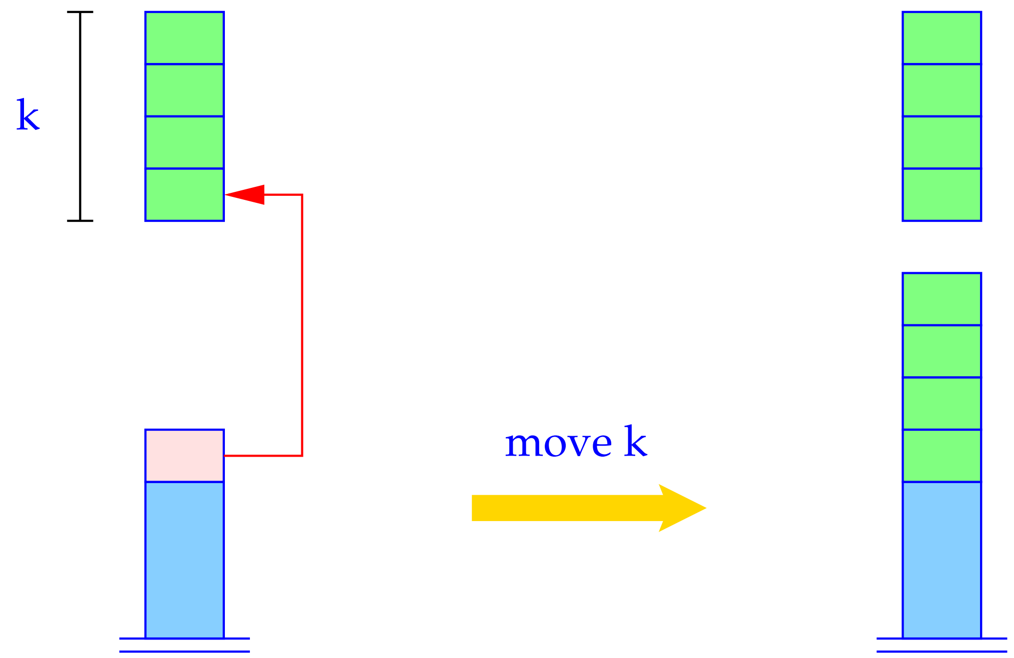
$$(*g)() \qquad \text{und} \qquad g()$$

  äquivalent! Per Normalisierung, muss man sich hier vorstellen, werden Dereferenzierungen eines Funktions-Zeigers ignoriert :-)
- Bei der Parameter-Übergabe von Strukturen werden diese kopiert.

Folglich:

$$\begin{array}{lcll}
\text{code}_\text{R}\ f\ \rho & = & \text{loadc}\ (\rho\, f) & f \text{ ein Funktions-Name} \\
\text{code}_\text{R}\ (*e)\ \rho & = & \text{code}_\text{R}\ e\ \rho & e \text{ ein Funktions-Zeiger} \\
\text{code}_\text{R}\ e\ \rho & = & \text{code}_\text{L}\ e\ \rho & \\
 & & \text{move k} & e \text{ eine Struktur der Größe } k
\end{array}$$

Dabei ist:

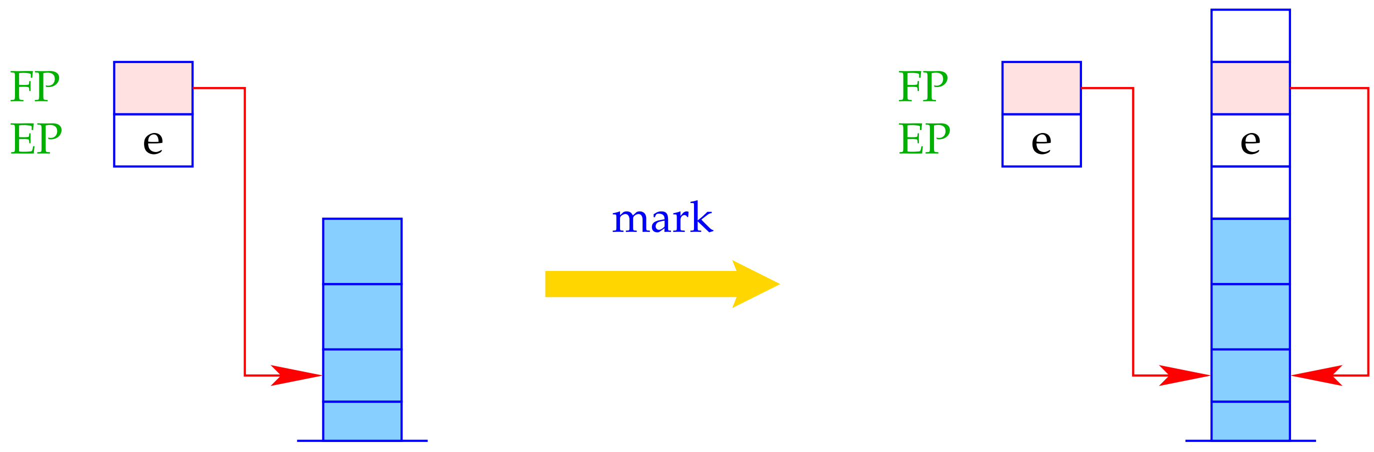


for (i = k-1; i≥0; i--)

  S[SP+i] = S[S[SP]+i];

SP = SP+k–1;

Der Befehl mark legt Platz für Rückgabewert und organisatorische Zellen an und rettet FP und EP.

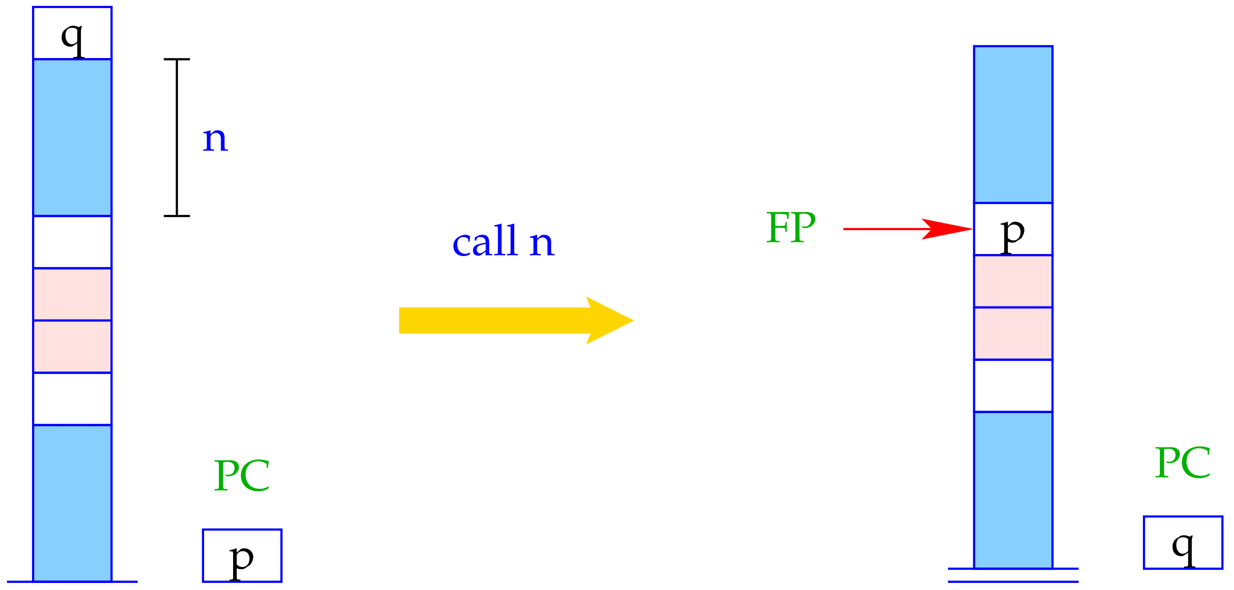


S[SP+2] = EP;

S[SP+3] = FP;

SP = SP + 4;

Der Befehl call n rettet die Fortsetzungs-Adresse und setzt FP, SP und PC auf die aktuellen Werte.



FP = SP - n - 1;

S[FP] = PC;

PC = S[SP];

SP--;

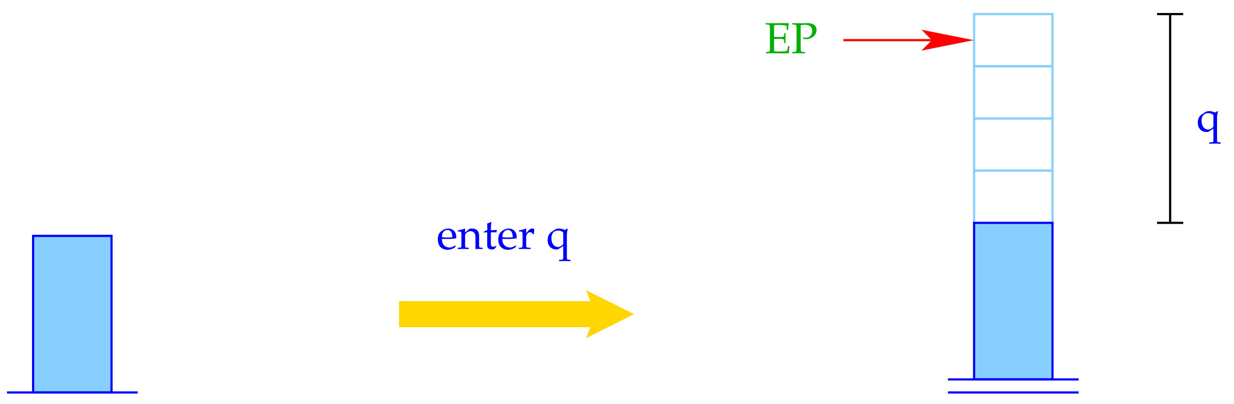
Entsprechend übersetzen wir eine Funktions-Definition:

$$\text{code}\quad t\ \text{f}\ (specs)\{V\_defs\quad ss\}\quad \rho\quad =$$

_f:	enter q	//	*setzen des EP*
	alloc k	//	*Anlegen der lokalen Variablen*
	code $ss\ \rho_{\text{f}}$		
	return	//	*Verlassen der Funktion*

wobei	q	=	$max + \text{k}$	wobei
	max	=	maximale Länge des lokalen Kellers	
	k	=	Platz für die lokalen Variablen	
	ρ_{f}	=	Adress-Umgebung für f	
			// *berücksichtigt specs, V_defs* und ρ	

Der Befehl enter q setzt den EP auf den neuen Wert. Steht nicht mehr genügend Platz zur Verfügung, wird die Programm-Ausführung abgebrochen.

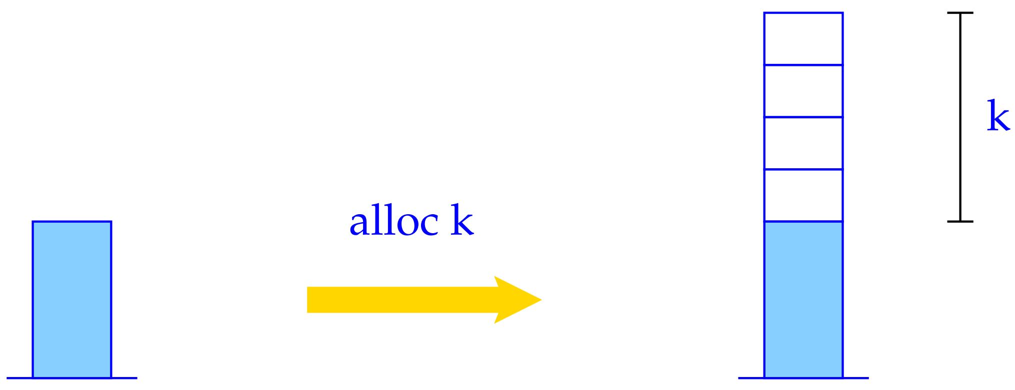


EP = SP + q;

if (EP $\geq$ NP)

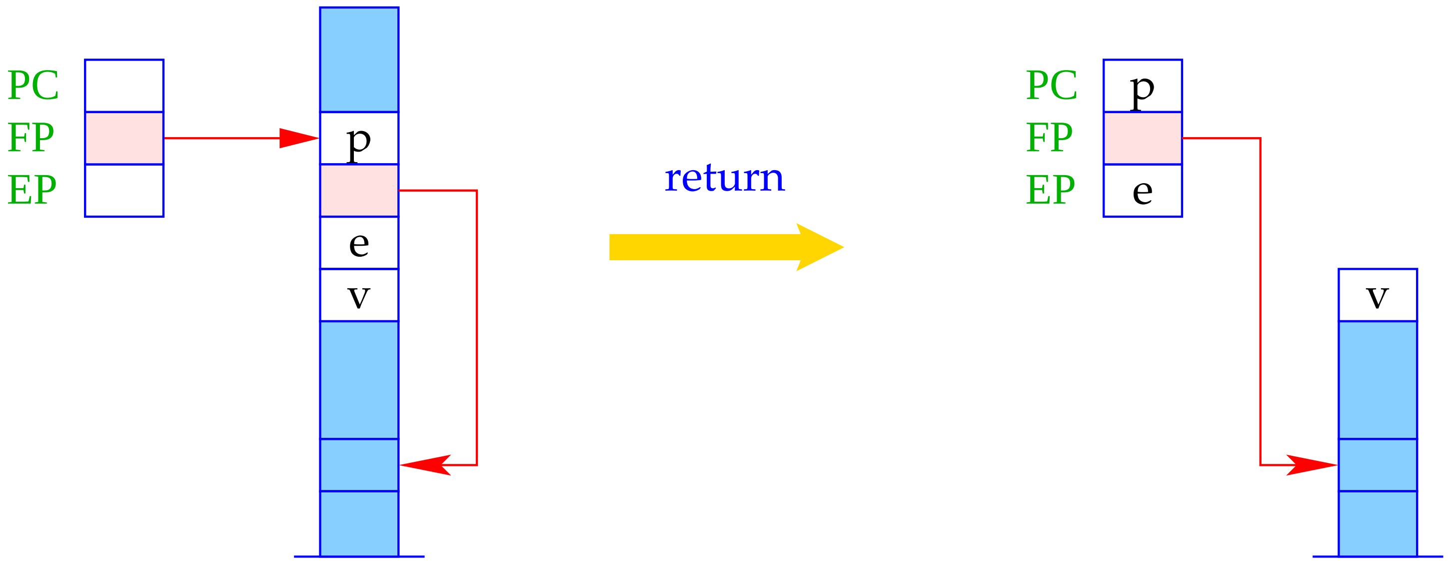
Error ("Stack Overflow");

Der Befehl alloc k reserviert auf dem Keller Platz für die lokalen Variablen.



SP = SP + k;

Der Befehl return gibt den aktuellen Keller-Rahmen auf. D.h. er restauriert die Register PC, EP und FP und hinterlässt oben auf dem Keller den Rückgabe-Wert.



PC = S[FP]; EP = S[FP-2];

if (EP $\geq$ NP) Error ("Stack Overflow");

SP = FP-3; FP = S[SP+2];

9.4 Zugriff auf Variablen, formale Parameter und Rückgabe von Werten

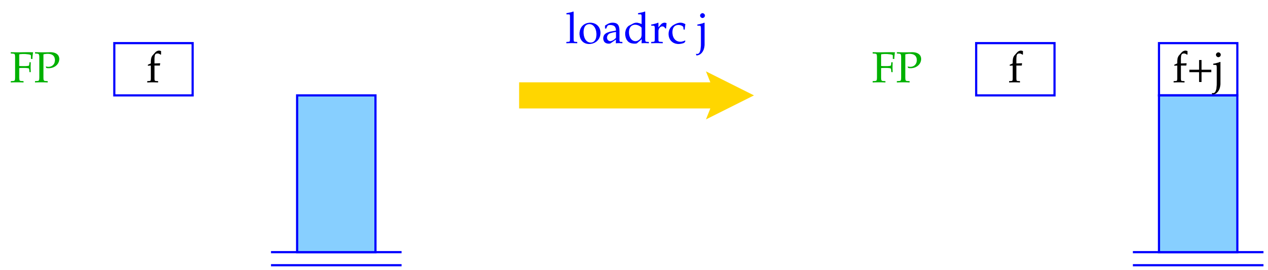
Zugriffe auf lokale Variablen oder formale Parameter erfolgen relativ zum aktuellen FP.

Darum modifizieren wir code_L für Variablen-Namen.

Für $\quad \rho\, x = (tag, j) \quad$ definieren wir

$$\text{code}_\text{L}\; x\; \rho = \begin{cases} \text{loadc j} & tag = G \\ \text{loadrc j} & tag = L \end{cases}$$

Der Befehl loadrc j berechnet die Summe von FP und j.



SP++;

S[SP] = FP+j;

Als Optimierung führt man analog zu loada j und storea j die Befehle loadr j und storer j ein:

$$\begin{array}{lcl} \text{loadr j} & = & \text{loadrc j} \\ & & \text{load} \\ \\ \text{bla; storer j} & = & \text{loadrc j; bla} \\ & & \text{store} \end{array}$$

Der Code für **return** e; entspricht einer Zuweisung an eine Variable mit Relativadresse -3.

$$\text{code } \mathbf{return}\, e;\ \rho \quad = \quad \begin{array}{l} \text{code}_\text{R}\ e\ \rho \\ \text{storer -3} \\ \text{return} \end{array}$$

Beispiel: Für die Funktion

$$\begin{array}{l} \mathbf{int}\ \text{fac}\ (\mathbf{int}\ x)\ \{ \\ \quad \mathbf{if}\ (x \leq 0)\ \mathbf{return}\ 1; \\ \quad \mathbf{else\ return}\ x * \text{fac}\ (x-1); \\ \} \end{array}$$

erzeugen wir:

```
_fac:   enter q
        alloc 0
        loadr 1
        loadc 0
        leq
        jumpz A
        loadc 1
        storer -3
        return
        jump B
A:      loadr 1
        mark
        loadr 1
        loadc 1
        sub
        loadc _fac
        call 1
        mul
        storer -3
        return
B:      return
```

Dabei ist $\rho_{\text{fac}} : x \mapsto (L, 1)$ und $\text{q} = 1 + 6 = 7$.

10 Übersetzung ganzer Programme

Vor der Programmausführung gilt:

$$\text{SP} = -1 \qquad \text{FP} = \text{EP} = 0 \qquad \text{PC} = 0 \qquad \text{NP} = \text{MAX}$$

Sei $p \equiv V\_defs \quad F\_def_1 \ldots F\_def_n,$ ein Programm, wobei F_def_i eine Funktion f_i definiert, von denen eine main heißt.

Der Code für das Programm p enthält:

- Code für die Funktions-Definitionen F_def_i;
- Code zum Anlegen der globalen Variablen;
- Code für den Aufruf von main();
- die Instruktion halt.

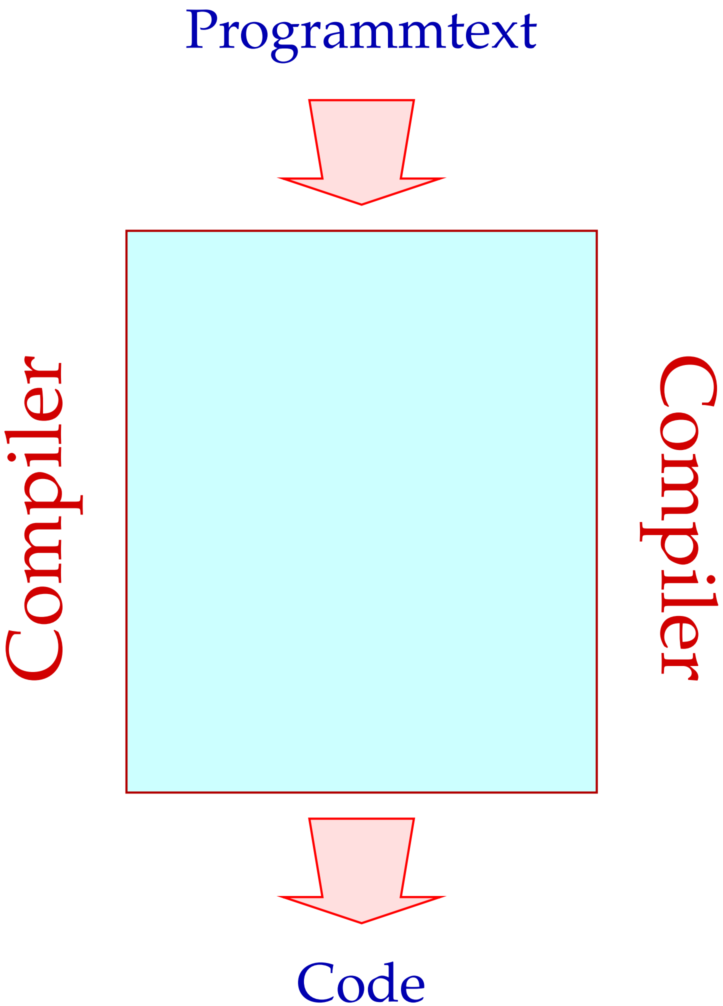
Dann definieren wir:

$$\begin{array}{llrl}
\text{code } p\ \emptyset & = & & \text{enter } (\text{k}+6) \\
 & & & \text{alloc } (\text{k}+1) \\
 & & & \text{mark} \\
 & & & \text{loadc _main} \\
 & & & \text{call } 0 \\
 & & & \text{pop} \\
 & & & \text{halt} \\
 & & \text{_f}_1: & \text{code } F_def_1\ \rho \\
 & & & \vdots \\
 & & \text{_f}_n: & \text{code } F_def_n\ \rho
\end{array}$$

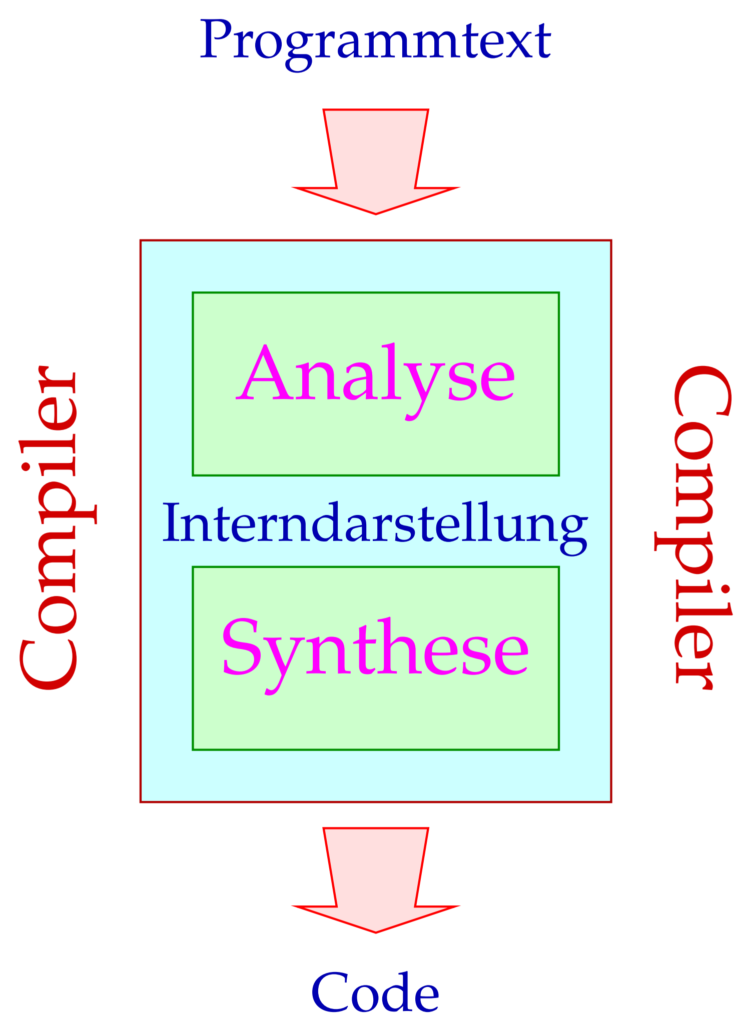
wobei	$\emptyset$	$\widehat{=}$	leere Adress-Umgebung;
	ρ	$\widehat{=}$	globale Adress-Umgebung;
	k	$\widehat{=}$	Platz für globale Variablen

Die Analyse-Phase

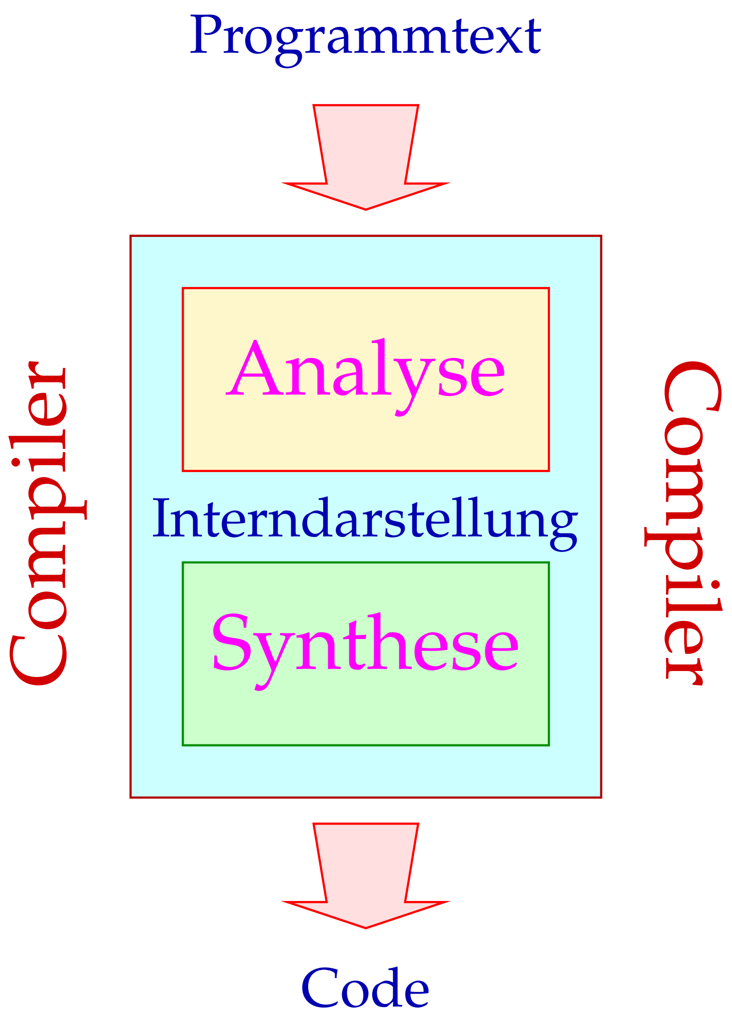
Orientierung:



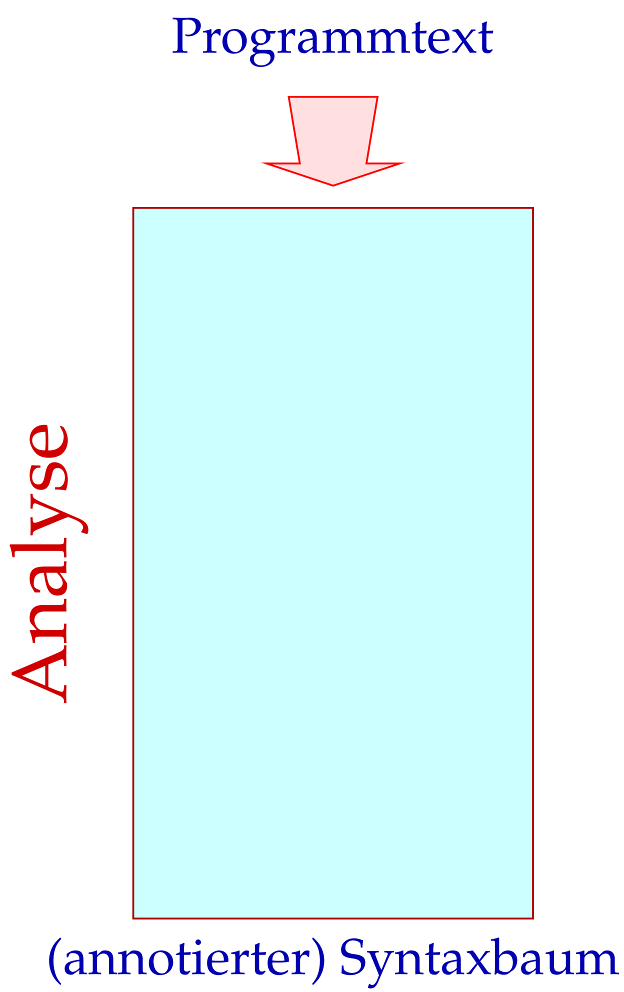
Orientierung:



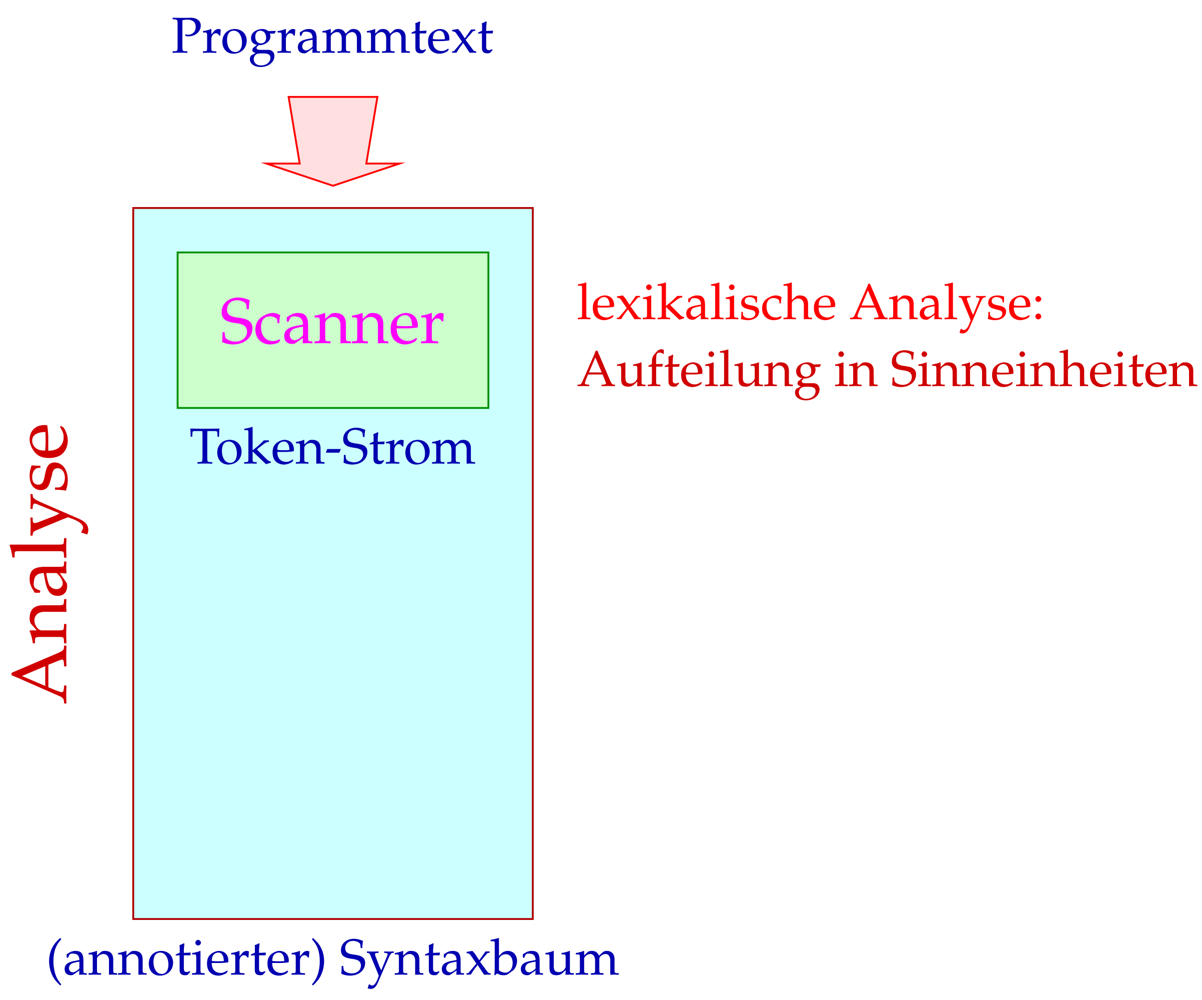
Orientierung:



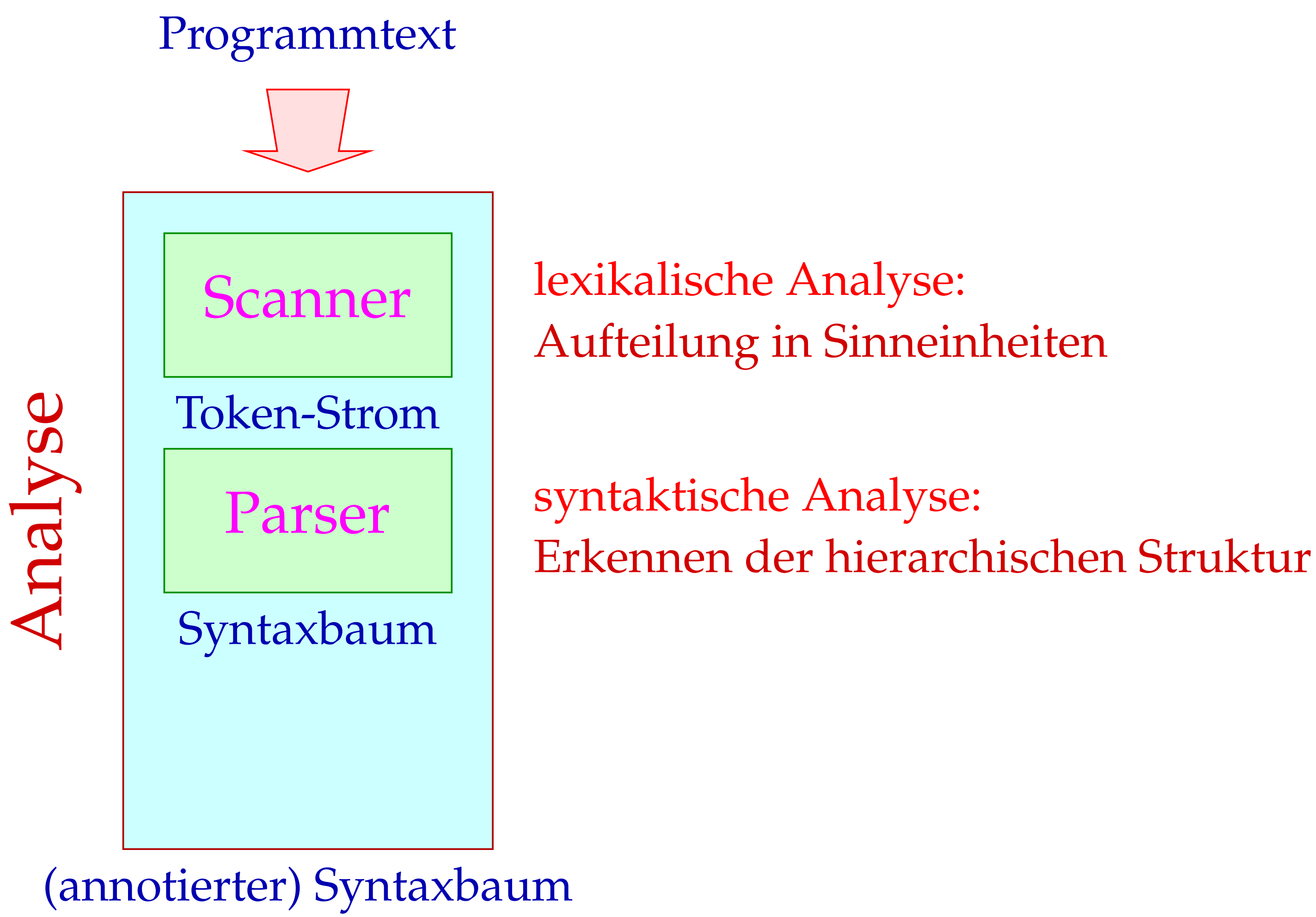
Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die Analyse-Phase :-)



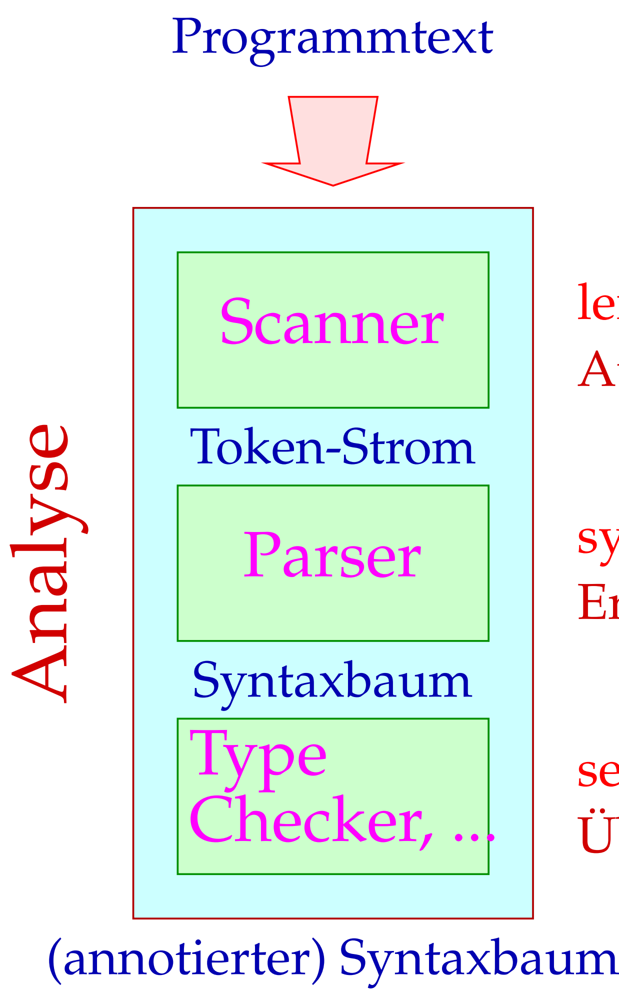
Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die Analyse-Phase :-)



Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die Analyse-Phase :-)



Nachdem wir Prinzipien der Code-Erzeugung kennen gelernt haben, behandeln wir nun die Analyse-Phase :-)

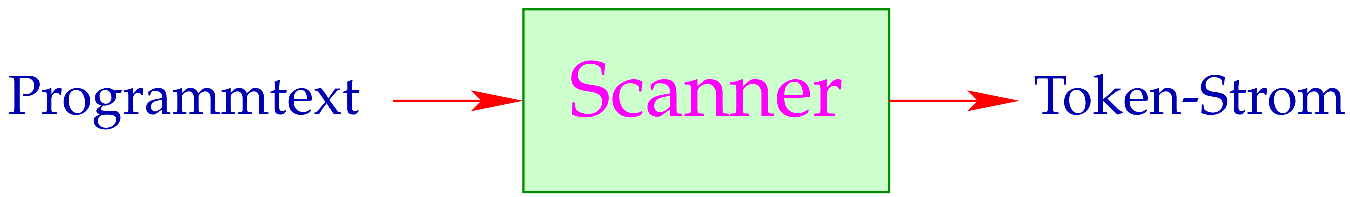


lexikalische Analyse:
Aufteilung in Sinneinheiten

syntaktische Analyse:
Erkennen der hierarchischen Struktur

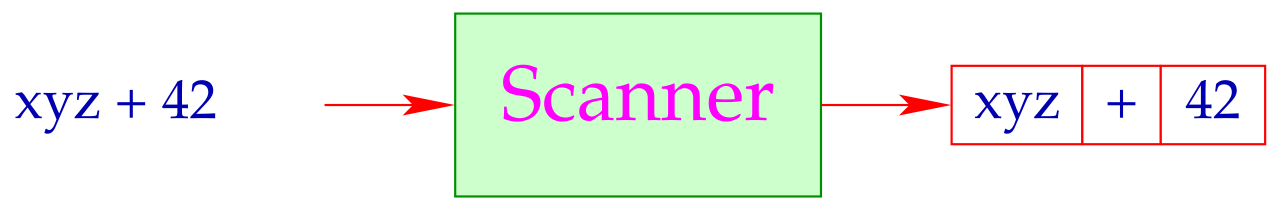
semantische Analyse:
Überprüfung/Ermittlung
semantischer Eigenschaften

1 Die lexikalische Analyse



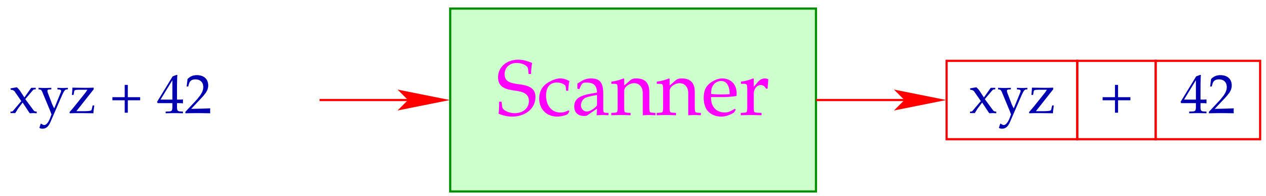
Ein Token ist eine Folge von Zeichen, die zusammen eine Einheit bilden.

1 Die lexikalische Analyse



Ein Token ist eine Folge von Zeichen, die zusammen eine Einheit bilden.

1 Die lexikalische Analyse



- Ein Token ist eine Folge von Zeichen, die zusammen eine Einheit bilden.
- Tokens werden in Klassen zusammen gefasst. Zum Beispiel:

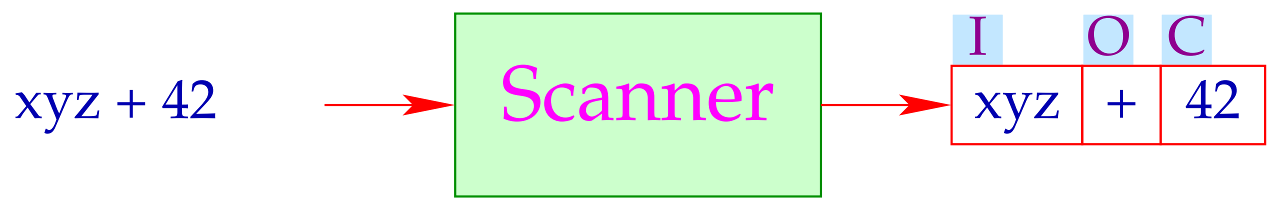
  → Namen (Identifier) wie `xyz`, `pi`, ...

  → Konstanten wie `42`, `3.14`, `"abc"`, ...

  → Operatoren wie `+`, ...

  → reservierte Worte wie `if`, `int`, ...

1 Die lexikalische Analyse



- Ein Token ist eine Folge von Zeichen, die zusammen eine Einheit bilden.
- Tokens werden in Klassen zusammen gefasst. Zum Beispiel:

  → Namen (Identifier) wie `xyz`, `pi`, ...

  → Konstanten wie `42`, `3.14`, `"abc"`, ...

  → Operatoren wie `+`, ...

  → reservierte Worte wie `if`, `int`, ...

Sind Tokens erst einmal klassifiziert, kann man die Teilwörter vorverarbeiten:

- Wegwerfen irrelevanter Teile wie Leerzeichen, Kommentaren,...
- Aussondern von Pragmas, d.h. Direktiven an den Compiler, die nicht Teil des Programms sind, wie include-Anweisungen;
- Ersetzen der Token bestimmter Klassen durch ihre Bedeutung / Interndarstellung, etwa bei:
  - $\rightarrow$ Konstanten;
  - $\rightarrow$ Namen: die typischerweise zentral in einer Symbol-Tabelle verwaltet, evt. mit reservierten Worten verglichen (soweit nicht vom Scanner bereits vorgenommen :-) und gegebenenfalls durch einen Index ersetzt werden.

$\Longrightarrow$ Sieber

Diskussion:

- Scanner und Sieber werden i.a. in einer Komponente zusammen gefasst, indem man dem Scanner nach Erkennen eines Tokens gestattet, eine Aktion auszuführen :-)
- Scanner werden i.a. nicht von Hand programmiert, sondern aus einer Spezifikation generiert:

Vorteile:

Produktivität:

Die Komponente lässt sich schneller herstellen :-)

Korrektheit:

Die Komponente realisiert (beweisbar :-) die Spezifikation.

Effizienz:

Der Generator kann die erzeugte Programmkomponente mit den effizientesten Algorithmen ausstatten.

Vorteile:

Produktivität:

Die Komponente lässt sich schneller herstellen :-)

Korrektheit:

Die Komponente realisiert (beweisbar :-) die Spezifikation.

Effizienz:

Der Generator kann die erzeugte Programmkomponente mit den effizientesten Algorithmen ausstatten.

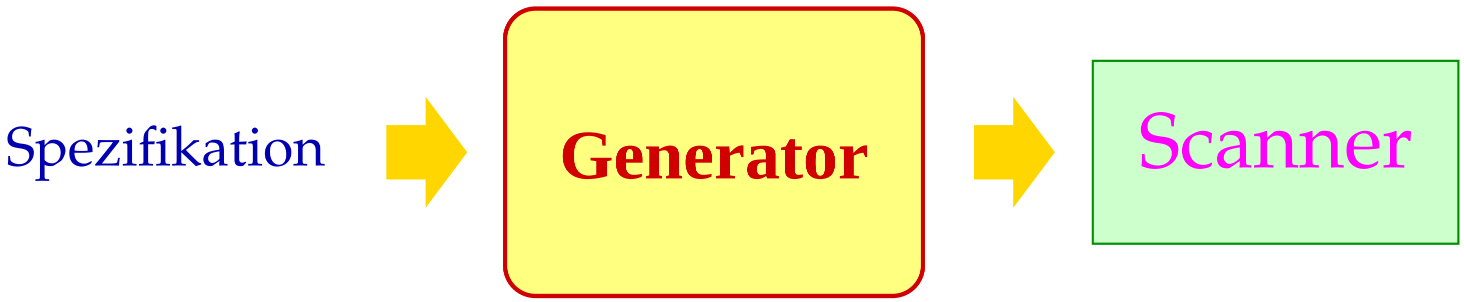
Einschränkungen:

→ Spezifizieren ist auch Programmieren — nur eventuell einfacher :-)

→ Generierung statt Implementierung lohnt sich nur für Routine-Aufgaben

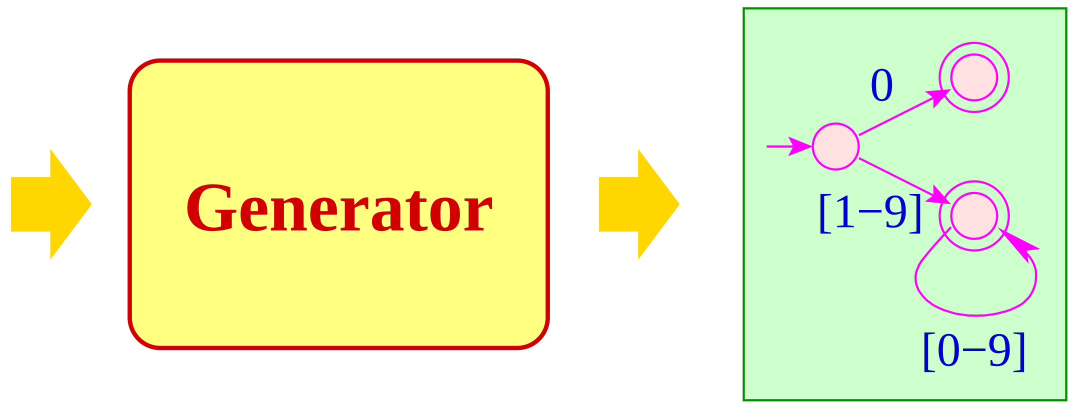
... und ist nur für Probleme möglich, die sehr gut verstanden sind :-(

... in unserem Fall:



... in unserem Fall:

0 | [1-9][0-9]*



Spezifikation von Token-Klassen: Reguläre Ausdrücke;

Generierte Implementierung: Endliche Automaten + X :-)

1.1 Grundlagen: Reguläre Ausdrücke

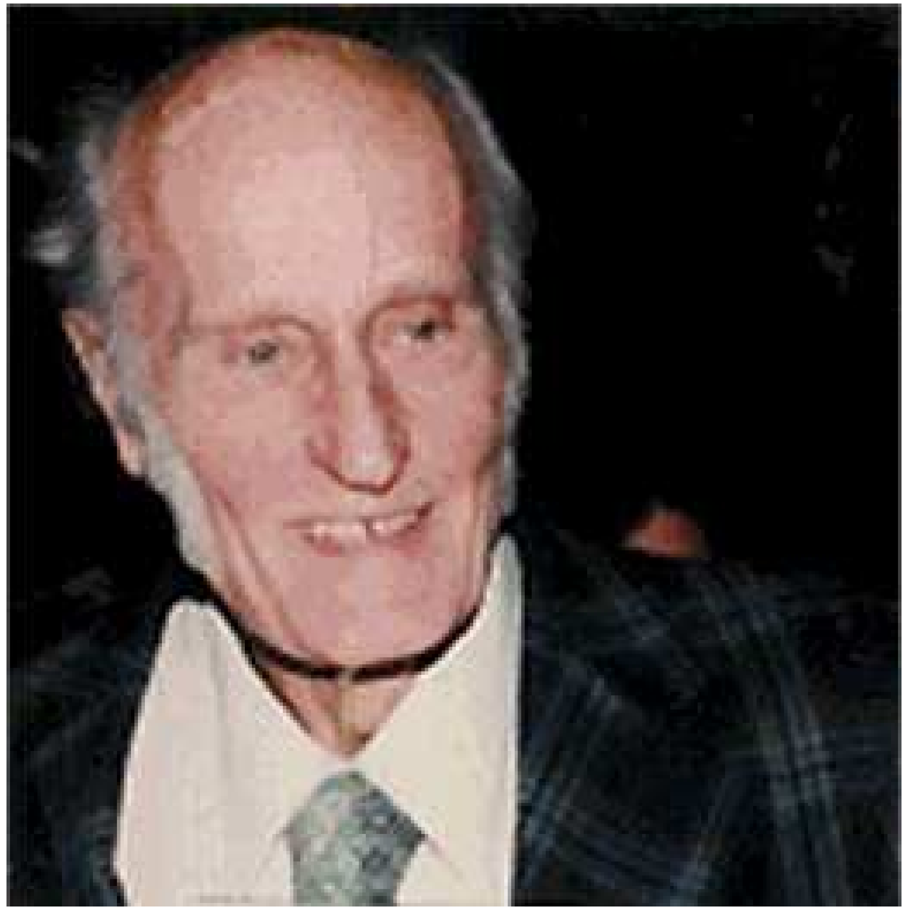
- Programmtext benutzt ein endliches Alphabet Σ von Eingabe-Zeichen, z.B. ASCII :-)
- Die Menge der Textabschnitte einer Token-Klasse ist i.a. regulär.
- Reguläre Sprachen kann man mithilfe regulärer Ausdrücke spezifizieren.

1.1 Grundlagen: Reguläre Ausdrücke

- Programmtext benutzt ein endliches Alphabet Σ von Eingabe-Zeichen, z.B. ASCII :-)
- Die Menge der Textabschnitte einer Token-Klasse ist i.a. regulär.
- Reguläre Sprachen kann man mithilfe regulärer Ausdrücke spezifizieren.

Die Menge $\mathcal{E}_\Sigma$ der (nicht-leeren) regulären Ausdrücke ist die kleinste Menge $\mathcal{E}$ mit:

- $\epsilon \in \mathcal{E}$ (ϵ neues Symbol nicht aus Σ);
- $a \in \mathcal{E}$ für alle $a \in \Sigma$;
- $(e_1 \mid e_2), (e_1 \cdot e_2), e_1{}^* \in \mathcal{E}$ sofern $e_1, e_2 \in \mathcal{E}$.



Stephen Kleene, Madison Wisconsin, 1909-1994

Beispiele:

$$((a \cdot b^*) \cdot a)$$

$$(a \mid b)$$

$$((a \cdot b) \cdot (a \cdot b))$$

Beispiele:

$$((a \cdot b^*) \cdot a)$$

$$(a \mid b)$$

$$((a \cdot b) \cdot (a \cdot b))$$

Achtung:

- Wir unterscheiden zwischen Zeichen $a, 0, |, \ldots$ und Meta-Zeichen $(, |,), \ldots$
- Um (hässliche) Klammern zu sparen, benutzen wir Operator-Präzedenzen:

$$^* > \cdot > |$$

und lassen "$\cdot$" weg :-)

Beispiele:

$$((a \cdot b^*) \cdot a)$$
$$(a \mid b)$$
$$((a \cdot b) \cdot (a \cdot b))$$

Achtung:

- Wir unterscheiden zwischen Zeichen $a, 0, |, \ldots$ und Meta-Zeichen $(, |,), \ldots$
- Um (hässliche) Klammern zu sparen, benutzen wir Operator-Präzedenzen:

  $$* > \cdot > |$$

  und lassen "$\cdot$" weg :-)
- Reale Spezifikations-Sprachen bieten zusätzliche Konstrukte wie:

  $$e? \equiv (\epsilon \mid e)$$
  $$e^+ \equiv (e \cdot e^*)$$

  und verzichten auf "ϵ" :-)

Spezifikationen benötigen eine Semantik :-)

Im Beispiel:

Spezifikation	Semantik
ab^*a	$\{ab^na \mid n \geq 0\}$
$a \mid b$	$\{a, b\}$
$abab$	$\{abab\}$

Für $e \in \mathcal{E}_\Sigma$ definieren wir die spezifizierte Sprache $[\![e]\!] \subseteq \Sigma^*$ induktiv durch:

$$\begin{array}{lll}
[\![\epsilon]\!] & = & \{\epsilon\} \\
[\![a]\!] & = & \{a\} \\
[\![e^*]\!] & = & ([\![e]\!])^* \\
[\![e_1 | e_2]\!] & = & [\![e_1]\!] \cup [\![e_2]\!] \\
[\![e_1 \cdot e_2]\!] & = & [\![e_1]\!] \cdot [\![e_2]\!]
\end{array}$$

Beachte:

- Die Operatoren $(\_)^*, \cup, \cdot$ sind die entsprechenden Operationen auf Wort-Mengen:

$$\begin{aligned} (L)^* &= \{w_1 \ldots w_k \mid k \geq 0, w_i \in L\} \\ L_1 \cdot L_2 &= \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\} \end{aligned}$$

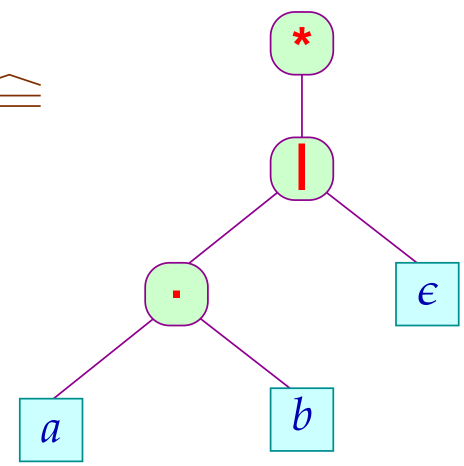
Beachte:

- Die Operatoren $(\_)^*, \cup, \cdot$ sind die entsprechenden Operationen auf Wort-Mengen:

$$\begin{array}{lcl} (L)^* & = & \{w_1 \ldots w_k \mid k \geq 0, w_i \in L\} \\ L_1 \cdot L_2 & = & \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\} \end{array}$$

- Reguläre Ausdrücke stellen wir intern als markierte geordnete Bäume dar:

$(ab|\epsilon)^* \quad \widehat{=}$



Innere Knoten: Operator-Anwendungen;
Blätter: einzelne Zeichen oder ϵ.

Finger-Übung:

Zu jedem regulären Ausdruck e können wir einen Ausdruck e' (evt. mit "?") konstruieren so dass:

- $[\![e]\!] = [\![e']\!]$;
- Falls $[\![e]\!] = \{\epsilon\}$, dann ist $e' \equiv \epsilon$;
- Falls $[\![e]\!] \neq \{\epsilon\}$, dann enthält e' kein "ϵ".

Finger-Übung:

Zu jedem regulären Ausdruck e können wir einen Ausdruck e' (evt. mit “?”) konstruieren so dass:

- $[\![e]\!] = [\![e']\!]$;
- Falls $[\![e]\!] = \{\epsilon\}$, dann ist $e' \equiv \epsilon$;
- Falls $[\![e]\!] \neq \{\epsilon\}$, dann enthält e' kein “ϵ”.

Konstruktion:

Wir definieren eine Transformation $\mathcal{T}$ von regulären Ausdrücken durch:

$$
\begin{array}{lclcl}
\mathcal{T}[\epsilon] & = & \epsilon & & \\
\mathcal{T}[a] & = & a & & \\
\mathcal{T}[e_1|e_2] & = & \mathsf{case}\ (\mathcal{T}[e_1], \mathcal{T}[e_2])\ \mathsf{of} & (\epsilon, \epsilon) : & \epsilon \\
 & & \quad | & (e_1', \epsilon) : & e_1'? \\
 & & \quad | & (\epsilon, e_2') : & e_2'? \\
 & & \quad | & (e_1', e_2') : & (e_1' \mid e_2') \\
\mathcal{T}[e_1 \cdot e_2] & = & \mathsf{case}\ (\mathcal{T}[e_1], \mathcal{T}[e_2])\ \mathsf{of} & (\epsilon, \epsilon) : & \epsilon \\
 & & \quad | & (e_1', \epsilon) : & e_1' \\
 & & \quad | & (\epsilon, e_2') : & e_2' \\
 & & \quad | & (e_1', e_2') : & (e_1' \cdot e_2') \\
\mathcal{T}[e^*] & = & \mathsf{case}\ \mathcal{T}[e]\ \mathsf{of}\quad \epsilon : \ \epsilon & & \\
 & & \quad |\quad e_1 : \ {e_1}^* & & \\
\mathcal{T}[e?] & = & \mathsf{case}\ \mathcal{T}[e]\ \mathsf{of}\quad \epsilon : \ \epsilon & & \\
 & & \quad |\quad e_1 : \ e_1? & &
\end{array}
$$

Unsere Anwendung:

Identifier in Java:

```
le = [a-zA-Z_\$]
di = [0-9]
Id = {le} ({le} | {di})*
```

Unsere Anwendung:

Identifier in Java:

```
le = [a-zA-Z_\$]
di = [0-9]
Id = {le} ({le} | {di})*
```

Bemerkungen:

- "`le`" und "`di`" sind Zeichenklassen.
- Definierte Namen werden in "`{`", "`}`" eingeschlossen.
- Zeichen werden von Meta-Zeichen durch "\" unterschieden.

Unsere Anwendung:

Identifier in Java:

```
le = [a-zA-Z_\$]
di = [0-9]
Id = {le} ({le}|{di})*
```

Gleitkommazahlen:

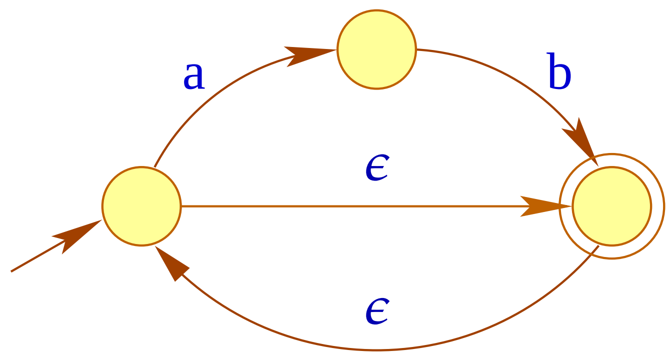
```
Float = {di}* (\.{di}|{di}\.) {di}*((e|E)(\+|\-)?{di}+)?
```

Bemerkungen:

- "`le`" und "`di`" sind Zeichenklassen.
- Definierte Namen werden in "`{`", "`}`" eingeschlossen.
- Zeichen werden von Meta-Zeichen durch "`\`" unterschieden.

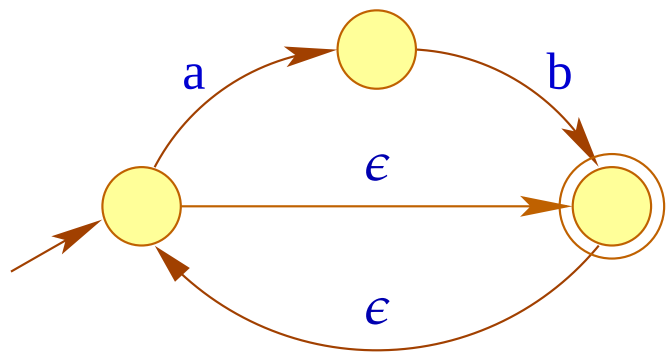
1.2 Grundlagen: Endliche Automaten

Beispiel:



1.2 Grundlagen: Endliche Automaten

Beispiel:



Knoten: Zustände;

Kanten: Übergänge;

Beschriftungen: konsumierter Input :-)

Michael O. Rabin, Stanford University

Dana S. Scott, Carnegy Mellon University, Pittsburgh

Formal ist ein nicht-deterministischer endlicher Automat mit ϵ-Übergängen (ϵ-NFA) ein Tupel $A = (Q, \Sigma, \delta, I, F)$ wobei:

Q	eine endliche Menge von Zuständen;
Σ	ein endliches Eingabe-Alphabet;
$I \subseteq Q$	die Menge der Anfangszustände;
$F \subseteq Q$	die Menge der Endzustände und
δ	die Menge der Übergänge (die Übergangs-Relation) ist.

Formal ist ein nicht-deterministischer endlicher Automat mit ϵ-Übergängen (ϵ-NFA) ein Tupel $\quad A = (Q, \Sigma, \delta, I, F) \quad$ wobei:

Q	eine endliche Menge von Zuständen;
Σ	ein endliches Eingabe-Alphabet;
$I \subseteq Q$	die Menge der Anfangszustände;
$F \subseteq Q$	die Menge der Endzustände und
δ	die Menge der Übergänge (die Übergangs-Relation) ist.

Für ϵ-NFAs ist:

$$\delta \quad \subseteq \quad Q \times (\Sigma \cup \{\epsilon\}) \times Q$$

Formal ist ein nicht-deterministischer endlicher Automat mit ϵ-Übergängen (ϵ-NFA) ein Tupel $A = (Q, \Sigma, \delta, I, F)$ wobei:

Q	eine endliche Menge von Zuständen;
Σ	ein endliches Eingabe-Alphabet;
$I \subseteq Q$	die Menge der Anfangszustände;
$F \subseteq Q$	die Menge der Endzustände und
δ	die Menge der Übergänge (die Übergangs-Relation) ist.

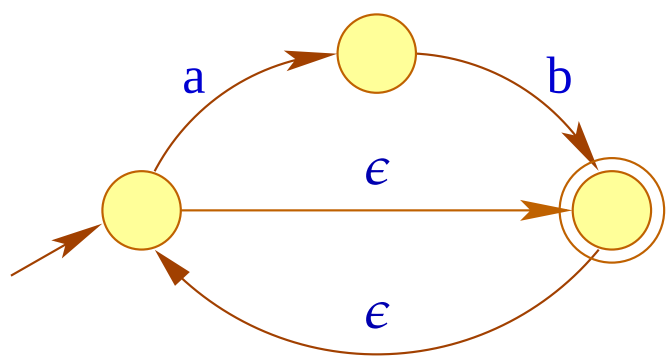
Für ϵ-NFAs ist:

$$\delta \quad \subseteq \quad Q \times (\Sigma \cup \{\epsilon\}) \times Q$$

- Gibt es keine ϵ-Übergänge (p, ϵ, q), ist A ein NFA.
- Ist $\delta : Q \times \Sigma \to Q$ eine Funktion und $\#I = 1$, heißt A deterministisch (DFA).

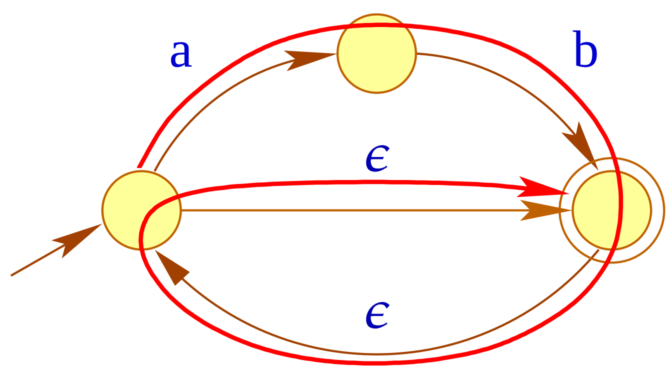
Akzeptierung

- Berechnungen sind Pfade im Graphen.
- akzeptierende Berechnungen führen von I nach F .
- Ein akzeptiertes Wort ist die Beschriftung eines akzeptierenden Pfades ...



Akzeptierung

- Berechnungen sind Pfade im Graphen.
- akzeptierende Berechnungen führen von I nach F .
- Ein akzeptiertes Wort ist die Beschriftung eines akzeptierenden Pfades ...



- Dazu definieren wir den transitiven Abschluss δ^* von δ als kleinste Menge δ' mit:

$$\begin{array}{l} (p, \epsilon, p) \in \delta' \quad \text{und} \\ (p, xw, q) \in \delta' \quad \text{sofern} \quad (p, x, p_1) \in \delta \quad \text{und} \quad (p_1, w, q) \in \delta'. \end{array}$$

δ^* beschreibt für je zwei Zustände, mit welchen Wörtern man vom einen zum andern kommt :-)

- Die Menge aller akzeptierten Worte, d.h. die von A akzeptierte Sprache können wir kurz beschreiben als:

$$\mathcal{L}(A) = \{w \in \Sigma^* \mid \exists\, i \in I, f \in F : \ (i, w, f) \in \delta^*\}$$

Satz:

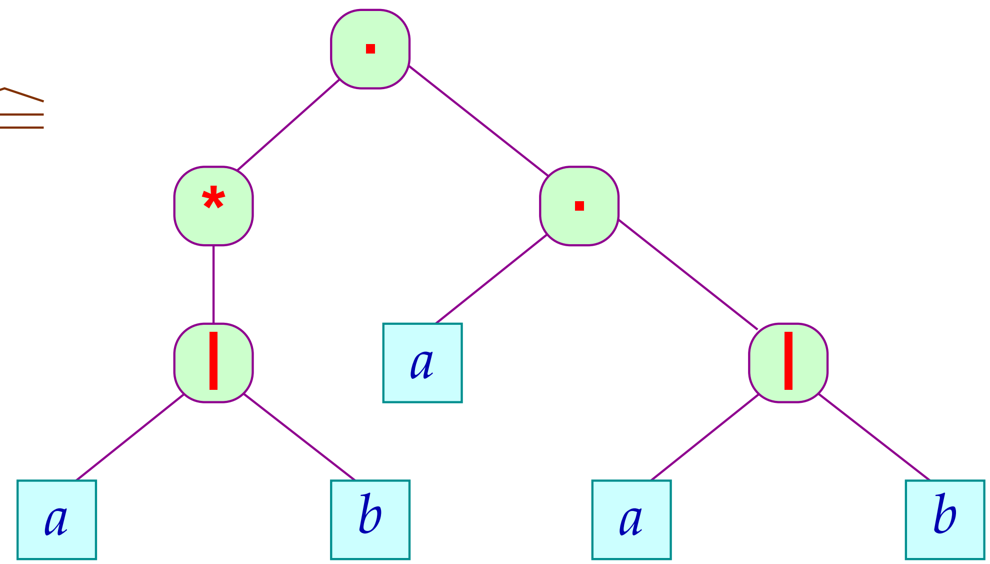
Für jeden regulären Ausdruck e kann (in linearer Zeit :-) ein ϵ-NFA konstruiert werden, der die Sprache $[\![e]\!]$ akzeptiert.

Idee:

Der Automat verfolgt (konzepionell mithilfe einer Marke "$\bullet$"), wohin man in e mit der Eingabe w gelangen kann.

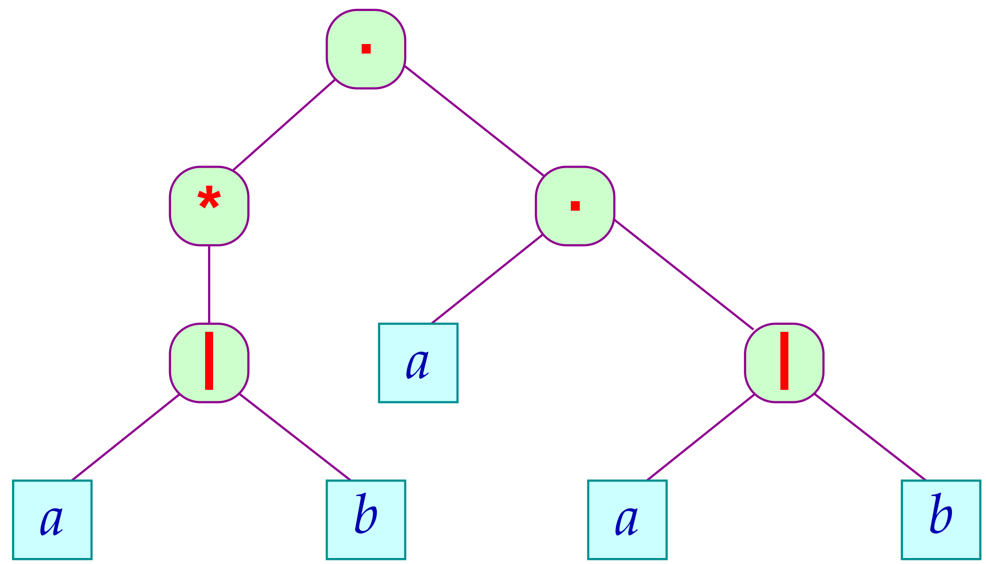
Beispiel:

$(a|b)^* a (a|b) \quad \widehat{=}$



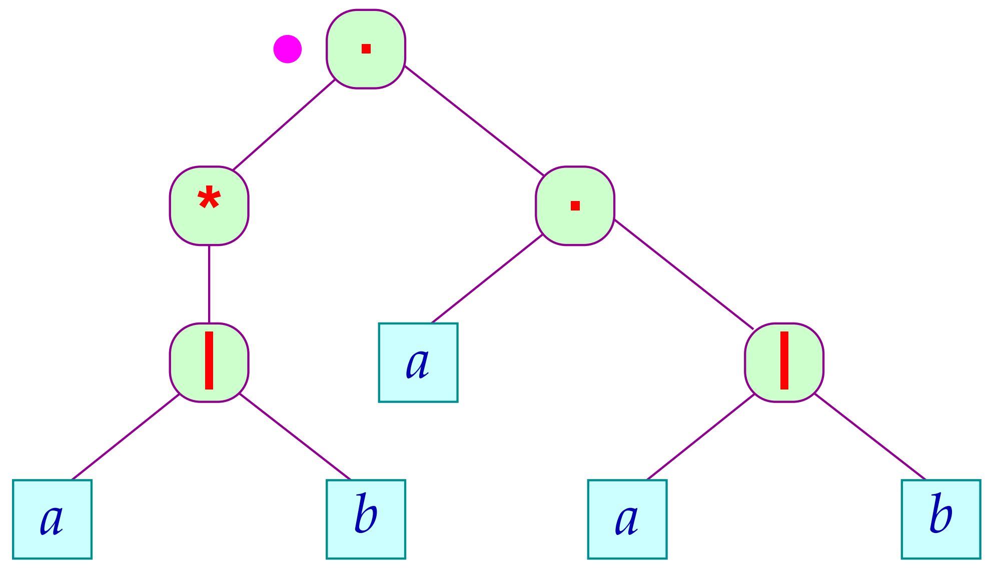
Beispiel:

$$w = bbaa \quad :$$



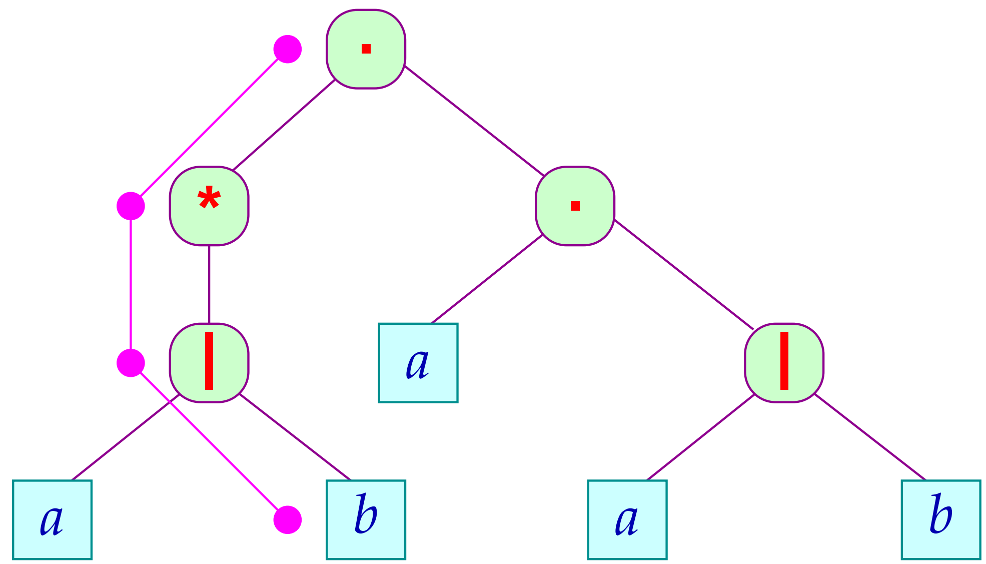
Beispiel:

$w \quad = \quad bbaa \quad :$



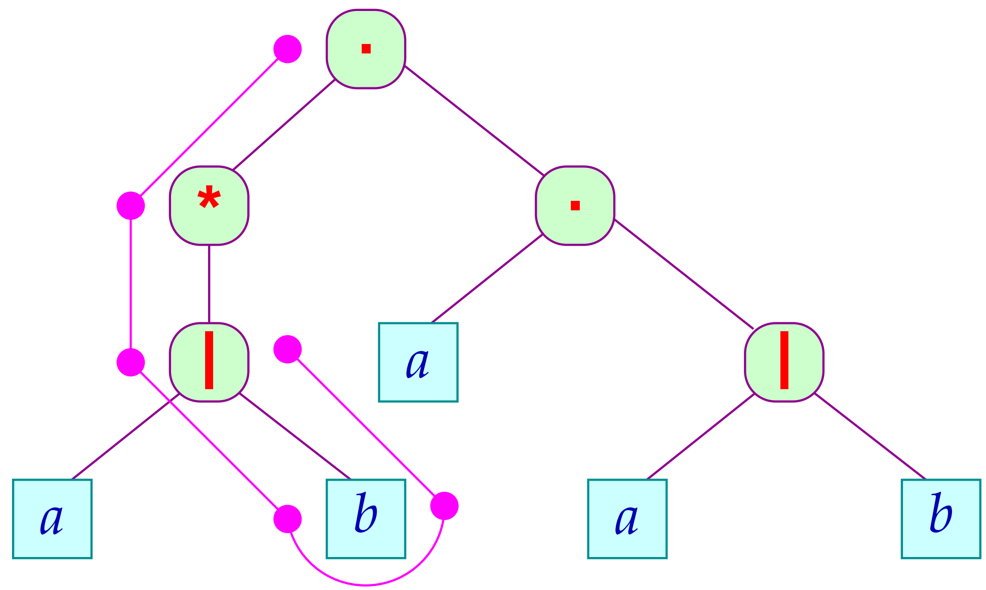
Beispiel:

$$w \quad = \quad bbaa \quad :$$



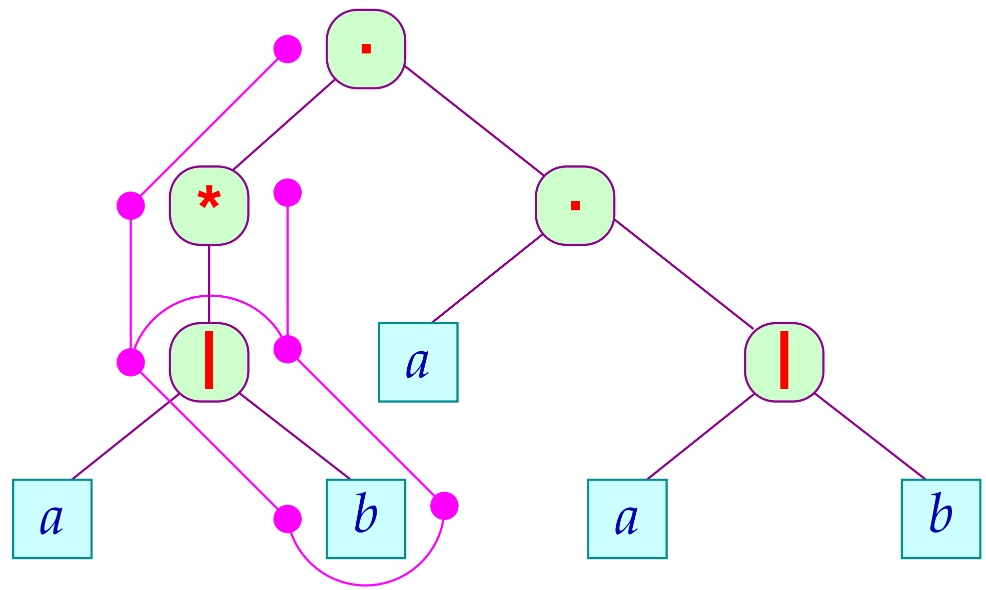
Beispiel:

$$w = bbaa \quad :$$



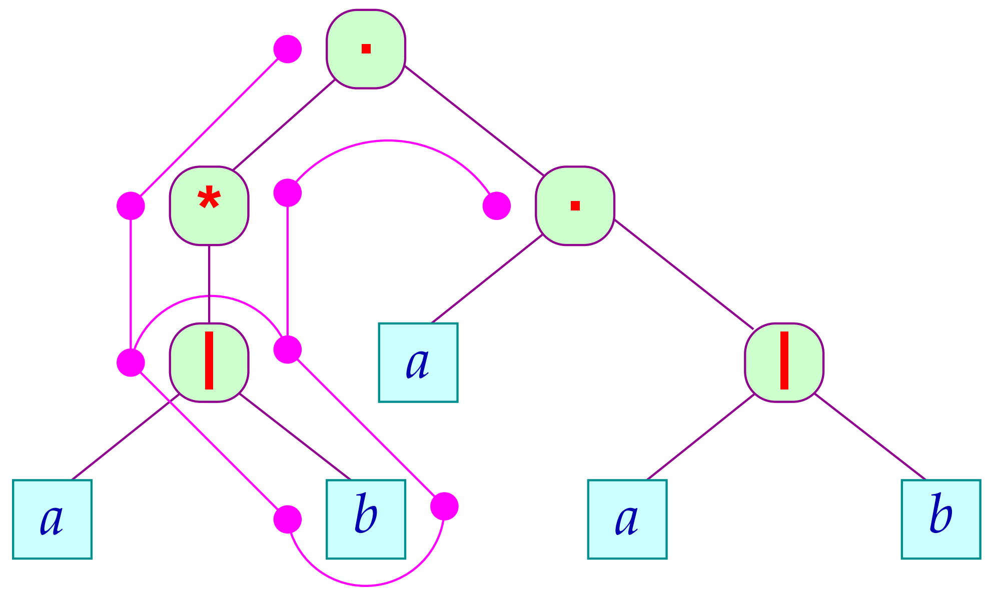
Beispiel:

$$w = bbaa \quad :$$



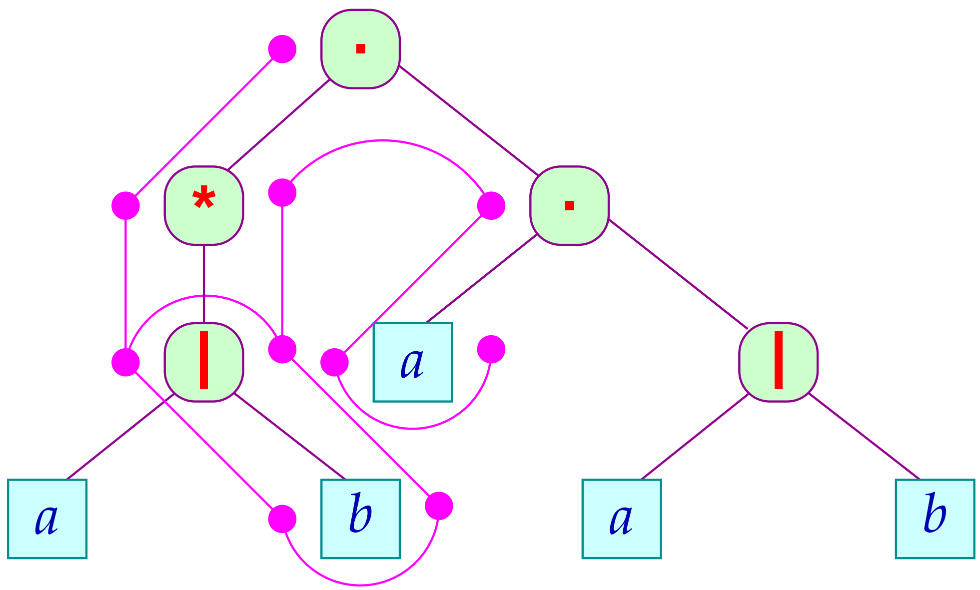
Beispiel:

$$w \quad = \quad bbaa \quad :$$



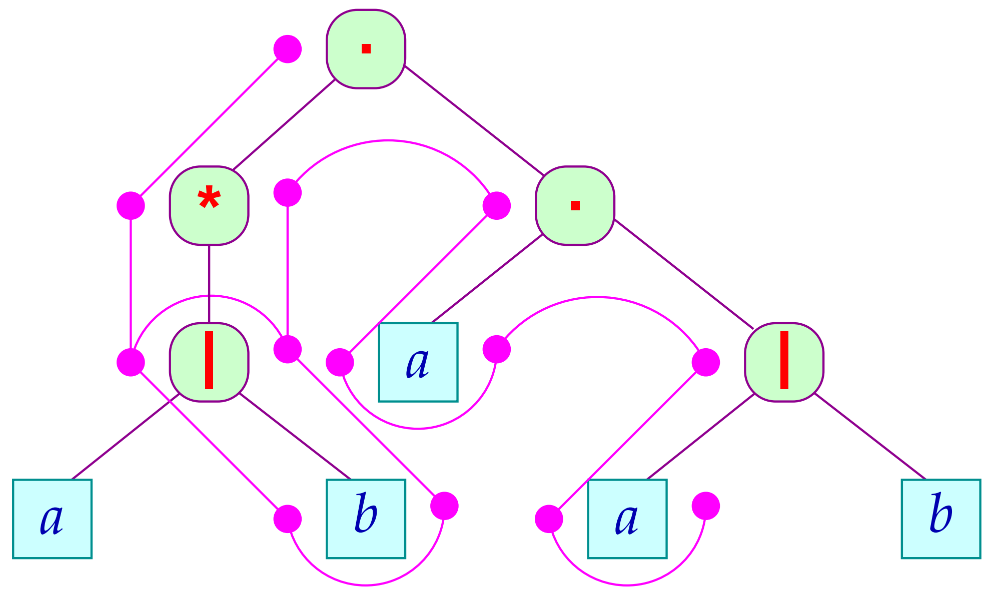
Beispiel:

$$w = bbaa \quad :$$



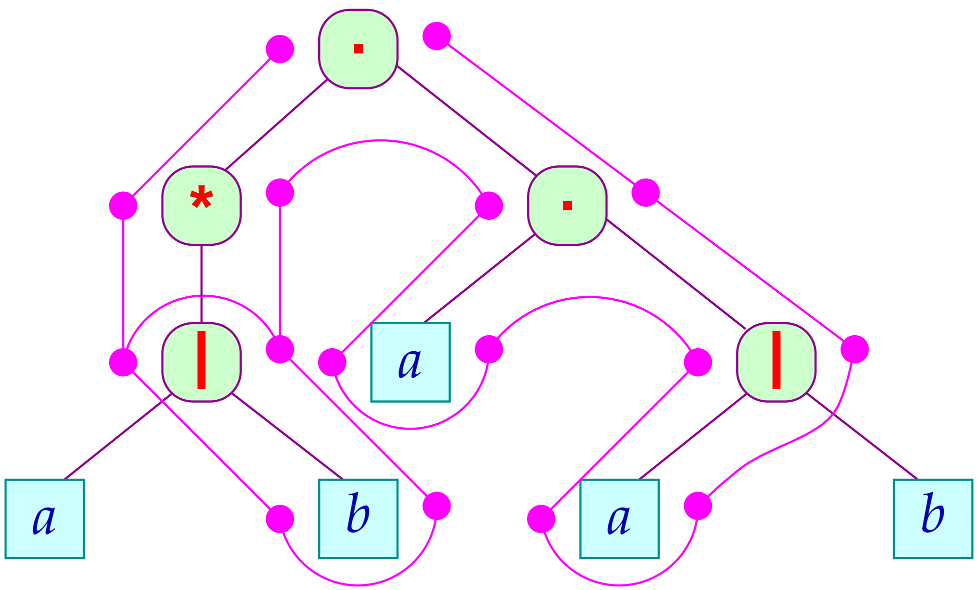
Beispiel:

$$w \quad = \quad bbaa \quad :$$



Beispiel:

$$w \quad = \quad bbaa \quad :$$

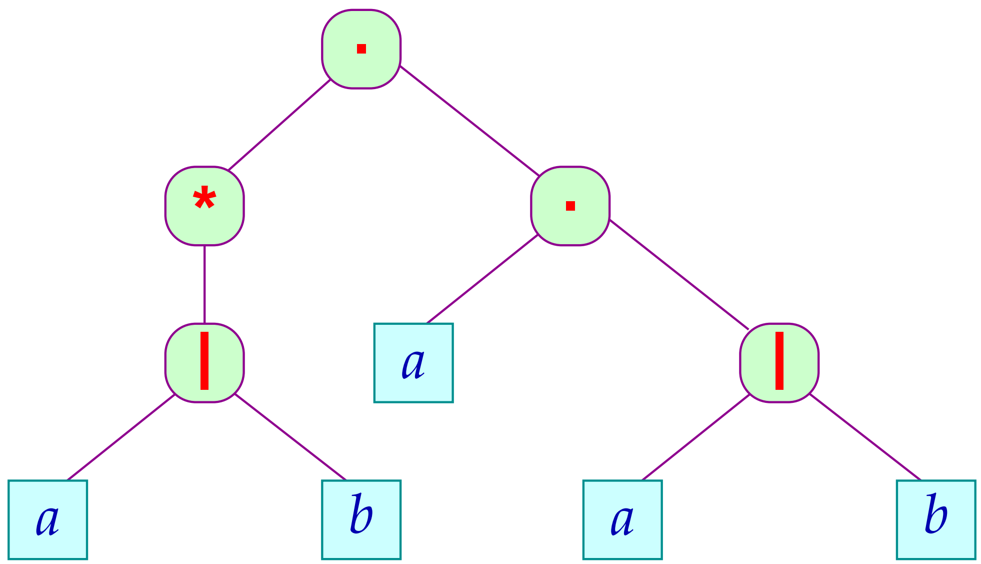


Beachte:

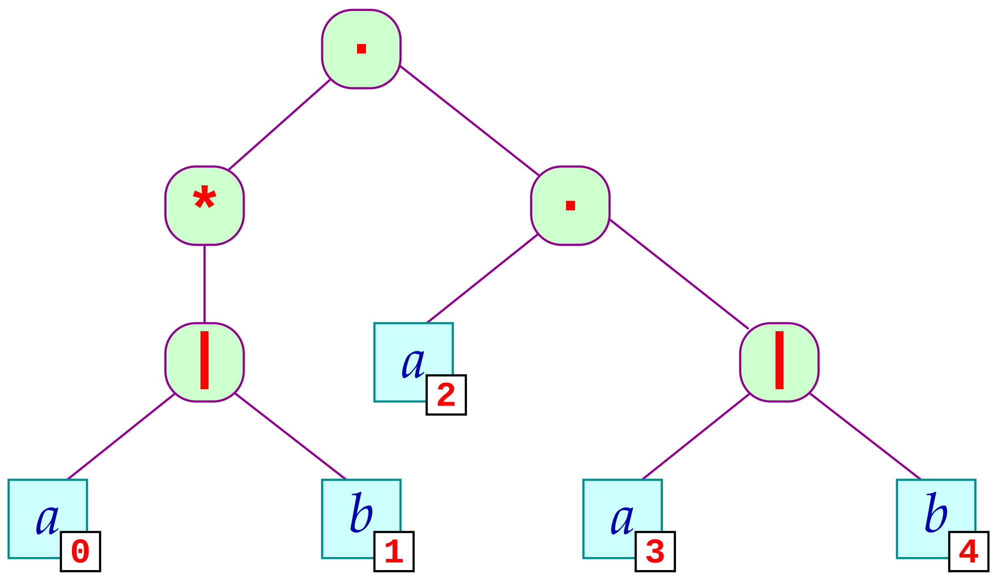
- Gelesen wird nur an den Blättern.
- Die Navigation im Baum erfolgt ohne Lesen, d.h. mit ϵ-Übergängen.
- Für eine formale Konstruktion müssen wir die Knoten im Baum bezeichnen.
- Dazu benutzen wir (hier) einfach den dargestellten Teilausdruck :-)
- Leider gibt es eventuell mehrere gleiche Teilausdrücke :-(

$\Longrightarrow$ Wir numerieren die Blätter durch ...

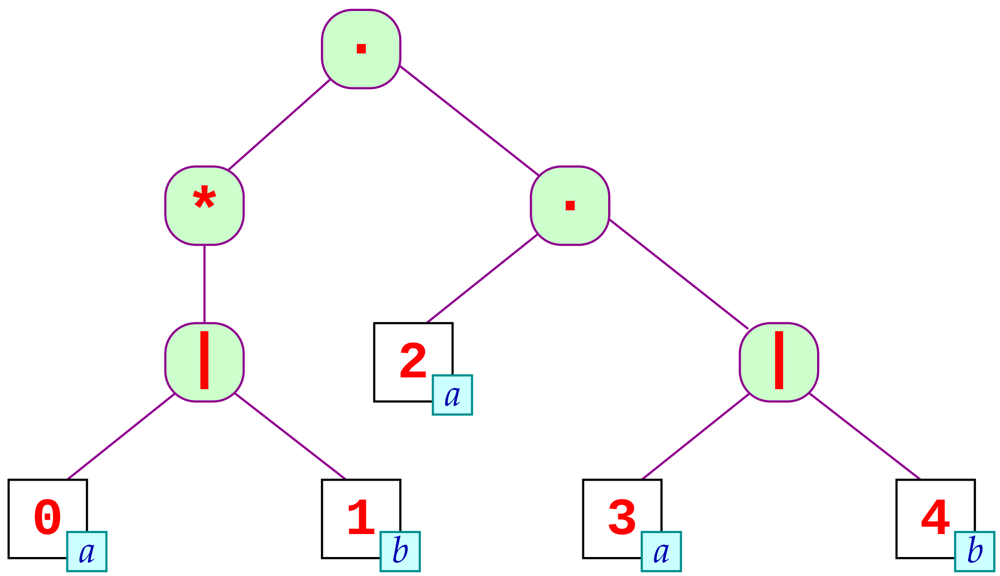
... im Beispiel:



... im Beispiel:



... im Beispiel:



Die Konstruktion:

Zustände: $\bullet r$, $r\bullet$ $\quad r$ Knoten von e;

Anfangszustand: $\bullet e$;

Endzustand: $e\bullet$;

Übergangsrelation:

Für Blätter $r \equiv \boxed{i \mid x}$ benötigen wir: $(\bullet r, x, r\bullet)$.

Die übrigen Übergänge sind:

r	Übergänge
$r_1 \mid r_2$	$(\bullet r, \epsilon, \bullet r_1)$
	$(\bullet r, \epsilon, \bullet r_2)$
	$(r_1 \bullet, \epsilon, r \bullet)$
	$(r_2 \bullet, \epsilon, r \bullet)$
$r_1 \cdot r_2$	$(\bullet r, \epsilon, \bullet r_1)$
	$(r_1 \bullet, \epsilon, \bullet r_2)$
	$(r_2 \bullet, \epsilon, r \bullet)$

r	Übergänge
r_1^*	$(\bullet r, \epsilon, r \bullet)$
	$(\bullet r, \epsilon, \bullet r_1)$
	$(r_1 \bullet, \epsilon, \bullet r_1)$
	$(r_1 \bullet, \epsilon, r \bullet)$
$r_1?$	$(\bullet r, \epsilon, r \bullet)$
	$(\bullet r, \epsilon, \bullet r_1)$
	$(r_1 \bullet, \epsilon, r \bullet)$

Diskussion:

- Die meisten Übergänge dienen dazu, im Ausdruck zu navigieren :-(
- Der Automat ist i.a. nichtdeterministisch :-(

Diskussion:

- Die meisten Übergänge dienen dazu, im Ausdruck zu navigieren :-(
- Der Automat ist i.a. nichtdeterministisch :-(

$\Longrightarrow$

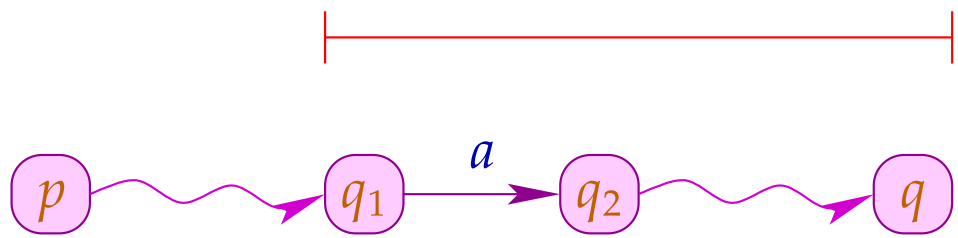
Strategie:

(1) Beseitigung der ϵ-Übergänge;

(2) Beseitigung des Nichtdeterminismus :-)

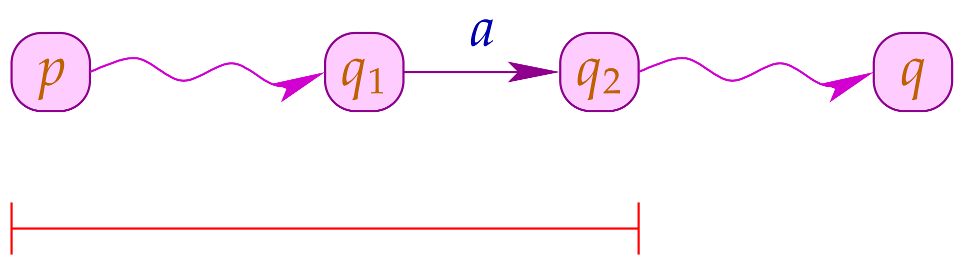
Beseitigung von ϵ-Übergängen:

Zwei einfache Ansätze:



Beseitigung von ϵ-Übergängen:

Zwei einfache Ansätze:

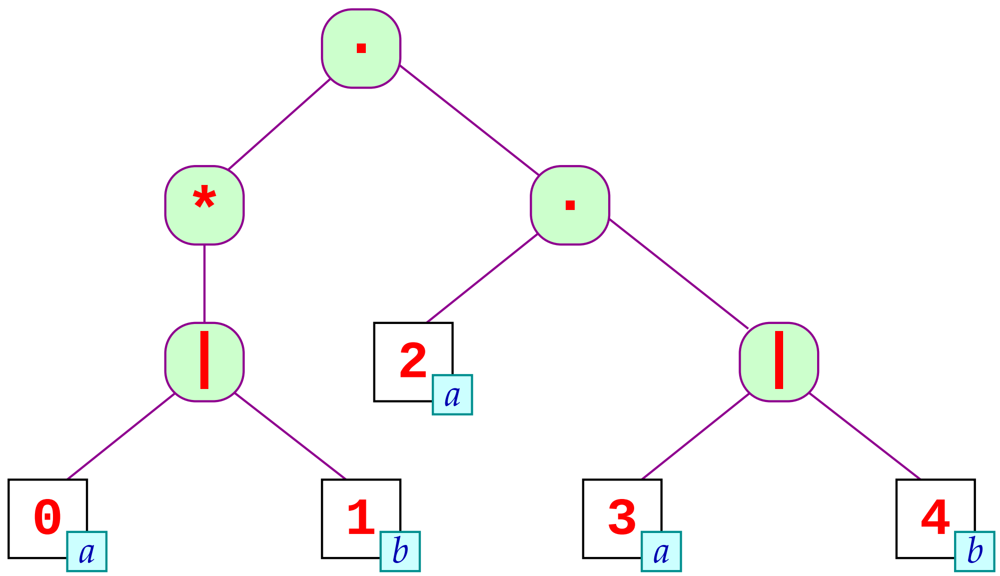


Wir benutzen hier den zweiten Ansatz.

Zur Konstruktion von Parsern werden wir später den ersten benutzen :-)

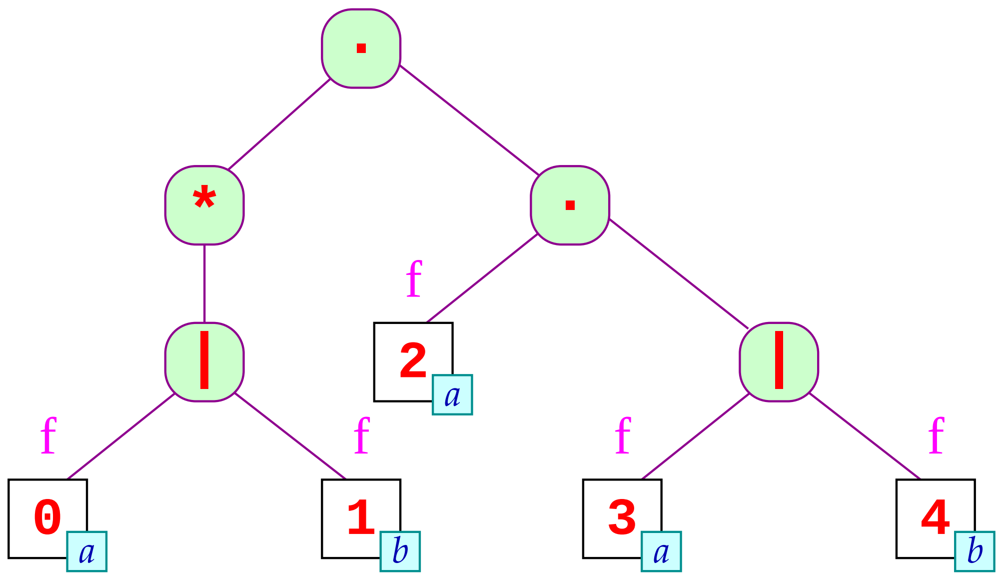
1. Schritt: $\text{empty}[r] = t$ gdw. $\epsilon \in [\![r]\!]$

... im Beispiel:



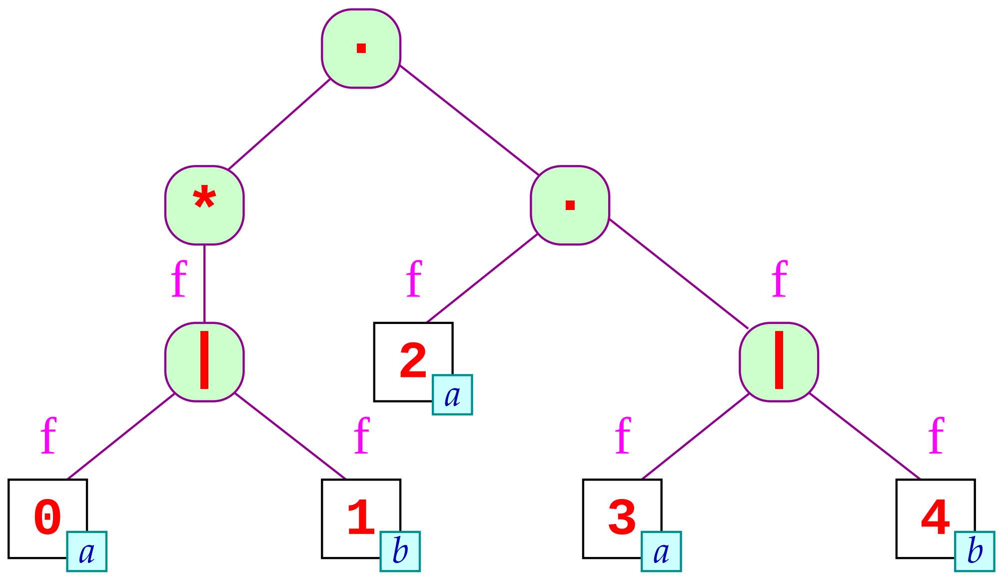
1. Schritt: $\qquad \mathsf{empty}[r] = t \quad \text{gdw.} \quad \epsilon \in [\![r]\!]$

... im Beispiel:



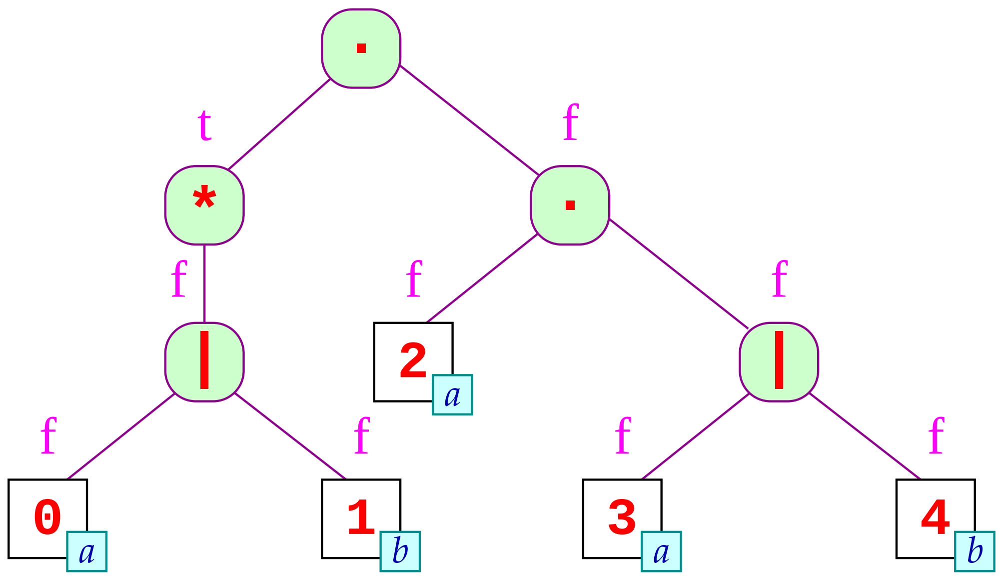
1. Schritt: $\qquad \mathsf{empty}[r] = t \quad \text{gdw.} \quad \epsilon \in [\![r]\!]$

... im Beispiel:



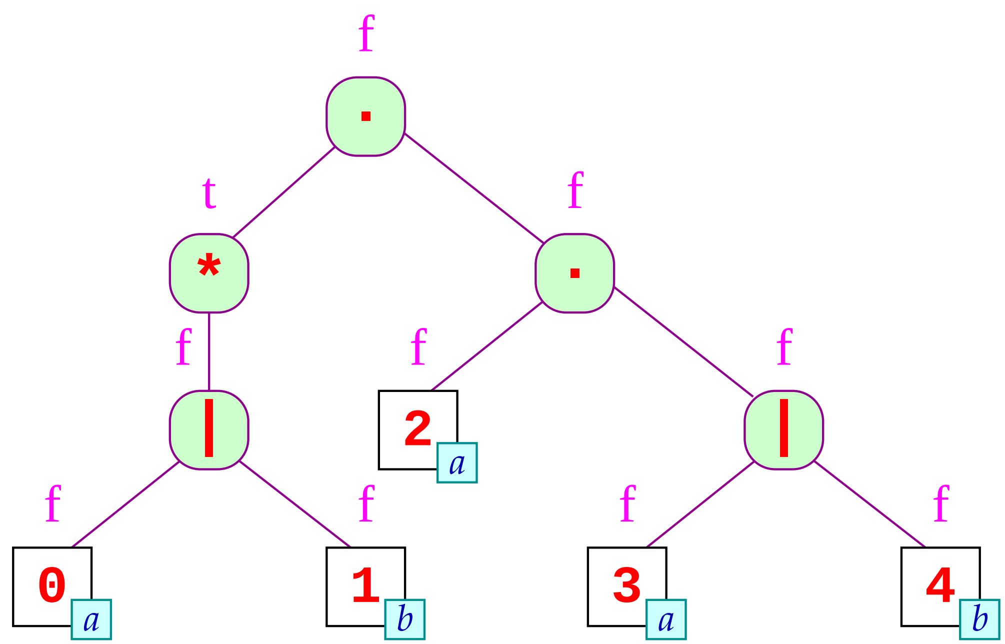
1. Schritt: $\mathsf{empty}[r] = t \quad \text{gdw.} \quad \epsilon \in [\![r]\!]$

... im Beispiel:



1. Schritt: $\quad \mathsf{empty}[r] = t \quad \text{gdw.} \quad \epsilon \in [\![r]\!]$

... im Beispiel:



Implementierung: DFS post-order Traversierung

Für Blätter $r \equiv \boxed{i \mid x}$ ist $\text{empty}[r] = (x \equiv \epsilon)$.

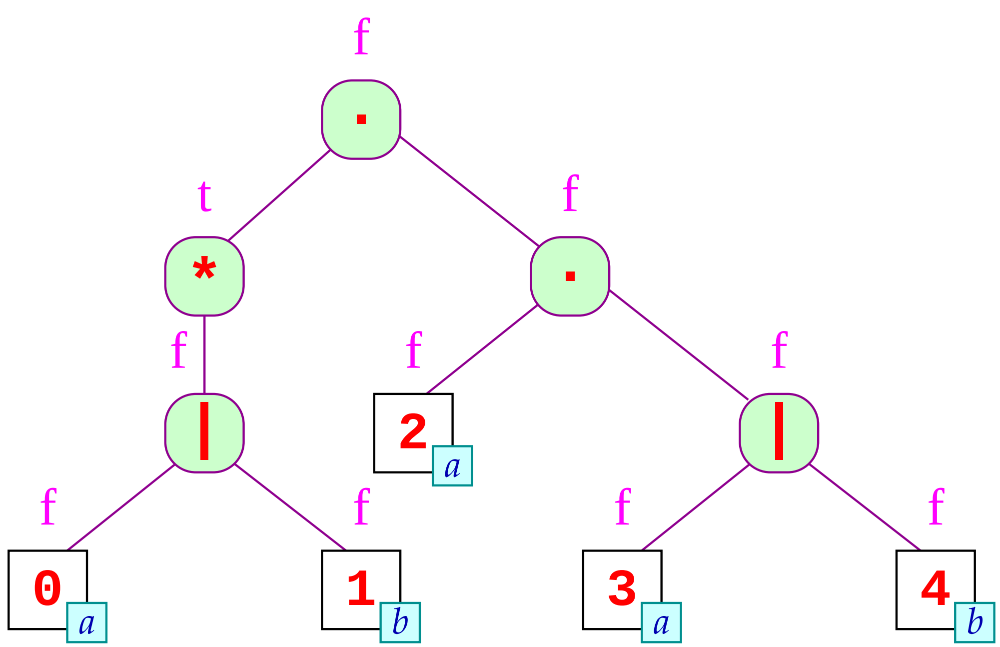
Andernfalls:

$$
\begin{array}{lll}
\text{empty}[r_1 \mid r_2] & = & \text{empty}[r_1] \vee \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] & = & \text{empty}[r_1] \wedge \text{empty}[r_2] \\
\text{empty}[r_1^*] & = & t \\
\text{empty}[r_1?] & = & t
\end{array}
$$

2. Schritt:

Die Menge erster Bätter: $\mathsf{first}[r] = \{i \text{ in } r \mid (\bullet r, \epsilon, \bullet\boxed{i \mid x}) \in \delta^*\}$

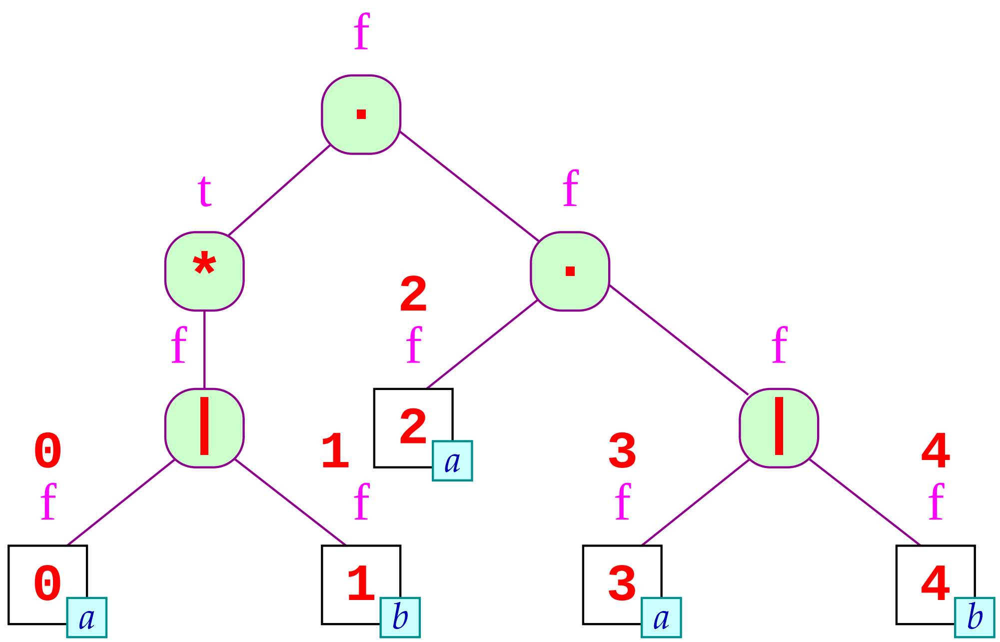
... im Beispiel:



2. Schritt:

Die Menge erster Bätter: $\text{first}[r] = \{i \text{ in } r \mid (\bullet r, \epsilon, \bullet\boxed{i \mid x}) \in \delta^*\}$

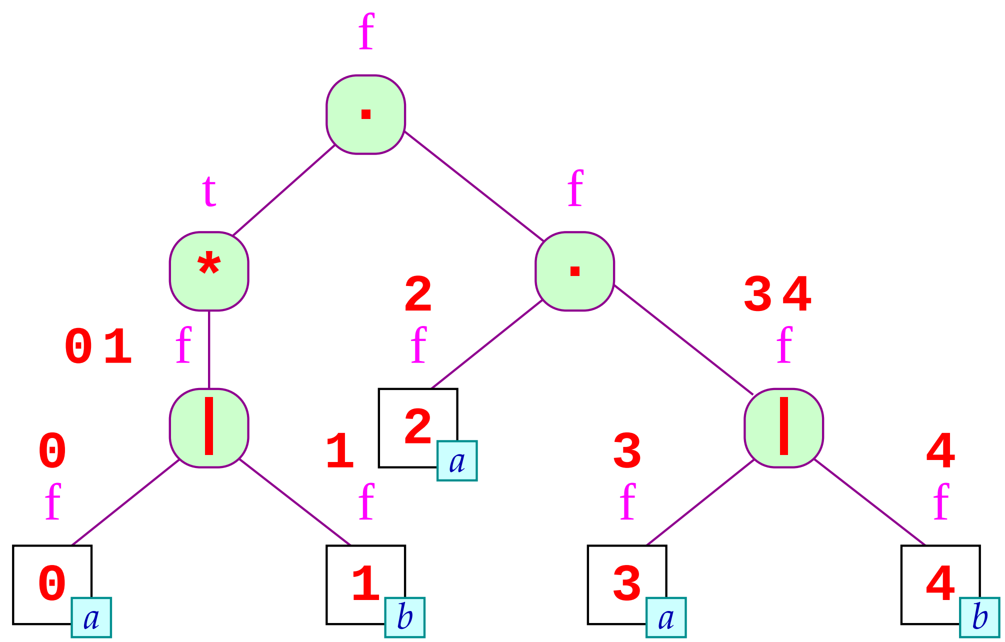
... im Beispiel:



2. Schritt:

Die Menge erster Bätter: $\mathsf{first}[r] = \{i \text{ in } r \mid (\bullet r, \epsilon, \bullet\boxed{i \mid x}) \in \delta^*\}$

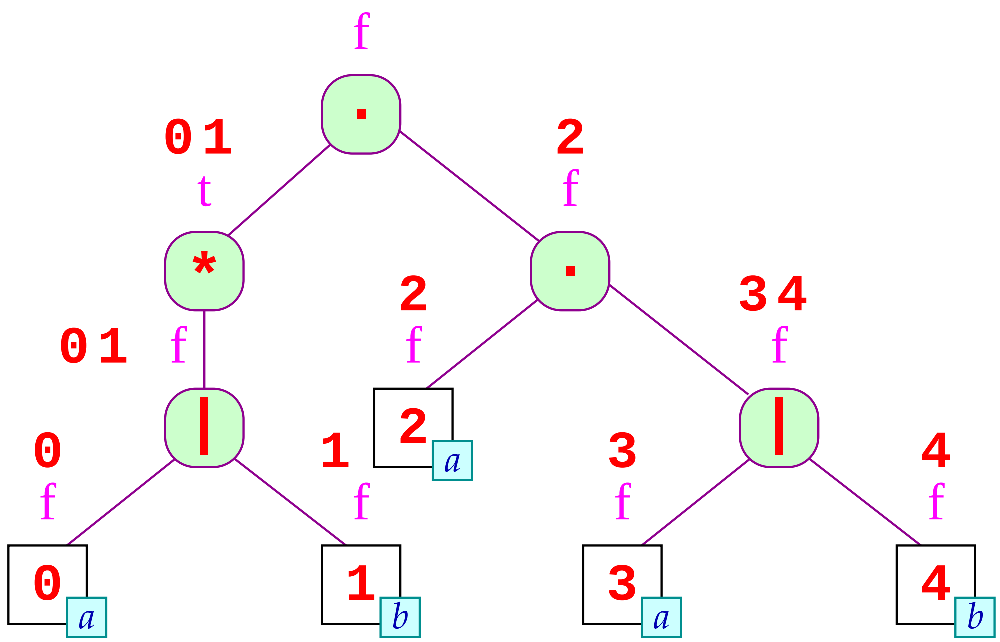
... im Beispiel:



2. Schritt:

Die Menge erster Bätter: $\textsf{first}[r] = \{i \text{ in } r \mid (\bullet r, \epsilon, \bullet\boxed{i \mid x}) \in \delta^*\}$

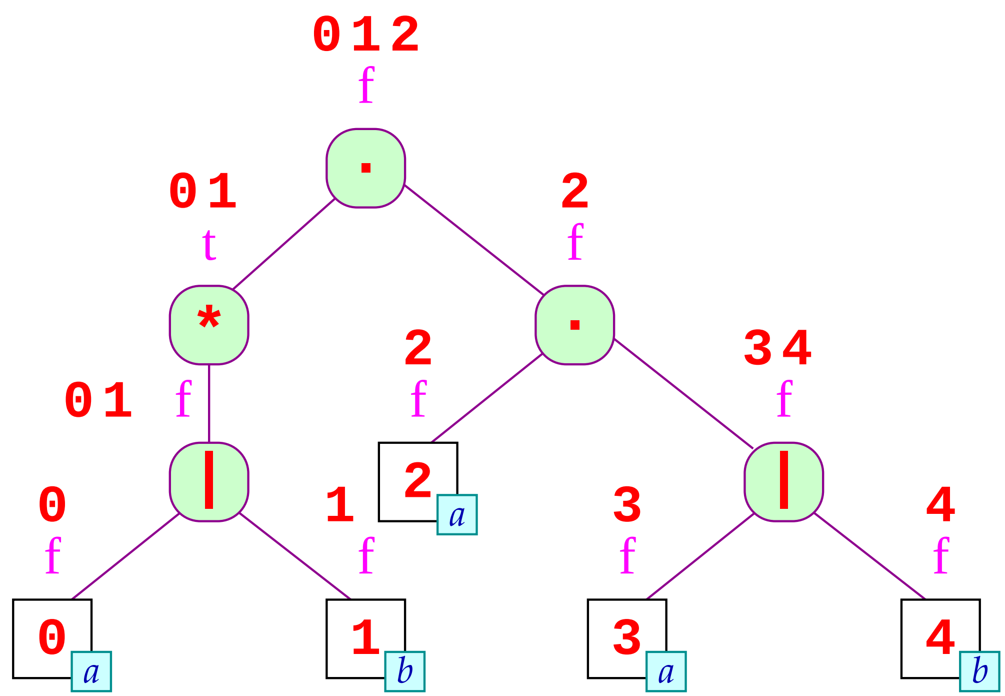
... im Beispiel:



2. Schritt:

Die Menge erster Bätter: $\textsf{first}[r] = \{i \text{ in } r \mid (\bullet r, \epsilon, \bullet\boxed{i \mid x}) \in \delta^*, x \neq \epsilon\}$

... im Beispiel:



Implementierung: DFS post-order Traversierung

Für Blätter $r \equiv \boxed{i \mid x}$ ist $\mathsf{first}[r] = \{i \mid x \not\equiv \epsilon\}$.

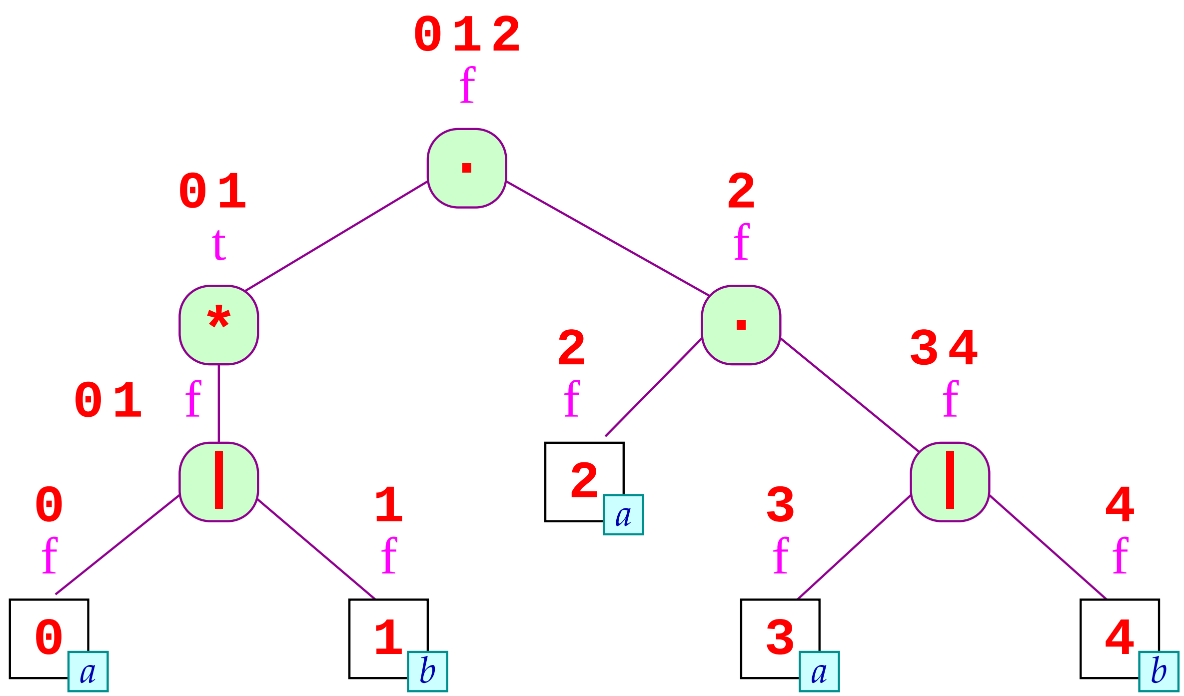
Andernfalls:

$$
\begin{array}{lll}
\mathsf{first}[r_1 \mid r_2] & = & \mathsf{first}[r_1] \cup \mathsf{first}[r_2] \\
\mathsf{first}[r_1 \cdot r_2] & = & \begin{cases} \mathsf{first}[r_1] \cup \mathsf{first}[r_2] & \text{falls} \quad \mathsf{empty}[r_1] = t \\ \mathsf{first}[r_1] & \text{falls} \quad \mathsf{empty}[r_1] = f \end{cases} \\
\mathsf{first}[r_1^*] & = & \mathsf{first}[r_1] \\
\mathsf{first}[r_1?] & = & \mathsf{first}[r_1]
\end{array}
$$

3. Schritt:

Die Menge nächster Bätter: $\textsf{next}[r] = \{i \mid (r\bullet, \epsilon, \bullet\boxed{i \mid x}) \in \delta^*\}$

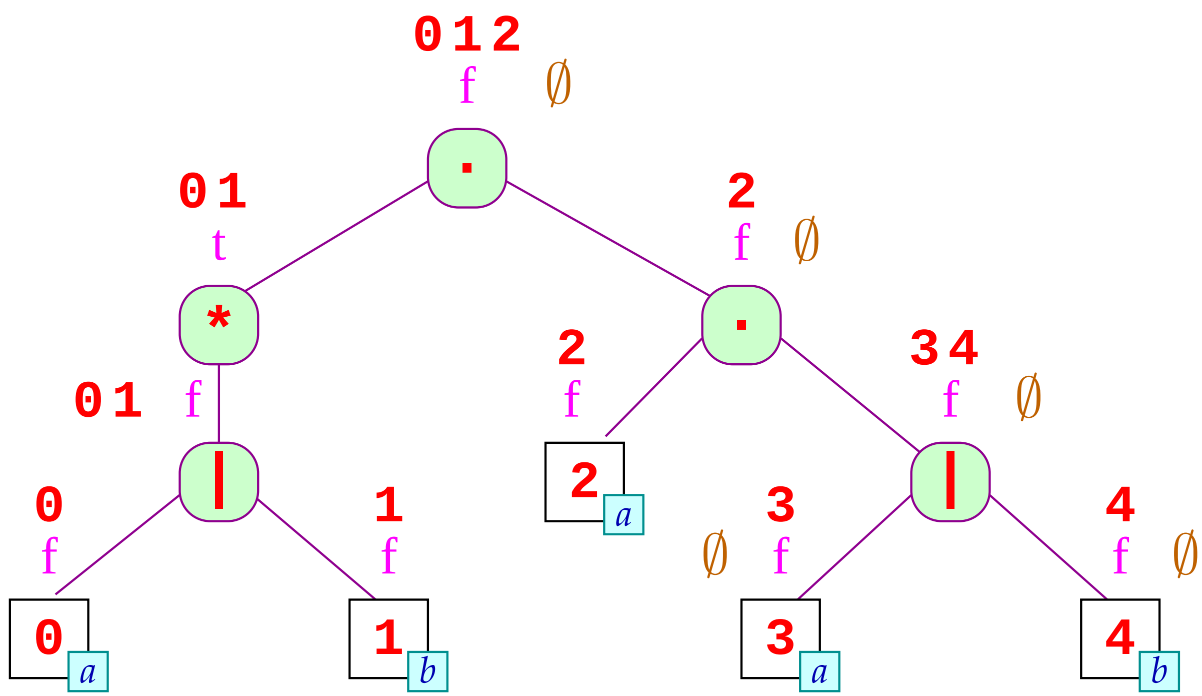
... im Beispiel:



3. Schritt:

Die Menge nächster Bätter: $\text{next}[r] = \{i \mid (r\bullet, \epsilon, \bullet\boxed{i \mid x}) \in \delta^*\}$

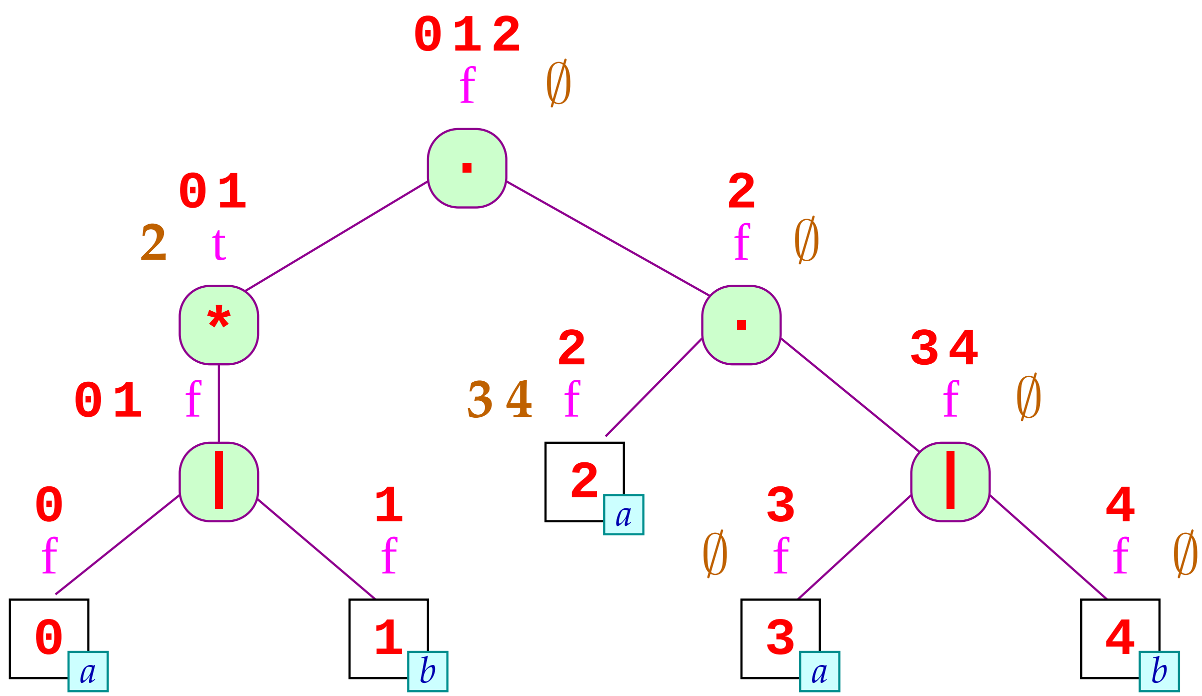
... im Beispiel:



3. Schritt:

Die Menge nächster Bätter: $\text{next}[r] = \{i \mid (r\bullet, \epsilon, \bullet\boxed{i \mid x}) \in \delta^*\}$

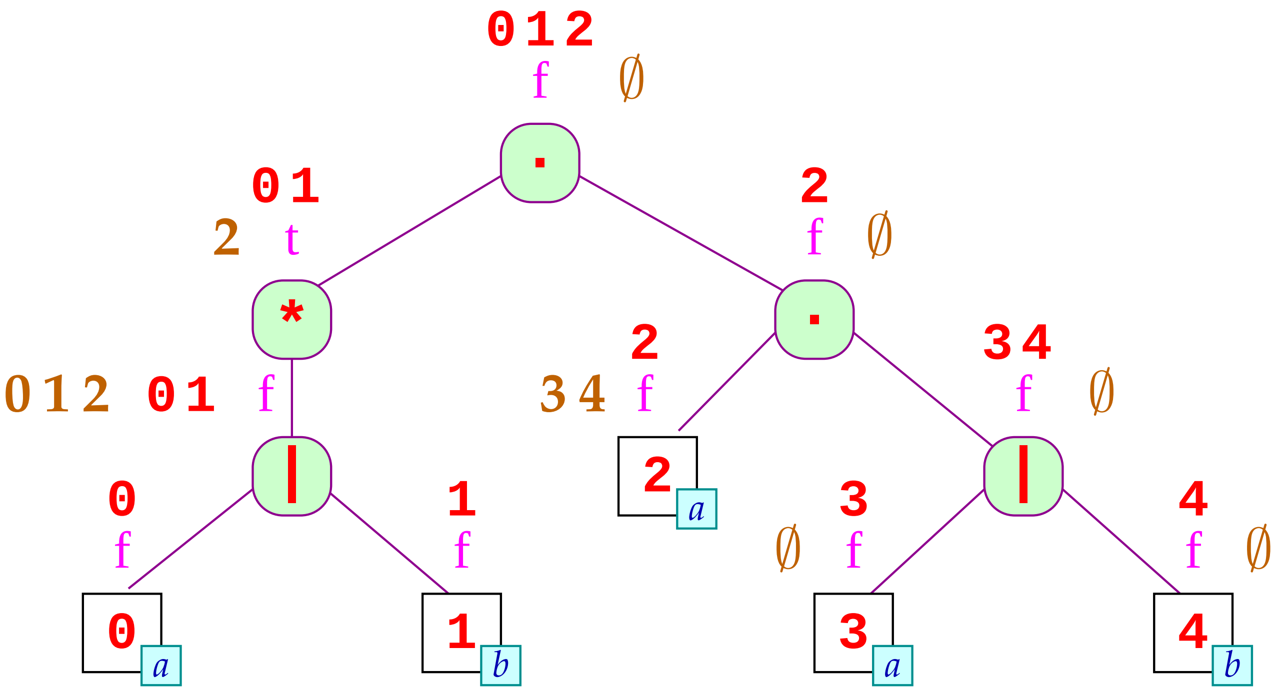
... im Beispiel:



3. Schritt:

Die Menge nächster Bätter: $\text{next}[r] = \{i \mid (r\bullet, \epsilon, \bullet\boxed{i \mid x}) \in \delta^*\}$

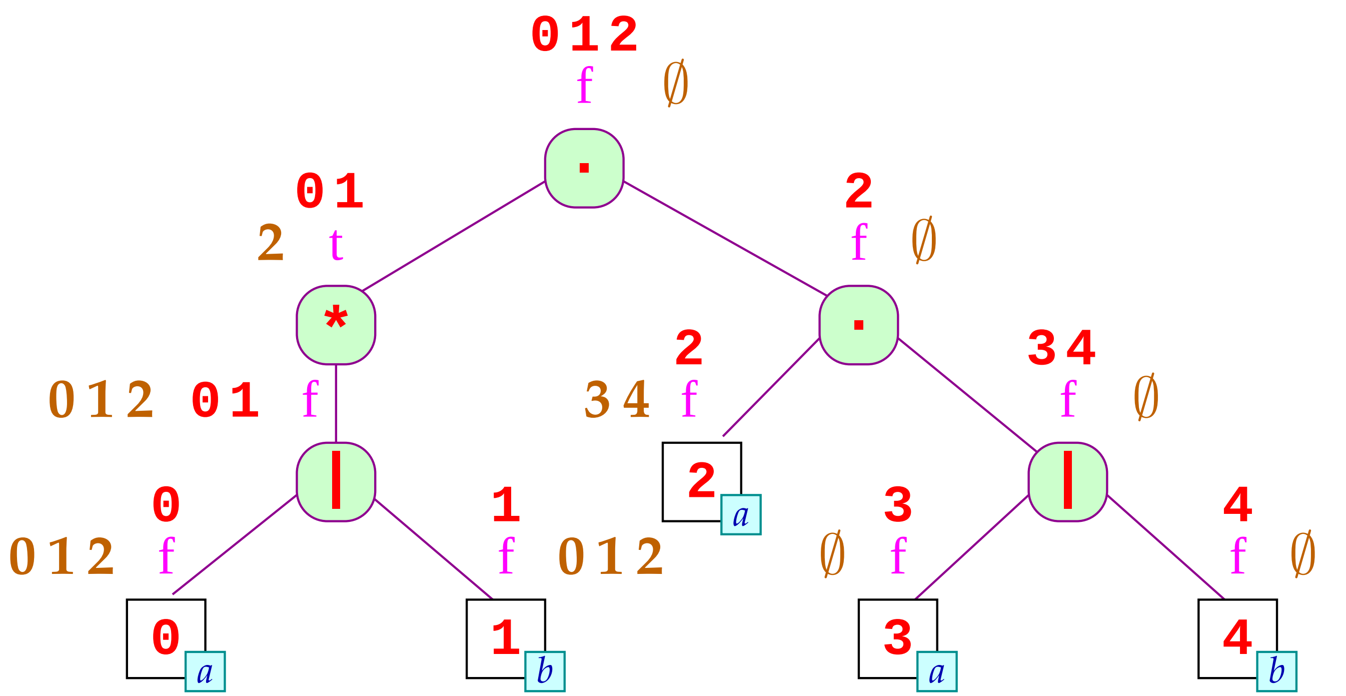
... im Beispiel:



3. Schritt:

Die Menge nächster Bätter: $\qquad \mathsf{next}[r] = \{i \mid (r\bullet, \epsilon, \bullet\boxed{i \mid x}) \in \delta^*\}$

... im Beispiel:



Implementierung: DFS pre-order Traversierung ;-)

Für die Wurzel haben wir:

$$\mathsf{next}[e] = \emptyset$$

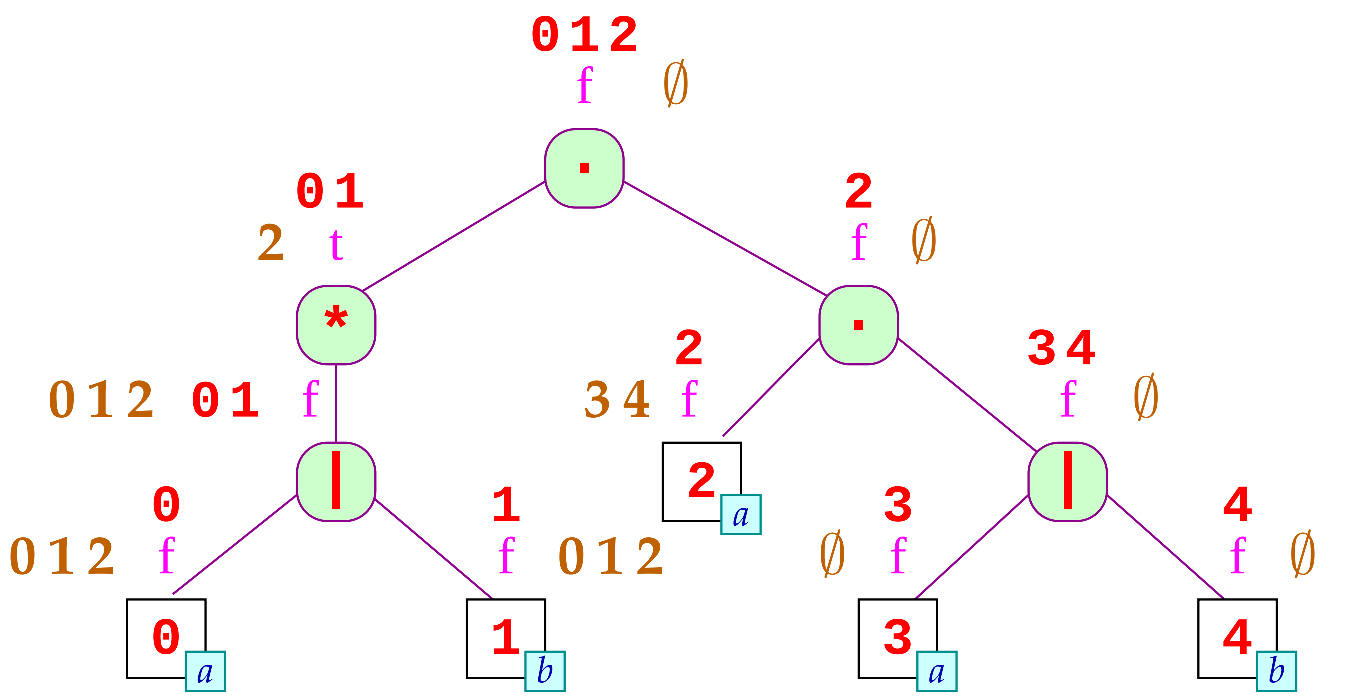
Ansonsten machen wir eine Fallunterscheidung über den Kontext:

r	Regeln
$r_1 \mid r_2$	$\mathsf{next}[r_1] = \mathsf{next}[r]$ $\mathsf{next}[r_2] = \mathsf{next}[r]$
$r_1 \cdot r_2$	$\mathsf{next}[r_1] = \begin{cases} \mathsf{first}[r_2] \cup \mathsf{next}[r] & \text{falls} \quad \mathsf{empty}[r_2] = t \\ \mathsf{first}[r_2] & \text{falls} \quad \mathsf{empty}[r_2] = f \end{cases}$ $\mathsf{next}[r_2] = \mathsf{next}[r]$
r_1^*	$\mathsf{next}[r_1] = \mathsf{first}[r_1] \cup \mathsf{next}[r]$
$r_1?$	$\mathsf{next}[r_1] = \mathsf{next}[r]$

4. Schritt:

Die Menge **letzter** Bätter: $\text{last}[r] = \{i \text{ in } r \mid (\boxed{i \mid x}\bullet, \epsilon, r\bullet) \in \delta^*\}$

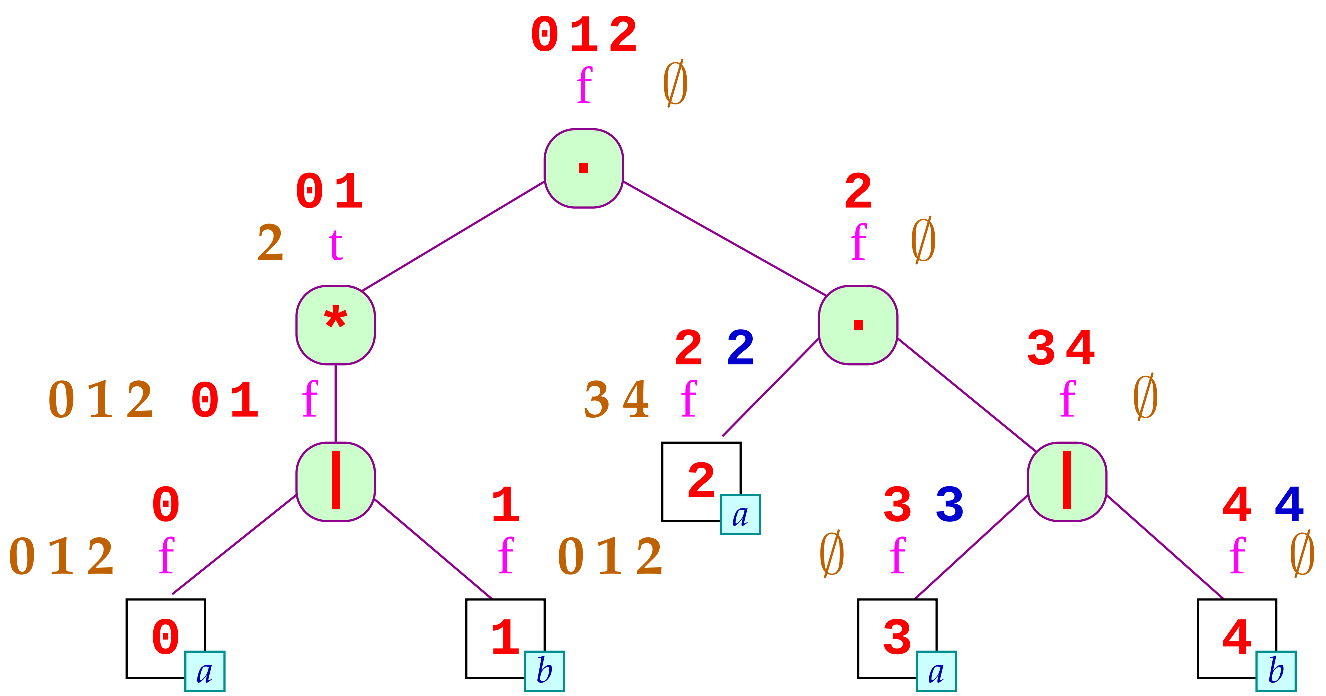
... im Beispiel:



4. Schritt:

Die Menge letzter Bätter: $\text{last}[r] = \{i \text{ in } r \mid (\boxed{i \mid x}\bullet, \epsilon, r\bullet) \in \delta^*\}$

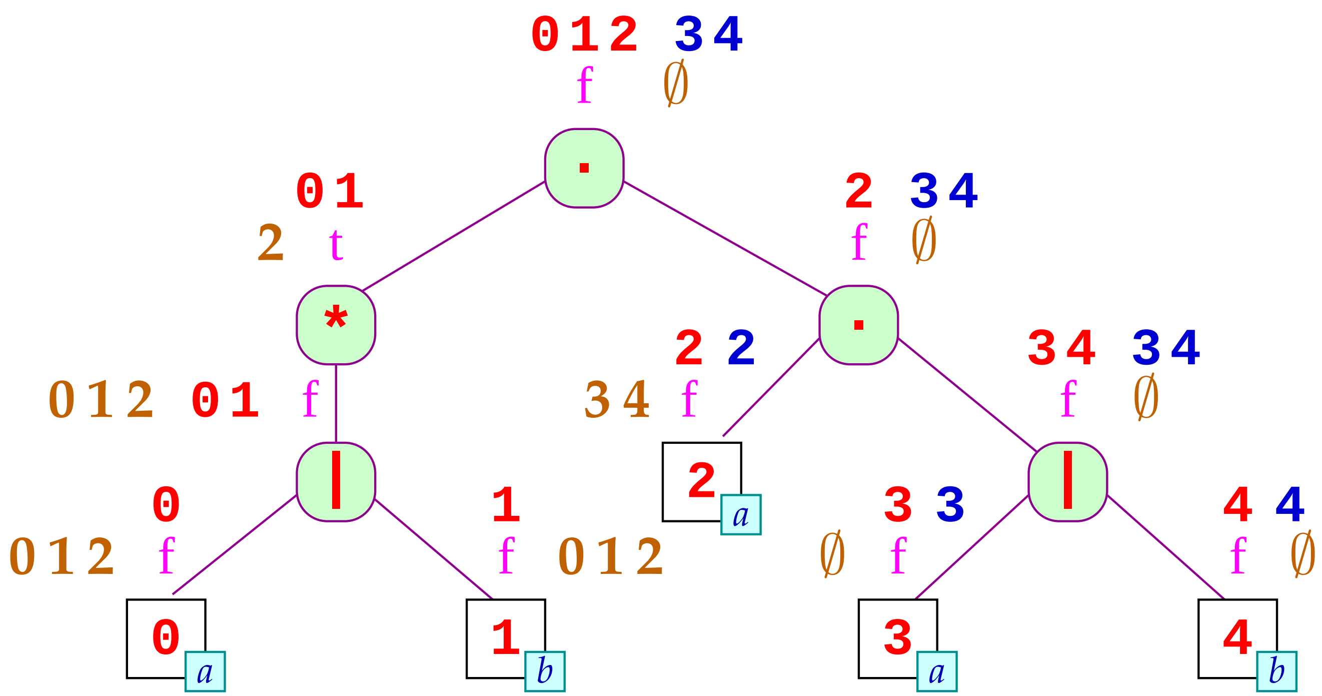
... im Beispiel:



4. Schritt:

Die Menge letzter Bätter: $\mathsf{last}[r] = \{i \text{ in } r \mid (\boxed{i \mid x}\bullet, \epsilon, r\bullet) \in \delta^*, x \neq \epsilon\}$

... im Beispiel:



Implementierung: DFS post-order Traversierung :-)

Für Blätter $r \equiv \boxed{i \mid x}$ ist $\mathsf{last}[r] = \{i \mid x \not\equiv \epsilon\}$.

Andernfalls:

$$
\begin{array}{lll}
\mathsf{last}[r_1 \mid r_2] & = & \mathsf{last}[r_1] \cup \mathsf{last}[r_2] \\
\mathsf{last}[r_1 \cdot r_2] & = & \begin{cases} \mathsf{last}[r_1] \cup \mathsf{last}[r_2] & \text{falls} \quad \mathsf{empty}[r_2] = t \\ \mathsf{last}[r_2] & \text{falls} \quad \mathsf{empty}[r_2] = f \end{cases} \\
\mathsf{last}[r_1^*] & = & \mathsf{last}[r_1] \\
\mathsf{last}[r_1?] & = & \mathsf{last}[r_1]
\end{array}
$$

Integration:

Zustände: $\{\bullet e\} \cup \{i\bullet \mid i \text{ Blatt}\}$

Startzustand: $\bullet e$

Endzustände:

Falls $\mathsf{empty}[e] = f$, dann $\mathsf{last}[e]$. Andernfalls: $\{\bullet e\} \cup \mathsf{last}[e]$.

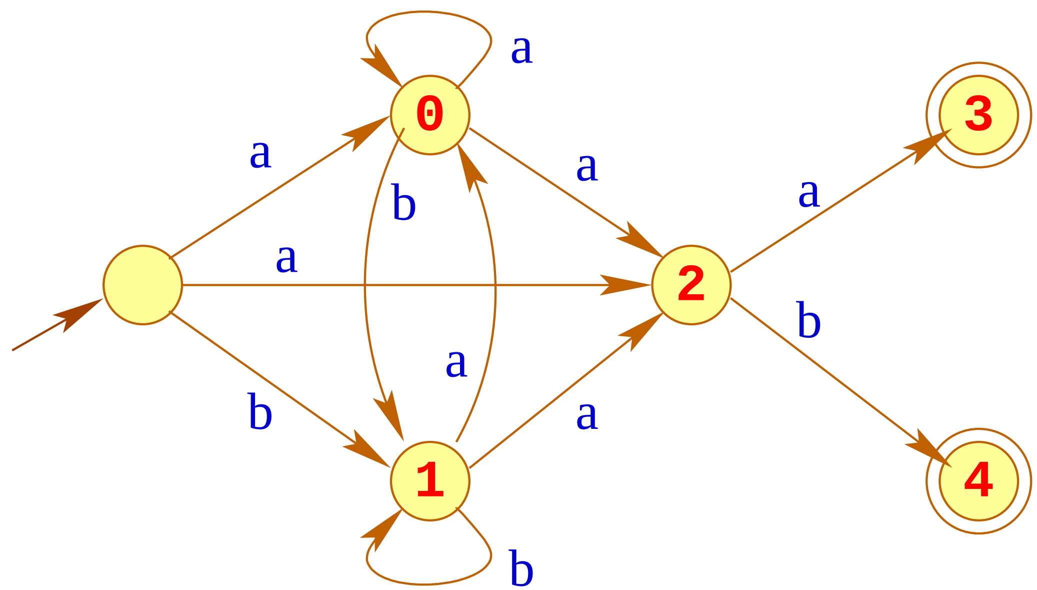
Übergänge:

$(\bullet e, a, i\bullet)$ falls $i \in \mathsf{first}[e]$ und i mit a beschriftet ist;

$(i\bullet, a, i'\bullet)$ falls $i' \in \mathsf{next}[i]$ und i' mit a beschriftet ist.

Den resultierenden Automaten bezeichnen wir mit A_e.

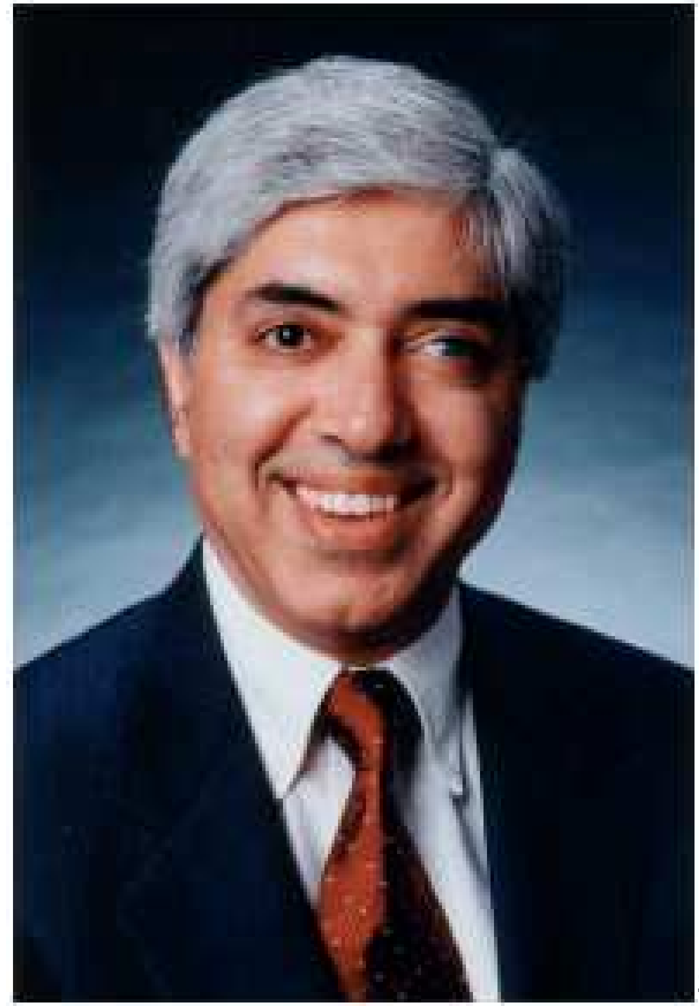
... im Beispiel:



Bemerkung:

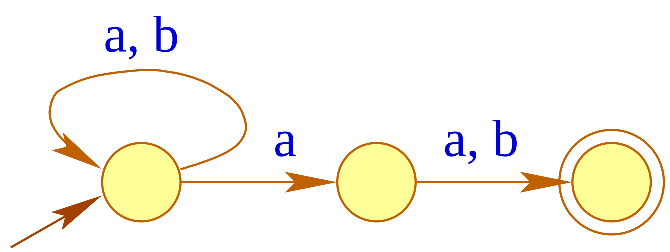
- Die Konstruktion heißt auch Berry-Sethi- oder Glushkow-Konstruktion.
- Sie wird in XML zur Definition von Content Models benutzt ;-)
- Das Ergebnis ist vielleicht nicht, was wir erwartet haben ...

Gerard Berry, Esterel Technologies



Ravi Sethi, Research VR, Lucent Technologies

Der erwartete Automat:

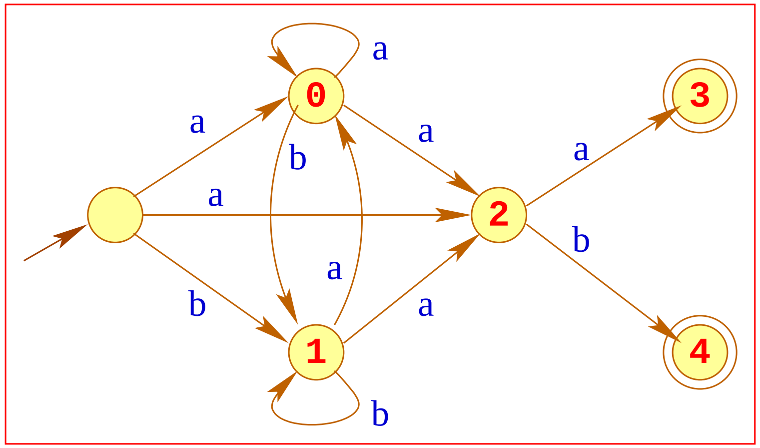


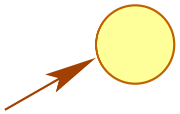
Bemerkung:

- in einen Zustand eingehende Kanten haben hier nicht unbedingt die gleiche Beschriftung :-)
- Dafür ist die Berry-Sethi-Konstruktion direkter ;-)
- In Wirklichkeit benötigen wir aber deterministische Automaten

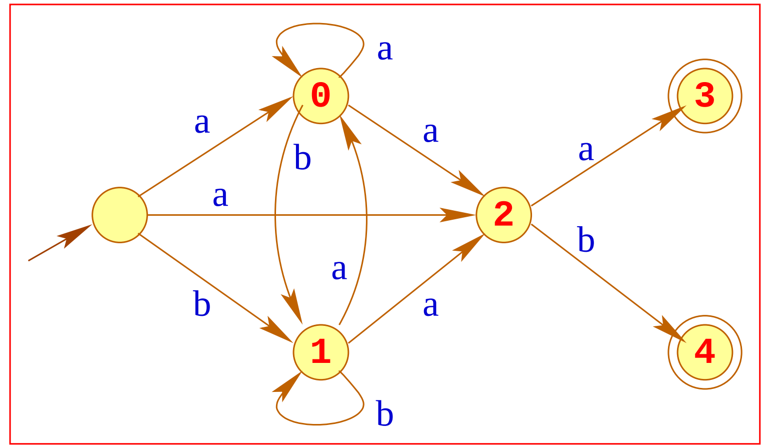
$\Longrightarrow$ Teilmengen-Konstruktion

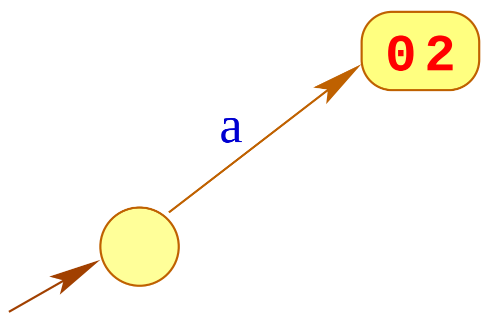
... im Beispiel:



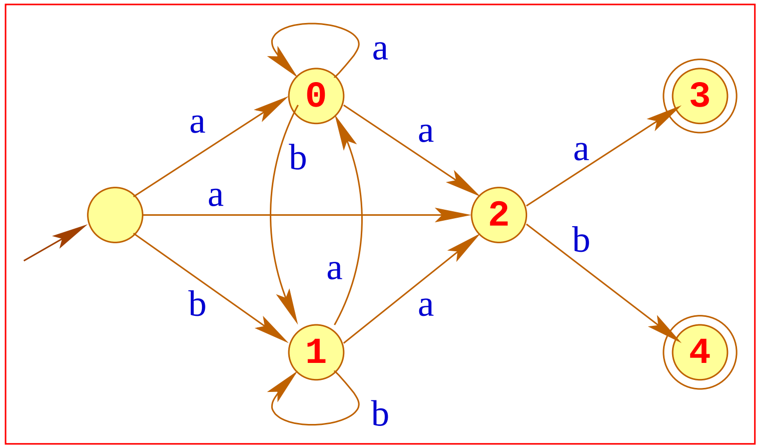


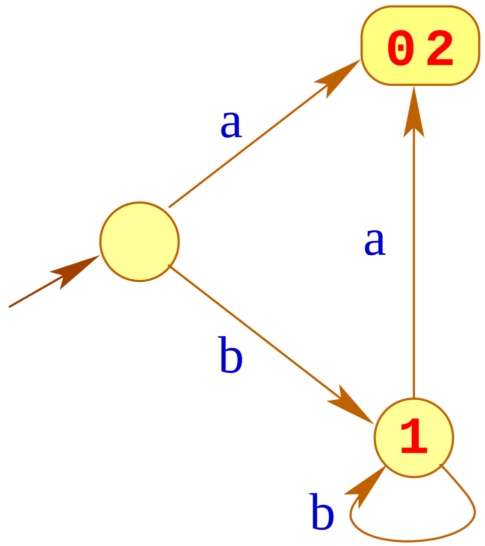
... im Beispiel:



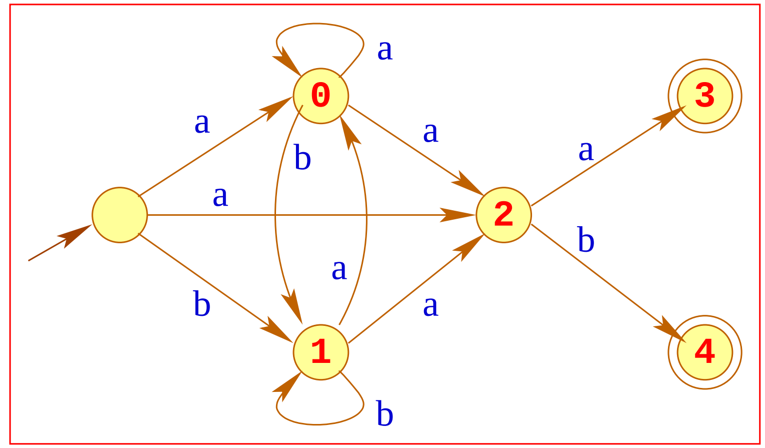


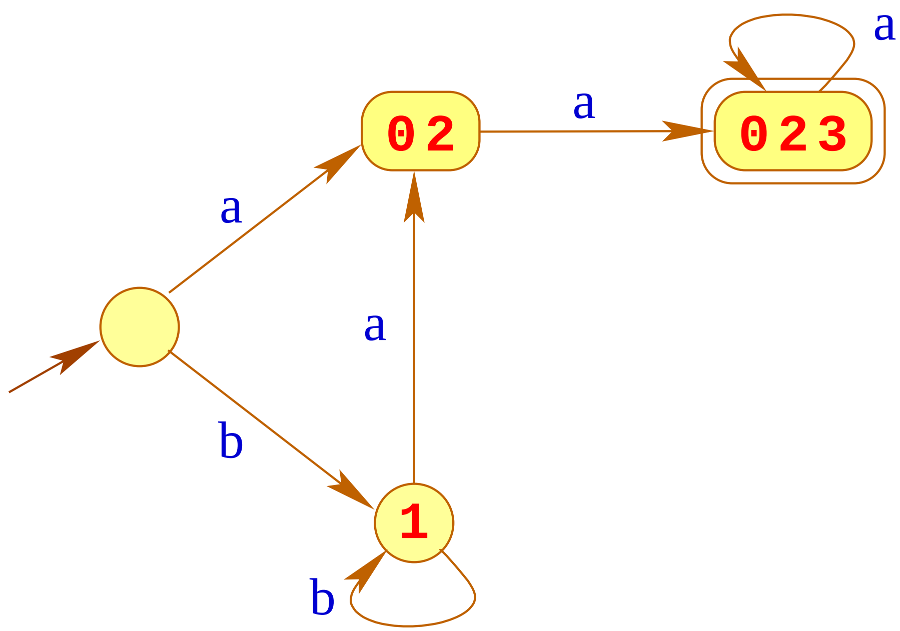
... im Beispiel:



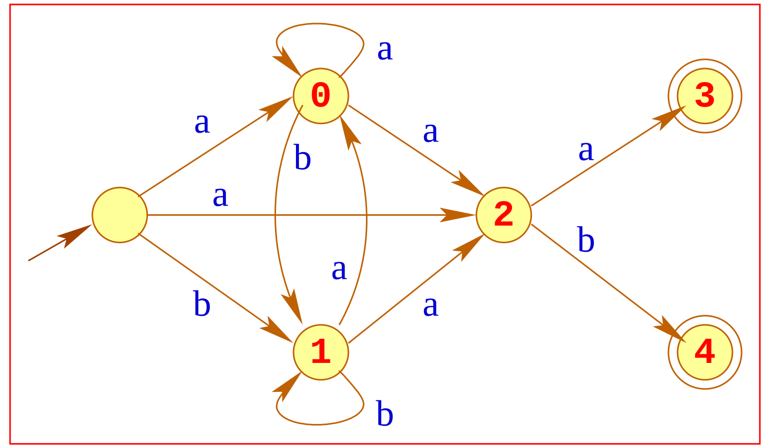


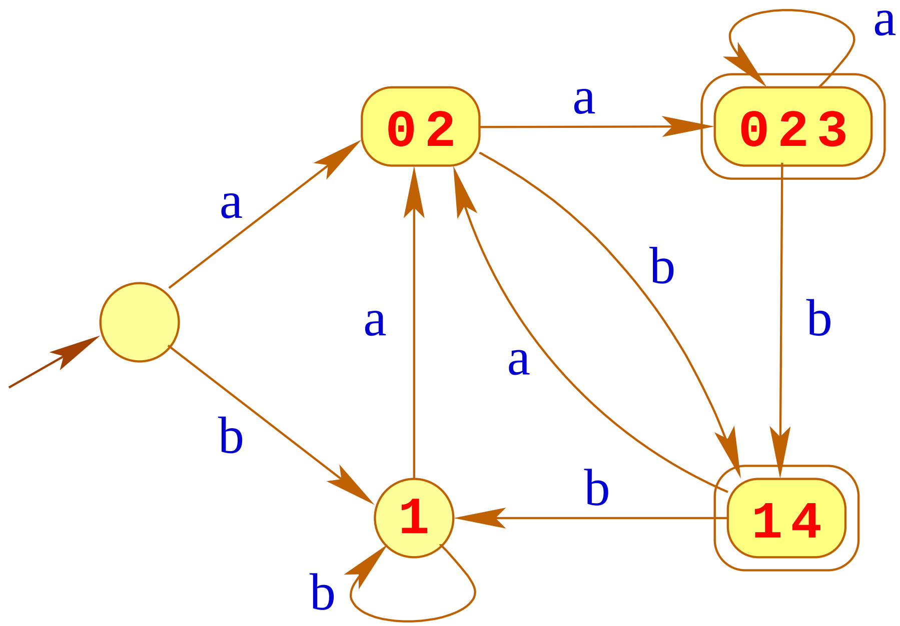
... im Beispiel:





... im Beispiel:





Satz:

Zu jedem nichtdeterministischen Automaten $A = (Q, \Sigma, \delta, I, F)$ kann ein deterministischer Automat $\mathcal{P}(A)$ konstruiert werden mit

$$\mathcal{L}(A) = \mathcal{L}(\mathcal{P}(A))$$

Satz:

Zu jedem nichtdeterministischen Automaten $A = (Q, \Sigma, \delta, I, F)$ kann ein deterministischer Automat $\mathcal{P}(A)$ konstruiert werden mit

$$\mathcal{L}(A) = \mathcal{L}(\mathcal{P}(A))$$

Konstruktion:

Zustände: Teilmengen von Q;

Anfangszustände: $\{I\}$;

Endzustände: $\{Q' \subseteq Q \mid Q' \cap F \neq \emptyset\}$;

Übergangsfunktion: $\delta_{\mathcal{P}}(Q', a) = \{q \in Q \mid \exists\, p \in Q' : \ (p, a, q) \in \delta\}$.

Achtung:

- Leider gibt es exponentiell viele Teilmengen von Q :-(
- Um nur nützliche Teilmengen zu betrachten, starten wir mit der Menge $Q_{\mathcal{P}} = \{I\}$ und fügen weitere Zustände nur nach Bedarf hinzu ...
- d.h., wenn wir sie von einem Zustand in $Q_{\mathcal{P}}$ aus erreichen können :-)
- Trotz dieser Optimierung kann der Ergebnisautomat riesig sein :-((

  ... was aber in der Praxis (so gut wie) nie auftritt :-))

Achtung:

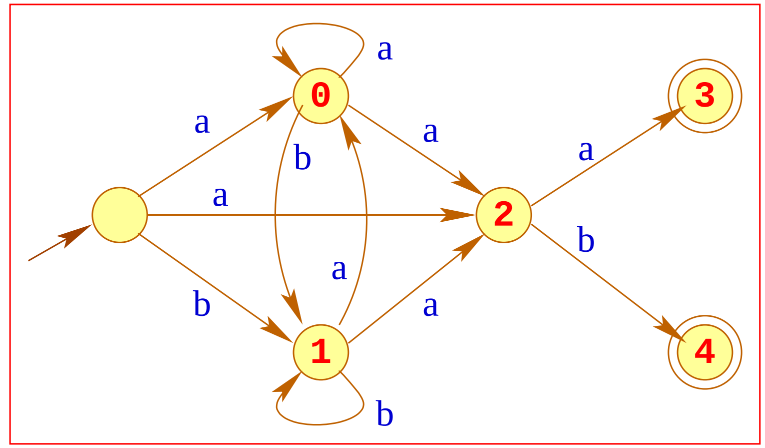
- Leider gibt es exponentiell viele Teilmengen von Q :-(
- Um nur nützliche Teilmengen zu betrachten, starten wir mit der Menge $Q_{\mathcal{P}} = \{I\}$ und fügen weitere Zustände nur nach Bedarf hinzu ...
- d.h., wenn wir sie von einem Zustand in $Q_{\mathcal{P}}$ aus erreichen können :-)
- Trotz dieser Optimierung kann der Ergebnisautomat riesig sein :-((

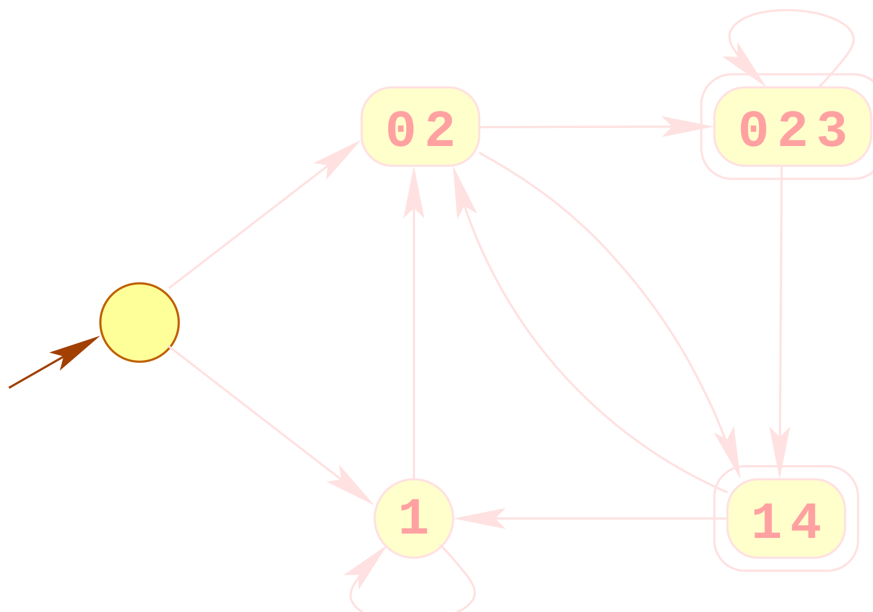
  ... was aber in der Praxis (so gut wie) nie auftritt :-))

- In Tools wie `grep` wird deshalb zu der DFA zu einem regulären Ausdruck nicht aufgebaut !!!
- Stattdessen werden während der Abbarbeitung der Eingabe genau die Mengen konstruiert, die für die Eingabe notwendig sind ...

... im Beispiel:

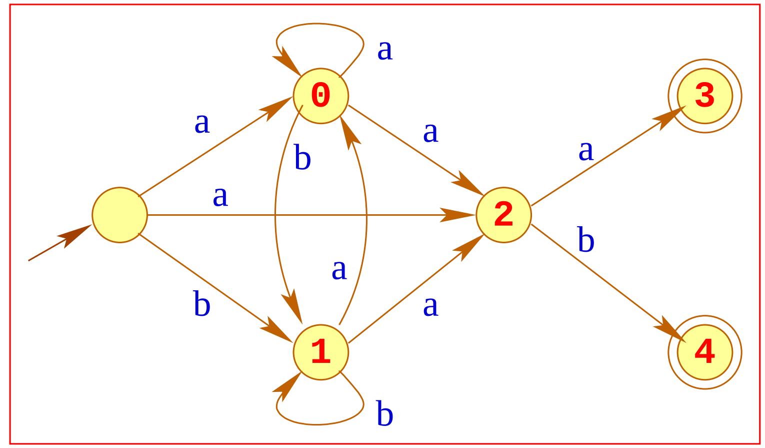
a	b	a	b

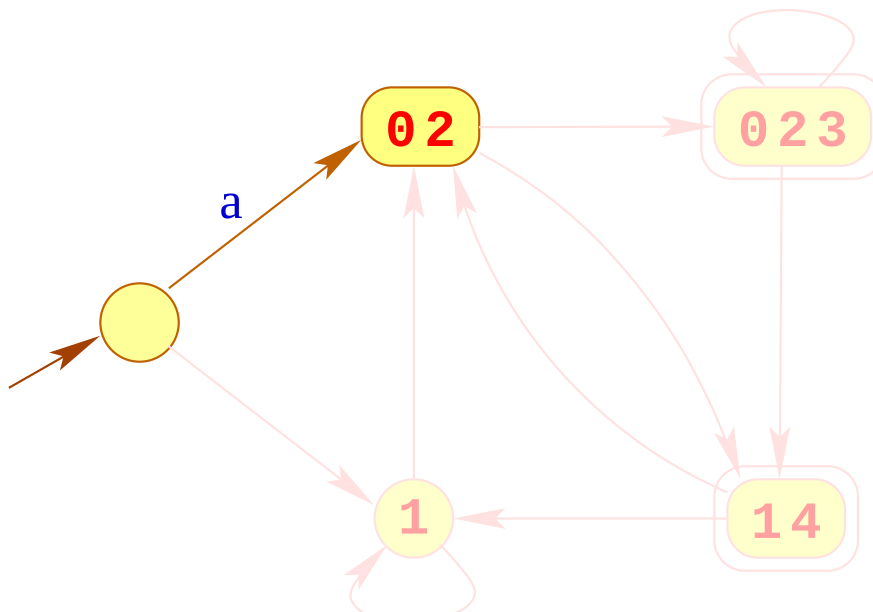




... im Beispiel:

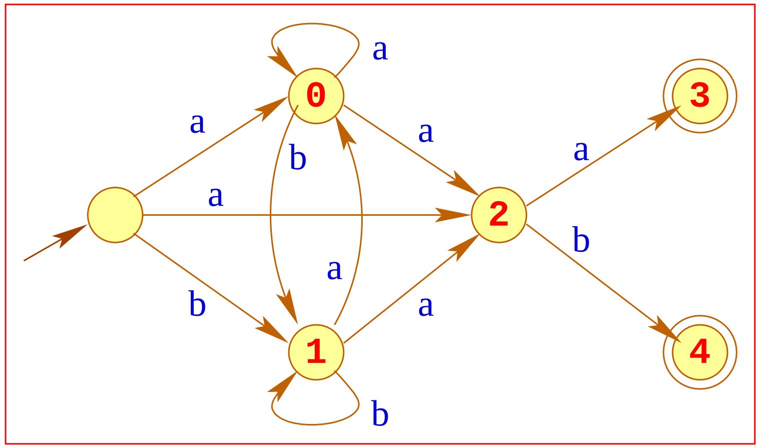
a	b	a	b

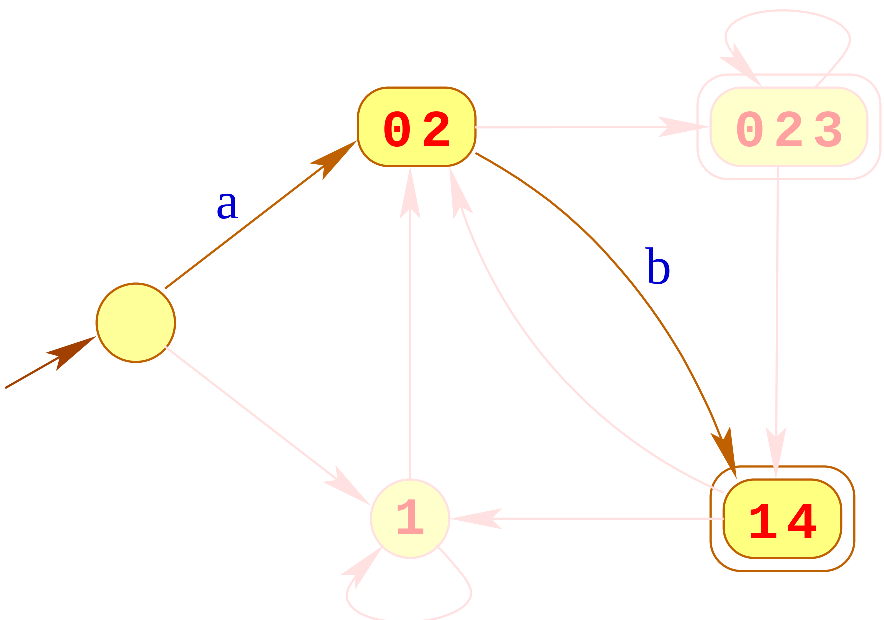




... im Beispiel:

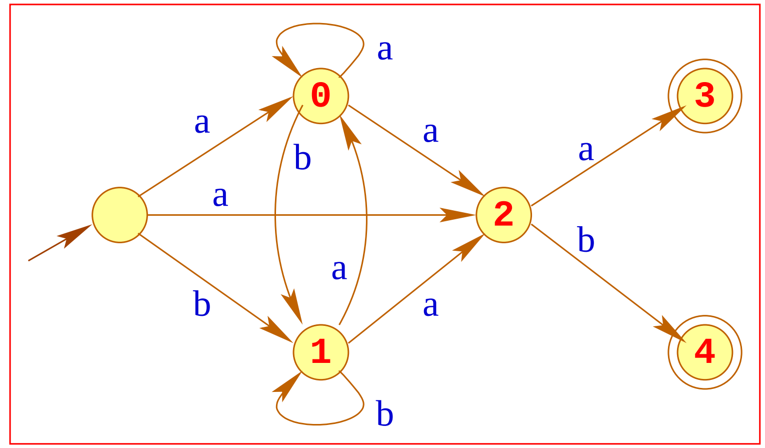
a	b	a	b

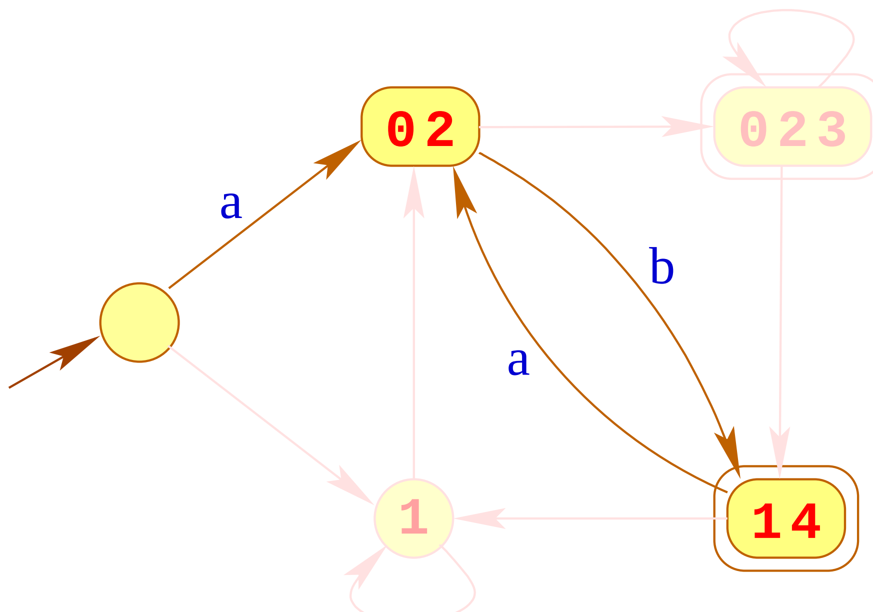




... im Beispiel:

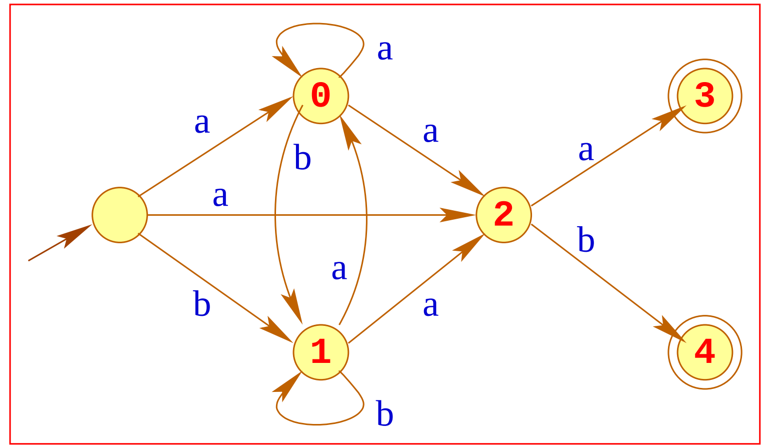
a	b	a	b

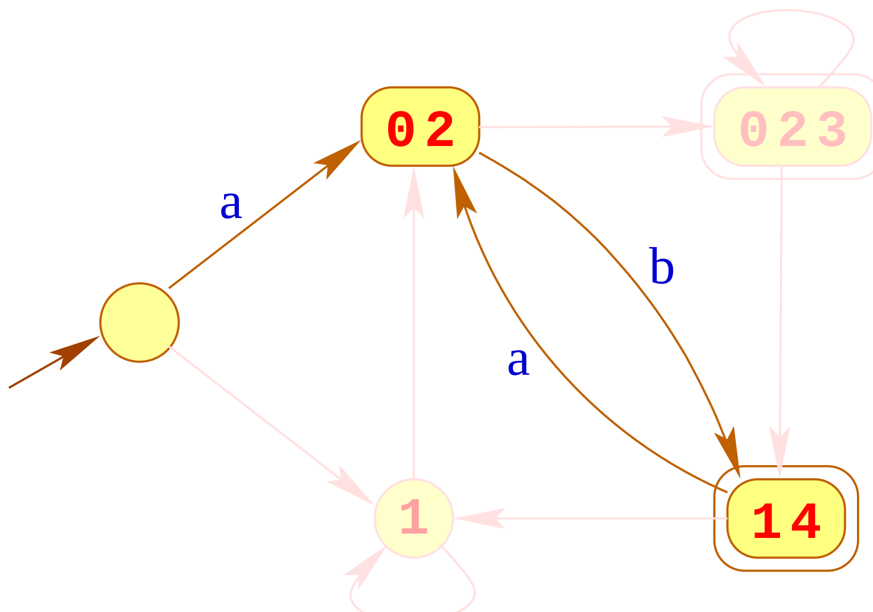




... im Beispiel:

a	b	a	b





Bemerkungen:

- Bei einem Eingabewort der Länge n werden maximal $\mathcal{O}(n)$ Mengen konstruiert :-)
- Ist eine Menge bzw. eine Kante des DFA einmal konstuiert, heben wir sie in einer Hash-Tabelle auf.
- Bevor wir einen neuen Übergang konstruieren, sehen wir erst nach, ob wir diesen nicht schon haben :-)

Bemerkungen:

- Bei einem Eingabewort der Länge n werden maximal $\mathcal{O}(n)$ Mengen konstruiert :-)
- Ist eine Menge bzw. eine Kante des DFA einmal konstuiert, heben wir sie in einer Hash-Tabelle auf.
- Bevor wir einen neuen Übergang konstruieren, sehen wir erst nach, ob wir diesen nicht schon haben :-)

Zusammen fassend finden wir:

Satz

Zu jedem regulären Ausdruck e kann ein deterministischer Automat $A = \mathcal{P}(A_e)$ konstruiert werden mit

$$\mathcal{L}(A) = [\![e]\!]$$

1.3 Design eines Scanners

Eingabe (vereinfacht): eine Menge von Regeln:

$$
\begin{array}{ll}
e_1 & \{\ \texttt{action}_1\ \} \\
e_2 & \{\ \texttt{action}_2\ \} \\
\ldots & \\
e_k & \{\ \texttt{action}_k\ \}
\end{array}
$$

1.3 Design eines Scanners

Eingabe (vereinfacht): eine Menge von Regeln:

$$\begin{array}{ll} e_1 & \{\ \mathtt{action}_1\ \} \\ e_2 & \{\ \mathtt{action}_2\ \} \\ & \dots \\ e_k & \{\ \mathtt{action}_k\ \} \end{array}$$

Ausgabe: ein Programm, das

... von der Eingabe ein maximales Präfix w liest, das $e_1 \mid \dots \mid e_k$ erfüllt;

... das minimale i ermittelt, so dass $w \in [\![e_i]\!]$;

... für w $\mathtt{action}_i$ ausführt.

Implementierung:

Idee:

- Konstruiere den DFA $\mathcal{P}(A_e) = (Q, \Sigma, \delta, \{q_0\}, F)$ zu dem Ausdruck $e = (e_1 \mid \ldots \mid e_k)$;
- Definiere die Mengen:

$$
\begin{array}{lcl}
F_1 & = & \{q \in F \mid q \cap \mathsf{last}[e_1] \neq \emptyset\} \\
F_2 & = & \{q \in (F \backslash F_1) \mid q \cap \mathsf{last}[e_2] \neq \emptyset\} \\
 & \ldots & \\
F_k & = & \{q \in (F \backslash (F_1 \cup \ldots \cup F_{k-1})) \mid q \cap \mathsf{last}[e_k] \neq \emptyset\}
\end{array}
$$

- Für Eingabe w gilt: $\delta^*(q_0, w) \in F_i$ genau dann wenn der Scanner für w $\texttt{action}_i$ ausführen soll :-)

Idee (Fortsetzung):

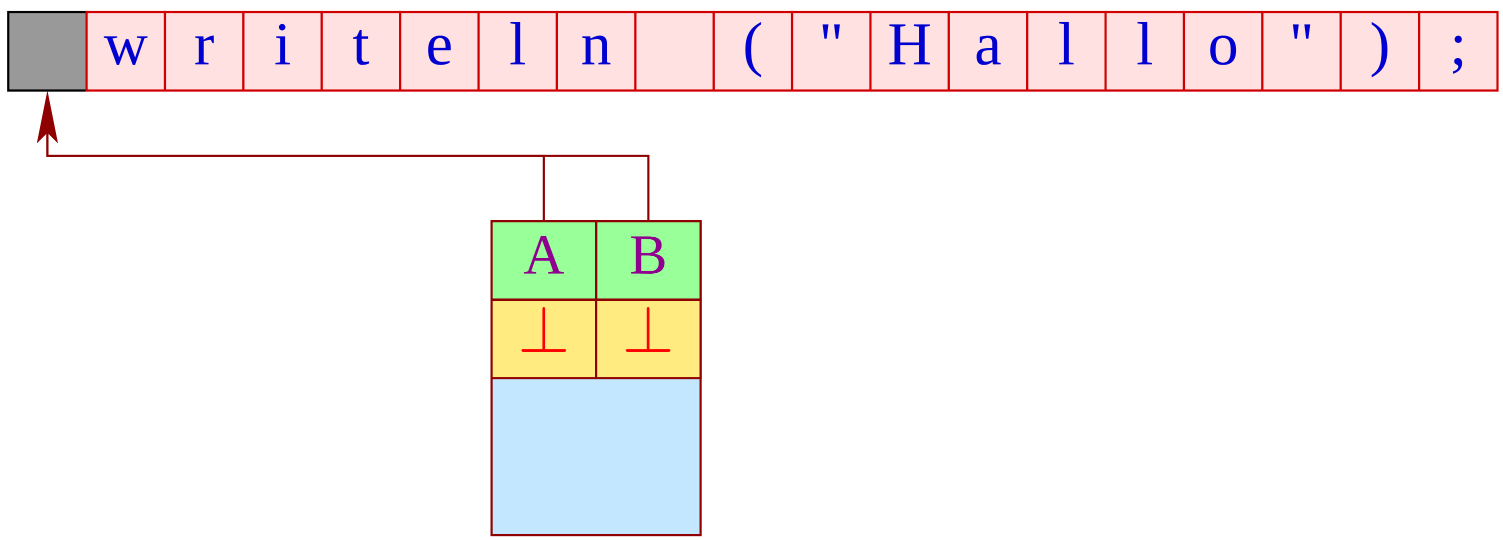
- Der Scanner verwaltet zwei Zeiger $\langle A, B\rangle$ und die zugehörigen Zustände $\langle q_A, q_B\rangle$...
- Der Zeiger A merkt sich die letzte Position in der Eingabe, nach der ein Zustand $q_A \in F$ erreicht wurde;
- Der Zeiger B verfolgt die aktuelle Position.

s	t	d	o	u	t	.	w	r	i	t	e	l	n		(	"	H	a	l	l	o	"	)	;

A	B

Idee (Fortsetzung):

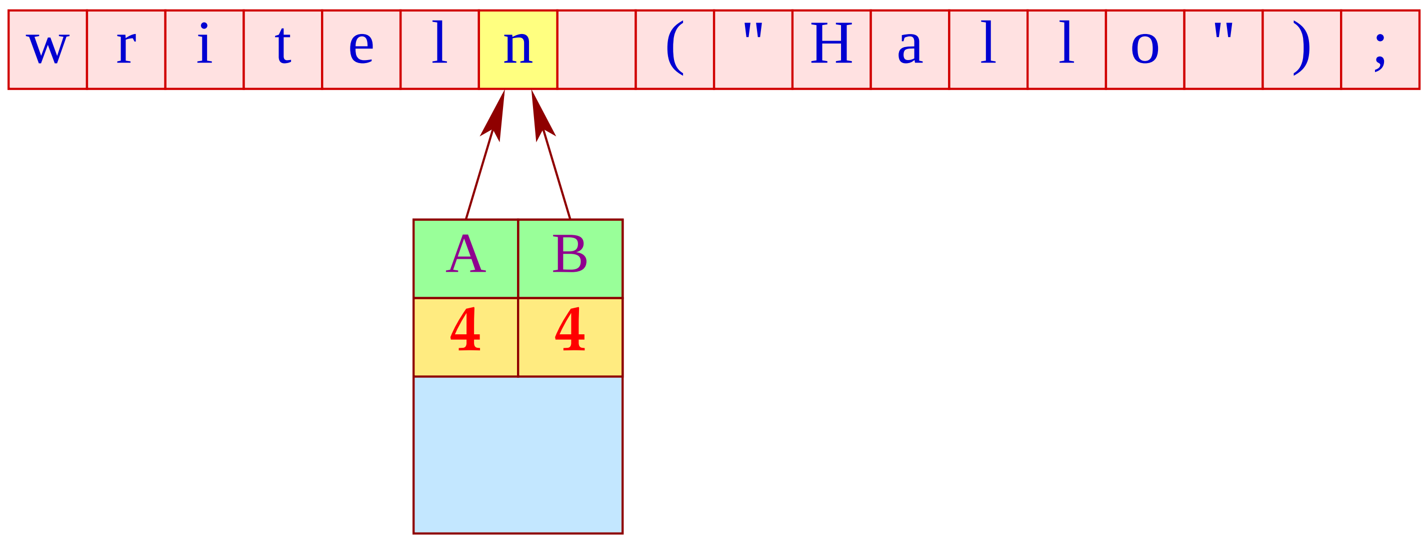
- Der Scanner verwaltet zwei Zeiger $\langle A, B \rangle$ und die zugehörigen Zustände $\langle q_A, q_B \rangle$...
- Der Zeiger A merkt sich die letzte Position in der Eingabe, nach der ein Zustand $q_A \in F$ erreicht wurde;
- Der Zeiger B verfolgt die aktuelle Position.



Idee (Fortsetzung):

- Ist der aktuelle Zustand $q_B = \emptyset$, geben wir Eingabe bis zur Postion A aus und setzen:

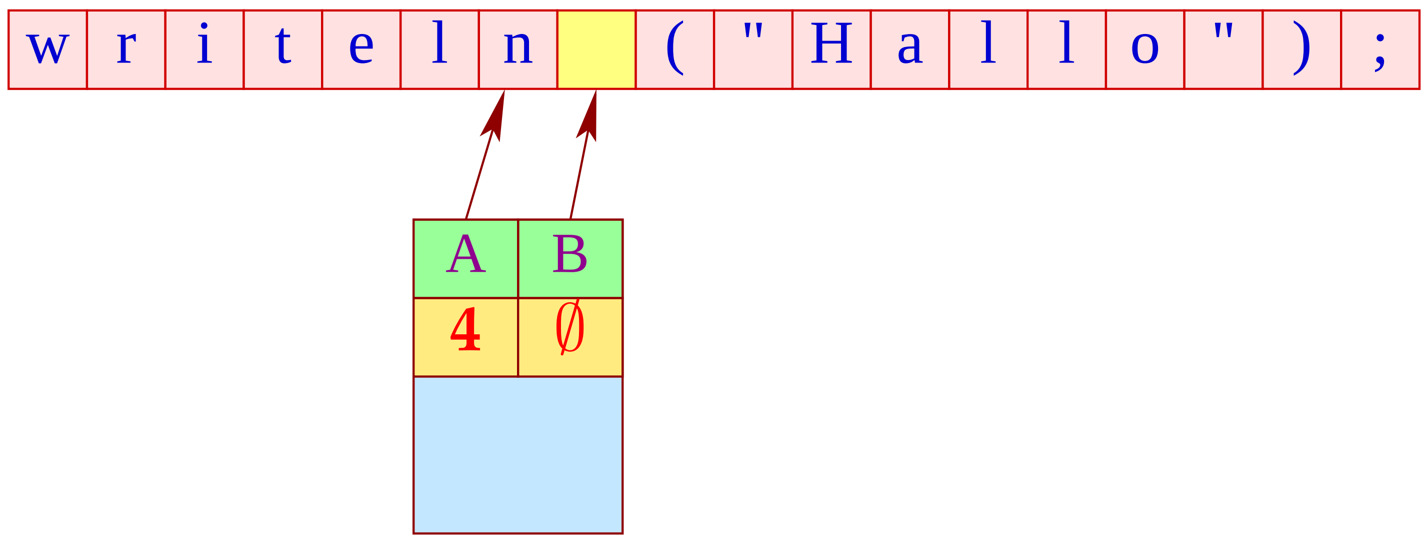
$$B := A; \qquad A := \perp;$$
$$q_B := q_0; \qquad q_A := \perp$$



Idee (Fortsetzung):

- Ist der aktuelle Zustand $q_B = \emptyset$, geben wir Eingabe bis zur Postion A aus und setzen:

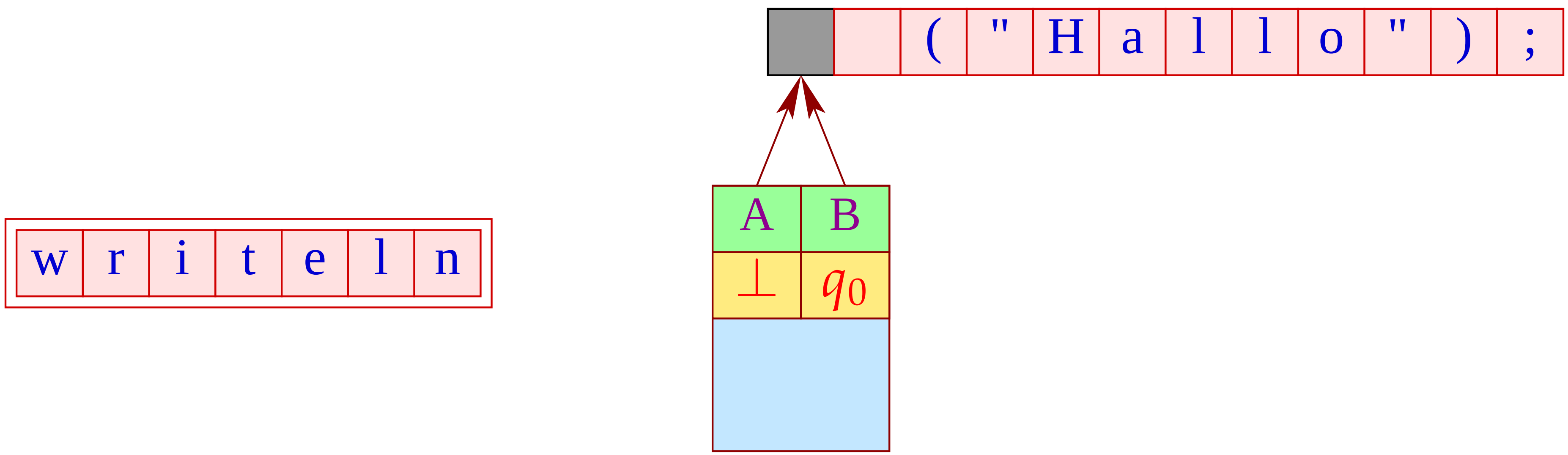
$$
\begin{array}{lllllll}
B & := & A; & \qquad & A & := & \perp; \\
q_B & := & q_0; & \qquad & q_A & := & \perp
\end{array}
$$



Idee (Fortsetzung):

- Ist der aktuelle Zustand $q_B = \emptyset$, geben wir Eingabe bis zur Postion A aus und setzen:

$$\begin{array}{lllllll} B & := & A; & \quad & A & := & \perp; \\ q_B & := & q_0; & \quad & q_A & := & \perp \end{array}$$



Erweiterung: Zustände

- Gelegentlich ist es nützlich, unterschiedliche Scanner-Zustände zu unterscheiden.
- In unterschiedlichen Zuständen sollen verschiedene Tokenklassen erkannt werden können.
- In Abhängigkeit der gelesenen Tokens kann der Scanner-Zustand geändert werden ;-)

Beispiel: Kommentare

Innerhalb eines Kommentars werden Identifier, Konstanten, Kommentare, ... nicht erkannt ;-)

Eingabe (verallgemeinert): eine Menge von Regeln:

$$
\begin{array}{llll}
\langle \mathtt{state} \rangle & \{ & e_1 & \{\ \mathtt{action}_1 \quad \mathtt{yybegin(state}_1\mathtt{);}\ \} \\
 & & e_2 & \{\ \mathtt{action}_2 \quad \mathtt{yybegin(state}_2\mathtt{);}\ \} \\
 & & \ldots & \\
 & & e_k & \{\ \mathtt{action}_k \quad \mathtt{yybegin(state}_k\mathtt{);}\ \} \\
 & \} & &
\end{array}
$$

- Der Aufruf `yybegin (state`$_i$`);` setzt den Zustand auf `state`$_i$.
- Der Startzustand ist (z.B. bei JFlex) `YYINITIAL`.

... im Beispiel:

$$
\begin{array}{llll}
\langle \mathtt{YYINITIAL} \rangle & & "/*" & \{\ \mathtt{yybegin(COMMENT);}\ \} \\
\langle \mathtt{COMMENT} \rangle & \{ & "*/" & \{\ \mathtt{yybegin(YYINITIAL);}\ \} \\
 & & . \mid \backslash\mathtt{n} & \{\ \ \} \\
 & \} & &
\end{array}
$$

Bemerkungen:

- “`.`” matcht alle Zeichen ungleich “`\n`”.
- Für jeden Zustand generieren wir den entsprechenden Scanner.
- Die Methode `yybegin (STATE);` schaltet zwischen den verschiedenen Scannern um.
- Kommentare könnte man auch direkt mithilfe einer geeigneten Token-Klasse implementieren. Deren Beschreibung ist aber ungleich komplizierter :-)
- Scanner-Zustände sind insbesondere nützlich bei der Implementierung von Präprozessoren, die in einen Text eingestreute Spezifikationen expandieren sollen.

1.4 Implementierung von DFAs

Aufgaben:

- Implementiere die Übergangsfunktion $\delta : Q \times \Sigma \to Q$
- Implementiere eine Klassifizierung $r : Q \to \mathbb{N}$

1.4 Implementierung von DFAs

Aufgaben:

- Implementiere die Übergangsfunktion $\delta : Q \times \Sigma \to Q$
- Implementiere eine Klassifizierung $r : Q \to \mathbb{N}$

Probleme:

- Die Anzahl der Zustände kann sehr groß sein :-(
- Das Alphabet kann sehr groß sein: z.B. Unicode :-((

Reduzierung der Anzahl der Zustände

Idee: Minimierung

- Identifiziere Zustände, die sich im Hinblick auf eine Klassifizierung r gleich verhalten :-)
- Sei $A = (Q, \Sigma, \delta, \{q_0\}, r)$ ein DFA mit Klassifizierung. Wir definieren auf den Zuständen eine Äquivalenzrelation durch:

$$p \equiv_r q \quad \text{gdw.} \quad \forall\, w \in \Sigma\ :\ r(\delta(p, w)) = r(\delta(q, w))$$

- Die neuen Zustände sind Äquivalenzklassen der alten Zustände :-)

Zustände	$[q]_r, q \in Q$
Anfangszustand	$[q_0]_r$
Klassifizierung	$r([q]_r) = r(q)$
Übergangsfunktion	$\delta([p]_r, a) = [\delta(p, a)]_r$

Problem: Wie berechnet man $\equiv_r$?

Idee:

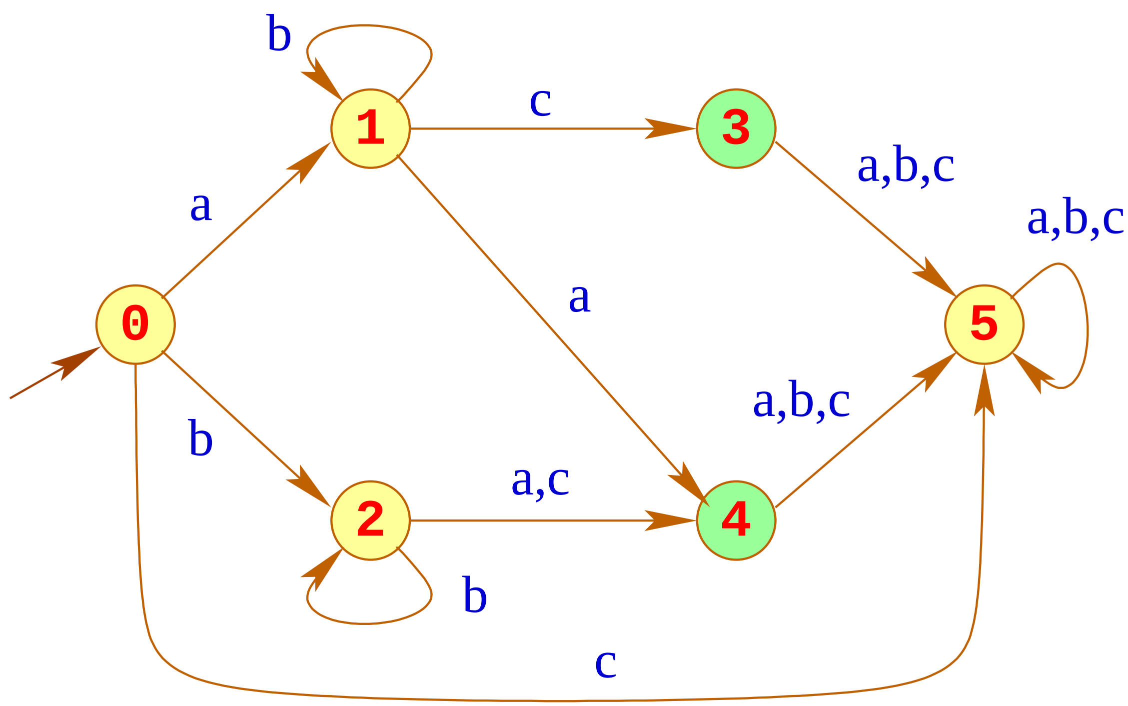
- Wir nehmen an, maximal viel sei äquivalent :-)

  Wir starten mit der Partition:

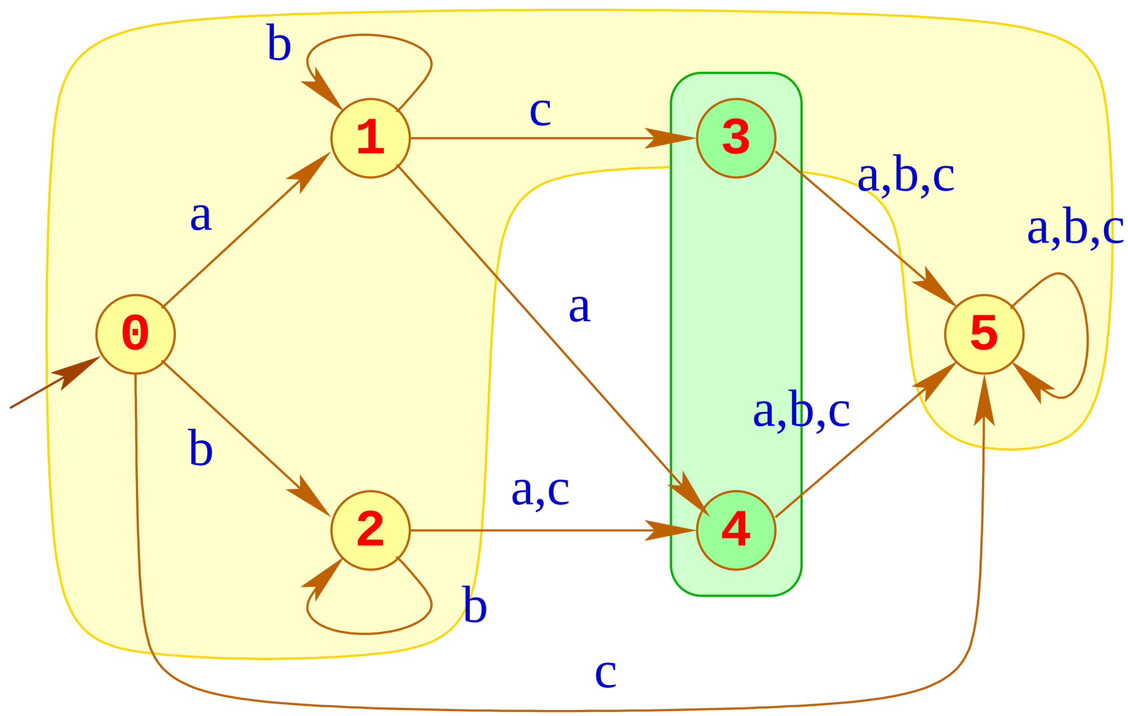
$$\overline{Q} = \{r^{-1}(i) \neq \emptyset \mid i \in \mathbb{N}\}$$

- Finden wir in $\bar{q} \in \overline{Q}$ Zustände p_1, p_2 sodass $\delta(p_1, a)$ und $\delta(p_2, a)$ in verschiedenen Äquivalenzklassen liegen (für irgend ein a), müssen wir $\bar{q}$ aufteilen ...

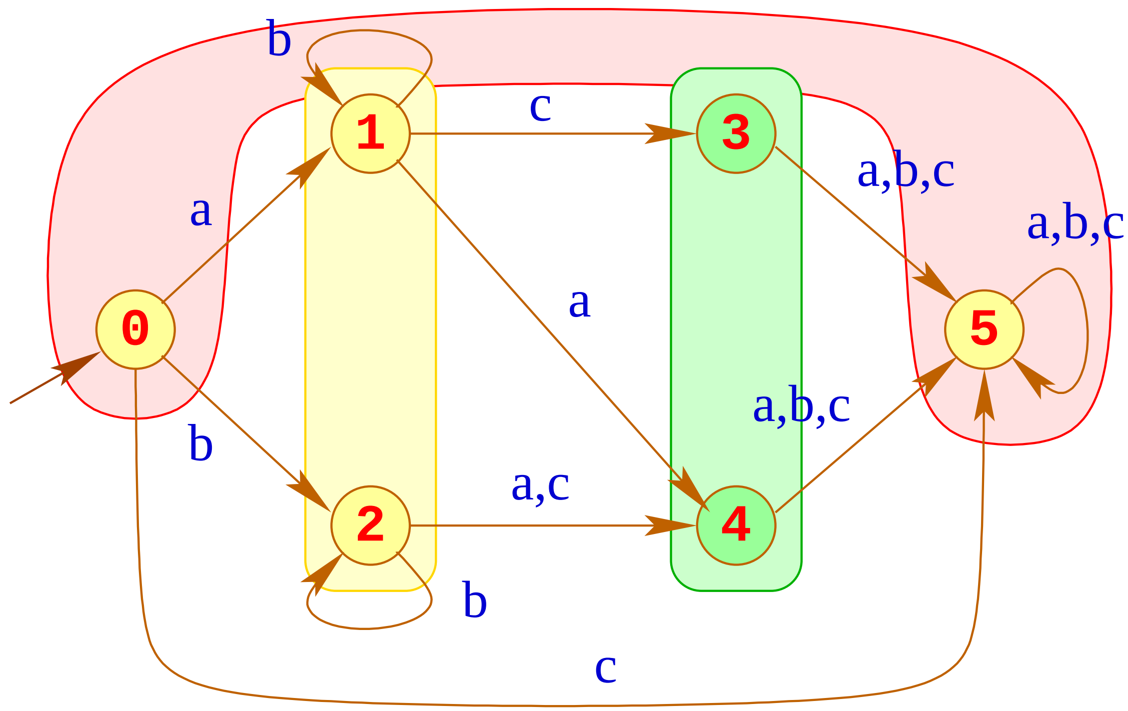
Beispiel:



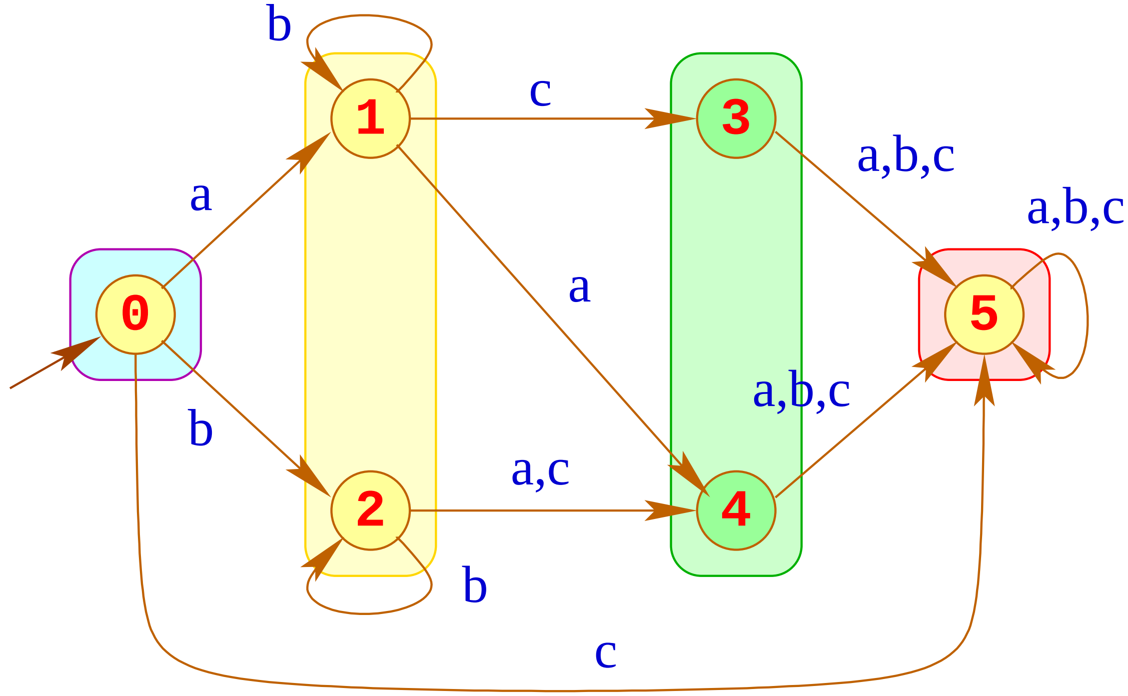
Beispiel:



Beispiel:



Beispiel:



Bemerkungen:

- Das Verfahren liefert die gröbste Partition $\overline{Q}$, die mit r und δ verträglich ist, d.h. für $\bar{q} \in \overline{Q}$,

$$(1) \quad p_1, p_2 \in \bar{q} \quad \Longrightarrow \quad r(p_1) = r(p_2)$$

$$(2) \quad p_1, p_2 \in \bar{q} \quad \Longrightarrow \quad \delta(p_1, a), \delta(p_2, a) \quad \text{gehören zur gleichen Klasse}$$

- Der Ergebnis-Automat ist der eindeutig bestimmte minimale Automat für $\mathcal{L}(A)$;-)
- Eine naive Implementierung erfordert Laufzeit $\mathcal{O}(n^2)$.

  Eine raffinierte Verwaltung der Partition liefert ein Verfahren mit Laufzeit $\mathcal{O}(n \cdot \log(n))$.

Anil Nerode , Cornell University, Ittaca

John E. Hopcroft, Cornell University, Ittaca

Reduzierung der Tabellengröße

Problem:

- Die Tabelle für δ wird mit Paaren (q, a) indiziert.
- Sie enthält eine Spalte für jedes $a \in \Sigma$.
- Das Alphabet Σ umfasst i.a. ASCII, evt. aber ganz Unicode :-(

1. Idee:

- Bei großen Alphabeten wird man in der Spezifikation i.a. nicht einzelne Zeichen auflisten, sondern Zeichenklassen benutzen :-)
- Lege Spalten nicht für einzelne Zeichen sondern für Klassen von Zeichen an, die sich gleich verhalten.

Beispiel:

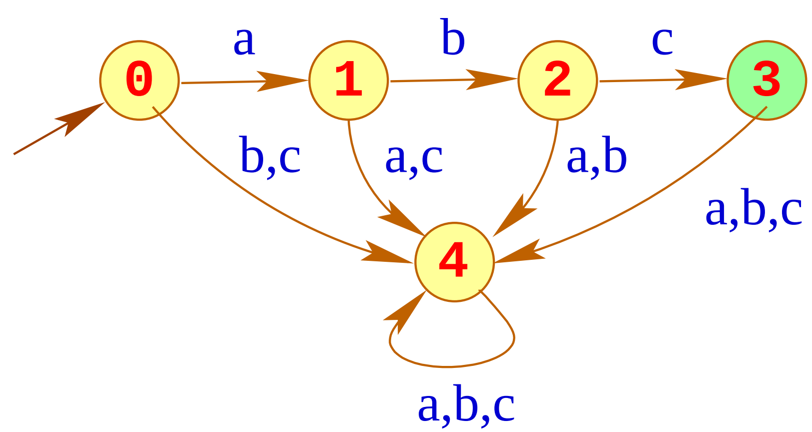
```
le = [a-zA-Z_\$]
ledi = [a-zA-Z_\$0-9]
Id = {le} {ledi}*
```

- Der Automat soll deterministisch sein.
- Sind die Klassen der Spezifikation nicht disjunkt, teilt man sie darum in Unterklassen auf, hier in die Klassen `[a-zA-Z_\$]` und `[0-9]` :-)

2. Idee:

- Finden wir, dass mehrere (Unter-) Klassen der Spezifikation in der Spalte übereinstimmen, können wir sie nachträglich wieder vereinigen :-)
- Wir können weitere Methoden der Tabellen-Komprimierung anwenden, z.B. Zeilenverschiebung (Row Displacement) ...

Beispiel:



... die zugehörige Tabelle (transponiert):

	0	1	2	3	4
a	1	4	4	4	4
b	4	2	4	4	4
c	4	4	3	4	4

Beobachtung:

- Viele Einträge in der Tabelle sind gleich einem Wert Default (hier: `4`)
- Diesen Wert brauchen wir nicht zu repräsentieren :-)

... die zugehörige Tabelle (transponiert):

	0	1	2	3	4
a	1				
b		2			
c			3		

Beobachtung:

- Viele Einträge in der Tabelle sind gleich einem Wert Default (hier: 4)
- Diesen Wert brauchen wir nicht zu repräsentieren :-)
- Dann legen wir einfach mehrere (transponierte) Spalten übereinander :-))

... im Beispiel:

	0	1	2
A	1	2	3
valid	a	b	c

- Feld valid teilt mit, für welches Element aus Σ der Eintrag gilt :-)
- Achtung: I.a. werden die Spalten nicht so perfekt übereinander passen!

  Dann verschieben wir sie so lange, bis die jeweils nächste in die bisherigen Löcher hineinpasst.
- Darum müssen wir ein zusätzliches Feld displacement verwalten, in dem wir uns die Verschiebung merken ;-)

Ein Feld-Zugriff $\delta(j, a)$ wird dann so realisiert:

$$\begin{array}{rcl}
\delta(j,a) & = & \mathsf{let}\quad d \;=\; \mathsf{displacement}[a] \\
 & & \mathsf{in}\;\; \mathsf{if}\;\; (\mathsf{valid}[d+j] \equiv a) \\
 & & \qquad\quad \mathsf{then}\;\; \mathsf{A}[d+j] \\
 & & \qquad\quad \mathsf{else}\;\; \mathsf{Default} \\
 & & \mathsf{end}
\end{array}$$

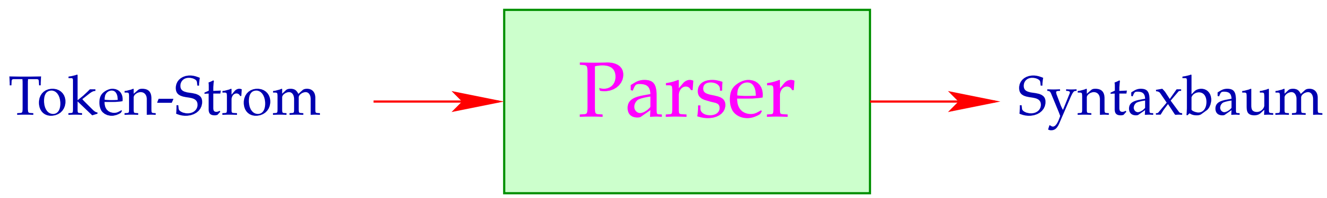
Ein Feld-Zugriff $\delta(j, a)$ wird dann so realisiert:

$$\begin{array}{lll} \delta(j,a) & = & \textsf{let} \quad d \quad = \quad \textsf{displacement}[a] \\ & & \textsf{in} \;\; \textsf{if} \;\; (\textsf{valid}[d+j] \equiv a) \\ & & \qquad\quad \textsf{then} \;\; \textsf{A}[d+j] \\ & & \qquad\quad \textsf{else} \;\; \textsf{Default} \\ & & \textsf{end} \end{array}$$

Diskussion:

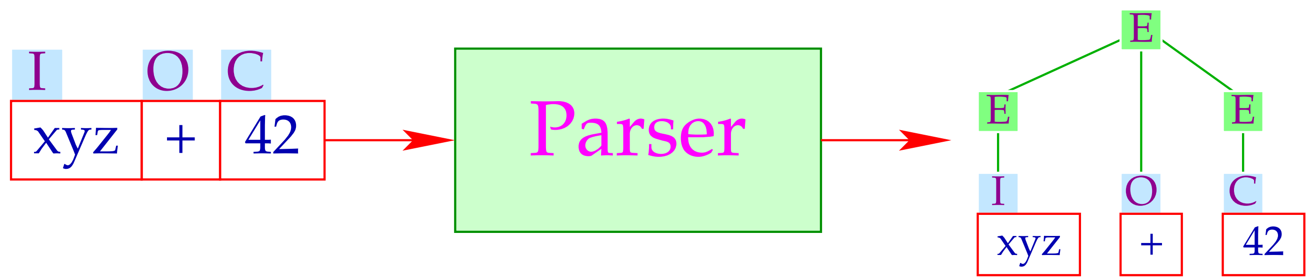
- Die Tabellen werden i.a. erheblich kleiner.
- Dafür werden Tabellenzugriffe etwas teurer.

2 Die syntaktische Analyse



- Die syntaktische Analyse versucht, Tokens zu größeren Programmeinheiten zusammen zu fassen.

2 Die syntaktische Analyse



- Die syntaktische Analyse versucht, Tokens zu größeren Programmeinheiten zusammen zu fassen.
- Solche Einheiten können sein:

  $\longrightarrow$ Ausdrücke;

  $\longrightarrow$ Statements;

  $\longrightarrow$ bedingte Verzweigungen;

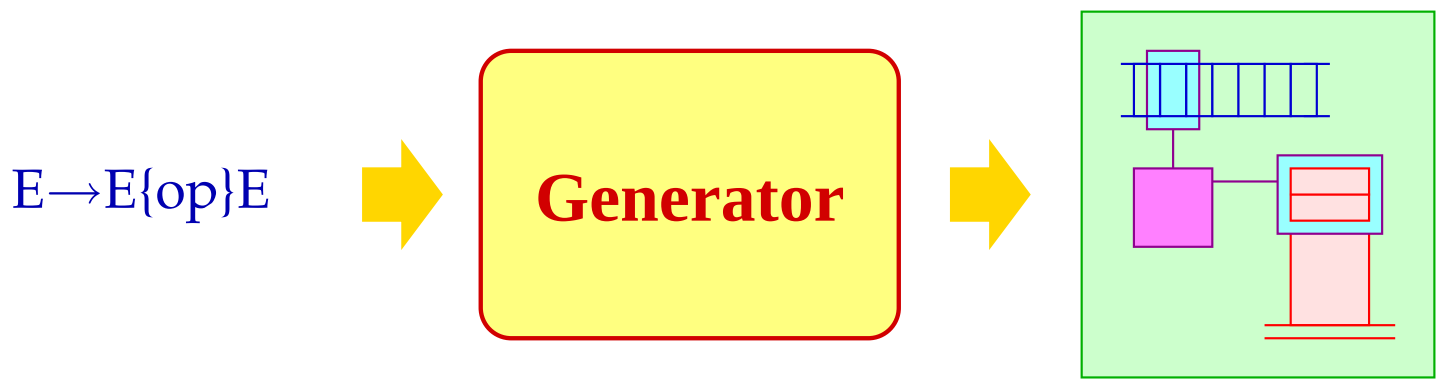
  $\longrightarrow$ Schleifen; ...

Diskussion:

Auch Parser werden i.a. nicht von Hand programmiert, sondern aus einer Spezifikation generiert:

Diskussion:

Auch Parser werden i.a. nicht von Hand programmiert, sondern aus einer Spezifikation generiert:



Spezifikation der hierarchischen Struktur: kontextfreie Grammatiken;

Generierte Implementierung: Kellerautomaten + X :-)

2.1 Grundlagen: Kontextfreie Grammatiken

- Programme einer Programmiersprache können unbeschränkt viele Tokens enthalten, aber nur endlich viele Token-Klassen :-)
- Als endliches Terminal-Alphabet T wählen wir darum die Menge der Token-Klassen.
- Die Schachtelung von Programm-Konstrukten lässt sich elegant mit Hilfe von kontextfreien Grammatiken beschreiben ...

2.1 Grundlagen: Kontextfreie Grammatiken

- Programme einer Programmiersprache können unbeschränkt viele Tokens enthalten, aber nur endlich viele Token-Klassen :-)
- Als endliches Terminal-Alphabet T wählen wir darum die Menge der Token-Klassen.
- Die Schachtelung von Programm-Konstrukten lässt sich elegant mit Hilfe von kontextfreien Grammatiken beschreiben ...

Eine kontextfreie Grammatik (CFG) ist ein 4-Tupel $G = (N, T, P, S)$, wobei:

- N die Menge der Nichtterminale,
- T die Menge der Terminale,
- P die Menge der Produktionen oder Regeln, und
- $S \in N$ das Startsymbol ist.

Noam Chomsky, MIT (Guru)

John Backus, IBM (Erfinder von Fortran)

Die Regeln kontextfreier Grammatiken sind von der Form:

$$A \longrightarrow \alpha \quad \text{mit} \quad A \in N \, , \;\; \alpha \in (N \cup T)^*$$

Die Regeln kontextfreier Grammatiken sind von der Form:

$$A \rightarrow \alpha \quad \text{mit} \quad A \in N \,, \quad \alpha \in (N \cup T)^*$$

Beispiel:

$$\begin{array}{lcl} S & \rightarrow & a\,S\,b \\ S & \rightarrow & \epsilon \end{array}$$

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

Die Regeln kontextfreier Grammatiken sind von der Form:

$$A \rightarrow \alpha \quad \text{mit} \quad A \in N \,, \quad \alpha \in (N \cup T)^*$$

Beispiel:

$$\begin{array}{lll} S & \rightarrow & a\, S\, b \\ S & \rightarrow & \epsilon \end{array}$$

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

Konventionen:

- In Beispielen ist die Spezifikation der Nichtterminale und Terminale i.a. implizit:
  - $\rightarrow$ Nichtterminale sind: $A, B, C, ..., \langle\mathsf{exp}\rangle, \langle\mathsf{stmt}\rangle, ...;$
  - $\rightarrow$ Terminale sind: $a, b, c, ..., \mathsf{int}, \mathsf{name}, ...;$

Weitere Beispiele:

$$
\begin{array}{lcl}
S & \to & \langle\mathsf{stmt}\rangle \\
\langle\mathsf{stmt}\rangle & \to & \langle\mathsf{if}\rangle \quad | \quad \langle\mathsf{while}\rangle \quad | \quad \langle\mathsf{rexp}\rangle; \\
\langle\mathsf{if}\rangle & \to & \mathsf{if}\ (\ \langle\mathsf{rexp}\rangle\ )\ \langle\mathsf{stmt}\rangle\ \mathsf{else}\ \langle\mathsf{stmt}\rangle \\
\langle\mathsf{while}\rangle & \to & \mathsf{while}\ (\ \langle\mathsf{rexp}\rangle\ )\ \langle\mathsf{stmt}\rangle \\
\langle\mathsf{rexp}\rangle & \to & \mathsf{int} \quad | \quad \langle\mathsf{lexp}\rangle \quad | \quad \langle\mathsf{lexp}\rangle = \langle\mathsf{rexp}\rangle \quad | \quad \ldots \\
\langle\mathsf{lexp}\rangle & \to & \mathsf{name} \quad | \quad \ldots
\end{array}
$$

Weitere Beispiele:

$$
\begin{array}{lll}
S & \to & \langle\mathsf{stmt}\rangle \\
\langle\mathsf{stmt}\rangle & \to & \langle\mathsf{if}\rangle \quad | \quad \langle\mathsf{while}\rangle \quad | \quad \langle\mathsf{rexp}\rangle; \\
\langle\mathsf{if}\rangle & \to & \mathsf{if}\ (\ \langle\mathsf{rexp}\rangle\ )\ \langle\mathsf{stmt}\rangle\ \mathsf{else}\ \langle\mathsf{stmt}\rangle \\
\langle\mathsf{while}\rangle & \to & \mathsf{while}\ (\ \langle\mathsf{rexp}\rangle\ )\ \langle\mathsf{stmt}\rangle \\
\langle\mathsf{rexp}\rangle & \to & \mathsf{int} \quad | \quad \langle\mathsf{lexp}\rangle \quad | \quad \langle\mathsf{lexp}\rangle = \langle\mathsf{rexp}\rangle \quad | \quad \ldots \\
\langle\mathsf{lexp}\rangle & \to & \mathsf{name} \quad | \quad \ldots
\end{array}
$$

Weitere Konventionen:

- Für jedes Nichtterminal sammeln wir die rechten Regelseiten und listen sie gemeinsam auf :-)
- Die j-te Regel für A können wir durch das Paar (A, j) bezeichnen ($j \geq 0$).

Weitere Grammatiken:

$$E \to E{+}E \mid E{*}E \mid (\,E\,) \mid \mathsf{name} \mid \mathsf{int}$$

$$\begin{array}{lcl} E & \to & E{+}T \mid T \\ T & \to & T{*}F \mid F \\ F & \to & (\,E\,) \mid \mathsf{name} \mid \mathsf{int} \end{array}$$

Die beiden Grammatiken beschreiben die gleiche Sprache ;-)

Weitere Grammatiken:

$$E \rightarrow E{+}E\ ^{0} \quad | \quad E{*}E\ ^{1} \quad | \quad (\ E\ )\ ^{2} \quad | \quad \mathsf{name}\ ^{3} \quad | \quad \mathsf{int}\ ^{4}$$

$$\begin{array}{lcl}
E & \rightarrow & E{+}T\ ^{0} \quad | \quad T\ ^{1} \\
T & \rightarrow & T{*}F\ ^{0} \quad | \quad F\ ^{1} \\
F & \rightarrow & (\ E\ )\ ^{0} \quad | \quad \mathsf{name}\ ^{1} \quad | \quad \mathsf{int}\ ^{2}
\end{array}$$

Die beiden Grammatiken beschreiben die gleiche Sprache ;-)

Grammatiken sind Wortersetzungssysteme.

Die Regeln geben die möglichen Ersetzungsschritte an.

Eine Folge solcher Ersetzungsschritte heißt auch Ableitung.

Grammatiken sind Wortersetzungssysteme.

Die Regeln geben die möglichen Ersetzungsschritte an.

Eine Folge solcher Ersetzungsschritte heißt auch Ableitung.

... im letzten Beispiel:

$$\underline{E}$$

Grammatiken sind Wortersetzungssysteme.

Die Regeln geben die möglichen Ersetzungsschritte an.

Eine Folge solcher Ersetzungsschritte heißt auch Ableitung.

... im letzten Beispiel:

$$\underline{E} \quad \rightarrow \quad \underline{E} + T$$

Grammatiken sind Wortersetzungssysteme.

Die Regeln geben die möglichen Ersetzungsschritte an.

Eine Folge solcher Ersetzungsschritte heißt auch Ableitung.

... im letzten Beispiel:

$$
\begin{array}{lcl}
\underline{E} & \rightarrow & \underline{E} + T \\
 & \rightarrow & \underline{T} + T
\end{array}
$$

Grammatiken sind Wortersetzungssysteme.

Die Regeln geben die möglichen Ersetzungsschritte an.

Eine Folge solcher Ersetzungsschritte heißt auch Ableitung.

... im letzten Beispiel:

$$\begin{array}{lll} \underline{E} & \rightarrow & \underline{E} + T \\ & \rightarrow & \underline{T} + T \\ & \rightarrow & T * \underline{F} + T \end{array}$$

Grammatiken sind Wortersetzungssysteme.

Die Regeln geben die möglichen Ersetzungsschritte an.

Eine Folge solcher Ersetzungsschritte heißt auch Ableitung.

... im letzten Beispiel:

$$
\begin{array}{lll}
\underline{E} & \rightarrow & \underline{E} + T \\
 & \rightarrow & \underline{T} + T \\
 & \rightarrow & T * \underline{F} + T \\
 & \rightarrow & \underline{T} * \mathsf{int} + T
\end{array}
$$

Grammatiken sind Wortersetzungssysteme.

Die Regeln geben die möglichen Ersetzungsschritte an.

Eine Folge solcher Ersetzungsschritte heißt auch Ableitung.

... im letzten Beispiel:

$$\begin{array}{lll} \underline{E} & \rightarrow & \underline{E} + T \\ & \rightarrow & \underline{T} + T \\ & \rightarrow & T * \underline{F} + T \\ & \rightarrow & \underline{T} * \mathsf{int} + T \\ & \rightarrow & \underline{F} * \mathsf{int} + T \end{array}$$

Grammatiken sind Wortersetzungssysteme.

Die Regeln geben die möglichen Ersetzungsschritte an.

Eine Folge solcher Ersetzungsschritte heißt auch Ableitung.

... im letzten Beispiel:

$$
\begin{array}{lll}
\underline{E} & \rightarrow & \underline{E} + T \\
 & \rightarrow & \underline{T} + T \\
 & \rightarrow & T * \underline{F} + T \\
 & \rightarrow & \underline{T} * \mathsf{int} + T \\
 & \rightarrow & \underline{F} * \mathsf{int} + T \\
 & \rightarrow & \mathsf{name} * \mathsf{int} + \underline{T}
\end{array}
$$

Grammatiken sind Wortersetzungssysteme.

Die Regeln geben die möglichen Ersetzungsschritte an.

Eine Folge solcher Ersetzungsschritte heißt auch Ableitung.

... im letzten Beispiel:

$$
\begin{array}{lll}
\underline{E} & \to & \underline{E} + T \\
 & \to & \underline{T} + T \\
 & \to & T * \underline{F} + T \\
 & \to & \underline{T} * \mathsf{int} + T \\
 & \to & \underline{F} * \mathsf{int} + T \\
 & \to & \mathsf{name} * \mathsf{int} + \underline{T} \\
 & \to & \mathsf{name} * \mathsf{int} + \underline{F}
\end{array}
$$

Grammatiken sind Wortersetzungssysteme.

Die Regeln geben die möglichen Ersetzungsschritte an.

Eine Folge solcher Ersetzungsschritte heißt auch Ableitung.

... im letzten Beispiel:

$$
\begin{array}{lll}
\underline{E} & \rightarrow & \underline{E} + T \\
 & \rightarrow & \underline{T} + T \\
 & \rightarrow & T * \underline{F} + T \\
 & \rightarrow & \underline{T} * \mathsf{int} + T \\
 & \rightarrow & \underline{F} * \mathsf{int} + T \\
 & \rightarrow & \mathsf{name} * \mathsf{int} + \underline{T} \\
 & \rightarrow & \mathsf{name} * \mathsf{int} + \underline{F} \\
 & \rightarrow & \mathsf{name} * \mathsf{int} + \mathsf{int}
\end{array}
$$

Formal ist $\longrightarrow$ eine Relation auf Wörtern über $V = N \cup T$, wobei

$$\alpha \longrightarrow \alpha' \quad \text{gdw.} \quad \alpha = \alpha_1 \, A \, \alpha_2 \;\; \wedge \;\; \alpha' = \alpha_1 \, \beta \, \alpha_2 \;\; \text{für ein} \;\; A \longrightarrow \beta \in P$$

Den reflexiven und transitiven Abschluss von $\longrightarrow$ schreiben wir: $\longrightarrow^*$:-)

Formal ist $\longrightarrow$ eine Relation auf Wörtern über $V = N \cup T$, wobei

$$\alpha \longrightarrow \alpha' \quad \text{gdw.} \quad \alpha = \alpha_1\ A\ \alpha_2 \ \wedge\ \alpha' = \alpha_1\ \beta\ \alpha_2 \ \text{ für ein } \ A \longrightarrow \beta \in P$$

Den reflexiven und transitiven Abschluss von $\longrightarrow$ schreiben wir: $\longrightarrow^*$:-)

Bemerkungen:

- Die Relation $\longrightarrow$ hängt von der Grammatik ab ;-)
- Eine Folge von Ersetzungsschritten: $\alpha_0 \longrightarrow \ldots \longrightarrow \alpha_m$ heißt Ableitung.
- In jedem Schritt einer Ableitung können wir:
  - eine Stelle auswählen, wo wir ersetzen wollen, sowie
  - eine Regel, wie wir ersetzen wollen.
- Die von G spezifizierte Sprache ist:

$$\mathcal{L}(G) = \{w \in T^* \mid S \longrightarrow^* w\}$$

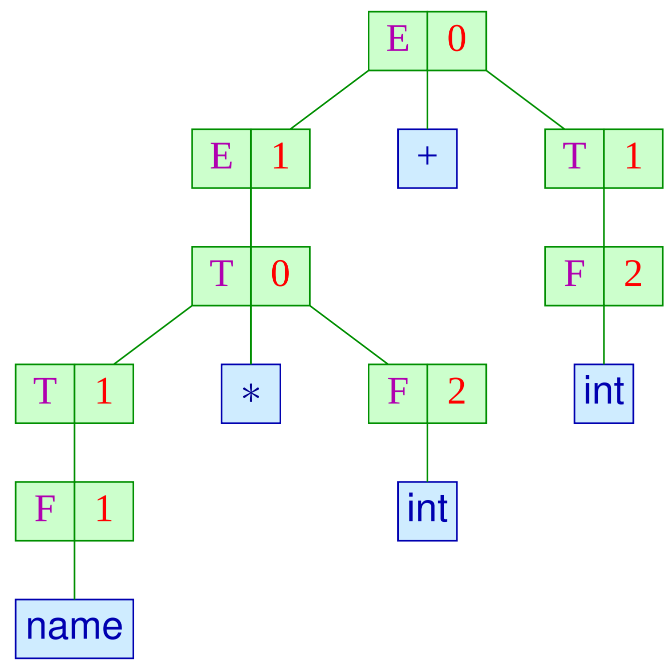
Achtung:

Die Reihenfolge, in der disjunkte Teile abgeleitet werden, ist unerheblich :-)

Ableitungen eines Symbols stellt man als Ableitungsbaum dar :-)

... im Beispiel:

$$
\begin{array}{lll}
\underline{E} & \to^0 & \underline{E} + T \\
 & \to^1 & \underline{T} + T \\
 & \to^0 & T * \underline{F} + T \\
 & \to^2 & \underline{T} * \mathsf{int} + T \\
 & \to^1 & \underline{F} * \mathsf{int} + T \\
 & \to^1 & \mathsf{name} * \mathsf{int} + \underline{T} \\
 & \to^1 & \mathsf{name} * \mathsf{int} + \underline{F} \\
 & \to^2 & \mathsf{name} * \mathsf{int} + \mathsf{int}
\end{array}
$$



Ein Ableitungsbaum für $A \in N$:

innere Knoten:	Regel-Anwendungen;
Wurzel:	Regel-Anwendung für A;
Blätter:	Terminale oder ϵ;

Die Nachfolger von (B, i) entsprechen der rechten Seite der Regel :-)

Ein Ableitungsbaum für $A \in N$:

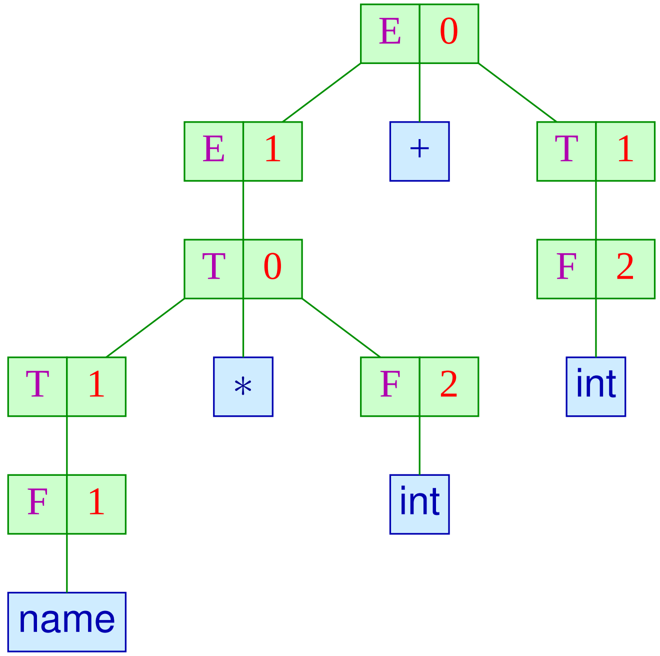
innere Knoten:	Regel-Anwendungen;
Wurzel:	Regel-Anwendung für A;
Blätter:	Terminale oder ϵ;

Die Nachfolger von (B, i) entsprechen der rechten Seite der Regel :-)

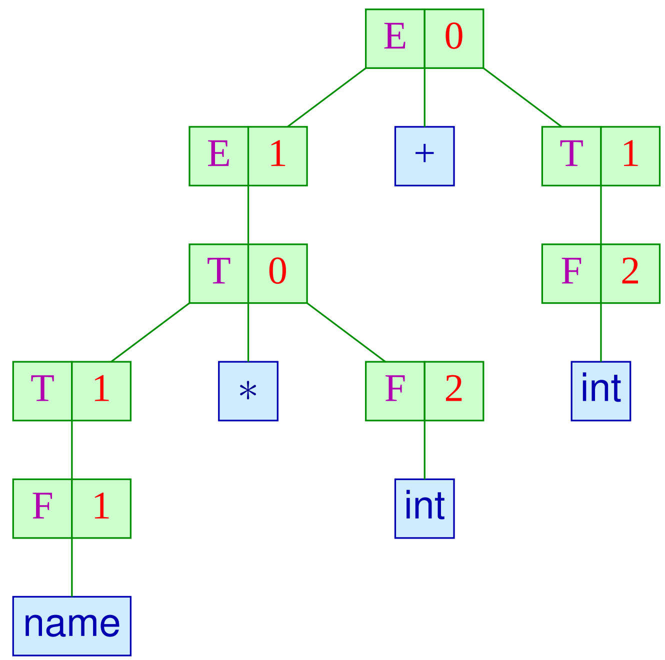
Beachte:

- Neben beliebiger Ableitungen betrachtet man solche, bei denen stets das linkste (bzw. rechteste) Vorkommen eines Nichtterminals ersetzt wird.
- Diese heißen Links- (bzw. Rechts-) Ableitungen und werden durch Index L bzw. R gekennzeichnet.
- Links-(bzw. Rechts-) Ableitungen entsprechen einem links-rechts (bzw. rechts-links) preorder-DFS-Durchlauf durch den Ableitungsbaum :-)
- Reverse Rechts-Ableitungen entsprechen einem links-rechts postorder-DFS-Durchlauf durch den Ableitungsbaum :-))

... im Beispiel:

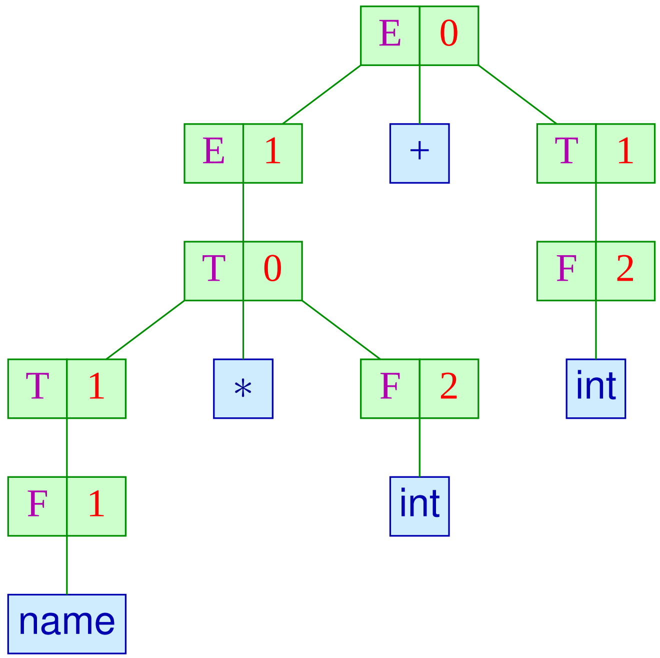


... im Beispiel:



Links-Ableitung: $(E, 0)\ (E, 1)\ (T, 0)\ (T, 1)\ (F, 1)\ (F, 2)\ (T, 1)\ (F, 2)$

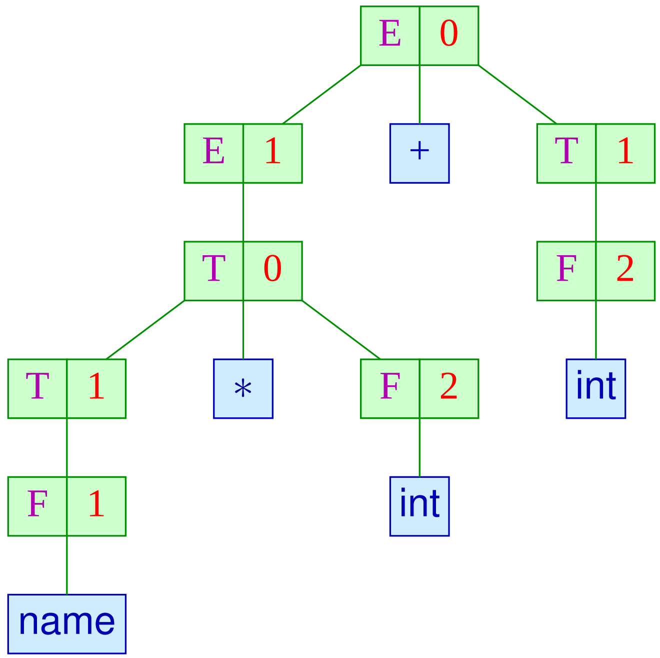
... im Beispiel:



Links-Ableitung: $(E, 0)\ (E, 1)\ (T, 0)\ (T, 1)\ (F, 1)\ (F, 2)\ (T, 1)\ (F, 2)$

Rechts-Ableitung: $(E, 0)\ (T, 1)\ (F, 2)\ (E, 1)\ (T, 0)\ (F, 2)\ (T, 1)\ (F, 1)$

... im Beispiel:



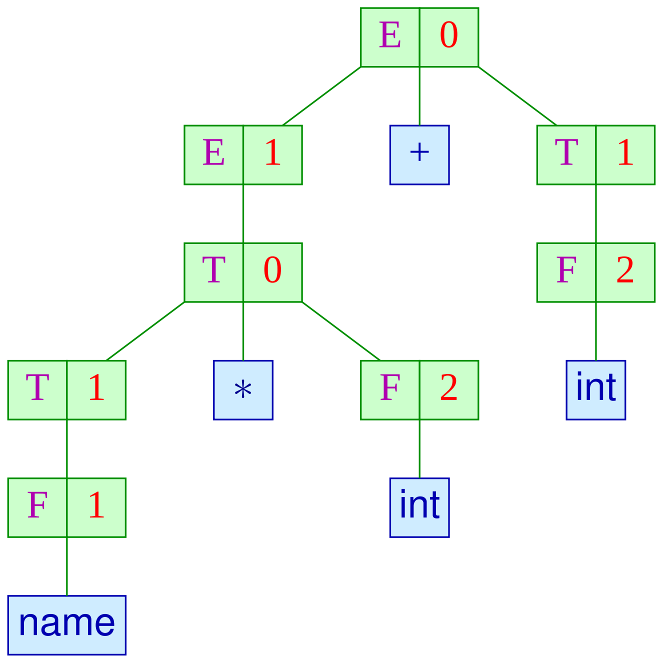
Links-Ableitung: $(E,0)\ (E,1)\ (T,0)\ (T,1)\ (F,1)\ (F,2)\ (T,1)\ (F,2)$

Rechts-Ableitung: $(E,0)\ (T,1)\ (F,2)\ (E,1)\ (T,0)\ (F,2)\ (T,1)\ (F,1)$

Reverse Rechts-Ableitung: $(F,1)\ (T,1)\ (F,2)\ (T,0)\ (E,1)\ (F,2)\ (T,1)\ (E,0)$

Die Konkatenation der Blätter des Ableitungsbaums t bezeichnen wir auch mit $\text{yield}(t)$.

... im Beispiel:



liefert die Konkatenation: $\text{name} * \text{int} + \text{int}$.

Die Grammatik G heißt eindeutig, falls es zu jedem $w \in T^*$ maximal einen Ableitungsbaum t von S gibt mit $\mathsf{yield}(t) = w$:-)

... unsere beiden Grammatiken:

$$E \;\to\; E{+}E^{\,0} \;\mid\; E{*}E^{\,1} \;\mid\; (\,E\,)^{\,2} \;\mid\; \mathsf{name}^{\,3} \;\mid\; \mathsf{int}^{\,4}$$

$$\begin{array}{lll} E & \to & E{+}T^{\,0} \;\mid\; T^{\,1} \\ T & \to & T{*}F^{\,0} \;\mid\; F^{\,1} \\ F & \to & (\,E\,)^{\,0} \;\mid\; \mathsf{name}^{\,1} \;\mid\; \mathsf{int}^{\,2} \end{array}$$

Die zweite ist eindeutig, die erste nicht :-)

Fazit:

- Ein Ableitungsbaum repräsentiert eine mögliche hierarchische Struktur eines Worts.
- Bei Programmiersprachen sind wir nur an Grammatiken interessiert, bei denen die Struktur stets eindeutig ist :-)
- Ableitungsbäume stehen in eins-zu-eins-Korrespondenz mit Links-Ableitungen wie auch (reversen) Rechts-Ableitungen.
- Links-Ableitungen entsprechen einem Topdown-Aufbau des Ableitungsbaums.
- Reverse Rechts-Ableitungen entsprechen einem Bottom-up-Aufbau des Ableitungsbaums.

Fingerübung: überflüssige Nichtterminale und Regeln

$A \in N$ heißt produktiv, falls $A \to^* w$ für ein $w \in T^*$.

$A \in N$ heißt erreichbar, falls $S \to^* \alpha\, A\, \beta$ für geeignete $\alpha, \beta \in (T \cup N)^*$.

Beispiel:

$$\begin{array}{lll} S & \to & a\,B\,B \;\mid\; b\,D \\ A & \to & B\,c \\ B & \to & S\,d \;\mid\; C \\ C & \to & a \\ D & \to & B\,D \end{array}$$

Fingerübung: überflüssige Nichtterminale und Regeln

$A \in N$ heißt produktiv, falls $A \rightarrow^* w$ für ein $w \in T^*$.

$A \in N$ heißt erreichbar, falls $S \rightarrow^* \alpha\, A\, \beta$ für geeignete $\alpha, \beta \in (T \cup N)^*$.

Beispiel:

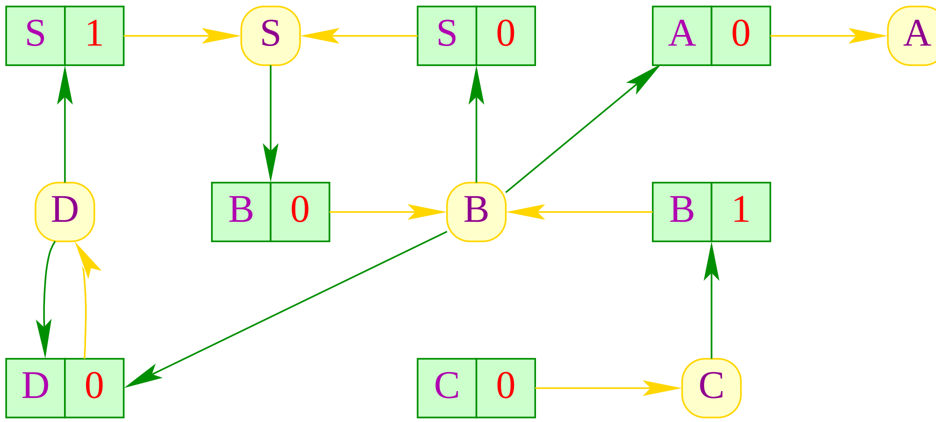
$$
\begin{array}{lcl}
S & \rightarrow & a\,B\,B \quad | \quad b\,D \\
A & \rightarrow & B\,c \\
B & \rightarrow & S\,d \quad | \quad C \\
C & \rightarrow & a \\
D & \rightarrow & B\,D
\end{array}
$$

Produktive Nichtterminale: S, A, B, C

Erreichbare Nichtterminale: S, B, C, D

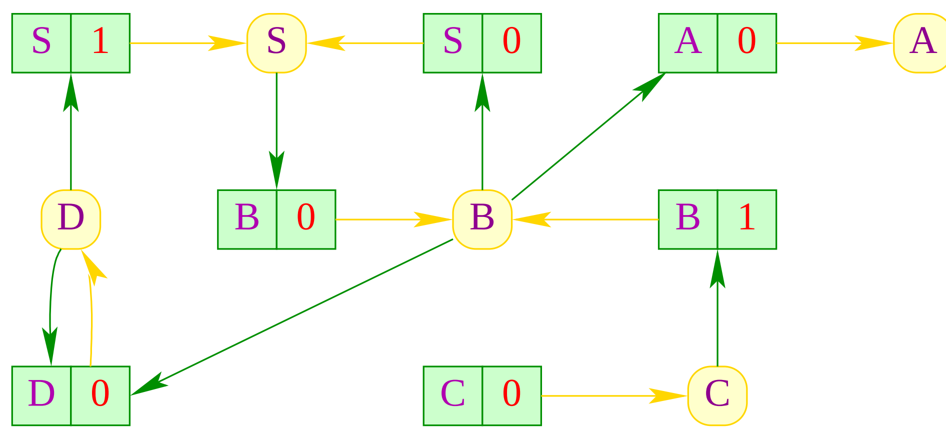
Idee für Produktivität: And-Or-Graph für die Grammatik

... hier:



Idee für Produktivität: And-Or-Graph für die Grammatik

... hier:



And-Knoten: Regeln

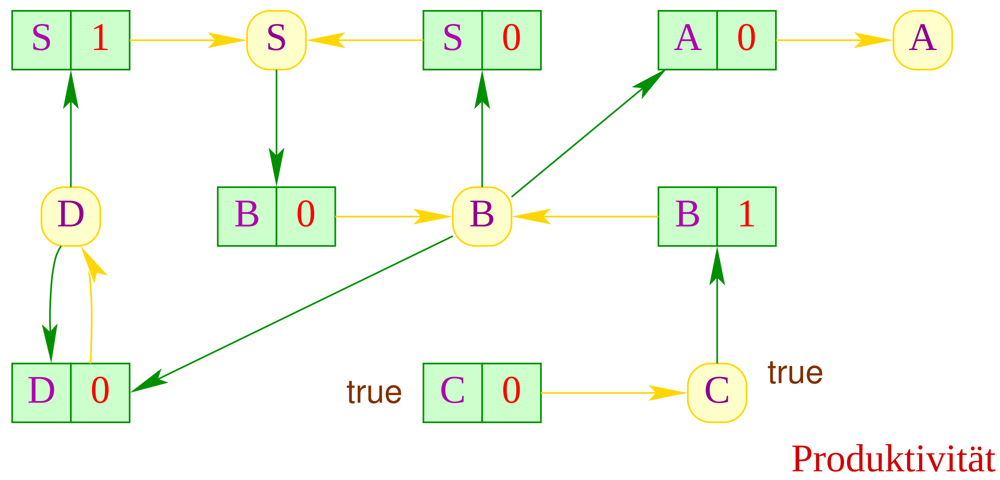
Or-Knoten: Nichtterminale

Kanten: $((B, i), B)$ für alle Regeln (B, i)

$(A, (B, i))$ falls $(B, i) \equiv B \rightarrow \alpha_1 \, A \, \alpha_2$

Idee für Produktivität: And-Or-Graph für die Grammatik

... hier:



And-Knoten: Regeln

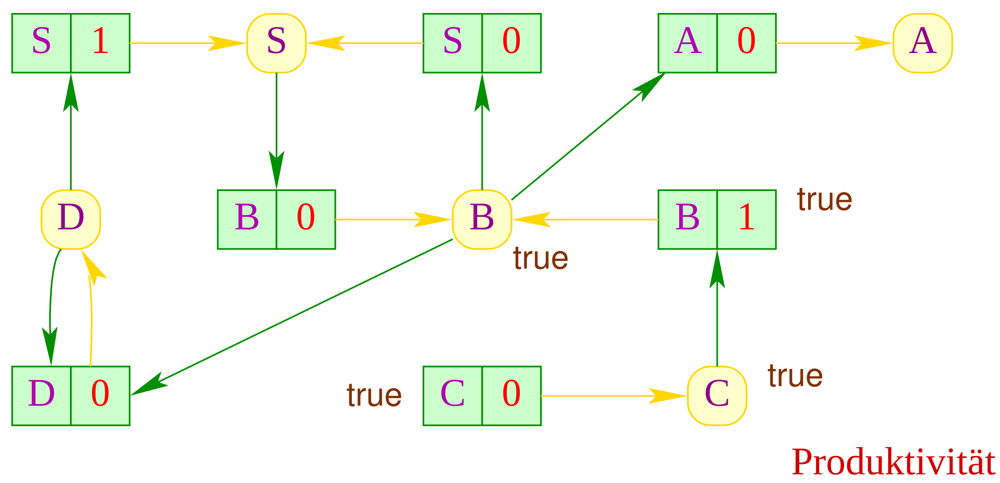
Or-Knoten: Nichtterminale

Kanten: $((B, i), B)$ für alle Regeln (B, i)

$(A, (B, i))$ falls $(B, i) \equiv B \to \alpha_1 \, A \, \alpha_2$

Idee für Produktivität: And-Or-Graph für die Grammatik

... hier:



And-Knoten: Regeln

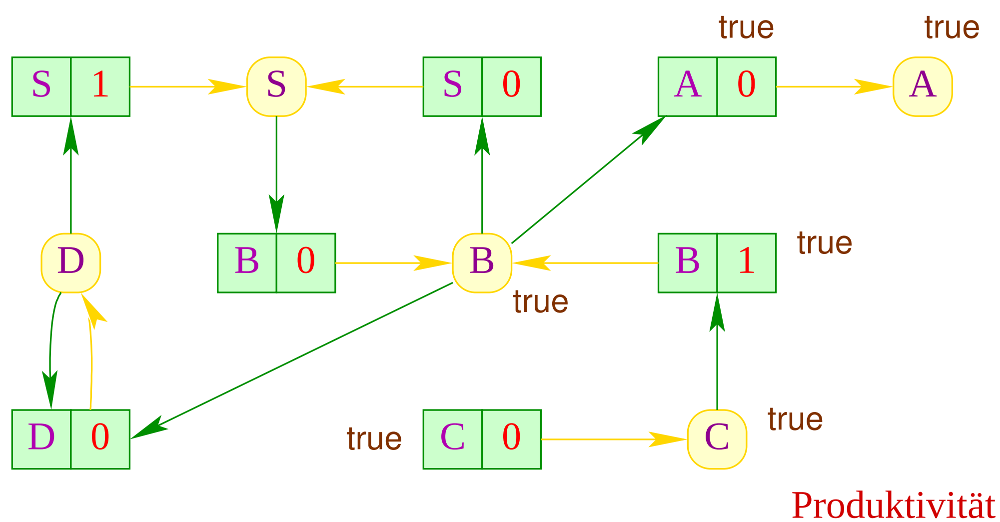
Or-Knoten: Nichtterminale

Kanten: $((B, i), B)$ für alle Regeln (B, i)

$(A, (B, i))$ falls $(B, i) \equiv B \to \alpha_1\ A\ \alpha_2$

Idee für Produktivität: And-Or-Graph für die Grammatik

... hier:



And-Knoten: Regeln

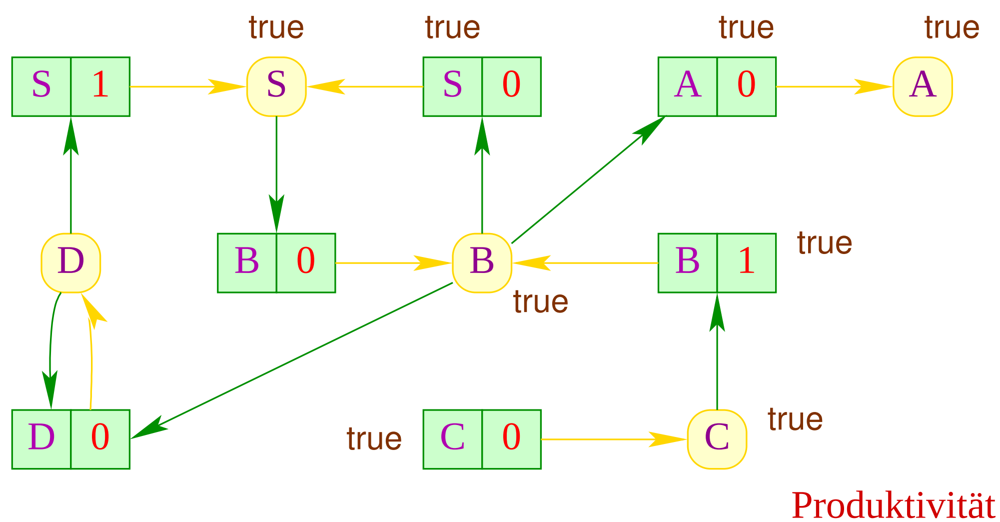
Or-Knoten: Nichtterminale

Kanten: $((B, i), B)$ für alle Regeln (B, i)

$(A, (B, i))$ falls $(B, i) \equiv B \rightarrow \alpha_1 \; A \; \alpha_2$

Idee für Produktivität: And-Or-Graph für die Grammatik

... hier:



And-Knoten: Regeln

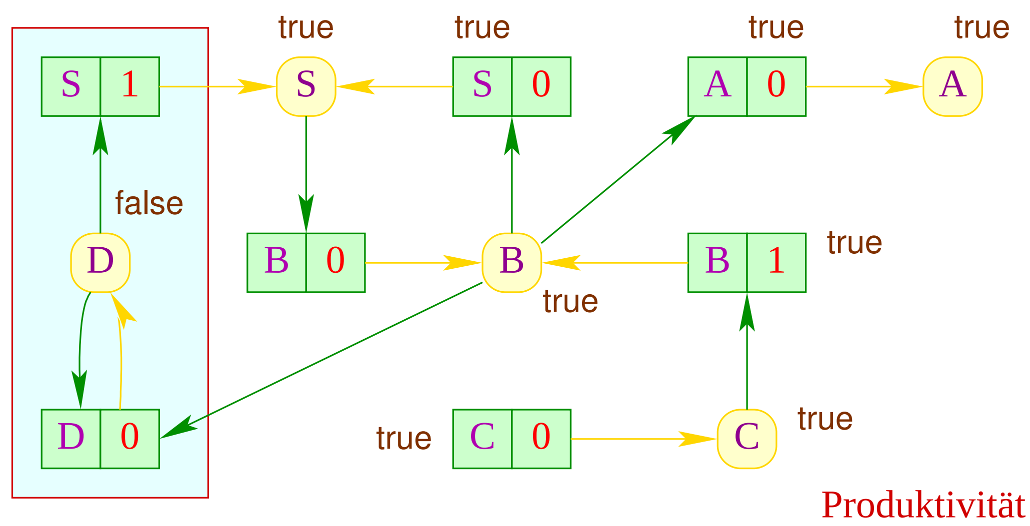
Or-Knoten: Nichtterminale

Kanten: $((B, i), B)$ für alle Regeln (B, i)

$(A, (B, i))$ falls $(B, i) \equiv B \to \alpha_1 \; A \; \alpha_2$

Idee für Produktivität: And-Or-Graph für die Grammatik

... hier:



And-Knoten: Regeln

Or-Knoten: Nichtterminale

Kanten: $((B, i), B)$ für alle Regeln (B, i)

$(A, (B, i))$ falls $(B, i) \equiv B \rightarrow \alpha_1\, A\, \alpha_2$

Algorithmus:

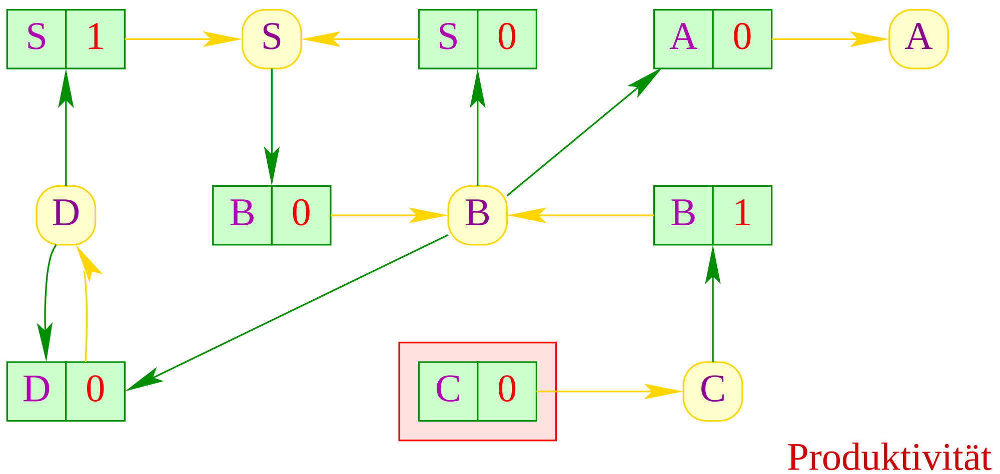
```
2^N   result = ∅;                        //  Ergebnis-Menge
int   count[P];                          //  Zähler für jede Regel
2^P   rhs[N];                            //  Vorkommen in rechten Seiten

forall (A ∈ N)   rhs[A] = ∅;             //  Initialisierung
forall ((A,i) ∈ P)   {                   //
          count[(A,i)] = 0;              //
          init(A,i);                     //  Initialisierung von rhs
          }                              //
...                                      //
```

Die Hilfsfunktion init zählt die Nichtterminal-Vorkommen in der rechten Seite und vermerkt sie in der Datenstruktur rhs :-)

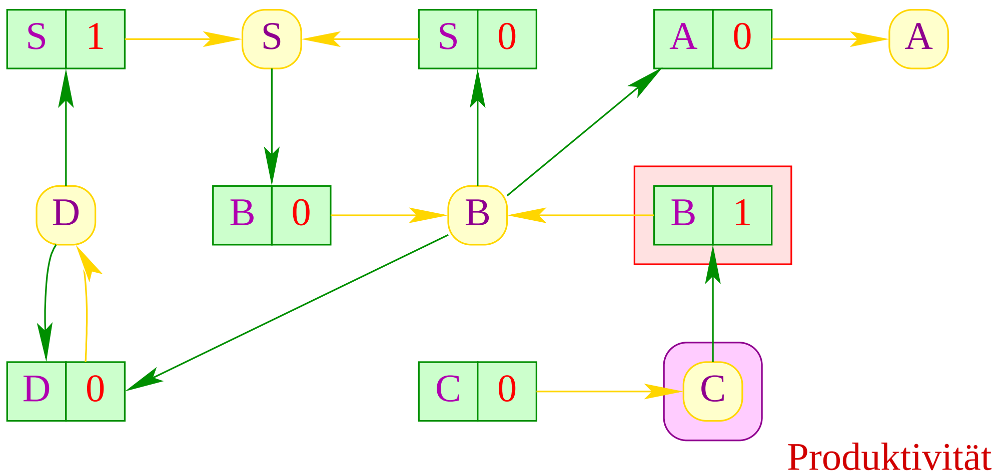
```
...                                                    //
2^P   W = {r | count[r] = 0};                          //   Workset
while (W ≠ ∅)   {                                      //
        (A, i) = extract(W);                           //
        if (A ∉ result)   {                            //
                result = result ∪ {A};                 //
                forall (r ∈ rhs[A])   {                //
                        count[r]−−;                    //
                        if (count[r] == 0)  W = W ∪ {r};  //
                        }                              //   end of   forall
                }                                      //   end of   if
        }                                              //   end of   while
```

Die Menge W verwaltet die Regeln, deren rechte Seiten nur produktive Nichtterminale enthalten :-))

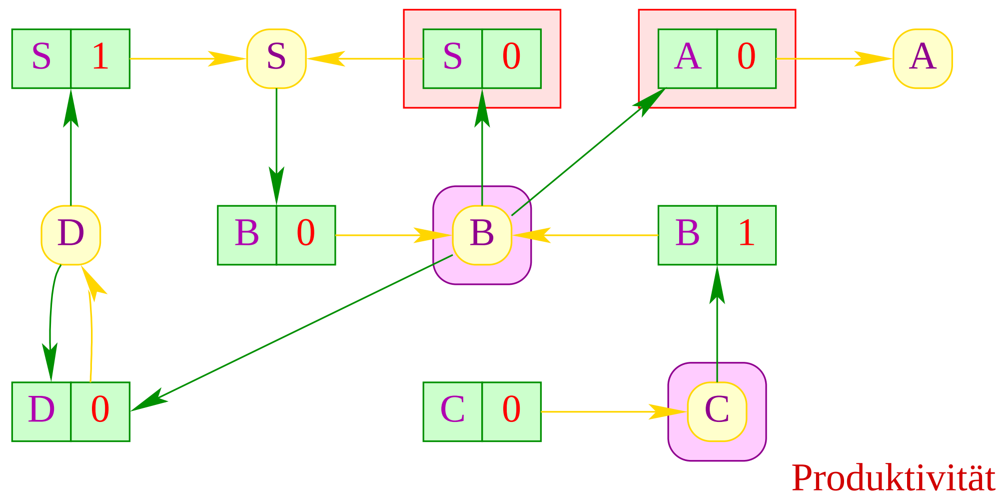
... im Beispiel:



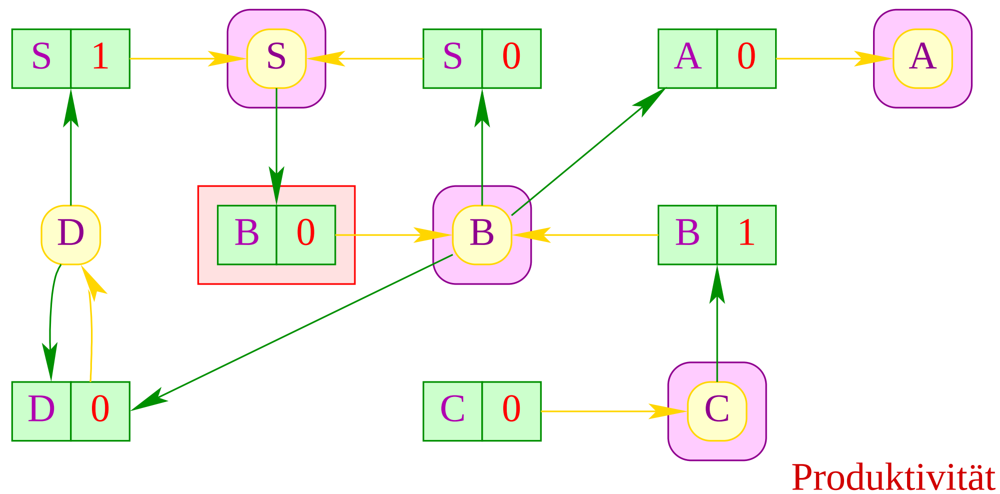
... im Beispiel:



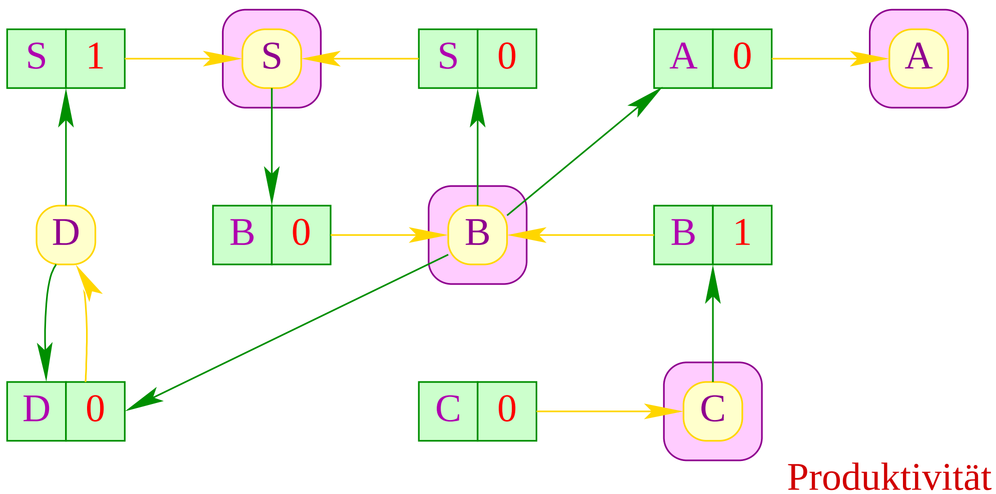
... im Beispiel:



... im Beispiel:



... im Beispiel:



Laufzeit:

- Die Initialisierung der Datenstrukturen erfordert lineare Laufzeit.
- Jede Regel wird maximal einmal in W eingefügt.
- Jedes A wird maximal einmal in result eingefügt.

$\Longrightarrow$ Der Gesamtaufwand ist linear in der Größe der Grammatik :-)

Korrektheit:

- Falls A in der j-ten Iteration der **while**-Schleife in result eingefügt, gibt es einen Ableitungsbaum für A der Höhe maximal $j-1$:-)
- Für jeden Ableitungsbaum wird die Wurzel einmal in W eingefügt :-)

Diskussion:

- Um den Test $(A \in \mathsf{result})$ einfach zu machen, repräsentiert man die Menge $\mathsf{result})$ durch ein Array.
- W wie auch die Mengen $\mathsf{rhs}[A]$ wird man dagegen als Listen repräsentieren :-)

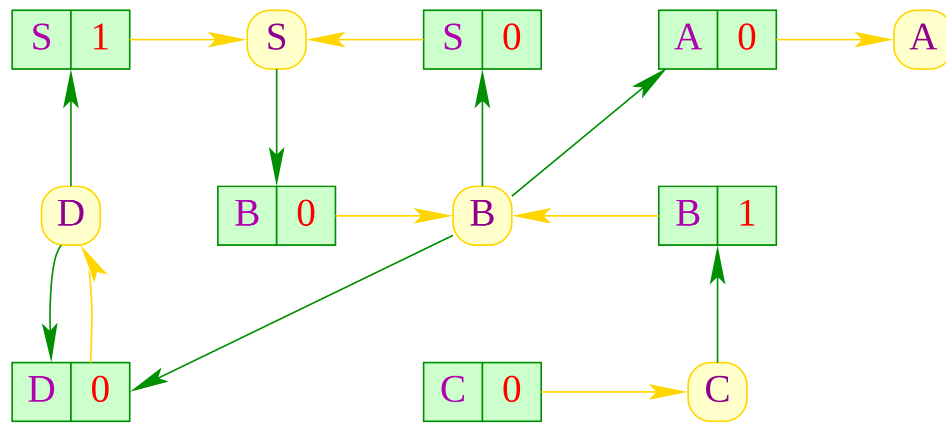
Diskussion:

- Um den Test $(A \in \mathsf{result})$ einfach zu machen, repräsentiert man die Menge $\mathsf{result})$ durch ein Array.
- W wie auch die Mengen $\mathsf{rhs}[A]$ wird man dagegen als Listen repräsentieren :-)

- Der Algorithmus funktioniert auch, um kleinste Lösungen von Booleschen Ungleichungssystemen zu bestimmen :-)
- Die Ermittlung der produktiven Nichtterminale kann benutzt werden, um festzustellen, ob $\mathcal{L}(G) \neq \emptyset$ ist ($\rightarrow$ Leerheitsproblem)

Idee für Erreichbarkeit: Abhängigkeits-Graph

... hier:

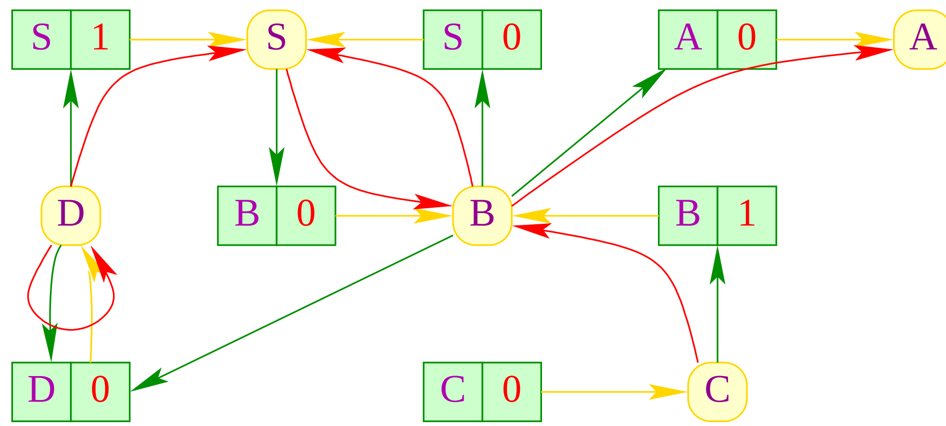


Knoten: Nichtterminale

Kanten: (A, B) falls $B \to \alpha_1\, A\, \alpha_2 \;\in\; P$

Idee für Erreichbarkeit: Abhängigkeits-Graph

... hier:

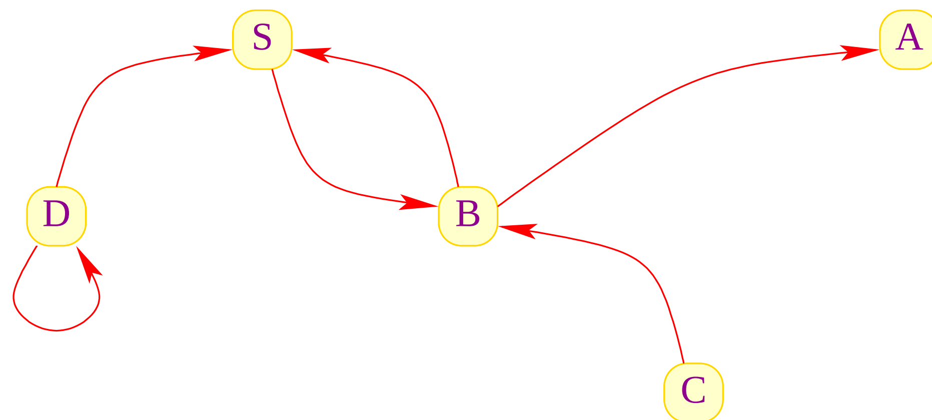


Knoten: Nichtterminale

Kanten: (A, B) falls $B \rightarrow \alpha_1\, A\, \alpha_2 \;\in\; P$

Idee für Erreichbarkeit: Abhängigkeits-Graph

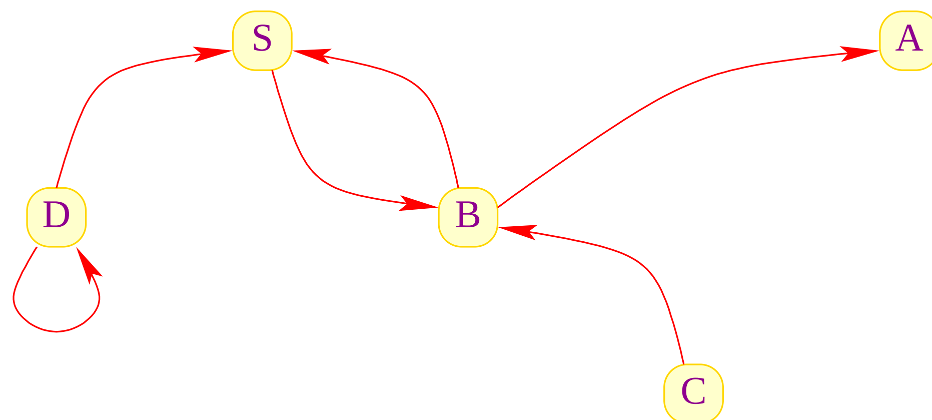
... hier:



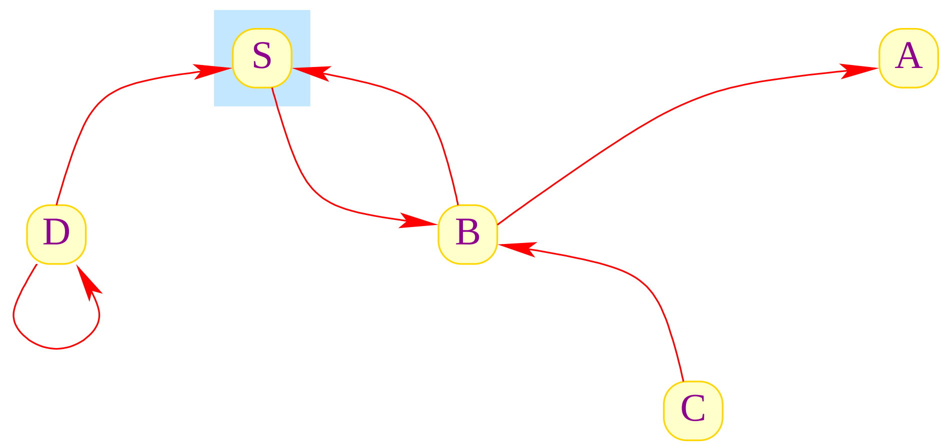
Knoten: Nichtterminale

Kanten: (A, B) falls $B \rightarrow \alpha_1\, A\, \alpha_2 \;\in\; P$

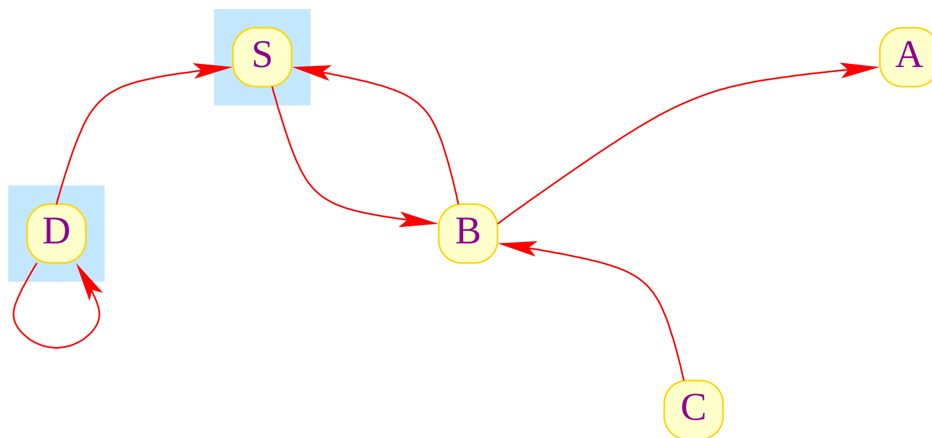
Das Nichtterminal A ist erreichbar, falls es im Abhängigkeitsgraphen einen Pfad von A nach S gibt :-)



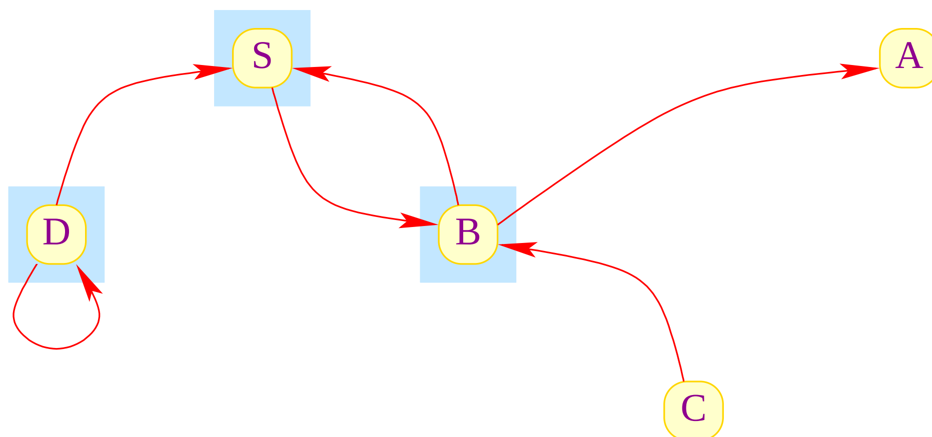
Das Nichtterminal A ist erreichbar, falls es im Abhängigkeitsgraphen einen Pfad von A nach S gibt :-)



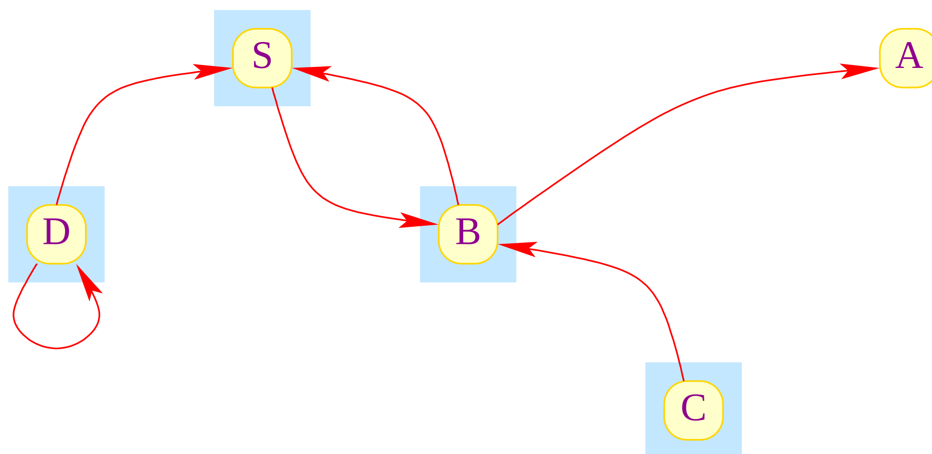
Das Nichtterminal A ist erreichbar, falls es im Abhängigkeitsgraphen einen Pfad von A nach S gibt :-)



Das Nichtterminal A ist erreichbar, falls es im Abhängigkeitsgraphen einen Pfad von A nach S gibt :-)



Das Nichtterminal A ist erreichbar, falls es im Abhängigkeitsgraphen einen Pfad von A nach S gibt :-)



Fazit:

- Erreichbarkeit in gerichteten Graphen kann mithilfe von DFS in linearer Zeit berechnet werden.
- Damit kann die Menge aller erreichbaren und produktiven Nichtterminale in linearer Zeit berechnet werden :-)

Fazit:

- Erreichbarkeit in gerichteten Graphen kann mithilfe von DFS in linearer Zeit berechnet werden.
- Damit kann die Menge aller erreichbaren und produktiven Nichtterminale in linearer Zeit berechnet werden :-)

Eine Grammatik G heißt reduziert, wenn alle Nichtterminale von G sowohl produktiv wie erreichbar sind ...

Fazit:

- Erreichbarkeit in gerichteten Graphen kann mithilfe von DFS in linearer Zeit berechnet werden.
- Damit kann die Menge aller erreichbaren und produktiven Nichtterminale in linearer Zeit berechnet werden :-)

Eine Grammatik G heißt reduziert, wenn alle Nichtterminale von G sowohl produktiv wie erreichbar sind ...

Satz

Zu jeder kontextfreien Grammatik $G = (N, T, P, S)$ mit $\mathcal{L}(G) \neq \emptyset$ kann in linearer Zeit eine reduzierte Grammatik G' konstruiert werden mit

$$\mathcal{L}(G) = \mathcal{L}(G')$$

Konstruktion:

1. Schritt:

Berechne die Teilmenge $N' \subseteq N$ aller produktiven und erreichbaren Nichtterminale von G .

Da $\mathcal{L}(G) \neq \emptyset$ ist insbesondere $S \in N'$:-)

Konstruktion:

1. Schritt:

Berechne die Teilmenge $N' \subseteq N$ aller produktiven und erreichbaren Nichtterminale von G .

Da $\mathcal{L}(G) \neq \emptyset$ ist insbesondere $S \in N'$:-)

2. Schritt:

Konstruiere: $P' = \{A \to \alpha \in P \mid A \in N' \wedge \alpha \in (N' \cup T)^*\}$

Konstruktion:

1. Schritt:

Berechne die Teilmenge $N' \subseteq N$ aller produktiven und erreichbaren Nichtterminale von G .

Da $\mathcal{L}(G) \neq \emptyset$ ist insbesondere $S \in N'$:-)

2. Schritt:

Konstruiere: $P' = \{A \to \alpha \in P \mid A \in N' \wedge \alpha \in (N' \cup T)^*\}$

Ergebnis: $G' = (N', T, P', S)$:-)

... im Beispiel:

$$
\begin{array}{lcl}
S & \rightarrow & a\,B\,B \quad | \quad b\,D \\
A & \rightarrow & B\,c \\
B & \rightarrow & S\,d \quad | \quad C \\
C & \rightarrow & a \\
D & \rightarrow & B\,D
\end{array}
$$

... im Beispiel:

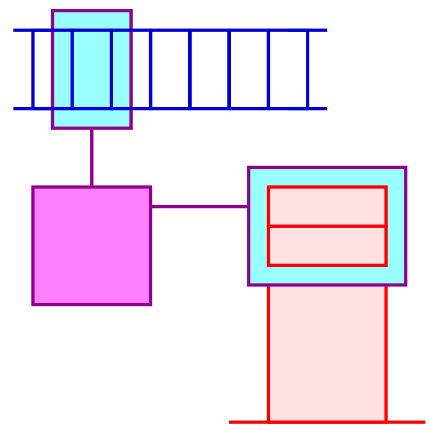
$$
\begin{array}{lcl}
S & \rightarrow & a\,B\,B \quad | \quad b\,D \\
A & \rightarrow & B\,c \\
B & \rightarrow & S\,d \quad | \quad C \\
C & \rightarrow & a \\
D & \rightarrow & B\,D
\end{array}
$$

... im Beispiel:

$$\begin{array}{lcl} S & \longrightarrow & a\,B\,B \\ \\ B & \longrightarrow & S\,d \quad | \quad C \\ C & \longrightarrow & a \end{array}$$

2.2 Grundlagen: Kellerautomaten

Durch kontextfreie Grammatiken spezifizierte Sprachen können durch Kellerautomaten (Pushdown Automata) akzeptiert werden:

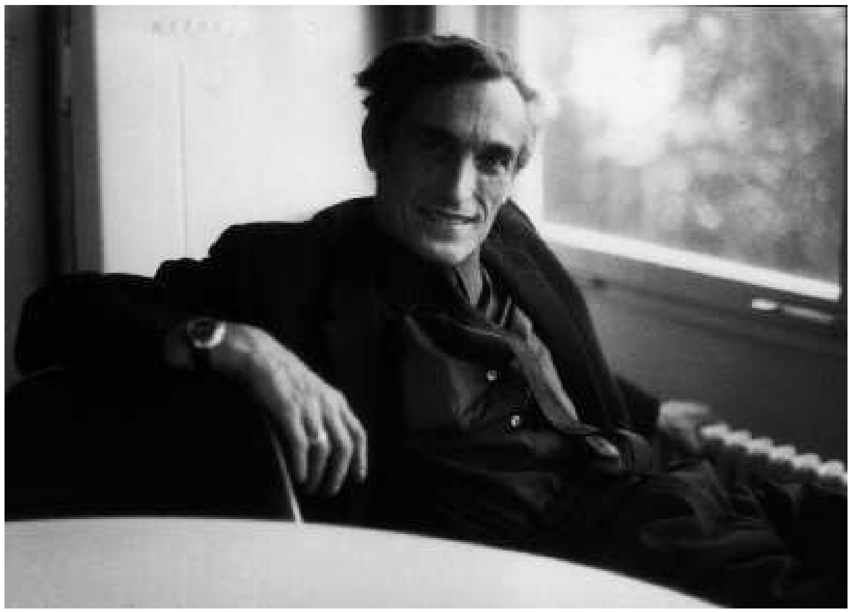


Der Keller wird z.B. benötigt, um korrekte Klammerung zu überprüfen :-)

Friedrich L. Bauer, TUM

Klaus Samelson, TUM

Kellerautomaten für kontextfreie Sprachen wurden erstmals vorgeschlagen von Michel Schützenberger und Antony G. Öttinger:



Marcel-Paul Schützenberger
(1920-1996), Paris

Antony G. Öttinger, Präsident der
ACM 1966-68

Beispiel:

Zustände: $0, 1, 2$

Anfangszustand: 0

Endzustände: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

Beispiel:

Zustände:	$0, 1, 2$
Anfangszustand:	0
Endzustände:	$0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

Achtung:

- Wir unterscheiden nicht zwischen Kellersymbolen und Zuständen :-)
- Das rechteste / oberste Kellersymbol repräsentiert den Zustand :-)
- Jeder Übergang liest / modifiziert einen oberen Abschnitt des Kellers :-)

Formal definieren wir deshalb einen Kellerautomaten (PDA) als ein Tupel:
$M = (Q, T, \delta, q_0, F)$ wobei:

- Q eine endliche Menge von Zuständen;
- T das Eingabe-Alphabet;
- $q_0 \in Q$ der Anfangszustand;
- $F \subseteq Q$ die Menge der Endzustände und
- $\delta \subseteq Q^+ \times (T \cup \{\epsilon\}) \times Q^*$ eine endliche Menge von Übergängen ist (das Programm :-)

Formal definieren wir deshalb einen Kellerautomaten (PDA) als ein Tupel:
$M = (Q, T, \delta, q_0, F)$ wobei:

- Q eine endliche Menge von Zuständen;
- T das Eingabe-Alphabet;
- $q_0 \in Q$ der Anfangszustand;
- $F \subseteq Q$ die Menge der Endzustände und
- $\delta \subseteq Q^+ \times (T \cup \{\epsilon\}) \times Q^*$ eine endliche Menge von Übergängen ist (das Programm :-)

Mithilfe der Übergänge definieren wir Berechnungen von Kellerautomaten :-)

Der jeweilige Berechnungszustand (die aktuelle Konfiguration) ist ein Paar:

$$(\gamma, w) \in Q^* \times T^*$$

bestehend aus dem Kellerinhalt und dem noch zu lesenden Input.

... im Beispiel:

Zustände: $0, 1, 2$

Anfangszustand: 0

Endzustände: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

... im Beispiel:

Zustände: $0, 1, 2$

Anfangszustand: 0

Endzustände: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

$$(0\,, \quad a\,a\,a\,b\,b\,b)$$

... im Beispiel:

Zustände: $0, 1, 2$

Anfangszustand: 0

Endzustände: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

$$(0\,,\quad a\,a\,a\,b\,b\,b) \quad \vdash \qquad (1\,1\,,\quad a\,a\,b\,b\,b)$$

... im Beispiel:

Zustände: $0, 1, 2$

Anfangszustand: 0

Endzustände: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

$$
\begin{array}{lll}
(0\,, \quad a\,a\,a\,b\,b\,b) & \vdash & (1\,1\,, \quad a\,a\,b\,b\,b) \\
 & \vdash & (1\,1\,1\,, \quad a\,b\,b\,b)
\end{array}
$$

... im Beispiel:

Zustände: $0, 1, 2$

Anfangszustand: 0

Endzustände: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

$$
\begin{array}{rcl}
(0\,, \quad a\,a\,a\,b\,b\,b) & \vdash & (1\,1\,, \quad a\,a\,b\,b\,b) \\
& \vdash & (1\,1\,1\,, \quad a\,b\,b\,b) \\
& \vdash & (1\,1\,1\,1\,, \quad b\,b\,b)
\end{array}
$$

... im Beispiel:

Zustände: $0, 1, 2$

Anfangszustand: 0

Endzustände: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

$$
\begin{array}{rcl}
(0\,, \quad a\,a\,a\,b\,b\,b) & \vdash & (1\,1\,, \quad a\,a\,b\,b\,b) \\
& \vdash & (1\,1\,1\,, \quad a\,b\,b\,b) \\
& \vdash & (1\,1\,1\,1\,, \quad b\,b\,b) \\
& \vdash & (1\,1\,2\,, \quad b\,b)
\end{array}
$$

... im Beispiel:

Zustände: $0, 1, 2$

Anfangszustand: 0

Endzustände: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

$$
\begin{array}{rcr}
(0\,,\quad a\,a\,a\,b\,b\,b) & \vdash & (1\,1\,,\quad a\,a\,b\,b\,b) \\
& \vdash & (1\,1\,1\,,\quad a\,b\,b\,b) \\
& \vdash & (1\,1\,1\,1\,,\quad b\,b\,b) \\
& \vdash & (1\,1\,2\,,\quad b\,b) \\
& \vdash & (1\,2\,,\quad b)
\end{array}
$$

... im Beispiel:

Zustände: $0, 1, 2$

Anfangszustand: 0

Endzustände: $0, 2$

0	a	11
1	a	11
11	b	2
12	b	2

$$
\begin{array}{lcr}
(0\,, \quad a\,a\,a\,b\,b\,b) & \vdash & (1\,1\,, \quad a\,a\,b\,b\,b) \\
& \vdash & (1\,1\,1\,, \quad a\,b\,b\,b) \\
& \vdash & (1\,1\,1\,1\,, \quad b\,b\,b) \\
& \vdash & (1\,1\,2\,, \quad b\,b) \\
& \vdash & (1\,2\,, \quad b) \\
& \vdash & (2\,, \quad \epsilon)
\end{array}
$$

Ein Berechnungsschritt wird durch die Relation $\quad \vdash \;\subseteq\; (Q^* \times T^*)^2$ beschrieben, wobei

$$(\alpha\,\gamma,\, x\,w) \vdash (\alpha\,\gamma',\, w) \quad \text{für} \quad (\gamma,\, x,\, \gamma') \,\in\, \delta$$

Ein Berechnungsschritt wird durch die Relation $\vdash \subseteq (Q^* \times T^*)^2$ beschrieben, wobei

$$(\alpha\,\gamma,\, x\,w) \vdash (\alpha\,\gamma',\, w) \quad \text{für} \quad (\gamma,\, x,\, \gamma') \in \delta$$

Bemerkungen:

- Die Relation $\vdash$ hängt natürlich vom Kellerautomaten M ab :-)
- Die reflexive und transitive Hülle von $\vdash$ bezeichnen wir mit $\vdash^*$.
- Dann ist die von M akzeptierte Sprache:

$$\mathcal{L}(M) = \{w \in T^* \mid \exists\, f \in F : \ (q_0, w) \vdash^* (f, \epsilon)\}$$

Ein Berechnungsschritt wird durch die Relation $\vdash \ \subseteq \ (Q^* \times T^*)^2$ beschrieben, wobei

$$(\alpha\,\gamma,\, x\,w) \vdash (\alpha\,\gamma',\, w) \quad \text{für} \quad (\gamma,\, x,\, \gamma') \in \delta$$

Bemerkungen:

- Die Relation $\vdash$ hängt natürlich vom Kellerautomaten M ab :-)
- Die reflexive und transitive Hülle von $\vdash$ bezeichnen wir mit $\vdash^*$.
- Dann ist die von M akzeptierte Sprache:

$$\mathcal{L}(M) \ = \ \{w \in T^* \mid \exists\, f \in F :\ (q_0, w) \vdash^* (f, \epsilon)\}$$

Wir akzeptieren also mit Endzustand und leerem Keller :-)

Der Kellerautomat M heißt deterministisch, falls jede Konfiguration maximal eine Nachfolge-Konfiguration hat.

Das ist genau dann der Fall wenn für verschiedene Übergänge (γ_1, x, γ_2), $(\gamma_1', x', \gamma_2') \in \delta$ gilt:

Ist γ_1 ein Suffix von γ_1', dann muss $x \neq x' \wedge x \neq \epsilon \neq x'$ sein.

Der Kellerautomat M heißt deterministisch, falls jede Konfiguration maximal eine Nachfolge-Konfiguration hat.

Das ist genau dann der Fall wenn für verschiedene Übergänge (γ_1, x, γ_2) , $(\gamma_1', x', \gamma_2') \in \delta$ gilt:

Ist γ_1 ein Suffix von γ_1', dann muss $x \neq x' \wedge x \neq \epsilon \neq x'$ sein.

... im Beispiel:

0	a	11
1	a	11
11	b	2
12	b	2

ist das natürlich der Fall :-))

Satz

Zu jeder kontextfreien Grammatik $G = (N, T, P, S)$ kann ein PDA M konstruiert werden mit $\mathcal{L}(G) = \mathcal{L}(M)$.

Der Satz ist für uns so wichtig, dass wir zwei Konstruktionen angeben :-)

Satz

Zu jeder kontextfreien Grammatik $G = (N, T, P, S)$ kann ein PDA M konstruiert werden mit $\mathcal{L}(G) = \mathcal{L}(M)$.

Der Satz ist für uns so wichtig, dass wir zwei Konstruktionen angeben :-)

Konstruktion 1:

- Die Eingabe wird sukzessive auf den Keller geschiftet.
- Liegt oben auf dem Keller eine vollständige rechte Seite (ein Handle) vor, wird dieses durch die zugehörige linke Seite ersetzt (reduziert) :-)

Beispiel:

$$
\begin{array}{lll}
S & \rightarrow & A\,B \\
A & \rightarrow & a \\
B & \rightarrow & b
\end{array}
$$

Der Kellerautomat:

Zustände: q_0, f, a, b, A, B, S;

Anfangszustand: q_0

Endzustand: f

q_0	a	$q_0\,a$
a	ϵ	A
A	b	$A\,b$
b	ϵ	B
$A\,B$	ϵ	S
$q_0\,S$	ϵ	f

Allgemein konstruieren wir einen Automaten $M_G^{(1)} = (Q, T, \delta, q_0, F)$ mit:

- $Q = T \cup N \cup \{q_0, f\}$ $\quad (q_0, f$ neu);
- $F = \{f\}$;
- Übergänge:

$$
\begin{array}{lllll}
\delta & = & \{(q, x, q\, x) \mid q \in Q, x \in T\} \;\cup & // & \text{Shift-Übergänge} \\
 & & \{(q\, \alpha, \epsilon, q\, A) \mid q \in Q, A \to \alpha \; \in P\} \;\cup & // & \text{Reduce-Übergänge} \\
 & & \{(q_0\, S, \epsilon, f)\} & // & \text{Abschluss} \quad \text{:-)}
\end{array}
$$

Allgemein konstruieren wir einen Automaten $M_G^{(1)} = (Q, T, \delta, q_0, F)$ mit:

- $Q = T \cup N \cup \{q_0, f\}$ $\quad (q_0, f$ neu);
- $F = \{f\}$;
- Übergänge:

$$\begin{array}{lll} \delta & = & \{(q, x, q\,x) \mid q \in Q, x \in T\} \;\cup \qquad // \text{ Shift-Übergänge} \\ & & \{(q\,\alpha, \epsilon, q\,A) \mid q \in Q, A \to \alpha \in P\} \;\cup \qquad // \text{ Reduce-Übergänge} \\ & & \{(q_0\,S, \epsilon, f)\} \qquad // \text{ Abschluss :-)} \end{array}$$

Eine Beispiel-Berechnung:

$$\begin{array}{rrr} (q_0, \; a\,b) & \vdash \; (q_0\,a, \; b) & \vdash \; (q_0\,A, \; b) \\ & \vdash \; (q_0\,A\,b, \; \epsilon) & \vdash \; (q_0\,A\,B, \; \epsilon) \\ & \vdash \; (q_0\,S, \; \epsilon) & \vdash \; (f, \; \epsilon) \end{array}$$

Allgemein konstruieren wir einen Automaten $M_G^{(1)} = (Q, T, \delta, q_0, F)$ mit:

- $Q = T \cup N \cup \{q_0, f\}$ $\quad (q_0, f$ neu);
- $F = \{f\}$;
- Übergänge:

$$\begin{array}{lll} \delta & = & \{(q, x, q\,x) \mid q \in Q, x \in T\} \;\cup \qquad // \text{ Shift-Übergänge} \\ & & \{(q\,\alpha, \epsilon, q\,A) \mid q \in Q, A \to \alpha \in P\} \;\cup \qquad // \text{ Reduce-Übergänge} \\ & & \{(q_0\,S, \epsilon, f)\} \qquad // \text{ Abschluss :-)} \end{array}$$

Eine Beispiel-Berechnung:

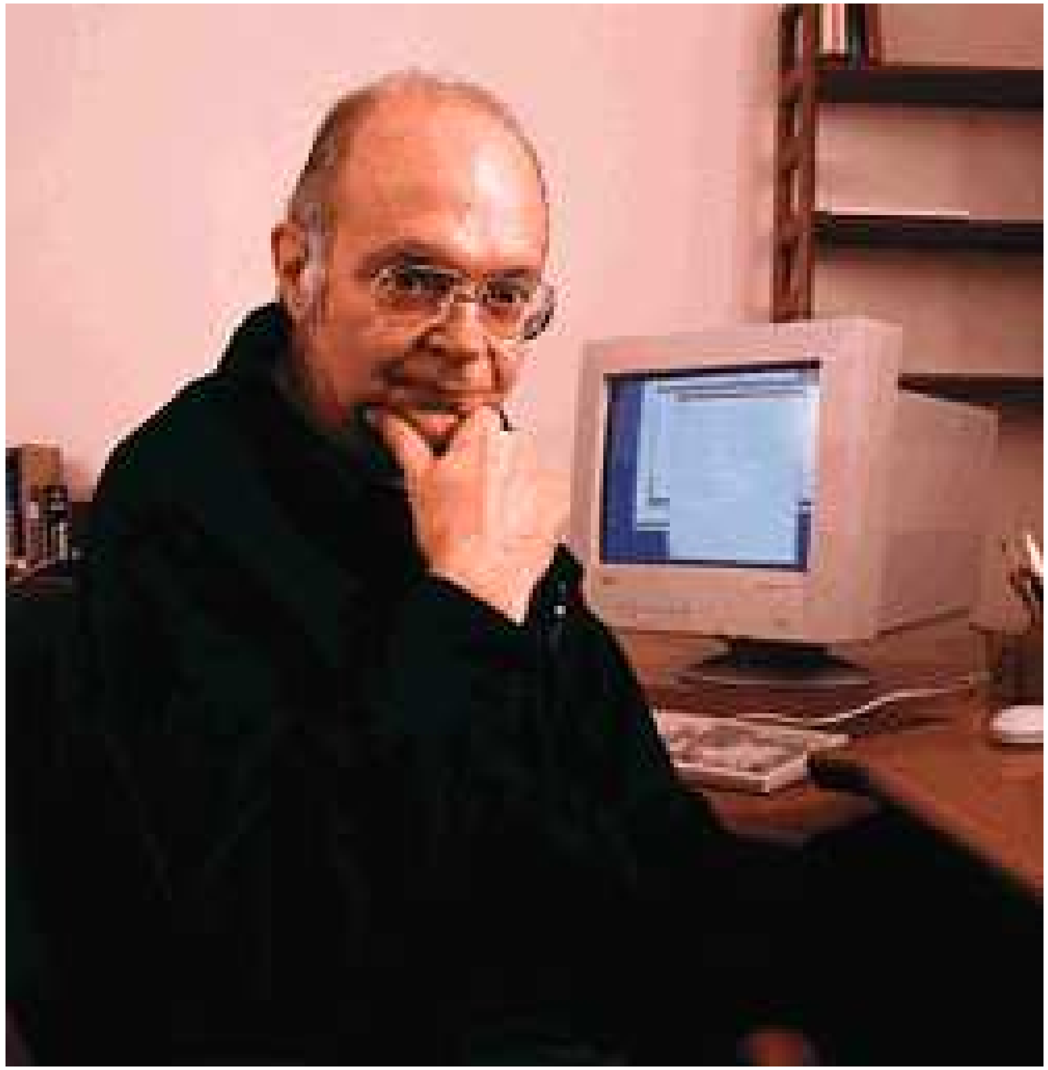
$$\begin{array}{llll} (q_0, \; a\,b) & \vdash & (q_0\,\boxed{a}, \; b) & \vdash \; (q_0\,A, \; b) \\ & \vdash & (q_0\,A\,\boxed{b}, \; \epsilon) & \vdash \; (q_0\,\boxed{A\,B}, \; \epsilon) \\ & \vdash & (q_0\,S, \; \epsilon) & \vdash \; (f, \; \epsilon) \end{array}$$

Offenbar gilt:

- Die Folge der Reduktionen entspricht einer reversen Rechtsableitung für die Eingabe :-)
- Zur Korrektheit zeigt man, dass für jedes q gilt:

$$(q, w) \vdash^* (q\, A, \epsilon) \qquad \text{gdw.} \qquad A \rightarrow^* w$$

- Der Kellerautomat $M_G^{(1)}$ ist i.a. nicht-deterministisch :-(
- Um ein deterministisches Parse-Verfahren zu erhalten, muss man die Reduktionsstellen identifizieren $\Longrightarrow$ LR-Parsing



Donald E. Knuth, Standford

Konstruktion 2: Item-Kellerautomat

- Rekonstruiere eine Linksableitung.
- Expandiere Nichtterminale mithilfe einer Regel.
- Verifiziere sukzessive, dass die gewählte Regel mit der Eingabe übereinstimmt.

  $\Longrightarrow$ Die Zustände sind jetzt Items.
- Ein Item ist eine Regel mit Punkt:

$$[A \to \alpha \bullet \beta]\,, \qquad A \to \alpha\,\beta \;\in\; P$$

Der Punkt gibt an, wieweit die Regel bereits abgearbeitet wurde :-)

Unser Beispiel:

$$S \quad \to \quad A\,B \qquad A \quad \to \quad a \qquad B \quad \to \quad b$$

Wir fügen eine Regel: $S' \to S$ hinzu ;-)

Dann konstruieren wir:

Anfangszustand: $[S' \to \bullet\ S]$

Endzustand: $[S' \to S\,\bullet]$

$[S' \to \bullet\ S]$	ϵ	$[S' \to \bullet\ S]\ [S \to \bullet\ A\,B]$
$[S \to \bullet\ A\,B]$	ϵ	$[S \to \bullet\ A\,B]\ [A \to \bullet\ a]$
$[A \to \bullet\ a]$	a	$[A \to a\,\bullet]$
$[S \to \bullet\ A\,B]\ [A \to a\,\bullet]$	ϵ	$[S \to A\ \bullet\ B]$
$[S \to A\ \bullet\ B]$	ϵ	$[S \to A\ \bullet\ B]\ [B \to \bullet\ b]$
$[B \to \bullet\ b]$	b	$[B \to b\,\bullet]$
$[S \to A\ \bullet\ B]\ [B \to b\,\bullet]$	ϵ	$[S \to A\,B\,\bullet]$
$[S' \to \bullet\ S]\ [S \to A\,B\,\bullet]$	ϵ	$[S' \to S\,\bullet]$

Der Item-Kellerautomat $M_G^{(2)}$ hat drei Arten von Übergängen:

Expansionen: $([A \to \alpha \bullet B\, \beta], \epsilon, [A \to \alpha \bullet B\, \beta]\, [B \to \bullet\, \gamma])$ für $A \to \alpha\, B\, \beta,\ B \to \gamma \in P$

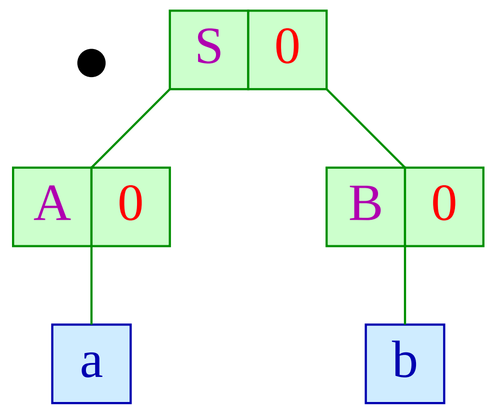
Shifts: $([A \to \alpha \bullet a\, \beta], a, [A \to \alpha\, a \bullet \beta])$ für $A \to \alpha\, a\, \beta \in P$

Reduce: $([A \to \alpha \bullet B\, \beta]\, [B \to \gamma \bullet], \epsilon, [A \to \alpha\, B \bullet \beta])$ für $A \to \alpha\, B\, \beta,\ B \to \gamma \in P$

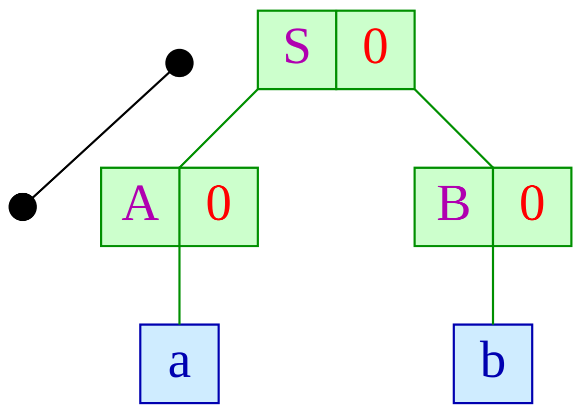
Items der Form: $[A \to \alpha \bullet]$ heißen auch vollständig :-)

Der Item-Kellerautomat schiebt den Punkt einmal um den Ableitungsbaum herum ...

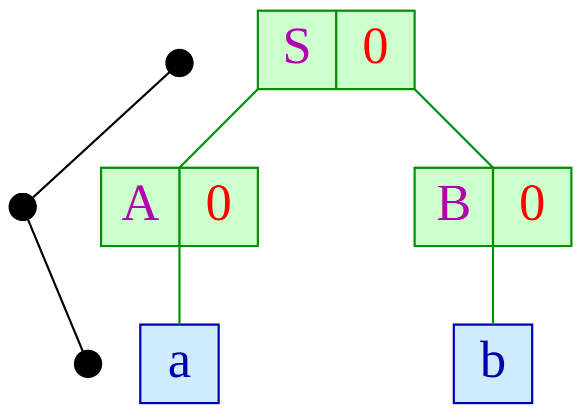
... im Beispiel:



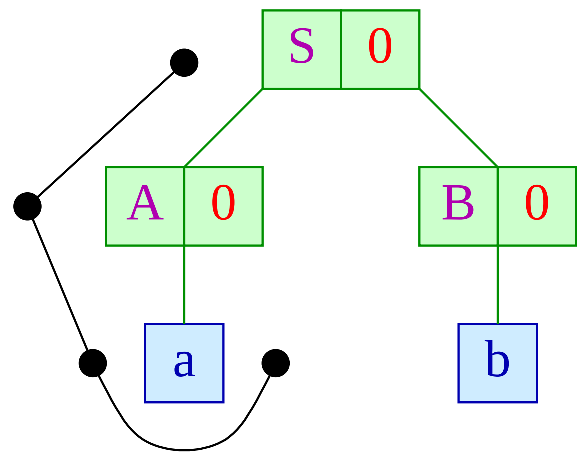
... im Beispiel:



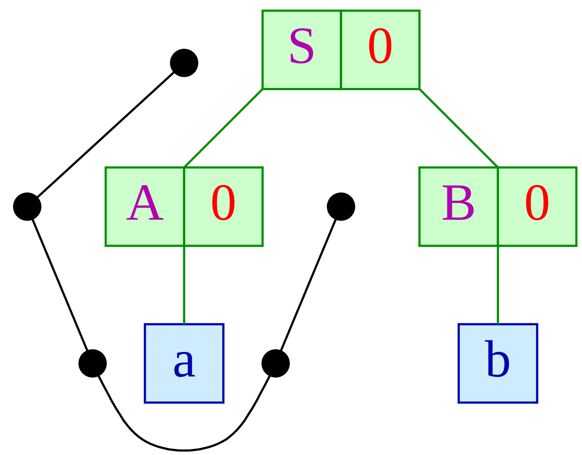
... im Beispiel:



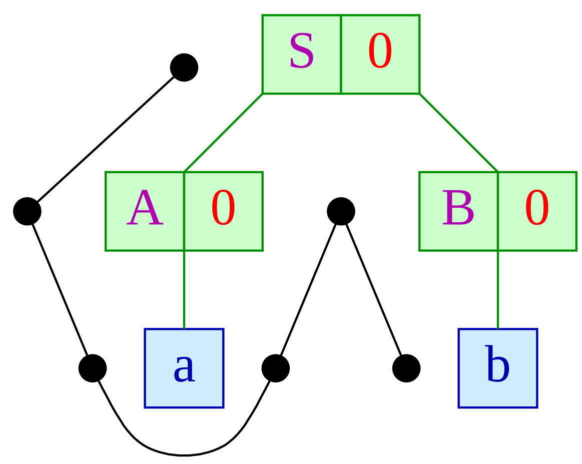
... im Beispiel:



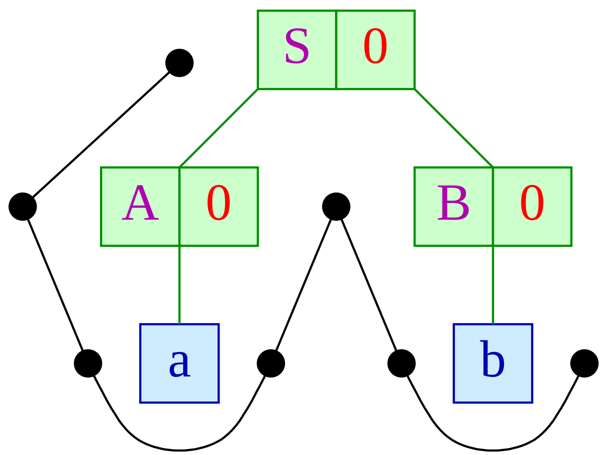
... im Beispiel:



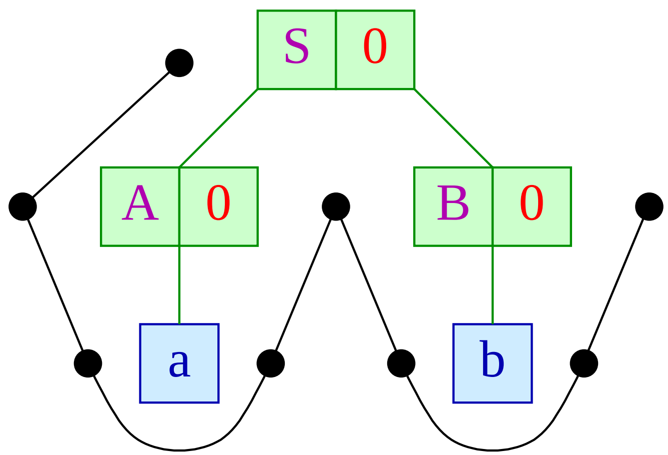
... im Beispiel:



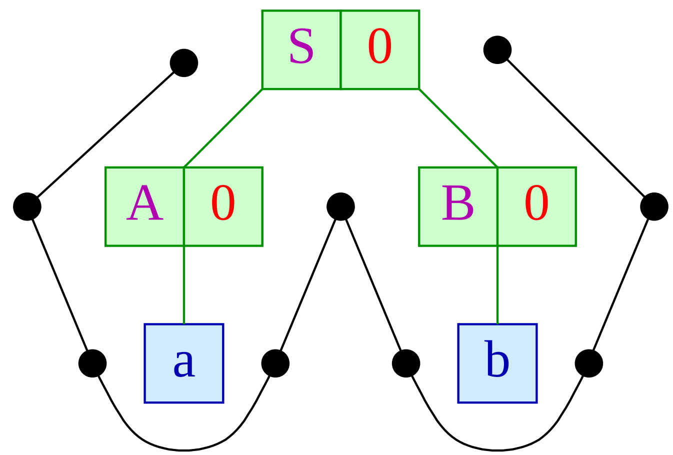
... im Beispiel:



... im Beispiel:



... im Beispiel:



Diskussion:

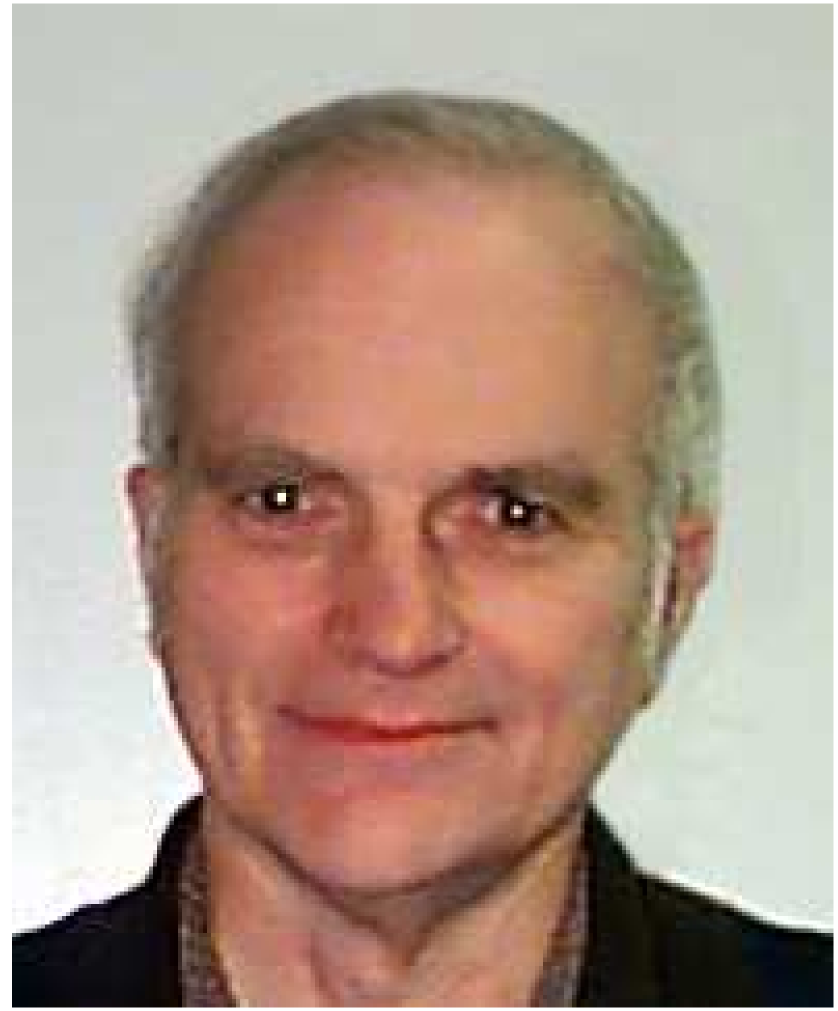
- Die Expansionen einer Berechnung bilden eine Linksableitung :-)
- Leider muss man bei den Expansionen nichtdeterministisch zwischen verschiedenen Regeln auswählen :-(

- Zur Korrektheit der Konstruktion zeigt man, dass für jedes Item $[A \to \alpha \bullet B\, \beta]$ gilt:

$$([A \to \alpha \bullet B\, \beta], w) \vdash^* ([A \to \alpha\, B \bullet \beta], \epsilon) \qquad \text{gdw.} \qquad B \to^* w$$

- LL-Parsing basiert auf dem Item-Kellerautomaten und versucht, die Expansionen durch Vorausschau deterministisch zu machen ...

Philip M. Lewis, SUNY



Richard E. Stearns, SUNY

Beispiel: $S \rightarrow \epsilon \quad | \quad a\,S\,b$

Die Übergänge des zugehörigen Item-Kellerautomat:

0	$[S' \rightarrow \bullet\, S]$	ϵ	$[S' \rightarrow \bullet\, S]\,[S \rightarrow \bullet]$
1	$[S' \rightarrow \bullet\, S]$	ϵ	$[S' \rightarrow \bullet\, S]\,[S \rightarrow \bullet\, a\,S\,b]$
2	$[S \rightarrow \bullet\, a\,S\,b]$	a	$[S \rightarrow a \bullet S\,b]$
3	$[S \rightarrow a \bullet S\,b]$	ϵ	$[S \rightarrow a \bullet S\,b]\,[S \rightarrow \bullet]$
4	$[S \rightarrow a \bullet S\,b]$	ϵ	$[S \rightarrow a \bullet S\,b]\,[S \rightarrow \bullet\, a\,S\,b]$
5	$[S \rightarrow a \bullet S\,b]\,[S \rightarrow \bullet]$	ϵ	$[S \rightarrow a\,S \bullet b]$
6	$[S \rightarrow a \bullet S\,b]\,[S \rightarrow a\,S\,b \bullet]$	ϵ	$[S \rightarrow a\,S \bullet b]$
7	$[S \rightarrow a\,S \bullet b]$	b	$[S \rightarrow a\,S\,b \bullet]$
8	$[S' \rightarrow \bullet\, S]\,[S \rightarrow \bullet]$	ϵ	$[S' \rightarrow S \bullet]$
9	$[S' \rightarrow \bullet\, S]\,[S \rightarrow a\,S\,b \bullet]$	ϵ	$[S' \rightarrow S \bullet]$

Konflikte gibt es zwischen den Übergängen $(0, 1)$ bzw. zwischen $(3, 4)$ – die sich durch Betrachten des nächsten Zeichens lösen ließen :-)

2.3 Vorausschau-Mengen

Für eine Menge $L \subseteq T^*$ definieren wir:

$$\mathsf{First}_k(L) = \{u \in L \mid |u| < k\} \cup \{u \in T^k \mid \exists\, v \in T^* : \; uv \in L\}$$

Beispiel:

ϵ
$a\,b$
$a\,a\,b\,b$
$a\,a\,a\,b\,b\,b$

2.3 Vorausschau-Mengen

Für eine Menge $L \subseteq T^*$ definieren wir:

$$\mathsf{First}_k(L) = \{u \in L \mid |u| < k\} \cup \{u \in T^k \mid \exists\, v \in T^* : \; uv \in L\}$$

Beispiel:

ϵ
$a\,b$
$a\,a$
$a\,a$

die Präfixe der Länge 2 :-)

Rechenregeln:

$\mathsf{First}_k(\_)$ ist verträglich mit Vereinigung und Konkatenation:

$$\begin{array}{lll}
\mathsf{First}_k(\emptyset) & = & \emptyset \\
\mathsf{First}_k(L_1 \;\cup\; L_2) & = & \mathsf{First}_k(L_1) \;\cup\; \mathsf{First}_k(L_2) \\
\mathsf{First}_k(L_1 \;\cdot\; L_2) & = & \mathsf{First}_k(\mathsf{First}_k(L_1) \;\cdot\; \mathsf{First}_k(L_2)) \\
 & := & \mathsf{First}_k(L_1) \;\odot\; \mathsf{First}_k(L_2)
\end{array}$$

k – Konkatenation

Beachte:

- Die Menge $\mathbb{D}_k = 2^{T^{\leq k}}$ ist endlich :-)
- Die Operation: $\odot : \mathbb{D}_k \times \mathbb{D}_k \to \mathbb{D}_k$ ist distributiv in jedem Argument:

$$\begin{array}{lllllll}
L \odot \emptyset & = & \emptyset & \qquad & L \odot (L_1 \cup L_2) & = & (L \odot L_1) \cup (L \odot L_2) \\
\emptyset \odot L & = & \emptyset & \qquad & (L_1 \cup L_2) \odot L & = & (L_1 \odot L) \cup (L_2 \odot L)
\end{array}$$

Für $\alpha \in (N \cup T)^*$ sind wir interessiert an der Menge:

$$\mathsf{First}_k(\alpha) = \mathsf{First}_k(\{w \in T^* \mid \alpha \rightarrow^* w\})$$

Für $k \geq 1$ gilt:

$$\begin{array}{lcll} \mathsf{First}_k(x) & = & \{x\} & \text{für} \quad x \in T \cup \{\epsilon\} \\ \mathsf{First}_k(\alpha_1\,\alpha_2) & = & \mathsf{First}_k(\alpha_1) \odot \mathsf{First}_k(\alpha_2) & \end{array}$$

Für $\alpha \in (N \cup T)^*$ sind wir interessiert an der Menge:

$$\mathsf{First}_k(\alpha) = \mathsf{First}_k(\{w \in T^* \mid \alpha \rightarrow^* w\})$$

Für $k \geq 1$ gilt:

$$\begin{array}{lcll} \mathsf{First}_k(x) & = & \{x\} & \text{für} \quad x \in T \cup \{\epsilon\} \\ \mathsf{First}_k(\alpha_1\,\alpha_2) & = & \mathsf{First}_k(\alpha_1) \odot \mathsf{First}_k(\alpha_2) & \end{array}$$

Frage: Wie berechnet man $\mathsf{First}_k(A)$??

Für $\alpha \in (N \cup T)^*$ sind wir interessiert an der Menge:

$$\text{First}_k(\alpha) = \text{First}_k(\{w \in T^* \mid \alpha \rightarrow^* w\})$$

Für $k \geq 1$ gilt:

$$\begin{array}{lcll} \text{First}_k(x) & = & \{x\} & \text{für} \quad x \in T \cup \{\epsilon\} \\ \text{First}_k(\alpha_1\, \alpha_2) & = & \text{First}_k(\alpha_1) \odot \text{First}_k(\alpha_2) & \end{array}$$

Frage: Wie berechnet man $\text{First}_k(A)$??

Idee: Stelle ein Ungleichungssystem auf!

Beispiel: $k = 2$

$$\begin{array}{lllll} E & \rightarrow & E + T & \mid & T \\ T & \rightarrow & T * F & \mid & F \\ F & \rightarrow & (\ E\ ) & \mid & \textsf{name} \quad \mid \quad \textsf{int} \end{array}$$

Jede Regel gibt Anlass zu einer Inklusionsbeziehung:

$$\begin{array}{lllllll} \textsf{First}_2(E) & \supseteq & \textsf{First}_2(E+T) & \quad & \textsf{First}_2(E) & \supseteq & \textsf{First}_2(T) \\ \textsf{First}_2(T) & \supseteq & \textsf{First}_2(T*F) & & \textsf{First}_2(T) & \supseteq & \textsf{First}_2(F) \\ \textsf{First}_2(F) & \supseteq & \textsf{First}_2((\ E\ )) & & \textsf{First}_2(F) & \supseteq & \{\textsf{name}, \textsf{int}\} \end{array}$$

Beispiel: $k = 2$

$$\begin{array}{lcllll} E & \to & E+T & \mid & T & \\ T & \to & T*F & \mid & F & \\ F & \to & (\ E\ ) & \mid & \mathsf{name} & \mid\ \mathsf{int} \end{array}$$

Jede Regel gibt Anlass zu einer Inklusionsbeziehung:

$$\begin{array}{lcllcl} \mathsf{First}_2(E) & \supseteq & \mathsf{First}_2(E+T) & \mathsf{First}_2(E) & \supseteq & \mathsf{First}_2(T) \\ \mathsf{First}_2(T) & \supseteq & \mathsf{First}_2(T*F) & \mathsf{First}_2(T) & \supseteq & \mathsf{First}_2(F) \\ \mathsf{First}_2(F) & \supseteq & \mathsf{First}_2((\ E\ )) & \mathsf{First}_2(F) & \supseteq & \{\mathsf{name}, \mathsf{int}\} \end{array}$$

Eine Inklusion $\mathsf{First}_2(E) \supseteq \mathsf{First}_2(E+T)$ kann weiter vereinfacht werden zu:

$$\mathsf{First}_2(E) \supseteq \mathsf{First}_2(E) \odot \{+\} \odot \mathsf{First}_2(T)$$

Insgesamt erhalten wir das Ungleichungssystem:

$$\begin{array}{lll} \mathsf{First}_2(E) & \supseteq & \mathsf{First}_2(E) \odot \{+\} \odot \mathsf{First}_2(T) \\ \mathsf{First}_2(T) & \supseteq & \mathsf{First}_2(T) \odot \{*\} \odot \mathsf{First}_2(F) \\ \mathsf{First}_2(F) & \supseteq & \{(\} \odot \mathsf{First}_2(E) \odot \{)\} \end{array} \qquad \begin{array}{lll} \mathsf{First}_2(E) & \supseteq & \mathsf{First}_2(T) \\ \mathsf{First}_2(T) & \supseteq & \mathsf{First}_2(F) \\ \mathsf{First}_2(F) & \supseteq & \{\mathsf{name}, \mathsf{int}\} \end{array}$$

Insgesamt erhalten wir das Ungleichungssystem:

$$\begin{array}{lll} \mathsf{First}_2(E) & \supseteq & \mathsf{First}_2(E) \odot \{+\} \odot \mathsf{First}_2(T) \\ \mathsf{First}_2(T) & \supseteq & \mathsf{First}_2(T) \odot \{*\} \odot \mathsf{First}_2(F) \\ \mathsf{First}_2(F) & \supseteq & \{(\} \odot \mathsf{First}_2(E) \odot \{)\} \end{array} \qquad \begin{array}{lll} \mathsf{First}_2(E) & \supseteq & \mathsf{First}_2(T) \\ \mathsf{First}_2(T) & \supseteq & \mathsf{First}_2(F) \\ \mathsf{First}_2(F) & \supseteq & \{\mathsf{name}, \mathsf{int}\} \end{array}$$

Allgemein:

$$\mathsf{First}_k(A) \quad \supseteq \quad \mathsf{First}_k(X_1) \odot \ldots \odot \mathsf{First}_k(X_m)$$

für jede Regel $A \to X_1 \ldots X_m \quad \in \quad P$ mit $X_i \in T \cup N$.

Gesucht:

- möglichst kleine Lösung (??)
- Algorithmus, der diese berechnet :-)

... im Beispiel:

$$\begin{array}{lll} \text{First}_2(E) & \supseteq & \text{First}_2(E) \odot \{+\} \odot \text{First}_2(T) \\ \text{First}_2(T) & \supseteq & \text{First}_2(T) \odot \{*\} \odot \text{First}_2(F) \\ \text{First}_2(F) & \supseteq & \{(\} \odot \text{First}_2(E) \odot \{)\} \end{array} \qquad \begin{array}{lll} \text{First}_2(E) & \supseteq & \text{First}_2(T) \\ \text{First}_2(T) & \supseteq & \text{First}_2(F) \\ \text{First}_2(F) & \supseteq & \{\text{name}, \text{int}\} \end{array}$$

... hat die Lösung:

E	name, int, (name, (int, ((, name $*$, int $*$, name $+$, int $+$
T	name, int, (name, (int, ((, name $*$, int $*$
F	name, int, (name, (int, ((

Beobachtung:

- Die Menge $\mathbb{D}_k$ der möglichen Werte für $\textsf{First}_k(A)$ bilden einen vollständigen Verband :-)

- Die Operatoren auf den rechten Seiten der Ungleichungen sind monoton, d.h. verträglich mit "$\subseteq$" :-)

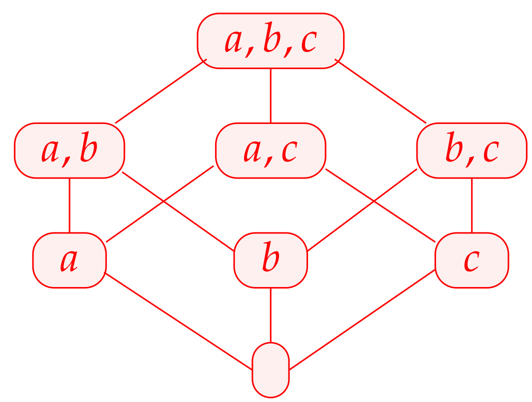
Exkurs: Vollständige Verbände

Eine Menge $\mathbb{D}$ mit einer Relation $\sqsubseteq \subseteq \mathbb{D} \times \mathbb{D}$ ist eine Halbordnung falls für allea $a, b, c \in \mathbb{D}$ gilt:

$$
\begin{array}{lll}
a \sqsubseteq a & & \textit{Reflexivität} \\
a \sqsubseteq b \wedge b \sqsubseteq a & \Longrightarrow \quad a = b & \textit{Anti-Symmetrie} \\
a \sqsubseteq b \wedge b \sqsubseteq c & \Longrightarrow \quad a \sqsubseteq c & \textit{Transitivität}
\end{array}
$$

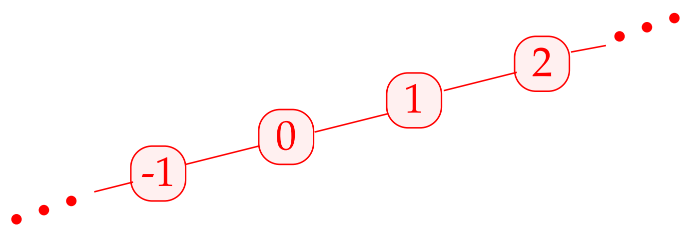
Beispiele:

1. $\mathbb{D} = 2^{\{a,b,c\}}$ mit der Relation "$\subseteq$" :

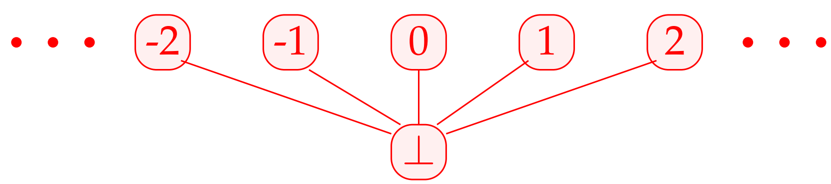


3. $\mathbb{Z}$ mit der Relation "$=$" :

3. $\mathbb{Z}$ mit der Relation "$\leq$" :



4. $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$ mit der Ordnung:



$d \in \mathbb{D}$ heißt obere Schranke für $X \subseteq \mathbb{D}$ falls

$$x \sqsubseteq d \qquad \text{für alle} \quad x \in X$$

$d \in \mathbb{D}$ heißt obere Schranke für $X \subseteq \mathbb{D}$ falls

$$x \sqsubseteq d \qquad \text{für alle} \qquad x \in X$$

d heißt kleinste obere Schranke (lub) falls

1. d eine obere Schranke ist und
2. $d \sqsubseteq y$ für jede obere Schranke y für X.

$d \in \mathbb{D}$ heißt obere Schranke für $X \subseteq \mathbb{D}$ falls

$$x \sqsubseteq d \qquad \text{für alle} \quad x \in X$$

d heißt kleinste obere Schranke (lub) falls

1. d eine obere Schranke ist und
2. $d \sqsubseteq y$ für jede obere Schranke y für X.

Achtung:

- $\{0, 2, 4, \ldots\} \subseteq \mathbb{Z}$ besitzt keine obere Schranke!
- $\{0, 2, 4\} \subseteq \mathbb{Z}$ besitzt die oberen Schranken $4, 5, 6, \ldots$

Ein vollständiger Verband (cl) $\mathbb{D}$ ist eine Halbordnung, in der jede Teilmenge $X \subseteq \mathbb{D}$ eine kleinste obere Schranke $\bigsqcup X \in \mathbb{D}$ besitzt.

Beachte:

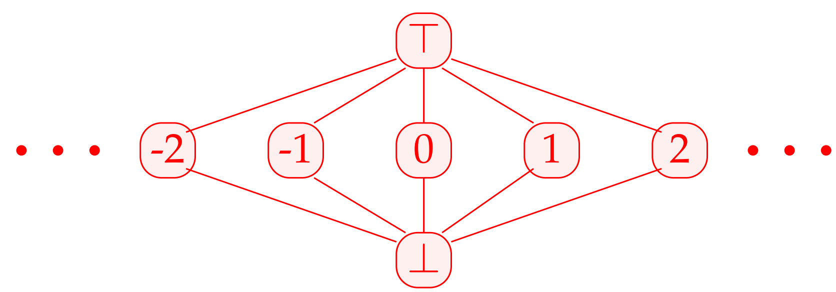
Jeder vollständige Verband besitzt

$\longrightarrow$ ein kleinstes Element $\bot = \bigsqcup \emptyset \in \mathbb{D}$;

$\longrightarrow$ ein größtes Element $\top = \bigsqcup \mathbb{D} \in \mathbb{D}$.

Beispiele:

1. $\mathbb{D} = 2^{\{a,b,c\}}$ ist ein cl :-)
2. $\mathbb{D} = \mathbb{Z}$ mit "$=$" ist keiner.
3. $\mathbb{D} = \mathbb{Z}$ mit "$\leq$" ebenfalls nicht.
4. $\mathbb{D} = \mathbb{Z}_\perp$ auch nicht :-(
5. Mit einem zusätzlichen Symbol $\top$ erhalten wir den flachen Verband $\mathbb{Z}_\perp^\top = \mathbb{Z} \cup \{\perp, \top\}$:



Es gilt:

Satz:

In jedem vollständigen Verband $\mathbb{D}$ besitzt jede Teilmenge $X \subseteq \mathbb{D}$ eine größte untere Schranke $\bigsqcap X$.

Es gilt:

Satz:

In jedem vollständigen Verband $\mathbb{D}$ besitzt jede Teilmenge $X \subseteq \mathbb{D}$ eine größte untere Schranke $\bigsqcap X$.

Beweis:

Konstruiere $U = \{u \in \mathbb{D} \mid \forall\, x \in X : \; u \sqsubseteq x\}$.

// die Menge der unteren Schranken von X :-)

Es gilt:

Satz:

In jedem vollständigen Verband $\mathbb{D}$ besitzt jede Teilmenge $X \subseteq \mathbb{D}$ eine größte untere Schranke $\bigsqcap X$.

Beweis:

Konstruiere $U = \{u \in \mathbb{D} \mid \forall\, x \in X : \; u \sqsubseteq x\}$.

// die Menge der unteren Schranken von X :-)

Setze: $g := \bigsqcup U$

Behauptung: $g = \bigsqcap X$

(1) g ist eine untere Schranke von X :

Für $x \in X$ gilt:

$u \sqsubseteq x$ für alle $u \in U$

$\Longrightarrow$ x ist obere Schranke von U

$\Longrightarrow$ $g \sqsubseteq x$:-)

(1) g ist eine untere Schranke von X :

Für $x \in X$ gilt:

$$u \sqsubseteq x \text{ für alle } u \in U$$

$$\Longrightarrow \quad x \text{ ist obere Schranke von } U$$

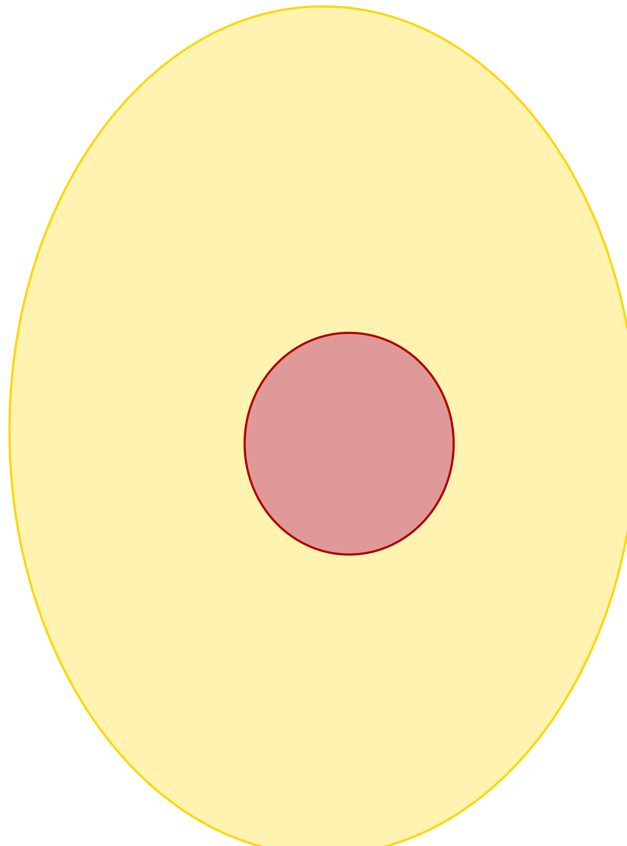
$$\Longrightarrow \quad g \sqsubseteq x \qquad \text{:-)}$$

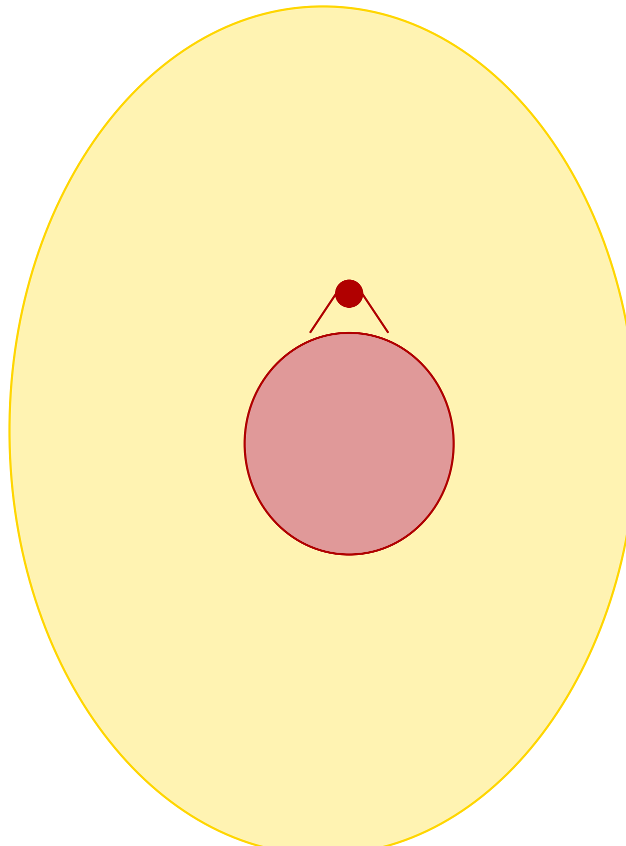
(2) g ist größte untere Schranke von X :

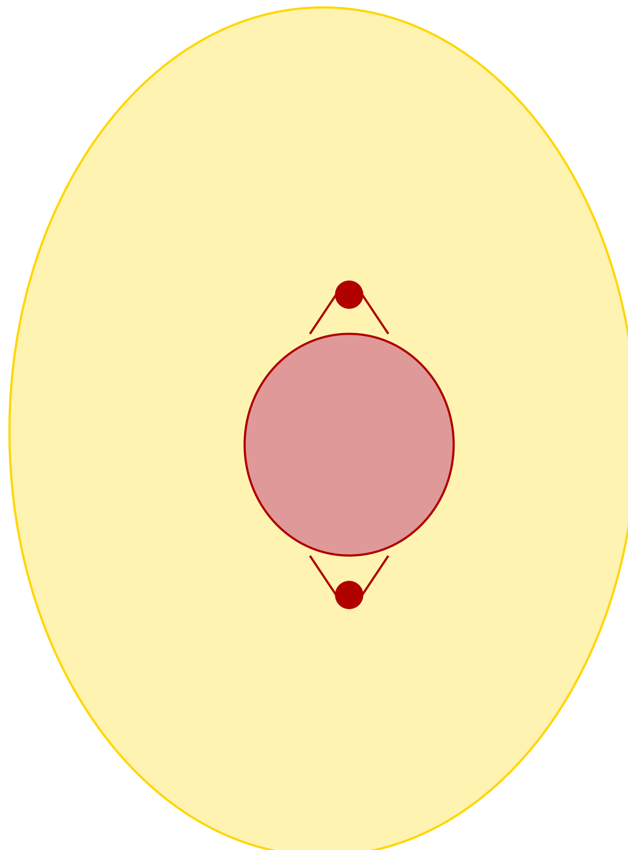
Für jede untere Schranke u von X gilt:

$$u \in U$$

$$\Longrightarrow \quad u \sqsubseteq g \qquad \text{:-))}$$







Wir suchen Lösungen für Ungleichungssysteme der Form:

$$x_i \quad \sqsupseteq \quad f_i(x_1, \ldots, x_n) \qquad (*)$$

Wir suchen Lösungen für Ungleichungssysteme der Form:

$$x_i \quad \sqsupseteq \quad f_i(x_1, \ldots, x_n) \qquad\qquad (*)$$

wobei:

x_i	Unbekannte	hier:	$\mathsf{First}_k(A)$
$\mathbb{D}$	Werte	hier:	$\mathbb{D}_k = 2^{T^{\leq k}}$
$\sqsubseteq \;\subseteq\; \mathbb{D} \times \mathbb{D}$	Ordnungsrelation	hier:	$\subseteq$
$f_i\colon \mathbb{D}^n \longrightarrow \mathbb{D}$	Bedingung	hier:	...

Wir suchen Lösungen für Ungleichungssysteme der Form:

$$x_i \quad \sqsupseteq \quad f_i(x_1, \ldots, x_n) \qquad\qquad (*)$$

wobei:

x_i	Unbekannte	hier:	$\mathsf{First}_k(A)$
$\mathbb{D}$	Werte	hier:	$\mathbb{D}_k = 2^{T^{\leq k}}$
$\sqsubseteq \;\subseteq\; \mathbb{D} \times \mathbb{D}$	Ordnungsrelation	hier:	$\subseteq$
$f_i \colon \mathbb{D}^n \longrightarrow \mathbb{D}$	Bedingung	hier:	...

Ungleichung für $\mathsf{First}_k(A)$:

$$\mathsf{First}_k(A) \quad \supseteq \quad \bigcup \{ \mathsf{First}_k(X_1) \odot \ldots \odot \mathsf{First}_k(X_m) \mid A \to X_1 \ldots X_m \;\in\; P \}$$

Wir suchen Lösungen für Ungleichungssysteme der Form:

$$x_i \quad \sqsupseteq \quad f_i(x_1, \ldots, x_n) \qquad\qquad (*)$$

wobei:

x_i	Unbekannte	hier:	$\mathsf{First}_k(A)$
$\mathbb{D}$	Werte	hier:	$\mathbb{D}_k = 2^{T^{\leq k}}$
$\sqsubseteq \quad \subseteq \quad \mathbb{D} \times \mathbb{D}$	Ordnungsrelation	hier:	$\subseteq$
$f_i \colon \mathbb{D}^n \to \mathbb{D}$	Bedingung	hier:	...

Ungleichung für $\mathsf{First}_k(A)$:

$$\mathsf{First}_k(A) \quad \supseteq \quad \bigcup\{\mathsf{First}_k(X_1) \odot \ldots \odot \mathsf{First}_k(X_m) \mid A \to X_1 \ldots X_m \in P\}$$

Denn:

$$x \sqsupseteq d_1 \wedge \ldots \wedge x \sqsupseteq d_k \quad \text{gdw.} \quad x \sqsupseteq \bigsqcup\{d_1, \ldots, d_k\} \qquad :\text{-})$$

Eine Abbildung $f : \mathbb{D}_1 \to \mathbb{D}_2$ heißt monoton, falls $f(a) \sqsubseteq f(b)$ für alle $a \sqsubseteq b$.

Eine Abbildung $f : \mathbb{D}_1 \to \mathbb{D}_2$ heißt monoton, falls $f(a) \sqsubseteq f(b)$ für alle $a \sqsubseteq b$.

Beispiele:

(1) $\mathbb{D}_1 = \mathbb{D}_2 = 2^U$ für eine Menge U und $f\,x = (x \cap a) \cup b$.

Offensichtlich ist jedes solche f monoton :-)

Eine Abbildung $f : \mathbb{D}_1 \to \mathbb{D}_2$ heißt monoton, falls $f(a) \sqsubseteq f(b)$ für alle $a \sqsubseteq b$.

Beispiele:

(1) $\mathbb{D}_1 = \mathbb{D}_2 = 2^U$ für eine Menge U und $f\, x = (x \cap a) \cup b$.

Offensichtlich ist jedes solche f monoton :-)

(2) $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{Z}$ (mit der Ordnung "$\leq$"). Dann gilt:

- $\mathsf{inc}\, x = x + 1$ ist monoton.
- $\mathsf{dec}\, x = x - 1$ ist monoton.

Eine Abbildung $f : \mathbb{D}_1 \to \mathbb{D}_2$ heißt monoton, falls $f(a) \sqsubseteq f(b)$ für alle $a \sqsubseteq b$.

Beispiele:

(1) $\mathbb{D}_1 = \mathbb{D}_2 = 2^U$ für eine Menge U und $f\,x = (x \cap a) \cup b$.

Offensichtlich ist jedes solche f monoton :-)

(2) $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{Z}$ (mit der Ordnung "$\leq$"). Dann gilt:

- $\mathsf{inc}\,x = x + 1$ ist monoton.
- $\mathsf{dec}\,x = x - 1$ ist monoton.
- $\mathsf{inv}\,x = -x$ ist nicht monoton :-)

Gesucht: möglichst kleine Lösung für:

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$$

wobei alle $f_i : \mathbb{D}^n \to \mathbb{D}$ monoton sind.

Gesucht: möglichst kleine Lösung für:

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$$

wobei alle $f_i : \mathbb{D}^n \to \mathbb{D}$ monoton sind.

Idee:

- Betrachte $F : \mathbb{D}^n \to \mathbb{D}^n$ mit

  $$F(x_1, \ldots, x_n) = (y_1, \ldots, y_n) \quad \text{wobei} \quad y_i = f_i(x_1, \ldots, x_n).$$

Gesucht: möglichst kleine Lösung für:

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$$

wobei alle $f_i : \mathbb{D}^n \to \mathbb{D}$ monoton sind.

Idee:

- Betrachte $F : \mathbb{D}^n \to \mathbb{D}^n$ mit

  $F(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ wobei $y_i = f_i(x_1, \ldots, x_n)$.

- Sind alle f_i monoton, dann auch F :-)

Gesucht: möglichst kleine Lösung für:

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$$

wobei alle $f_i : \mathbb{D}^n \to \mathbb{D}$ monoton sind.

Idee:

- Betrachte $F : \mathbb{D}^n \to \mathbb{D}^n$ mit

  $$F(x_1, \ldots, x_n) = (y_1, \ldots, y_n) \quad \text{wobei} \quad y_i = f_i(x_1, \ldots, x_n).$$

- Sind alle f_i monoton, dann auch F :-)
- Wir approximieren sukzessive eine Lösung. Wir konstruieren:

  $$\underline{\perp}, \quad F\,\underline{\perp}, \quad F^2\,\underline{\perp}, \quad F^3\,\underline{\perp}, \quad \ldots$$

Hoffnung: Wir erreichen irgendwann eine Lösung ... ???

Beispiel: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq \; = \; \subseteq$

$$
\begin{array}{lll}
x_1 & \supseteq & \{a\} \cup x_3 \\
x_2 & \supseteq & x_3 \cap \{a, b\} \\
x_3 & \supseteq & x_1 \cup \{c\}
\end{array}
$$

Beispiel: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq \; = \; \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

Die Iteration:

	0	1	2	3	4
x_1	$\emptyset$				
x_2	$\emptyset$				
x_3	$\emptyset$				

Beispiel: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq \, = \, \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

Die Iteration:

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$			
x_2	$\emptyset$	$\emptyset$			
x_3	$\emptyset$	$\{c\}$			

Beispiel: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq\, = \,\subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

Die Iteration:

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$	$\{a, c\}$		
x_2	$\emptyset$	$\emptyset$	$\emptyset$		
x_3	$\emptyset$	$\{c\}$	$\{a, c\}$		

Beispiel: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq \; = \; \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

Die Iteration:

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$	$\{a, c\}$	$\{a, c\}$	
x_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{a\}$	
x_3	$\emptyset$	$\{c\}$	$\{a, c\}$	$\{a, c\}$	

Beispiel: $\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq = \subseteq$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

Die Iteration:

	0	1	2	3	4
x_1	$\emptyset$	$\{a\}$	$\{a, c\}$	$\{a, c\}$	dito
x_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{a\}$	
x_3	$\emptyset$	$\{c\}$	$\{a, c\}$	$\{a, c\}$	

Offenbar gilt:

- Gilt $F^k \underline{\perp} = F^{k+1} \underline{\perp}$, ist eine Lösung gefunden :-)
- $\underline{\perp}, F\,\underline{\perp}, F^2\,\underline{\perp}, \ldots$ bilden eine aufsteigende Kette :

$$\underline{\perp} \quad \sqsubseteq \quad F\,\underline{\perp} \quad \sqsubseteq \quad F^2\,\underline{\perp} \quad \sqsubseteq \quad \ldots$$

- Sind alle aufsteigenden Ketten endlich, gibt es k immer.

Offenbar gilt:

- Gilt $F^k \underline{\perp} = F^{k+1} \underline{\perp}$, ist eine Lösung gefunden :-)
- $\underline{\perp}, F \underline{\perp}, F^2 \underline{\perp}, \ldots$ bilden eine aufsteigende Kette :

$$\underline{\perp} \quad \sqsubseteq \quad F \underline{\perp} \quad \sqsubseteq \quad F^2 \underline{\perp} \quad \sqsubseteq \quad \ldots$$

- Sind alle aufsteigenden Ketten endlich, gibt es k immer.

Die zweite Aussage folgt mit vollständiger Induktion:

Offenbar gilt:

- Gilt $F^k \underline{\perp} = F^{k+1} \underline{\perp}$, ist eine Lösung gefunden :-)
- $\underline{\perp}, F\,\underline{\perp}, F^2\,\underline{\perp}, \ldots$ bilden eine aufsteigende Kette :

$$\underline{\perp} \quad \sqsubseteq \quad F\,\underline{\perp} \quad \sqsubseteq \quad F^2\,\underline{\perp} \quad \sqsubseteq \quad \ldots$$

- Sind alle aufsteigenden Ketten endlich, gibt es k immer.

Die zweite Aussage folgt mit vollständiger Induktion:

Anfang: $F^0\,\underline{\perp} = \underline{\perp} \sqsubseteq F^1\,\underline{\perp}$:-)

Offenbar gilt:

- Gilt $F^k \underline{\perp} = F^{k+1} \underline{\perp}$, ist eine Lösung gefunden :-)
- $\underline{\perp}, F\,\underline{\perp}, F^2\,\underline{\perp}, \ldots$ bilden eine aufsteigende Kette :

$$\underline{\perp} \quad \sqsubseteq \quad F\,\underline{\perp} \quad \sqsubseteq \quad F^2\,\underline{\perp} \quad \sqsubseteq \quad \ldots$$

- Sind alle aufsteigenden Ketten endlich, gibt es k immer.

Die zweite Aussage folgt mit vollständiger Induktion:

Anfang: $F^0\,\underline{\perp} = \underline{\perp} \sqsubseteq F^1\,\underline{\perp}$:-)

Schluss: Gelte bereits $F^{i-1}\,\underline{\perp} \sqsubseteq F^i\,\underline{\perp}$. Dann

$$F^i\,\underline{\perp} = F\,(F^{i-1}\,\underline{\perp}) \sqsubseteq F\,(F^i\,\underline{\perp}) = F^{i+1}\,\underline{\perp}$$

da F monoton ist :-)

Fazit:

Wenn $\mathbb{D}$ endlich ist, finden wir mit Sicherheit eine Lösung :-)

Fragen:

Fazit:

Wenn $\mathbb{D}$ endlich ist, finden wir mit Sicherheit eine Lösung :-)

Fragen:

1. Gibt es eine kleinste Lösung ?

Fazit:

Wenn $\mathbb{D}$ endlich ist, finden wir mit Sicherheit eine Lösung :-)

Fragen:

1. Gibt es eine kleinste Lösung ?
2. Wenn ja: findet Iteration die kleinste Lösung ??

Fazit:

Wenn $\mathbb{D}$ endlich ist, finden wir mit Sicherheit eine Lösung :-)

Fragen:

1. Gibt es eine kleinste Lösung ?
2. Wenn ja: findet Iteration die kleinste Lösung ??
3. Was, wenn $\mathbb{D}$ nicht endlich ist ???

Satz

Kleene

In einer vollständigen Halbordnung $\mathbb{D}$ hat jede stetige Funktion $f : \mathbb{D} \to \mathbb{D}$ einen kleinsten Fixpunkt d_0.

Dieser ist gegeben durch $d_0 = \bigsqcup_{k \geq 0} f^k \bot$.

Satz — Kleene

In einer vollständigen Halbordnung $\mathbb{D}$ hat jede stetige Funktion $f : \mathbb{D} \to \mathbb{D}$ einen kleinsten Fixpunkt d_0.

Dieser ist gegeben durch $d_0 = \bigsqcup_{k \geq 0} f^k \bot$.

Bemerkung:

- Eine Funktion f heißt stetig, falls für jede aufsteigende Kette $d_0 \sqsubseteq \ldots \sqsubseteq d_m \sqsubseteq \ldots$ gilt: $f\,(\bigsqcup_{m \geq 0} d_m) = \bigsqcup_{m \geq 0}(f\,d_m)$.
- Werden alle aufsteigenden Ketten irgendwann stabil, ist jede monotone Funktion automatisch stetig :-)

Satz **Kleene**

In einer vollständigen Halbordnung $\mathbb{D}$ hat jede stetige Funktion $f : \mathbb{D} \to \mathbb{D}$ einen kleinsten Fixpunkt d_0.

Dieser ist gegeben durch $d_0 = \bigsqcup_{k\geq 0} f^k \bot$.

Bemerkung:

- Eine Funktion f heißt stetig, falls für jede aufsteigende Kette $d_0 \sqsubseteq \ldots \sqsubseteq d_m \sqsubseteq \ldots$ gilt: $f\,(\bigsqcup_{m\geq 0} d_m) = \bigsqcup_{m\geq 0}(f\,d_m)$.
- Werden alle aufsteigenden Ketten irgendwann stabil, ist jede monotone Funktion automatisch stetig :-)
- Eine Halbordnung heißt vollständig (CPO), falls alle aufsteigenden Ketten kleinste obere Schranken haben :-)
- Jeder vollständige Verband ist auch eine vollständige Halbordnung :-)

Beweis:

(1) $f\,d_0 = d_0$:

$$\begin{array}{lll} f\,d_0 & = & f\left(\bigsqcup_{m\geq 0}(f^m\bot)\right) \\ & = & \bigsqcup_{m\geq 0}(f^{m+1}\bot) \qquad \text{wegen Stetigkeit} \quad \text{:-)} \\ & = & \bot \sqcup \left(\bigsqcup_{m\geq 0}(f^{m+1}\bot)\right) \\ & = & \bigsqcup_{m\geq 0}(f^m\bot) \\ & = & d_0 \end{array}$$

(2) d_0 ist kleinster Fixpunkt:

Sei $f\,d_1 = d_1$ weiterer Fixpunkt. Wir zeigen: $\forall\, m \geq 0 : \; f^m\bot \sqsubseteq d_1$.

$$\begin{array}{lrcll} m = 0 : & \bot & \sqsubseteq & d_1 & \text{nach Definition} \\ m > 0 : \; \text{Gelte} & f^{m-1}\bot & \sqsubseteq & d_1 & \text{Dann folgt:} \\ & f^m\bot & = & f\,(f^{m-1}\bot) & \\ & & \sqsubseteq & f\,d_1 & \text{wegen Monotonie} \quad \text{:-)} \\ & & = & d_1 & \end{array}$$

Bemerkung:

- Jede stetige Funktion ist auch monoton :-)
- Betrachte die Menge der Postfixpunkte:

$$P = \{x \in \mathbb{D} \mid x \sqsupseteq f\,x\}$$

Der kleinste Fixpunkt d_0 ist in P und untere Schranke :-)

$\Longrightarrow$ d_0 ist der kleinste Wert x mit $x \sqsupseteq f\,x$

Bemerkung:

- Jede stetige Funktion ist auch monoton :-)
- Betrachte die Menge der Postfixpunkte:

$$P = \{x \in \mathbb{D} \mid x \sqsupseteq f\,x\}$$

Der kleinste Fixpunkt d_0 ist in P und untere Schranke :-)

$\Longrightarrow$ d_0 ist der kleinste Wert x mit $x \sqsupseteq f\,x$

Anwendung:

Sei $$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$$

ein Ungleichungssystem, wobei alle $f_i : \mathbb{D}^n \to \mathbb{D}$ monoton sind.

Bemerkung:

- Jede stetige Funktion ist auch monoton :-)
- Betrachte die Menge der Postfixpunkte:

$$P = \{x \in \mathbb{D} \mid x \sqsupseteq f\,x\}$$

Der kleinste Fixpunkt d_0 ist in P und untere Schranke :-)

$\Longrightarrow$ d_0 ist der kleinste Wert x mit $x \sqsupseteq f\,x$

Anwendung:

Sei $$x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n \qquad (*)$$

ein Ungleichungssystem, wobei alle $f_i : \mathbb{D}^n \to \mathbb{D}$ monoton sind.

$\Longrightarrow$ kleinste Lösung von $(*)$ $==$ kleinster Fixpunkt von F :-)

Der Kleenesche Fixpunkt-Satz liefert uns nicht nur die Existenz einer kleinsten Lösung sondern auch eine Charakterisierung :-)

Satz

Die Mengen $\mathsf{First}_k(\{w \in T^* \mid A \to^* w\})$, $A \in N$, sind die kleinste Lösung des Ungleichungssystems:

$$\mathsf{First}_k(A) \quad \supseteq \quad \mathsf{First}_k(X_1) \odot \ldots \odot \mathsf{First}_k(X_m) \ , \quad A \to X_1 \ldots X_m \quad \in \quad P$$

Der Kleenesche Fixpunkt-Satz liefert uns nicht nur die Existenz einer kleinsten Lösung sondern auch eine Charakterisierung :-)

Satz

Die Mengen $\text{First}_k(\{w \in T^* \mid A \to^* w\})$, $A \in N$, sind die kleinste Lösung des Ungleichungssystems:

$$\text{First}_k(A) \quad \supseteq \quad \text{First}_k(X_1) \odot \ldots \odot \text{First}_k(X_m) \quad , \quad A \to X_1 \ldots X_m \quad \in \quad P$$

Beweis-Idee:

Sei $\mathsf{F}^{(m)}(A)$ die m-te Approximation an den Fixpunkt.

(1) Falls $A \to^m u$, dann $\text{First}_k(u) \subseteq \mathsf{F}^{(m)}(A)$.

(2) Falls $w \in \mathsf{F}^{(m)}(A)$, dann $A \to^* u$ für $u \in T^*$ mit $\text{First}_k(u) = \{w\}$:-)

Fazit:

Wir können First_k durch Fixpunkt-Iteration berechnen, d.h. durch wiederholtes Einsetzen :-)

Fazit:

Wir können First_k durch Fixpunkt-Iteration berechnen, d.h. durch wiederholtes Einsetzen :-)

Achtung: Naive Fixpunkt-Iteration ist ziemlich ineffizient :-(

Fazit:

Wir können First_k durch Fixpunkt-Iteration berechnen, d.h. durch wiederholtes Einsetzen :-)

Achtung: Naive Fixpunkt-Iteration ist ziemlich ineffizient :-(

Idee: Round Robin Iteration

Benutze bei der Iteration nicht die Werte der letzten Iteration, sondern die jeweils aktuellen :-)

Unser Mini-Beispiel:

$$\mathbb{D} = 2^{\{a,b,c\}}, \quad \sqsubseteq \; = \; \subseteq$$

$$\begin{array}{lll} x_1 & \supseteq & \{a\} \cup x_3 \\ x_2 & \supseteq & x_3 \cap \{a, b\} \\ x_3 & \supseteq & x_1 \cup \{c\} \end{array}$$

Die Round-Robin-Iteration:

	1	2	3
x_1	$\{a\}$	$\{a, c\}$	dito
x_2	$\emptyset$	$\{a\}$	
x_3	$\{a, c\}$	$\{a, c\}$	

Der Code für Round Robin Iteration sieht in Java so aus:

```
for (i = 1; i ≤ n; i++) x_i = ⊥;
do {
        finished = true;
        for (i = 1; i ≤ n; i++) {
                new = f_i(x_1, ..., x_n);
                if (!(x_i ⊒ new)) {
                        finished = false;
                        x_i = x_i ⊔ new;
                }
        }
} while (!finished);
```

Zur Korrektheit:

Sei $y_i^{(d)}$ die i-te Komponente von $F^d \underline{\perp}$.

Sei $x_i^{(d)}$ der Wert von x_i nach der i-ten RR-Iteration.

Zur Korrektheit:

Sei $y_i^{(d)}$ die i-te Komponente von $F^d\,\underline{\perp}$.

Sei $x_i^{(d)}$ der Wert von x_i nach der i-ten RR-Iteration.

Man zeigt:

(1) $y_i^{(d)} \sqsubseteq x_i^{(d)}$:-)

Zur Korrektheit:

Sei $y_i^{(d)}$ die i-te Komponente von $F^d\,\underline{\perp}$.

Sei $x_i^{(d)}$ der Wert von x_i nach der i-ten RR-Iteration.

Man zeigt:

(1) $y_i^{(d)} \sqsubseteq x_i^{(d)}$:-)

(2) $x_i^{(d)} \sqsubseteq z_i$ für jede Lösung $(z_1, \ldots, z_n)$:-)

Zur Korrektheit:

Sei $y_i^{(d)}$ die i-te Komponente von $F^d \underline{\perp}$.

Sei $x_i^{(d)}$ der Wert von x_i nach der i-ten RR-Iteration.

Man zeigt:

(1) $y_i^{(d)} \sqsubseteq x_i^{(d)}$:-)

(2) $x_i^{(d)} \sqsubseteq z_i$ für jede Lösung $(z_1, \ldots, z_n)$:-)

(3) Terminiert RR-Iteration nach d Runden, ist
$(x_1^{(d)}, \ldots, x_n^{(d)})$ eine Lösung :-))

Unsere Anwendung:

$$\begin{array}{lcll}
\mathsf{First}_2(E) & \supseteq & \mathsf{First}_2(E) \odot \{+\} \odot \mathsf{First}_2(T) & \cup \quad \mathsf{First}_2(T) \\
\mathsf{First}_2(T) & \supseteq & \mathsf{First}_2(T) \odot \{*\} \odot \mathsf{First}_2(F) & \cup \quad \mathsf{First}_2(F) \\
\mathsf{First}_2(F) & \supseteq & \{(\} \odot \mathsf{First}_2(E) \odot \{)\} & \cup \quad \{\mathsf{name}, \mathsf{int}\}
\end{array}$$

Die RR-Iteration:

First_2	1	2	3
F	name, int	(name, (int	((
T	name, int	(name, (int, name $*$, int $*$	((
E	name, int	(name, (int, name $*$, int $*$, name $+$, int $+$	((

Der Einfachheit halber haben wir in jeder Iteration nur die neuen Elemente vermerkt :-)

Diskussion:

- Die Länge h der längsten echt aufsteigenden Kette nennen wir auch Höhe von $\mathbb{D}$ ...
- Im Falle von First_k ist die Höhe des Verbands exponentiell in k :-(
- Die Anzahl der Runden von RR-Iteration ist beschränkt durch $\mathcal{O}(n \cdot h)$ (n die Anzahl der Variablen)
- Die praktische Effizienz von RR-Iteration hängt allerdings auch von der Anordnung der Variablen ab :-)
- Anstelle von RR-Iteration gibt es auch schnellere Fixpunkt-Verfahren, die aber im schlimmsten Fall immer noch exponentiell sind :-((

$\Longrightarrow$ Man beschränkt sich i.a. auf kleine k !!!

2.4 Topdown Parsing

Idee:

- Benutze den Item-Kellerautomaten.
- Benutze die nächsten k Zeichen, um die Regeln für die Expansionen zu bestimmen ;-)
- Eine Grammatik heißt $LL(k)$, falls dies immer eindeutig möglich ist.

2.4 Topdown Parsing

Idee:

- Benutze den Item-Kellerautomaten.
- Benutze die nächsten k Zeichen, um die Regeln für die Expansionen zu bestimmen ;-)
- Eine Grammatik heißt $LL(k)$, falls dies immer eindeutig möglich ist.

Wir definieren:

Eine reduzierte Grammatik heißt dann $LL(k)$, falls für je zwei verschiedene Regeln $A \to \alpha$, $A \to \alpha' \in P$ und jede Ableitung $S \to_L^* u\, A\, \beta$ mit $u \in T^*$ gilt:

$$\mathsf{First}_k(\alpha\, \beta) \;\cap\; \mathsf{First}_k(\alpha'\, \beta) = \emptyset$$

Beispiel 1:

$$\begin{array}{lll} S & \to & \mathsf{if}\ (\ E\ )\ S\ \mathsf{else}\ S \quad | \\ & & \mathsf{while}\ (\ E\ )\ S \quad | \\ & & E\ ; \\ E & \to & \mathsf{id} \end{array}$$

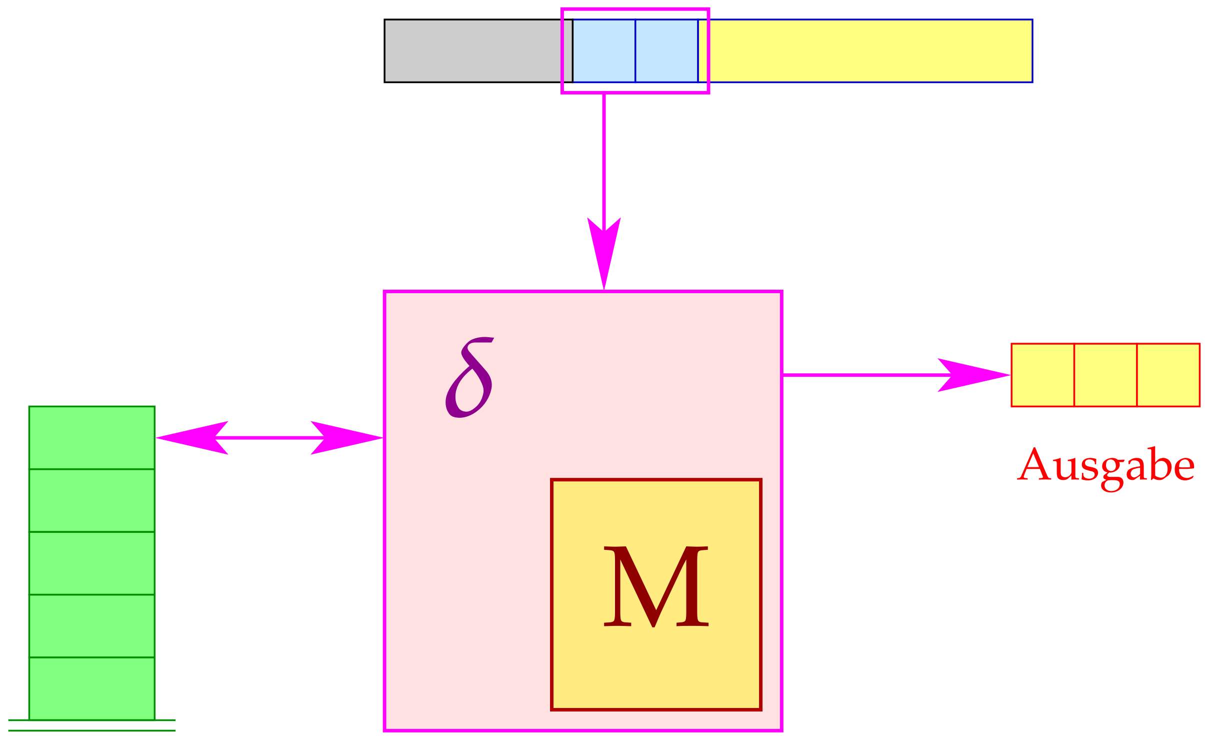
ist $LL(1)$, da $\quad \mathsf{First}_k(E) = \{\mathsf{id}\}$:-)

Beispiel 2:

$$
\begin{array}{lll}
S & \rightarrow & \mathsf{if}\ (\ E\ )\ S\ \mathsf{else}\ S \quad | \\
 & & \mathsf{if}\ (\ E\ )\ S \quad | \\
 & & \mathsf{while}\ (\ E\ )\ S \quad | \\
 & & E\ ; \\
E & \rightarrow & \mathsf{id}
\end{array}
$$

... ist nicht $LL(k)$ für jedes $k > 0$.

Struktur des $LL(k)$-Parsers:



- Der Parser sieht ein Fenster der Länge k der Eingabe;
- er realisiert im Wesentlichen den Item-Kellerautomaten;
- die Tabelle $M[q, w]$ enthält die jeweils zuwählende Regel :-)

... im Beispiel:

$$
\begin{array}{lll}
S & \to & \text{if } (\; E \;) \; S \text{ else } S^{\,0} \quad | \\
 & & \text{while } (\; E \;) \; S^{\,1} \quad | \\
 & & E \, ;^{\,2} \\
E & \to & \text{id}^{\,0}
\end{array}
$$

Zustände: Items

Tabelle:

	if	while	id
$[\ldots \to \ldots \bullet S \ldots]$	0	1	2
$[\ldots \to \ldots \bullet E \ldots]$	–	–	0

Im Allgemeinen ...

- ist die Menge der möglichen nächsten k Zeichen gegeben durch:

$$\mathsf{First}_k(\alpha\,\beta) \;=\; \mathsf{First}_k(\alpha) \odot \mathsf{First}_k(\beta)$$

  wobei:

  (1) α die rechte Seite der passenden Regel;

  (2) β ein möglicher rechter Kontext von A ist :-)

- $\mathsf{First}_k(\beta)$ müssen wir dynamisch akkumulieren.

$\Longrightarrow$ Wir erweitern Items um Vorausschau-Mengen ...

Ein erweitertes Item ist ein Paar: $[A \rightarrow \alpha \bullet \gamma\,,\; L] \quad (A \rightarrow \alpha\,\gamma \in P,\; L \subseteq T^{\leq k})$

Die Menge L benutzen wir, um $\text{First}_k(\beta)$ für den rechten Kontext β von A zu repräsentieren :-)

Ein erweitertes Item ist ein Paar: $[A \to \alpha \bullet \gamma\,,\ L]$ $(A \to \alpha\,\gamma \in P,\ L \subseteq T^{\leq k})$

Die Menge L benutzen wir, um $\text{First}_k(\beta)$ für den rechten Kontext β von A zu repräsentieren :-)

Konstruktion:

Zustände: erweiterte Items

Anfangszustand: $[S' \to \bullet\ S,\ \{\epsilon\}]$

Endzustand: $[S' \to S\,\bullet,\ \{\epsilon\}]$

Übergänge:

Ein erweitertes Item ist ein Paar: $[A \to \alpha \bullet \gamma\,,\; L] \quad (A \to \alpha\,\gamma \in P,\; L \subseteq T^{\leq k})$

Die Menge L benutzen wir, um $\text{First}_k(\beta)$ für den rechten Kontext β von A zu repräsentieren :-)

Konstruktion:

Zustände: erweiterte Items

Anfangszustand: $[S' \to \bullet\; S,\; \{\epsilon\}]$

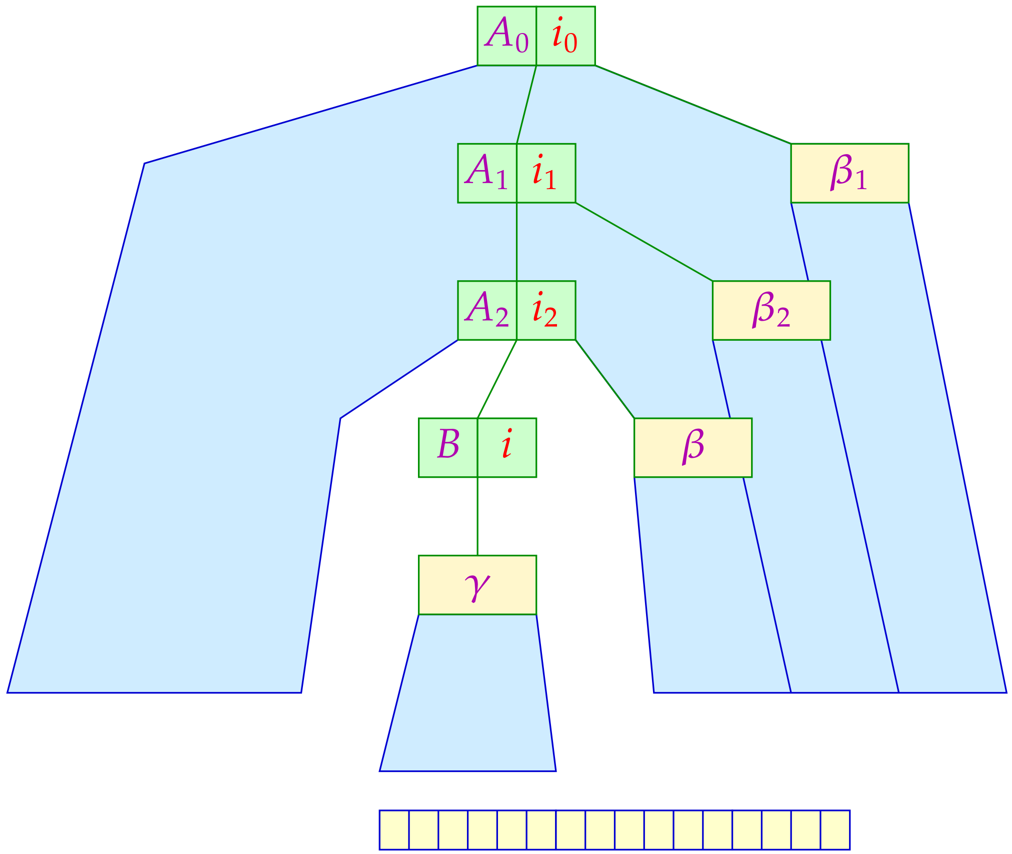
Endzustand: $[S' \to S\,\bullet,\; \{\epsilon\}]$

Übergänge:

Expansionen: $([A \to \alpha \bullet B\,\beta\,,\; L], \epsilon, [A \to \alpha \bullet B\,\beta\,,\; L]\; [B \to \bullet\,\gamma\,,\; \boxed{\text{First}_k(\beta) \odot L}])$ für $A \to \alpha\,B\,\beta,\; B \to \gamma \in P$

Shifts: $([A \to \alpha \bullet a\,\beta\,,\; L], a, [A \to \alpha\,a \bullet \beta\,,\; L])$ für $A \to \alpha\,a\,\beta \in P$

Reduce: $([A \to \alpha \bullet B\,\beta\,,\; L]\; [B \to \gamma\bullet\,,\; L'], \epsilon, [A \to \alpha\,B \bullet \beta\,,\; L])$ für $A \to \alpha\,B\,\beta,\; B \to \gamma \in P$



Die Vorausschau-Tabelle:

Wir setzen $M[\,[A \to \alpha \bullet B\,\beta\,,\,L],\,w] \;=\; i$ genau dann wenn (B, i) die Regel $B \to \gamma$ ist und: $w \in \mathsf{First}_k(\gamma) \odot \mathsf{First}_k(\beta) \odot L$

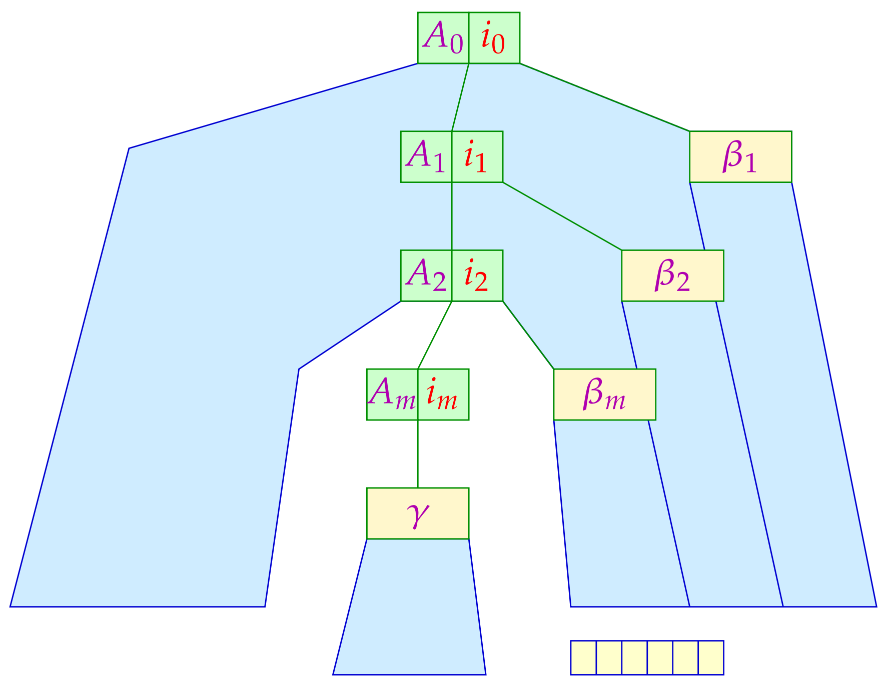
$$([A_0 \to \bullet\, \alpha_1\, A_1\, \beta_1,\ L_1],\ uv) \quad \vdash^* \quad ([A_0 \to \alpha_1 \bullet A_1\, \beta_1,\ L_1] \ldots [A_{m-1} \to \alpha_m \bullet A_m\, \beta_m,\ L_m],\ v)$$

$$\vdash^* \quad ([A_0 \to \alpha_1\, A_1\, \beta_1 \bullet,\ L_1],\ \epsilon) \quad \ldots \quad \text{gilt genau dann wenn:}$$

(1) $\alpha_1 \ldots \alpha_m \to^* u$

(2) $A_m\, \beta_m \ldots \beta_1 \to^* v$

(3) $L_m = \mathsf{First}_k(\beta_{m-1}) \odot \ldots \odot \mathsf{First}_k(\beta_1) \odot L_1$



Satz

Die reduzierte kontextfreie Grammatik G ist $LL(k)$ genau dann wenn die k-Vorausschau-Tabelle für alle benötigten erweiterten Items wohl-definiert ist.

Diskussion:

- Der erweiterte Item-Kellerautomat zusammen mit einer k-Vorausschau-Tabelle erlaubt die deterministische Rekonstruktion einer Links-Ableitung :-)
- Die Anzahl der Vorausschau-Mengen L kann sehr groß sein :-(
- ...

Beispiel:

$$S \rightarrow \epsilon \quad | \quad a\,S\,b$$

Die Übergänge des erweiterten Item-Kellerautomat $(k = 1)$:

0	$[S' \rightarrow \bullet\, S, \{\epsilon\}]$	ϵ	$[S' \rightarrow \bullet\, S, \{\epsilon\}]\,[S \rightarrow \bullet, \{\epsilon\}]$
1	$[S' \rightarrow \bullet\, S, \{\epsilon\}]$	ϵ	$[S' \rightarrow \bullet\, S, \{\epsilon\}]\,[S \rightarrow \bullet\, a\,S\,b, \{\epsilon\}]$
2	$[S \rightarrow \bullet\, a\,S\,b, \{\epsilon\}]$	a	$[S \rightarrow a \bullet S\,b, \{\epsilon\}]$
	$[S \rightarrow \bullet\, a\,S\,b, \{b\}]$	a	$[S \rightarrow a \bullet S\,b, \{b\}]$
3	$[S \rightarrow a \bullet S\,b, \{\epsilon\}]$	ϵ	$[S \rightarrow a \bullet S\,b, \{\epsilon\}]\,[S \rightarrow \bullet, \{b\}]$
	$[S \rightarrow a \bullet S\,b, \{b\}]$	ϵ	$[S \rightarrow a \bullet S\,b, \{b\}]\,[S \rightarrow \bullet, \{b\}]$
4	$[S \rightarrow a \bullet S\,b.\{\epsilon\}]$	ϵ	$[S \rightarrow a \bullet S\,b.\{\epsilon\}]\,[S \rightarrow \bullet\, a\,S\,b, \{b\}]$
	$[S \rightarrow a \bullet S\,b.\{b\}]$	ϵ	$[S \rightarrow a \bullet S\,b.\{b\}]\,[S \rightarrow \bullet\, a\,S\,b, \{b\}]$
5	$[S \rightarrow a \bullet S\,b, \{\epsilon\}]\,[S \rightarrow \bullet, \{b\}]$	ϵ	$[S \rightarrow a\,S \bullet b, \{\epsilon\}]$
	$[S \rightarrow a \bullet S\,b, \{b\}]\,[S \rightarrow \bullet, \{b\}]$	ϵ	$[S \rightarrow a\,S \bullet b, \{b\}]$
	$\ldots$		$\ldots$

	…		…
6	$[S \to a \bullet S\, b, \{\epsilon\}]\ [S \to a\, S\, b \bullet, \{b\}]$	ϵ	$[S \to a\, S \bullet b, \{\epsilon\}]$
	$[S \to a \bullet S\, b, \{b\}]\ [S \to a\, S\, b \bullet, \{b\}]$	ϵ	$[S \to a\, S \bullet b, \{b\}]$
7	$[S \to a\, S \bullet b, \{\epsilon\}]$	b	$[S \to a\, S\, b \bullet, \{\epsilon\}]$
	$[S \to a\, S \bullet b, \{b\}]$	b	$[S \to a\, S\, b \bullet, \{b\}]$
8	$[S' \to \bullet\, S, \{\epsilon\}]\ [S \to \bullet, \{\epsilon\}]$	ϵ	$[S' \to S \bullet, \{\epsilon\}]$
9	$[S' \to \bullet\, S, \{\epsilon\}]\ [S \to a\, S\, b \bullet, \{\epsilon\}]$	ϵ	$[S' \to S \bullet, \{\epsilon\}]$

Die Vorausschau-Tabelle:

	ϵ	a	b
$[S' \to \bullet\, S, \{\epsilon\}]$	0	1	–
$[S \to a \bullet S\, b, \{\epsilon\}]$	–	1	0
$[S \to a \bullet S\, b, \{b\}]$	–	1	0

Beobachtung:

- Die auszuwählende Regel hängt hier ja gar nicht von den Erweiterungen der Items ab !!!
- Unter dieser Voraussetzung können wir den Item-Kellerautomaten ohne Erweiterung benutzen :-)
- Hängt die auszuwählende Regel nur von der aktuellen Vorausschau w ab, nennen wir G auch stark $LL(k)$ ...

Beobachtung:

- Die auszuwählende Regel hängt hier ja gar nicht von den Erweiterungen der Items ab !!!
- Unter dieser Voraussetzung können wir den Item-Kellerautomaten ohne Erweiterung benutzen :-)
- Hängt die auszuwählende Regel nur von der aktuellen Vorausschau w ab, nennen wir G auch stark $LL(k)$ ...

Wir definieren: $\mathsf{Follow}_k(A) \;=\; \bigcup\{\mathsf{First}_k(\beta) \mid S \rightarrow_L^* u\, A\, \beta\}$.

Beobachtung:

- Die auszuwählende Regel hängt hier ja gar nicht von den Erweiterungen der Items ab !!!
- Unter dieser Voraussetzung können wir den Item-Kellerautomaten ohne Erweiterung benutzen :-)
- Hängt die auszuwählende Regel nur von der aktuellen Vorausschau w ab, nennen wir G auch stark $LL(k)$ ...

Wir definieren:

$$\mathsf{Follow}_k(A) = \bigcup\{\mathsf{First}_k(\beta) \mid S \rightarrow^*_L u\, A\, \beta\} .$$

Die reduzierte kontextfreie Grammatik G heißt stark $LL(k)$, falls für je zwei verschiedene $A \rightarrow \alpha\,,\ A \rightarrow \alpha' \in P$:

$$\mathsf{First}_k(\alpha) \odot \mathsf{Follow}_k(A) \cap \mathsf{First}_k(\alpha') \odot \mathsf{Follow}_k(A) = \emptyset$$

... im Beispiel:

$$S \to \epsilon \quad | \quad a\,S\,b$$

$$\text{Follow}_1(S) \quad = \quad \{\epsilon, b\}$$

$$\begin{array}{lclcl}
\text{First}_1(\epsilon) \odot \text{Follow}_1(S) & = & \{\epsilon\} \odot \{\epsilon, b\} & = & \{\epsilon, b\} \\
\text{First}_1(a\,S\,b) \odot \text{Follow}_1(S) & = & \{a\} \odot \{\epsilon, b\} & = & \{a\}
\end{array}$$

Wir schließen: Die Grammatik ist in der Tat stark $LL(1)$:-)

Ist G eine starke $LL(k)$-Grammatik, können wir die Vorausschau-Tabelle statt mit (erweiterten) Items mit Nichtterminalen indizieren :-)

Wir setzen $M[B, w] = i$ genau dann wenn (B, i) die Regel $B \to \gamma$ ist und: $w \in \mathsf{First}_k(\gamma) \odot \mathsf{Follow}_k(B)$.

... im Beispiel:

$$S \to \epsilon \quad | \quad a\,S\,b$$

	ϵ	a	b
S	0	1	0

Ist G eine starke $LL(k)$-Grammatik, können wir die Vorausschau-Tabelle statt mit (erweiterten) Items mit Nichtterminalen indizieren :-)

Wir setzen $M[B, w] = i$ genau dann wenn (B, i) die Regel $B \to \gamma$ ist und: $w \in \mathsf{First}_k(\gamma) \odot \mathsf{Follow}_k(B)$.

... im Beispiel:

$$S \to \epsilon \quad | \quad a\,S\,b$$

	ϵ	a	b
S	0	1	0

Satz

- Jede starke $LL(k)$-Grammatik ist auch $LL(k)$:-)
- Jede $LL(1)$-Grammatik ist bereits stark $LL(1)$:-))

Beweis:

Sei G stark $LL(k)$.

Betrachte eine Ableitung $S \rightarrow_L^* u\, A\, \beta$ und Regeln $A \rightarrow \alpha\,,\; A \rightarrow \alpha' \;\in\; P$.

Dann haben wir:

$$\begin{array}{lll} \mathsf{First}_k(\alpha\,\beta) \;\cap\; \mathsf{First}_k(\alpha'\,\beta) & = & \mathsf{First}_k(\alpha) \odot \mathsf{First}_k(\beta) \;\cap\; \mathsf{First}_k(\alpha') \odot \mathsf{First}_k(\beta) \\ & \subseteq & \mathsf{First}_k(\alpha) \odot \mathsf{Follow}_k(A) \;\cap\; \mathsf{First}_k(\alpha') \odot \mathsf{Follow}_k(A) \\ & = & \emptyset \end{array}$$

Folglich ist G auch $LL(k)$:-)

Sei G $LL(1)$.

Betrachte zwei verschiedene Regeln $A \to \alpha\,,\ A \to \alpha' \ \in \ P$.

Fall 1: $\epsilon \in \mathsf{First}_1(\alpha) \cap \mathsf{First}_1(\alpha')$.

Dann kann G nicht $LL(1)$ sein :-)

Sei G $LL(1)$.

Betrachte zwei verschiedene Regeln $A \rightarrow \alpha\,,\ A \rightarrow \alpha' \ \in\ P$.

Fall 1: $\epsilon \in \text{First}_1(\alpha) \cap \text{First}_1(\alpha')$.

Dann kann G nicht $LL(1)$ sein :-)

Fall 2: $\epsilon \notin \text{First}_1(\alpha) \cup \text{First}_1(\alpha')$.

Sei $S \rightarrow_L^* u\,A\,\beta$. Da G $LL(1)$ ist, gilt:

$$\begin{aligned}
\text{First}_1(\alpha) \odot \text{Follow}_1(A) \cap \text{First}_1(\alpha') \odot \text{Follow}_1(A) \\
&= \text{First}_1(\alpha) \cap \text{First}_1(\alpha') \\
&= \text{First}_1(\alpha) \odot \text{First}_1(\beta) \cap \text{First}_1(\alpha') \odot \text{First}_1(\beta) \\
&= \emptyset
\end{aligned}$$

Fall 3: $\epsilon \in \mathsf{First}_1(\alpha)$ und $\epsilon \notin \mathsf{First}_1(\alpha')$.

Dann gilt:

$$
\begin{aligned}
&\mathsf{First}_1(\alpha) \odot \mathsf{Follow}_1(A) \cap \mathsf{First}_1(\alpha') \odot \mathsf{Follow}_1(A) \\
&\qquad = \mathsf{First}_1(\alpha) \odot \mathsf{Follow}_1(A) \cap \mathsf{First}_1(\alpha') \\
&\qquad = \mathsf{First}_1(\alpha) \odot (\bigcup\{\mathsf{First}_1(\beta) \mid S \rightarrow_L^* u\, A\, \beta\}) \cap \mathsf{First}_1(\alpha') \\
&\qquad = (\bigcup\{\mathsf{First}_1(\alpha) \odot \mathsf{First}_1(\beta) \mid S \rightarrow_L^* u\, A\, \beta\}) \cap \mathsf{First}_1(\alpha') \\
&\qquad = \bigcup\{\mathsf{First}_1(\alpha) \odot \mathsf{First}_1(\beta) \cap \mathsf{First}_1(\alpha') \mid S \rightarrow_L^* u\, A\, \beta\} \\
&\qquad = \bigcup\{\emptyset \mid S \rightarrow_L^* u\, A\, \beta\} \\
&\qquad = \emptyset
\end{aligned}
$$

Fall 3: $\epsilon \in \mathsf{First}_1(\alpha)$ und $\epsilon \notin \mathsf{First}_1(\alpha')$.

Dann gilt:

$$
\begin{aligned}
\mathsf{First}_1(\alpha) \odot \mathsf{Follow}_1(A) \cap \mathsf{First}_1(\alpha') \odot \mathsf{Follow}_1(A) \\
&= \mathsf{First}_1(\alpha) \odot \mathsf{Follow}_1(A) \cap \mathsf{First}_1(\alpha') \\
&= \mathsf{First}_1(\alpha) \odot (\bigcup\{\mathsf{First}_1(\beta) \mid S \rightarrow_L^* u\, A\, \beta\}) \cap \mathsf{First}_1(\alpha') \\
&= (\bigcup\{\mathsf{First}_1(\alpha) \odot \mathsf{First}_1(\beta) \mid S \rightarrow_L^* u\, A\, \beta\}) \cap \mathsf{First}_1(\alpha') \\
&= \bigcup\{\mathsf{First}_1(\alpha) \odot \mathsf{First}_1(\beta) \cap \mathsf{First}_1(\alpha') \mid S \rightarrow_L^* u\, A\, \beta\} \\
&= \bigcup\{\emptyset \mid S \rightarrow_L^* u\, A\, \beta\} \\
&= \emptyset
\end{aligned}
$$

Fall 4: $\epsilon \notin \mathsf{First}_1(\alpha)$ und $\epsilon \in \mathsf{First}_1(\alpha')$: analog :-)

Beispiel:

$$\begin{array}{lll} S & \rightarrow & a\,A\,a\,a^{\,0} \;\mid\; b\,A\,b\,a^{\,1} \\ A & \rightarrow & b^{\,0} \;\mid\; \epsilon^{\,1} \end{array}$$

Offenbar ist die Grammatik $LL(2)$:-) Andererseits gilt:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathsf{First}_2(b) \odot \mathsf{Follow}_2(A) \;\cap\; \mathsf{First}_2(\epsilon) \odot \mathsf{Follow}_2(A)} \\ = & \{b\} \odot \{a\,a\,,\; b\,a\} \;\cap\; \{\epsilon\} \odot \{a\,a\,,\; b\,a\} \\ = & \{b\,a\,,\; b\,b\} \;\cap\; \{a\,a\,,\; b\,a\} \\ \neq & \emptyset \end{array}$$

Folglich ist die Grammatik nicht stark $LL(2)$:-(

Beispiel:

$$\begin{array}{lll} S & \to & a\,A\,a\,a^{\,0} \;\mid\; b\,A\,b\,a^{\,1} \\ A & \to & b^{\,0} \;\mid\; \epsilon^{\,1} \end{array}$$

Offenbar ist die Grammatik $LL(2)$:-) Andererseits gilt:

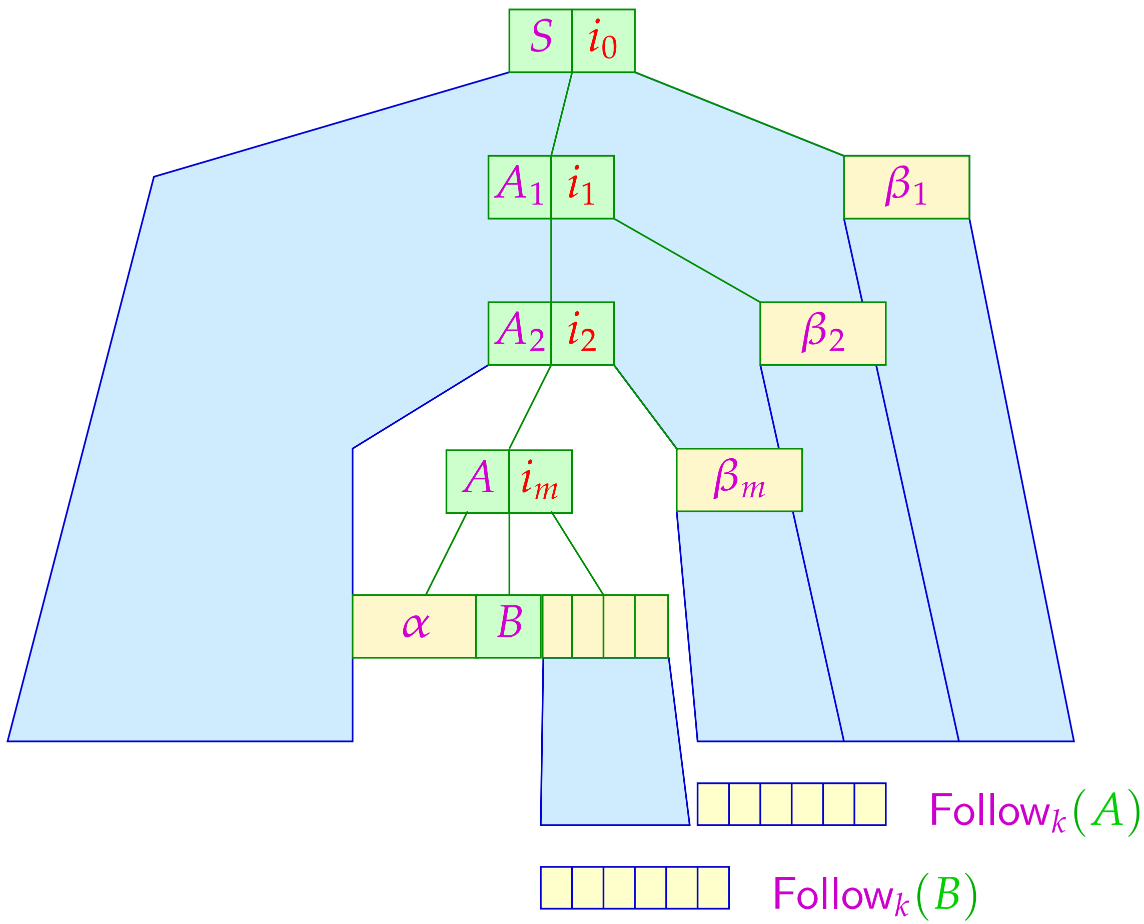
$$\begin{array}{ll} \multicolumn{2}{l}{\mathsf{First}_2(b) \odot \mathsf{Follow}_2(A) \;\cap\; \mathsf{First}_2(\epsilon) \odot \mathsf{Follow}_2(A)} \\ = & \{b\} \odot \{a\,a\,,\; b\,a\} \;\cap\; \{\epsilon\} \odot \{a\,a\,,\; b\,a\} \\ = & \{b\,a\,,\; b\,b\} \;\cap\; \{a\,a\,,\; b\,a\} \\ \neq & \emptyset \end{array}$$

Folglich ist die Grammatik nicht stark $LL(2)$:-(

Wir schließen:

- Für $k > 1$ ist nicht jede $LL(k)$-Grammatik automatisch stark $LL(k)$.
- Zu jeder $LL(k)$-Grammatik kann jedoch eine äquivalente starke $LL(k)$-Grammatik konstruiert werden $\Longrightarrow$ Übung!

Berechnung von $\text{Follow}_k(B)$:



Berechnung von $\mathsf{Follow}_k(B)$:

Idee:

- Wir stellen ein Ungleichungssystem auf :-)
- ϵ ist ein möglicher rechter Kontext von S :-)
- Mögliche rechte Kontexte der linken Seite einer Regel propagieren wir ans Ende jeder rechten Seite ...

... im Beispiel: $S \to \epsilon \quad | \quad a\,S\,b$

$$\begin{array}{lll} \mathsf{Follow}_k(S) & \supseteq & \{\epsilon\} \\ \mathsf{Follow}_k(S) & \supseteq & \{b\} \odot \mathsf{Follow}_k(S) \end{array}$$

Allgemein:

$$\begin{array}{lll} \mathsf{Follow}_k(S) & \supseteq & \{\epsilon\} \\ \mathsf{Follow}_k(B) & \supseteq & \mathsf{First}_k(X_1) \odot \ldots \odot \mathsf{First}_k(X_m) \odot \mathsf{Follow}_k(A) \\ & & \text{für} \quad A \to \alpha\, B\, X_1 \ldots X_m \in P \end{array}$$

Diskussion:

- Man überzeugt sich, dass die kleinste Lösung dieses Ungleichungssystems tatsächlich die Mengen $\mathsf{Follow}_k(B)$ liefert :-)
- Die Größe der auftretenden Mengen steigt mit k rapide :-(
- In praktischen Systemen wird darum meist nur der Fall $k = 1$ implementiert ...

2.5 Schnelle Berechnung von Vorausschau-Mengen

Im Fall $k = 1$ lassen sich First, Follow besonders effizient berechnen ;-)

Beobachtung:

Seien $L_1, L_2 \subseteq T \cup \{\epsilon\}$ mit $L_1 \neq \emptyset \neq L_2$. Dann ist:

$$L_1 \odot L_2 = \begin{cases} L_1 & \text{falls} \quad \epsilon \notin L_1 \\ (L_1 \backslash \{\epsilon\}) \cup L_2 & \text{sonst} \end{cases}$$

Ist G reduziert, sind alle Mengen $\text{First}_1(A)$ nichtleer :-)

Idee:

- Behandle ϵ separat!

  Sei $\mathsf{empty}(X) = \mathsf{true}$ gdw. $X \rightarrow^* \epsilon$.

- Definiere die ϵ-freien First_1-Mengen

$$
\begin{array}{lll}
F_\epsilon(a) & = & \{a\} \qquad\qquad\qquad \text{für} \quad a \in T \\
F_\epsilon(A) & = & \mathsf{First}_1(A) \backslash \{\epsilon\} \qquad \text{für} \quad A \in N
\end{array}
$$

- Konstruiere direkt ein Ungleichungssystem für $F_\epsilon(A)$:

$$
\begin{array}{lllll}
F_\epsilon(A) & \supseteq & F_\epsilon(X_j) & \text{falls} & A \rightarrow X_1 \ldots X_m \ \in P, \\
 & & & & \mathsf{empty}(X_1) \ \wedge \ldots \wedge \ \mathsf{empty}(X_{j-1})
\end{array}
$$

... im Beispiel:

$$\begin{array}{lllllll} E & \to & E + T & \mid & T & & \\ T & \to & T * F & \mid & F & & \\ F & \to & (\ E\ ) & \mid & \mathsf{name} & \mid & \mathsf{int} \end{array}$$

wobei $\mathsf{empty}(E) = \mathsf{empty}(T) = \mathsf{empty}(F) = \mathsf{false}$.

Deshalb erhalten wir:

$$\begin{array}{lllllll} F_\epsilon(S') & \supseteq & F_\epsilon(E) & \quad & F_\epsilon(E) & \supseteq & F_\epsilon(E) \\ F_\epsilon(E) & \supseteq & F_\epsilon(T) & & F_\epsilon(T) & \supseteq & F_\epsilon(T) \\ F_\epsilon(T) & \supseteq & F_\epsilon(F) & & F_\epsilon(F) & \supseteq & \{\ (\ , \mathsf{name}, \mathsf{int}\} \end{array}$$

Entsprechend konstruieren wir zur Berechnung von Follow_1 :

$$\begin{array}{lllll}
\mathsf{Follow}_1(S) & \supseteq & \{\epsilon\} & & \\
\mathsf{Follow}_1(B) & \supseteq & F_\epsilon(X_j) & \text{falls} & A \to \alpha\, B\, X_1 \ldots X_m \ \in P, \\
 & & & & \mathsf{empty}(X_1) \wedge \ldots \wedge \mathsf{empty}(X_{j-1}) \\
\mathsf{Follow}_1(B) & \supseteq & \mathsf{Follow}_1(A) & \text{falls} & A \to \alpha\, B\, X_1 \ldots X_m \ \in P, \\
 & & & & \mathsf{empty}(X_1) \wedge \ldots \wedge \mathsf{empty}(X_m)
\end{array}$$

Entsprechend konstruieren wir zur Berechnung von Follow_1 :

$$\begin{array}{lllll}
\mathsf{Follow}_1(S) & \supseteq & \{\epsilon\} & & \\
\mathsf{Follow}_1(B) & \supseteq & F_\epsilon(X_j) & \text{falls} & A \to \alpha\, B\, X_1 \ldots X_m \ \in P, \\
 & & & & \mathsf{empty}(X_1) \wedge \ldots \wedge \mathsf{empty}(X_{j-1}) \\
\mathsf{Follow}_1(B) & \supseteq & \mathsf{Follow}_1(A) & \text{falls} & A \to \alpha\, B\, X_1 \ldots X_m \ \in P, \\
 & & & & \mathsf{empty}(X_1) \wedge \ldots \wedge \mathsf{empty}(X_m)
\end{array}$$

... im Beispiel:

$$\begin{array}{lllllll}
E & \to & E + T & \mid & T & & \\
T & \to & T * F & \mid & F & & \\
F & \to & (\ E\ ) & \mid & \mathsf{name} & \mid & \mathsf{int}
\end{array}$$

Entsprechend konstruieren wir zur Berechnung von Follow_1 :

$$\begin{array}{lcll}
\mathsf{Follow}_1(S) & \supseteq & \{\epsilon\} & \\
\mathsf{Follow}_1(B) & \supseteq & F_\epsilon(X_j) & \text{falls} \quad A \to \alpha\, B\, X_1 \ldots X_m \in P, \\
 & & & \quad \mathsf{empty}(X_1) \wedge \ldots \wedge \mathsf{empty}(X_{j-1}) \\
\mathsf{Follow}_1(B) & \supseteq & \mathsf{Follow}_1(A) & \text{falls} \quad A \to \alpha\, B\, X_1 \ldots X_m \in P, \\
 & & & \quad \mathsf{empty}(X_1) \wedge \ldots \wedge \mathsf{empty}(X_m)
\end{array}$$

... im Beispiel:

$$\begin{array}{lcllll}
E & \to & E + T & \mid & T & \\
T & \to & T * F & \mid & F & \\
F & \to & (\ E\ ) & \mid & \mathsf{name} & \mid \ \mathsf{int}
\end{array}$$

... erhalten wir:

$$\begin{array}{lcllcl}
\mathsf{Follow}_1(S') & \supseteq & \{\epsilon\} & \mathsf{Follow}_1(E) & \supseteq & \mathsf{Follow}_1(S') \\
\mathsf{Follow}_1(E) & \supseteq & \{+,)\} & \mathsf{Follow}_1(T) & \supseteq & \{*\} \\
\mathsf{Follow}_1(T) & \supseteq & \mathsf{Follow}_1(E) & \mathsf{Follow}_1(F) & \supseteq & \mathsf{Follow}_1(T)
\end{array}$$

Diskussion:

- Diese Ungleichungssysteme bestehen aus Ungleichungen der Form:

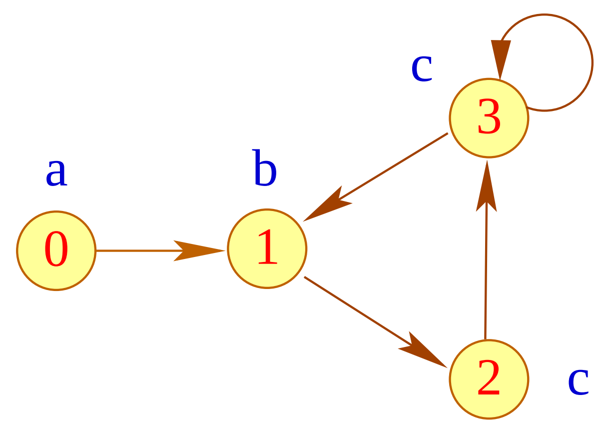
$$x \sqsupseteq y \qquad \text{bzw.} \qquad x \sqsupseteq d$$

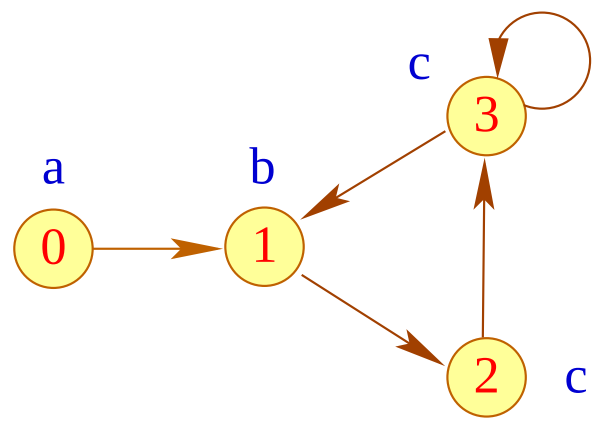
  für Variablen x, y und $d \in \mathbb{D}$.

- Solche Ungleichungssysteme heißen reine Vereinigungs-Probleme :-)
- Diese Probleme können mit linearem Aufwand gelöst werden ...

Beispiel: $\mathbb{D} = 2^{\{a,b,c\}}$

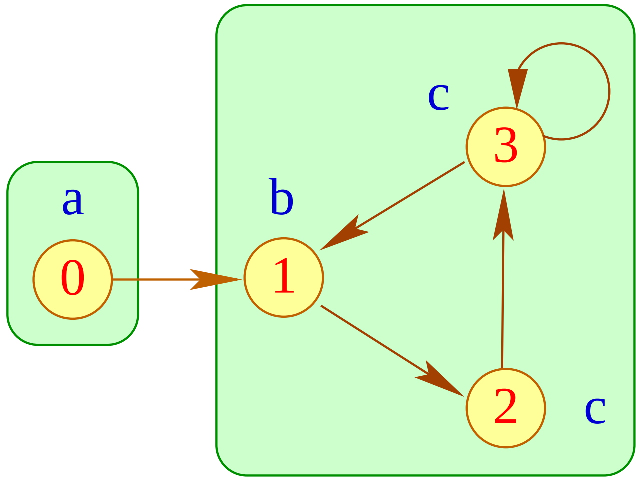
$$\begin{array}{lll}
x_0 \supseteq \{a\} & & \\
x_1 \supseteq \{b\} & x_1 \supseteq x_0 & x_1 \supseteq x_3 \\
x_2 \supseteq \{c\} & x_2 \supseteq x_1 & \\
x_3 \supseteq \{c\} & x_3 \supseteq x_2 & x_3 \supseteq x_3
\end{array}$$





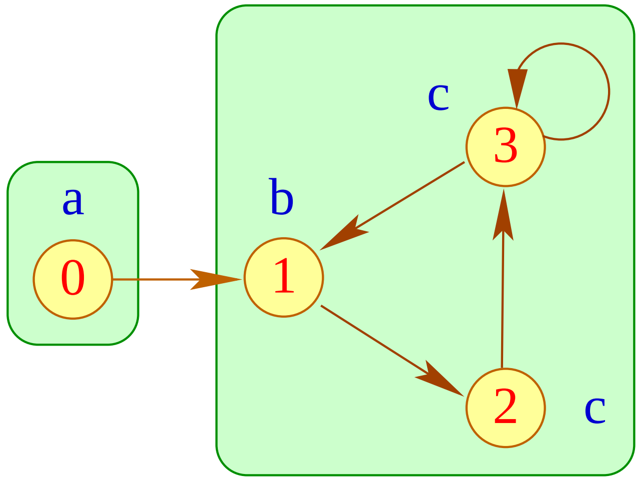
Vorgehen:

- Konstruiere den Variablen-Abhängigkeitsgraph zum Ungleichungssystem.



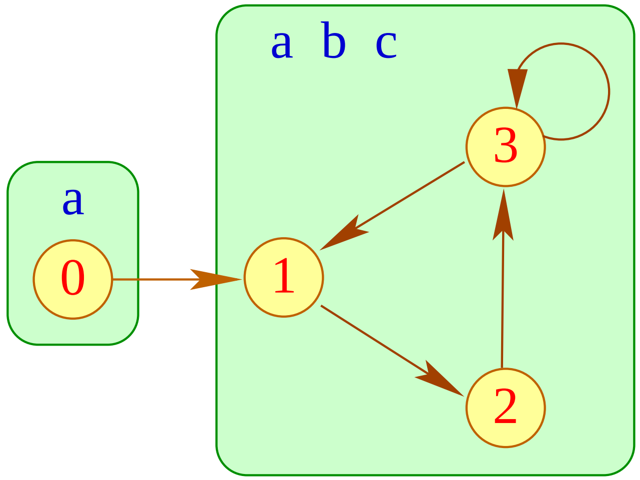
Vorgehen:

- Konstruiere den Variablen-Abhängigkeitsgraph zum Ungleichungssystem.
- Innerhalb einer starken Zusammenhangskomponente haben alle Variablen den gleichen Wert :-)



Vorgehen:

- Konstruiere den Variablen-Abhängigkeitsgraph zum Ungleichungssystem.
- Innerhalb einer starken Zusammenhangskomponente haben alle Variablen den gleichen Wert :-)
- Hat eine SZK keine eingehenden Kanten, erhält man ihren Wert, indem man die kleinste obere Schranke aller Werte in der SZK berechnet :-)

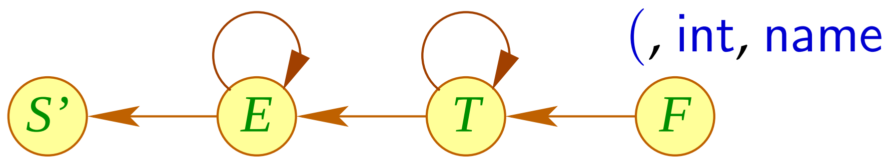


Vorgehen:

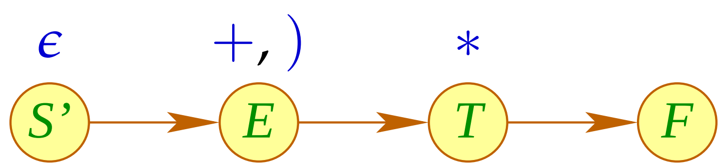
- Konstruiere den Variablen-Abhängigkeitsgraph zum Ungleichungssystem.
- Innerhalb einer starken Zusammenhangskomponente haben alle Variablen den gleichen Wert :-)
- Hat eine SZK keine eingehenden Kanten, erhält man ihren Wert, indem man die kleinste obere Schranke aller Werte in der SZK berechnet :-)
- Gibt es eingehende Kanten, muss man zusätzlich die Werte an deren Startknoten hinzufügen :-)

... für unsere Beispiel-Grammatik:

First_1 :



Follow_1 :



2.6 Bottom-up Analyse

Achtung:

- Viele Grammatiken sind nicht $LL(k)$:-(
- Eine Grund ist Links-Rekursivität ...
- Die Grammatik G heißt links-rekursiv, falls

$$A \rightarrow^{+} A\,\beta \qquad \text{für ein} \quad A \in N,\ \beta \in (T \cup N)^*$$

2.6 Bottom-up Analyse

Achtung:

- Viele Grammatiken sind nicht $LL(k)$:-(
- Eine Grund ist Links-Rekursivität ...
- Die Grammatik G heißt links-rekursiv, falls

$$A \rightarrow^{+} A\,\beta \qquad \text{für ein} \quad A \in N\,,\ \beta \in (T \cup N)^*$$

Beispiel:

$$\begin{array}{lcl} E & \rightarrow & E + T \quad \mid \quad T \\ T & \rightarrow & T * F \quad \mid \quad F \\ F & \rightarrow & (\ E\ ) \quad \mid \quad \text{name} \quad \mid \quad \text{int} \end{array}$$

... ist links-rekursiv :-)

Satz

Ist die Grammatik G reduziert und links-rekursiv, dann ist G nicht $LL(k)$ für jedes k .

Satz

Ist die Grammatik G reduziert und links-rekursiv, dann ist G nicht $LL(k)$ für jedes k .

Beweis: Vereinfachung: $A \to A\,\beta \in P$

A erreichbar $\Longrightarrow$ $S \to_L^* u\,A\,\gamma \to_L^* u\,A\,\beta^n\,\gamma$ für jedes $n \geq 0$.

A produktiv $\Longrightarrow$ $\exists\, A \to \alpha \,:\, \alpha \neq A\,\beta$.

Satz

Ist die Grammatik G reduziert und links-rekursiv, dann ist G nicht $LL(k)$ für jedes k .

Beweis:

Vereinfachung: $A \to A\,\beta \;\in\; P$

A erreichbar $\Longrightarrow$ $S \to_L^* u\,A\,\gamma \to_L^* u\,A\,\beta^n\,\gamma$ für jedes $n \geq 0$.

A produktiv $\Longrightarrow$ $\exists\, A \to \alpha \;:\; \alpha \neq A\,\beta$.

Annahme: G ist $LL(k)$;-) Dann gilt für alle $n \geq 0$:

$$\mathsf{First}_k(\alpha\,\beta^n\,\gamma) \quad\cap\quad \mathsf{First}_k(A\,\beta\,\beta^n\,\gamma) \quad=\quad \emptyset$$

Weil $\quad \mathsf{First}_k(\alpha\,\beta^{n+1}\,\gamma) \quad\subseteq\quad \mathsf{First}_k(A\,\beta^{n+1}\,\gamma)$

folgt: $\quad \mathsf{First}_k(\alpha\,\beta^n\,\gamma) \quad\cap\quad \mathsf{First}_k(\alpha\,\beta^{n+1}\,\gamma) \quad=\quad \emptyset$

Satz

Ist die Grammatik G reduziert und links-rekursiv, dann ist G nicht $LL(k)$ für jedes k .

Beweis:

Vereinfachung: $A \to A\,\beta \in P$

A erreichbar $\Longrightarrow$ $S \to_L^* u\,A\,\gamma \to_L^* u\,A\,\beta^n\,\gamma$ für jedes $n \geq 0$.

A produktiv $\Longrightarrow$ $\exists\, A \to \alpha \,:\, \alpha \neq A\,\beta$.

Annahme: G ist $LL(k)$;-) Dann gilt für alle $n \geq 0$:

	$\text{First}_k(\alpha\,\beta^n\,\gamma)$	$\cap$	$\text{First}_k(A\,\beta\,\beta^n\,\gamma)$	$=$	$\emptyset$
Weil	$\text{First}_k(\alpha\,\beta^{n+1}\,\gamma)$	$\subseteq$	$\text{First}_k(A\,\beta^{n+1}\,\gamma)$		
folgt:	$\text{First}_k(\alpha\,\beta^n\,\gamma)$	$\cap$	$\text{First}_k(\alpha\,\beta^{n+1}\,\gamma)$	$=$	$\emptyset$

Fall 1: $\beta \to^* \epsilon$ — Widerspruch !!!

Fall 2: $\beta \to^* w \neq \epsilon$ $\Longrightarrow$ $\text{First}_k(\alpha\,\beta^k\,\gamma) \cap \text{First}_k(\alpha\,\beta^{k+1}\,\gamma) \neq \emptyset$:-(

Bottom-up Parsing:

Wir rekonstruieren reverse Rechtsableitungen :-)

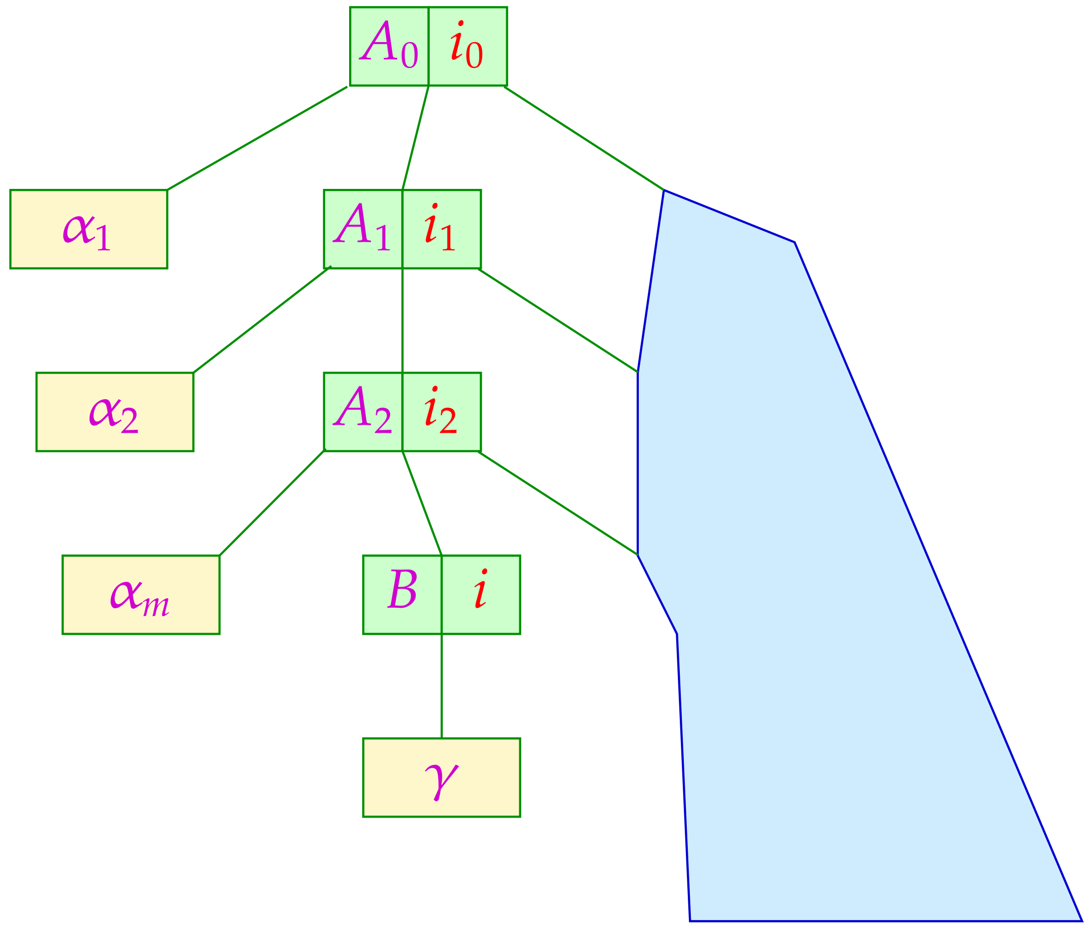
Dazu versuchen wir, in $M_G^{(1)}$ Reduktionsstellen zu entdecken.

Betrachte eine Berechnung dieses Kellerautomaten:

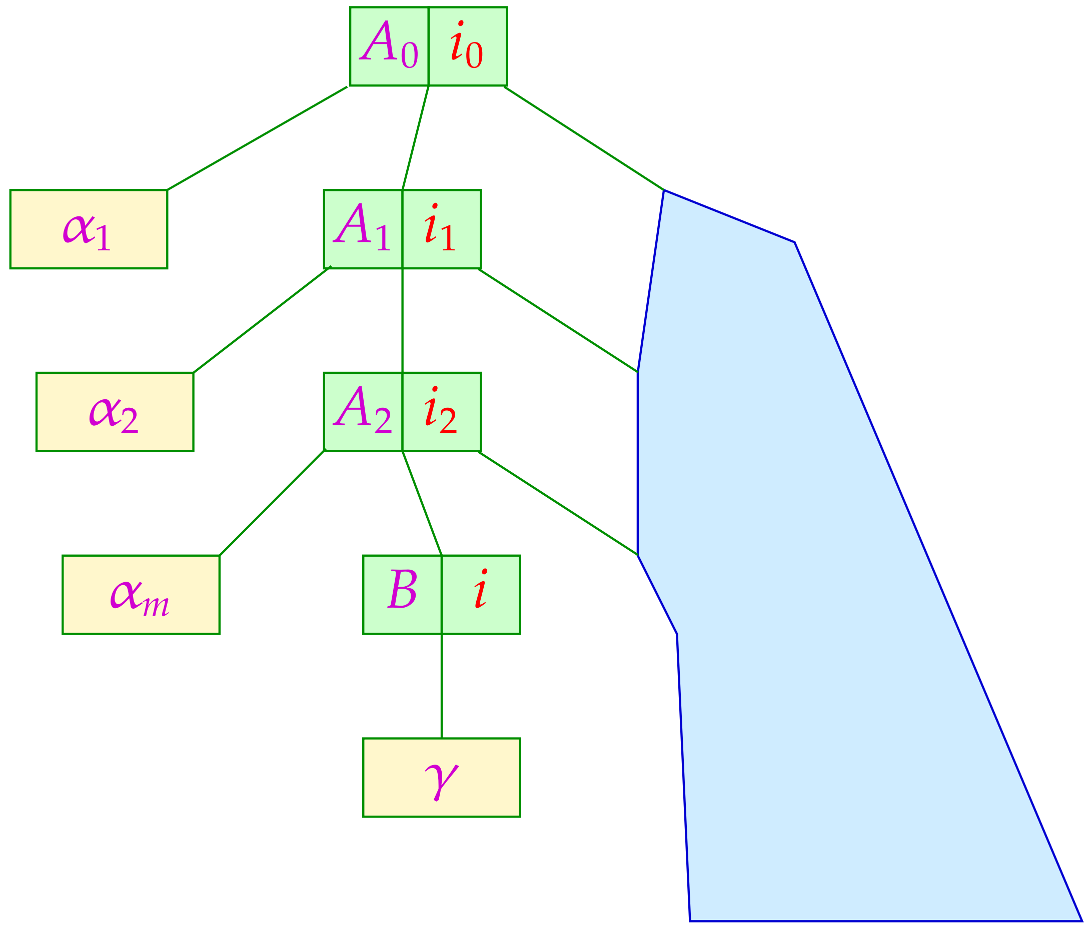
$$(q_0\,\alpha\,\gamma,\; v) \vdash (q_0\,\alpha\,B,\; v) \vdash^* (q_0\,S,\; \epsilon)$$

$\alpha\,\gamma$ nennen wir zuverlässiges Präfix für das vollständige Item $[B \to \gamma\bullet]$.

Dann ist $\alpha\,\gamma$ zuverlässig für $[B \to \gamma\bullet]$ gdw. $S \to_R^* \alpha\,B\,v$:-)



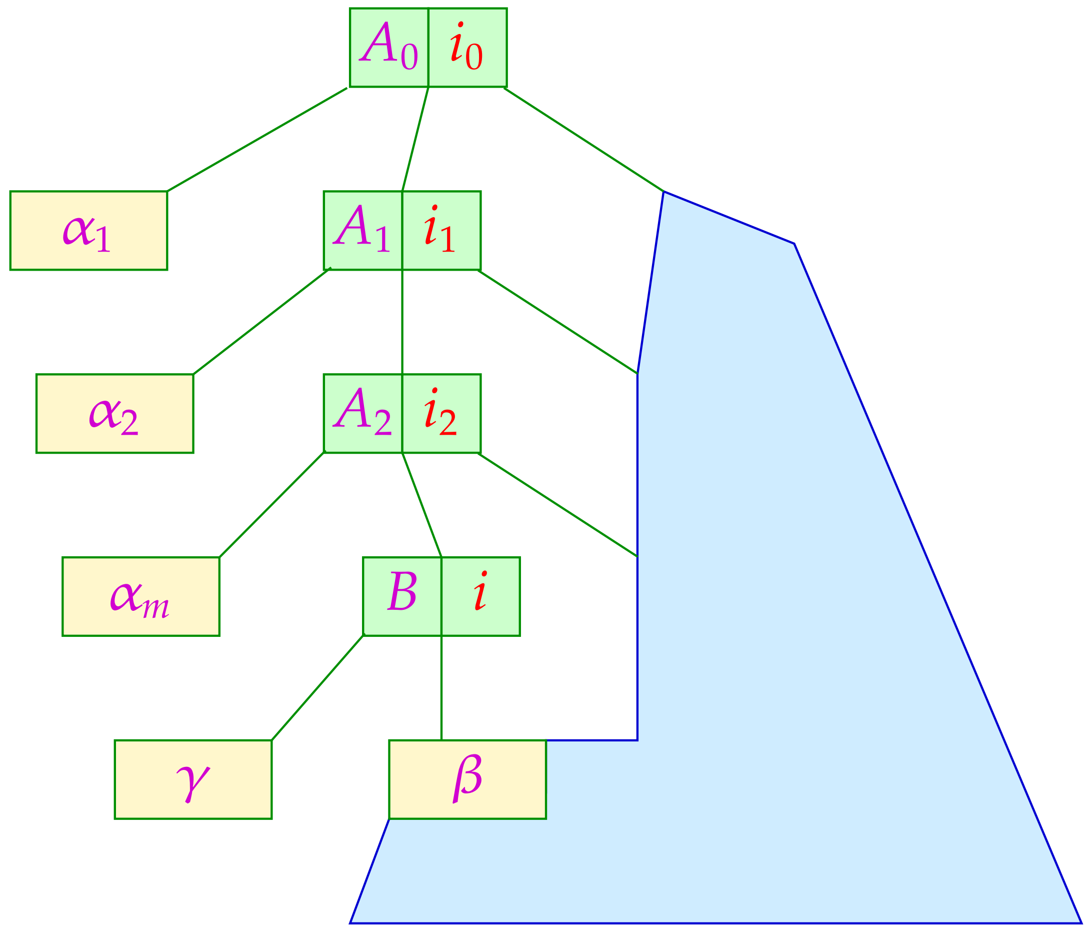
... wobei $\quad \alpha = \alpha_1 \; \ldots \; \alpha_m \quad$:-)



... wobei $\alpha = \alpha_1 \ldots \alpha_m$:-)

Umgekehrt können wir zu jedem möglichen Wort α' die Menge aller möglicherweise später passenden Regeln ermitteln ...

Das Item $[B \to \gamma \bullet \beta]$ heißt gültig für α' gdw. $S \to_R^* \alpha\, B\, v$ mit $\alpha' = \alpha\,\gamma$:



... wobei $\alpha = \alpha_1 \ldots \alpha_m$:-)

Beobachtung:

Die Menge der zuverlässigen Präfixe aus $(N \cup T)^*$ für (vollständige) Items kann mithilfe eines endlichen Automaten berechnet werden :-)

Zustände: Items :-)

Anfangszustand: $[S' \to \bullet\, S]$

Endzustände: $\{[B \to \gamma \bullet] \mid B \to \gamma \in P\}$

Übergänge:

(1) $([A \to \alpha \bullet X\, \beta], X, [A \to \alpha\, X \bullet \beta])$, $X \in (N \cup T), A \to \alpha\, X\, \beta \in P$;

(2) $([A \to \alpha \bullet B\, \beta], \epsilon, [B \to \bullet\, \gamma])$, $A \to \alpha\, B\, \beta\,,\ B \to \gamma \in P$;

Den Automaten $c(G)$ nennen wir charakteristischen Automaten für G.

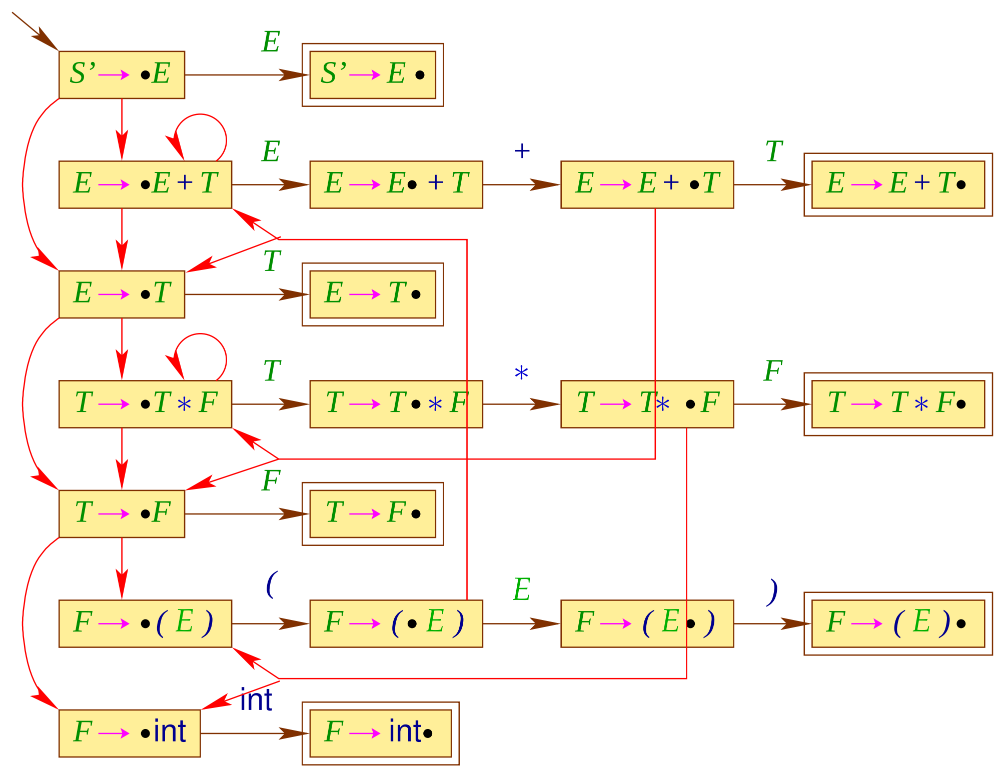
Beispiel:

$$
\begin{array}{lcll}
E & \rightarrow & E + T & \mid \; T \\
T & \rightarrow & T * F & \mid \; F \\
F & \rightarrow & (\, E \,) & \mid \; \text{int}
\end{array}
$$

$$
\begin{tikzcd}
\boxed{S' \rightarrow \bullet E} \arrow[r, "E"] & \boxed{S' \rightarrow E \bullet} & & \\
\boxed{E \rightarrow \bullet E + T} \arrow[r, "E"] & \boxed{E \rightarrow E \bullet + T} \arrow[r, "+"] & \boxed{E \rightarrow E + \bullet T} \arrow[r, "T"] & \boxed{E \rightarrow E + T \bullet} \\
\boxed{E \rightarrow \bullet T} \arrow[r, "T"] & \boxed{E \rightarrow T \bullet} & & \\
\boxed{T \rightarrow \bullet T * F} \arrow[r, "T"] & \boxed{T \rightarrow T \bullet * F} \arrow[r, "*"] & \boxed{T \rightarrow T * \bullet F} \arrow[r, "F"] & \boxed{T \rightarrow T * F \bullet} \\
\boxed{T \rightarrow \bullet F} \arrow[r, "F"] & \boxed{T \rightarrow F \bullet} & & \\
\boxed{F \rightarrow \bullet (\, E \,)} \arrow[r, "("] & \boxed{F \rightarrow (\, \bullet E \,)} \arrow[r, "E"] & \boxed{F \rightarrow (\, E \bullet \,)} \arrow[r, ")"] & \boxed{F \rightarrow (\, E \,) \bullet} \\
\boxed{F \rightarrow \bullet \text{int}} \arrow[r, "\text{int}"] & \boxed{F \rightarrow \text{int} \bullet} & &
\end{tikzcd}
$$

Beispiel:

$$
\begin{array}{lcll}
E & \rightarrow & E + T & \mid \; T \\
T & \rightarrow & T * F & \mid \; F \\
F & \rightarrow & (\, E \,) & \mid \; \mathsf{int}
\end{array}
$$

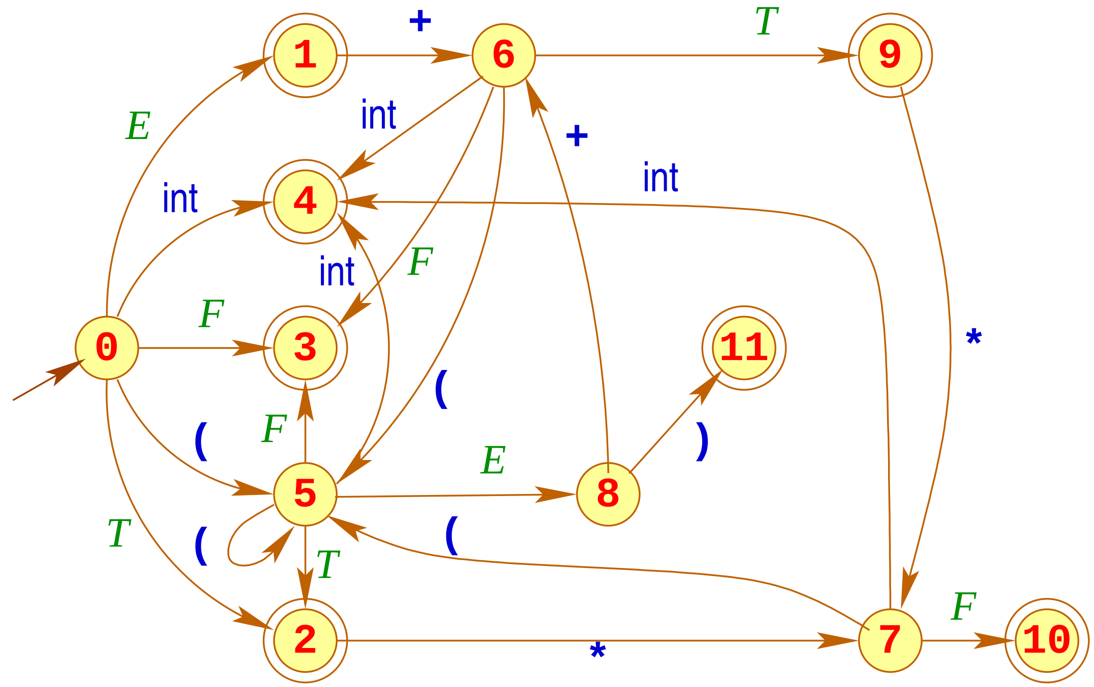


Den kanonischen $LR(0)$-Automaten $LR(G)$ erhalten wir aus $c(G)$, indem wir:

(1) nach jedem lesenden Übergang beliebig viele ϵ-Übergänge einschieben (unsere Konstruktion 1 zur Beseitigung von ϵ-Übergängen :-)

(2) die Teilmengenkonstruktion anwenden.

... im Beispiel:



Dazu konstruieren wir:

$$\begin{array}{lll} q_0 & = & \{[S' \to \bullet\, E], \\ & & [E \to \bullet\, E + T], \\ & & [E \to \bullet\, T], \\ & & [T \to \bullet\, T * F]\} \\ & & [T \to \bullet\, F], \\ & & [F \to \bullet\, (\, E\,)\,], \\ & & [F \to \bullet\, \mathsf{int}]\} \end{array}$$

$$\begin{array}{lllll} q_1 & = & \delta(q_0, E) & = & \{[S' \to E \bullet], \\ & & & & [E \to E \bullet + T]\} \\ \\ q_2 & = & \delta(q_0, T) & = & \{[E \to T \bullet], \\ & & & & [T \to T \bullet * F]\} \\ \\ q_3 & = & \delta(q_0, F) & = & \{[T \to F \bullet]\} \\ \\ q_4 & = & \delta(q_0, \mathsf{int}) & = & \{[F \to \mathsf{int} \bullet]\} \end{array}$$

$$
\begin{aligned}
q_5 &= \delta(q_0, (\,) &= \{&[F \to (\, \bullet E\,)\,],\\
&&&[E \to \bullet E + T],\\
&&&[E \to \bullet T],\\
&&&[T \to \bullet T * F],\\
&&&[T \to \bullet F],\\
&&&[F \to \bullet (\, E\,)\,],\\
&&&[F \to \bullet \mathsf{int}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_6 &= \delta(q_1, +) &= \{&[E \to E + \bullet T],\\
&&&[T \to \bullet T * F],\\
&&&[T \to \bullet F],\\
&&&[F \to \bullet (\, E\,)\,],\\
&&&[F \to \bullet \mathsf{int}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_7 &= \delta(q_2, *) &= \{&[T \to T * \bullet F],\\
&&&[F \to \bullet (\, E\,)\,],\\
&&&[F \to \bullet \mathsf{int}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_8 &= \delta(q_5, E) &= \{&[F \to (\, E \bullet\,)\,]\}\\
&&&[E \to E \bullet + T]\}
\end{aligned}
$$

$$
\begin{aligned}
q_9 &= \delta(q_6, T) &= \{&[E \to E + T\bullet],\\
&&&[T \to T \bullet * F]\}
\end{aligned}
$$

$$
q_{10} = \delta(q_7, F) = \{[T \to T * F\bullet]\}
$$

$$
q_{11} = \delta(q_8, \,)\,) = \{[F \to (\, E\,) \bullet]\}
$$

Beachte:

Der kanonische $LR(0)$-Automat kann auch direkt aus der Grammatik konstruiert werden :-)

Man benötigt die Hilfsfunktion:

$$\delta^*_\epsilon(q) = q \cup \{[B \to \bullet\,\gamma] \mid \exists\, [A \to \alpha \bullet B'\,\beta'] \in q\,, \quad \beta \in (N \cup T)^* : \quad B' \to^* B\,\beta\}$$

Zustände: Mengen von Items;

Anfangszustand: $\delta^*_\epsilon\,\{[S' \to \bullet\, S]\}$

Endzustände: $\{q \mid \exists\, A \to \alpha \in P : \; [A \to \alpha\,\bullet] \in q\}$

Übergänge: $\delta(q, X) = \delta^*_\epsilon\,\{[A \to \alpha\, X \bullet\, \beta] \mid [A \to \alpha \bullet X\,\beta] \in q\}$

Idee zu einem Parser:

- Der Parser verwaltet ein zuverlässiges Präfix $\alpha = X_1 \ldots X_m$ auf dem Keller und benutzt $LR(G)$, um Reduktionsstellen zu entdecken.
- Er kann mit einer Regel $A \to \gamma$ reduzieren, falls $[A \to \gamma \bullet]$ für α gültig ist :-)
- Damit der Automat nicht immer wieder neu über den Kellerinhalt laufen muss, kellern wir anstelle der X_i jeweils die Zustände !!!

Achtung:

Dieser Parser ist nur dann deterministisch, wenn jeder Endzustand des kanonischen $LR(0)$-Automaten keine Konflikte enthält ...

... im Beispiel:

$$
\begin{array}{lll}
q_1 & = & \{[S' \to E\,\bullet], \\
 & & \;[E \to E\,\bullet\,+T]\}
\end{array}
$$

$$
\begin{array}{lllllll}
q_2 & = & \{[E \to T\,\bullet], & \qquad & q_9 & = & \{[E \to E+T\,\bullet], \\
 & & \;[T \to T\,\bullet * F]\} & & & & \;[T \to T\,\bullet * F]\} \\
 & & & & & & \\
q_3 & = & \{[T \to F\,\bullet]\} & & q_{10} & = & \{[T \to T * F\,\bullet]\} \\
 & & & & & & \\
q_4 & = & \{[F \to \mathsf{int}\,\bullet]\} & & q_{11} & = & \{[F \to (\,E\,)\,\bullet]\}
\end{array}
$$

Die Endzustände q_1, q_2, q_9 enthalten mehr als ein Item :-(

Aber wir haben ja auch noch nicht Vorausschau eingesetzt :-)

Die Konstruktion des $LR(0)$-Parsers:

Zustände: $Q \cup \{f\}$ $\quad$ (f neu :-)

Anfangszustand: q_0

Endzustand: f

Übergänge:

Shift:	$(p, a, p\,q)$	falls	$q = \delta(p, a) \neq \emptyset$
Reduce:	$(p\,q_1 \ldots q_m, \epsilon, p\,q)$	falls	$[A \to X_1 \ldots X_m \bullet] \in q_m$,
			$q = \delta(p, A)$
Finish:	$(q_0\,p, \epsilon, f)$	falls	$[S' \to S \bullet] \in p$

wobei $\quad LR(G) = (Q, T, \delta, q_0, F)$.

Zur Korrektheit:

Man zeigt:

Die akzeptierenden Berechnungen des $LR(0)$-Parsers stehen in eins-zu-eins Beziehung zu denen des Kellerautomaten $M_G^{(1)}$.

Wir folgern:

$\Longrightarrow$ Die akzeptierte Sprache ist genau $\mathcal{L}(G)$:-)

$\Longrightarrow$ Die Folge der Reduktionen einer akzeptierenden Berechnung für ein Wort $w \in T$ liefert eine reverse Rechts-Ableitung von G für w :-)

Leider ist der $LR(0)$-Parser i.a. nicht-deterministisch :-(

Wir identifizieren zwei Gründe:

Reduce-Reduce-Konflikt:

$$[A \to \gamma \bullet] \,,\; [A' \to \gamma' \bullet] \quad \in \quad q \quad \text{mit} \quad A \neq A' \vee \gamma \neq \gamma'$$

Shift-Reduce-Konflikt:

$$[A \to \gamma \bullet] \,,\; [A' \to \alpha \bullet a\, \beta] \quad \in \quad q \quad \text{mit} \quad a \in T$$

für einen Zustand $q \in Q$.

Solche Zustände nennen wir ungeeignet.

Idee:

Benutze k-Vorausschau, um Konflikte zu lösen.

Wir definieren:

Die reduzierte kontextfreie Grammatik G heißt $LR(k)$-Grammatik, falls für $\mathsf{First}_k(w) = \mathsf{First}_k(x)$ aus:

$$\left.\begin{array}{lllll} S & \rightarrow_R^* & \alpha\, A\, w & \rightarrow & \alpha\, \beta\, w \\ S & \rightarrow_R^* & \alpha'\, A'\, w' & \rightarrow & \alpha\, \beta\, x \end{array}\right\} \quad \text{folgt:} \quad \alpha = \alpha' \;\wedge\; A = A' \;\wedge\; w' = x$$

Beispiele:

(1) $S \rightarrow A \mid B \qquad A \rightarrow a\,A\,b \mid 0 \qquad B \rightarrow a\,B\,b\,b \mid 1$

... ist nicht $LL(k)$ für jedes k — aber $LR(0)$:

Sei $S \rightarrow^*_R \alpha\,X\,w \rightarrow \alpha\,\beta\,w$. Dann ist $\alpha\,\underline{\beta}$ von einer der Formen:

$$\underline{A}\;,\;\underline{B}\;,\;a^n\,\underline{a\,A\,b}\;,\;a^n\,\underline{a\,B\,b\,b}\;,\;a^n\,\underline{0}\;,\;a^n\,\underline{1} \qquad (n \geq 0)$$

Beispiele:

(1) $S \to A \mid B \qquad A \to a\,A\,b \mid 0 \qquad B \to a\,B\,b\,b \mid 1$

... ist nicht $LL(k)$ für jedes k — aber $LR(0)$:

Sei $S \to_R^* \alpha\,X\,w \to \alpha\,\beta\,w$. Dann ist $\alpha\,\underline{\beta}$ von einer der Formen:

$$\underline{A}\,,\ \underline{B}\,,\ a^n\,\underline{a\,A\,b}\,,\ a^n\,\underline{a\,B\,b\,b}\,,\ a^n\,\underline{0}\,,\ a^n\,\underline{1} \qquad (n \geq 0)$$

(2) $S \to a\,A\,c \qquad A \to A\,b\,b \mid b$

... ist ebenfalls $LR(0)$:

Sei $S \to_R^* \alpha\,X\,w \to \alpha\,\beta\,w$. Dann ist $\alpha\,\underline{\beta}$ von einer der Formen:

$$a\,\underline{b}\,,\ a\,\underline{A\,b\,b}\,,\ \underline{a\,A\,c}$$

(3) $S \to a\, A\, c \qquad A \to b\, b\, A \;\mid\; b$ ... ist nicht $LR(0)$, aber $LR(1)$:

Für $S \to_R^* \alpha\, X\, w \to \alpha\, \beta\, w$ mit $\{y\} = \mathsf{First}_k(w)$ ist $\alpha\, \underline{\beta}\, y$ von einer der Formen:

$$a\, b^{2n}\, \underline{b}\, c \;,\; a\, b^{2n}\, \underline{b\, b\, A}\, c \;,\; \underline{a\, A\, c}$$

(3) $S \rightarrow a\, A\, c \qquad A \rightarrow b\, b\, A \;\mid\; b$ ... ist nicht $LR(0)$, aber $LR(1)$:

Für $S \rightarrow_R^* \alpha\, X\, w \rightarrow \alpha\, \beta\, w$ mit $\{y\} = \text{First}_k(w)$ ist $\alpha\, \underline{\beta}\, y$ von einer der Formen:

$$a\, b^{2n}\, \underline{b}\, c \;,\; a\, b^{2n}\, \underline{b\, b\, A}\, c \;,\; \underline{a\, A\, c}$$

(4) $S \rightarrow a\, A\, c \qquad A \rightarrow b\, A\, b \;\mid\; b$ ... ist nicht $LR(k)$ für jedes $k \geq 0$:

Betrachte einfach die Rechtsableitungen:

$$S \rightarrow_R^* a\, b^n\, A\, b^n\, c \rightarrow a\, b^n\, \underline{b}\, b^n\, c$$

In der Tat gilt:

Satz:

Die reduzierte Grammatik G ist genau dann $LR(0)$ wenn der kanonische $LR(0)$-Automat $LR(G)$ keine ungeeigneten Zustände enthält.

In der Tat gilt:

Satz:

Die reduzierte Grammatik G ist genau dann $LR(0)$ wenn der kanonische $LR(0)$-Automat $LR(G)$ keine ungeeigneten Zustände enthält.

Beweis:

Enthalte G einen ungeeigneten Zustand q.

Fall 1: $[A \to \gamma\bullet], [A' \to \gamma'\bullet] \in q$ mit $A \to \gamma \neq A' \to \gamma'$

Dann gibt es ein zuverlässiges Präfix $\alpha\,\gamma = \alpha'\,\gamma'$ mit

$$S \to_R^* \alpha\,A\,w \to \alpha\,\gamma\,w \quad \wedge \quad S \to_R^* \alpha'\,A'\,x \to \alpha'\,\gamma'\,x$$

$\Longrightarrow$ G ist nicht $LR(0)$:-)

Fall 2: $[A \to \gamma \bullet]\,,\ [A' \to \beta \bullet a\, \beta'] \ \in\ q$

Dann gibt es ein zuverlässiges Präfix $\alpha\,\gamma = \alpha'\,\beta$ mit

$$S \to_R^* \alpha\, A\, w \to \alpha\,\gamma\, w \quad \wedge \quad S \to_R^* \alpha'\, A'\, x \to \alpha'\,\beta\, a\,\beta'\, x$$

Ist $\beta' \in T^*$, dann ist G nicht $LR(0)$:-)

Andernfalls $\beta' \to_R^* v_1\, X\, v_2 \to v_1\, u\, v_2$. Damit erhalten wir:

$$S \to_R^* \alpha'\,\beta\, a\, v_1\, X\, v_2\, x \to \alpha'\,\beta\, a\, v_1\, u\, v_2\, x$$

$$\Longrightarrow \qquad G \quad \text{ist nicht } LR(0) \quad \text{:-)}$$

Fall 2: $[A \to \gamma \bullet]\,,\ [A' \to \beta \bullet a\, \beta'] \ \in\ q$

Dann gibt es ein zuverlässiges Präfix $\alpha\,\gamma = \alpha'\,\beta$ mit

$$S \to_R^* \alpha\, A\, w \to \alpha\,\gamma\, w \qquad \wedge \qquad S \to_R^* \alpha'\, A'\, x \to \alpha'\,\beta\, a\, \beta'\, x$$

Ist $\beta' \in T^*$, dann ist G nicht $LR(0)$:-)

Andernfalls $\beta' \to_R^* v_1\, X\, v_2 \to v_1\, u\, v_2$. Damit erhalten wir:

$$S \to_R^* \alpha'\,\beta\, a\, v_1\, X\, v_2\, x \to \alpha'\,\beta\, a\, v_1\, u\, v_2\, x$$

$$\Longrightarrow \qquad G \quad \text{ist nicht } LR(0) \quad \text{:-)}$$

Enthalte $LR(G)$ keine ungeeigneten Zustände. Betrachte:

$$S \to_R^* \alpha\, A\, w \to \alpha\,\gamma\, w \qquad\qquad S \to_R^* \alpha'\, A'\, w' \to \alpha'\,\gamma'\, x$$

Sei $\delta(q_0, \alpha\,\gamma) = q$. Insbesondere ist $[A \to \gamma \bullet] \in q$.

Annahme: $(\alpha, A, w') \neq (\alpha', A', x)$.

Fall 1: $w' = x$. Dann muss q $[A' \to \gamma' \bullet]$ enthalten :-)

Fall 2: $w' \neq x$. Weitere Fallunterscheidung :-))

Sei $k > 0$.

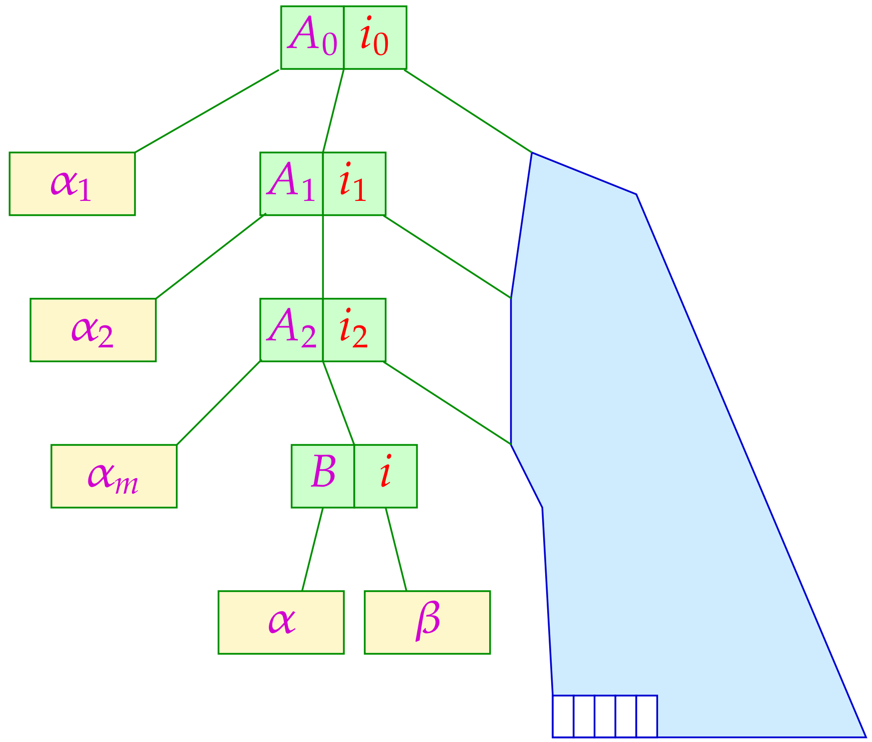
Idee: Wir statten Items mit k-Vorausschau aus :-)

Ein $LR(k)$-Item ist dann ein Paar:

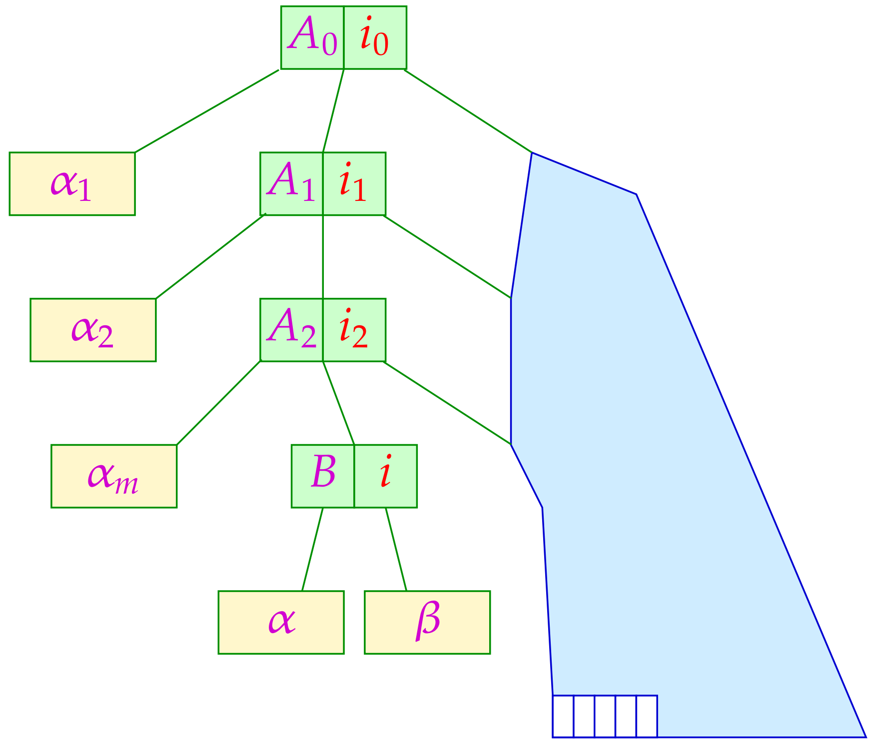
$$[B \to \alpha \bullet \beta, x] \, , \quad x \in \text{Follow}_k(B)$$

Dieses Item ist gültig für $\gamma \alpha$ falls:

$$S \to_R^* \gamma B w \qquad \text{mit} \qquad \{x\} = \text{First}_k(w)$$



... wobei $\alpha_1 \dots \alpha_m = \gamma$



... wobei $\quad \alpha_1 \ldots \alpha_m = \gamma$

Die Menge der gültigen $LR(k)$-Items für zuverlässige Präfixe berechnen wir wieder mithilfe eines endlichen Automaten :-)

Der Automat $c(G, k)$:

Zustände: $LR(k)$-Items :-)

Anfangszustand: $[S' \to \bullet\, S,\ \epsilon]$

Endzustände: $\{[B \to \gamma\bullet,\ x] \mid B \to \gamma\ \in\ P, x\ \in\ \textsf{Follow}_k(B)\}$

Übergänge:

(1) $([A \to \alpha \bullet X\, \beta,\ x], X, [A \to \alpha\, X \bullet \beta,\ x]),\quad X\ \in\ (N \cup T)$

(2) $([A \to \alpha \bullet B\, \beta,\ x], \epsilon,\ [B \to\ \bullet\, \gamma,\ x']),$

$$A \to \alpha\, B\, \beta\ ,\quad B \to \gamma\ \in\ P, x'\ \in\ \textsf{First}_k(\beta) \odot \{x\};$$

Der Automat $c(G, k)$:

Zustände: $LR(k)$-Items :-)

Anfangszustand: $[S' \to \bullet\, S,\ \epsilon]$

Endzustände: $\{[B \to \gamma\bullet,\ x] \mid B \to \gamma \in P, x \in \textsf{Follow}_k(B)\}$

Übergänge:

(1) $([A \to \alpha \bullet X\, \beta,\ x], X, [A \to \alpha\, X \bullet \beta,\ x]), \quad X \in (N \cup T)$

(2) $([A \to \alpha \bullet B\, \beta,\ x], \epsilon, [B \to \bullet\, \gamma,\ x']),$

$$A \to \alpha\, B\, \beta\ , \quad B \to \gamma \in P, x' \in \textsf{First}_k(\beta) \odot \{x\};$$

Dieser Automat arbeitet wie $c(G)$ — verwaltet aber zusätzlich ein k-Präfix aus dem $\textsf{Follow}_k$ der linken Seiten.

Den kanonischen $LR(k)$-Automaten $LR(G,k)$ erhält man aus $c(G,k)$, indem man nach jedem Übergang beliebig viele ϵ liest und dann den Automaten deterministisch macht ...

Den kanonischen $LR(k)$-Automaten $LR(G, k)$ erhält man aus $c(G, k)$, indem man nach jedem Übergang beliebig viele ϵ liest und dann den Automaten deterministisch macht ...

Man kann ihn aber auch direkt aus der Grammatik konstruieren werden :-)

Wie bei $LR(0)$ benötigt man eine Hilfsfunktion:

$$\begin{array}{lll} \delta^*_\epsilon(q) = q \cup \{[B \to \bullet\, \gamma, x] \mid & \exists\, [A \to \alpha \bullet B'\, \beta', x'] \in q\,, \\ & \beta \in (N \cup T)^* : \quad B' \to^* B\, \beta\} \wedge \\ & x \in \mathsf{First}_k(\beta\, \beta') \odot \{x'\}\} \end{array}$$

Den kanonischen $LR(k)$-Automaten $LR(G,k)$ erhält man aus $c(G,k)$, indem man nach jedem Übergang beliebig viele ϵ liest und dann den Automaten deterministisch macht ...

Man kann ihn aber auch direkt aus der Grammatik konstruieren werden :-)

Wie bei $LR(0)$ benötigt man eine Hilfsfunktion:

$$\begin{array}{rl} \delta^*_\epsilon(q) = q \cup \{[B \to \bullet\, \gamma,\, x] \mid & \exists\, [A \to \alpha \bullet B'\, \beta',\, x'] \in q\,, \\ & \beta \in (N \cup T)^* : \quad B' \to^* B\, \beta\} \wedge \\ & x \in \text{First}_k(\beta\, \beta') \odot \{x'\}\} \end{array}$$

Zustände: Mengen von $LR(k)$-Items;

Anfangszustand: $\delta^*_\epsilon\, \{[S' \to \bullet\, S,\, \epsilon]\}$

Endzustände: $\{q \mid \exists\, A \to \alpha \in P : \;\; [A \to \alpha \bullet,\, x] \in q\}$

Übergänge: $\delta(q, X) = \delta^*_\epsilon\, \{[A \to \alpha\, X \bullet\, \beta,\, x] \mid [A \to \alpha \bullet X\, \beta,\, x] \in q\}$

Im Beispiel:

$$
\begin{array}{lllll}
q_0 & = & & & \{[S' \to \bullet\, E],\\
 & & & & [E \to \bullet\, E + T],\\
 & & & & [E \to \bullet\, T],\\
 & & & & [T \to \bullet\, T * F],\\
 & & & & [T \to \bullet\, F],\\
 & & & & [F \to \bullet\, (\, E\,)],\\
 & & & & [F \to \bullet\, \mathsf{int}]\}
\end{array}
$$

$$
\begin{array}{lllll}
q_1 & = & \delta(q_0, E) & = & \{[S' \to E\, \bullet],\\
 & & & & [E \to E\, \bullet + T]\}
\end{array}
$$

$$
\begin{array}{lllll}
q_2 & = & \delta(q_0, T) & = & \{[E \to T\, \bullet],\\
 & & & & [T \to T\, \bullet * F]\}
\end{array}
$$

$$
\begin{array}{lllll}
q_3 & = & \delta(q_0, F) & = & \{[T \to F\, \bullet]\}
\end{array}
$$

$$
\begin{array}{lllll}
q_4 & = & \delta(q_0, \mathsf{int}) & & \{[F \to \mathsf{int}\, \bullet]\}
\end{array}
$$

$$
\begin{array}{lllll}
q_5 & = & \delta(q_0, (\,) & = & \{[F \to (\, \bullet\, E\,)],\\
 & & & & [E \to \bullet\, E + T],\\
 & & & & [E \to \bullet\, T],\\
 & & & & [T \to \bullet\, T * F],\\
 & & & & [T \to \bullet\, F],\\
 & & & & [F \to \bullet\, (\, E\,)],\\
 & & & & [F \to \bullet\, \mathsf{int}]\}
\end{array}
$$

Im Beispiel:

$$
\begin{array}{lll}
q_0 & = & \{[S' \to \bullet\, E,\ \{\epsilon\}], \\
 & & \ [E \to \bullet\, E + T,\ \{\epsilon, +\}], \\
 & & \ [E \to \bullet\, T,\ \{\epsilon, +\}], \\
 & & \ [T \to \bullet\, T * F,\ \{\epsilon, +, *\}], \\
 & & \ [T \to \bullet\, F,\ \{\epsilon, +, *\}], \\
 & & \ [F \to \bullet\, (\, E\,),\ \{\epsilon, +, *\}], \\
 & & \ [F \to \bullet\, \mathsf{int}, \{\epsilon, +, *\}]\}
\end{array}
$$

$$
\begin{array}{lllll}
q_1 & = & \delta(q_0, E) & = & \{[S' \to E\, \bullet \quad], \\
 & & & & \ [E \to E \bullet + T \quad]\}
\end{array}
$$

$$
\begin{array}{lllll}
q_2 & = & \delta(q_0, T) & = & \{[E \to T\, \bullet \quad], \\
 & & & & \ [T \to T \bullet * F \quad]\}
\end{array}
$$

$$
\begin{array}{lllll}
q_3 & = & \delta(q_0, F) & = & \{[T \to F\, \bullet \quad]\}
\end{array}
$$

$$
\begin{array}{lllll}
q_4 & = & \delta(q_0, \mathsf{int}) & & \{[F \to \mathsf{int}\, \bullet \quad]\}
\end{array}
$$

$$
\begin{array}{lllll}
q_5 & = & \delta(q_0, (\,)) & = & \{[F \to (\, \bullet\, E\,) \quad], \\
 & & & & \ [E \to \bullet\, E + T \quad], \\
 & & & & \ [E \to \bullet\, T \quad], \\
 & & & & \ [T \to \bullet\, T * F \quad], \\
 & & & & \ [T \to \bullet\, F \quad], \\
 & & & & \ [F \to \bullet\, (\, E\,) \quad], \\
 & & & & \ [F \to \bullet\, \mathsf{int} \quad]\}
\end{array}
$$

Im Beispiel:

$$
\begin{array}{lllll}
q_0 & = & & & \{[S' \to \bullet\, E,\ \{\epsilon\}], \\
& & & & \ [E \to \bullet\, E + T,\ \{\epsilon, +\}], \\
& & & & \ [E \to \bullet\, T,\ \{\epsilon, +\}], \\
& & & & \ [T \to \bullet\, T * F,\ \{\epsilon, +, *\}], \\
& & & & \ [T \to \bullet\, F,\ \{\epsilon, +, *\}], \\
& & & & \ [F \to \bullet\, (\, E\,)\,,\ \{\epsilon, +, *\}], \\
& & & & \ [F \to \bullet\, \mathsf{int}, \{\epsilon, +, *\}]\} \\
& & & & \\
q_1 & = & \delta(q_0, E) & = & \{[S' \to E \bullet,\ \{\epsilon\}], \\
& & & & \ [E \to E \bullet + T,\ \{\epsilon, +\}]\} \\
& & & & \\
q_2 & = & \delta(q_0, T) & = & \{[E \to T \bullet,\ \{\epsilon, +\}], \\
& & & & \ [T \to T \bullet * F,\ \{\epsilon, +, *\}]\}
\end{array}
$$

$$
\begin{array}{lllll}
q_3 & = & \delta(q_0, F) & = & \{[T \to F \bullet,\ \{\epsilon, +, *\}]\} \\
& & & & \\
q_4 & = & \delta(q_0, \mathsf{int}) & & \{[F \to \mathsf{int} \bullet,\ \{\epsilon, +, *\}]\} \\
& & & & \\
q_5 & = & \delta(q_0,\, (\,) & = & \{[F \to (\, \bullet\, E\,) \quad], \\
& & & & \ [E \to \bullet\, E + T \quad], \\
& & & & \ [E \to \bullet\, T \quad], \\
& & & & \ [T \to \bullet\, T * F \quad], \\
& & & & \ [T \to \bullet\, F \quad], \\
& & & & \ [F \to \bullet\, (\, E\,) \quad], \\
& & & & \ [F \to \bullet\, \mathsf{int} \quad]\}
\end{array}
$$

Im Beispiel:

$$
\begin{aligned}
q_0 &= \{[S' \to \bullet\, E, \{\epsilon\}], \\
&\quad [E \to \bullet\, E + T, \{\epsilon, +\}], \\
&\quad [E \to \bullet\, T, \{\epsilon, +\}], \\
&\quad [T \to \bullet\, T * F, \{\epsilon, +, *\}], \\
&\quad [T \to \bullet\, F, \{\epsilon, +, *\}], \\
&\quad [F \to \bullet\, (\, E\,)\,, \{\epsilon, +, *\}], \\
&\quad [F \to \bullet\, \mathsf{int}, \{\epsilon, +, *\}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_1 = \delta(q_0, E) &= \{[S' \to E \bullet, \{\epsilon\}], \\
&\quad [E \to E \bullet + T, \{\epsilon, +\}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_2 = \delta(q_0, T) &= \{[E \to T \bullet, \{\epsilon, +\}], \\
&\quad [T \to T \bullet * F, \{\epsilon, +, *\}]\}
\end{aligned}
$$

$$
q_3 = \delta(q_0, F) = \{[T \to F \bullet, \{\epsilon, +, *\}]\}
$$

$$
q_4 = \delta(q_0, \mathsf{int}) \quad \{[F \to \mathsf{int} \bullet, \{\epsilon, +, *\}]\}
$$

$$
\begin{aligned}
q_5 = \delta(q_0, (\,) &= \{[F \to (\, \bullet\, E\,)\,, \{\epsilon, +, *\}], \\
&\quad [E \to \bullet\, E + T, \{\,)\,, +\}], \\
&\quad [E \to \bullet\, T, \{\,)\,, +\}], \\
&\quad [T \to \bullet\, T * F, \{\,)\,, +, *\}], \\
&\quad [T \to \bullet\, F, \{\,)\,, +, *\}], \\
&\quad [F \to \bullet\, (\, E\,)\,, \{\,)\,, +, *\}], \\
&\quad [F \to \bullet\, \mathsf{int}, \{\,)\,, +, *\}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_5' = \delta(q_5, (\,) = \{ & [F \longrightarrow (\,\bullet E\,)], \\
& [E \longrightarrow \bullet E + T], \\
& [E \longrightarrow \bullet T], \\
& [T \longrightarrow \bullet T * F], \\
& [T \longrightarrow \bullet F], \\
& [F \longrightarrow \bullet (\,E\,)], \\
& [F \longrightarrow \bullet \mathsf{int}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_6 = \delta(q_1, +) = \{ & [E \longrightarrow E + \bullet T], \\
& [T \longrightarrow \bullet T * F], \\
& [T \longrightarrow \bullet F], \\
& [F \longrightarrow \bullet (\,E\,)], \\
& [F \longrightarrow \bullet \mathsf{int}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_7 = \delta(q_2, *) = \{ & [T \longrightarrow T * \bullet F], \\
& [F \longrightarrow \bullet (\,E\,)], \\
& [F \longrightarrow \bullet \mathsf{int}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_8 = \delta(q_5, E) = \{ & [F \longrightarrow (\,E \bullet)]\} \\
& [E \longrightarrow E \bullet + T]\}
\end{aligned}
$$

$$
\begin{aligned}
q_9 = \delta(q_6, T) = \{ & [E \longrightarrow E + T \bullet], \\
& [T \longrightarrow T \bullet * F]\}
\end{aligned}
$$

$$
q_{10} = \delta(q_7, F) = \{[T \longrightarrow T * F \bullet]\}
$$

$$
q_{11} = \delta(q_8, \,)\,) = \{[F \longrightarrow (\,E\,) \bullet]\}
$$

$$
\begin{aligned}
q_5' &= \delta(q_5, (\,) &&= \{[F \to (\,\bullet E\,),\ \{\,)\,,+,*\}],\\
&&& \quad [E \to \bullet E + T,\ \{\,)\,,+\}],\\
&&& \quad [E \to \bullet T,\ \{\,)\,,+\}],\\
&&& \quad [T \to \bullet T * F,\ \{\,)\,,+,*\}],\\
&&& \quad [T \to \bullet F,\ \{\,)\,,+,*\}],\\
&&& \quad [F \to \bullet (\,E\,),\ \{\,)\,,+,*\}],\\
&&& \quad [F \to \bullet \mathsf{int},\ \{\,)\,,+,*\}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_6 &= \delta(q_1, +) &&= \{[E \to E + \bullet T\ ],\\
&&& \quad [T \to \bullet T * F\ ],\\
&&& \quad [T \to \bullet F\ ],\\
&&& \quad [F \to \bullet (\,E\,)\ ],\\
&&& \quad [F \to \bullet \mathsf{int}\ ]\}
\end{aligned}
$$

$$
\begin{aligned}
q_7 &= \delta(q_2, *) &&= \{[T \to T * \bullet F\ ],\\
&&& \quad [F \to \bullet (\,E\,)\ ],\\
&&& \quad [F \to \bullet \mathsf{int}\ ]\}
\end{aligned}
$$

$$
\begin{aligned}
q_8 &= \delta(q_5, E) &&= \{[F \to (\,E \bullet)\ ]\}\\
&&& \quad [E \to E \bullet + T\ ]\}
\end{aligned}
$$

$$
\begin{aligned}
q_9 &= \delta(q_6, T) &&= \{[E \to E + T \bullet\ ],\\
&&& \quad [T \to T \bullet * F\ ]\}
\end{aligned}
$$

$$
q_{10} = \delta(q_7, F) = \{[T \to T * F \bullet\ ]\}
$$

$$
q_{11} = \delta(q_8, \,)\,) = \{[F \to (\,E\,) \bullet\ ]\}
$$

$$
\begin{aligned}
q_5' \;=\; \delta(q_5, () \;=\; \{&[F \to (\bullet E),\ \{),+,* \}],\\
&[E \to \bullet E + T,\ \{),+ \}],\\
&[E \to \bullet T,\ \{),+ \}],\\
&[T \to \bullet T * F,\ \{),+,* \}],\\
&[T \to \bullet F,\ \{),+,* \}],\\
&[F \to \bullet (E),\ \{),+,* \}],\\
&[F \to \bullet \mathsf{int},\ \{),+,* \}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_6 \;=\; \delta(q_1, +) \;=\; \{&[E \to E + \bullet T,\ \{\epsilon,+\}],\\
&[T \to \bullet T * F,\ \{\epsilon,+,*\}],\\
&[T \to \bullet F,\ \{\epsilon,+,*\}],\\
&[F \to \bullet (E),\ \{\epsilon,+,*\}],\\
&[F \to \bullet \mathsf{int},\ \{\epsilon,+,*\}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_7 \;=\; \delta(q_2, *) \;=\; \{&[T \to T * \bullet F\ ],\\
&[F \to \bullet (E)\ ],\\
&[F \to \bullet \mathsf{int}\ ]\}
\end{aligned}
$$

$$
\begin{aligned}
q_8 \;=\; \delta(q_5, E) \;=\; \{&[F \to (E \bullet)\ ]\}\\
&[E \to E \bullet + T\ ]\}
\end{aligned}
$$

$$
\begin{aligned}
q_9 \;=\; \delta(q_6, T) \;=\; \{&[E \to E + T \bullet\ ],\\
&[T \to T \bullet * F\ ]\}
\end{aligned}
$$

$$
q_{10} \;=\; \delta(q_7, F) \;=\; \{[T \to T * F \bullet\ ]\}
$$

$$
q_{11} \;=\; \delta(q_8,)) \;=\; \{[F \to (E) \bullet\ ]\}
$$

$$
\begin{aligned}
q_5' = \delta(q_5, () = \{&[F \longrightarrow (\bullet E), \{), +, *\}],\\
&[E \longrightarrow \bullet E + T, \{), +\}],\\
&[E \longrightarrow \bullet T, \{), +\}],\\
&[T \longrightarrow \bullet T * F, \{), +, *\}],\\
&[T \longrightarrow \bullet F, \{), +, *\}],\\
&[F \longrightarrow \bullet (E), \{), +, *\}],\\
&[F \longrightarrow \bullet \mathsf{int}, \{), +, *\}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_6 = \delta(q_1, +) = \{&[E \longrightarrow E + \bullet T, \{\epsilon, +\}],\\
&[T \longrightarrow \bullet T * F, \{\epsilon, +, *\}],\\
&[T \longrightarrow \bullet F, \{\epsilon, +, *\}],\\
&[F \longrightarrow \bullet (E), \{\epsilon, +, *\}],\\
&[F \longrightarrow \bullet \mathsf{int}, \{\epsilon, +, *\}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_7 = \delta(q_2, *) = \{&[T \longrightarrow T * \bullet F, \{\epsilon, +, *\}],\\
&[F \longrightarrow \bullet (E), \{\epsilon, +, *\}],\\
&[F \longrightarrow \bullet \mathsf{int}, \{\epsilon, +, *\}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_8 = \delta(q_5, E) = \{&[F \longrightarrow (E \bullet), \{\epsilon, +, *\}]\}\\
&[E \longrightarrow E \bullet + T, \{), +\}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_9 = \delta(q_6, T) = \{&[E \longrightarrow E + T\bullet, \{\epsilon, +\}],\\
&[T \longrightarrow T \bullet * F, \{\epsilon, +, *\}]\}
\end{aligned}
$$

$$
q_{10} = \delta(q_7, F) = \{[T \longrightarrow T * F\bullet, \{\epsilon, +, *\}]\}
$$

$$
q_{11} = \delta(q_8,)) = \{[F \longrightarrow (E)\bullet, \{\epsilon, +, *\}]\}
$$

$$
\begin{aligned}
q_2' &= \delta(q_5', T) &&= \{[E \rightarrow T\bullet, \{\,)\,,+\}],\\
&&&\quad [T \rightarrow T \bullet * F, \{\,)\,,+,*\}]\}
\end{aligned}
$$

$$
q_3' = \delta(q_5', F) = \{[F \rightarrow F\bullet, \{\,)\,,+,*\}]\}
$$

$$
q_4' = \delta(q_5', \mathsf{int}) = \{[F \rightarrow \mathsf{int}\bullet, \{\,)\,,+,*\}]\}
$$

$$
\begin{aligned}
q_6' &= \delta(q_8, +) &&= \{[E \rightarrow E + \bullet T, \{\,)\,,+\}],\\
&&&\quad [T \rightarrow \bullet T * F, \{\,)\,,+,*\}],\\
&&&\quad [T \rightarrow \bullet F, \{\,)\,,+,*\}],\\
&&&\quad [F \rightarrow \bullet\,(\,E\,)\,, \{\,)\,,+,*\}],\\
&&&\quad [F \rightarrow \bullet\,\mathsf{int}, \{\,)\,,+,*\}]\}
\end{aligned}
$$

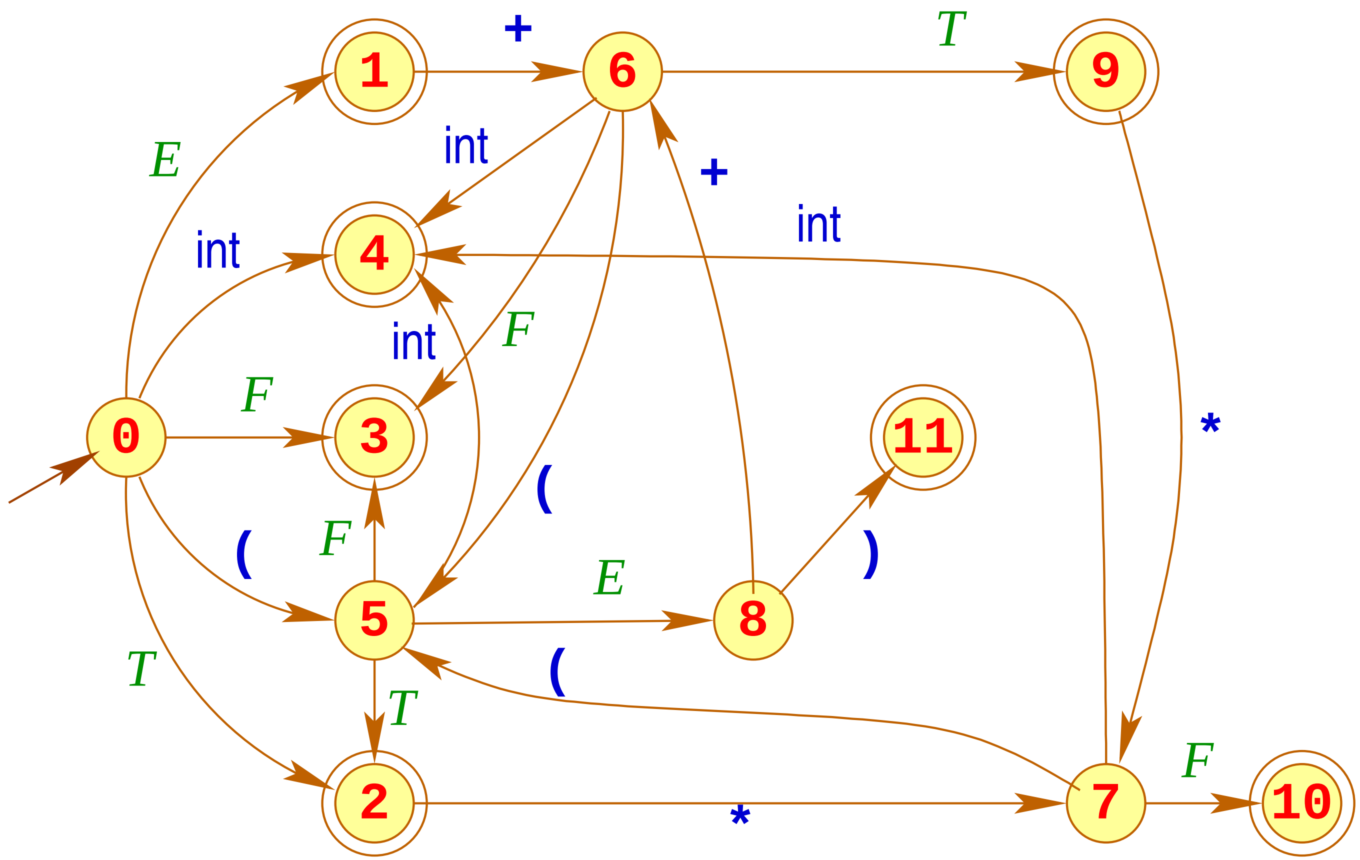
$$
\begin{aligned}
q_7' &= \delta(q_9, *) &&= \{[T \rightarrow T * \bullet F, \{\,)\,,+,*\}],\\
&&&\quad [F \rightarrow \bullet\,(\,E\,)\,, \{\,)\,,+,*\}],\\
&&&\quad [F \rightarrow \bullet\,\mathsf{int}, \{\,)\,,+,*\}]\}
\end{aligned}
$$

$$
\begin{aligned}
q_8' &= \delta(q_5', E) &&= \{[F \rightarrow (\,E \bullet\,)\,, \{\,)\,,+,*\}]\\
&&&\quad [E \rightarrow E \bullet + T, \{\,)\,,+\}]\}
\end{aligned}
$$

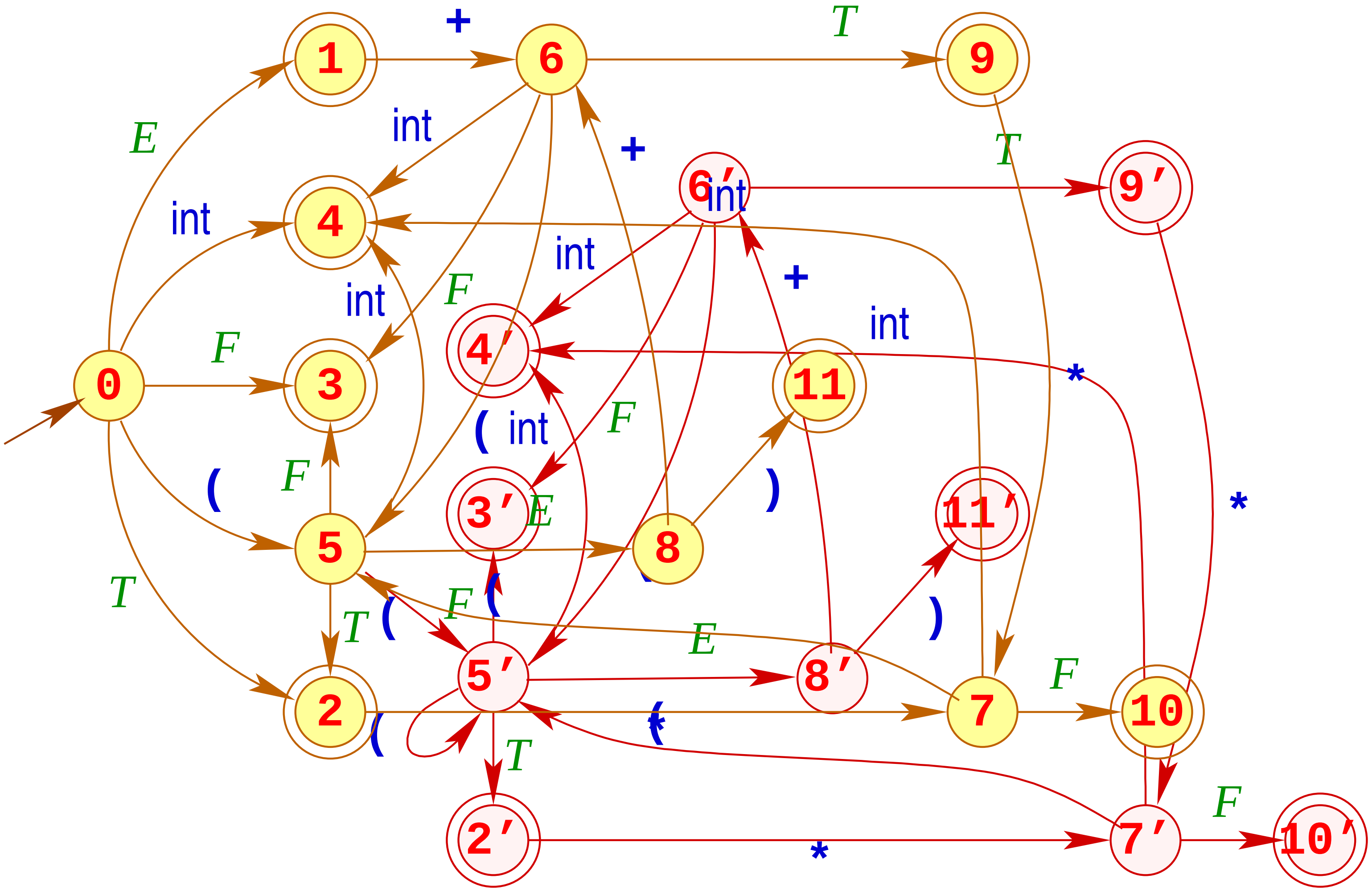
$$
\begin{aligned}
q_9' &= \delta(q_6', T) &&= \{[E \rightarrow E + T\bullet, \{\,)\,,+\}],\\
&&&\quad [T \rightarrow T \bullet * F, \{\,)\,,+,*\}]
\end{aligned}
$$

$$
q_{10}' = \delta(q_7', F) = \{[T \rightarrow T * F\bullet, \{\,)\,,+,*\}]\}
$$

$$
q_{11}' = \delta(q_8', \,)\,) = \{[F \rightarrow (\,E\,)\bullet, \{\,)\,,+,*\}]\}
$$



0
1
2
3
4
5
6
7
8
9
10
11
E
int
F
(
T
+
int
int
F
F
E
T
T
+
int
)
(
(
*
*
F


0
1
2
3
4
5
6
7
8
9
10
11
2'
3'
4'
5'
6'
7'
8'
9'
10'
11'
E
T
F
int
(
)
+
*

Diskussion:

- Im Beispiel hat sich die Anzahl der Zustände fast verdoppelt :-)

  Es kann noch schlimmer kommen :-(

- Die Konflikte in den Zuständen q_1, q_2, q_9 sind nun aufgelöst ...

  Z.B. haben wir für:

$$\begin{aligned} q_9 = \; & \{[E \to E + T\bullet, \{\epsilon, +\}], \\ & [T \to T \bullet * F, \{\epsilon, +, *\}]\} \end{aligned}$$

$$\{\epsilon, +\} \cap (\mathsf{First}_1(* F) \odot \{\epsilon, +, *\}) = \{\epsilon, +\} \cap \{*\} = \emptyset$$

Allgemein: Wir identifizieren zwei Konflikte:

Reduce-Reduce-Konflikt:

$$[A \to \gamma \bullet, x], \quad [A' \to \gamma' \bullet, x] \quad \in \quad q \quad \text{mit} \quad A \neq A' \vee \gamma \neq \gamma'$$

Shift-Reduce-Konflikt:

$$[A \to \gamma \bullet, x], \quad [A' \to \alpha \bullet a\, \beta, y] \quad \in \quad q \quad \text{mit} \quad a \in T \quad \text{und}$$

$$x \in \{a\} \odot \mathsf{First}_k(\beta) \odot \{y\}\,.$$

für einen Zustand $q \in Q$.

Solche Zustände nennen wir jetzt $LR(k)$-ungeeignet :-)

Satz

Eine reduzierte kontextfreie Grammatik G ist genau dann $LR(k)$ wenn der kanonische $LR(k)$-Automat $LR(G, k)$ keine $LR(k)$-ungeeigneten Zustände besitzt.

Satz

Eine reduzierte kontextfreie Grammatik G ist genau dann $LR(k)$ wenn der kanonische $LR(k)$-Automat $LR(G, k)$ keine $LR(k)$-ungeeigneten Zustände besitzt.

Diskussion:

- Unser Beispiel ist offenbar $LR(1)$:-)
- Im Allgemeinen hat der kanonische $LR(k)$-Automat sehr viel mehr Zustände als $LR(G) = LR(G, 0)$:-(
- Man betrachtet darum i.a. Teilklassen von $LR(k)$-Grammatiken, bei denen man nur $LR(G)$ benutzt ...

Satz

Eine reduzierte kontextfreie Grammatik G ist genau dann $LR(k)$ wenn der kanonische $LR(k)$-Automat $LR(G, k)$ keine $LR(k)$-ungeeigneten Zustände besitzt.

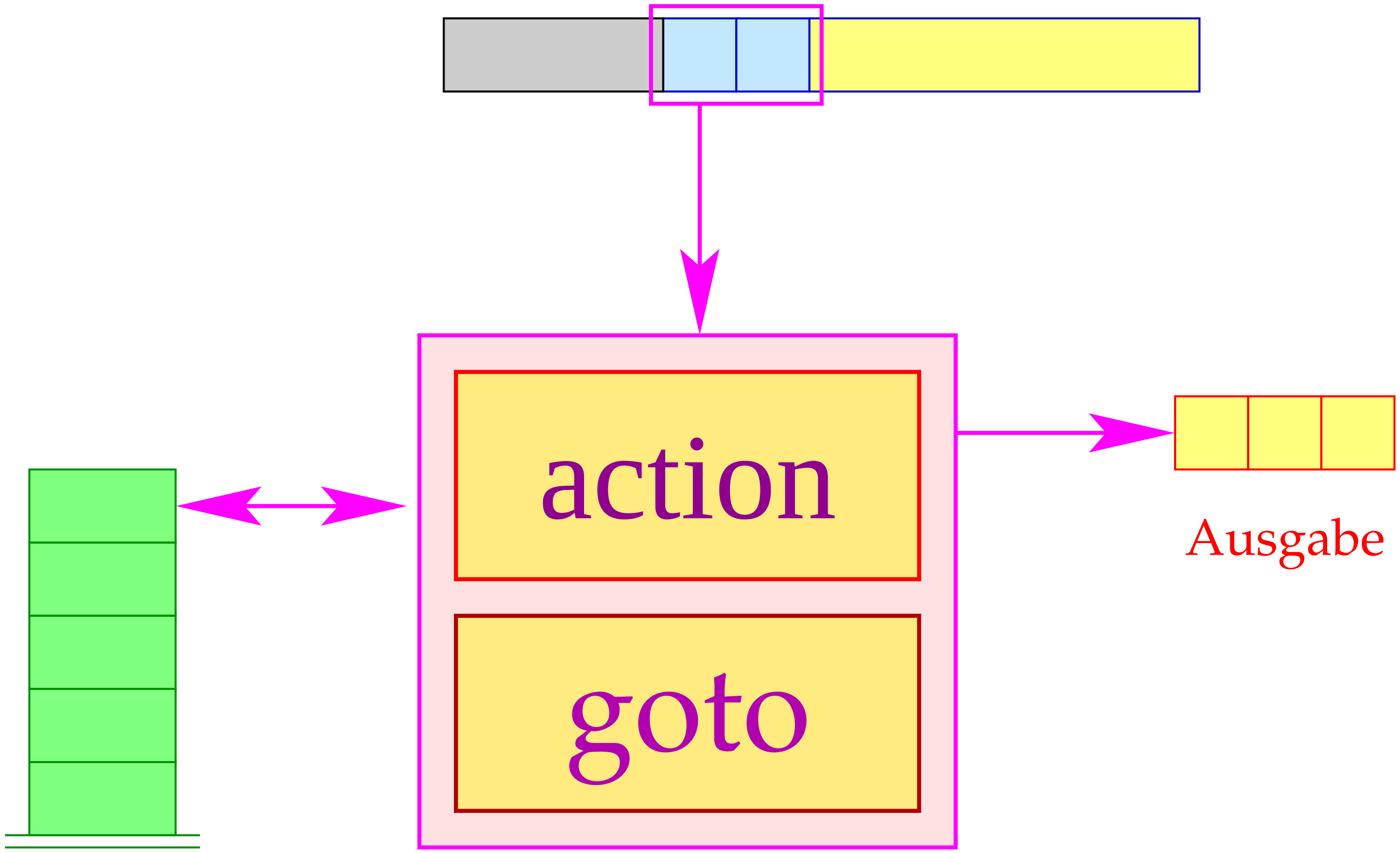
Diskussion:

- Unser Beispiel ist offenbar $LR(1)$:-)
- Im Allgemeinen hat der kanonische $LR(k)$-Automat sehr viel mehr Zustände als $LR(G) = LR(G, 0)$:-(
- Man betrachtet darum i.a. Teilklassen von $LR(k)$-Grammatiken, bei denen man nur $LR(G)$ benutzt ...
- Zur Konflikt-Auflösung ordnet man den Items in den Zuständen Vorausschau-Mengen zu:

  (1) Die Zuordnung ist unabhängig vom Zustand $\Longrightarrow$ Simple $LR(k)$

  (2) Die Zuordnung hängt vom Zustand ab $\Longrightarrow$ $LALR(k)$

Der $LR(k)$-Parser:



Erläuterung:

- Die goto-Tabelle kodiert die Zustandsübergänge:

$$\text{goto}[q, X] = \delta(q, X) \quad \in \quad Q$$

- Die action-Tabelle beschreibt für jeden Zustand q und möglichen Look-ahead w die erforderliche Aktion.

Diese sind:

shift	//	Shift-Operation
reduce $(A \to \gamma)$	//	Reduktion mit Ausgabe
error	//	Fehler

... im Beispiel:

$$
\begin{array}{llllll}
E & \rightarrow & E + T^{\,0} & \mid & T^{\,1} \\
T & \rightarrow & T * F^{\,0} & \mid & F^{\,1} \\
F & \rightarrow & (\ E\ )^{\,0} & \mid & \mathsf{int}^{\,1}
\end{array}
$$

action	ϵ	int	(	)	$+$	$*$
q_1	$S', 0$					s
q_2	$E, 1$					s
q_2'				$E, 1$		s
q_3	$T, 1$				$T, 1$	$T, 1$
q_3'				$T, 1$	$T, 1$	$T, 1$
q_4	$F, 1$				$F, 1$	$F, 1$
q_4'				$F, 1$	$F, 1$	$F, 1$
q_9	$E, 0$				$E, 0$	s
q_9'				$E, 0$	$E, 0$	s
q_{10}	$T, 0$				$T, 0$	$T, 0$
q_{10}'				$T, 0$	$T, 0$	$T, 0$
q_{11}	$F, 0$				$F, 0$	$F, 0$
q_{11}'				$F, 0$	$F, 0$	$F, 0$

2.7 Spezielle Bottom-up-Verfahren mit $LR(G)$

Idee 1: Benutze Follow_k-Mengen zur Konflikt-Lösung ...

Reduce-Reduce-Konflikt:

Falls für $[A \to \gamma \bullet], \; [A' \to \gamma' \bullet] \in q$ mit $A \neq A' \vee \gamma \neq \gamma'$,

$$\text{Follow}_k(A) \cap \text{Follow}_k(A') \neq \emptyset$$

Shift-Reduce-Konflikt:

Falls für $[A \to \gamma \bullet], \; [A' \to \alpha \bullet a\, \beta] \in q$ mit $a \in T$,

$$\text{Follow}_k(A) \cap (\{a\} \odot \text{First}_k(\beta) \odot \text{Follow}_k(A')) \neq \emptyset$$

für einen Zustand $q \in Q$.

Dann nennen wir den Zustand q $SLR(k)$-ungeeignet :-)

Die reduzierte Grammatik G nennen wir $SLR(k)$ (simple $LR(k)$:-), falls der kanonische $LR(0)$-Automat $LR(G)$ keine $SLR(k)$-ungeeigneten Zustände enthält :-)

Die reduzierte Grammatik G nennen wir $SLR(k)$ (simple $LR(k)$:-), falls der kanonische $LR(0)$-Automat $LR(G)$ keine $SLR(k)$-ungeeigneten Zustände enthält :-)

... im Beispiel:

Bei unserer Beispiel-Grammatik treten Konflikte möglicherweise in den Zuständen q_1, q_2, q_9 auf:

$$
\begin{array}{lll}
q_1 & = & \{[S' \to E\bullet], \\
& & \;[E \to E \bullet + T]\}
\end{array}
\qquad
\begin{array}{lll}
\mathsf{Follow}_1(S') \cap \{+\} \odot \{\ldots\} & = & \{\epsilon\} \cap \{+\} \\
& = & \emptyset
\end{array}
$$

Die reduzierte Grammatik G nennen wir $SLR(k)$ (simple $LR(k)$:-), falls der kanonische $LR(0)$-Automat $LR(G)$ keine $SLR(k)$-ungeeigneten Zustände enthält :-)

... im Beispiel:

Bei unserer Beispiel-Grammatik treten Konflikte möglicherweise in den Zuständen q_1, q_2, q_9 auf:

$$\begin{array}{lll} q_1 & = & \{[S' \to E\bullet], \\ & & \;[E \to E \bullet + T]\} \end{array} \qquad \begin{array}{lll} \text{Follow}_1(S') \cap \{+\} \odot \{\dots\} & = & \{\epsilon\} \cap \{+\} \\ & = & \emptyset \end{array}$$

$$\begin{array}{lll} q_2 & = & \{[E \to T\bullet], \\ & & \;[T \to T \bullet * F]\} \end{array} \qquad \begin{array}{lll} \text{Follow}_1(E) \cap \{*\} \odot \{\dots\} & = & \{\epsilon, +,)\} \cap \{*\} \\ & = & \emptyset \end{array}$$

$$\begin{array}{lll} q_9 & = & \{[E \to E + T\bullet], \\ & & \;[T \to T \bullet * F]\} \end{array} \qquad \begin{array}{lll} \text{Follow}_1(E) \cap \{*\} \odot \{\dots\} & = & \{\epsilon, +,)\} \cap \{*\} \\ & = & \emptyset \end{array}$$

Idee 2: Berechne für jeden Zustand q Follow-Mengen :-)

Für $[A \rightarrow \alpha \bullet \beta] \in q$ definieren wir:

$$\begin{array}{lll} \Lambda_k(q, [A \rightarrow \alpha \bullet \beta]) & = & \{\mathsf{First}_k(w) \mid S' \rightarrow_R^* \gamma\, A\, w \;\wedge \\ & & \qquad\qquad \delta(q_0, \gamma\,\alpha) = q\} \\ & // & \subseteq \;\; \mathsf{Follow}_k(A) \end{array}$$

Idee 2: Berechne für jeden Zustand q Follow-Mengen :-)

Für $[A \to \alpha \bullet \beta] \in q$ definieren wir:

$$\begin{array}{lll} \Lambda_k(q, [A \to \alpha \bullet \beta]) & = & \{\mathsf{First}_k(w) \mid S' \to_R^* \gamma\, A\, w \;\wedge \\ & & \qquad \delta(q_0, \gamma\, \alpha) = q\} \\ & // \subseteq & \mathsf{Follow}_k(A) \end{array}$$

Reduce-Reduce-Konflikt:

$[A \to \gamma \bullet]\,,\; [A' \to \gamma' \bullet] \in q$ mit $A \neq A' \vee \gamma \neq \gamma'$ wobei:

$$\Lambda_k(q, [A \to \gamma \bullet]) \cap \Lambda_k(q, [A' \to \gamma' \bullet]) \neq \emptyset$$

Idee 2: Berechne für jeden Zustand q Follow-Mengen :-)

Für $[A \to \alpha \bullet \beta] \in q$ definieren wir:

$$\begin{array}{lll} \Lambda_k(q, [A \to \alpha \bullet \beta]) & = & \{\mathsf{First}_k(w) \mid S' \to_R^* \gamma\, A\, w \;\wedge \\ & & \qquad\qquad \delta(q_0, \gamma\, \alpha) = q\} \\ & // & \subseteq\; \mathsf{Follow}_k(A) \end{array}$$

Reduce-Reduce-Konflikt:

$[A \to \gamma \bullet],\; [A' \to \gamma' \bullet] \in q$ mit $A \neq A' \vee \gamma \neq \gamma'$ wobei:

$$\Lambda_k(q, [A \to \gamma \bullet]) \cap \Lambda_k(q, [A' \to \gamma' \bullet]) \neq \emptyset$$

Shift-Reduce-Konflikt:

$[A \to \gamma \bullet],\; [A' \to \alpha \bullet a\, \beta] \in q$ mit $a \in T$ wobei:

$$\Lambda_k(q, [A \to \gamma \bullet]) \cap (\{a\} \odot \mathsf{First}_k(\beta) \odot \Lambda_k(q, [A' \to \alpha \bullet a\, \beta])) \neq \emptyset$$

Solche Zustände nennen wir jetzt $LALR(k)$-ungeeignet :-)

Die reduzierte Grammatik G nennen wir $LALR(k)$, falls der kanonische $LR(0)$-Automat $LR(G)$ keine $LALR(k)$-ungeeigneten Zustände enthält :-)

Bevor wir Beispiele betrachten, überlegen wir erst, wie die Mengen $\Lambda_k(q, [A \to \alpha \bullet \beta])$ berechnet werden können :-)

Die reduzierte Grammatik G nennen wir $LALR(k)$, falls der kanonische $LR(0)$-Automat $LR(G)$ keine $LALR(k)$-ungeeigneten Zustände enthält :-)

Bevor wir Beispiele betrachten, überlegen wir erst, wie die Mengen $\Lambda_k(q, [A \to \alpha \bullet \beta])$ berechnet werden können :-)

Idee: Stelle ein Ungleichungssystem auf !!!

$$
\begin{array}{lll}
\Lambda_k(q_0, [S' \to \bullet S]) & \supseteq & \{\epsilon\} \\
\Lambda_k(q, [A \to \alpha X \bullet \beta]) & \supseteq & \Lambda_k(p, [A \to \alpha \bullet X \beta]) \qquad \text{falls} \quad \delta(p, X) = q \\
\Lambda_k(q, [A \to \bullet \gamma]) & \supseteq & \mathsf{First}_k(\beta) \odot \Lambda_k(q, [B \to \alpha \bullet A \beta]) \qquad \text{falls} \quad [B \to \alpha \bullet A \beta] \in q
\end{array}
$$

Beispiel:

$$\begin{array}{lll} S & \rightarrow & A\,b\,B \mid B \\ A & \rightarrow & a \mid b\,B \\ B & \rightarrow & A \end{array}$$

Der kanonische $LR(0)$-Automat hat dann die folgenden Zustände:

$$\begin{array}{lll} q_0 & = & \{[S' \rightarrow \bullet\, S], \\ & & [S \rightarrow \bullet\, A\,b\,B], \\ & & [A \rightarrow \bullet\, a], \\ & & [A \rightarrow \bullet\, b\,B], \\ & & [S \rightarrow \bullet\, B], \\ & & [B \rightarrow \bullet\, A]\} \end{array}$$

$$q_1 = \delta(q_0, S) = \{[S' \rightarrow S \bullet]\}$$

$$q_2 = \delta(q_0, a) = \{[A \rightarrow a \bullet]\}$$

$$\begin{array}{lllll} q_3 & = & \delta(q_0, b) & = & \{[A \rightarrow b \bullet B], \\ & & & & [B \rightarrow \bullet\, A], \\ & & & & [A \rightarrow \bullet\, a], \\ & & & & [A \rightarrow \bullet\, b\,B]\} \end{array}$$

$$q_4 = \delta(q_0, B) = \{[S \rightarrow B \bullet]\}$$

$$
\begin{array}{lllllll}
q_5 & = & \delta(q_0, A) & = & \{[S \to A \bullet b\, B], \\
 & & & & [B \to A \bullet]\}
\end{array}
$$

$$
q_6 = \delta(q_3, A) = \{[B \to A \bullet]\}
$$

$$
q_7 = \delta(q_3, B) = \{[A \to b\, B \bullet]\}
$$

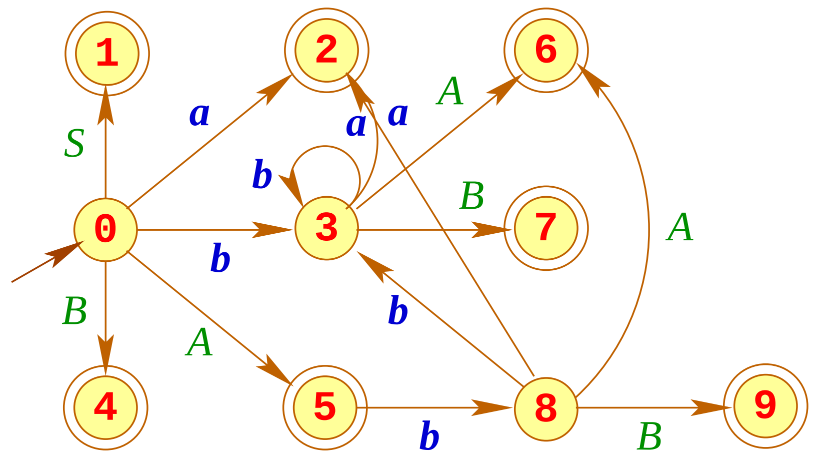
$$
\begin{array}{lllll}
q_8 & = & \delta(q_5, b) & = & \{[S \to A\, b \bullet B], \\
 & & & & [B \to \bullet A], \\
 & & & & [A \to \bullet a], \\
 & & & & [A \to \bullet b\, B]\}
\end{array}
$$

$$
q_9 = \delta(q_8, B) = \{[S \to A\, b\, B \bullet]\}
$$

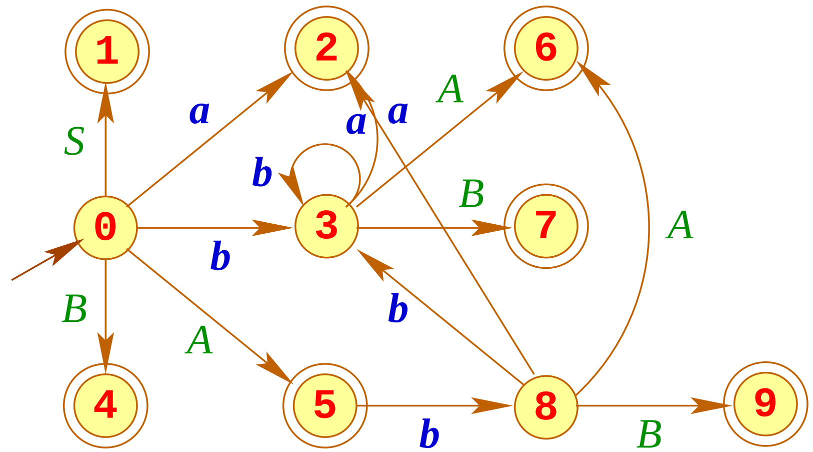
Shift-Reduce-Konflikt:

$$
\begin{array}{lll}
q_5 & = & \{[S \to A \bullet b\, B], \\
 & & [B \to A \bullet]\}
\end{array}
$$

Dabei ist: $\mathsf{Follow}_1(B) \cap \{b\} \odot \{\dots\} = \{\epsilon, b\} \cap \{b\} \neq \emptyset$



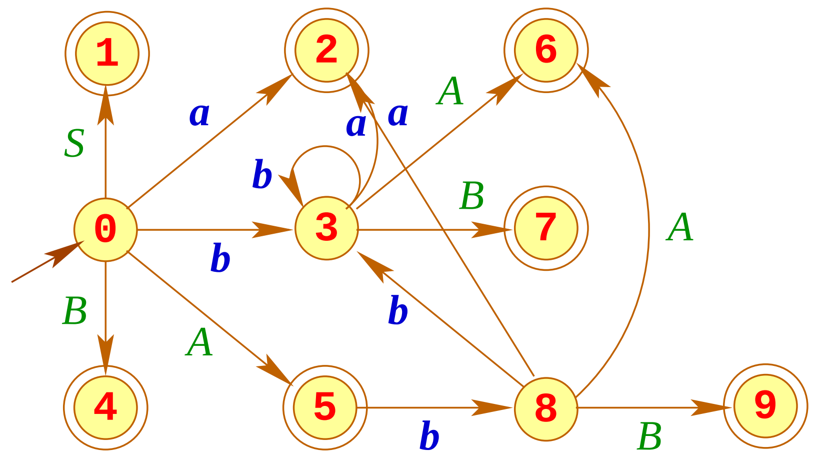
0
1
2
3
4
5
6
7
8
9
S
a
b
B
A
a
a
A
b
B
b
A
b
B



Ausschnitt des Ungleichungssystems:

$$\Lambda_1(q_5, [B \to A\bullet]) \supseteq \Lambda_1(q_0, [B \to \bullet\, A])$$

$$\begin{aligned}
\Lambda_1(q_0, [B \to \bullet\, A]) &\supseteq \Lambda_1(q_0, [S \to \bullet\, B]) \\
\Lambda_1(q_0, [S \to \bullet\, B]) &\supseteq \Lambda_1(q_0, [S' \to \bullet\, S]) \\
\Lambda_1(q_0, [S' \to \bullet\, S]) &\supseteq \{\epsilon\}
\end{aligned}$$



Ausschnitt des Ungleichungssystems:

$$\Lambda_1(q_1, [B \to A\bullet]) \;\supseteq\; \Lambda_1(q_0, [B \to \bullet\, A])$$

$$\begin{array}{lll} \Lambda_1(q_0, [B \to \bullet\, A]) & \supseteq & \Lambda_1(q_0, [S \to \bullet\, B]) \\ \Lambda_1(q_0, [S \to \bullet\, B]) & \supseteq & \Lambda_1(q_0, [S' \to \bullet\, S]) \\ \Lambda_1(q_0, [S' \to \bullet\, S]) & \supseteq & \{\epsilon\} \end{array}$$

Folglich: $\Lambda_1(q_5, [B \to A\bullet]) = \{\epsilon\}$

Diskussion:

- Das Beispiel ist folglich nicht $SLR(1)$, aber $LALR(1)$:-)
- Das Beispiel ist nicht so an den Haaren herbei gezogen, wie es scheint ...
- Umbenennung: $A \Rightarrow L$ $B \Rightarrow R$ $a \Rightarrow \mathsf{id}$ $b \Rightarrow * \,/ =$ liefert:

$$\begin{array}{lll} S & \rightarrow & L = R \mid R \\ L & \rightarrow & \mathsf{id} \mid *\, R \\ R & \rightarrow & L \end{array}$$

... d.h. ein Fragment der Grammatik für C-Ausdrücke ;-)

Für $k = 1$ lassen sich die Mengen $\Lambda_k(q, [A \to \alpha \bullet \beta])$ wieder effizient berechnen :-)

Das verbesserte Ungleichungsssystem:

$$
\begin{array}{lllll}
\Lambda_1(q_0, [S' \to \bullet\, S]) & \supseteq & \{\epsilon\} & & \\
\Lambda_1(q, [A \to \alpha\, X \bullet \beta]) & \supseteq & \Lambda_1(p, [A \to \alpha \bullet X\, \beta]) & \text{falls} & \delta(p, X) = q \\
\Lambda_1(q, [A \to \bullet\, \gamma]) & \supseteq & F_\epsilon(X_j) & \text{falls} & [B \to \alpha \bullet A\, X_1 \ldots X_m] \in q \\
 & & & \text{und} & \text{empty}(X_1) \wedge \ldots \wedge \text{empty}(X_{j-1}) \\
\Lambda_1(q, [A \to \bullet\, \gamma]) & \supseteq & \Lambda_1(q, [B \to \alpha \bullet A\, X_1 \ldots X_m]) & \text{falls} & [B \to \alpha \bullet A\, X_1 \ldots X_m] \in q \\
 & & & \text{und} & \text{empty}(X_1) \wedge \ldots \wedge \text{empty}(X_m)
\end{array}
$$

$\Longrightarrow$ ein reines Vereinigungsproblem :-))

3 Semantische Analyse

- Lexikalisch und syntaktisch korrekte Programme können trotzdem fehlerhaft sein ;-(
- Einige von diesen Fehlern werden bereits durch die Sprachdefinition ausgeschlossen und müssen vom Compiler überprüft werden :-)
- Weitere Analysen sind erforderlich, um:

  $\longrightarrow$ Bezeichner eindeutig zu machen;

  $\longrightarrow$ die Typen von Variablen zu ermitteln;

  $\longrightarrow$ Möglichkeiten zur Programm-Optimierung zu finden.

3.1 Symbol-Tabellen

Beispiel:

```
void foo() {
    int A;
    void fee() {
        double A;
        A = 0.5;
        write(A);
    }
    A = 2;
    fee();
    write(A);
}
```

Diskussion:

- Innerhalb des Rumpfs von fee wird die Definition von A durch die lokale Definition verdeckt :-)
- Für die Code-Erzeugung benötigen wir für jede Benutzung eines Bezeichners die zugehörige Definitionsstelle.
- Statische Bindung bedeutet, dass dass die Definition eines Namens A an allen Programmpunkten innerhalb ihres gesamten Blocks gültig ist.
- Sichtbar ist sie aber nur außerhalb derjenigen Teilbereiche, in an denen eine weitere Definition von A gültig ist :-)

... im Beispiel:

```
void foo() {
    int A;
    void fee() {
        double A;
        A = 0.5;
        write(A);
    }
    A = 2;
    fee();
    write(A);
}
```

Kompliziertere Regeln der Sichtbarkeit gibt es in objektorientierten Programmiersprachen wie Java ...

Beispiel:

```
public class Foo {
    protected int x = 17;
    protected int y = 5;
    private int z = 42;
    public int b() { return 1; }
  }
class Fee extends Foo {
    protected double y = .5;
    public int b(int a) { return a; }
  }
```

Diskussion:

- **private** Members sind nur innerhalb der aktuellen Klasse gültig :-)
- **protected** Members sind innerhalb der Klasse, in den Unterklassen sowie innerhalb des gesamten package gültig :-)
- Methoden b gleichen Namens sind stets verschieden, wenn ihre Argument-Typen verschieden sind !!!
- Bei Aufrufen einer Methode wird dynamisch entschieden, welche Definition gemeint ist ...

Beispiel:

```
public class Foo {
        protected int foo() { return 1; }
    }
class Fee extends Foo {
        protected int foo() { return 2; }
        public int test(boolean b) {
                Foo x = (b) ? new Foo() : new Fee();
                return x.foo();
            }
    }
```

Aufgabe: Finde zu jeder Benutzung eines Bezeichners die zugehörige Definition

1. Schritt: Ersetze Bezeichner durch eindeutige Nummern !

Input: Folge von Strings

Output:
(1) Folge von Nummern
(2) Tabelle, die zu Nummern die Strings auflistet

Beispiel:

das	schwein	ist	dem	schwein	was	...

...	das	schwein	dem	menschen	ist	wurst

... liefert:

0	1	2	3	1	4	0	1	3	5	2	6

0	das
1	schwein
2	ist
3	dem
4	was
5	menschen
6	wurst

Implementierung 1:

Wir benutzen eine partielle Abbildung: $S : \textbf{String} \to \textbf{int}$ verwaltet :-)

Wir verwaltet einen Zähler $\textbf{int}\ \mathsf{count} = 0;$ für die Anzahl der bereits gefundenen Wörter :-)

Damit definieren wir eine Funktion: $\textbf{int}\ \mathsf{getIndex}(\textbf{String}\ w):$

$$
\begin{array}{l}
\textbf{int}\ \mathsf{getIndex}(\textbf{String}\ w)\ \{ \\
\quad \textbf{if}\ (S\,(w)\ \equiv\ \mathsf{undefined})\ \{ \\
\quad\quad S = S \oplus \{w \mapsto \mathsf{count}\}; \\
\quad\quad \textbf{return}\ \mathsf{count}{+}{+}; \\
\quad \textbf{else}\ \textbf{return}\ S\,(w); \\
\}
\end{array}
$$

Implementierung 2: Partielle Abbildungen

Ideen:

- Liste von Paaren $(w, i) \in$ **String** $\times$ **int** :

  Einfügen: $\mathcal{O}(1)$

  Finden: $\mathcal{O}(n)$ $\Longrightarrow$ zu teuer :-(

- balancierte Bäume :

  Einfügen: $\mathcal{O}(\log(n))$

  Finden: $\mathcal{O}(\log(n))$ $\Longrightarrow$ zu teuer :-(

- Hash Tables :

  Einfügen: $\mathcal{O}(1)$

  Finden: $\mathcal{O}(1)$ ... zumindest im Mittel :-)

... im Beispiel:

- Wir legen ein Feld M von hinreichender Größe m an :-)
- Wir wählen eine Hash-Funktion $H : \textbf{String} \rightarrow [0, m-1]$ mit den Eigenschaften:

  $\rightarrow$ $H(w)$ ist leicht zu berechnen :-)

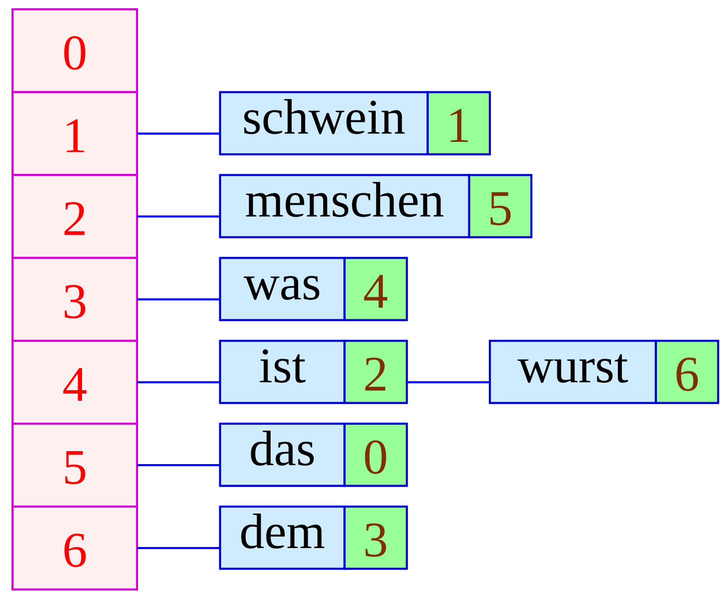
  $\rightarrow$ H streut die vorkommenden Wörter gleichmäßig über $[0, m-1]$:-)

Mögliche Wahlen:

$$
\begin{array}{lcl}
H_0(x_0 \ldots x_{r-1}) & = & (x_0 + x_{r-1}) \,\%\, m \\
H_1(x_0 \ldots x_{r-1}) & = & (\sum_{i=0}^{r-1} x_i \cdot p^i) \,\%\, m \\
 & = & (x_0 + p \cdot (x_1 + p \cdot (\ldots + p \cdot x_{r-1} \cdots))) \,\%\, m \\
 & & \text{für eine Primzahl } p \quad \text{(z.B. } 31 \text{ :-)}
\end{array}
$$

- Das Argument-Wert-Paar (w, i) legen wir dann in $M[H(w)]$ ab :-)

Mit $m = 7$ und H_0 erhalten wir:



Um den Wert des Worts w zu finden, müssen wir w mit allen Worten x vergleichen, für die $H(w) = H(x)$:-)

2. Schritt: Symboltabellen

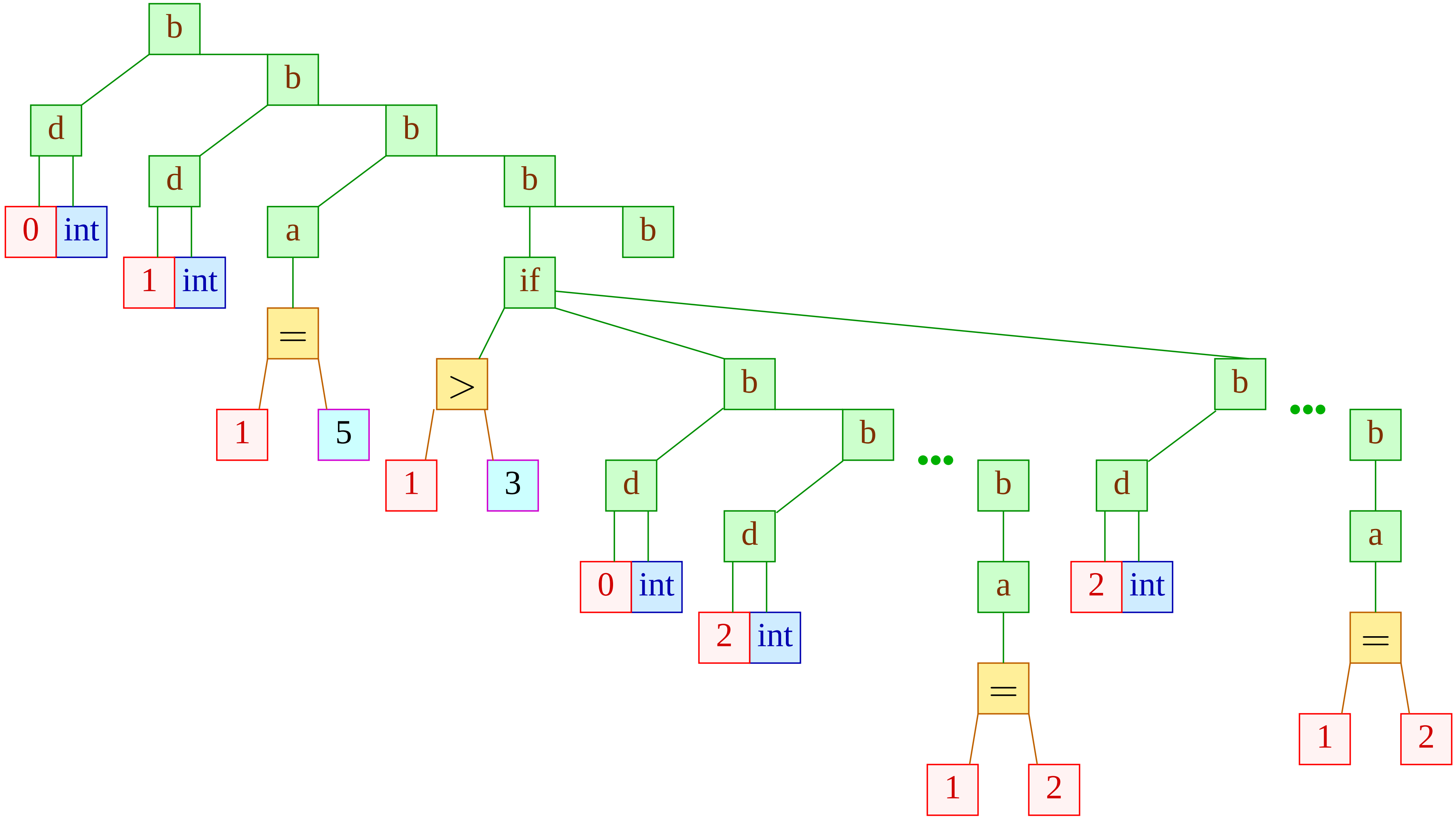
- Durchmustere den Syntaxbaum in einer geeigneten Reihenfolge, die
  - $\longrightarrow$ jede Definition vor ihren Benutzungen besucht :-)
  - $\longrightarrow$ die jeweils aktuell sichtbare Definition zuletzt besucht :-)
- Für jeden Bezeichner verwaltet man einen Keller der gültigen Definitionen.
- Trifft man bei der Durchmusterung auf eine Definition eines Bezeichners, schiebt man sie auf den Keller.
- Verlässt man den Gültigkeitsbereich, muss man sie wieder vom Keller werfen :-)
- Trifft man bei der Durchmusterung auf eine Benutzung, schlägt man die letzte Definition auf dem Keller nach ...
- Findet man keine Definition, haben wir einen Fehler gefunden :-)

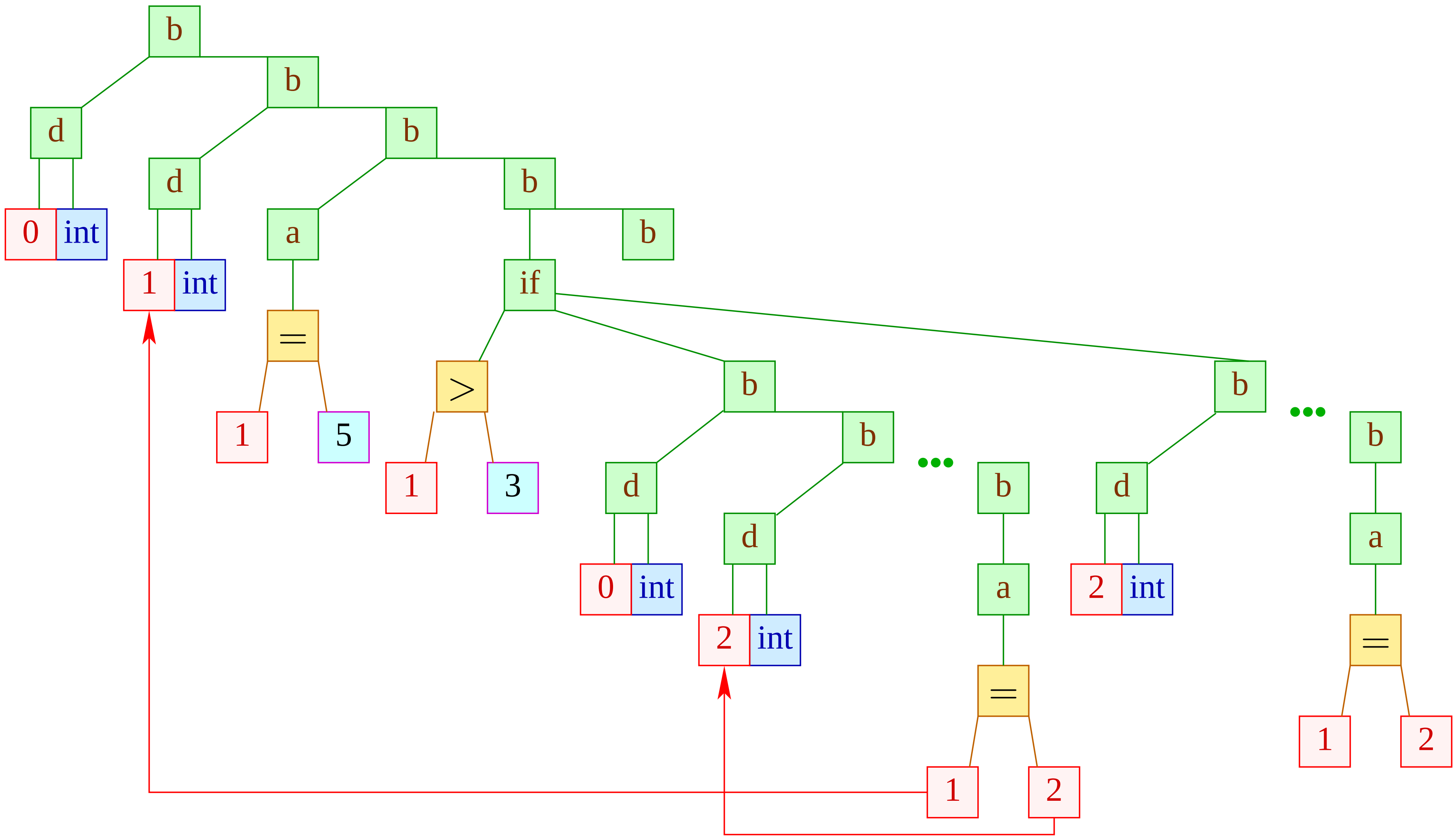
Beispiel:

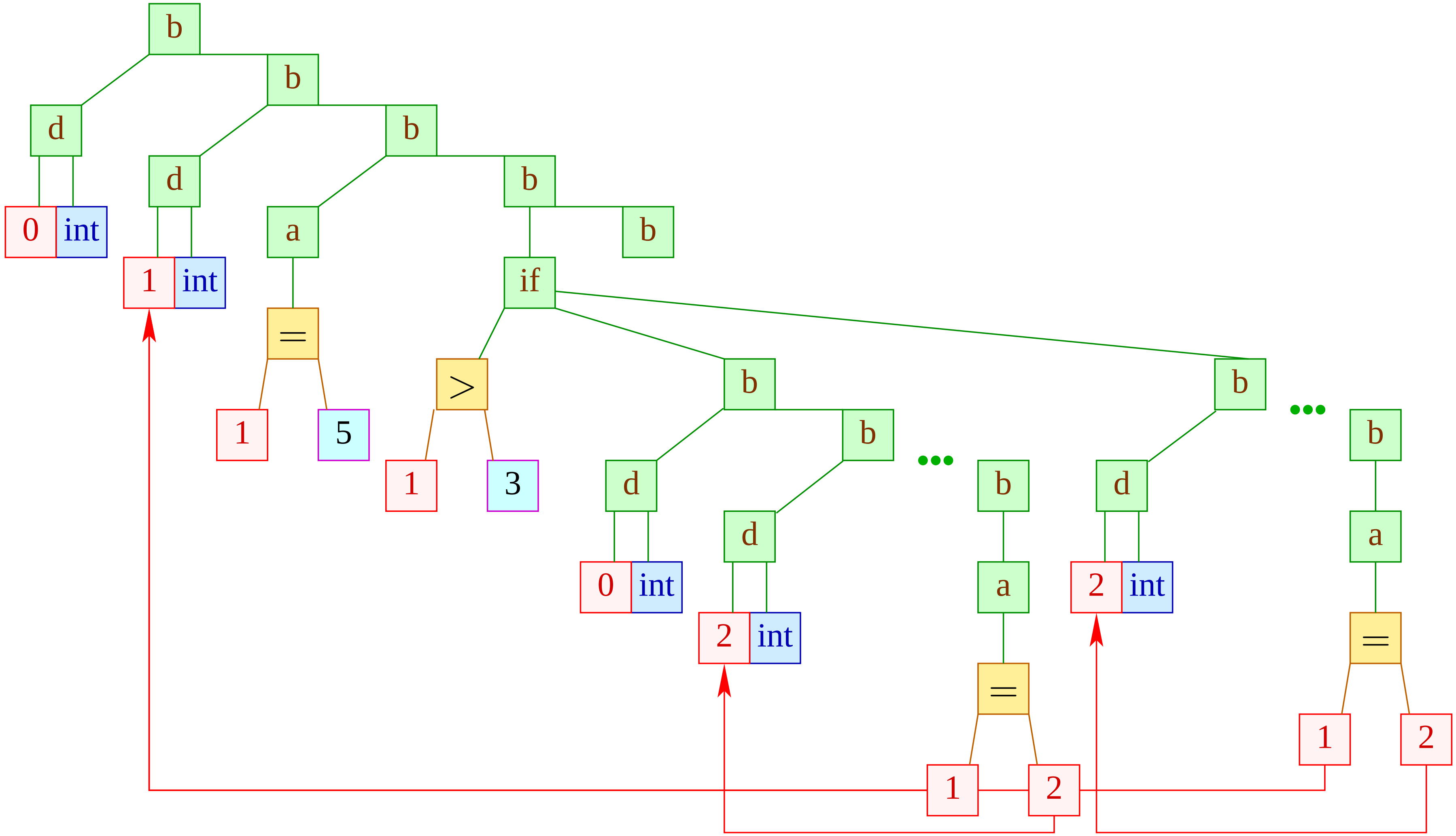
$$
\begin{array}{ll}
\{\ \textbf{int}\ a, b; & \}\ \textbf{else}\ \{ \\
\quad a = 5; & \qquad \textbf{int}\ c; \\
\quad \textbf{if}\ (a > 3)\ \{ & \qquad c = a + 1; \\
\qquad \textbf{int}\ a, c; & \qquad b = c; \\
\qquad a = 3; & \quad \} \\
\qquad c = a + 1; & \quad b = a + b; \\
\qquad b = c; & \}
\end{array}
$$

0	a
1	b
2	c

Der zugehörige Syntaxbaum ...


b
b
d
b
d
0
int
a
b
1
int
b
if
=
1
5
>
1
3
b
d
b
0
int
d
2
int
b
a
=
1
2
b
d
2
int
b
a
=
1
2


b
b
d
b
0
int
d
b
1
int
a
b
if
b
=
1
5
>
1
3
b
b
d
b
0
int
d
2
int
b
a
=
1
2
b
d
2
int
b
a
=
1
2


b
b
d
b
0
int
d
a
b
b
1
int
if
=
b
b
>
1
5
b
d
b
1
3
d
b
a
d
int
0
int
2
int
=
2
=
1
2
1
2

Diskussion:

- Der Durchlauf ist hier einfach links-rechts DFS.
- Benutzt man eine Listen-Implementierung der Keller und eine rekursive Implementierung, kann man auf das Beseitigen der jeweils neuen Definitionen verzichten :-)
- Anstelle erst die Namen durch Nummern zu ersetzen und dann die Zuordnung von Benutzungen zu Definitionen vorzunehmen, kann man auch gleich eindeutige Nummern vergeben :-))

Diskussion:

- Der Durchlauf ist hier einfach links-rechts DFS.
- Benutzt man eine Listen-Implementierung der Keller und eine rekursive Implementierung, kann man auf das Beseitigen der jeweils neuen Definitionen verzichten :-)
- Anstelle erst die Namen durch Nummern zu ersetzen und dann die Zuordnung von Benutzungen zu Definitionen vorzunehmen, kann man auch gleich eindeutige Nummern vergeben :-))

Achtung:

- Manche Programmiersprachen verbieten eine Mehrfach-Deklaration des selben Namens innerhalb eines Blocks ;-)
- Dann muss man für jede Deklaration einen Pointer auf den Block verwalten, zu dem sie gehört.
- Gibt es eine weitere Deklaration des gleichen Namens mit dem selben Pointer, muss ein Fehler gemeldet werden :-))

Erweiterung:

- Hat man mehrere wechselseitig rekursive Funktionsdefinitionen in einem Block, müssen deren Namen vor Durchmustern der Rümpfe in die Tabelle eingetragen werden ...

$$
\begin{array}{llcl}
\textbf{fun} & \mathsf{odd}\,0 & = & \textbf{false} \\
| & \mathsf{odd}\,1 & = & \textbf{true} \\
| & \mathsf{odd}\,x & = & \mathsf{even}\,(x-1) \\
\textbf{and} & \mathsf{even}\,0 & = & \textbf{true} \\
| & \mathsf{even}\,1 & = & \textbf{false} \\
| & \mathsf{even}\,x & = & \mathsf{odd}\,(x-1)
\end{array}
$$

- Hat man eine objektorientierte Sprache mit Vererbung zwischen Klassen, sollte die übergeordnete Klasse vor der Unterklasse besucht werden :-)
- Bei Überladung muss simultan eine Typüberprüfung vorgenommen werden ...

3.2 Typ-Überprüfung

In modernen (imperativen / objektorientierten / funktionalen) Programmiersprachen besitzen Variablen und Funktionen einen Typ, z.B. **int**, **struct** { **int** x; **int** y; }.

Typen sind nützlich für:

- die Speicherverwaltung;
- die Vermeidung von Laufzeit-Fehlern :-)

In imperativen /objektorientierten Programmiersprachen muss der Typ bei der Deklaration spezifiziert und vom Compiler die typ-korrekte Verwendung überprüft werden :-)

Typen werden durch Typ-Ausdrücke beschrieben.

Die Menge T der Typausdrücke enthält:

(1) Basis-Typen: **int**, **boolean**, **float**, **void** ...

(2) Typkonstruktoren, angewendet auf Typen, z.B.:

- Verbunde: **struct** $\{ t_1\ a_1; \ldots t_k\ a_k; \}$
- Zeiger: $t\ *$
- Felder: t []

  Achtung:
  In C muss zusätzlich eine Größe spezifiziert werden; die Variable muss dann zwischen t und $[n]$ stehen :-(

- Funktionen: $t\ (t_1, \ldots, t_k)$

  Achtung:
  In C muss die Variable zwischen t und $(t_1, \ldots, t_k)$ stehen.

  In SML dagegen würde man diesen Typ anders herum schreiben:
  $t_1 * \ldots * t_k \to t$:-)

  Wir benutzen: $(t_1, \ldots, t_k)$ als Tupel-Typen.

(3) Typ-Namen.

Typ-Namen sind nützlich:

- als Abkürzung :-)

  In C kann man diese mithilfe von **typedef** einführen:

  $$\textbf{typedef}\ t\ \ x\ ;$$

- zur Konstruktion rekursiver Typen ...

Beispiel:

```
struct list0 {
            int info;
            struct list1 * next;
      };
```

```
struct list1 {
            int info;
            struct list0 * next;
      };
```

Aufgabe:

Gegeben: eine Menge von Typ-Deklarationen $\Gamma = \{t_1\ x_1; \ldots t_m\ x_m;\}$

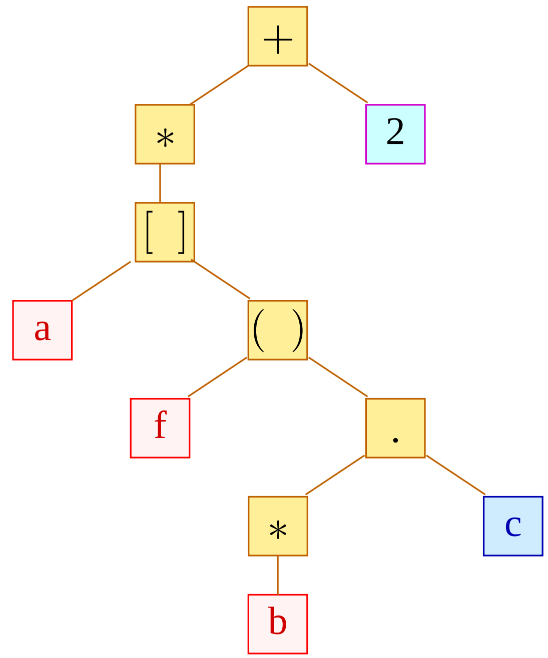
Überprüfe: Kann ein Ausdruck e mit dem Typ t versehen werden?

Beispiel:

$$
\begin{array}{l}
\textbf{struct}\ \mathsf{list}\ \{\textbf{int}\ \mathsf{info}; \textbf{struct}\ \mathsf{list}\ *\mathsf{next};\}; \\
\textbf{int}\ \mathsf{f}(\textbf{struct}\ \mathsf{list} * \mathsf{l})\ \{\textbf{return}\ 1;\}; \\
\textbf{struct}\ \{\textbf{struct}\ \mathsf{list} * \mathsf{c};\} * \mathsf{b}; \\
\textbf{int} * \mathsf{a}[11];
\end{array}
$$

Betrachte den Ausdruck:

$$*\mathsf{a}[f(\mathsf{b} \rightarrow \mathsf{c})] + 2;$$



Idee:

- Traversiere den Syntaxbaum bottom-up.
- Für Bezeichner sagt uns Γ den richtigen Typ :-)
- Konstanten wie 2 oder 0.5 sehen wir den Typ direkt an ;-)
- Die Typen für die inneren Knoten erschießen wir mithilfe von Typ-Regeln.

Formal betrachten wir Aussagen der Form:

$$\Gamma \vdash e \, : \, t$$

// (In der Typ-Umgebung Γ hat e den Typ t)

Axiome:

Const: $\Gamma \vdash c \, : \, t_c$ (t_c Typ der Konstante c)

Var: $\Gamma \vdash x \, : \, \Gamma(x)$ (x Variable)

Regeln:

Ref: $\dfrac{\Gamma \vdash e \, : \, t}{\Gamma \vdash \& e \, : \, t*}$ Deref: $\dfrac{\Gamma \vdash e \, : \, t*}{\Gamma \vdash *e \, : \, t}$

Array: $$\frac{\Gamma \vdash e_1 \ : \ t* \qquad \Gamma \vdash e_2 \ : \ \textbf{int}}{\Gamma \vdash e_1[e_2] \ : \ t}$$

Array: $$\frac{\Gamma \vdash e_1 \ : \ t\,[\,] \qquad \Gamma \vdash e_2 \ : \ \textbf{int}}{\Gamma \vdash e_1[e_2] \ : \ t}$$

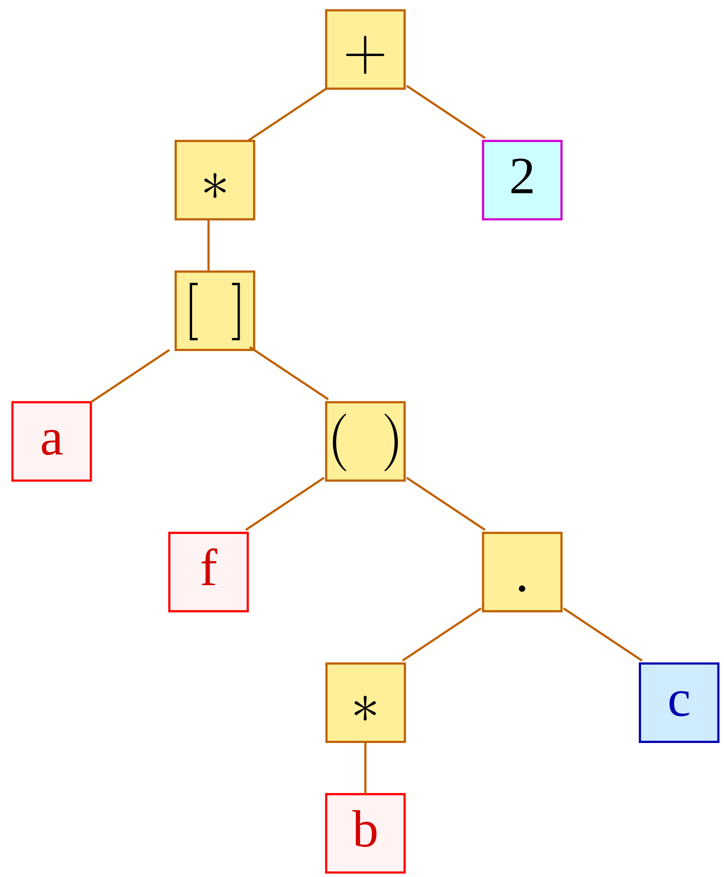
Struct: $$\frac{\Gamma \vdash e \ : \ \textbf{struct}\ \{t_1\ a_1; \ldots t_m\ a_m;\}}{\Gamma \vdash e.a_i \ : \ t_i}$$

App: $$\frac{\Gamma \vdash e \ : \ t\,(t_1, \ldots, t_m) \qquad \Gamma \vdash e_1 \ : \ t_1 \quad \ldots \quad \Gamma \vdash e_m \ : \ t_m}{\Gamma \vdash e(e_1, \ldots, e_m) \ : \ t}$$

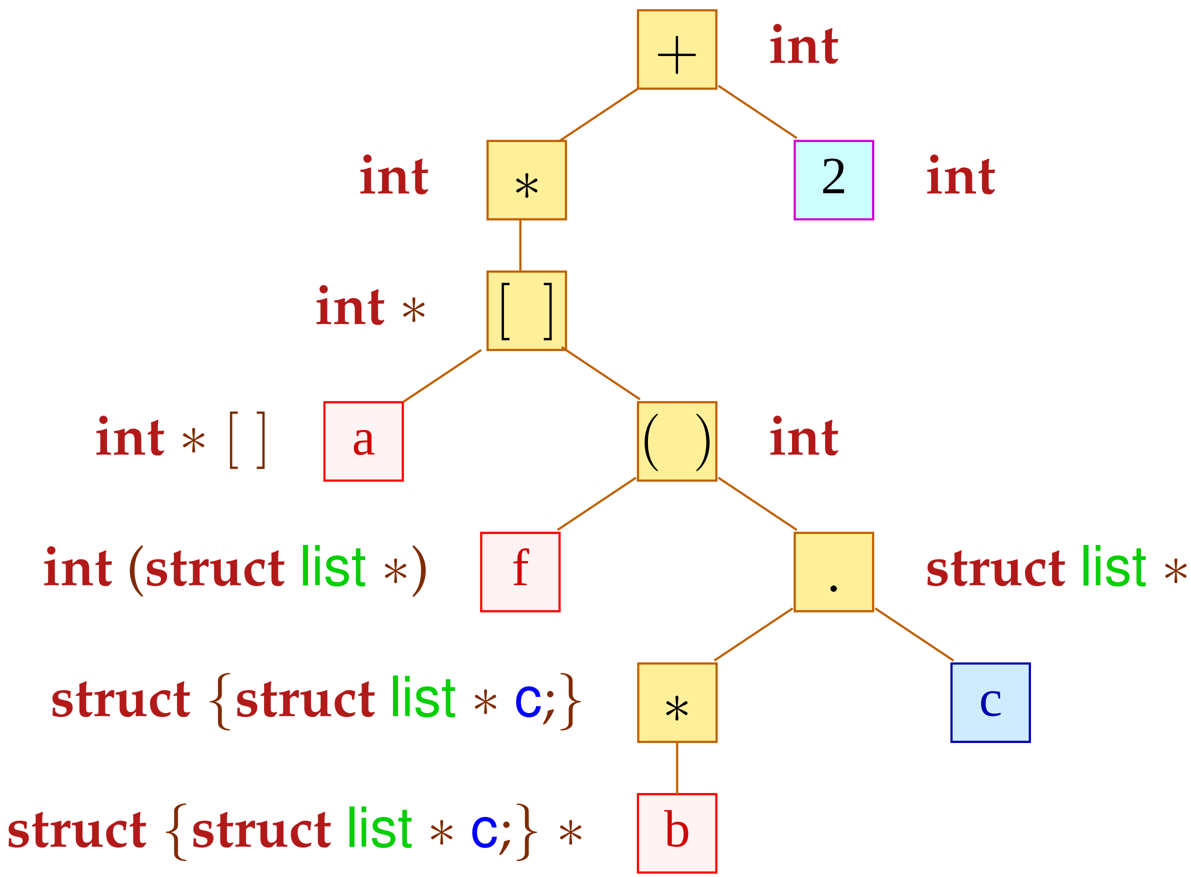
Op: $$\frac{\Gamma \vdash e_1 \ : \ \textbf{int} \qquad \Gamma \vdash e_2 \ : \ \textbf{int}}{\Gamma \vdash e_1 + e_2 \ : \ \textbf{int}}$$

Cast: $$\frac{\Gamma \vdash e \ : \ t_1 \qquad t_1 \text{ in } t_2 \text{ konvertierbar}}{\Gamma \vdash (t_2)\ e \ : \ t_2}$$

... im Beispiel:



... im Beispiel:



+
int
*
int
2
int
int ∗
[ ]
int ∗ [ ]
a
( )
int
int (struct list ∗)
f
.
struct list ∗
struct {struct list ∗ c;}
*
c
struct {struct list ∗ c;} ∗
b

Diskussion:

- Welche Regel an einem Knoten angewendet werden muss, ergibt sich aus den Typen für die bereits bearbeiteten Kinderknoten :-)
- Dazu muss die Gleichheit von Typen festgestellt werden.

  Achtung:

  struct A {} und **struct** B {} werden als verschieden betrachtet !!

  Nach:

  typedef int C;

  bezeichnen C und **int** immer noch den gleichen Typ :-)

- ...

Diskussion (Forts.):

- ...
- Manche Operatoren wie z.B. $+$ sind überladen: sie besitzen mehrere verschiedene Bedeutungen.
- Welche Bedeutung ausgewählt werden soll, entscheidet sich aufgrund der Argument-Typen. Der Operator $+$ kann zum Beispiel bedeuten:

  $\longrightarrow$ Addition auf **short**, **int**, **long**, **float** oder **double** :-)

  $\longrightarrow$ Pointer-Arithmetik :-))

- Ist die Bedeutung ermittelt, wird (in bestimmten Fällen) für das Argument, das noch nicht den richtigen Typ hat, eine Typ-Konvertierung eingefügt.

Strukturelle Typ-Gleichheit:

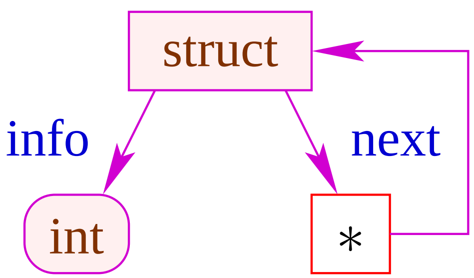
Semantisch können wir zwei rekursive Typen t_1, t_2 als gleich betrachten, falls sie die gleiche Menge von Pfaden zulassen.

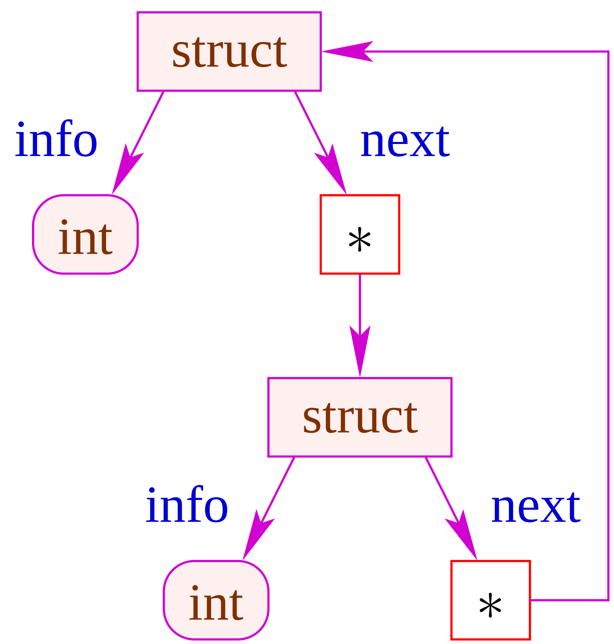
Beispiel:

```
struct list {                 struct list1 {
    int info;                     int info;
    struct list * next;           struct {
}                                     int info;
                                      struct list1 * next;
                                  } * next;
                              }
```

Rekursive Typen können wir als gerichtete Graphen darstellen.

... im Beispiel:





Beobachtung:

- Hat ein Knoten mehr als einen Nachfolger, tragen die ausgehenden Kanten unterschiedliche Beschriftungen :-)
- Das kann man auch für Funktions-Knoten erreichen :-)
- Der Typgraph kann damit als deterministischer endlicher Automat aufgefasst werden, der alle Pfade durch den Typ akzeptiert :-))
- Zwei Typen können wir dann als äquivalent auffassen, wenn ihre Typgraphen, aufgefasst als DFAs äquivalent sind.
- Insbesondere gibt es stets einen eindeutig bestimmten minimalen Typgraphen für jeden Typ :-)
- Strukturelle Äquivalenz rekursiver Typen ist deshalb schnell entscheidbar !!!

Alternativer Algorithmus:

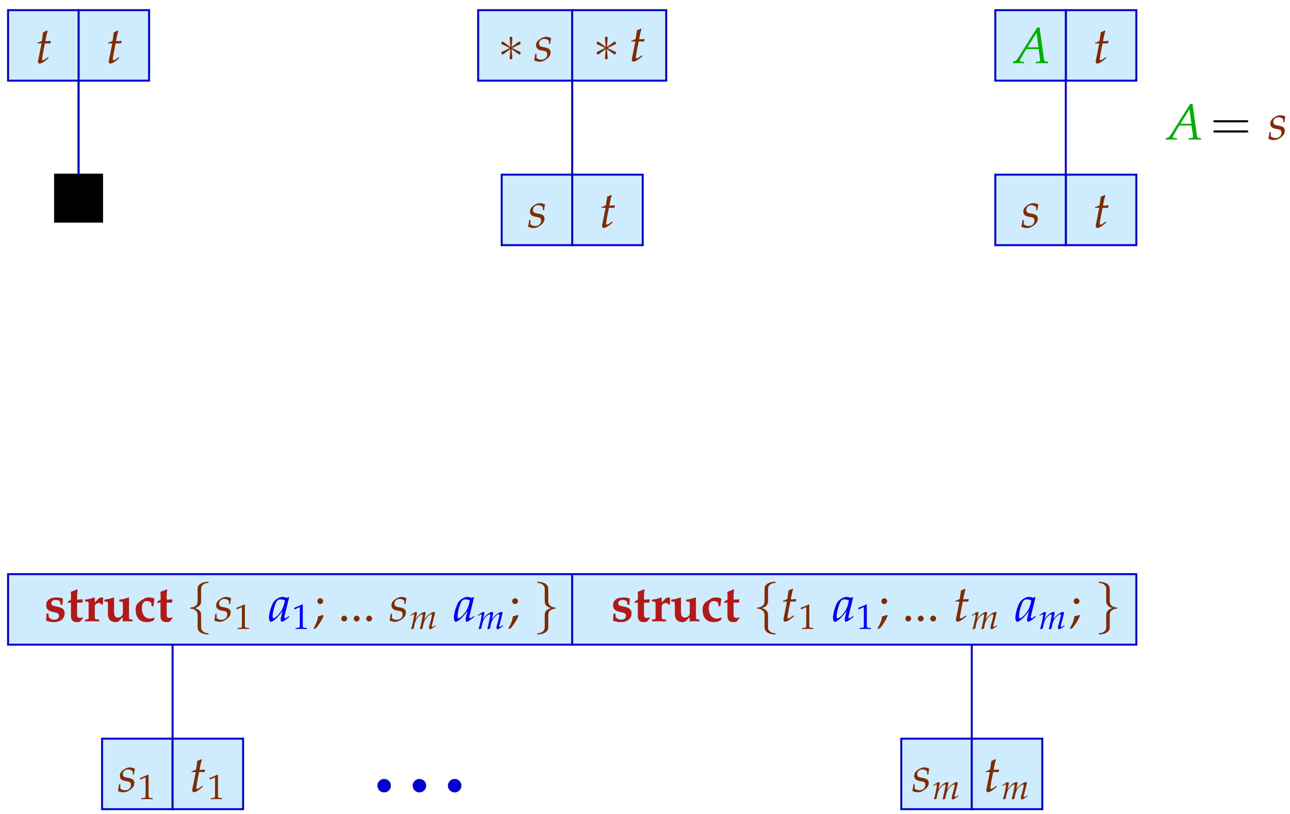
Idee:

- Verwalte Äquivalenz-Anfragen für je zwei Typausdrücke ...
- Sind die beiden Ausdrücke syntaktisch gleich, ist alles gut :-)
- Andernfalls reduziere die Äquivalenz-Anfrage zwischen Äquivalenz-Anfragen zwischen (hoffentlich einfacheren anderen Typausdrücken :-)

Nehmen wir an, rekursive Typen würden mithilfe von Typ-Gleichungen der Form:

$$A = t$$

eingeführt ...


t t
∗s ∗t
s t
A t
A = s
s t
struct {s1 a1; ... sm am; } struct {t1 a1; ... tm am; }
s1 t1
sm tm

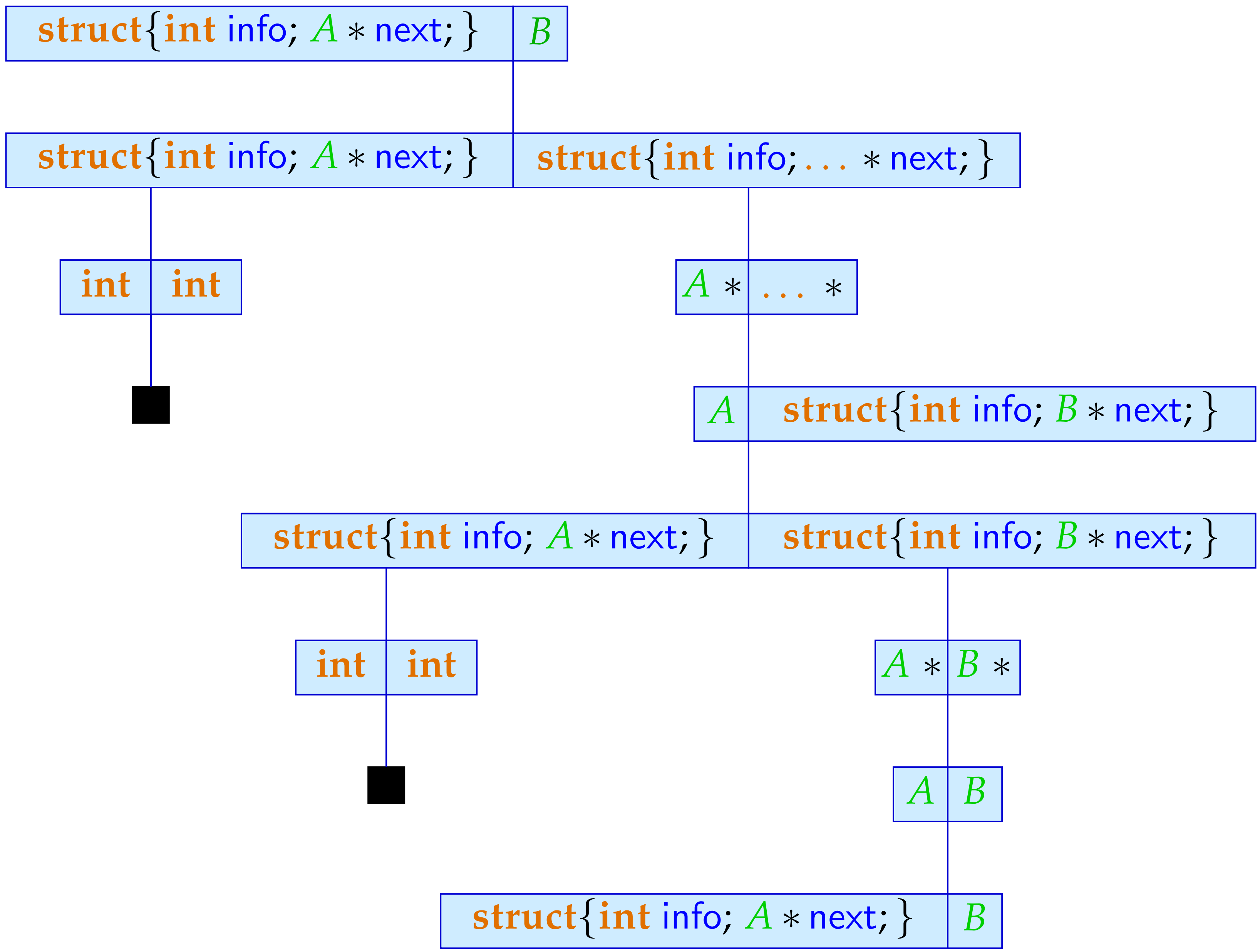
... im Beispiel:

$$\begin{array}{lll} A & = & \textbf{struct}\ \{\textbf{int}\ \mathsf{info};\ A * \mathsf{next};\} \\ B & = & \textbf{struct}\ \{\textbf{int}\ \mathsf{info}; \\ & & \textbf{struct}\ \{\textbf{int}\ \mathsf{info};\ B * \mathsf{next};\} * \mathsf{next};\} \end{array}$$

Wir fragen uns etwa, ob gilt:

$$\textbf{struct}\ \{\textbf{int}\ \mathsf{info};\ A * \mathsf{next};\} = B$$

Dazu konstruieren wir:


struct{int info; A * next; }
B
struct{int info; A * next; }
struct{int info; . . . * next; }
int
int
A *
. . . *
A
struct{int info; B * next; }
struct{int info; A * next; }
struct{int info; B * next; }
int
int
A *
B *
A
B
struct{int info; A * next; }
B

Diskussion:

- Stoßen wir bei der Konstruktion des Beweisbaums auf eine Äquivalenz-Anfrage, auf die keine Regel anwendbar ist, gibt es einen Widerspruch !!!
- Die Konstruktion des Beweisbaums kann dazu führen, dass die gleiche Äquivalenz-Anfrage ein weiteres Mal auftritt ...
- Taucht eine Äquivalenz-Anfrage ein weiteres Mal auf, können wir hier abbrechen ;-)

$\Longrightarrow$ die Anzahl zu betrachtender Anfragen ist endlich :-)

$\Longrightarrow$ das Verfahren terminiert :-))

Teiltypen

- Auf den arithmetischen Basistypen **char**, **int**,**long**, ... gibt es i.a. eine reichhaltige Teiltypen-Beziehungen.
- Dabei bedeutet $t_1 \leq t_2$, dass die Menge der Werte vom Typ t_1

  (1) eine Teilmenge der Werte vom Typ t_2 sind :-)

  (2) in einen Wert vom Typ t_2 konvertiert werden können :-)

  (3) die Anforderungen an Werte vom Typ t_2 erfüllen ...

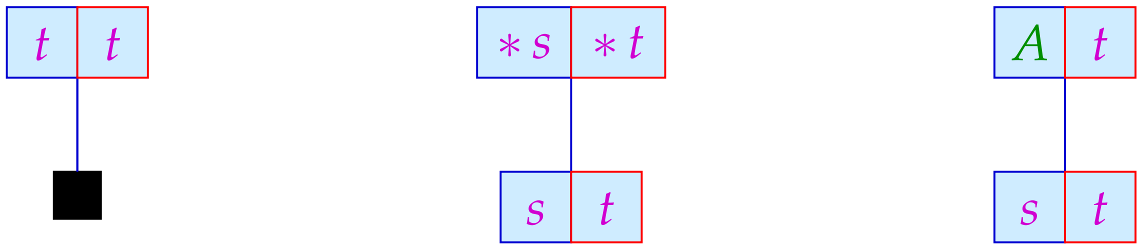
$\Longrightarrow$

Erweitere Teiltypen-Beziehungen der Basistypen auf komplexe Typen :-)

Beispiel:

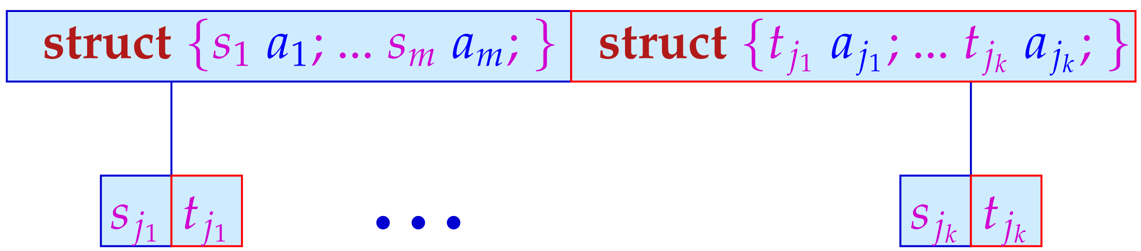
```
string extractInfo (struct { string info; } x) {
        return x.info;
}
```

- Offenkundig funktioniert extractInfo für alle Argument-Strukturen, die eine Komponente **string** info besitzen :-)
- Die Idee ist vergleichbar zur Anwendbarkeit auf Unterklassen (aber allgemeiner :-)
- Wann $t_1 \leq t_2$ gelten soll, beschreiben wir durch Regeln ...

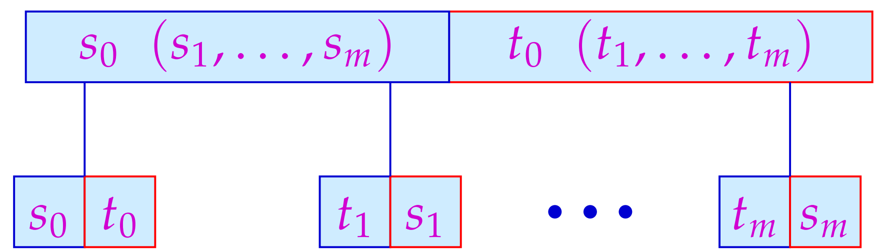


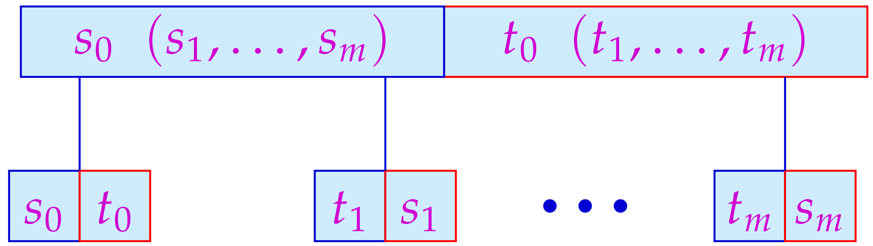
t
t
∗ s
∗ t
s
t
A
t
s
t

$A = s$



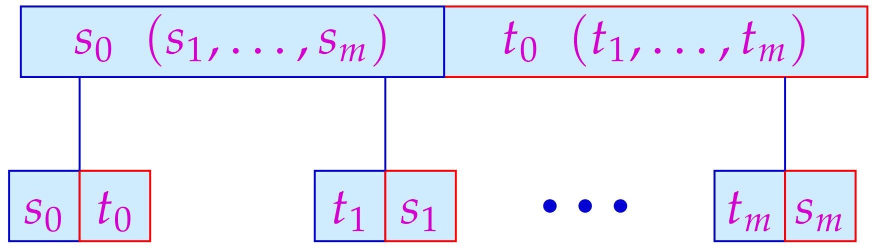
struct {s1 a1; ... sm am; }
struct {tj1 aj1; ... tjk ajk; }
sj1
tj1
sjk
tjk


$s_0 \ (s_1, \ldots, s_m)$
$t_0 \ (t_1, \ldots, t_m)$
s_0
t_0
t_1
s_1
t_m
s_m



Beispiele:

$$\textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\} \quad \leq \quad \textbf{struct}\ \{\textbf{float}\ a;\}$$
$$\textbf{int}\ (\textbf{int}) \quad \not\leq \quad \textbf{float}\ (\textbf{float})$$



Beispiele:

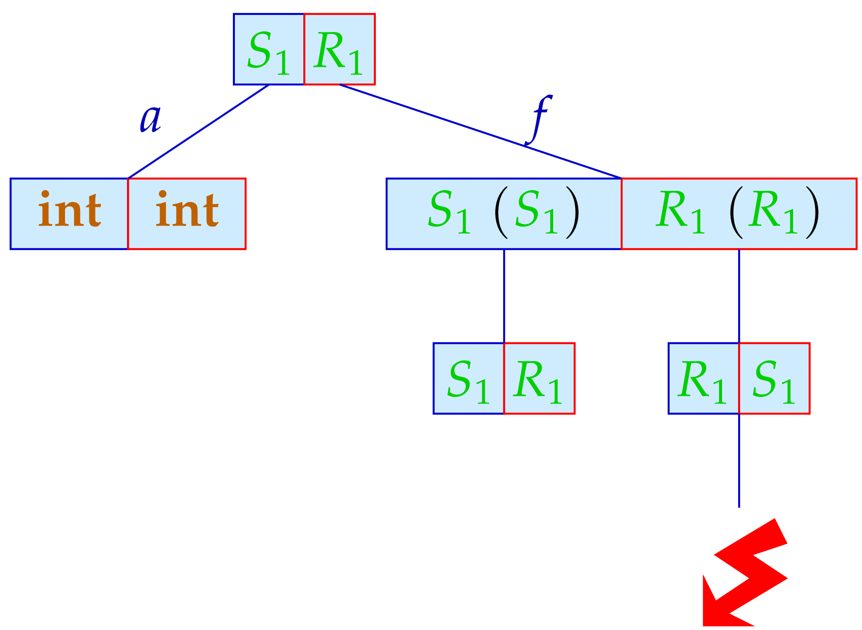
$$\textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\} \quad \leq \quad \textbf{struct}\ \{\textbf{float}\ a;\}$$
$$\textbf{int}\ (\textbf{int}) \quad \not\leq \quad \textbf{float}\ (\textbf{float})$$

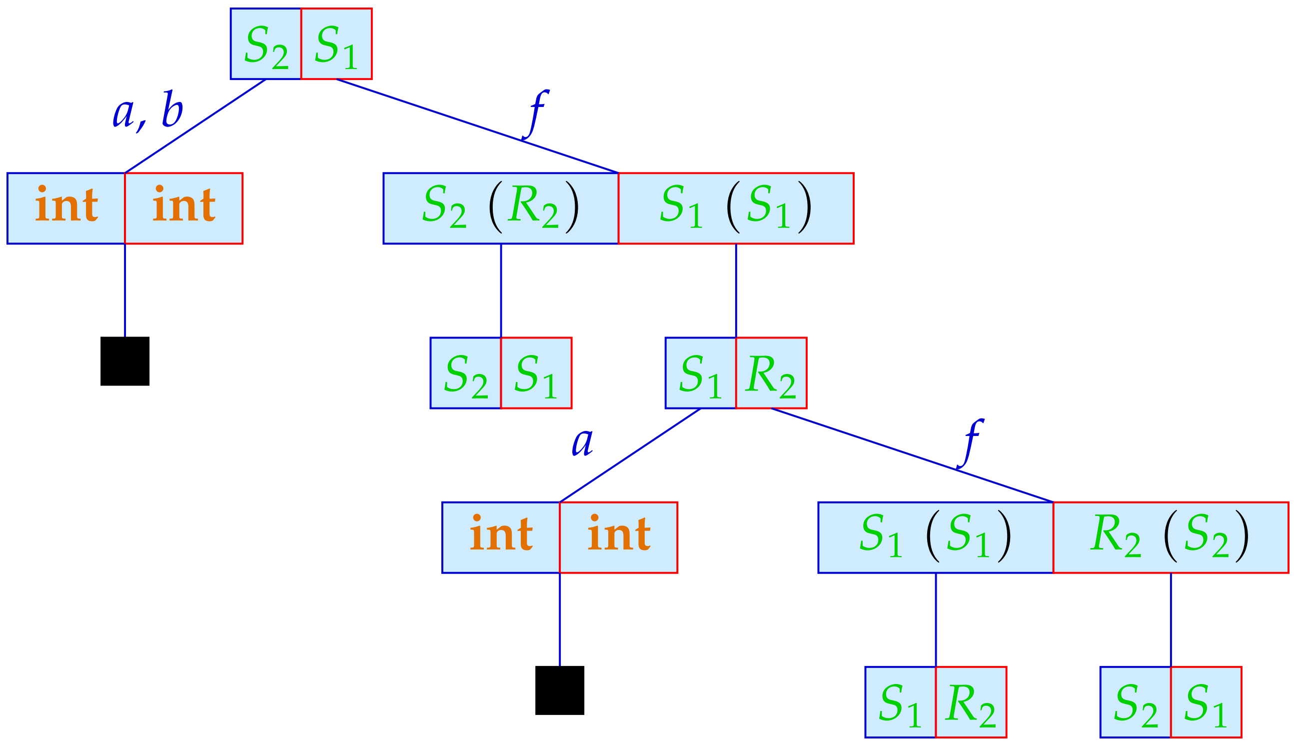
Achtung:

- Bei den Argumenten dreht sich die Anordnung der Typen gerade um !!!
- Diese Regeln können wir direkt benutzen, um auch für rekursive Typen die Teiltyp-Relation zu entscheiden :-)

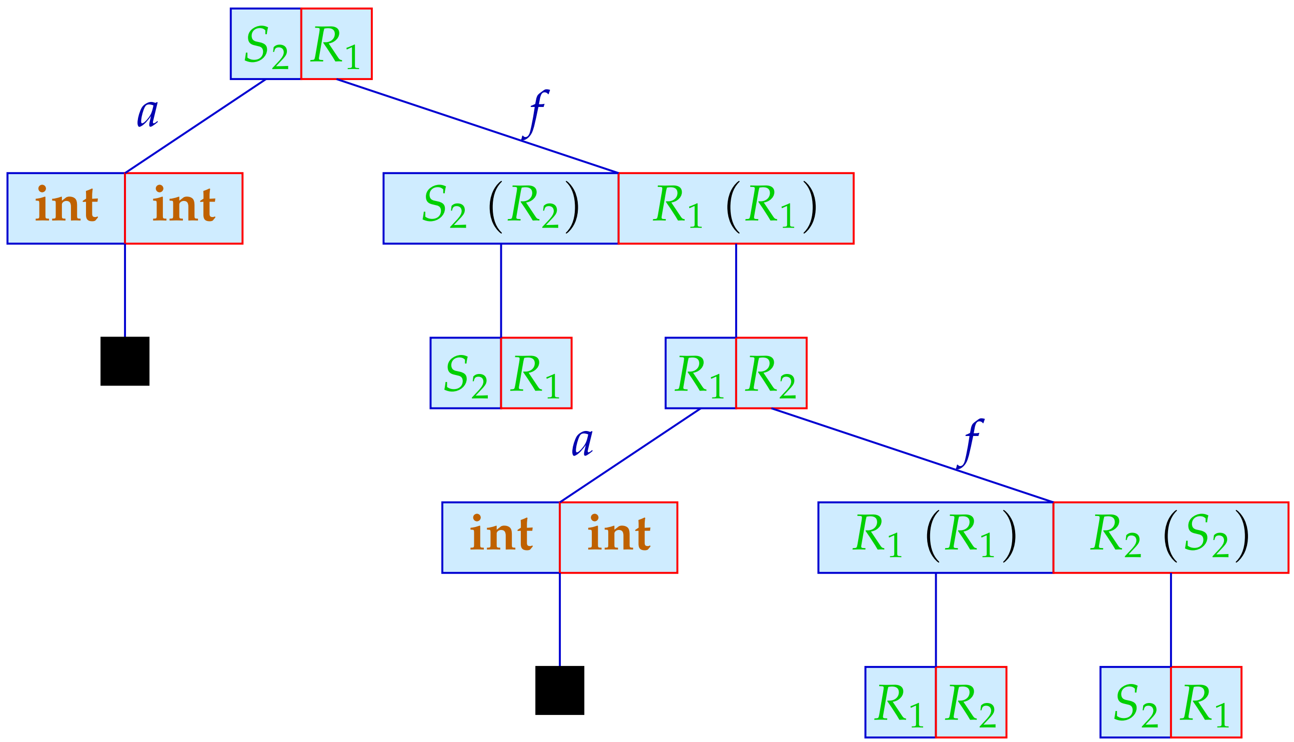
Beispiel:

$$\begin{array}{lll} R_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_1\ (R_1)\ f;\} \\ S_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\ (S_1)\ f;\} \\ R_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_2\ (S_2)\ f;\} \\ S_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_2\ (R_2)\ f;\} \end{array}$$





S_2 S_1
a, b
f
int int
$S_2\ (R_2)$ $S_1\ (S_1)$
S_2 S_1
S_1 R_2
a
f
int int
$S_1\ (S_1)$ $R_2\ (S_2)$
S_1 R_2
S_2 S_1


S_2 R_1
a
f
int int
$S_2\,(R_2)$ $R_1\,(R_1)$
S_2 R_1
R_1 R_2
a
f
int int
$R_1\,(R_1)$ $R_2\,(S_2)$
R_1 R_2
S_2 R_1

Diskussion:

- Um die Beweisbäume nicht in den Himmel wachsen zu lassen, wurden einige Zwischenknoten ausgelassen :-)
- Strukturelle Teiltypen sind sehr mächtig und deshalb nicht ganz leicht zu durchschauen.
- Java verallgemeinert Strukturen zu Objekten / Klassen.
- Teiltyp-Beziehungen zwischen Klassen müssen explizit deklariert werden :-)
- Durch Vererbung wird sichergestellt, dass Unterklassen über die (sichtbaren) Komponenten der Oberklasse verfügen :-))
- Überschreiben einer Komponente mit einem spezielleren Typ ist möglich — aber nur, wenn diese keine Methode ist :-(

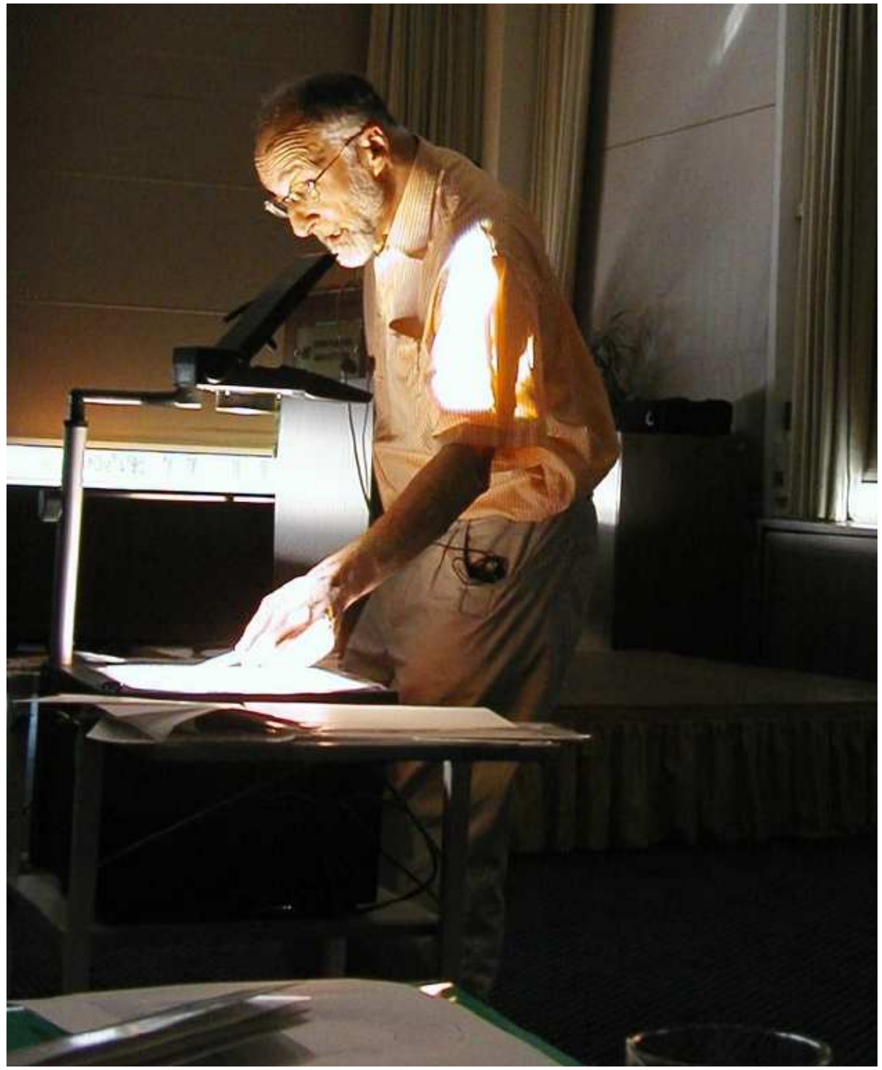
3.3 Inferieren von Typen

- Im Gegensatz zu imperativen Sprachen kann in funktionalen Programmiersprachen der Typ von Bezeichnern (i.a.) weggelassen werden.
- Diese werden dann automatisch hergeleitet :-)

Beispiel:

$$\begin{array}{lll} \textbf{fun}\ \ \mathsf{fac}\,\mathsf{x} & = & \textbf{if}\ \ \mathsf{x} \leq 0\ \ \textbf{then}\ \ 1 \\ & & \textbf{else}\ \ \mathsf{x} \cdot \mathsf{fac}\ (\mathsf{x} - 1) \end{array}$$

Dafür findet der SML-Compiler: $\mathsf{fac}\ :\ \textbf{int} \to \textbf{int}$



Robin (Dumbledore) Milner, Edinburgh

Idee: J.R. Hindley, R. Milner

Stelle Axiome und Regeln auf, die den Typ eines Ausdrucks in Beziehung setzen zu den Typen seiner Teilausdrücke :-)

Der Einfachheit halber betrachten wir nur eine funktionale Kernsprache ...

$$
\begin{array}{lll}
e & ::= & b \mid x \mid (\Box_1\ e) \mid (e_1\ \Box_2\ e_2) \\
& \mid & (\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) \\
& \mid & (e_1, \ldots, e_k) \mid [\,] \mid (e_1\ :\ e_2) \\
& \mid & (\mathbf{case}\ e_0\ \mathbf{of}\ [\,] \rightarrow e_1;\ h : t \rightarrow e_2) \\
& \mid & (e_1\ e_2) \mid (\mathbf{fn}\ (x_1, \ldots, x_m) \Rightarrow e) \\
& \mid & (\mathbf{letrec}\ x_1 = e_1; \ldots; x_n = e_n\ \mathbf{in}\ e_0) \\
& \mid & (\mathbf{let}\ x_1 = e_1; \ldots; x_n = e_n\ \mathbf{in}\ e_0)
\end{array}
$$

Beispiel:

$$
\begin{array}{rlcl}
\textbf{letrec} & \mathsf{rev} & = & \textbf{fn}\ x \Rightarrow \mathsf{r}\ x\ [\,]; \\
 & \mathsf{r} & = & \textbf{fn}\ x \Rightarrow \textbf{fn}\ y \Rightarrow\ \ \textbf{case}\ x\ \textbf{of} \\
 & & & \qquad\qquad\quad [\,] \ \to\ y; \\
 & & & \qquad\qquad\quad h : t \ \to\ \mathsf{r}\ t\ (h\ :\ y) \\
\textbf{in} & \multicolumn{3}{l}{\mathsf{rev}\ (1\ :\ 2\ :\ 3\ :\ [\,])}
\end{array}
$$

Wir benutzen die üblichen Präzedenz-Regeln und Assoziativitäten, um hässliche Klammern zu sparen :-)

Als einzige Datenstrukturen betrachten wir Tupel und List :-))

Wir benutzen eine Syntax von Typen, die an SML angelehnt ist ...

$$t \quad ::= \quad \textbf{int} \mid \textbf{bool} \mid (t_1, \ldots, t_m) \mid \mathsf{list}\ t \mid t_1 \ \rightarrow \ t_2$$

Wir betrachten wieder Typ-Aussagen der Form:

$$\Gamma \ \vdash \ e \ : \ t$$

Wir benutzen eine Syntax von Typen, die an SML angelehnt ist ...

$$t \quad ::= \quad \mathbf{int} \mid \mathbf{bool} \mid (t_1, \ldots, t_m) \mid \mathsf{list}\ t \mid t_1 \rightarrow t_2$$

Wir betrachten wieder Typ-Aussagen der Form:

$$\Gamma \vdash e \ : \ t$$

Axiome:

Const:	$\Gamma \vdash c \ : \ t_c$	(t_c Typ der Konstante c)
Nil:	$\Gamma \vdash [\,] \ : \ \mathsf{list}\ t$	(t beliebig)
Var:	$\Gamma \vdash x \ : \ \Gamma(x)$	(x Variable)

Regeln:

Op:
$$\frac{\Gamma \vdash e_1 : \mathbf{int} \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}}$$

If:
$$\frac{\Gamma \vdash e_0 : \mathbf{bool} \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash (\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) : t}$$

Tupel:
$$\frac{\Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_m : t_m}{\Gamma \vdash (e_1, \ldots, e_m) : (t_1, \ldots, t_m)}$$

App:
$$\frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\ e_2) : t_2}$$

Fun:
$$\frac{\Gamma \oplus \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\} \vdash e : t}{\Gamma \vdash \mathbf{fn}\ (x_1, \ldots, x_m) \Rightarrow e : (t_1, \ldots, t_m) \to t}$$

...

$\ldots$

Cons: $$\frac{\Gamma \vdash e_1 \; : \; t \qquad \Gamma \vdash e_2 \; : \; \mathsf{list}\ t}{\Gamma \vdash (e_1 : e_2) \; : \; \mathsf{list}\ t}$$

Case: $$\frac{\Gamma \vdash e_0 \; : \; \mathsf{list}\ t_1 \qquad \Gamma \vdash e_1 \; : \; t \qquad \Gamma \oplus \{x \mapsto t_1, y \mapsto \mathsf{list}\ t_1\} \vdash e_2 \; : \; t}{\Gamma \vdash (\textbf{case}\ e_0\ \textbf{of}\ [\,] \longrightarrow e_1;\ x : y \longrightarrow e_2) \; : \; t}$$

Letrec: $$\frac{\Gamma' \vdash e_1 \; : \; t_1 \quad \ldots \quad \Gamma' \vdash e_m \; : \; t_m \quad \Gamma' \vdash e_0 \; : \; t}{\Gamma \vdash (\textbf{letrec}\ x_1 = e_1; \ldots; x_m = e_m\ \textbf{in}\ e_0) \; : \; t}$$

wobei $\quad \Gamma' = \Gamma \oplus \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\}$

$\ldots$

Cons:
$$\frac{\Gamma \vdash e_1 \;:\; t \qquad \Gamma \vdash e_2 \;:\; \textsf{list}\; t}{\Gamma \vdash (e_1 : e_2) \;:\; \textsf{list}\; t}$$

Case:
$$\frac{\Gamma \vdash e_0 \;:\; \textsf{list}\; t_1 \qquad \Gamma \vdash e_1 \;:\; t \qquad \Gamma \oplus \{x \mapsto t_1, y \mapsto \textsf{list}\; t_1\} \vdash e_2 \;:\; t}{\Gamma \vdash (\textbf{case}\; e_0\; \textbf{of}\; [\,] \to e_1;\; x : y \to e_2) \;:\; t}$$

Letrec:
$$\frac{\Gamma' \vdash e_1 \;:\; t_1 \quad \ldots \quad \Gamma' \vdash e_m \;:\; t_m \quad \Gamma' \vdash e_0 \;:\; t}{\Gamma \vdash (\textbf{letrec}\; x_1 = e_1; \ldots; x_m = e_m\; \textbf{in}\; e_0) \;:\; t}$$

$$\text{wobei} \quad \Gamma' = \Gamma \oplus \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\}$$

Könnten wir die Typen für alle Variablen-Vorkommen raten, ließe sich mithilfe der Regeln überprüfen, dass unsere Wahl korrekt war :-)

$\ldots$

Cons:
$$\frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : \textsf{list}\ t}{\Gamma \vdash (e_1 : e_2) : \textsf{list}\ t}$$

Case:
$$\frac{\Gamma \vdash e_0 : \textsf{list}\ t_1 \qquad \Gamma \vdash e_1 : t \qquad \Gamma \oplus \{x \mapsto t_1, y \mapsto \textsf{list}\ t_1\} \vdash e_2 : t}{\Gamma \vdash (\textbf{case}\ e_0\ \textbf{of}\ [\,] \to e_1;\ x : y \to e_2) : t}$$

Letrec:
$$\frac{\Gamma' \vdash e_1 : t_1 \quad \ldots \quad \Gamma' \vdash e_m : t_m \quad \Gamma' \vdash e_0 : t}{\Gamma \vdash (\textbf{letrec}\ x_1 = e_1; \ldots; x_m = e_m\ \textbf{in}\ e_0) : t}$$

wobei $\Gamma' = \Gamma \oplus \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\}$

Könnten wir die Typen für alle Variablen-Vorkommen raten, ließe sich mithilfe der Regeln überprüfen, dass unsere Wahl korrekt war :-)

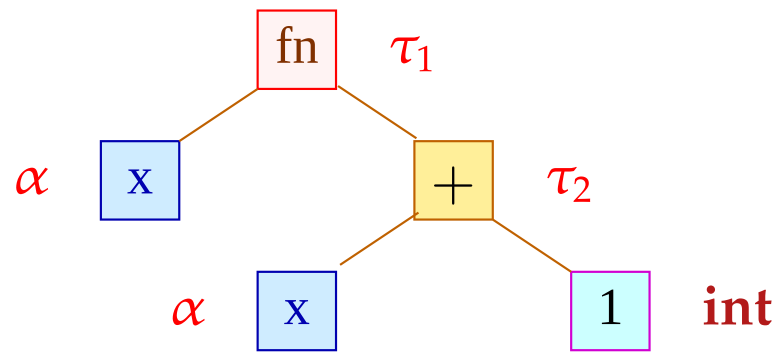
Wie raten wir die Typen der Variablen ???

Idee:

- Mache die Namen der verschiedenen Variablen eindeutig.
- Führe Typ-Variablen für die unbekannten Typen der Variablen und Teilausdrücke ein.
- Sammle die Gleichungen, die notwendigerweise zwischen den Typ-Variablen gelten müssen.
- Finde für diese Gleichungen Lösungen :-)

Beispiel:

$$\textbf{fn } x \Rightarrow x + 1$$



Gleichungen:

$$
\begin{array}{rcl}
\tau_1 & = & \alpha \to \tau_2 \\
\tau_2 & = & \mathbf{int} \\
\alpha & = & \mathbf{int}
\end{array}
$$

Wir schließen: $\tau_1 = \mathbf{int} \to \mathbf{int}$

Für jede Programm-Variable x und für jedes Vorkommen eines Teilausdrucks e führen wir die Typ-Variable $\alpha[x]$ bzw. $\tau[e]$ ein.

Jede Regel-Anwendung gibt dann Anlass zu einigen Gleichungen ...

Const:	$e \equiv c$	$\Longrightarrow$	$\tau[e] = \tau_c$
Nil:	$e \equiv [\,]$	$\Longrightarrow$	$\tau[e] = \mathsf{list}\,\alpha$ $\quad(\alpha$ neu$)$
Var:	$e \equiv x$	$\Longrightarrow$	$\tau[e] = \alpha[x]$
Op:	$e \equiv e_1 + e_2$	$\Longrightarrow$	$\tau[e] = \tau[e_1] = \tau[e_2] = \textbf{int}$
Tupel:	$e \equiv (e_1, \ldots, e_m)$	$\Longrightarrow$	$\tau[e] = (\tau[e_1], \ldots, \tau[e_m])$
Cons:	$e \equiv e_1 : e_2$	$\Longrightarrow$	$\tau[e] = \tau[e_2] = \mathsf{list}\ \tau[e_1]$
	$\ldots$		

$\ldots$

If:	$e \equiv \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2$	$\Longrightarrow$	$\tau[e_0] = \textbf{bool}$ $\tau[e] = \tau[e_1] = \tau[e_2]$
Case:	$e \equiv \textbf{case}\ e_0\ \textbf{of}\ [\,] \to e_1;\ x : y \to e_2$	$\Longrightarrow$	$\tau[e_0] = \alpha[y] = \mathsf{list}\ \alpha[x]$ $\tau[e] = \tau[e_1] = \tau[e_2]$
Fun:	$e \equiv \textbf{fn}\ (x_1, \ldots, x_m) \Rightarrow e_1$	$\Longrightarrow$	$\tau[e] = (\alpha[x_1], \ldots, \alpha[x_m]) \to \tau[e_1]$
App:	$e \equiv e_1\ e_2$	$\Longrightarrow$	$\tau[e_1] = \tau[e_2] \to \tau[e]$
Letrec:	$e \equiv \textbf{letrec}\ x_1 = e_1; \ldots; x_m = e_m\ \textbf{in}\ e_0$	$\Longrightarrow$	$\alpha[x_1] = \tau[e_1] \ldots$ $\alpha[x_m] = \tau[e_m]$ $\tau[e] = \tau[e_0]$

Bemerkung:

- Die möglichen Typ-Zuordnungen an Variablen und Programm-Ausdrücke erhalten wir als Lösung eines Gleichungssystems über Typ-Termen :-)
- Das Lösen von Systemen von Term-Gleichungen nennt man auch Unifikation :-)

Bemerkung:

- Die möglichen Typ-Zuordnungen an Variablen und Programm-Ausdrücke erhalten wir als Lösung eines Gleichungssystems über Typ-Termen :-)
- Das Lösen von Systemen von Term-Gleichungen nennt man auch Unifikation :-)

Beispiel:

$$g(z, f(x)) = g(f(x), f(a))$$

Eine Lösung dieser Gleichung ist die Substitution $\{x \mapsto a, z \mapsto f(a)\}$

In dem Fall ist das offenbar die einzige :-)

Satz:

Jedes System von Term-Gleichungen:

$$s_i = t_i \qquad i = 1, \ldots, m$$

hat entweder keine Lösung oder eine allgemeinste Lösung.

Satz:

Jedes System von Term-Gleichungen:

$$s_i = t_i \qquad i = 1, \ldots, m$$

hat entweder keine Lösung oder eine allgemeinste Lösung.

Eine allgemeinste Lösung ist eine Substitution σ mit den Eigenschaften:

- σ ist eine Lösung, d.h. $\sigma(s_i) = \sigma(t_i)$ für alle i.
- σ ist allgemeinst, d.h. für jede andere Lösung τ gilt: $\tau = \tau' \circ \sigma$ für eine Substitution τ' :-)

Beispiele:

(1) $f(a) = g(x)$ — hat keine Lösung :-)

(2) $x = f(x)$ — hat ebenfalls keine Lösung ;-)

(3) $f(x) = f(a)$ — hat genau eine Lösung:-)

(4) $f(x) = f(g(y))$ — hat unendlich viele Lösungen :-)

(5) $x_0 = f(x_1, x_1), \ldots, x_{n-1} = f(x_n, x_n)$ —

hat mindestens exponentiell große Lösungen !!!

Bemerkungen:

- Es gibt genau eine Lösung, falls die allgemeinste Lösung keine Variablen enthält, d.h. ground ist :-)
- Gibt es zwei verschiedene Lösungen, dann bereits unendlich viele ;-)
- Achtung: Es kann mehrere allgemeinste Lösungen geben !!!

  Beispiel: $x = y$

  Allgemeinste Lösungen sind : $\{x \mapsto y\}$ oder $\{y \mapsto x\}$

  Diese sind allerdings nicht sehr verschieden :-)
- Eine allgemeinste Lösung kann immer idempotent gewählt werden, d.h. $\sigma = \sigma \circ \sigma$.

  Beispiel: $x = x \qquad y = y$

  Nicht idempotente Lösung: $\{x \mapsto y, y \mapsto x\}$

  Idempotente Lösung: $\{x \mapsto x, y \mapsto y\}$

Berechnung einer allgemeinsten Lösung:

$$
\begin{array}{lll}
\textbf{fun}\ \ \mathsf{occurs}\,(x,t) & = & \textbf{case}\ \ t \\
& & \quad \begin{array}{lll}
\textbf{of}\ \ x & \to & \mathsf{true} \\
\mid\ \ f(t_1,\ldots,t_k) & \to & \mathsf{occurs}\,(x,t_1) \vee \ldots \vee \mathsf{occurs}\,(x,t_k) \\
\mid\ \ \_ & \to & \mathsf{false}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\textbf{fun}\ \ \mathsf{unify}\,(s,t)\,\theta\ \ =\ \ \textbf{if}\ \ \theta\,s \equiv \theta\,t\ \ \textbf{then}\ \ \theta \\
\qquad\qquad\quad \textbf{else}\ \ \textbf{case}\ \ (\theta\,s, \theta\,t) \\
\qquad\qquad\qquad \begin{array}{lll}
\textbf{of}\ \ (x,t) & \to & \textbf{if}\ \ \mathsf{occurs}\,(x,t)\ \ \textbf{then}\ \ \mathsf{Fail} \\
& & \quad\ \ \textbf{else}\ \ \{x \mapsto t\} \circ \theta \\
\mid\ \ (t,x) & \to & \textbf{if}\ \ \mathsf{occurs}\,(x,t)\ \ \textbf{then}\ \ \mathsf{Fail} \\
& & \quad\ \ \textbf{else}\ \ \{x \mapsto t\} \circ \theta \\
\mid\ \ (f(s_1,\ldots,s_k), f(t_1,\ldots,t_k)) & \to & \mathsf{unifyList}\ [(s_1,t_1),\ldots,(s_k,t_k)]\ \theta \\
\mid\ \ \_ & \to & \mathsf{Fail}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad \ldots \\
\textbf{and}\ \ \mathsf{unifyList}\ \mathsf{list}\ \theta \ \ = \ \ \textbf{case}\ \ \mathsf{list} \\
\qquad\qquad\qquad \textbf{of}\ \ [\,]\ \ \longrightarrow\ \ \theta \\
\qquad\qquad\qquad \ \ |\ \ \ ((s,t) :: \mathsf{rest})\ \ \longrightarrow\ \ \textbf{let val}\ \ \theta = \mathsf{unify}\ (s,t)\ \theta \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ \ \textbf{if}\ \ \theta = \mathsf{Fail}\ \ \textbf{then}\ \ \mathsf{Fail} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ \mathsf{unifyList}\ \mathsf{rest}\ \theta \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{end}
\end{array}
$$

$\ldots$

$$
\begin{array}{llll}
\textbf{and}\ \ \mathsf{unifyList}\ \mathsf{list}\ \theta & = & \textbf{case}\ \ \mathsf{list} & \\
 & & \textbf{of}\ \ [\,] \ \rightarrow\ \theta & \\
 & & \mid\ \ ((s,t) :: \mathsf{rest}) \ \rightarrow & \textbf{let val}\ \ \theta = \mathsf{unify}\ (s,t)\ \theta \\
 & & & \textbf{in}\ \ \textbf{if}\ \ \theta = \mathsf{Fail}\ \ \textbf{then}\ \ \mathsf{Fail} \\
 & & & \quad\ \textbf{else}\ \ \mathsf{unifyList}\ \mathsf{rest}\ \theta \\
 & & & \textbf{end}
\end{array}
$$

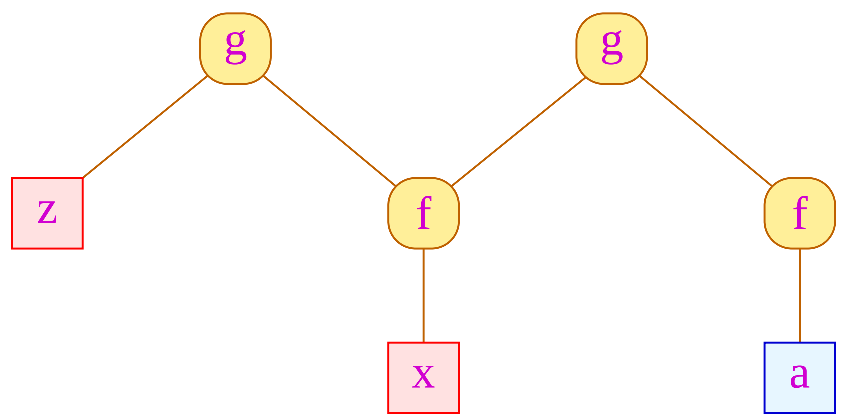
Diskussion:

- Der Algorithmus startet mit $\mathsf{unifyList}\ [(s_1, t_1), \ldots, (s_m, t_m)]\ \{\ \}$ ...
- Der Algorithmus liefert sogar eine idempotente allgemeinste Lösung :-)
- Leider hat er möglicherweise exponentielle Laufzeit :-(
- Lässt sich das verbessern ???

Idee:

- Wir repräsentieren die Terme der Gleichungen als Graphen.
- Dabei identifizieren wir bereits isomorphe Teilterme ;-)
- ...

... im Beispiel: $g(z, f(x)) \;=\; g(f(x), f(a))$

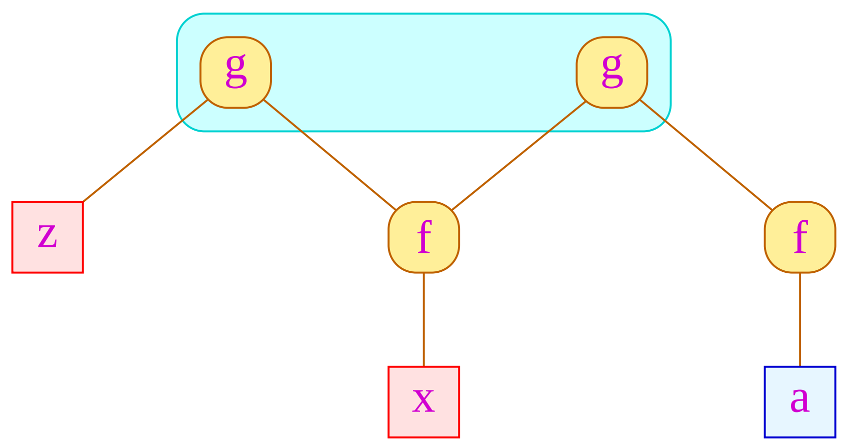


Idee:

- Wir repräsentieren die Terme der Gleichungen als Graphen.
- Dabei identifizieren wir bereits isomorphe Teilterme ;-)
- ...

... im Beispiel:

$$g(z, f(x)) \;=\; g(f(x), f(a))$$

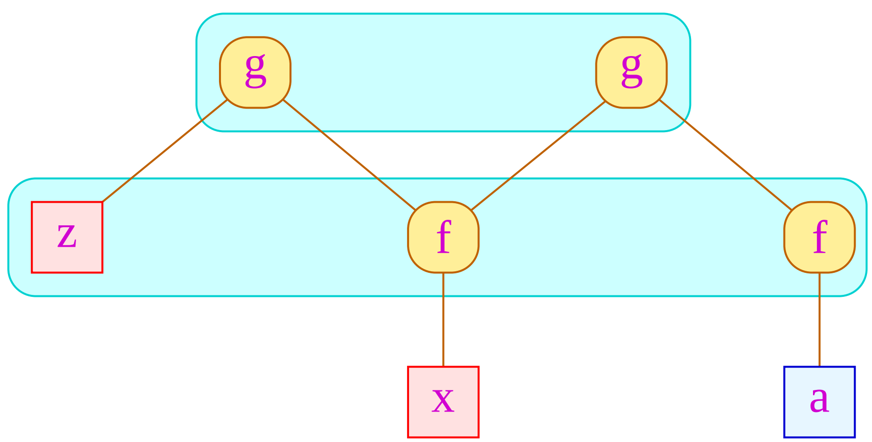


Idee:

- Wir repräsentieren die Terme der Gleichungen als Graphen.
- Dabei identifizieren wir bereits isomorphe Teilterme ;-)
- ...

... im Beispiel:

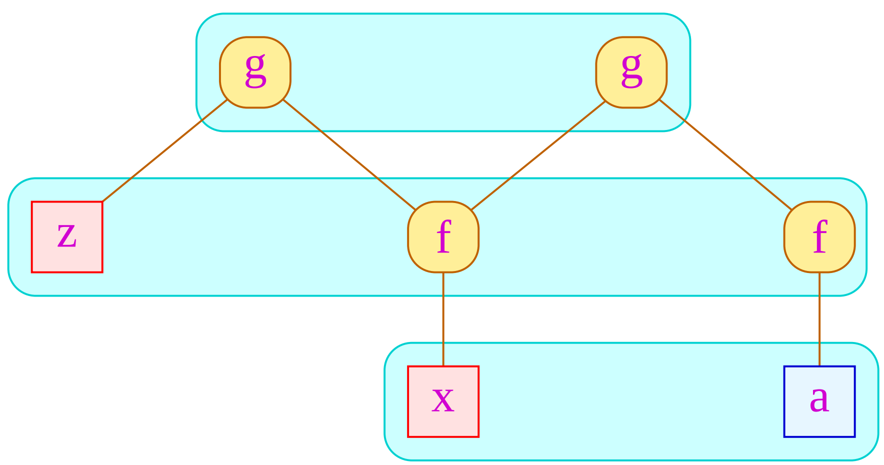
$$g(z, f(x)) = g(f(x), f(a))$$



Idee:

- Wir repräsentieren die Terme der Gleichungen als Graphen.
- Dabei identifizieren wir bereits isomorphe Teilterme ;-)
- ...

... im Beispiel: $g(z, f(x)) \;=\; g(f(x), f(a))$



Idee (Forts.):

- ...
- Wir berechnen eine Äquivalenz-Relation $\equiv$ auf den Knoten mit den folgenden Eigenschaften:

  → $s \equiv t$ für jede Gleichung unseres Gleichungssystems;

  → $s \equiv t$ nur, falls entweder s oder t eine Variable ist oder beide den gleichen Top-Konstruktor haben.

  → Falls $s \equiv t$ und $s = f(s_1, \ldots, s_k), t = f(t_1, \ldots, t_k)$ dann auch $s_1 \equiv t_1, \ldots, s_k \equiv t_k$.

Idee (Forts.):

- ...
- Wir berechnen eine Äquivalenz-Relation $\equiv$ auf den Knoten mit den folgenden Eigenschaften:
  - $\longrightarrow$ $s \equiv t$ für jede Gleichung unseres Gleichungssystems;
  - $\longrightarrow$ $s \equiv t$ nur, falls entweder s oder t eine Variable ist oder beide den gleichen Top-Konstruktor haben.
  - $\longrightarrow$ Falls $s \equiv t$ und $s = f(s_1, \ldots, s_k), t = f(t_1, \ldots, t_k)$ dann auch $s_1 \equiv t_1, \ldots, s_k \equiv t_k$.
- Falls keine solche Äquivalenz-Relation existiert, ist das System unlösbar.
- Falls eine solche Äquivalenz-Relation gilt, müssen wir überprüfen, dass der Graph modulo der Äquivalenz-Relation azyklisch ist.
- Ist er azyklisch, können wir aus der Äquivalenzklasse jeder Variable eine allgemeinste Lösung ablesen ...

Implementierung:

- Wir verwalten eine Partition der Knoten;
- Wann immer zwei Knoten äquivalent sein sollen, vereinigen wir ihre Äquivalenzklassen und fahren mit den Söhnen entsprechend fort.
- Notwendige Operationen auf der Datenstruktur π für eine Partition:

  → init (Nodes) liefert eine Repräsentation für die Partition $\pi_0 = \{\{v\} \mid v \in \text{Nodes}\}$

  → find (π, u) liefert einen Repräsentanten der Äquivalenzklasse — der wann immer möglich keine Variable sein soll :-)

  → union (π, u_1, u_2) vereinigt die Äquivalenzklassen von u_1, u_2 :-)

- Der Algorithmus startet mit einer Liste

$$W = [(u_1, v_1), \ldots, (u_m, v_m)]$$

der Paare von Wurzelknoten der zu unifizierenden Terme ...

```
π = init(Nodes);
while (W ≠ ∅) {
        (u, v) = Extract (W);
        u = find (π, u);  v = find (π, v);
        if (u ≢ v) {
                π = union (π, u, v);
                if (u ∉ Vars ∧ v ∉ Vars)
                        if (label(u) ≠ label(v)) return Fail
                        else {
                                (u_1, ..., u_k) = Successors(u);
                                (v_1, ..., v_k) = Successors(v);
                                W = (u_1, v_1) :: ... :: (u_k, v_k) :: W;
                        }
        }
}
```

Komplexität:

$\mathcal{O}(\#\, Knoten)$	Aufrufe von	union
$\mathcal{O}(\#\, Kanten + \#\, Gleichungen)$	Aufrufe von	find

$\Longrightarrow$ Wir benötigen effiziente Union-Find-Datenstruktur :-)

Komplexität:

$\mathcal{O}(\#\,Knoten)$	Aufrufe von	union
$\mathcal{O}(\#\,Kanten + \#\,Gleichungen)$	Aufrufe von	find

$\Longrightarrow$ Wir benötigen effiziente Union-Find-Datenstruktur :-)

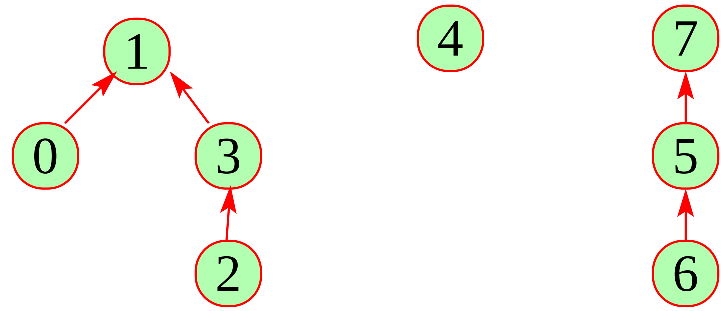
Idee:

Repräsentiere Partition von U als gerichteten Wald:

- Zu $u \in U$ verwalten wir einen Vater-Verweis $F[u]$.
- Elemente u mit $F[u] = u$ sind Wurzeln.

Einzelne Bäume sind Äquivalenzklassen.

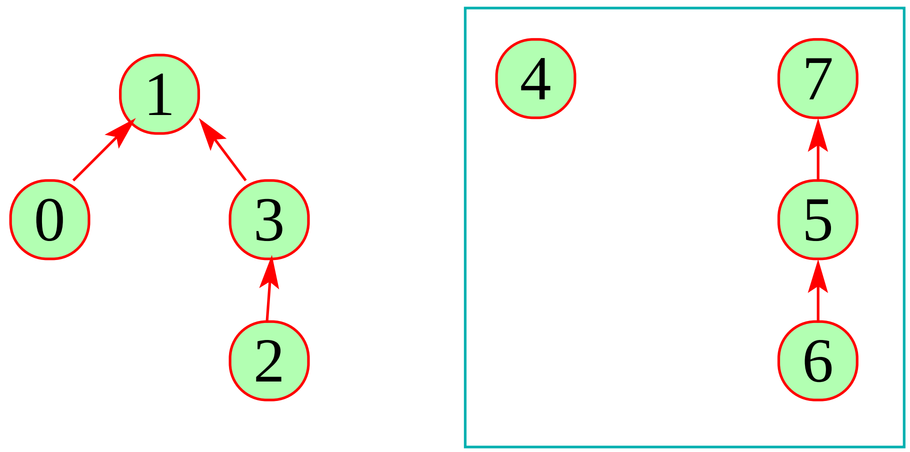
Ihre Wurzeln sind die Repräsentanten ...



0	1	2	3	4	5	6	7
1	1	3	1	4	7	5	7

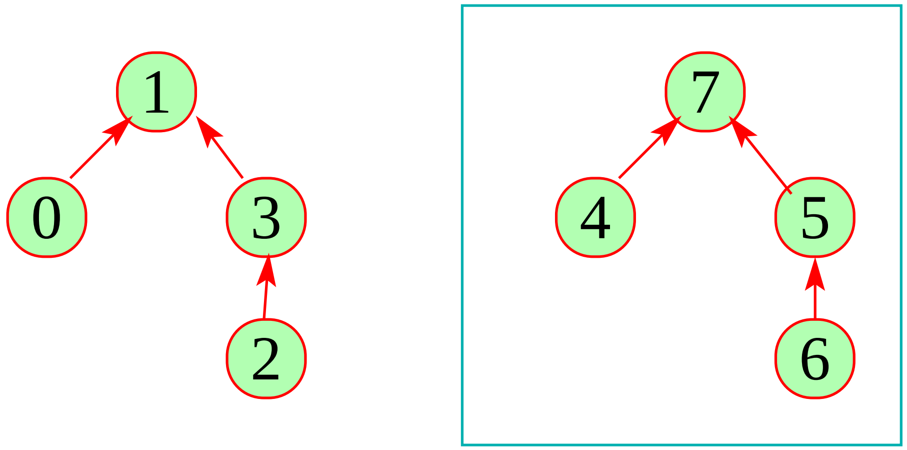
$\longrightarrow$ find (π, u) folgt den Vater-Verweisen :-)

$\longrightarrow$ union (π, u_1, u_2) hängt den Vater-Verweis eines u_i um ...



0	1	2	3	4	5	6	7

1	1	3	1	4	7	5	7



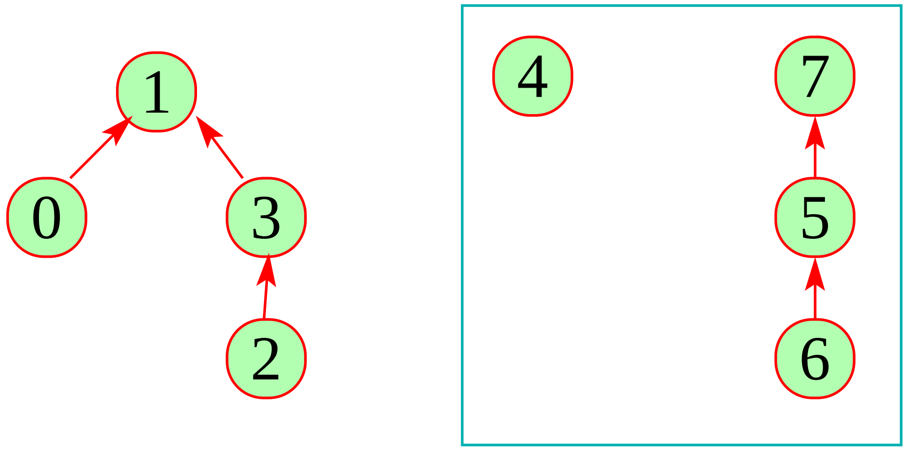
0	1	2	3	4	5	6	7
1	1	3	1	7	7	5	7

Die Kosten:

union	:	$\mathcal{O}(1)$	:-)
find	:	$\mathcal{O}(depth(\pi))$	:-(

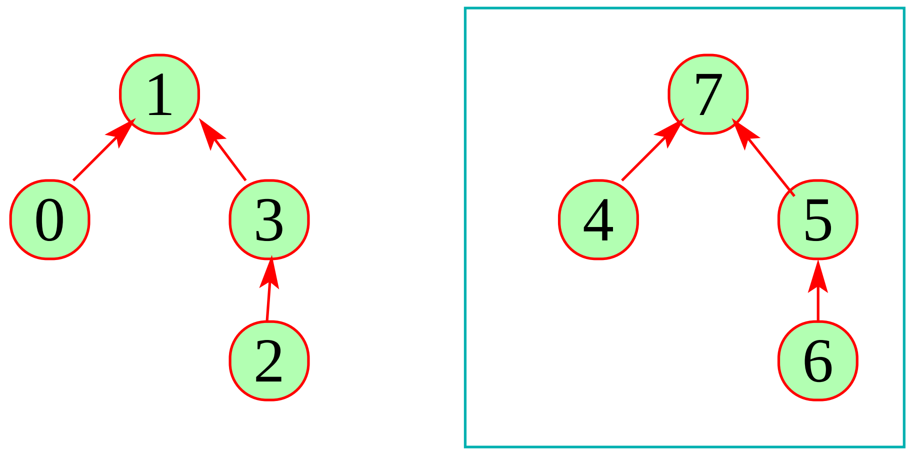
Strategie zur Vermeidung tiefer Bäume:

- Hänge den kleineren Baum unter den größeren !
- Benutze find , um Pfade zu komprimieren ...

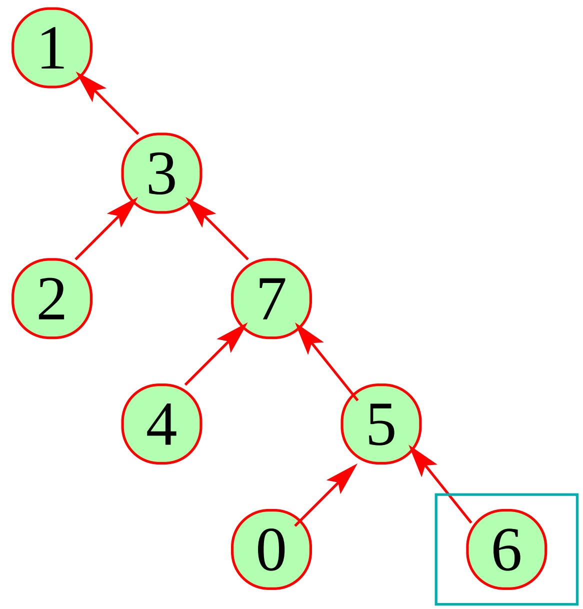


0	1	2	3	4	5	6	7

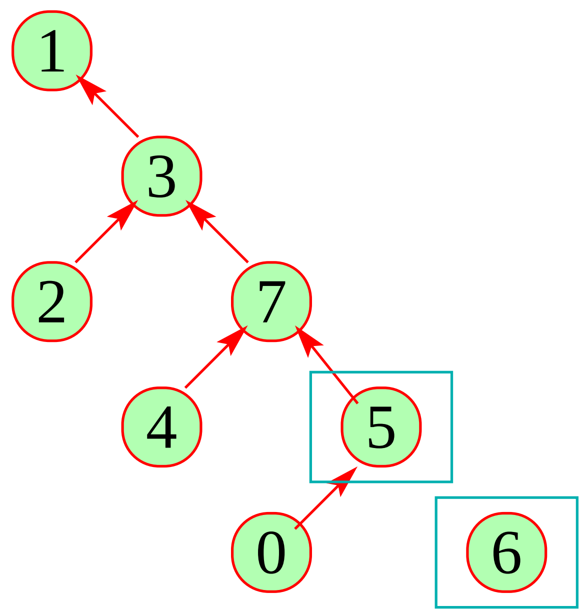
1	1	3	1	4	7	5	7



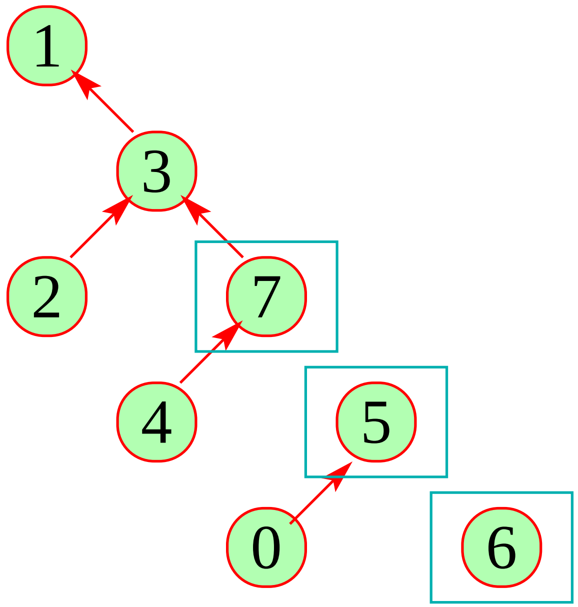
0	1	2	3	4	5	6	7
1	1	3	1	7	7	5	7



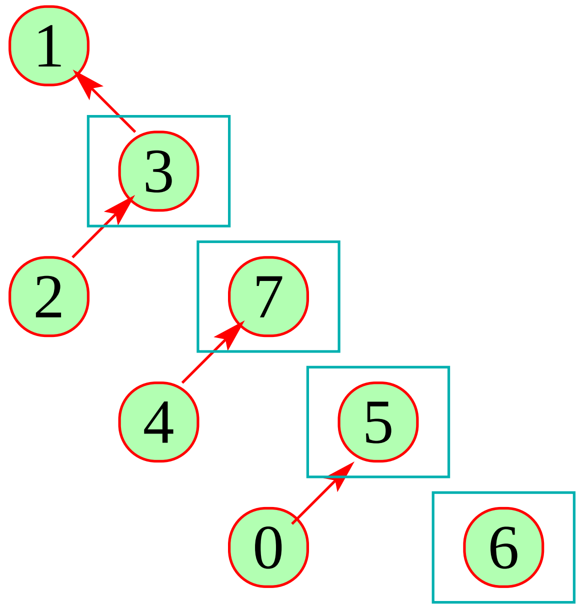
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



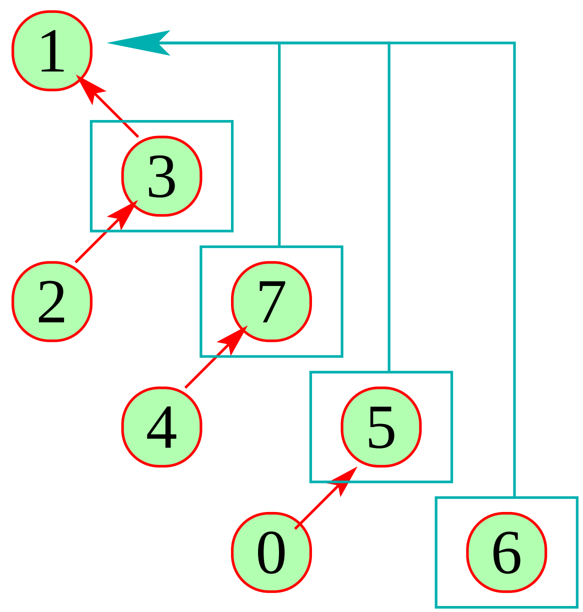
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



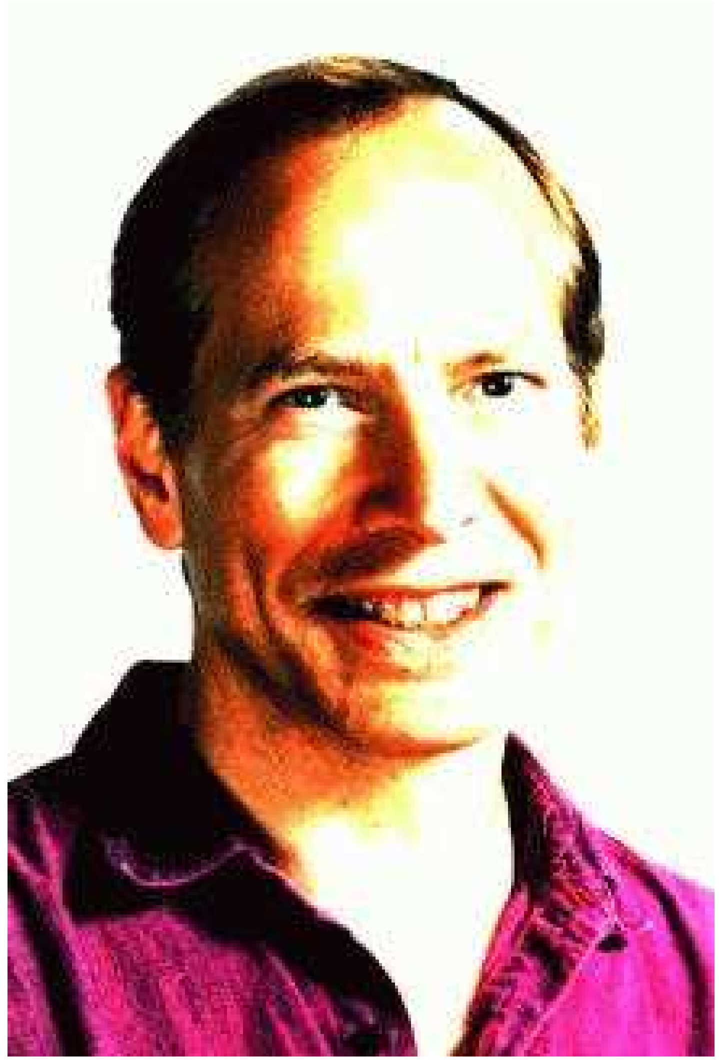
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	1	7	1	1



Robert Endre Tarjan, Princeton

Beachte:

- Mit dieser Datenstruktur dauern n union- und m find-Operationen $\mathcal{O}(n + m \cdot \alpha(n, n))$

  // α die inverse Ackermann-Funktion :-)

- Für unsere Anwendung müssen wir union nur so modifizieren, dass an den Wurzeln nach Möglichkeit keine Variablen stehen.
- Diese Modifikation vergrößert die asymptotische Laufzeit nicht :-)

Fazit:

- Wenn Typ-Gleichungen für ein Programm lösbar sind, dann gibt es eine allgemeinste Zuordnung von Programm-Variablen und Teil-Ausdrücken zu Typen, die alle Regeln erfüllen   :-)
- Eine solche allgemeinste Typisierung können wir in (fast) linearer Zeit berechnen   :-)

Fazit:

- Wenn Typ-Gleichungen für ein Programm lösbar sind, dann gibt es eine allgemeinste Zuordnung von Programm-Variablen und Teil-Ausdrücken zu Typen, die alle Regeln erfüllen :-)
- Eine solche allgemeinste Typisierung können wir in (fast) linearer Zeit berechnen :-)

Achtung:

In der berechneten Typisierung können Typ-Variablen vorkommen !!!

Beispiel:

$$e \equiv \mathbf{fn}\ (f, x) \Rightarrow f\ x$$

Mit $\alpha \equiv \alpha[x]$ und $\beta \equiv \tau[f\ x]$ finden wir:

$$\begin{array}{lll} \alpha[f] & = & \alpha \to \beta \\ \tau[e] & = & (\alpha \to \beta, \alpha) \to \beta \end{array}$$

Diskussion:

- Die Typ-Variablen bedeuten offenbar, dass die Funktionsdefinition für jede mögliche Instantiierung funktioniert $\Longrightarrow$ Polymorphie

  Wir kommen darauf zurück :-)
- Das bisherige Verfahren, um Typisierungen zu berechnen, hat den Nachteil, dass es nicht syntax-gerichtet ist ...
- Wenn das Gleichungssystem zu einem Programm keine Lösung besitzt, erhalten wir keine Information, wo der Fehler stecken könnte :-(

$\Longrightarrow$ Wir benötigen ein syntax-gerichtetes Verfahren !!!

... auch wenn es möglicherweise ineffizienter ist :-)

Der Algorithmus $\mathcal{W}$:

$$
\begin{array}{lll}
\textbf{fun } \mathcal{W}\ e\ (\Gamma, \theta) & = & \textbf{case } e \\
\quad \textbf{of } \ c & \to & (t_c, \theta) \\
\quad \mid \ \ [\,] & \to & \textbf{let val } \alpha = \mathsf{new}() \\
 & & \textbf{in } (\mathsf{list}\ \alpha, \theta) \\
 & & \mathsf{end} \\
\quad \mid \ \ x & \to & (\Gamma(x), \theta) \\
\quad \mid \ \ (e_1, \ldots, e_m) & \to & \textbf{let val } (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
 & & \qquad\qquad \ldots \\
 & & \qquad \textbf{val } (t_m, \theta) = \mathcal{W}\ e_m\ (\Gamma, \theta) \\
 & & \textbf{in } ((t_1, \ldots, t_m), \theta) \\
 & & \textbf{end} \\
 & \ldots &
\end{array}
$$

Der Algorithmus $\mathcal{W}$ (Forts.):

$$
\begin{array}{llll}
| & (e_1 : e_2) & \to & \textbf{let}\ \ \textbf{val}\ \ (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
 & & & \qquad \textbf{val}\ \ (t_2, \theta) = \mathcal{W}\ e_2\ (\Gamma, \theta) \\
 & & & \qquad \textbf{val}\ \ \theta = \boxed{\mathsf{unify}\ (\mathsf{list}\ t_1, t_2)\ \theta} \\
 & & & \textbf{in}\ \ (t_2, \theta) \\
 & & & \textbf{end} \\
| & (e_1\ e_2) & \to & \textbf{let}\ \ \textbf{val}\ \ (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
 & & & \qquad \textbf{val}\ \ (t_2, \theta) = \mathcal{W}\ e_2\ (\Gamma, \theta) \\
 & & & \qquad \textbf{val}\ \ \alpha = \mathsf{new}\,() \\
 & & & \qquad \textbf{val}\ \ \theta = \boxed{\mathsf{unify}\ (t_1, t_2 \to \alpha)\ \theta} \\
 & & & \textbf{in}\ \ (\alpha, \theta) \\
 & & & \textbf{end} \\
 & & \ldots &
\end{array}
$$

Der Algorithmus $\mathcal{W}$ (Forts.):

$$
\begin{array}{lll}
\mid \;(\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2) & \to & \textbf{let}\ \ \textbf{val}\ \ (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
 & & \qquad \textbf{val}\ \ \theta = \boxed{\mathsf{unify}\ (\textbf{bool}, t_0)\ \theta} \\
 & & \qquad \textbf{val}\ \ (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
 & & \qquad \textbf{val}\ \ (t_2, \theta) = \mathcal{W}\ e_2\ (\Gamma, \theta) \\
 & & \qquad \textbf{val}\ \ \theta = \boxed{\mathsf{unify}\ (t_1, t_2)\ \theta} \\
 & & \textbf{in}\ \ (t_1, \theta) \\
 & & \textbf{end}
\end{array}
$$

$\ldots$

Der Algorithmus $\mathcal{W}$ (Forts.):

$$
\begin{array}{lll}
| & (\textbf{case } e_0 \textbf{ of } [\,] \to e_1 \;;\; (x:y) \to e_2) & \\
& \quad \to \quad \textbf{let } \textbf{val } (t_0, \theta) = \mathcal{W}\; e_0\; (\Gamma, \theta) & \\
& \qquad\qquad \textbf{val } \alpha = \mathsf{new}() & \\
& \qquad\qquad \textbf{val } \theta = \boxed{\mathsf{unify}\; (\mathsf{list}\; \alpha, t_0)\; \theta} & \\
& \qquad\qquad \textbf{val } (t_1, \theta) = \mathcal{W}\; e_1\; (\Gamma, \theta) & \\
& \qquad\qquad \textbf{val } (t_2, \theta) = \mathcal{W}\; e_2\; (\Gamma \oplus \{x \mapsto \alpha, y \mapsto \mathsf{list}\; \alpha\}, \theta) & \\
& \qquad\qquad \textbf{val } \theta = \boxed{\mathsf{unify}\; (t_1, t_2)\; \theta} & \\
& \qquad\quad \textbf{in } (t_1, \theta) & \\
& \qquad\quad \textbf{end} & \\
& \ldots &
\end{array}
$$

Der Algorithmus $\mathcal{W}$ (Forts.):

$$
\begin{array}{lll}
| & \textbf{fn}\ (x_1,\ldots,x_m)\ \Rightarrow\ e & \\
 & \quad \rightarrow & \textbf{let}\ \ \textbf{val}\ \ \alpha_1 = \mathsf{new}() \\
 & & \qquad\qquad \ldots \\
 & & \qquad \textbf{val}\ \ \alpha_m = \mathsf{new}() \\
 & & \qquad \textbf{val}\ \ (t,\theta) = \mathcal{W}\ e\ (\Gamma \oplus \{x_1 \mapsto \alpha_1,\ldots,x_m \mapsto \alpha_m\},\theta) \\
 & & \textbf{in}\ \ ((\alpha_1,\ldots,\alpha_m) \rightarrow t,\theta) \\
 & & \textbf{end} \\
 & \ldots &
\end{array}
$$

Der Algorithmus $\mathcal{W}$ (Forts.):

$$
\begin{array}{ll}
| & (\textbf{letrec } x_1 = e_1; \ldots; x_m = e_m \textbf{ in } e_0) \\
& \quad \to \quad \textbf{let val } \alpha_1 = \mathsf{new}() \\
& \qquad\qquad\qquad \ldots \\
& \qquad\qquad \textbf{val } \alpha_m = \mathsf{new}() \\
& \qquad\qquad \textbf{val } \Gamma = \Gamma \oplus \{x_1 \mapsto \alpha_1, \ldots, x_m \mapsto \alpha_m\} \\
& \qquad\qquad \textbf{val } (t_1, \theta) = \mathcal{W}\; e_1\; (\Gamma, \theta) \\
& \qquad\qquad \textbf{val } \theta = \boxed{\mathsf{unify}\; (\alpha_1, t_1)\; \theta} \\
& \qquad\qquad\qquad \ldots \\
& \qquad\qquad \textbf{val } (t_m, \theta) = \mathcal{W}\; e_m\; (\Gamma, \theta) \\
& \qquad\qquad \textbf{val } \theta = \boxed{\mathsf{unify}\; (\alpha_m, t_m)\; \theta} \\
& \qquad\qquad \textbf{val } (t_0, \theta) = \mathcal{W}\; e_0\; (\Gamma, \theta) \\
& \qquad\quad \textbf{in } (t_0, \theta) \\
& \qquad\quad \textbf{end}
\end{array}
$$

$\ldots$

Der Algorithmus $\mathcal{W}$ (Forts.):

$$
\begin{array}{ll}
| & (\textbf{let}\ x_1 = e_1; \ldots ; x_m = e_m\ \textbf{in}\ e_0) \\
& \quad \rightarrow \quad \textbf{let}\ \ \textbf{val}\ \ (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
& \qquad\qquad\qquad \textbf{val}\ \ \Gamma = \Gamma \oplus \{x_1 \mapsto t_1\} \\
& \qquad\qquad\qquad\qquad \ldots \\
& \qquad\qquad\qquad \textbf{val}\ \ (t_m, \theta) = \mathcal{W}\ e_m\ (\Gamma, \theta) \\
& \qquad\qquad\qquad \textbf{val}\ \ \Gamma = \Gamma \oplus \{x_m \mapsto t_m\} \\
& \qquad\qquad\qquad \textbf{val}\ \ (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
& \qquad\qquad \textbf{in}\ \ (t_0, \theta) \\
& \qquad\qquad \textbf{end} \\
& \ldots
\end{array}
$$

Bemerkungen:

- Am Anfang ist $\Gamma = \emptyset$ und $\theta = \emptyset$:-)
- Der Algorithmus unifiziert nach und nach die Typ-Gleichungen :-)
- Der Algorithmus liefert bei jedem Aufruf einen Typ t zusammen mit einer Substitution θ zurück.
- Der inferierte allgemeinste Typ ergibt sich als $\theta(t)$.
- Die Hilfsfunktion new() liefert jeweils eine neue Typvariable :-)
- Bei jedem Aufruf von unify() kann die Typinferenz fehlschlagen ...
- Bei Fehlschlag sollte die Stelle, wo der Fehler auftrat gemeldet werden, die Typ-Inferenz aber mit plausiblen Werten fortgesetzt werden :-}

Beispiel:

$$
\begin{array}{llll}
\textbf{let} & \mathsf{dup} & = & \textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\,(f\,x); \\
 & \mathsf{inc} & = & \textbf{fn}\ y \Rightarrow y+1; \\
 & \mathsf{single} & = & \textbf{fn}\ y \Rightarrow y : [\,] \\
\multicolumn{4}{l}{\textbf{in}\ \ \mathsf{dup}\ \mathsf{single}\ (\mathsf{dup}\ \mathsf{inc}\ 1)} \\
\multicolumn{4}{l}{\textbf{end}}
\end{array}
$$

Beispiel:

$$\begin{array}{lllllll}
\textbf{let} & \textsf{dup} & = & \textbf{fn}\ f & \Rightarrow & \textbf{fn}\ x\ \Rightarrow & f\,(f\,x); \\
 & \textsf{inc} & = & \textbf{fn}\ y & \Rightarrow & y+1; & \\
 & \textsf{single} & = & \textbf{fn}\ y & \Rightarrow & y : [\,] & \\
\textbf{in} & \multicolumn{6}{l}{\textsf{dup single}\ (\textsf{dup inc}\ 1)} \\
\textbf{end} & & & & & &
\end{array}$$

Wir finden:

$$\begin{array}{lll}
\alpha[\textsf{dup}] & = & (\alpha \to \beta) \to \alpha \to \beta \\
\alpha[\textsf{inc}] & = & \textbf{int} \to \textbf{int} \\
\alpha[\textsf{single}] & = & \gamma \to \textsf{list}\ \gamma
\end{array}$$

- Durch die Anwendung: dup single erhalten wir:

$$\begin{array}{lcl} \alpha & = & \gamma \\ \beta & = & \mathsf{list}\ \gamma \\ \alpha[\mathsf{dup}] & = & (\gamma \to \mathsf{list}\ \gamma) \to \gamma \to \mathsf{list}\ \gamma \end{array}$$

- Durch die Anwendung: dup inc erhalten wir:

$$\begin{array}{lcl} \alpha & = & \mathbf{int} \\ \beta & = & \mathbf{int} \\ \alpha[\mathsf{dup}] & = & (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int} \end{array}$$

$\Longrightarrow$ Typ-Fehler ???

Idee 1: Kopiere jede Definition für jede Benutzung ...

... im Beispiel:

$$
\begin{array}{llll}
\textbf{let} & \mathsf{dup} & = & \textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\,(f\,x); \\
 & \mathsf{inc} & = & \textbf{fn}\ y \Rightarrow y+1; \\
 & \mathsf{single} & = & \textbf{fn}\ y \Rightarrow y : [\,] \\
\textbf{in} & \multicolumn{3}{l}{(\textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\,(f\,x))\ \mathsf{single}} \\
 & \multicolumn{3}{l}{((\textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\,(f\,x))\ \mathsf{inc}\ 1)} \\
\textbf{end} & & &
\end{array}
$$

Idee 1: Kopiere jede Definition für jede Benutzung ...

... im Beispiel:

$$
\begin{array}{llll}
\textbf{let} & \mathsf{dup} & = & \textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\,(f\,x); \\
 & \mathsf{inc} & = & \textbf{fn}\ y \Rightarrow y + 1; \\
 & \mathsf{single} & = & \textbf{fn}\ y \Rightarrow y : [\,] \\
\textbf{in} & \multicolumn{3}{l}{(\textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\,(f\,x))\ \mathsf{single}} \\
 & \multicolumn{3}{l}{((\textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\,(f\,x))\ \mathsf{inc}\ 1)} \\
\textbf{end} & & &
\end{array}
$$

+ Die beiden Teilausdrücke $(\textbf{fn}\ f \Rightarrow \textbf{fn}\ x \Rightarrow f\,(f\,x))$ erhalten jeweils einen eigenen Typ mit unabhängigen Typ-Variablen :-)
+ Das expandierte Programm ist typbar :-))
− Das expandierte Programm kann seeehr groß werden :-(
− Typ-Checking ist nicht mehr modular :-((

Idee 2: Kopiere die Typen für jede Benutzung ...

- Wir erweitern Typen zu Typ-Schemata:

$$
\begin{array}{lll}
t & ::= & \alpha \mid \mathbf{bool} \mid \mathbf{int} \mid (t_1, \ldots, t_m) \mid \mathsf{list}\ t \mid t_1 \to t_2 \\
\sigma & ::= & t \mid \forall\, \alpha_1, \ldots, \alpha_k . t
\end{array}
$$

- Achtung: Der Operator $\forall$ erscheint nur auf dem Top-Level !!!
- Typ-Schemata werden für **let**-definierte Variablen eingeführt.
- Bei deren Benutzung wird der Typ im Schema mit frischen Typ-Variablen instantiiert ...

Neue Regeln:

$$\text{Inst:} \quad \frac{\Gamma(x) = \forall\, \alpha_1, \ldots, \alpha_k . t}{\Gamma \;\vdash\; x \;:\; t[t_1/\alpha_1, \ldots, t_k/\alpha_k]} \qquad (t_1, \ldots, t_k \quad \text{beliebig})$$

$$\text{Let:} \quad \frac{\begin{array}{lllll} \Gamma_0 & \vdash & e_1 & : & t_1 \qquad \Gamma_1 = \Gamma_0 \oplus \{x_1 \mapsto \mathsf{close}\; t_1\; \Gamma_0\} \\ & \ldots & & & \qquad\qquad \ldots \\ \Gamma_{m-1} & \vdash & e_m & : & t_m \qquad \Gamma_m = \Gamma_{m-1} \oplus \{x_m \mapsto \mathsf{close}\; t_m\; \Gamma_{m-1}\} \\ \Gamma_m & \vdash & e_0 & : & t_0 \end{array}}{\Gamma_0 \;\vdash\; (\mathbf{let}\; x_1 = e_1; \ldots; x_m = e_m\; \mathbf{in}\; e_0) \;:\; t_0}$$

Der Aufruf $\mathsf{close}\ t\ \Gamma$ macht alle Typ-Variablen in t generisch (d.h. instantiierbar), die nicht auch in Γ vorkommen ...

$$
\begin{array}{l}
\textbf{fun}\ \ \mathsf{close}\ t\ \Gamma \quad = \quad \textbf{let} \\
\qquad\qquad \textbf{val}\ \ \alpha_1, \ldots, \alpha_k = \mathsf{free}\,(t) \setminus \mathsf{free}\,(\Gamma) \\
\qquad\qquad \textbf{in}\ \ \forall\, \alpha_1, \ldots, \alpha_k.\, t \\
\qquad\qquad \textbf{end}
\end{array}
$$

Eine Instantiierung mit frischen Typ-Variablen leistet die Funktion:

$$
\begin{array}{l}
\textbf{fun}\ \ \mathsf{inst}\ \sigma \quad = \quad \textbf{let} \\
\qquad\qquad \textbf{val}\ \ \forall\, \alpha_1, \ldots, \alpha_k.\, t = \sigma \\
\qquad\qquad \textbf{val}\ \ \beta_1 = \mathsf{new}() \quad \ldots \quad \textbf{val}\ \ \beta_k = \mathsf{new}() \\
\qquad\qquad \textbf{in}\ \ t[\beta_1/\alpha_1, \ldots, \beta_k/\alpha_k] \\
\qquad\qquad \textbf{end}
\end{array}
$$

Der Algorithmus $\mathcal{W}$ (erweitert):

$$
\begin{array}{lllll}
& \ldots & & & \\
\mid & x & \to & \mathsf{inst}\ (\Gamma(x)) & \\
\mid & (\mathbf{let}\ x_1 = e_1; \ldots; x_m = e_m\ \mathbf{in}\ e_0) & & & \\
& & \to & \mathbf{let}\ \ \mathbf{val} & (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
& & & \phantom{\mathbf{let}}\ \ \mathbf{val} & \sigma_1 = \mathsf{close}\ (\theta\ t_1)\ (\theta\ \Gamma) \\
& & & \phantom{\mathbf{let}}\ \ \mathbf{val} & \Gamma = \Gamma \oplus \{x_1 \mapsto \sigma_1\} \\
& & & & \quad\ldots \\
& & & \phantom{\mathbf{let}}\ \ \mathbf{val} & (t_m, \theta) = \mathcal{W}\ e_m\ (\Gamma, \theta) \\
& & & \phantom{\mathbf{let}}\ \ \mathbf{val} & \sigma_m = \mathsf{close}\ (\theta\ t_m)\ (\theta\ \Gamma) \\
& & & \phantom{\mathbf{let}}\ \ \mathbf{val} & \Gamma = \Gamma \oplus \{x_m \mapsto \sigma_m\} \\
& & & \phantom{\mathbf{let}}\ \ \mathbf{val} & (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
& & \mathbf{in}\ \ (t_0, \theta) & & \\
& & \mathbf{end} & &
\end{array}
$$

Beispiel:

$$\begin{array}{llllllll}
\textbf{let} & \mathsf{dup} & = & \textbf{fn}\ f & \Rightarrow & \textbf{fn}\ x & \Rightarrow & f\,(f\,x); \\
 & \mathsf{inc} & = & \textbf{fn}\ y & \Rightarrow & y+1; & & \\
 & \mathsf{single} & = & \textbf{fn}\ y & \Rightarrow & y : [\,] & & \\
\textbf{in} & \multicolumn{7}{l}{\mathsf{dup\ single}\ (\mathsf{dup\ inc}\ 1)} \\
\textbf{end} & & & & & & &
\end{array}$$

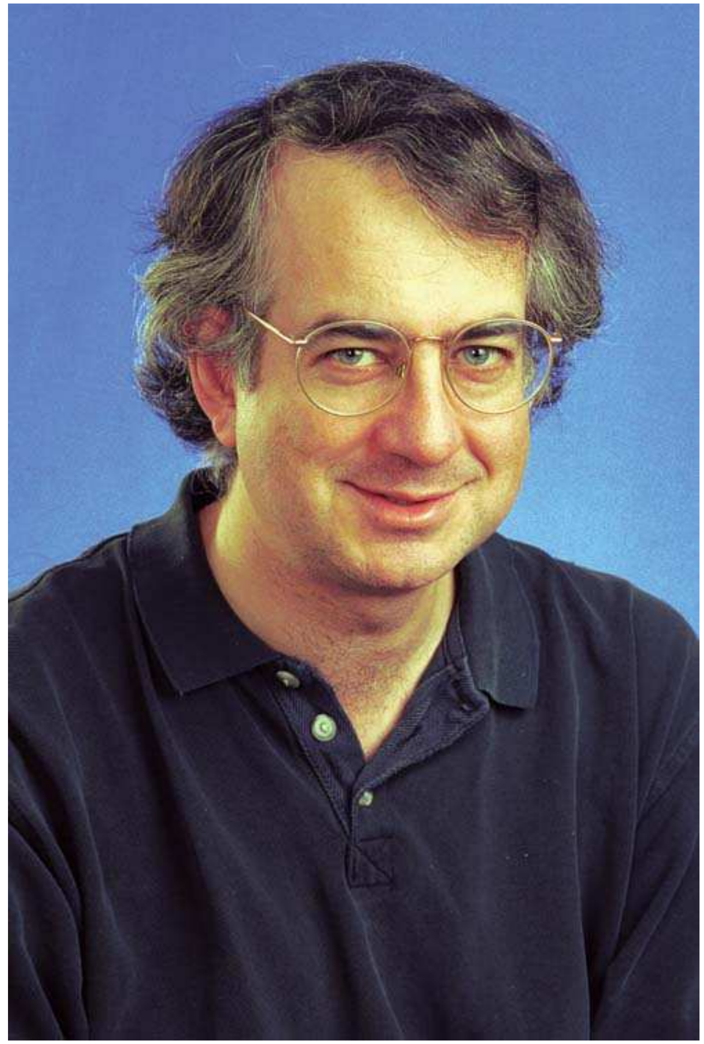
Wir finden:

$$\begin{array}{lll}
\alpha[\mathsf{dup}] & = & \forall\,\alpha,\beta.\,(\alpha\to\beta)\to\alpha\to\beta \\
\alpha[\mathsf{inc}] & = & \textbf{int}\to\textbf{int} \\
\alpha[\mathsf{single}] & = & \forall\,\gamma.\,\gamma\to\mathsf{list}\ \gamma
\end{array}$$

Bemerkungen:

- Der erweiterte Algorithmus berechnet nach wie vor allgemeinste Typen :-)
- Instantiierung von Typ-Schemata bei jeder Benutzung ermöglicht polymorphe Funktionen sowie modulare Typ-Inferenz :-))
- Die Möglichkeit der Instantiierung erlaubt die Codierung von DEXPTIME-schwierigen Problemen in die Typ-Inferenz ??

  ... ein in der Praxis eher marginales Problem :-)
- Die Einführung von Typ-Schemata ist nur für nicht-rekursive Definitionen möglich: die Ermittlung eines allgemeinsten Typ-Schemas für rekursive Definitionen ist nicht berechenbar !!!



Harry Mairson, Brandeis University

Seiteneffekte

- Für ein elegantes Programmieren sind gelegentlich Variablen, deren Wert geändert werden kann, ganz nützlich :-)
- Darum erweitern wir unsere kleine Programmiersprache um Referenzen:

$$e \; ::= \; \ldots \; | \; \mathbf{ref}\; e \; | \; !e \; | \; e_1 := e_2$$

Seiteneffekte

- Für ein elegantes Programmieren sind gelegentlich Variablen, deren Wert geändert werden kann, ganz nützlich :-)
- Darum erweitern wir unsere kleine Programmiersprache um Referenzen:

$$e \;::=\; \ldots \;|\; \mathbf{ref}\; e \;|\; !e \;|\; e_1 := e_2$$

Beispiel:

$$
\begin{array}{lll}
\mathbf{let}\ \mathsf{count} & = & \mathbf{ref}\ 0; \\
\quad\ \mathsf{new} & = & \mathbf{fn}\ () \Rightarrow \mathbf{let} \\
 & & \qquad\qquad \mathsf{ret} \;=\; !\mathsf{count}; \\
 & & \qquad\qquad \_ \;\;\;\;=\; \mathsf{count} := \mathsf{ret} + 1 \\
 & & \qquad\quad\ \mathbf{in}\ \mathsf{ret} \\
\mathbf{in}\ \mathsf{new}() + \mathsf{new}() & &
\end{array}
$$

Als neuen Typ benötigen wir:

$$t \ ::= \ \ldots \ \textbf{ref}\ t \ \ldots$$

Neue Regeln:

Ref:
$$\frac{\Gamma \ \vdash \ e \ : \ t}{\Gamma \ \vdash \ (\textbf{ref}\ e) \ : \ \textbf{ref}\ t}$$

Deref:
$$\frac{\Gamma \ \vdash \ e \ : \ \textbf{ref}\ t}{\Gamma \ \vdash \ (!\, e) \ : \ t}$$

Assign:
$$\frac{\Gamma \ \vdash \ e_1 \ : \ \textbf{ref}\ t \qquad \Gamma \ \vdash \ e_2 \ : \ t}{\Gamma \ \vdash \ (e_1 := e_2) \ : \ ()}$$

Achtung:

Diese Regeln vertragen sich nicht mit Polymorphie !!!

Beispiel:

$$\begin{array}{lll} \textbf{let } y & = & \textbf{ref } [\,]; \\ \quad\_ & = & y \; := \; 1 : (!\, y); \\ \quad\_ & = & y \; := \; \textbf{true} : (!\, y) \\ \textbf{in } 1 & & \end{array}$$

Achtung:

Diese Regeln vertragen sich nicht mit Polymorphie !!!

Beispiel:

$$
\begin{array}{lllll}
\textbf{let } y & = & \textbf{ref}\, [\,]; & & \\
\quad\; \_ & = & y & := & 1 : (!\, y); \\
\quad\; \_ & = & y & := & \textbf{true} : (!\, y) \\
\textbf{in } 1 & & & &
\end{array}
$$

Für y erhalten wir den Typ: $\forall\, \alpha.\, \textbf{ref}\, (\text{list}\ \alpha)$

$\Longrightarrow$ Die Typ-Inferenz liefert keinen Fehler

$\Longrightarrow$ Zur Laufzeit entsteht eine Liste mit **int** und **bool** :-(

Ausweg: Die Value-Restriction

- Generalisiere nur solche Typen, die Werte repräsentieren, d.h. keine Verweise auf Speicherstellen enthalten :-)
- Die Menge der Value-Typen lässt sich einfach beschreiben:

$$v ::= \textbf{bool} \mid \textbf{int} \mid \textsf{list}\ v \mid (v_1, \ldots, v_m) \mid t \to t$$

Ausweg: Die Value-Restriction

- Generalisiere nur solche Typen, die Werte repräsentieren, d.h. keine Verweise auf Speicherstellen enthalten :-)
- Die Menge der Value-Typen lässt sich einfach beschreiben:

$$v \; ::= \; \mathbf{bool} \mid \mathbf{int} \mid \text{list } v \mid (v_1, \ldots, v_m) \mid t \to t$$

... im Beispiel:

Der Typ: $\mathbf{ref}\ (\text{list } \alpha)$ ist kein Value-Typ.

Darum darf er nicht generalisiert werden $\Longrightarrow$ Problem gelöst :-)

Matthias Felleisen, Northeastern University

Schlussbemerkung:

- Polymorphie ist ein sehr nützliches Hilfsmittel bei der Programmierung :-)
- In Form von Templates hält es in Java 1.5 Einzug.
- In der Programmiersprache Haskell hat man Polymorphie in Richtung bedingter Polymorphie weiter entwickelt ...

Schlussbemerkung:

- Polymorphie ist ein sehr nützliches Hilfsmittel bei der Programmierung :-)
- In Form von Templates hält es in Java 1.5 Einzug.
- In der Programmiersprache Haskell hat man Polymorphie in Richtung bedingter Polymorphie weiter entwickelt ...

Beispiel:

$$
\begin{array}{lcl}
\textbf{fun}\ \ \mathsf{member}\ x\ \mathsf{list} & = & \textbf{case}\ \ \mathsf{list} \\
 & & \begin{array}{lll}
\textbf{of} & [\,] & \to\ \ \textbf{false} \\
\mid & h{::}t & \to\ \ \textbf{if}\ \ x = h\ \ \textbf{then}\ \ \textbf{true} \\
 & & \ \ \ \textbf{else}\ \ \mathsf{member}\ x\ t
\end{array}
\end{array}
$$

Schlussbemerkung:

- Polymorphie ist ein sehr nützliches Hilfsmittel bei der Programmierung :-)
- In Form von Templates hält es in Java 1.5 Einzug.
- In der Programmiersprache Haskell hat man Polymorphie in Richtung bedingter Polymorphie weiter entwickelt ...

Beispiel:

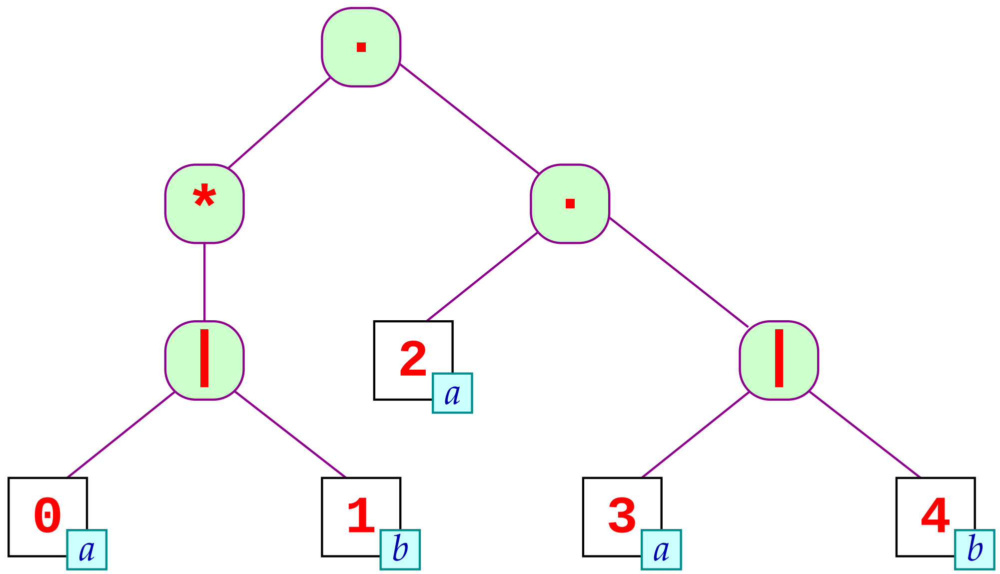
$$\begin{array}{lllll} \textbf{fun} \;\; \mathsf{member}\; x \;\mathsf{list} & = & \textbf{case} \;\; \mathsf{list} & & \\ & & \textbf{of} & [\,] & \to & \textbf{false} \\ & & | & h{::}t & \to & \textbf{if} \;\; x = h \;\; \textbf{then} \;\; \textbf{true} \\ & & & & & \textbf{else} \;\; \mathsf{member}\; x\; t \end{array}$$

member hat den Typ: $\alpha' \to \mathsf{list}\ \alpha' \to \textbf{bool}$ für jedes α' mit Gleichheit !!

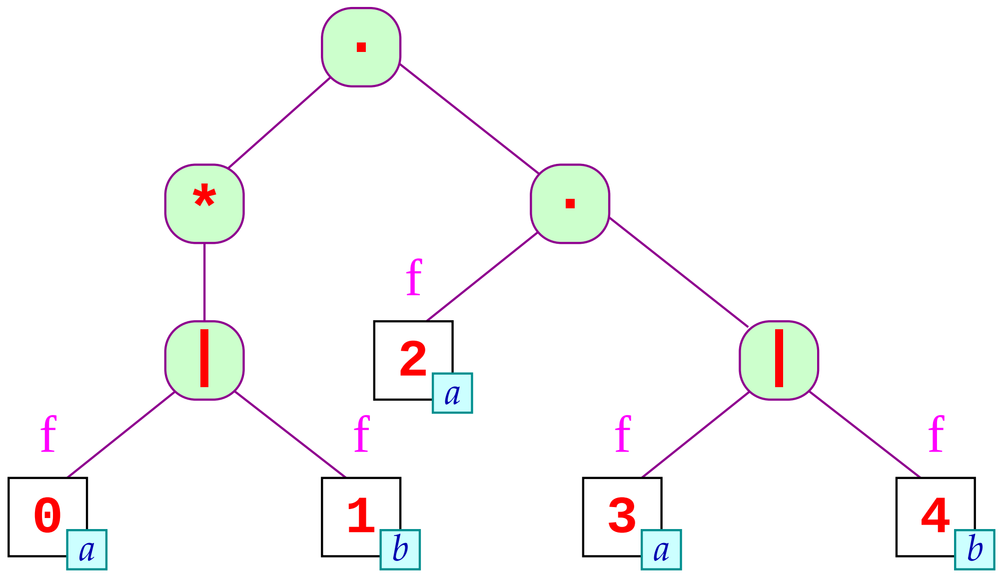
3.4 Attributierte Grammatiken

- Viele Berechnungen der semantischen Analyse wie während der Code-Generierung arbeiten auf dem Syntaxbaum.
- An jedem Knoten greifen sie auf bereits berechnete Informationen zu und berechnen daraus neue Informationen :-)
- Was lokal zu tun ist, hängt nur von der Sorte des Knotens ab !!!
- Damit die zu lesenden Werte an jedem Knoten bei jedem Lesen bereits vorliegen, müssen die Knoten des Syntaxbaums in einer bestimmten Reihenfolge durchlaufen werden ...

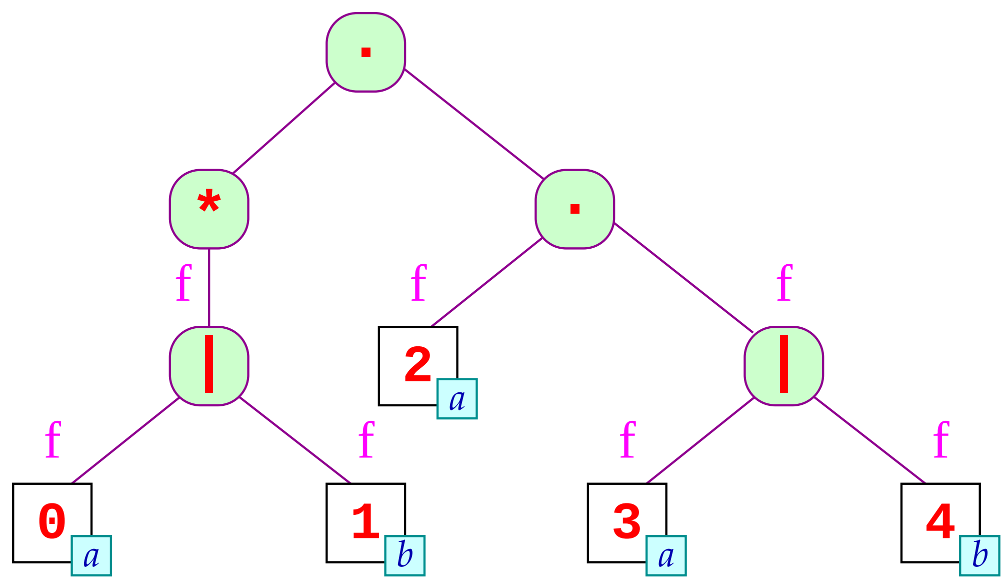
Beispiel: Berechnung des Prädikats $\text{empty}[r]$



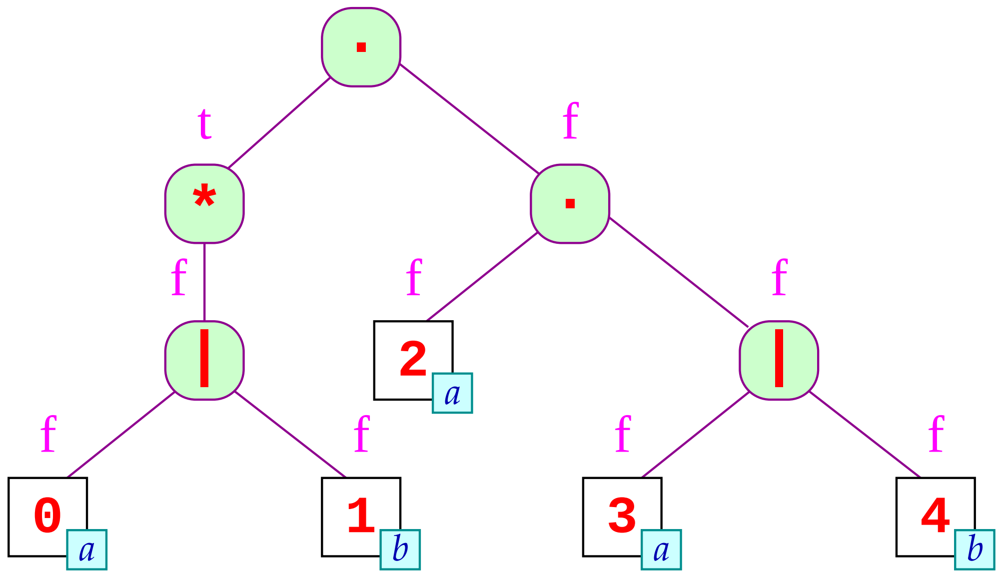
Beispiel: Berechnung des Prädikats empty$[r]$



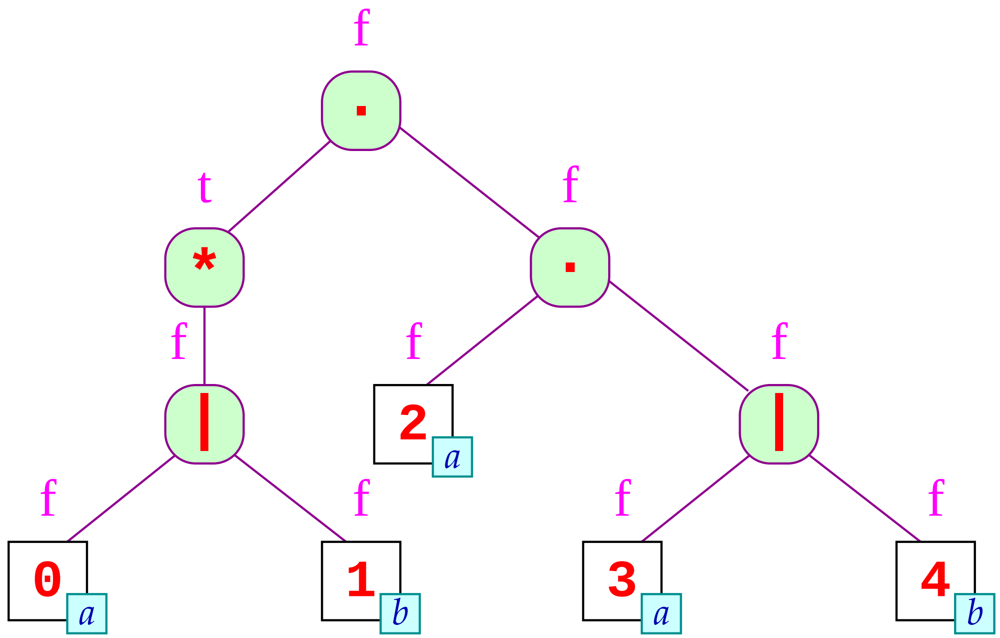
Beispiel: Berechnung des Prädikats $\text{empty}[r]$



Beispiel: Berechnung des Prädikats $\text{empty}[r]$



Beispiel: Berechnung des Prädikats $\mathsf{empty}[r]$



Idee zur Implementierung:

- Für jeden Knoten führen wir ein Attribut `empty` ein.
- Die Attribute werden in einer DFS post-order Traversierung berechnet:
  - → An einem Blatt lässt sich der Wert des Attributs unmittelbar ermitteln ;-)
  - → Das Attribut an einem inneren Knoten hängt darum nur von den Attributen der Nachfolger ab :-)
- Wie das Attribut lokal zu berechnen ist, ergibt sich aus dem Typ des Knotens ...

Für Blätter $r \equiv \boxed{i \mid x}$ ist $\text{empty}[r] = (x \equiv \epsilon)$.

Andernfalls:

$$
\begin{array}{lcl}
\text{empty}[r_1 \mid r_2] & = & \text{empty}[r_1] \vee \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] & = & \text{empty}[r_1] \wedge \text{empty}[r_2] \\
\text{empty}[r_1^*] & = & t \\
\text{empty}[r_1?] & = & t
\end{array}
$$

Diskussion:

- Wir benötigen einen einfachen und flexiblen Mechanismus, mit dem wir über die Attribute an einem Knoten und seinen Nachfolgern reden können.
- Der Einfachheit geben wir ihnen einen fortlaufenden Index:

  $\text{empty}[0]$: das Attribut des Vater-Knotens

  $\text{empty}[i]$: das Attribut des i-ten Sohns $(i > 0)$

Diskussion:

- Wir benötigen einen einfachen und flexiblen Mechanismus, mit dem wir über die Attribute an einem Knoten und seinen Nachfolgern reden können.
- Der Einfachkeit geben wir ihnen einen fortlaufenden Index:

$\text{empty}[0]$: das Attribut des Vater-Knotens

$\text{empty}[i]$: das Attribut des i-ten Sohns $(i > 0)$

... im Beispiel:

$$
\begin{array}{lcll}
\boxed{x} & : & \text{empty}[0] & := (x \equiv \epsilon) \\
\boxed{|} & : & \text{empty}[0] & := \text{empty}[1] \vee \text{empty}[2] \\
\boxed{\cdot} & : & \text{empty}[0] & := \text{empty}[1] \wedge \text{empty}[2] \\
\boxed{*} & : & \text{empty}[0] & := t \\
\boxed{?} & : & \text{empty}[0] & := t
\end{array}
$$

Diskussion:

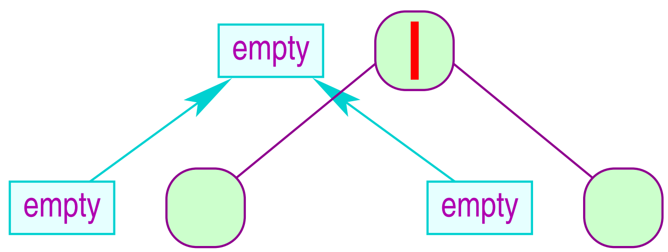
- Die lokalen Berechnungen der Attributwerte müssen zu einem globalen Algorithmus zusammen gesetzt werden :-)
- Dazu benötigen wir:

  (1) eine Besuchsreihenfolge der Knoten des Baums;

  (2) lokale Berechnungsreihenfolgen ...

- Die Auswertungsstrategie sollte aber mit den Attribut-Abhängigkeiten kompatibel sein :-)

... im Beispiel:



Achtung:

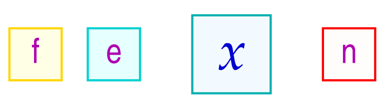
- Zur Ermittlung einer Auswertungsstrategie reicht es nicht, sich die lokalen Attribut-Abhängigkeiten anzusehen.
- Es kommt auch darauf an, wie sie sich global zu einem Abhängigkeitsgraphen zusammen setzen !!!
- Im Beispiel sind die Abhängigkeiten stets von den Attributen der Söhne zu den Attributen des Vaters gerichtet.

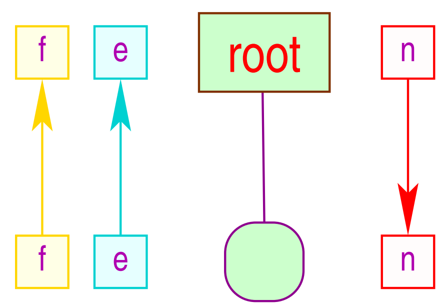
  $\Longrightarrow$ Postorder-DFS-Traversierung
- Die Variablen-Abhängigkeiten können aber auch komplizierter sein ...

Beispiel: Simultane Berechnung von empty, first, next :

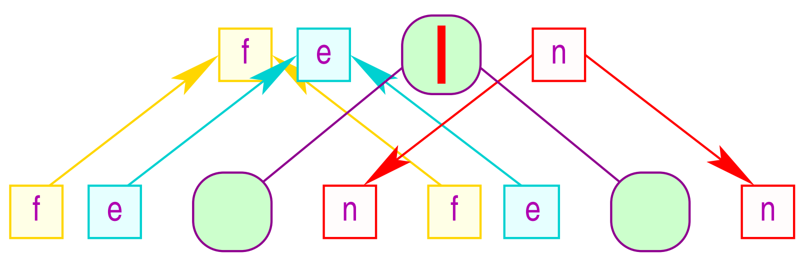
$$\begin{array}{lll}
\boxed{x} \quad : & \text{empty}[0] & := (x \equiv \epsilon) \\
 & \text{first}[0] & := \{x \mid x \neq \epsilon\} \\
 & & // \quad \text{(keine Gleichung für next !!!)}
\end{array}$$

$$\begin{array}{lll}
\boxed{\text{root:}} \quad : & \text{empty}[0] & := \text{empty}[1] \\
 & \text{first}[0] & := \text{first}[1] \\
 & \text{next}[0] & := \emptyset \\
 & \text{next}[1] & := \text{next}[0]
\end{array}$$

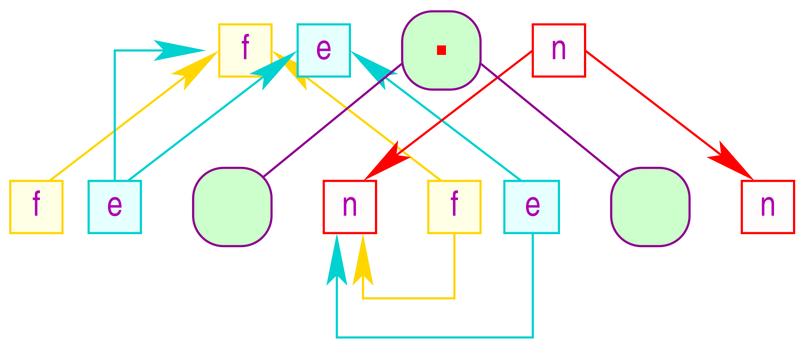




$$\boxed{\,|\,} \quad : \quad \begin{array}{lcl} \mathsf{empty}[0] & := & \mathsf{empty}[1] \vee \mathsf{empty}[2] \\ \mathsf{first}[0] & := & \mathsf{first}[1] \cup \mathsf{first}[2] \\ \mathsf{next}[1] & := & \mathsf{next}[0] \\ \mathsf{next}[2] & := & \mathsf{next}[0] \end{array}$$

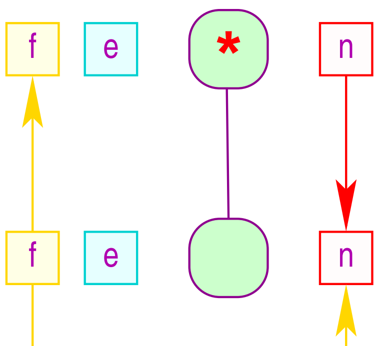


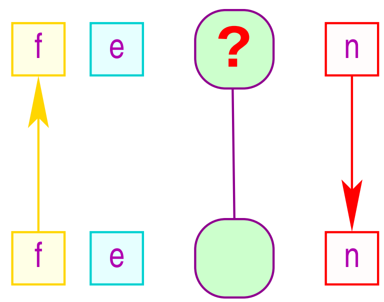
$$
\begin{array}{llll}
\boxed{\cdot} & : & \mathsf{empty}[0] & := \mathsf{empty}[1] \wedge \mathsf{empty}[2] \\
& & \mathsf{first}[0] & := \mathsf{first}[1] \cup (\mathsf{empty}[1]) \; ? \; \mathsf{first}[2] : \emptyset \\
& & \mathsf{next}[1] & := \mathsf{first}[2] \cup (\mathsf{empty}[2]) \; ? \; \mathsf{next}[0] \\
& & \mathsf{next}[2] & := \mathsf{next}[0]
\end{array}
$$



$$\boxed{*} \quad : \quad \begin{array}{lcl} \mathsf{empty}[0] & := & t \\ \mathsf{first}[0] & := & \mathsf{first}[1] \\ \mathsf{next}[1] & := & \mathsf{first}[1] \cup \mathsf{next}[0] \end{array}$$

$$\boxed{?} \quad : \quad \begin{array}{lcl} \mathsf{empty}[0] & := & t \\ \mathsf{first}[0] & := & \mathsf{first}[1] \\ \mathsf{next}[1] & := & \mathsf{next}[0] \end{array}$$





Problem:

- Eine Auswertungsstrategie kann es nur dann geben, wenn die Variablen-Abhängigkeiten in jedem attributierten Baum azyklisch sind !!!
- Es ist DEXPTIME-vollständig, herauszufinden, ob keine zyklischen Variablenabhängigkeiten vorkommen können :-(

Problem:

- Eine Auswertungsstrategie kann es nur dann geben, wenn die Variablen-Abhängigkeiten in jedem attributierten Baum azyklisch sind !!!
- Es ist DEXPTIME-vollständig, herauszufinden, ob keine zyklischen Variablenabhängigkeiten vorkommen können :-(

Ideen:

(1) Die Benutzerin soll die Strategie spezifizieren ;-)

(2) In der Praxis wirds schon nicht so schlimm kommen ;-}

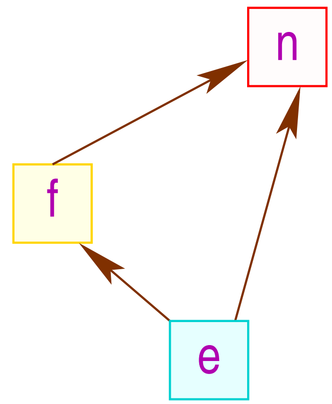
(3) Betrachte Teilklassen ...

Stark azyklische Attributierung:

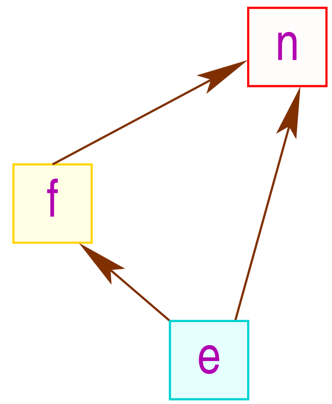
Berechne eine partielle Ordnung auf den Attributen eines Knotens, die kompatibel mit den lokalen Attribut-Abhängigkeiten ist:

- Wir starten mit der trivialen Ordung $\sqsubseteq \; = \; =$:-)
- Die aktuelle Ordnung setzen wir an den Sohn-Knoten in die lokalen Abhängigkeitsgraphen ein.
- Ergibt sich ein Kreis, geben wir auf :-))
- Andernfalls fügen wir alle Beziehungen $a \sqsubseteq b$ hinzu, für die es jetzt einen Pfad von $a[0]$ nach $b[0]$ gibt.
- Lässt sich $\sqsubseteq$ nicht mehr vergrößern, hören wir auf ...

... im Beispiel:



... im Beispiel:



Diskussion:

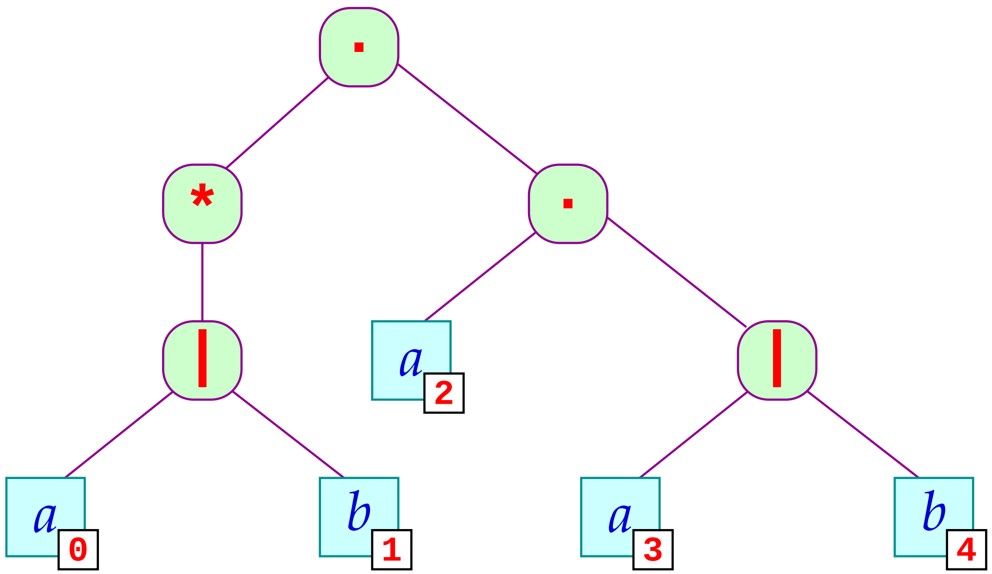
- Die Berechnung der partiellen Ordnung $\sqsubseteq$ ist eine Fixpunkt-Berechnung :-)
- Die partielle Ordnung können wir in eine lineare Ordnung einbetten ...
- Die lineare Ordnung gibt uns an, in welcher Reihenfolge die Attribute berechnet werden müssen :-)
- Die lokalen Abhängigkeitsgraphen zusammen mit der linearen Ordnung erlauben die Berechnung einer Strategie ...

Mögliche Strategien:

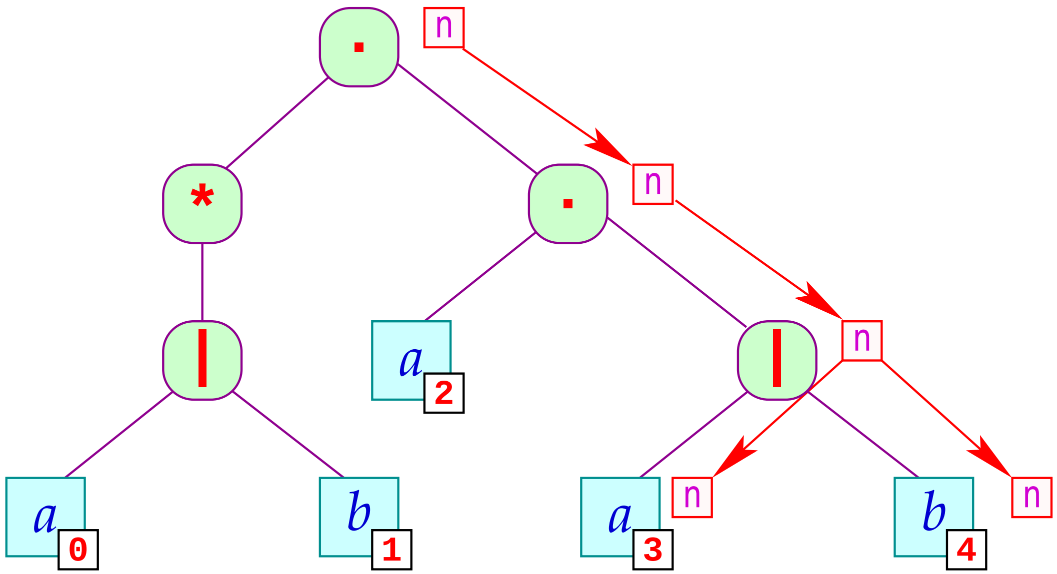
(1) **Bedarfsgetriebene Auswertung:**

- Beginne mit der Berechnung eines Attributs.
- Sind die Argument-Attribute noch nicht berechnet, berechne rekursiv deren Werte :-)
- Besuche die Knoten des Baum nach Bedarf...

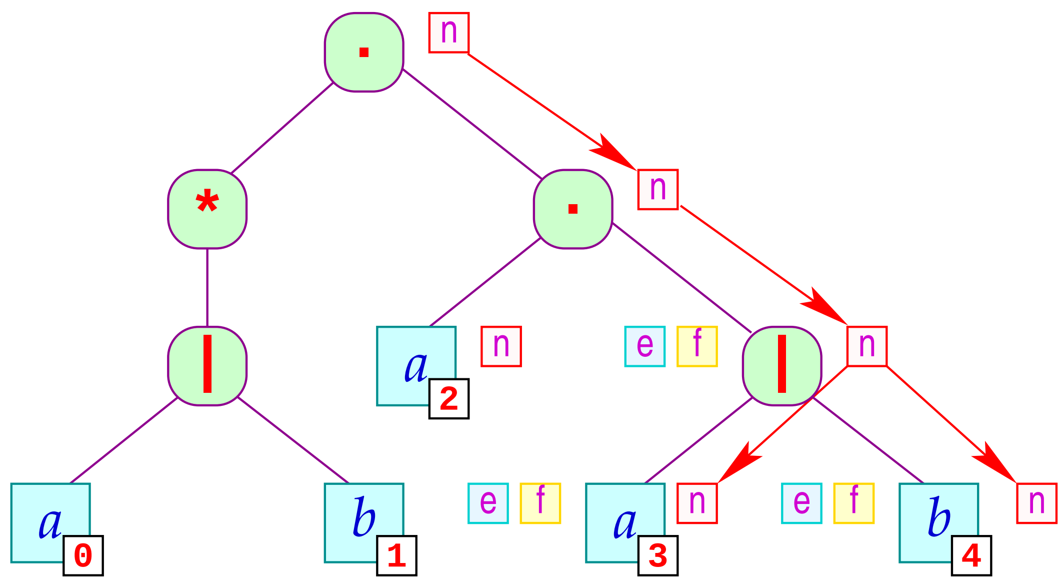
Beispiel, bedarfsgetrieben:



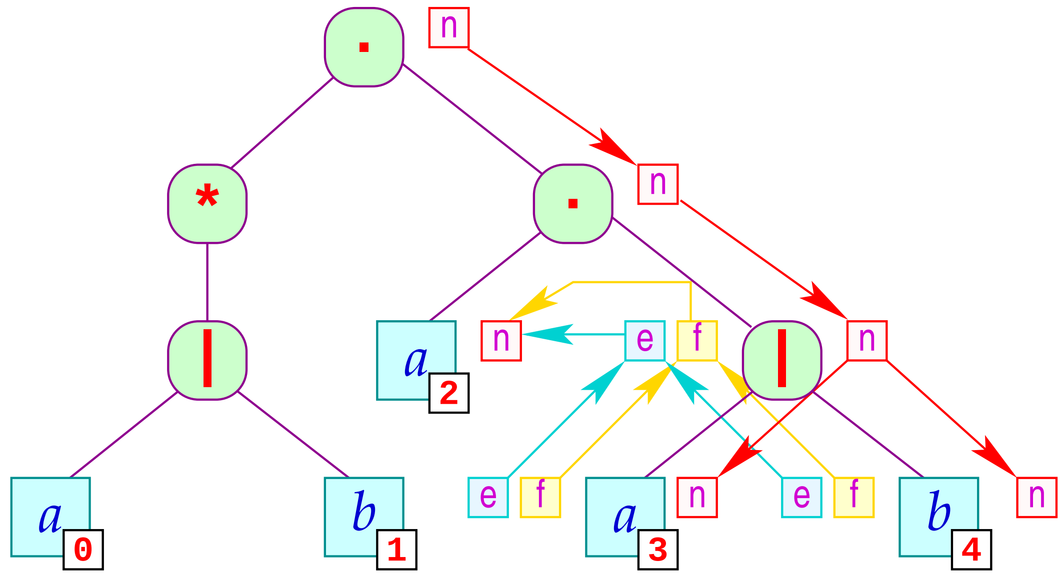
Beispiel, bedarfsgetrieben:



Beispiel, bedarfsgetrieben:



Beispiel, bedarfsgetrieben:



Diskussion:

- Die Reihenfolge hängt i.a. vom zu attributierenden Baum ab.
- Der Algorithmus muss sich merken, welche Attribute er bereits berechnete :-(
- Der Algorithmus besucht manche Knoten unnötig oft.
- Der Algorithmus ist nicht-lokal :-((

Mögliche Strategien (Forts.):

(2) Auswertung in Pässen:

- Minimiere die Anzahl der Besuche an jedem Knoten.
- Organisiere die Auswertung in Durchläufe durch den Baum.
- Berechne für jeden Pass eine Besuchsstrategie für die Knoten zusammen mit einer lokalen Strategie für jeden Knoten-Typ ...

Achtung:

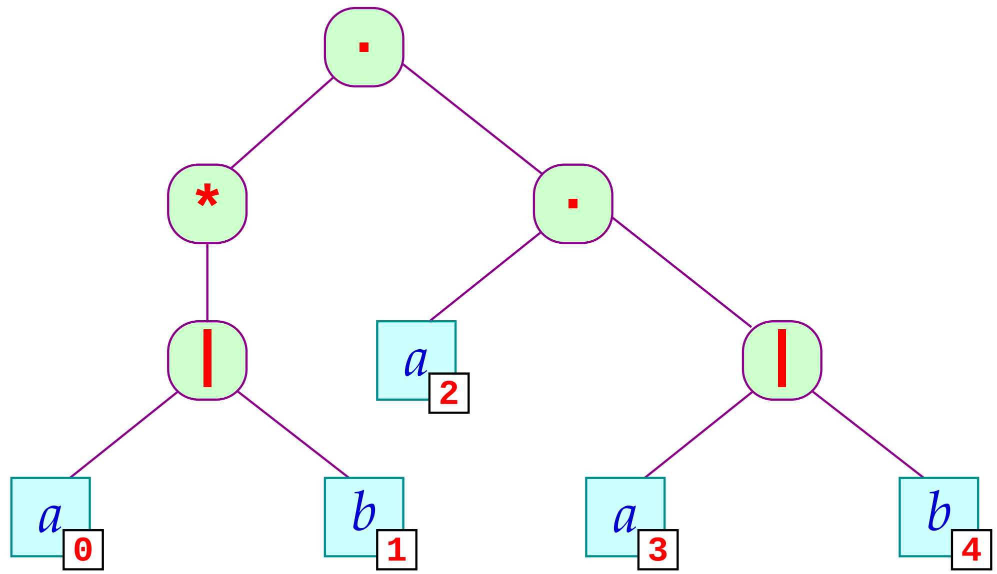
- Das minimale Attribut in der Anordnung für stark azyklische Attributierungen lässt sich stets in einem Pass berechnen :-)
- Man braucht folglich für stark azyklische Attributierungen maximal so viele Pässe, wie es Attribute gibt :-))
- Hat man einen Baum-Durchlauf zur Berechnung einiger Attribute, kann man überprüfen, ob er geeignet ist, gleichzeitig weitere Attribute auszuwerten $\Longrightarrow$ Optimierungsproblem

... im Beispiel:

empty und first lassen sich gemeinsam berechnen.

next muss in einem weiteren Pass berechnet werden :-(

Weiteres Beispiel: Nummerierung der Blätter eines Baums:

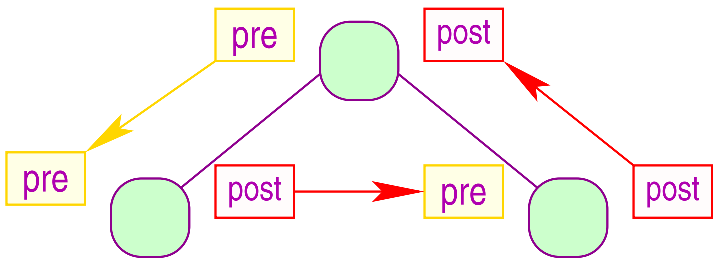


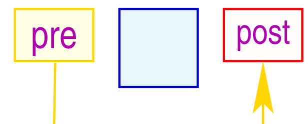
Idee:

- Führe Hilfsattribute pre und post ein !
- Mit pre reichen wir einen Zählerstand nach unten
- Mit post reichen wir einen Zählerstand wieder nach oben ...

$$\begin{array}{llll}
\text{Root:} & \mathsf{pre}[0] & := & 0 \\
 & \mathsf{pre}[1] & := & \mathsf{pre}[0] \\
 & \mathsf{post}[0] & := & \mathsf{post}[1] \\
 & & & \\
\text{Node:} & \mathsf{pre}[1] & := & \mathsf{pre}[0] \\
 & \mathsf{pre}[2] & := & \mathsf{post}[1] \\
 & \mathsf{post}[0] & := & \mathsf{post}[2] \\
 & & & \\
\text{Leaf:} & \mathsf{post}[0] & := & \mathsf{pre}[0] + 1
\end{array}$$

... die lokalen Attribut-Abhängigkeiten:





- Die Attributierung ist offenbar stark azyklisch :-)
- Man kann alle Attribute in einem Links-Rechts-Durchlauf auswerten :-))
- So etwas nennen wir L-Attributierung.
- L-Attributierung liegt auch unseren Query-Tools zur Suche in XML-Dokumenten zugrunde $\Longrightarrow$ fxgrep

Praktische Erweiterungen:

- Symboltabellen, Typ-Überprüfung / Inferenz und (einfache) Codegenerierung können durch Attributierung berechnet werden :-)
- In diesen Anwendungen werden stets Syntaxbäume annotiert.
- Die Knoten-Beschriftungen entsprechen den Regeln einer kontextfreien Grammatik :-)
- Knotenbeschriftungen können in Sorten eingeteilt werden — entsprechend den Nichtterminalen auf der linken Seite ...
- Unterschiedliche Nichtterminale benötigen evt. unterschiedliche Mengen von Attributen.
- Eine attributierte Grammatik ist eine CFG erweitert um:

  $\longrightarrow$ Attribute für jedes Nichtterminal;

  $\longrightarrow$ lokale Attribut-Gleichungen.

- Damit können die syntaktische, Teile der semantischen Analyse wie der Codeerzeugung generiert werden :-)

4 Die Optimierungsphase

1. Vermeidung überflüssiger Berechnungen

   $\longrightarrow$ verfügbare Ausdrücke

   $\longrightarrow$ Konstantenpropagation/Array-Bound-Checks

   $\longrightarrow$ Code Motion

2. Ersetzen teurer Berechnungen durch billige

   $\longrightarrow$ Peep Hole Optimierung

   $\longrightarrow$ Inlining

   $\longrightarrow$ Reduction of Strength

   ...

3. Anpassung an Hardware

$\longrightarrow$ Instruktions-Selektion

$\longrightarrow$ Registerverteilung

$\longrightarrow$ Scheduling

$\longrightarrow$ Speicherverwaltung

Beobachtung 1: Intuitive Programme sind oft ineffizient.

Beispiel:

```
void swap (int i, int j) {
     int t;
     if (a[i] > a[j]) {
         t = a[j];
         a[j] = a[i];
         a[i] = t;
        }
     }
```

Ineffizienzen:

- Adressen `a[i]`, `a[j]` werden je dreimal berechnet :-(
- Werte `a[i]`, `a[j]` werden zweimal geladen :-(

Verbesserung:

- Gehe mit Pointer durch das Feld `a`;
- speichere die Werte von `a[i]`, `a[j]` zwischen!

```
void swap (int *p, int *q) {
     int t, ai, aj;
     ai = *p; aj = *q;
     if (ai > aj) {
         t = aj;
         *q = ai;
         *p = t;     // t kann auch noch
        }            // eingespart werden!
     }
```

Beobachtung 2:

Höhere Programmiersprachen (sogar C :-) abstrahieren von Hardware und Effizienz.

Aufgabe des Compilers ist es, den natürlich erzeugten Code an die Hardware anzupassen.

Beispiele:

... Füllen von Delay-Slots;

... Einsatz von Spezialinstruktionen;

... Umorganisation der Speicherzugriffe für besseres Cache-Verhalten;

... Beseitigung (unnötiger) Tests auf Overflow/Range.

Beobachtung 3:

Programm-Verbesserungen sind nicht immer korrekt :-(

Beispiel:

`y = f() + f();` $\Longrightarrow$ `y = 2 * f();`

Idee: Spare zweite Auswertung von `f()` ...

Beobachtung 3:

Programm-Verbesserungen sind nicht immer korrekt :-(

Beispiel:

```
y = f() + f();      ⟹      y = 2 * f();
```

Idee: Spare zweite Auswertung von `f()` ???

Problem: Die zweite Auswertung könnte ein anderes Ergebnis liefern als die erste (z.B. wenn `f()` aus der Eingabe liest :-)

Folgerungen:

$\Longrightarrow$ Optimierungen haben Voraussetzungen.

$\Longrightarrow$ Die Voraussetzungen muss man:

- formalisieren,
- überprüfen :-)

$\Longrightarrow$ Man muss beweisen, dass die Optimierung korrekt ist, d.h. die Semantik erhält !!!

Beobachtung 4:

Optimierungs-Techniken hängen von der Programmiersprache ab:

$\longrightarrow$ welche Ineffizienzen auftreten;

$\longrightarrow$ wie gut sich Programme analysieren lassen;

$\longrightarrow$ wie schwierig / unmöglich es ist, Korrektheit zu beweisen ...

Beispiel: Java

Unvermeidbare Ineffizienzen:

- ∗ Array-Bound Checks;
- ∗ dynamische Methoden-Auswahl;
- ∗ bombastische Objekt-Organisation ...

Analysierbarkeit:

- \+ keine Pointer-Arithmetik;
- \+ keine Pointer in den Stack;
- − dynamisches Klassenladen;
- − Reflection, Exceptions, Threads, ...

Korrektheitsbeweise:

- \+ mehr oder weniger definierte Semantik;
- − Features, Features, Features;
- − Bibliotheken mit wechselndem Verhalten ...

Beispiel: Zwischendarstellung von `swap()`

$$
\begin{array}{lllll}
0: & A_1 = A_0 + 1 * i; & & // & A_0 == \&a \\
1: & R_1 = M[A_1]; & & // & R_1 == a[i] \\
2: & A_2 = A_0 + 1 * j; & & & \\
3: & R_2 = M[A_2]; & & // & R_2 == a[j] \\
4: & \text{if } (R_1 > R_2)\ \{ & & & \\
5: & \quad A_3 = A_0 + 1 * j; & & & \\
6: & \quad t = M[A_3]; & & & \\
7: & \quad A_4 = A_0 + 1 * j; & & & \\
8: & \quad A_5 = A_0 + 1 * i; & & & \\
9: & \quad R_3 = M[A_5]; & & & \\
10: & \quad M[A_4] = R_3; & & & \\
11: & \quad A_6 = A_0 + 1 * i; & & & \\
12: & \quad M[A_6] = t; & & & \\
 & \quad \} & & &
\end{array}
$$

Optimierung 1: $1 * R \Longrightarrow R$

Optimierung 2: Wiederbenutzung von Teilausdrücken

$$A_1 == A_5 == A_6$$
$$A_2 == A_3 == A_4$$

$$M[A_1] == M[A_5]$$
$$M[A_2] == M[A_3]$$

$$R_1 == R_3$$

Damit erhalten wir:

$$
\begin{array}{lll}
A_1 & = & A_0 + i; \\
R_1 & = & M[A_1]; \\
A_2 & = & A_0 + j; \\
R_2 & = & M[A_2]; \\
\multicolumn{3}{l}{\textsf{if } (R_1 > R_2) \ \{} \\
 & & \begin{array}{lll} t & = & R_2; \\ M[A_2] & = & R_1; \\ M[A_1] & = & t; \\ \} & & \end{array}
\end{array}
$$

Optimierung 3: Verkürzung von Zuweisungsketten :-)

Ersparnis:

	vorher	nachher
$+$	6	2
$*$	6	0
load	4	2
store	2	2
$>$	1	1
$=$	6	2

Perspektiven

Herausforderungen:

- neue Hardware;
- neue Programmiersprachen;
- neue Anwendungen für Compiler-Technologie :-)

1 Hardware

Die Code-Erzeugung soll die Möglichkeiten der Hardware optimal ausnutzen ...

Herausforderungen:

Neue Hardware:

→ Speicher-Hierarchie mit unterschiedlich schnellen Caches für verschiedene Zwecke;

→ On-Board Nebenläufigkeit mit Pipelines, mehreren ALUs, spekulativer Parallelität, ...

→ Interaktion mit mächtigen Zusatzkomponenten wie Graphik-Karten ...

Eingeschränkte Hardware:

z.B. auf Chip-Karten, in Kühlschränken, Bremsanlagen, Steuerungen ...

$\Longrightarrow$ ubiquitous Computing

- $\longrightarrow$ minimaler Energie-Verbrauch :-)
- $\longrightarrow$ minimaler Platz :-)
- $\longrightarrow$ Echtzeit-Anforderungen;
- $\longrightarrow$ Korrektheit;
- $\longrightarrow$ Fehler-Toleranz.

2 Programmiersprachen

Spezielle Features:

- mobiler Code;
- Nebenläufigkeit;
- graphische Benutzeroberflächen;
- Sicherheits-Komponenten;
- neue / bessere Typsysteme;
- Unterstützung für Unicode und XML.

Neue Programmiersprachen:

- XSLT;
- XQuery;
- Web-Services;
- anwendungs-spezifische Sprachen ...

3 Programmierumgebungen

Diverse Programmierhilfsmittel benutzen Compiler-Technologie ...

- syntax-gesteuerte Editoren;
- Programm-Visualisierung;
- automatische Programm-Dokumentation;
- partielle Codeerzeugung aus UML-Modellen;
- UML-Modell-Extraktion $\Longrightarrow$ reverse engineering
- Konsistenz-Überprüfungen, Fehlersuche;
- Portierung.

4 Neue Anforderungen

- Zuverlässigkeit
- Sicherheit

5 Unser Programm nächstes Semester

Vorlesungen:

- Sicherheitskonzepte in Programmiersprachen — H.S. (3-stündig)

  $\Longrightarrow$ neuer Schwerpunkt: Sicherheit !!!

- Dokumentenverarbeitung — H.S. (2-stündig)
- Scripting Sprachen — Kumar Neeraj Verma (2-stündig)

Praktikum:

- Implementierung eines (Mini-) C-Compilers — Michael Petter

Seminar:

- XML-Query-Sprachen — Alex Berlea