Prof. Dr. H. Seidl

Skript zur Vorlesung

Compilerbau

Dieses Skript entstand während der Vorlesung *Compilerbau*, die im Sommersemester 1997 an der Universität Trier gelesen wurde. LaTeX–Version by Martin Toepfer / Frank Burch / Johannes Lehnert.

Inhaltsverzeichnis

1 Einführung **4**

1.1 Vorgehen eines Compilers 4

2 Eine abstrakte Maschine für C **6**

2.1 Erster Überblick über die Architektur der CMa 6

2.1.1 Datenstrukturen 6

2.1.2 Maschinenzyklus 7

2.2 Einfache Ausdrücke und Zuweisungen 7

2.2.1 Arithmetische Ausdrücke 7

2.2.2 Vergleiche 8

2.2.3 Adressen und Inhalte von Variablen 8

2.2.4 Wertzuweisung 9

2.3 Anweisungen und Anweisungsfolgen 10

2.4 (Un-)bedingte und iterative Anweisungen 10

2.4.1 Bedingte Anweisungen 10

2.4.2 Schleifen 11

2.4.3 `switch`-Anweisung 13

2.4.4 Übersetzung von `break` 14

2.5 Speicherbelegung für Variablen 14

2.5.1 Felder 14

2.5.2 Strukturen 15

2.6 Zeiger und dynamische Speicherverwaltung 15

2.6.1 Adreß- und Dereferenzierungsoperatoren 16

2.6.2 Übersicht über Schemata für Ausdrücke 17

2.6.3 Freigabe dynamisch erzeugter Datenobjekte 18
2.7 Funktionen 18
2.7.1 Speicherorganisation für Funktionen 19
2.7.2 Bestimmung der Adreßumgebung 20
2.7.3 Betreten und Verlassen von Funktionen 21
2.7.4 Zugriff auf Variablen und formale Parameter, Rückgabe von Werten . 23
2.7.5 Übersetzung eines Programms 25

3 Lexikalische Analyse **26**
3.1 Reguläre Ausdrücke 27
3.2 Endliche Automaten 27
3.3 Konstruktion eines ε–NFA aus einem regulären Ausdruck 28
3.4 Konstruktion eines NFA aus einem regulären Ausdruck 29
3.5 Konstruktion von DFAs aus NFAs 31
3.6 Minimierung von deterministischen endlichen Automaten 32
3.7 Implementierung eines DFA 34
3.8 Spezifikation von Symbolklassen 35
3.9 Generierung eines Scanners 36

4 Syntaktische Analyse **37**
4.1 Theoretische Grundlagen 37
4.1.1 Kontextfreie Grammatiken 37
4.1.2 Ableitungsbäume 39
4.1.3 Reduzierte kontextfreie Grammatiken 40
4.1.4 Kellerautomaten 43
4.2 Verbandstheorie 46
4.3 Berechnung von Vorausschaumengen 47
4.3.1 $First_k$ 47
4.3.2 $Follow_k$ 51
4.3.3 Schnelle Verfahren für Spezialfälle von Constraint–Systemen 52
4.4 Top–Down–Syntaxanalyse 55
4.4.1 Der $LL(k)$-Parser 57

4.5 Bottom–Up–Syntaxanalyse . . . 60
4.5.1 Der $LR(G)$–Automat . . . 60
4.5.2 $LR(k)$–Grammatiken . . . 66
4.5.3 Der $LR(G, k)$–Automat . . . 68
4.5.4 $LR(k)$–Parser . . . 70
4.5.5 $SLR(k)$–Parser . . . 72
4.5.6 $LALR(k)$–Parser . . . 72

5 Semantische Analyse **75**
5.1 Explizit getypte Sprachen . . . 75
5.2 Subtypen . . . 80
5.3 Implizite Typisierung und Überladung . . . 82
5.3.1 Implizite Typisierung . . . 82
5.3.2 Überladung . . . 83

Kapitel 1

Einführung

Sei P ein Programm, e die Eingabe des Programms. Wir suchen die Ausgabe a von P für e.

Es gibt mehrere Möglichkeiten, diese Ausgabe zu erhalten:

1. **Interpreter**

   $P, e \longrightarrow \boxed{I} \longrightarrow a$

   Bei einem Interpreter ist keinerlei Vorberechnung nötig, allerdings muß jeder Programmpunkt bei jedem Besuch neu untersucht werden, wodurch der Interpreter sehr langsam wird.

   Interpreter sind sehr flexibel und eignen sich gut zum debuggen.

2. **Compiler/Übersetzer**

   Der Compiler arbeitet in zwei Phasen:

   (a) Übersetzungsphase:

   $P \longrightarrow \boxed{\ddot{U}} \longrightarrow P_M$

   Hierbei liegt P in der Programmiersprache und P_M in der Maschinensprache vor.

   (b) Laufzeitphase:

   $e \longrightarrow \boxed{P_M} \longrightarrow a$

   In der ersten Phase sieht man das Programm als Ganzes und kann so globale Optimierungen vornehmen und effizientere Datenstrukturen anlegen.

1.1 Vorgehen eines Compilers

$P \longrightarrow$ $\boxed{\text{lexikalische Analyse}}$ $\longrightarrow$ *Folge von Tokens* $\longrightarrow$ $\boxed{\text{syntaktische Analyse}}$ $\longrightarrow$
$\longrightarrow$ *Syntaxbaum* $\longrightarrow$ $\boxed{\text{semantische Analyse}}$ $\longrightarrow$ *annotierter Syntaxbaum* $\longrightarrow$
$\longrightarrow$ $\boxed{\text{Codeerzeugung für Maschine } M}$ $\longrightarrow$ *Folge von M-Instruktionen*

In den einzelen Schritten geschieht folgendes:

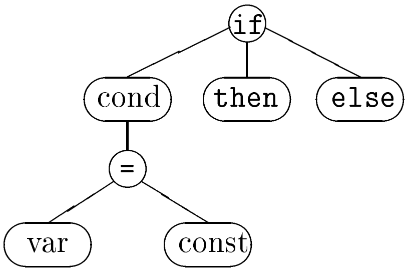
1. **lexikalische Analyse**

   Erkennen von Schlüsselwörtern, Identifiern, Zahlen und Kommentaren.

2. **Folge von Tokens**

   Eine Folge von Tokens kann z.B. so aussehen: `<integer,1997> THEN <name,"hallo">`

3. **Syntaxbaum**

   Das Programm wird in Programmkonstrukte aufgeteilt, die als Baum dargestellt werden:

4. **Semantische Analyse**

   - Erstellen einer Symboltabelle (kann auch schon während der lexikalischen Analyse geschehen)
   - Typüberprüfung
   - Konstantenpropagation

5. **Codeerzeugung**

   Hier werden die Hardwareressourcen der Maschine M (z.B. Register, Pipelines, Caches etc.) ausgenutzt.

Alternativ kann im letzten Schritt auch für eine abstrakte Maschine A Code erzeugt werden, so daß man eine Folge von A-Instruktionen erhält. Diese abstrakte Maschine A ist an die Sprachkonstrukte angepaßt, läßt sich aber dennoch auf einer konkreten Maschine leicht implementieren.

- **Nachteil:** Wir benötigen einen Interpreter für A-Instruktionen, um das Programm ausführen zu können.

- **Vorteil:** Der erzeugte Code ist plattform-unabhängig: um den Code auf einer konkreten Maschine laufen zu lassen, muß auf dieser nur der Interpreter installiert werden (Ursprungsidee: p-Code für Pascal)

Kapitel 2

Eine abstrakte Maschine für C

Quellsprache: eine Teilmenge der Programmiersprache C

Zielrechner: *CMa*

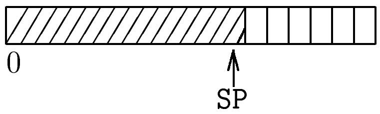
Sprachkonzepte:

- Variablen: Bezeichner für Speicherstellen, an denen Daten stehen
- Ausdrücke
- explizite Angabe des Kontrollflusses mit `if`/`for`/`while`/...
- Funktionen

2.1 Erster Überblick über die Architektur der CMa

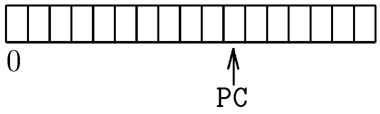
2.1.1 Datenstrukturen

- Speicher `S`

  Datenspeicher, Vergabe gemäß LIFO, also ein Keller/Stack. Jede Zelle von S kann eine Adresse oder einen elementaren Wert aufnehmen (z. B. `char, int, float`)

- Codespeicher `C`

  Code- oder Befehlsspeicher. Jede Zelle aus C kann einen CMa-Befehl aufnehmen.

- Register
  `SP` Stackpointer, zeigt auf die oberste belegte Stackzelle
  `PC` Program Counter, zeigt zu Beginn eines Maschinenzyklus auf die nächste auszuführende Instruktion

2.1.2 Maschinenzyklus

Ein Zyklus führt eine Instruktion aus und erhöht implizit den `PC`:

1. Lade Instruktion aus `C[PC]`
2. `PC++;`
3. Führe Instruktion aus

Die erste Instruktion liegt in `C[0]`, daher setzen wir anfangs `PC = 0`. Die Schleife wird solange durchlaufen, bis die Instruktion **`halt`** ausgeführt wird.

Den Instruktionssatz der *CMa* entwickeln wir nach Bedarf.

Wie real ist die *CMa* ?

- Sie besitzt keine Speicherhierarchie (Register, Caches, Primär-, Sekundärspeicher).
- Sie kennt nur elementare Typen von konstanter Größe (Speicherzelle).

Nutzen der *CMa* ?

- Erzeugung von Zwischencode für portablen Übersetzer.
- Grundlage für erweiterte Übersetzungsschemata.

2.2 Einfache Ausdrücke und Zuweisungen

2.2.1 Arithmetische Ausdrücke

Aufgabe: Werte Ausdrücke wie etwa `(1+7)*3` aus.

Idee:

- Lade die Konstanten auf den Stack
- Wende die Operatoren auf die beiden obersten Stackelemente an.

Alternativ könnten wir die *CMa* auch um abstrakte Register erweitern.

Befehl	Bedeutung
`loadc q`	`SP++; S[SP] = q;`
`mul`	`S[SP-1] = S[SP-1] * S[SP]; SP--;`
`add/sub/div/mod`	Analog zu `mul`. Konvention: 1. Argument: `S[SP-1]` 2. Argument: `S[SP]`
`neg`	`S[SP] = -S[SP];`

Beispiel: Code, um `(1+7)*3` zu berechnen:

```
loadc 1
loadc 7
add
loadc 3
mul
```

Invariante: Am Ende der Ausdrucksauswertung liegt der Wert des Ausdrucks oben auf dem Stack.

2.2.2 Vergleiche

Instruktionen für arithmetische und Boole'sche Vergleiche von Ausdrücken:

Befehl	Bedeutung
`and`	`S[SP-1] = (S[SP-1] && S[SP]); SP--;`
`or`	`S[SP-1] = (S[SP-1] || S[SP]); SP--;`
`not`	`S[SP] = !(S[SP]);`
`eq`	`S[SP-1] = (S[SP-1] == S[SP]); SP--;`
`geq`	`S[SP-1] = (S[SP-1] >= S[SP]); SP--;`
`leq`	`S[SP-1] = (S[SP-1] <= S[SP]); SP--;`
`gr`	`S[SP-1] = (S[SP-1] > S[SP]); SP--;`
`le`	`S[SP-1] = (S[SP-1] < S[SP]); SP--;`
`neq`	`S[SP-1] = (S[SP-1] != S[SP]); SP--;`

2.2.3 Adressen und Inhalte von Variablen

Variablen ordnen wir Speicherzellen in S zu.
Sei `x = y` eine Zuweisung, `x`, `y` Variablen. Von `x` benötigen wir die *Adresse*, wo der Inhalt von `x` aufbewahrt wird, während wir von `y` nur den *Inhalt* benötigen.

L-Wert von `x` = Adresse der Variablen `x`
R-Wert von `x` = Inhalt der Variablen `x`

Also benötigen wir als weiteres Argument für die Übersetzungsfunktion eine Funktion ρ, die für jede Variable `x` die (Relativ-) Adresse von `x` liefert. ρ heißt *Adreßumgebung*.

Wir definieren zwei Übersetzungsfunktionen:

$code_L$ erzeugt für Ausdruck e Code, der den L-Wert von e berechnet (formal $code_L\ e\ \rho$);

$code_R$ erzeugt für Ausdruck e Code, der den R-Wert von e berechnet (formal $code_R\ e\ \rho$).

Beispiel:

$$\begin{array}{lcl}
code_R\ (e_1 + e_2)\ \rho & = & code_R\ e_1\ \rho \\
 & & code_R\ e_2\ \rho \\
 & & \texttt{add} \\
code_R\ (e_1 * e_2)\ \rho & = & code_R\ e_1\ \rho \\
 & & code_R\ e_2\ \rho \\
 & & \texttt{mul} \\
code_R\ (-e)\ \rho & = & code_R\ e\ \rho \\
 & & \texttt{neg}
\end{array}$$

Analog werden alle anderen ein- und zweistelligen Operatoren behandelt.

Sei x eine Variable, q eine Konstante. Dann gilt:

$$\begin{array}{lcl}
code_R\ q\ \rho & = & \texttt{loadc}\ q \\
code_R\ x\ \rho & = & \texttt{loada}\ \rho(x) \\
code_L\ x\ \rho & = & \texttt{loadc}\ \rho(x)
\end{array}$$

Befehl	Bedeutung
`loada` a	`SP++; S[SP] = S[`a`];`
`store`	`S[S[SP]] = S[SP-1]; SP--;`
`pop`	`SP--;`
`jump` q	`PC =` q

`jumpz` q	`if (S[SP] == 0)`
	`  PC =` q`;`
	`SP--;`

2.2.4 Wertzuweisung

$$\begin{array}{lcl}
code_R\ (x = e)\ \rho & = & code_R\ e\ \rho \\
 & & code_L\ x\ \rho \\
 & & \texttt{store}
\end{array}$$

Beispiel: $e \equiv$ `a=(b+7)-c`, $\rho : [\texttt{a} \longmapsto 5,\ \texttt{b} \longmapsto 6,\ \texttt{c} \longmapsto 7]$

$$\begin{array}{lcl}
code_R\ e\ \rho & = & code_R\ \texttt{((b+7)-c)}\ \ \rho \\
 & & code_L\ \texttt{a}\ \ \rho \\
 & & \texttt{store} \\
 & = & code_R\ \texttt{((b+7)-c)}\ \ \rho \\
 & & \texttt{loadc}\ \rho(\texttt{a}) \\
 & & \texttt{store} \\
 & = & code_R\ \texttt{(b+7)}\ \ \rho \\
 & & code_R\ \texttt{(c)}\ \ \rho \\
 & & \texttt{sub} \\
 & & \texttt{loadc 5} \\
 & & \texttt{store}
\end{array}$$

$=$

```
loada 6
loadc 7
add
loada 7
sub
loadc 5
store
```

2.3 Anweisungen und Anweisungsfolgen

Wir können aus einem Ausdruck e eine Anweisung (Statement) e; machen. Anweisungen liefern *keine* Werte zurück, daher ist der `SP` vor und nach der Ausführung gleich.

code $(e;)\ \rho$	$=$	$\mathit{code}_R\ e\ \rho$
		`pop`
code $(s\ ss)\ \rho$	$=$	*code* $s\ \rho$
		code $ss\ \rho$

2.4 (Un-)bedingte und iterative Anweisungen

Für `goto` *label* erzeugen wir einen abstrakten Befehl `jump` *label*, der den Programmzähler `PC` auf den Wert $\#label$ setzt, wobei $\#label$ die Anfangsadresse im Codespeicher C des Codes für das Statement ist, vor dem das Label steht.

2.4.1 Bedingte Anweisungen

- einseitig: $s \equiv$ `if` $(e)\ s'$

  Der Compiler muß weitere Sprungziele für beliebige Sprünge festlegen. Daher erzeugt er ebenfalls zuerst *symbolische* Sprungziele, die in einem zweiten Pass aufgelöst werden. (Alternative: Back-Patching[1])

code $s\ \rho$	$=$	$\mathit{code}_R\ e\ \rho$
		`jumpz` l
		code $s'\ \rho$
	$l:$	$\ldots$

  Hierbei ist l die symbolische Adresse der ersten Instruktion nach dem Code für s'.

- zweiseitig: $s \equiv$ `if` $(e)\ s_1$ `else` s_2

code $s\ \rho$	$=$	$\mathit{code}_R\ e\ \rho$
		`jumpz` l_1
		code $s_1\ \rho$
		`jump` l_2
	$l_1:$	*code* $s_2\ \rho$
	$l_2:$	$\ldots$

[1] Der Compiler erzeugt den Code in einem Pass und ersetzt nachträglich im bereits erzeugten Code die symbolischen Sprungziele

Beispiel: $s \equiv$
```
if (a > b)
  c = a;
else
  c = b;
```
liefert folgenden Code:

$$
\begin{array}{lrl}
code\ s\ \rho & = & code_R\ (\texttt{a > b})\ \rho \\
 & & \texttt{jumpz}\ l_1 \\
 & & code\ (\texttt{c = a;})\ \rho \\
 & & \texttt{jump}\ l_2 \\
 & l_1: & code\ (\texttt{c = b;})\ \rho \\
 & l_2: & \ldots \\
 & = & \texttt{loada}\ \rho(\texttt{a}) \\
 & & \texttt{loada}\ \rho(\texttt{b}) \\
 & & \texttt{gr} \\
 & & \texttt{jumpz}\ l_1 \\
 & & \texttt{loada}\ \rho(\texttt{a}) \\
 & & \texttt{loadc}\ \rho(\texttt{c}) \\
 & & \texttt{store} \\
 & & \texttt{pop} \\
 & & \texttt{jump}\ l_2 \\
 & l_1: & \texttt{loada}\ \rho(\texttt{b}) \\
 & & \texttt{loadc}\ \rho(\texttt{c}) \\
 & & \texttt{store} \\
 & & \texttt{pop} \\
 & l_2: & \ldots
\end{array}
$$

2.4.2 Schleifen

- `while`-Schleife:

  Sei $s \equiv$ `while(`e`)` s'. Dann gilt:

$$
\begin{array}{lrl}
code\ s\ \rho & = l_1: & code_R\ e\ \rho \\
 & & \texttt{jumpz}\ l_2 \\
 & & code\ s'\ \rho \\
 & & \texttt{jump}\ l_1 \\
 & l_2: & \ldots
\end{array}
$$

  Beispiel: $s \equiv$
```
while (a > 0) {
  c = c + 1;
  a = a - b;
}
```
  liefert:

$$
\begin{array}{lrl}
\textit{code}\ s\ \rho & = l_1: & \texttt{loada}\ \rho(\texttt{a}) \\
 & & \texttt{loadc}\ 0 \\
 & & \texttt{gr} \\
 & & \texttt{jumpz}\ l_2 \\
 & & \texttt{loada}\ \rho(\texttt{c}) \\
 & & \texttt{loadc}\ 1 \\
 & & \texttt{add} \\
 & & \texttt{loadc}\ \rho(\texttt{c}) \\
 & & \texttt{store} \\
 & & \texttt{pop} \\
 & & \texttt{loada}\ \rho(\texttt{a}) \\
 & & \texttt{loada}\ \rho(\texttt{b}) \\
 & & \texttt{sub} \\
 & & \texttt{loadc}\ \rho(\texttt{a}) \\
 & & \texttt{store} \\
 & & \texttt{pop} \\
 & & \texttt{jump}\ l_1 \\
 & l_2: & \ldots
\end{array}
$$

- `for`-Schleife:

  $s \equiv \texttt{for(}e_1\texttt{;}e_2\texttt{;}e_3\texttt{)}\ s'$

$$
\begin{array}{lrl}
\textit{code}\ s\ \rho & = & \textit{code}_R\ e_1\ \rho \\
 & & \texttt{pop} \\
 & l_1: & \textit{code}_R\ e_2\ \rho \\
 & & \texttt{jumpz}\ l_2 \\
 & & \textit{code}\ s'\ \rho \\
 & & \textit{code}_R\ e_3\ \rho \\
 & & \texttt{pop} \\
 & & \texttt{jump}\ l_1 \\
 & l_2: & \ldots
\end{array}
$$

- `do-while`-Schleife:

  $s \equiv \texttt{do}\ s'\ \texttt{while(}e\texttt{)}$

$$
\begin{array}{lrl}
\textit{code}\ s\ \rho & = l_1: & \textit{code}\ s'\ \rho \\
 & & \textit{code}_R\ e\ \rho \\
 & & \texttt{jumpz}\ l_2 \\
 & & \texttt{jump}\ l_1 \\
 & l_2 & \ldots
\end{array}
$$

2.4.3 `switch`-Anweisung

Wir betrachten hier nur eine vereinfachte Form der `switch`-Anweisung:

$s \equiv$ `switch(`e`) {`
- `case 0:` ss_0 `break;`
- `case 1:` ss_1 `break;`
- $\vdots$
- `case k-1:` ss_{k-1} `break;`
- `default :` ss_k

`}`

Um `switch` effizient zu übersetzen, benötigen wir einen neuen abstrakten Befehl `jumpi`, der *indizierte* Sprünge erlaubt. Zusätzlich legen wir eine *Sprungtabelle* an.

code $s\ \rho$	$=$		*code*$_R$ $e\ \rho$
			`copy`
			`loadc` k
			`le`
			`jumpz` $l+k$
			`copy`
			`loadc` 0
			`geq`
			`jumpz` $l+k$
			`jumpi` l
		l_0 :	*code* $ss_0\ \rho$
			`jump` l'
		l_1 :	*code* $ss_1\ \rho$
			`jump` l'
			$\vdots$
		l_k :	*code* $ss_k\ \rho$
			`jump` l'
		l :	`jump` l_0
		$l+1$:	`jump` l_1
$\vdots$			
		$l+k-1$:	`jump` l_{k-1}
		$l+k$:	`pop`
			`jump` l_k
		l' :	$\ldots$

Der Befehl `copy` ist notwendig, da der R–Wert von e zweimal benötigt wird. Im Default-Fall muß `pop` aufgerufen werden, da `jumpi` nicht ausgeführt wurde.

Die Anordnung der Sprungtabelle hinter die Alternativen ermöglicht die Codeerzeugung in einem Übersetzungsschritt (Pass).

Befehl	Bedeutung
`jumpi` l	`PC =` l `+ S[SP]; SP--;`
`copy`	`S[SP+1] = S[SP]; SP++;`

2.4.4 Übersetzung von `break`

`break` entspricht einem Sprung ans Ende der `switch`-Anweisung bzw. Schleife, die das Vorkommen von `break` unmittelbar umgibt.

Problem: Wir müssen das richtige Sprungziel l zum textuellen Vorkommen von `break` transportieren.

Idee: wir erweitern die Code-Funktion um ein weiteres Argument, das jeweils das richtige Label enthält.

Dann gilt z.B.:

$$\begin{array}{lrl} \mathit{code}\ \texttt{break}\ \rho\ l & = & \texttt{jump}\ l \\ \mathit{code}\ \texttt{while(}e\texttt{)}\ s\ \rho\ l & = l_1: & \mathit{code}_R\ e\ \rho \\ & & \texttt{jumpz}\ l_2 \\ & & \mathit{code}\ \ s\ \rho\ l_2 \\ & & \texttt{jump}\ l_1 \\ & l_2 & \ldots \end{array}$$

2.5 Speicherbelegung für Variablen

Ziel: Ordne einer Variablen `x` statisch, d.h. zur Übersetzungszeit feste (relative) Adressen zu.

Annahme:

- Variablen vom Typ `int`, `float` etc. erhalten eine Speicherzelle.
- Variaben im Definitionsteil werden konsekutiv ab Adresse 1 in S abgelegt.

Sei $d \equiv t_1\ x_1\texttt{;}\ \ldots\ t_k\ x_k\texttt{;}$ mit t_j einfacher Typ und x_j Variable.

Dann ist $\rho(x_i) = i$ für $i = 1, \ldots, k$.

2.5.1 Felder

Beispiel: `int a[11];`

`a` enthält 11 Elemente, benötigt also 11 Zellen. Die Speicherzelle $\rho(\texttt{a})$ enthält die Adresse des Elements `a[0]`.

Wir benötigen also eine Funktion $|\cdot|$, die den Platzbedarf eines Typs t berechnet. Dabei gilt:

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

Damit ergibt sich für $d \equiv t_1\ x_1; \ldots t_k\ x_k;$: $\rho(x_1) = 1$
$\rho(x_i) = \rho(x_{i-1}) + |t_{i-1}|$ für $i > 1$

Wichtig: Der Platzbedarf eines Typs muß zur Übersetzungszeit bekannt sein.

Erweiterung des $code_L$ -Schemas für Felder:

Sei $t\ \ a[c]$ eine Felddefinition. Falls t ein einfacher Typ ist, so liegt die Adresse von $a[i]$ bei $\rho(a) + i\ \ (0 \leq i < c)$. Andernfalls ermittelt sich die Adresse von $a[i]$ zu $\rho(a) + i \cdot |t|$.

Falls der R-Wert von i zur Übersetzungszeit nicht bekannt ist, so erzeugen wir Code, der ihn zur Laufzeit berechnet.

$code_L\ e_1[e_2]\,\rho$ = $code_R\ e_1\,\rho$
$code_R\ e_2\,\rho$
`loadc` $|t|$
`mul`
`add`

Die Anfangsadresse des Speicherbereichs eines Feldes a ist auch der R-Wert von a.
Für ein Feld a gilt also stets $code_R\ a\,\rho\ = code_L\ a\,\rho$.

$code_R\ e_1[e_2]\,\rho$ = $code_L\ e_1[e_2]\,\rho$
`load`

Befehl	Bedeutung
`load`	`S[SP] = S[S[SP]]`

2.5.2 Strukturen

Wir vereinfachen das Problem, indem wir die mehrmalige Benutzung von Komponentennamen verbieten. Dadurch können wir Strukturen ebenfalls in der Adreßumgebung verwalten. Sei z.B. folgende Struktur gegeben:

```
struct {int a; int b} x;
```

`x` erhält die erste freie Zelle, für `a` und `b` vergeben wir Relativadressen innerhalb des Speicherbereichs von `x`.

Allgemein:

Sei $t \equiv$ `struct {` $t_1\,c_1$ `;` $\ldots$ `;` $t_k\,c_k$ `}`.

Dann ist $|t| = \sum_{j=1}^{k} |t_j|$, wobei $\rho(c_1) = 0$ und $\rho(c_j) = \sum_{i=0}^{j-1} |t_i|$ mit $j > 0$

$code_L\ (e.c)\,\rho$ = $code_L\ e\,\rho$
`loadc` $\rho(c)$
`add`

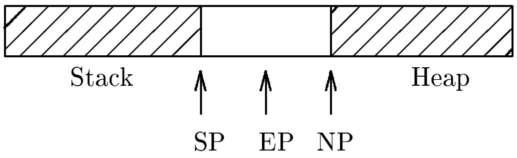
2.6 Zeiger und dynamische Speicherverwaltung

Bisher haben wir Datenobjekte durch Definitionen erzeugt; wir haben ihnen feste (relative) Adressen zugeordnet und über Namen auf sie zugegriffen.

Zeiger gestatten den Zugriff auf anonyme Datenobjekte, von denen eine statisch nicht bekannte Anzahl erzeugt werden kann. Wir erzeugen diese Datenobjekte mit einer neuen *CMa*-Instruktion `new`, die ein entsprechend großes Stück Speicher reserviert.

Problem: Zur Laufzeit erzeugte Datenobjekte können den aktuellen Prozeduraufruf überleben, sie genügen daher nicht der LIFO-Disziplin. Daher müssen wir die Speicherorganisation erweitern: wir benötigen nun zwei unabhängige, beliebig große Datenstrukturen.

Hier implementieren wir beide Datenstrukturen in einem Segment:



`NP` – New Pointer, zeigt auf das oberste belegte Element des Heaps.

`EP` – Extreme Pointer, gibt an, wieviele Stackzellen die aktuelle Prozedur maximal benötigt.

$code_R$ `malloc(`e`)` ρ = $code_R\ e\ \rho$
`new`

Befehl	Bedeutung

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

2.6.1 Adreß- und Dereferenzierungsoperatoren

`&`e liefert die Adresse der Variablen e, d.h. den L-Wert von e
`*`e liefert Datenobjekt, auf das e zeigt, d.h. dessen L-Wert in e abgespeichert ist.

$code_R$ (`&`e) ρ = $code_L\ e\ \rho$

$code_L$ (`*`e) ρ = $code_R\ e\ \rho$

$code_L$ (e`->`a) ρ = $code_R\ e\ \rho$
`loadc` $\rho(a)$
`add`

Beispiel:
```
struct t {
  int a[7];
  struct t *p;
}
int      i, j;
struct t *pt;
```

Sei ρ definiert als: $\rho : [\mathtt{a} \longmapsto 0, \mathtt{p} \longmapsto 7, \mathtt{i} \longmapsto 1, \mathtt{j} \longmapsto 2, \mathtt{pt} \longmapsto 3]$.

$code_L$ `(((pt->p)->a)[i+1])`ρ = $code_R$ `((pt->p)->a)` ρ
`loada` 1
`loadc` 1
`add`
`loadc` 1
`mul`
`add`

= $code_L$ `((pt->p)->a)` ρ

$\vdots$

= $code_L$ `(pt->p)` ρ
`load`
`loadc` 0
`add`

$\vdots$

= `loadc` 3
`load`
`loadc` 7
`add`
`load`
`loadc` 0
`add`

$\vdots$

Achtung: Wenn e ein Feld ist, so wird sein Wert als (evtl. konstanter) *Zeiger* auf das nullte Feldelement aufgefaßt. Insbesondere gilt e `== &(`e`[0])` und folglich ist $code_R\ e\ \rho = code_L\ e\ \rho$.

Dies gilt allerdings nur in *unseren* Schemata für die Übersetung. L-Werte von Feldern kommen in Programmen nicht vor.

2.6.2 Übersicht über Schemata für Ausdrücke

$code_L$ $(e_1$`[`e_2`]`$)\ \rho$ = $code_R\ e_1\ \rho$
$code_R\ e_2\ \rho$
`loadc` $|t|$
`mul`
`add` falls t e_1

$code_L$ $(e$`.`$a)\ \rho$ = $code_L\ e\ \rho$
`loadc` $\rho(a)$
`add`

$code_L$ $(e$`->`$a)\ \rho$ = $code_R\ e\ \rho$
`loadc` $\rho(a)$
`add`

$code_L$ `(*`e`)` ρ = $code_R\ e\ \rho$

$code_L\ x\ \rho$ = `loadc` $\rho(x)$ für Variable x

$code_R\ c\ \rho$ = `loadc` c für Konstante c

$code_R\ (\circ\ e)\ \rho$	$=$	$code_R\ e\ \rho$ op	 mit $\circ \in \{$`-`,`!`$\}$ und $op \in \{$`neg`,`not`$\}$
$code_R\ (e_1 \Box e_2)\ \rho$	$=$	$code_R\ e_1\ \rho$ $code_R\ e_2\ \rho$ op	 mit $\Box \in \{$`+`,`*`,`||`,`==`$,\ldots\}$ und $op \in \{$`add`,`mul`,`or`,`eq`$,\ldots\}$
$code_R\ (e_1$ `=` $e_2)\ \rho$	$=$	$code_R\ e_2\ \rho$ $code_L\ e_1\ \rho$ `store`	
$code_R$ (`&`e) ρ	$=$	$code_L\ e\ \rho$	
$code_R\ e\ \rho$	$=$	$code_L\ e\ \rho$	falls e Feld
$code_R\ x\ \rho$	$=$	`loada` $\rho(x)$	falls x einfach oder Zeiger
$code_R\ e\ \rho$	$=$	$code_L\ e\ \rho$ `load`	 sonst

2.6.3 Freigabe dynamisch erzeugter Datenobjekte

Problem:

- Durch die Freigabe entstehen „Löcher“ im Heap
- Speicherfehler (Allozierungsfehler) können entstehen, wenn noch Verweise auf schon freigegebene Datenobjekte bestehen

Daher machen wir es uns sehr einfach: wir tun nichts.

$code_R$ `free(`e`)` ρ	$=$	$code_R\ e\ \rho$ `pop`

2.7 Funktionen

Die Definition einer Funktion besteht aus:

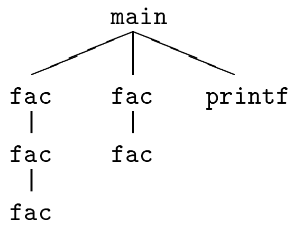
- einem Namen, mit dem sie aufgerufen werden kann
- einer Spezifikation der formalen Parameter (Argumente), die die Eingabeschnittstelle darstellen
- einem evtl. Ergebnistyp
- einem Rumpf oder Anwendungsteil, der das auszuführende Programmstück enthält

Beispiel: Berechnung der Fakultät

```
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",n);
}
```

Zu einem Ausführungszeitpunkt können dabei mehrere Instanzen einer Funktion aktiv, d.h. aufgerufen, aber noch nicht verlassen sein. Wir müssen diese verschiedenen Instanzen auseinanderhalten.

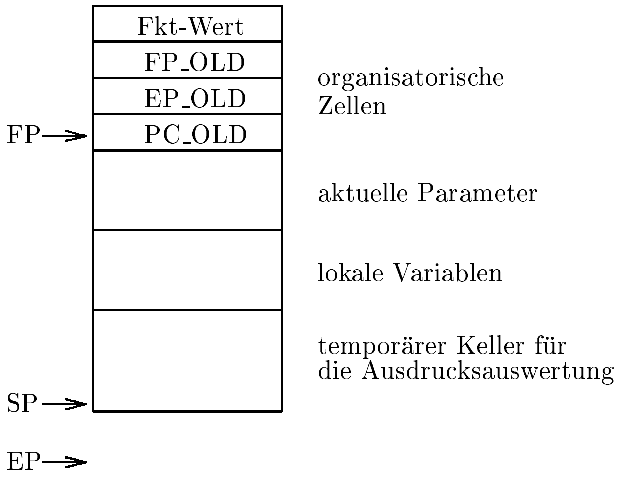
Rekursionsbaum:



Idee: lege für jede aufgerufene Funktion jeweils einen neuen Kellerrahmen an. Wenn die Funktion verlassen wird geben wir den Kellerrahmen auf und fahren im darunterliegenden Rahmen fort.

Bemerkung: In Programmiersprachen wie z.B. Modula–2 sind auch geschachtelte Prozedurdefinitionen zulässig.

2.7.1 Speicherorganisation für Funktionen



- `FP` – Frame Pointer. Er zeigt auf eine feste Zelle innerhalb des Kellerrahmens. *Relativ* dazu werden formale Parameter und lokale Variablen adressiert. Bei uns zeigt `FP` auf die dritte organisatorische Zelle, die die Rücksprungadresse enthält.
- `PC_OLD` – Rücksprungadresse, d.h. die Adresse des (abstrakten) Befehls, der unmittelbar nach dem Funktionsaufruf ausgeführt werden muß.
- `FP_OLD`, `EP_OLD` – Werte von `FP`/`EP` *vor* dem Aufruf; diese müssen ja nach Verlassen der Funktion restauriert werden.

Vereinfachung: der Rückgabewert der Funktion muß in eine Zelle passen.

2.7.2 Bestimmung der Adreßumgebung

In einem Programm gibt es zwei Arten von Variablen (Sichtbarkeitsbereiche):

- *globale* Variablen werden außerhalb von Funktionen definiert
- *lokale* Variablen werden innerhalb einer Funktion definiert.

Daher ordnet ρ den Variablen Paare $(\text{tag}, a) \in \{G, L\} \times I\!N$ zu.

Verschiedene Programmabschnitte haben verschiedene Adreßumgebungen.

Wir betrachten keine *statischen* lokalen Variablen.

Beispiel:

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

int ith(struct list *x, int i) {
  if (i == 1)
    return x->info;
  else
    return ith(x->next, i-1);
}

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

Adreßumgebungen:

außerhalb der Funktionen	innerhalb von `ith`	innerhalb von `main`
ρ_0 : `i` $\longmapsto (G,1)$	ρ_1 : `x` $\longmapsto (L,1)$	ρ_2 : `i` $\longmapsto (G,1)$
`l` $\longmapsto (G,2)$	`i` $\longmapsto (L,2)$	`l` $\longmapsto (G,2)$
`info` $\longmapsto (,0)$	`l` $\longmapsto (G,2)$	`k` $\longmapsto (L,1)$
`next` $\longmapsto (,1)$	`info` $\longmapsto (,0)$	`info` $\longmapsto (,0)$
`ith` $\longmapsto (G,l_{ith})$	`next` $\longmapsto (,1)$	`next` $\longmapsto (,1)$
`main` $\longmapsto (G,l_{main})$	`ith` $\longmapsto (G,l_{ith})$	`ith` $\longmapsto (G,l_{ith})$
$\vdots$	`main` $\longmapsto (G,l_{main})$	`main` $\longmapsto (G,l_{main})$
		$\vdots$

Beachte:

1. Durch neue Definitionen werden existierende Variablen gleichen Namens *verdeckt.* Die neuen Definitionen gelten aber nur innerhalb des Blocks, an dessen Anfang sie stehen.
2. Jede Funktion (oder genauer: jeder Block) hat eine eigene Adreßumgebung
3. ρ_0 heißt globale Adreßumgebung. Insbesondere ordnet ρ_0 den Funktionsnamen f das Paar (G, l_f) zu, wobei l_f die Anfangsadresse des Codes von f ist.

2.7.3 Betreten und Verlassen von Funktionen

Sei f die aktuelle Funktion; f rufe eine Funktion g auf. Wir müssen beim Betreten von g folgende Schritte ausführen:

1. Retten von `EP`, `FP`
2. Bestimmen der aktuellen Parameter von g
3. Inkrementieren von `FP`
4. Retten der Fortsetzungsadresse und Sprung zur Anfangsadresse von g
5. Inkrementieren von `EP`
6. Inkrementieren von `SP`, d.h. Anlegen der lokalen Variablen

Entsprechend müssen wir auch beim Verlassen von g handeln:

1. Zurücksetzen von `FP`, `EP` und `SP`
2. Rücksprung in den Code für f

Dazu benötigen wir einige neue abstrakte Befehle:

Befehl	Bedeutung

`mark`	`S[SP+2] = FP;`
	`S[SP+3] = EP;`
	`SP = SP + 4;`

`call` n	`FP = SP -` n `- 1;` // Platz für aktuelle Parameter `S[FP] = PC;` // Adresse nach dem `call` retten `PC = S[SP];` // Anfangsadresse von g vom Stack lesen `SP--;`

`enter` p	`EP = SP +` p`;` // p ist die maximale Tiefe des lokalen Kellers
	`if (EP >= NP) {`
	`  error("Stack overflow");`
	`  exit;`
	`}`

`alloc` q	`SP = SP +` q`;` // Platz für lokale Variablen reservieren

`return`	`SP = FP - 3;` // Rückgabewert
	`PC = S[FP];`
	`EP = S[FP-1];`
	`if (EP >= NP) {`
	`  error("Stack overflow");`
	`  exit;`
	`}`
	`FP = S[FP-2];`

Parameter-Übergabe:

- alle Parameter werden *by value* übergeben
- statt eines Feldes wird die Anfangsadresse des Feldes übergeben
- statt einer Funktion wird die Anfangsadresse, d.h. ein Pointer auf die Funktion übergeben
- Strukturen werden kopiert

Zum Kopieren von Strukturen benötigen wir einen neuen Befehl:

Befehl	Bedeutung

`move` n	`for(i=1; i<=n; i++)`
	`  S[SP+`n`-i] = S[S[SP]+`n`-i];`
	`SP = SP +` n `- 1;`

Damit erhalten wir für *Funktionsaufrufe*:

$$
\begin{array}{lll}
code_R\ g(e_1,\ldots,e_k,\underbrace{e_{k+1},\ldots,e_m}_{\text{Strukturen}})\ \rho & = & \texttt{mark} \\
 & & code_R\ e_1\ \rho \\
 & & \quad\vdots \\
 & & code_R\ e_k\ \rho \\
 & & code_L\ e_{k+1}\ \rho \\
 & & \texttt{move}\ n_{k+1} \\
 & & \quad\vdots \\
 & & code_L\ e_m\ \rho \\
 & & \texttt{move}\ n_m \\
 & & code_R\ g\ \rho \qquad // \text{ Sprungadresse } \ell' \\
 & & \texttt{call}\ N
\end{array}
$$

Dabei ist $n_j =$ Größe des Typs von e_j und $N = \sum_{i=1}^{m} n_i$ entspricht dem Platz für die aktuellen Parameter.

Für *Funktionsdefinitionen* gilt:

Sei $d = t\ \ f(\text{specs})\ \{$
    V_defs
    ss
$\}$

eine Funktionsdefinition. (specs: Übergabeparameter, V_defs: lokale Variablen)

$$
\begin{array}{lll}
code\ d\ \rho & = & \ell' : \texttt{enter}\ max \qquad // \text{ Einsprungadresse } \ell' \\
 & & \quad \texttt{alloc}\ ra \\
 & & \quad code\ ss\ \rho' \\
 & & \quad \texttt{return}
\end{array}
$$

Dabei ist ρ die globale Adreßumgebung; max ist die maximale Länge des lokalen Kellers, ra ist der Platz für die lokalen Variablen aus V_defs und ρ' ist die neue Adreßumgebung innerhalb von f.

2.7.4 Zugriff auf Variablen und formale Parameter, Rückgabe von Werten

Alle Zugriffe auf lokale Variablen und formale Parameter erfolgen *relativ* zum Framepointer FP. Dadurch ändert sich der Zugriff auf den Namen x.

Für $(t_j, j) = \rho(x)$ ist $code_L\ x\ \rho = \begin{cases} \texttt{loadc}\ j & \text{falls } t_j = G \\ \texttt{loadrc}\ j & \text{falls } t_j = L \end{cases}$

und

$code_R\ x\ \rho = \begin{cases} \texttt{loada}\ j & \text{falls } t_j = G \\ \texttt{loadr}\ j & \text{falls } t_j = L \end{cases}$

Befehl	Bedeutung
loadrc j	SP++;
	S[SP] = FP + j;
loadr j	SP++;
	S[SP] = S[FP+j];

Ansonsten ändert sich nichts, wir müssen nur zusätzlich noch Code für **`return`** e`;` erzeugen. Der Code entspricht einer Zuweisung an eine Variable mit Relativadresse -3 und der Aufgabe des Kellerrahmens.

code (**`return`** e`;`) ρ	=	$code_R\ e\ \rho$
		`loadrc` -3
		`store`
		`return`

Beispiel:

```
int fac(int x) {
  if (x == 0)
    return 1;
  else
    return x * fac(x-1);
}
```

Code für `fac`:

l_{fac} :	`enter` max
	`alloc` 0
	`loadr` 1
	`loadc` 0
	`eq`
	`jumpz` l_1
	`loadc` 1
	`loadrc` -3
	`store`
	`return`
	`jump` l_2
l_1 :	`loadr` 1
	`mark`
	`loadr` 1
	`loadc` 1
	`sub`
	`loadc` l_{fac}
	`call` 1
	`mul`
	`loadrc` -3
	`store`
	`return`
l_2 :	`return`

Beachte:

Die Befehle `jump` l_2 und das letzte `return` sind hier zwar überflüssig, werden aber gemäß Übersetzungsschema erzeugt. Auch das `alloc` 0 kann hier weggelassen werden.

2.7.5 Übersetzung eines Programms

Vor der Programmausführung werden `SP` etc. initialisiert:

`SP` = -1
`FP` = 0
`EP` = 0
`PC` = 0
`NP` = `MAXSTORE`

Sei $p \equiv$ V_defs $\text{F\_def}_1 \ldots \text{F\_def}_n$ ein Programm mit n Funktionen, wobei F_def_n `main` definiert.

Der Code für p

- enthält Code für alle Funktionsdefinitionen
- reserviert Platz für alle globalen Variablendefinitionen;
- springt den Code für `main` an;
- hält nach dem Ende von `main` an;

$$
\begin{array}{lrl}
\textit{code}\ p\,\rho & = & \texttt{enter}\ 4 + ra \\
 & & \texttt{alloc}\ ra \\
 & & \texttt{mark} \\
 & & \texttt{loadc}\ l_{main} \\
 & & \texttt{call}\ 0 \\
 & & \texttt{halt} \\
 & l_{f_1}: & \textit{code}\ \text{F\_def}_1\ \rho \\
 & \vdots & \quad\vdots \\
 & l_{f_n}: & \textit{code}\ \text{F\_def}_n\ \rho
\end{array}
$$

Dabei ist ra der Platz für die globalen Variablen aus V_defs und ρ die externe Adreßumgebung.

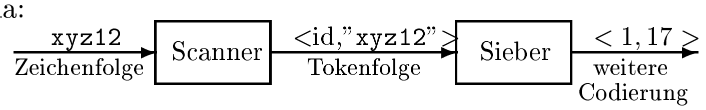
<u>Beachte:</u>

- $l_{f_n} = l_{main}$
- falls Variablen mit einem Anfangswert initialisiert werden sollen, muß der entsprechende Code hinter dem `alloc` eingefügt werden.

Kapitel 3

Lexikalische Analyse

Schema:



Der Scanner liest Zeichen von der Eingabe und generiert eine Folge von Tokens (Token = Paar aus Tokenklasse und String).

Aufgaben des Siebers:

- Entfernen irrelevanter Teile
- Aussondern von `pragma`–Anweisungen
- Umlenken des Eingabestroms auf andere Dateien
- Ersetzen der Tokens bestimmter Tokenklassen duch ihre "Bedeutung", d. h. Berechnen des Wertes von Konstanten, Codieren von Namen, usw.
- eventuelles Erkennen von Schlüsselwörtern

Im allgemeinen sind Scanner und Sieber keine getrennten Programme.

Beispiel:

Tokenklasse	Symbol
id	`pi`, `const`, `D2`, `x`, ...
num	`3.14159`, ...
literal	`"core dumped"`, ...
rel	`<`, `<=`, ...
const	`const`
for	`for`
if	`if`

Man erkennt, daß die Symbole einer Klasse i. a. eine reguläre Sprache bilden.

Problem: Wie generiert man aus Spezifikationen von regulären Ausdrücken effiziente Scanner für selbige ?

3.1 Reguläre Ausdrücke

Reguläre Sprachen können durch reguläre Ausrücke beschrieben werden.
Sei im Weiteren Σ die Menge der möglichen Eingabezeichen (das *Alphabet*).

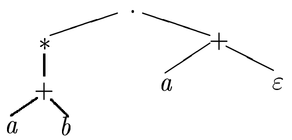
Definition:

- Die *Menge der regulären Ausdrücke* E_Σ über Σ ist die kleinste Menge E mit den Eigenschaften $\emptyset, \varepsilon \in E$,
  $a \in \Sigma \Rightarrow a \in E$,
  $e_1, e_2 \in E \Rightarrow (e_1 + e_2), (e_1 \cdot e_2), (e_1^*) \in E \quad (\text{'('}, \text{')'}, \text{'}\cdot\text{'}, \text{'+'} \notin \Sigma)$.
- Bezeichne $[e]$ die Sprache, die durch den regulären Ausdruck e beschrieben wird.
  Wir definieren $[\emptyset] = \emptyset$, $[\varepsilon] = \{\varepsilon\}$, $[a] = \{a\}$ für $a \in \Sigma$,
  $[(e_1+e_2)] = [e_1] \cup [e_2]$, $[(e_1 \cdot e_2)] = [e_1] \cdot [e_2]$ und $[(e_1)^*] = [e_1]^*$,
  wobei $L_1 \cdot L_2 = \{uv \mid u \in L_1 \text{ und } v \in L_2\}$
  und $L^* = \bigcup_{j \geq 0} L^j \quad (L^0 = \{\varepsilon\} \text{ und } L^j = L^{j-1} \cdot L \text{ für } j > 0)$.

Präzedenzen $* > \cdot > +$.

Beispiel:

$((a \cdot b)^* \cdot a \cdot b)$ repräsentiert $\{(ab)^n \mid n \geq 1\}$
$(a + b)$ repräsentiert $\{a, b\}$
Repräsentation als Baum: $((a + b)^* \cdot (a + \varepsilon))$



Definition: Ein regulärer Ausdruck $e \in E_\Sigma$ heißt $\emptyset$–frei, falls $\emptyset$ nicht in e vorkommt.

Satz: Zu jedem $e \in E_\Sigma$ kann in linearer Zeit ein Ausdruck $e' \in E_\Sigma$ gefunden werden mit $[e] = [e']$, wobei

- $[e] = \emptyset \;\Rightarrow\; e' = \emptyset$.
- $[e] \neq \emptyset \;\Rightarrow\; e'$ ist $\emptyset$–frei.

3.2 Endliche Automaten

Definition:

- Ein *nichtdeterministischer endlicher Automat* mit ε–Bewegungen (ε–NFA) $\mathcal{A}$ ist ein 5–Tupel $(Q, \Sigma, \delta, q_0, F)$.
  Dabei ist Q eine endliche Menge von Zuständen, Σ das Eingabealphabet, $q_0 \in Q$ der Anfangszustand, $F \subseteq Q$ eine Menge von Endzuständen und $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ die Übergangsrelation.
  Solch ein Automat kann als gerichteter kantenbeschrifteter Graph dargestellt werden.

- Das Wort $\pi = q_0(y_1, q_1)(y_2, q_2) \cdots (y_n, q_n) \in Q \cdot ((\Sigma \cup \{\varepsilon\}) \cdot Q)^*$, mit $w = y_1 y_2 \cdots y_n$ und $(q_{i-1}, y_i, q_i) \in \delta$ für $1 \leq i \leq n$ heißt eine *Berechnung* des Automaten $\mathcal{A}$ für ein Wort $w \in \Sigma^*$. Man sagt: π geht von q_0 nach q_n über.
  Im Graphen entspricht π ein mit w beschrifteter Pfad.
  Wir schreiben auch: $(q_0, w, q_n) \in \delta^*$, $\delta^* \subseteq Q \times \Sigma^* \times Q$, wobei $\delta^*(q, \varepsilon) = q$.
- Die *Sprache* $L(\mathcal{A})$ der von $\mathcal{A}$ akzeptierten Worte ist definiert durch $L(\mathcal{A}) = \{w \in \Sigma^* \mid \exists f \in F : (q_0, w, f) \in \delta^*\}$.

3.3 Konstruktion eines ε–NFA aus einem regulären Ausdruck

Satz: Zu jedem $\emptyset$–freien Ausdruck $e \in E_\Sigma$ kann in linearer Zeit ein ε–NFA $\tilde{\mathcal{A}}_e$ bestimmt werden mit $[e] = L(\tilde{\mathcal{A}}_e)$.

Beweis:

Idee: Vergleiche ein Eingabewort w mit e. Markiere mit '.', wohin man in e mit einem Präfix von w gelangen kann.

Konstruktion:

Die aus der Konstruktion resultierenden Zustände sind die Knoten im Syntaxbaum für e, erweitert um einen Punkt links oder rechts. Der Einfachheit halber bezeichnen wir solche Knoten n mit $[r]$, wobei r der Teilausdruck von e mit Wurzel n ist (nicht zu verwechseln mit der Sprache $[r]$).
Um verschiedene Teilausdrucksvorkommen zu unterscheiden, indizieren wir die Symbolvorkommen in e von links nach rechts.

Ist r ein Teilausdrucksvorkommen, dann sind $[.r]$ und $[r.]$ Zustände.
$[.e]$ ist Anfangszustand, $[e.]$ Endzustand.
Wir erhalten für Teilausdrücke r folgende Übergänge:

$[r] =$	$\rightsquigarrow$ Übergang
$[x_i]$	$([.x_i], x, [x_i.]) \in \delta, \;\; x \in \Sigma \cup \{\varepsilon\}$
$[r_1 + r_2]$	$\{\, ([.r], \varepsilon, [.r_i]),\; ([r_i.], \varepsilon, [r.]) \,\} \subseteq \delta, \; i \in \{1, 2\}$
$[r_1 \cdot r_2]$	$\{\, ([.r], \varepsilon, [.r_1]),\; ([r_1.], \varepsilon, [.r_2]), ([r_2.], \varepsilon, [r.]) \,\} \subseteq \delta$
$[r_1^*]$	$\{\, ([.r], \varepsilon, [r.]),\; ([.r], \varepsilon, [.r_1]),\; ([r_1.], \varepsilon, [.r_1]),\; ([r_1.], \varepsilon, [r.]) \,\} \subseteq \delta$

Behauptung: $([.r], w, [r.]) \in \delta^* \iff w \in [r]$.
Beweis: Induktion über die Teilausdrücke von r und Länge von w. □

Problem: Ein ε–NFA ist nicht sehr effizient zu implementieren.
Um uns einer effizienten Implementierung anzunähern, führen wir zunächst folgenden Schritt durch.

Definition: Sei $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ ein ε–NFA.
$\mathcal{A}$ heißt ε–frei bzw. *NFA*, falls für seine Übergangsrelation gilt: $\delta \subseteq Q \times \Sigma \times Q$

Satz: Zu jedem ε–NFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ kann in linearer Zeit ein NFA $\mathcal{A}' = (Q', \Sigma, \delta', q_0, F')$ konstruiert werden mit $L(\mathcal{A}) = L(\mathcal{A}')$.

Beweis:

Definiere die ε–*Hülle* δ_ε^* eines Zustands q als

$$\delta_\varepsilon^*(q) := \{p \in Q \mid (q, \varepsilon, p) \in \delta^* \text{ und } \exists a \in \Sigma, p' \in Q : (p, a, p') \in \delta\}.$$

Definiere: $Q' := \{q_0\} \cup \{p \in Q \mid \exists a \in \Sigma, q \in Q : (q, a, p) \in \delta\}$

$\delta' := \{(q', a, q) \mid \exists p \in \delta_\varepsilon^*(q') : (p, a, q) \in \delta,\ q' \in Q'\}$

$F' := \{p \in Q' \mid \exists f \in F : (p, \varepsilon, f) \in \delta^*\}$

Behauptung: $L(\mathcal{A}) = L(\mathcal{A}')$

Beweis: Induktion über die Länge einer Berechnung. □

3.4 Konstruktion eines NFA aus einem regulären Ausdruck

Im Folgenden soll nun ein Verfahren angegeben werden, daß die direkte Konstruktion eines NFA $\mathcal{A}_e = (Q', \Sigma, \delta', q_0, F')$ aus einem regulären Ausdruck e gestattet.

Dazu definieren wir die **Zustandsmenge** als $Q' := \{[.e]\} \cup \{[a_i.] \mid a \in e\}$.
Anfangszustand ist $q_0 := [.e]$.

Wir gehen bei der Konstruktion in vier Schritten vor:

Schritt 1:
Definition des Hilfsprädikats $\varepsilon[r] = \text{true} \iff \varepsilon \in [r]$.
Das Hilfsprädikat kann folgendermaßen "bottom-up" auf e berechnet werden:

$[r] =$	$\leadsto\ \varepsilon$
$[x_i]$	$\varepsilon[r] = \text{true} \iff x = \varepsilon$
$[r_1 + r_2]$	$\varepsilon[r] = \varepsilon[r_1] \vee \varepsilon[r_2]$
$[r_1 \cdot r_2]$	$\varepsilon[r] = \varepsilon[r_1] \wedge \varepsilon[r_2]$
$[r_1^*]$	$\varepsilon[r] = \text{true}$

Schritt 2:
Bottom-Up-Berechnung der Menge der *ersten* Blätter in $[r]$ (d. h. $\delta_\varepsilon^*([.r])$):

$[r] =$	$\leadsto\ F$
$[a_i]$, $a \in \Sigma$	$F[r] = \{[r]\}$
$[\varepsilon_i]$	$F[r] = \emptyset$
$[r_1 + r_2]$	$F[r] = F[r_1] \cup F[r_2]$
$[r_1 \cdot r_2]$	$F[r] = \begin{cases} F[r_1] \cup F[r_2], & \text{falls } \varepsilon[r_1] = \text{true} \\ F[r_1], & \text{sonst} \end{cases}$
$[r_1^*]$	$F[r] = F[r_1]$

Schritt 3:
Top-Down-Berechnung der Menge der *nächsten* Blätter nach $[r]$ (d. h. $\delta_\varepsilon^*([r.])$):

$[r] =$	$\rightsquigarrow N$
$[e]$	$N[r] = \emptyset$
$[r_1 + r_2]$	$N[r_1] = N[r],\ N[r_2] = N[r]$
$[r_1 \cdot r_2]$	$N[r_2] = N[r],\ N[r_1] = \begin{cases} F[r_2] \cup N[r], & \text{falls } \varepsilon[r_2] = \text{true} \\ F[r_2], & \text{sonst} \end{cases}$
$[r_1^*]$	$N[r_1] = F[r_1] \cup N[r]$

Damit erhalten wir die neue **Übergangsrelation**:

$$\delta' := \{([.e], a, [a_i.]) \mid [a_i] \in F[e]\} \cup \{([a_i.], b, [b_j.]) \mid [b_j] \in N[a_i]\}$$

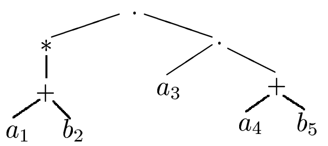
Schritt 4:
Bottom-Up-Berechnung der Menge der *letzten* Blätter
(d. h. $\{[a_i.] \mid ([a_i.], \varepsilon, [r.]) \in \delta^*\}$):

$[r] =$	$\rightsquigarrow L$
$[a_i]$, $a \in \Sigma$	$L[r] = \{[r]\}$
$[\varepsilon]$	$L[r] = \emptyset$
$[r_1 + r_2]$	$L[r] = L[r_1] \cup L[r_2]$
$[r_1 \cdot r_2]$	$L[r] = \begin{cases} L[r_1] \cup L[r_2], & \text{falls } \varepsilon[r_2] = \text{true} \\ L[r_2], & \text{sonst} \end{cases}$
$[r_1^*]$	$L[r] = L[r_1]$

Damit erhalten wir die neue **Endzustandsmenge**:

$$F' := \begin{cases} \{[.e]\} \cup \{[a_i.] \mid a_i \in L[e]\}, & \text{falls } \varepsilon[e] = \text{true} \\ \{[a_i.] \mid a_i \in L[e]\}, & \text{sonst} \end{cases}$$

Beispiel: $e = (a + b)^* \cdot (a \cdot (a + b))$



Die Zustandsmenge ist $Q' = \{\, [.e], [a_1.], [b_2.], [a_3.], [a_4.], [b_5.] \,\}$.

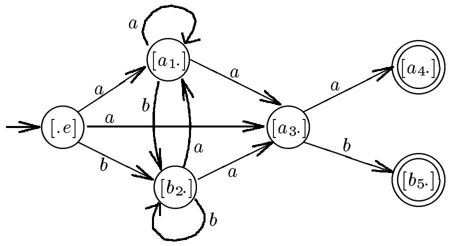
Wir erhalten für ε, F, N und L:

$[r] =$	$\varepsilon[r] =$	$F[r] =$	$N[r] =$	$L[r] =$
$[a_1]$	false	$\{[a_1]\}$	$\{[a_1], [b_2], [a_3]\}$	$\{[a_1]\}$
$[b_2]$	false	$\{[b_2]\}$	$\{[a_1], [b_2], [a_3]\}$	$\{[b_2]\}$
$[a_1 + b_2]$	false	$\{[a_1], [b_2]\}$	$\{[a_1], [b_2], [a_3]\}$	$\{[a_1], [b_2]\}$
$[(a_1 + b_2)^*]$	true	$\{[a_1], [b_2]\}$	$\{[a_3]\}$	$\{[a_1], [b_2]\}$
$[a_3]$	false	$\{[a_3]\}$	$\{[a_4], [b_5]\}$	$\{[a_3]\}$
$[a_4]$	false	$\{[a_4]\}$	$\emptyset$	$\{[a_4]\}$
$[b_5]$	false	$\{[b_5]\}$	$\emptyset$	$\{[b_5]\}$
$[a_4 + b_5]$	false	$\{[a_4], [b_5]\}$	$\emptyset$	$\{[a_4], [b_5]\}$
$[a_3 \cdot (a_4 + b_5)]$	false	$\{[a_3]\}$	$\emptyset$	$\{[a_4], [b_5]\}$
$[e]$	false	$\{[a_1], [b_2], [a_3]\}$	$\emptyset$	$\{[a_4], [b_5]\}$

Wir erhalten $\delta' = \{ ([.e], a, [a_1.]), ([.e], a, [a_3.]), ([.e], b, [b_2.]), ([a_1.], a, [a_1.]),$
$([a_1.], a, [a_3.]), ([a_1.], b, [b_2.]), ([b_2.], a, [a_1.]), ([b_2.], a, [a_3.]),$
$([b_2.], b, [b_2.]), ([a_3.], a, [a_4.]), ([a_3.], b, [b_5.]) \}$

und $F' = \{ [a_4.], [b_5.] \}$.

Graph des Automaten:



Bemerkung: Die gleiche NFA-Konstruktion findet sich auch in SGML wieder. Dort wird allerdings erwartet, daß $\mathcal{A}_e$ bereits deterministisch ist, was aber i. a. nicht der Fall ist!

Problem: Auch ein NFA ist nicht effizient implementierbar.

3.5 Konstruktion von DFAs aus NFAs

Definition: Ein NFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ heißt *deterministischer endlicher Automat (DFA)*, falls für alle $(q, a) \in Q \times \Sigma$ gilt: $\sharp\{p \in Q \mid (q, a, p) \in \delta\} \leq 1$.
Wir schreiben: $\delta(q, a) = p$ (falls ein Solches existiert), d. h. $\delta : Q \times \Sigma \to Q$ ist eine partielle Funktion.

Satz: Zu jedem NFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ existiert ein DFA $P(\mathcal{A})$ mit $L(\mathcal{A}) = L(P(\mathcal{A}))$.

Beweis: (Teilmengenkonstruktion)

Sei $P(\mathcal{A}) = (Q', \Sigma, \delta', q_0', F')$ mit $Q' \subseteq 2^Q$, $q_0' = \{q_0\}$, $F' = \{S \in Q' \mid S \cap F \neq \emptyset\}$,
$Q' = \bigcup_{j \geq 0} Q^{(j)}$ und $\delta' = \bigcup_{j \geq 0} \delta^{(j)}$.
Dabei ist $Q^{(0)} = \{\{q_0\}\}$, $\delta^{(0)} = \emptyset$ und für $j > 0$ gilt:
Sei $S \in Q^{(j-1)}$ und $a \in \Sigma$, dann definiere $S' = \{p \in Q \mid \exists q \in S : (q, a, p) \in \delta\}$.
Ist $S' \neq \emptyset$ und $S' \notin \bigcup_{0 \leq i < j} Q^{(i)}$, dann ist $S' \in Q^{(j)}$ und $(S, a, S') \in \delta^{(j)}$.

Algorithmus:

```
Q’ = ∅;  δ’ = ∅;  W = {{q0}};
while (W ≠ ∅) {
  S = extract(W);  // S aus Worklist entnehmen
  if (S ∉ Q’) {
    Q’ = Q’ ∪ {S};
    forall (a ∈ Σ ) do {
      S’ = {p | ∃q ∈S:(q,a,p)∈ δ};
      δ’ = δ’ ∪ {(S,a,S’)};
      W = W ∪ {S’};
      }
    }  /* if */
  }  /* while */
```

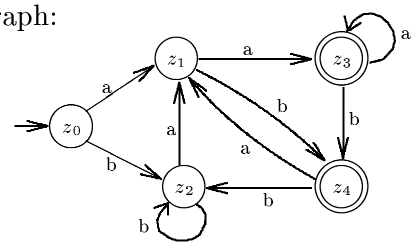
Beispiel: DFA für den NFA aus obigem Beispiel

Wir erhalten $Q' = \{z_0, z_1, z_2, z_3, z_4\}$, $q'_0 = z_0$, $F' = \{z_3, z_4\}$
mit $z_0 = \{[.e]\}$, $z_1 = \{[a_1.], [a_3.]\}$, $z_2 = \{[b_2.]\}$, $z_3 = \{[a_1.], [a_3.], [a_4.]\}$,
$z_4 = \{[b_2.], [b_5.]\}$, und $Q^{(0)} = \{z_0\}$, $Q^{(1)} = \{z_1, z_2\}$, $Q^{(2)} = \{z_3, z_4\}$.

Tabelle für δ':

δ'	a	b
z_0	z_1	z_2
z_1	z_3	z_4
z_2	z_1	z_2
z_3	z_3	z_4
z_4	z_1	z_2

Graph:



Bemerkungen:

- Der Automat $P(\mathcal{A})$ kann eventuell exponentiell viele Zustände haben (was aber i. a. selten der Fall ist).
- Zu $P(\mathcal{A}_e)$ sind die Endzustände "informativ". D. h.: die darin enthaltenen letzten Blätter geben Auskunft darüber, in welchen Teilausdrücken einer Vereinigung die Eingabe enthalten ist ($\exists\, [a_i] \in q' \in F'$ mit $[a_i] \in L[r]$).

3.6 Minimierung von deterministischen endlichen Automaten

Definition: Ein DFA $\mathcal{A}$ heißt *total*, falls $\delta : Q \times \Sigma \to Q$ eine totale Funktion ist.

Satz: Zu jedem totalen DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ existiert genau ein (bzgl. der Zustände) minimaler DFA $\mathcal{A}'$ mit $L(\mathcal{A}) = L(\mathcal{A}')$. $\mathcal{A}'$ kann in Zeit $O(\sharp Q \cdot \log \sharp Q)$ berechnet werden (mit $\sharp\Sigma$ konstant).

Beweis:
Sei o.B.d.A. jeder Zustand von $\mathcal{A}$ von q_0 aus erreichbar, d. h.:
$$\forall q \in Q : \exists w \in \Sigma^* : (q_0, w, q) \in \delta^*.$$

Wir definieren folgende Äquivalenzrelation $\equiv \subseteq Q \times Q$ auf Q durch:
$$p \equiv q \;:\Longleftrightarrow\; \forall w \in \Sigma^* : (\delta^*(p, w) \in F \;\Leftrightarrow\; \delta^*(q, w) \in F).$$

Sei $\mathcal{A}' = (Q', \Sigma, \delta', q'_0, F')$ definiert durch:
$Q' = Q/_{\equiv} = \{[q] \mid q \in Q\}$, $q'_0 = [q_0]$, $F' = \{[f] \mid f \in F\}$
und $\delta'([q], a) = \{\delta(p, a) \mid p \in [q]\} = [\delta(q, a)]$, falls $\delta(q, a)$ definiert.

Zu zeigen: 1. $L(\mathcal{A}) = L(\mathcal{A}')$.
2. $\mathcal{A}'$ ist minimal.
3. $\mathcal{A}'$ kann effizient berechnet werden.

zu 1.: intuitiv klar

zu 2.:
Sei $L = L(\mathcal{A})$.
Dann definiert L eine rechtsinvariante Äquivalenzrelation $\equiv_L \subseteq \Sigma^* \times \Sigma^*$ durch:
$$w \equiv_L v \;:\Longleftrightarrow\; \forall y \in \Sigma^* : (wy \in L \;\Leftrightarrow\; vy \in L).$$

Sei $\mathcal{A}_L = (Q_L, \Sigma, \delta_L, q_{0,L}, F_L)$ definiert durch:
$Q_L = \Sigma^*/_{\equiv_L} = \{[w]_L \mid w \in \Sigma^*\}$, $q_{0,L} = [\varepsilon]_L$, $F_L = \{[w]_L \mid w \in L\}$
und $\delta_L([w]_L, a) = [wa]_L$ (wohldefiniert).

Es gilt:
- L regulär $\iff Q_L$ endlich.
- $L(\mathcal{A}_L) = L(\mathcal{A})$.
- $\mathcal{A}_L$ ist minimal bezüglich der Anzahl der Zustände.
- $w \equiv_L v \iff \delta^*(q_0, w) = \delta^*(q_0, v)$
  $\implies [w]_L \mapsto [\delta^*(q_0, w)]$ ist Bijektion.

Da alle Zustände von $\mathcal{A}$ von q_0 aus erreichbar sind, folgt, daß die Konstruktion $\mathcal{A}'$ genau $\mathcal{A}_L$ liefert (bis auf Umbenennung der Zustände).

zu 3.:
Berechnung von Q':

Idee: Wir verwalten Äquivalenzen von Zuständen.
Jede Äquivalenzrelation definiert eine Partition Π auf Q.

Notation für Partitionen Π auf Q:

- Eine Partition Π_1 heißt *Verfeinerung* von Π_2 ($\Pi_1 \leq \Pi_2$), falls gilt:
  $\forall \pi_1 \in \Pi_1 : \exists \pi_2 \in \Pi_2 : \pi_1 \subseteq \pi_2$.
- Π heißt *invariant* unter δ, falls δ die Äquivalenz von Zuständen erhält, d. h.:
  $\forall \pi \in \Pi, a \in \Sigma : \exists \pi' \in \Pi : \delta'(\pi, a) = \{\delta(q, a) \mid q \in \pi\} \subseteq \pi'$.
  Insbesondere ist Q' invariant unter δ.

Hilfssatz: Sei $Q \neq F \neq \emptyset$ und Π eine Partition auf Q mit $Q' \leq \Pi \leq \{Q \setminus F, F\}$.
Ist Π invariant unter δ, dann ist $\Pi = Q'$.

Beweis: Sei $\pi \in \Pi$, $q_1, q_2 \in \pi$ und $w \in \Sigma^*$ beliebig.
Zu zeigen: $\delta^*(q_1, w) \in F \Leftrightarrow \delta^*(q_2, w) \in F$ (d. h. $q_1 \equiv q_2$)
Denn: Da Π invariant unter δ ist, folgt: $\exists \pi' \in \Pi : \delta^*(q_i, w) \in \delta'^*(\pi, w) \subseteq \pi'$, $i = 1, 2$.
Mit $\Pi \leq \{Q \setminus F, F\}$ folgt $\pi' \subseteq F$ oder $\pi' \cap F = \emptyset$. $\implies$ Beh. $\Box$

Idee des Algorithmus':

(1) $F = \emptyset \implies Q' = \{Q\}$, $F' = \emptyset$

(2) $Q \setminus F = \emptyset \implies Q' = \{Q\}$, $F' = \{Q\}$

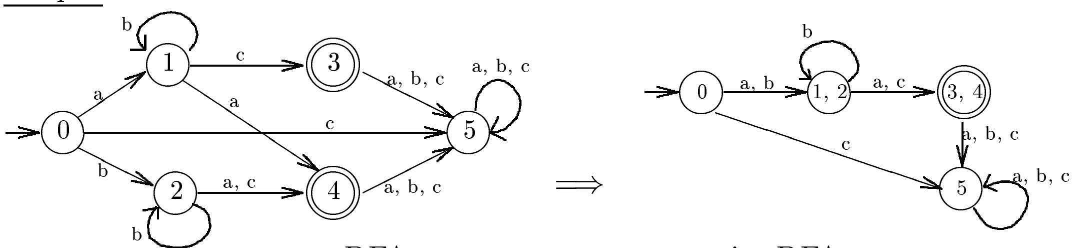
(3) $Q \setminus F \neq \emptyset$ und $F \neq \emptyset$:
Wir starten mit $\Pi_0 = \{Q \setminus F, F\}$ und verfeinern solange, bis die Partition invariant unter δ ist.
Sei Π_{j-1} bereits konstruiert und $Q' \lneqq \Pi_{j-1} \leq \Pi_0$.
Verfeinerungsschritt: Ist Π_{j-1} nicht invariant unter δ, dann gilt:
$$\exists \pi, \pi' \in \Pi_{j-1}, a \in \Sigma : \ \delta'(\pi, a) \cap \pi' \neq \emptyset \quad (1)$$
$$\text{und } \delta'(\pi, a) \setminus \pi' \neq \emptyset \quad (\Leftrightarrow \delta'(\pi, a) \not\subseteq \pi') \quad (2)\ .$$
D. h.: Einige Übergänge unter a führen aus der Klasse π in die Klasse π' (1), allerdings gilt dies nicht für alle Übergänge unter a (2).

Wir spalten π auf in π_1, π_2 mit $\pi = \pi_1 \cup \pi_2$ und $\pi_1 \cap \pi_2 = \emptyset$:

$$\pi_1 = \{q \in \pi \mid \delta(q, a) \in \pi',\, a \in \Sigma\},$$
$$\pi_2 = \pi \setminus \pi_1,$$
$$\Pi_j = (\Pi_{j-1} \setminus \{\pi\}) \cup \{\pi_1, \pi_2\}.$$

Dann gilt: $Q' \leq \Pi_j \lneqq \Pi_{j-1}$. Folglich ist $Q' = \Pi_k$ für ein $k \leq \sharp Q - 2$.
Kosten einer Verfeinerung $\mathrm{O}(|\mathcal{A}|) \implies$ Gesamtlaufzeit $\mathrm{O}(|\mathcal{A}| \cdot \sharp Q)$ (worst–case), wobei $(|\mathcal{A}| = \sharp\delta = \sharp Q \cdot \sharp\Sigma)$ die Größe des Automaten ist ($\implies \mathrm{O}(\sharp Q^2)$).

Beispiel



DFA min. DFA

Partitionen: $\Pi_0 = \{\{0, 1, 2, 5\}, \{3, 4\}\}$,
$\Pi_1 = \{\{0, 5\}, \{1, 2\}, \{3, 4\}\}$,
$\Pi_2 = \{\{0\}, \{5\}, \{1, 2\}, \{3, 4\}\} = Q/_{\equiv}$

Algorithmus:

```
Π = {F, Q\F};  W =  Π;
infl(F) = ∅;  infl(Q\F) = ∅;
while (W ≠ ∅) {
  π = extract(W);  // π aus Worklist entnehmen
  forall (a ∈ Σ) do
    if (∃ π' ∈ Π : δ(π, a) ⊆ π' )
      infl(π') = infl(π') ∪ infl(π);
    else {     // aufteilen
      π1 = {q ∈ π | δ(q, a) ∈ π'};
      π2 = π \ π1;
      Π = Π \ {π} ∪ {π1, π2};
      infl(π1) = ∅;  infl(π2) = ∅;
      forall (τ ∈ infl(π)) do
        if (τ ∈ Π )
          W = W ∪ {τ};
      break;
    }  /* if */
}  /* while */
```

3.7 Implementierung eines DFA

Matrix: Repräsentiere die Übergangsfunktion δ des DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ als zweidimensionale Matrix.

Dazu codieren wir die Zustandsmenge Q und das Alphabet Σ als Integerzahlen aus den Intervallen $[0, \sharp Q - 1]$ bzw. $[0, \sharp\Sigma - 1]$. Der Nachfolgezustand nach Einlesens eines Zeichens wird durch Nachschlagen in der Matrix bestimmt.
Problem: Da die Matrix i. a. dünn besetzt ist, verschwenden wir Platz.

Beispiel:

Tabelle für δ:

	a	b	c
q_0	q_1	–	–
q_1	–	q_2	q_1
q_2	–	–	q_3
q_3	–	–	–

Matrix M:

	0	1	2
0	1	–	–
1	–	2	1
2	–	–	3
3	–	–	–

Komprimierte Matrix: Wir gehen ähnlich vor wie unter 1, legen aber die Zeilen der Matrix übereinander, in der Hoffnung, daß die nächste Matrixzeile in die "Löcher" ihrer Vorgängerzeile paßt. Ist dies nicht der Fall, so verschieben wir sie um einen Offset $\mathit{off}[q]$ für eine Zeile (Zustand) q. Für unser Beispiel von oben erhalten wir folgende Matrix M:

	0	1	2	3
M	1	2	1	3

mit $\mathit{off}[0] = 0$, $\mathit{off}[1] = 0$ und $\mathit{off}[2] = 1$.

Der Nachfolgezustand von q unter a ist dann $M[a + \mathit{off}[q]]$.

Problem: Plötzlich gibt es Übergänge, die nicht zulässig sind (etwa von 0 unter b nach 2). Wir benötigen also ein weiteres Feld V (valid), um uns zu merken für welchen Zustand der Eintrag in M gültig ist:

	0	1	2	3
V	0	1	1	2

.

3.8 Spezifikation von Symbolklassen

Reguläre Ausdrücke erweisen sich oft als zu unhandlich.
Für eine Real–Konstante erhalten wir etwa den Ausdruck:
$(\varepsilon + \texttt{+} + \texttt{-}) \cdot (\texttt{0}+\texttt{1}+\cdots+\texttt{9})^* \cdot \texttt{.} \cdot (\texttt{0}+\texttt{1}+\cdots+\texttt{9})^+ \cdot (\varepsilon + \texttt{E} \cdot (\varepsilon + \texttt{+} + \texttt{-}) \cdot (\texttt{0}+\texttt{1}+\cdots+\texttt{9}) \cdot (\texttt{0}+\texttt{1}+\cdots+\texttt{9}))$.

Verbesserungen:

1. Da Alphabete üblicherweise eine natürliche Ordnung besitzen, können wir *Zeichenintervalle* angeben, wie etwa $[\texttt{a} - \texttt{z}] \equiv (\texttt{a} + \texttt{b} + \cdots + \texttt{z})$.
   Wir können auch Zeichenklassen definieren, wie etwa $bu = ([\texttt{a} - \texttt{z}] + [\texttt{A} - \texttt{Z}])$ für Buchstaben oder $zi = [\texttt{0} - \texttt{9}]$ für Ziffern.

2. *Sequenzen* regulärer Ausdrücke erlauben uns die Einführung von Namen für wiederkehrende Teilausdrücke und ermöglichen die Behandlung mehrerer Zeichenklassen.

Sei $A_1 = e_1, \ldots, A_n = e_n$ eine Folge von Definitionen für $A_1, \ldots, A_n$ mit $e_i \in E_{\Sigma \cup \{A_1, \ldots, A_n\}}$.
Die A_i dienen als Platzhalter; das System sollte **nicht** rekursiv sein (etwa sollte e_1 nicht A_1 enthalten), d. h.: für alle A_i in e_j gilt $i < j$.
Sei $\bar{e}_i$ $(1 \leq i \leq n)$ definiert durch $\bar{e}_1 = e_1$, $\bar{e}_2 = e_2[\bar{e}_1/A_1]$, ..., $\bar{e}_j = e_j[\bar{e}_1/A_1, \ldots, \bar{e}_{j-1}/A_{j-1}]$.
Dann repräsentiert A_j gerade $[\bar{e}_j]$. ($e[x/y]$ bedeutet: ersetze alle Vorkommen von y durch x).
Die A_i können als Makros aufgefaßt werden, die textuell ersetzt werden.

3.9 Generierung eines Scanners

1) Die definierten Zeichenklassen müssen nicht disjunkt sein. (z. B. $bu = [\mathsf{a} - \mathsf{z}]$ und $buzi = [\mathsf{a} - \mathsf{z}\,\mathsf{0} - \mathsf{9}]$). Deshalb muß der Generator die Klassen in maximale disjunkte Teilklassen aufteilen (hier bu und $buzi \setminus bu$).

2) Für eine Folge von regulären Definitionen ist das maximale Präfix $w \in \Sigma^*$ der Eingabe gesucht, sodaß $w \in [\bar{e}_j]$ mit $j = \min\{i \mid w \in [\bar{e}_i]\}$.

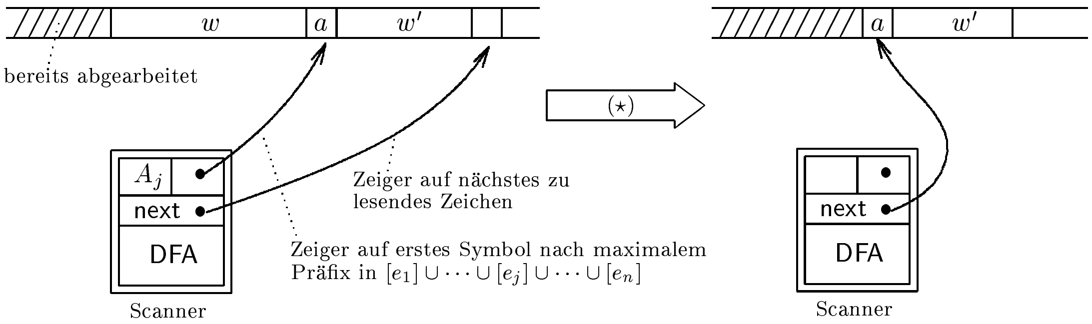
Methode:

a) Sei $e = \bar{e}_1 + \bar{e}_2 + \cdots + \bar{e}_n$ und $L_i = L[\bar{e}_i]$ für $1 \le i \le n$ (Mengen der letzten Blätter). Konstruiere zu e den DFA $P(\mathcal{A}_e)$.

b) Sei F die Menge der Endzustände von $P(\mathcal{A}_e)$.
Definiere F_j induktiv durch: $F_1 = \{S \in F \mid S \cap L_1 \ne \emptyset\}$ und
$$F_j = \{S \in F \setminus \bigcup_{i=1}^{j-1} F_i \mid S \cap L_j \ne \emptyset\} \text{ für } j > 1.$$
Setze $\Pi = \{F_1, \ldots, F_n, Q \setminus F\}$, wobei $Q \setminus F \ne \emptyset$ sei.
Dies garantiert, daß Teilautomaten, welche Teilausdrücke $\bar{e}_i$ von regulären Teilausdrücken $\bar{e}_j$ mit $i < j$ berechnen, nicht wieder zusammengefaßt werden!
Bestimme nun die gröbste Verfeinerung von Π, die unter δ invariant ist. Dies liefert einen minimalen DFA, der $F_1, \ldots, F_n$ unterscheidet.

c) Stelle den DFA aus b) mittels komprimierter Matrizen dar.

Arbeitweise des Scanners:

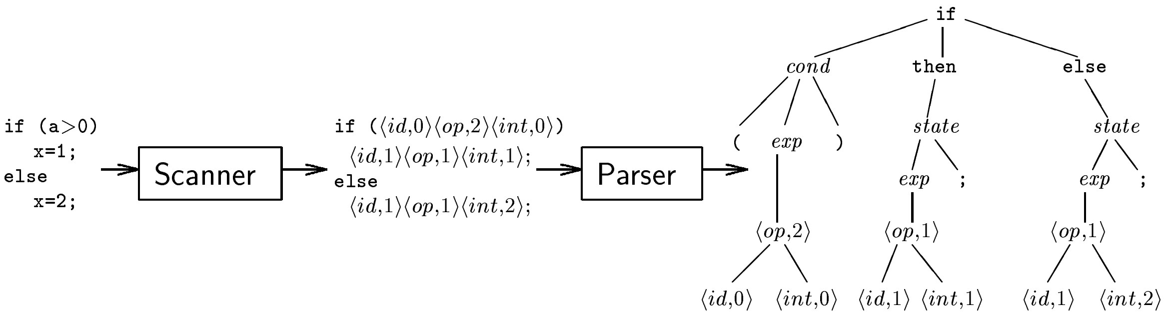


(⋆) Falls waw' nicht mehr fortsetzbar ist, übergeben wir $\langle A_j, w \rangle$ an den Sieber.
next wird auf a zurückgesetzt.

Kapitel 4

Syntaktische Analyse

Der Scanner liest die Zeichen des Programmtexts von der Eingabe und generiert eine Folge von Tokens.



Der Parser liefert eine interne Repräsentation des Programms in Form eines *abstrakten Syntaxbaums*.

Beschreibungsmittel für die Syntax von Programmiersprachen sind die *kontextfreien Grammatiken* (KfG bzw. CfG).
Ein Parsergenerator liefert dafür einen Kellerautomaten, erweitert um Aktionen, wie etwa zur Konstruktion eines abstrakten Syntaxbaums.

4.1 Theoretische Grundlagen

4.1.1 Kontextfreie Grammatiken

Definition: Eine *kontextfreien Grammatik* ist ein 4–Tupel $G = (N, T, P, S)$, wobei N ein endliches Alphabet von *Nichtterminalen* (syntaktische Einheiten), T ein endliches Alphabet von *Terminalen* (Symbolklassen), $S \in N$ das *Startsymbol* und P eine endliche Menge von *Produktionen* (Regeln) ist.
Die Produktionen haben die Form $A \to \alpha$ mit $A \in N$ und $\alpha \in (N \cup T)^*$.

Wir treffen folgende Konventionen: $\langle exp\rangle, A, B, C \in N$
$X, Y, Z \in N \cup T =: V$
$a, b, c, \ldots, \texttt{if}, \texttt{case}, \texttt{int}, \texttt{+}, \ldots \in T$
$u, v, w, \ldots \in T^*$
$\alpha, \beta, \gamma, \ldots \in V^*$
$A \to \alpha_1 \mid \cdots \mid \alpha_k$ steht für $A \to \alpha_1, \ldots, A \to \alpha_k$
$S, S', S_1, \ldots$ Startsymbole

Beispiele für Grammatiken:

1)
$$\begin{aligned}
\langle state\rangle &\to \langle if\rangle \mid \langle while\rangle \mid \langle exp\rangle \;\texttt{;} \mid \cdots \\
\langle if\rangle &\to \texttt{if}\ \langle cond\rangle\ \langle state\rangle\ \texttt{else}\ \langle state\rangle \\
\langle while\rangle &\to \texttt{while}\ \langle cond\rangle\ \langle state\rangle \\
\langle cond\rangle &\to \texttt{(}\ \langle exp\rangle\ \texttt{)} \\
\langle exp\rangle &\to int \mid \langle Rexp\rangle \mid \langle Lexp\rangle \texttt{=} \langle Rexp\rangle \mid \cdots \\
\langle Lexp\rangle &\to id \mid \cdots
\end{aligned}$$

2) Zwei Grammatiken für $E := \langle Rexp\rangle$:

a) $E \;\to\; int \mid id \mid E+E \mid E*E \mid (E)$

b)
$$\begin{aligned}
E &\to E+T \mid T && (= \ (E,0) \mid (E,1)) \\
T &\to T*F \mid F && (= \ (T,0) \mid (T,1)) \\
F &\to (E) \mid id \mid int && (= \ (F,0) \mid (F,1) \mid (F,2))
\end{aligned}$$

Die hinter jeder Regel in Klammern angegebene alternative Schreibweise (A,j) besagt, daß es sich um die j–te Regel mit linker Seite A handelt.

Definition: Sei $G = (N, T, P, S)$.
Für $p = (A,j) = A \to \gamma \in P$ ist die Relation $\xrightarrow{p}_G \subseteq V^* \times V^*$ definiert durch:

$$\alpha \xrightarrow{p}_G \beta \quad :\Longleftrightarrow \quad \alpha = \alpha_1 A \alpha_2 \text{ und } \beta = \alpha_1 \gamma \alpha_2$$

d. h. β ist in einem Schritt aus α mittels p ableitbar.
Wir definieren $\longrightarrow_G := \bigcup\limits_{p \in P} \xrightarrow{p}_G$ und $\xrightarrow{*}_G$ als reflexive und transitive Hülle von $\longrightarrow_G$.

Bemerkungen:

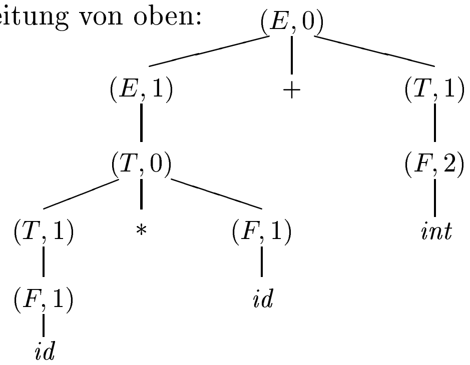
- Das "G" in $\longrightarrow_G$ wird meistens weggelassen.
- Wir unterscheiden zwei spezielle Regelanwendungen:
  Linksableitung:
  Das am weitesten links stehende Nichtterminal wird jeweils ersetzt ($\longrightarrow_{G,L}$).
  Rechtsableitung:
  Das am weitesten rechts stehende Nichtterminal wird jeweils ersetzt ($\longrightarrow_{G,R}$).

Für das Beispiel 2) b) erhalten wir als mögliche Ableitung:

$$E \overset{(E,0)}{\longrightarrow} E+T \overset{(E,1)}{\longrightarrow} T+T \overset{(T,0)}{\longrightarrow} T*F+T \overset{(F,1)}{\longrightarrow} T*id+T \overset{(T,1)}{\longrightarrow} F*id+T$$
$$\overset{(F,1)}{\longrightarrow} id*id+T \overset{(T,1)}{\longrightarrow} id*id+F \overset{(F,2)}{\longrightarrow} id*id+int$$

4.1.2 Ableitungsbäume

Man erreicht durch Verwendung “sortierter” *Ableitungsbäume* eine bessere Repräsentation.
Die inneren Knoten dieser Bäume stellen die bei der Ableitung verwendeten Regeln dar.
Die abgeleiteten Terminale sind die Blätter der Bäume.
Konkateniert man die Folge der Blätter von links nach rechts, erhält man das abgeleitete Wort.
Ableitungsbaum für die Ableitung von oben:



Das Nichtterminal in der Wurzel des Baums bezeichnen wir als *Sorte* des Baums (im Beisp. E).

Mit folgender Definition formalisieren wir den Begriff des Ableitungsbaums.

Definition: Die Menge $\mathcal{T}_G^X$ der Ableitungsbäume der Sorte $X \in V \cup \{\varepsilon\}$ für eine Grammatik G ist das kleinste System von Mengen $\mathcal{T}^X$, für das gilt:

(1) Ist $X \in T \cup \{\varepsilon\}$, dann ist $X \in \mathcal{T}^X$.

(2) Für $p = A \to X_1 \cdots X_n \in P$ mit $n \geq 1$ und $X_i \in V$ ist $p(t_1, \ldots, t_n) \in \mathcal{T}^A$ für $t_i \in \mathcal{T}^{X_i}$.

(3) Für $p = A \to \varepsilon \in P$ ist $p(\varepsilon) \in \mathcal{T}^A$.

Partielle Bäume erhält man, indem man unter (1) auch Nichtterminale $X \in N$ zuläßt, d. h. $A \in \mathcal{T}^A$ für $A \in N$.

Für $t \in \mathcal{T}_G^X$ definieren wir eine Folge von Blättern als $yield(t)$, wobei

- $yield(t) = X$, falls $X \in T \cup \{\varepsilon\}$,
- $yield(t) = yield(t_1) \cdots yield(t_n)$, falls $t = p(t_1, \ldots, t_n)$.

(Für partielle Bäume können wir eine entsprechende Definition mit Nichtterminalen angeben.)

Beachte:

Für $t \in \mathcal{T}_G^X$ entspricht $X \xrightarrow{*}_{G,L} w$ einem Links–Rechts–Präorderdurchlauf im Ableitungsbaum t.
$X \xrightarrow{*}_{G,R} w$ entspricht einem einem Rechts–Links–Präorderdurchlauf in t (äquivalent zu einem reversen Links–Rechts–Postorderdurchlauf).

Für unser Beispiel von oben erhalten wir:
Links–Rechts–Präorderdurchlauf: $(E, 0)$, $(E, 1)$, $(T, 0)$, $(T, 1)$, $(F, 1)$, $(F, 1)$, $(T, 1)$, $(F, 2)$
Rechts–Links–Präorderdurchlauf: $(E, 0)$, $(T, 1)$, $(F, 2)$, $(E, 1)$, $(T, 0)$, $(F, 1)$, $(T, 1)$, $(F, 1)$
(Links–Rechts–Postorderdurchlauf: $(F, 1)$, $(T, 1)$, $(F, 1)$, $(T, 0)$, $(E, 1)$, $(F, 2)$, $(T, 1)$, $(E, 0)$)

Definition: G ist genau dann eindeutig, wenn für alle $w \in T^*$ gilt: $\sharp\{t \in \mathcal{T}_G^S \mid \mathit{yield}(t) = w\} \leq 1$.

Offenbar ist die Grammatik aus Beispiel 2) a) nicht eindeutig, wohingegen die Grammatik 2) b) eindeutig ist.

Kontextfreie Grammatiken für Programmiersprachen sollten stets eindeutig sein !

Definition: Sei $G = (N, T, P, S)$ eine Grammatik.
Dann ist die durch G erzeugte Sprache $L(G)$ definiert als
$L(G) = \{w \in T^* \mid S \xrightarrow{*}_G w\} = \{\mathit{yield}(t) \mid t \in \mathcal{T}_G^S\}$.

4.1.3 Reduzierte kontextfreie Grammatiken

Definition: Sei $G = (N, T, P, S)$ eine Grammatik.
Ein Nichtterminal $A \in N$ heißt *produktiv*, falls es ein Wort $w \in T^*$ gibt mit $A \xrightarrow{*}_G w$ ($\Longleftrightarrow$ $\mathcal{T}_G^A \neq \emptyset$).
Ein Nichtterminal $A \in N$ heißt *erreichbar*, falls es Worte $\alpha, \beta \in V^*$ gibt mit $S \xrightarrow{*}_G \alpha A \beta$ ($\Longleftrightarrow$ es gibt einen partiellen Ableitungsbaum t der Sorte S mit $\mathit{yield}(t) = \alpha A \beta$).
G heißt *reduziert*, falls alle Nichtterminale $A \in N$ produktiv und erreichbar sind.

Satz: Zu jeder KfG $G = (N, T, P, S)$ mit $L(G) \neq \emptyset$ kann in linearer Zeit eine reduzierte KfG $G' = (N', T, P', S)$ konstruiert werden mit $L(G) = L(G')$ und $N' \subseteq N$, $P' \subseteq P$.

Beweis:

Sei $N_1 = \{A \in N \mid A \text{ ist produktiv}\}$ (nach Voraussetzung ist $S \in N_1$),
$P_1 = \{A \to \alpha \in P \mid \alpha \in (T \cup N_1)^*\}$, $G_1 = (N_1, T, P_1, S)$.
Es gilt $L(G_1) = L(G)$ und alle $A \in N_1$ sind produktiv.

Sei $N_2 = \{A \in N_1 \mid A \text{ ist erreichbar bzgl. } G_1\}$ (insbesondere ist $S \in N_2$),
$P_2 = \{A \to \alpha \in P_1 \mid A \in N_2\}$, $G' := (N_2, T, P_2, S)$.
Es gilt $L(G') = L(G_1)$ und alle $A \in N_2$ sind erreichbar.

Dieses Verfahren ist linear, falls N_1 und N_2 in linearer Zeit berechnet werden können.

Berechnung der produktiven Nichtterminale:
Wir stellen fest, daß N_1 die kleinste Menge $\tilde{N}$ ist mit $A \to \alpha \in P$.
D. h. für $\alpha \in (\tilde{N} \cup T)^*$ muß also auch $A \in \tilde{N}$ sein.

Definiere $y_A \in \{0, 1\}$, $A \in N$ durch: $y_A = 1 \iff A$ produktiv.
Wir stellen folgende Beziehung auf:

$$y_A = \bigvee_{A \to \alpha \in P} y_{A_1} \wedge \cdots \wedge y_{A_k}, \quad \text{wobei } \alpha = w_0 A_1 w_1 \cdots w_{k-1} A_k w_k,\ w_i \in T^*.$$

Beachte: $y_A = 1$ für $A \to w$, $w \in T^*$.

Im Beispiel: Für $P = \{A \to Ba \mid b,\ B \to bB\}$ erhalten wir das lineare Gleichungssystem
$$y_A = y_B \vee 1$$
$$y_B = y_B.$$

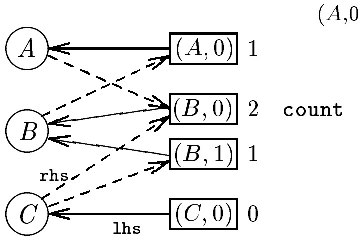
Mögliche Lösungen sind $y_A = 1$, $y_B = 0$ und $y_A = 1$, $y_B = 1$.

Problem: Wie löst man ein solches Gleichungssystem ?
Nur die erste Lösung ist die richtige Lösung für diese Problemstellung.
Später werden wir feststellen, daß wir die kleinste Lösung des LGS benötigen !

Wir geben nun noch ein Verfahren zur effizienten Berechnung von N_1 an.
Die Idee ist, für jede Regel der Grammatik einen Zähler für die Anzahl der eventuell vorkommenden nichtproduktiven Terminale in der rechten Seite (rhs) einer Regel zu verwalten.
Ist dieser Zähler nach Beenden der Berechnung 0, dann ist die linke Seite der betreffenden Regel (lhs) produktiv.

Für eine Grammatik mit Produktionen $\{A \rightarrow \underbrace{aBB}_{(A,0)},\ B \rightarrow \underbrace{AbC}_{(B,0)} \mid \underbrace{bC}_{(B,1)},\ C \rightarrow \underbrace{ab}_{(C,0)}\}$ erhalten wir

die Repräsentation:



Im folgenden Algorithmus bezeichnet der Typ `N` Variablen vom Typ ‘Nichtterminal’ und `SetOfN` Variablen vom Typ ‘Menge von Nichtterminalen’.
Analog ist `P` der Typ ‘Produktion’ und `SetOfP` der Typ ‘Menge von Produktionen’.
Mit `a[T]` bezeichnen wir Felder, die mit Werten vom Typ `T` indiziert werden.
extract`(M)` liefert ein Element aus der Menge `M` und entfernt es anschließend aus `M`.

Algorithmus:

```
SetOfN  W = ∅;       // Worklist
SetOfN  R = ∅;       // Resultatmenge mit prod. Nichtterminalen
N       lhs[P];      // linke Seiten der Regeln
int     count[P];    // Zähler für Nichtterminale
SetOfP  rhs[N];      // Nichtterminal–Vorkommen

void init (P r, V* α) {
  switch (α) {
    case ε: if (count[r] == 0)  W = W ∪ { lhs[r] };
            break;
    case a α': init(r,α');
              break;
    case A α' : if (r ∉ rhs[A]) {
                  count[r]++;
                  rhs[A] = rhs[A] ∪ {r};
                  }
                init(r,α');
    }
  }
```

```
/* Initialisierung */
forall (A ∈ N) do rhs[A] = ∅;
forall (r ≡ A → α ∈ P ) do {
  lhs[r] = A;
  count[r] = 0;
  init (r,α);
  }
/* eigentlicher Algorithmus */
while (W ≠ ∅) {
  A = extract(W);   // entferne Element aus Worklist
  if (A ∉ R) {
    R = R ∪ {A};
    while (rhs[A] ≠ ∅ ) {
      r = extract(rhs[A]);
      count[r]--;
      if (count[r] == 0)  W = W ∪ { lhs[r] };
      }  /* while */
    }  /* if */
  }  /* while */
```

Zur Korrektheit des Algorithmus':

Seien $\mathtt{R}_i$, $\mathtt{W}_i$ und $\mathtt{count}_i$ die Werte der entsprechenden Variablen nach dem i–ten Durchlauf der Schleife. Dann gilt:

1) $\mathtt{R}_i \subseteq \mathtt{R}_{i+1}$

2) $\forall \mathtt{r} \equiv A \to \alpha \in P : \alpha \in (T \cup \mathtt{R}_i)^* \iff A \in \mathtt{R}_i \cup \mathtt{W}_i$.
   Ist die Worklist leer, dann gilt sogar: $\alpha \in (T \cup \mathtt{R}_i)^* \iff A \in \mathtt{R}_i$.

3) $\mathtt{R}_i \subseteq N_1$.
   Aus 2) und 3) folgt, daß — sofern der Algorithmus nach $k-1$ Iterationen terminiert — $\mathtt{R}_k = N_1$.

4) Es gibt maximal $\sharp P$ viele Einfügungen eines Nichtterminals in die Worklist
   $\Longrightarrow\ i \leq \sharp P + 1$. D. h. der Algorithmus terminiert immer !

Sei $|G| = \sum\limits_{A \to \alpha \in P} (1 + |\alpha|) + \sharp N$.

Zur Komplexität des Algorithmus':

1) Die Initialisierung benötigt $\mathrm{O}(|G|)$ Schritte.

2) Die äußere `while`–Schleife benötigt
   – $\mathrm{O}(\sharp P)$ Einfügungen/Extraktionen von Nichtterminalen;
   – aufgrund jedes `A`'s im `if`–Statement insgesamt $\mathrm{O}(|\mathtt{rhs[A]}|)$, wobei $\sum\limits_{A \in N} \mathrm{O}(|\mathtt{rhs[A]}|) \leq \sum\limits_{A \to \alpha \in P} \mathrm{O}(|\alpha|) \leq \mathrm{O}(|G|)$.

Insgesamt benötigt der Algorithmus also $\mathrm{O}(|G|)$ Schritte.

Berechnung der erreichbaren Nichtterminale:
Sei $D = (N_1, E)$ ein Digraph mit $E = \{(A, B) \mid A \to \alpha B \beta \in P_1\}$.
Dann gilt offenbar: C erreichbar bzgl. $G \iff C$ von S aus erreichbar in D (zur Lösung dieses Problems existieren Algorithmen mit linearer Laufzeit).

4.1.4 Kellerautomaten

Die von KfGen erzeugten Sprachen können von Kellerautomaten akzeptiert werden.
Der Keller ist nötig, um den DFS–Durchlauf durch den Ableitungsbaum zu organisieren (beispielsweise zur Überprüfung der korrekten Klammerung von Ausdrücken).

Definition:
Ein *Kellerautomat (pushdown automaton PDA)* ist ein Quintupel $\mathcal{A} = (Q, T, \delta, q_0, F)$, wobei Q die Menge der Zustände, $q_0 \in Q$ der Startzustand und $F \subseteq Q$ die Menge der Endzustände bezeichne. T ist die Menge der Terminalsymbole.
Die Übergangsrelation δ ist definiert als $\delta \subseteq Q^+ \times (T \cup \{\varepsilon\}) \times Q^*$.

Beachte:

- Wir identifizieren die Zustandsmenge mit dem Kelleralphabet, wobei das oberste Kellerzeichen dem aktuellen Zustand entspricht.
- ε–Übergänge sind zulässig.
- $\mathcal{A}$ ist i. a. nichtdeterministisch.

Eine Konfiguration von $\mathcal{A}$ enthält alle Informationen über den aktuellen Berechnungszustand des Automaten.

Definition:
Eine *Konfiguration* von $\mathcal{A}$ ist ein Paar $(\gamma, w) \in Q^+ \times T^*$, wobei γ der aktuelle Kellerinhalt und w der noch zu lesende Teil der Eingabe ist.
Berechnungsschritte auf Konfigurationen beschreiben wir durch die Relation $\vdash_{\mathcal{A}}$:

$$(\gamma\alpha, xw) \vdash_{\mathcal{A}} (\gamma'\alpha, w) \;:\Longleftrightarrow\; (\gamma, x, \gamma') \in \delta \text{ für } \alpha \in Q^*,\ w \in T^*,\ x \in T \cup \{\varepsilon\}.$$

Eine Folge $\gamma_0\gamma_1 \cdots \gamma_n$ von Konfigurationen mit $\gamma_{i-1} \vdash_{\mathcal{A}} \gamma_i$ für $1 \leq i \leq n$ heißt eine *Berechnung* des Kellerautomaten $\mathcal{A}$.
Wir bezeichnen die reflexive und transitive Hülle von $\vdash_{\mathcal{A}}$ mit $\vdash_{\mathcal{A}}^*$.
Die von $\mathcal{A}$ erkannte Sprache ist $L(\mathcal{A}) = \{w \in T^* \mid (q_0, w) \vdash_{\mathcal{A}}^* (f, \varepsilon),\ f \in F\}$. Der Kellerautomat akzeptiert also stets mit einem Keller, der nur noch ein Kellerzeichen enthält.

Beobachtung:

1) $(\gamma, vw) \vdash_{\mathcal{A}}^* (\gamma', w) \implies (\gamma\alpha, vwu) \vdash_{\mathcal{A}}^* (\gamma'\alpha, wu)$ für alle $\alpha \in Q^*$, $u \in T^*$.
   Kellerautomaten sind i. a. nichtdeterministisch, d. h., daß es für eine Konfiguration **mehrere** Nachfolgekonfigurationen geben kann.
   Ein Kellerautomat $\mathcal{A}$ heißt *deterministisch*, falls für alle $(\gamma, w) \in Q^+ \times T^*$ gilt:
   $$\sharp\{(\gamma', w') \mid (\gamma, w) \vdash_{\mathcal{A}} (\gamma', w')\} \leq 1.$$
2) Ein Kellerautomat $\mathcal{A}$ ist genau dann deterministisch, falls für verschiedene Übergänge $(\beta_1, x, \beta_2), (\gamma_1, y, \gamma_2) \in \delta$ gilt: Ist β_1 Präfix von γ_1, dann ist $x \neq y$ und $x \neq \varepsilon \neq y$ (dann können wir δ auch als partielle Funktion schreiben).

Definition:
Ein Tripel (A, α, β) mit $A \to \alpha\beta \in P$ nennen wir *Item* und schreiben dafür auch $[A \to \alpha.\beta]$. α entspricht dem bereits abgearbeiteten Teil der Regel. Ein Item $[A \to \alpha.]$ heißt *vollständig*. Für eine Folge $\rho = [A_k \to \alpha_k.\beta_k] \dots [A_1 \to \alpha_1.\beta_1]$ von Items heißt $H(\rho) = \alpha_1 \cdots \alpha_k$ die *Geschichte* von ρ. $F(\rho) = \beta_k \cdots \beta_1$ nennen wir *Zukunft* von ρ.

Satz: Zu jeder KfG G existiert ein Kellerautomat $\mathcal{A}_G$ mit $L(G) = L(\mathcal{A}_G)$.
$\mathcal{A}_G$ kann in linearer Zeit berechnet werden (linear in der Ausgabe, nicht in der Eingabe).

Beweis:
Für $G = (N, T, P, S)$ sei $G' = (N', T, P', S')$ definiert durch $N' := N \cup \{S'\}$ für ein $S' \notin N$ und $P' = P \cup \{S' \to S\}$.

Wir definieren $\mathcal{A}_G = (I_{G'}, T, \delta, q_0, F)$ durch die Menge $I_{G'}$ der Items zu G',
$q_0 = [S' \to .S]$ und $F = \{[S' \to S.]\}$.

δ ist gegeben durch:

$$
\begin{array}{rl}
\textbf{Expansion:} & ([A \to \alpha.B\beta], \varepsilon, [B \to .\gamma][A \to \alpha.B\beta]) \text{ für } B \to \gamma \in P \\
\textbf{Shift:} & ([A \to \alpha.a\beta], a, [A \to \alpha a.\beta]) \text{ für } a \in T \\
\textbf{Reduktion:} & ([B \to \gamma.][A \to \alpha.B\beta], \varepsilon, [A \to \alpha B.\beta])
\end{array}
$$

Korrektheit:
Wir zeigen: $([A \to \alpha.X\beta], w) \vdash^*_{\mathcal{A}} ([A \to \alpha X.\beta], \varepsilon) \iff X \xrightarrow{*}_{G'} w$.
Die Behauptung ergibt sich dann mit $A = S'$, $\alpha = \beta = \varepsilon$ und $X = S$.
Beweis: Induktion nach der Anzahl der Berechnungsschritte
"$\Rightarrow$" Fall 1: Ist $X = a \in T$, dann ist $w = a$. $\surd$
Fall 2: Für $X = B \in N$ gilt:
$([A \to \alpha.X\beta], w) \vdash_{\mathcal{A}} ([X \to .\gamma][A \to \alpha.X\beta], w) \vdash^*_{\mathcal{A}}$
$([X \to \gamma.][A \to \alpha.X\beta], \varepsilon) \vdash_{\mathcal{A}} ([A \to \alpha X.\beta], \varepsilon)$

Sei $\gamma = X_1 \cdots X_n$.
Für $n = 0$ ist $w = \varepsilon$.
Betrachte für $n > 0$ und $w = v_1 \cdots v_n$ die Teilberechnungen der Form
$([X \to X_1 \cdots X_{j-1}.X_j X_{j+1} \cdots X_n], v_j) \vdash^*_{\mathcal{A}} ([X \to X_1 \cdots X_{j-1} X_j.X_{j+1} \cdots X_n], \varepsilon)$,
wobei $1 \le j \le n$, $X_j \in V$.
Nach Induktionsvoraussetzung gilt $X_j \xrightarrow{*}_{G'} v_j$.
Damit gilt $X \longrightarrow_{G'} X_1 \cdots X_n \xrightarrow{*}_{G'} v_1 X_2 \cdots X_n \xrightarrow{*}_{G'} v_1 \cdots v_n = w$.
Daraus folgt die Behauptung $X \xrightarrow{*}_{G'} w$.

"$\Leftarrow$": analog

Beispiel:

Die Grammatik G mit $P = \{S \to AB,\ A \to a,\ B \to b\}$ wird
zu G' mit $P' = \{S' \to S,\ S \to AB,\ A \to a,\ B \to b\}$.

Wir erhalten als Itemmenge:
$I_{G'} = \{\ [S' \to .S],\ [S' \to S.],\ [S \to .AB],\ [S \to A.B],\ [S \to AB.],$
$[A \to .a],\ [A \to a.],\ [B \to .b],\ [B \to b.]\ \}$

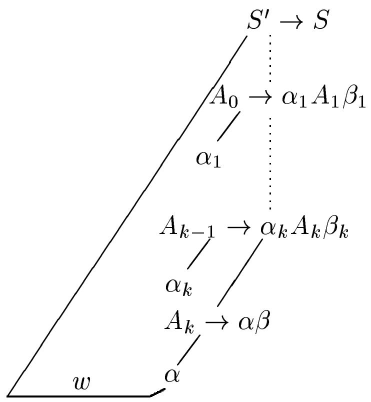
Als Übergansrelation δ:

$$\begin{aligned}\delta = \{ & ([S' \to .S], \varepsilon, [S \to .AB][S' \to .S]),\ ([S \to .AB], \varepsilon, [A \to .a][S \to .AB]),\\ & ([A \to .a], a, [A \to a.]),\ ([A \to a.][S \to .AB], \varepsilon, [S \to A.B]),\\ & ([S \to A.B], \varepsilon, [B \to .b][S \to A.B]),\ ([B \to .b], b, [B \to b.]),\\ & ([B \to b.][S \to A.B], \varepsilon, [S \to AB.]),\ ([S \to AB.][S' \to .S], \varepsilon, [S' \to S.]) \}\end{aligned}$$

Wir erhalten als Berechnung auf dem Wort ab:

$$\begin{aligned}& ([S' \to .S], ab) \vdash\\ & ([S \to .AB][S' \to .S], ab) \vdash ([A \to .a][S \to .AB][S' \to .S], ab) \vdash\\ & ([A \to a.][S \to .AB][S' \to .S], b) \vdash ([S \to A.B][S' \to .S], b) \vdash\\ & ([B \to .b][S \to A.B][S' \to .S], b) \vdash ([B \to b.][S \to A.B][S' \to .S], \varepsilon) \vdash\\ & ([S \to AB.][S' \to .S], \varepsilon) \vdash ([S' \to S.], \varepsilon)\end{aligned}$$

Beobachtung: $([S' \to .S], w) \vdash^*_{\mathcal{A}_G} (\rho, \varepsilon) \implies H(\rho) \xrightarrow{*}_{G'} w$.



$\rho = [A_k \to \alpha.\beta][A_{k-1} \to \alpha_k.A_k\beta_k] \cdots$
$\cdots [A_0 \to \alpha_1.A_1\beta_1]\rho'[S' \to .S]$
entspricht dem Pfad von der Wurzel zu α im partiellen Ableitungsbaum.

$H(\rho) = H(\rho')\alpha_1 \cdots \alpha_k\alpha$ entspricht dem Teil links des Pfades.

w ist das bereits abgeleitete Wort.

Um einen Ableitungsbaum für die Eingabe aufzubauen, versehen wir den Kellerautomaten $\mathcal{A}_G$ mit einer *Ausgabe*.

Möglichkeiten:

1) Die Ausgabe soll eine L-Ableitung liefern. In diesem Fall müssen die Regeln bei der Expansion ausgegeben werden.

2) Die Ausgabe soll eine R-Ableitung liefern. Hier müssen die Regeln bei der Reduktion ausgegeben und diese Ausgabe rückwärts gelesen werden.

Problem: Der Kellerautomat wählt i. a. nichtdeterministisch zwischen den verschiedenen Expansionsübergangen aus. Da die Implementierung des Nichtdeterminismus schwierig (und nicht immer erfolgreich) ist, beseitigen wir ihn.

Abhilfe:

1) Die Überlegung, daß eine k–Vorausschau auf die zukünftige Eingabe genügt, um die Expansionsübergänge festzulegen, führt zur LL(k)–Analyse bzw. zu LL(k)–Grammatiken.

2) Versuche, durch k–Vorausschau eindeutig die Reduktionen zu erkennen, führen zur LR(k)–Analyse bzw. zu LR(k)–Grammatiken. Eventuell müssen mehrere Expansionen "parallel" verfolgt werden (partielle Teilmengenkonstruktion).

4.2 Einige Grundbegriffe aus der Verbandstheorie

Definition: Sei D eine Menge und $\sqsubseteq\ \subseteq D \times D$ eine zweistellige Relation über D.
Eine *partielle Ordnung (partial order (po))* ist ein Paar $(D, \sqsubseteq)$ für das gilt:

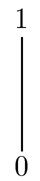
1. $x \sqsubseteq x \quad \forall x \in D$ (Reflexivität)
2. $x \sqsubseteq y$ und $y \sqsubseteq x \Rightarrow x = y \quad \forall x, y \in D$ (Antisymmetrie)
3. $x \sqsubseteq y$ und $y \sqsubseteq z \Rightarrow x \sqsubseteq z \quad \forall x, y, z \in D$ (Transitivität)

Bemerkung: Ist '$\sqsubseteq$' bekannt, dann schreiben wir auch D statt $(D, \sqsubseteq)$.

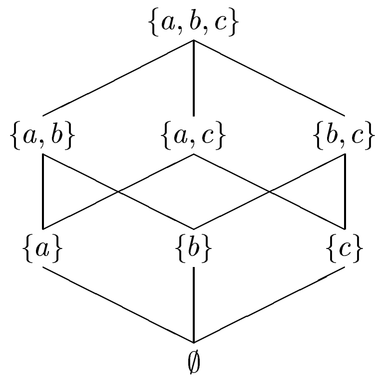
Beispiel für partielle Ordnungen: (Hasse–Diagramme)

0 1 2 3 . . .

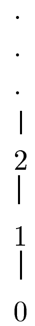
1) 'flache Ordnung' auf $(I\!N, =)$



2) $\mathbf{2}$



3) Teilmengenverband



4) $I\!N$ mit 'natürlicher Ordnung'

Die strukturerhaltenden Abbildungen zu part. Ordnungen sind die monotonen Funktionen.

Definition: Seien $(D_1, \sqsubseteq)$ und $(D_2, \sqsubseteq)$ partielle Ordnungen und f eine Funktion $f : D_1 \longrightarrow D_2$.
Dann heißt f *monoton*, falls für alle $x, y \in D_1$ gilt: $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$

Definition: Sei $(D, \sqsubseteq)$ eine partielle Ordung, $X \subseteq D$.
$d \in D$ heißt *obere Schranke* von X, falls $x \sqsubseteq d \ \forall x \in X$.
Eine obere Schranke d heißt *kleinste obere Schranke (least upper bound (lub))* von X, falls $d \sqsubseteq d'$ für jede obere Schranke d' von X.

Wir schreiben für kleinste obere Schranken: $d = \bigsqcup X$ oder $d = x \sqcup y$, falls $X = \{x, y\}$
oder $d = \bigsqcup\limits_{i \geq 0} x_i$, falls $X = \{x_i \mid i \geq 0\}$.
Analog definiert man untere Schranken, größte untere Schranken (greatest lower bounds (glb)) und notiert diese mit $\sqcap$.
Das kleinste Element einer part. Ordnung (falls vorhanden) bezeichnen wir mit $\perp$ (*Bottom*).

Definition: Eine *(aufsteigende) Kette (ascending chain (ac))* in einer part. Ordnung $(D, \sqsubseteq)$ ist eine (evtl. unendliche) Folge von Elementen $x_0, x_1, x_2, \ldots \in D$ mit $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$. Gilt sogar $x_0 \sqsubset x_1 \sqsubset x_2 \sqsubset \cdots$, so sprechen wir von einer *echt* aufsteigenden Kette in D.

4.3 Berechnung von Vorausschaumengen

4.3.1 First_k

Definition:
Für $w = a_1 \cdots a_n \in T^n$ sei $\mathit{First}_k(w) = \begin{cases} w, & \text{falls } n \leq k \\ a_1 \cdots a_k & \text{sonst} \end{cases}$
Für $A \in N$ sei $\mathit{First}_k(A) = \{\mathit{First}_k(w) \mid A \xrightarrow{*} w\}$
Für $\alpha \in V^*$ sei $\mathit{First}_k(\alpha) = \{\mathit{First}_k(w) \mid \alpha \xrightarrow{*} w\}$

Wir möchten $\mathit{First}_k(\alpha)$ aus den Mengen $\mathit{First}_k(Y_j)$ für $\alpha = Y_1 \cdots Y_n$ berechnen.

Beobachtung: $\mathit{First}_k(w_1 w_2) = \mathit{First}_k(\mathit{First}_k(w_1) \cdot \mathit{First}_k(w_2))$.

Definition: Sei $T^{\leq k} := \bigcup_{j=0}^{k} T^j$ die Menge der Worte über T, die höchstens Länge k haben, und $D := 2^{T^{\leq k}}$ die Menge von Teilmengen solcher Worte.
Wir definieren die *k–Konkatenation* $\oplus_k : D \times D \to D$ durch $L_1 \oplus_k L_2 = \mathit{First}_k(L_1 \cdot L_2) = \{\mathit{First}_k(w_1 w_2) \mid w_i \in L_i\}$.

Beobachtung: $\mathit{First}_k(\alpha) = \mathit{First}_k(Y_1) \oplus_k \cdots \oplus_k \mathit{First}_k(Y_n)$ für $\alpha = Y_1 \cdots Y_n$.

Wir können hierbei die Klammerung weglassen, da die k–Konkatenation assoziativ ist. Wir betrachten im folgenden die Eigenschaften der k–Konkatenation:

1. $\oplus_k$ ist assoziativ, da die Konkatenation assoziativ ist
2. Neutrales Element : $\{\varepsilon\}$, d. h. $\{\varepsilon\} \oplus_k L = L \oplus_k \{\varepsilon\} = \mathit{First}_k(L)$
3. Null–Element : $\emptyset$, d. h. $\emptyset \oplus_k L = L \oplus_k \emptyset = \emptyset$
4. $\oplus_k$ ist monoton, d. h. $M_1 \subseteq L_1, M_2 \subseteq L_2 \implies M_1 \oplus_k M_2 \subseteq L_1 \oplus_k L_2$

Zur Berechnung der $\mathit{First}_k(X)$–Mengen, $X \in V$ für eine Grammatik $G = (N, T, P, S)$, stellen wir eine Menge von Bedingungen (*Constraintmenge, -system*) auf. Sei im folgenden $k \geq 1$.

- $\mathit{First}_k(a) \supseteq \{a\}$, $a \in T$
- $\mathit{First}_k(A) \supseteq \mathit{First}_k(\alpha) = \mathit{First}_k(Y_1) \oplus_k \cdots \oplus_k \mathit{First}_k(Y_n)$ für $A \to \alpha \in P, \alpha = Y_1 \cdots Y_n$

Die Menge der $\mathit{First}_k(X)$–Mengen für $X \in V$, erfüllt nicht nur diese Bedingungen, sondern ist auch das **kleinste** System von Mengen, welches diese Bedingungen erfüllt, d. h.:
Für jedes andere System $\mathcal{S}$ von Mengen $S_Y \subseteq D$, $Y \in V$, welches die Bedingungen $S_a \supseteq \{a\}$ und $S_A \supseteq S_{Y_1} \oplus_k \cdots \oplus_k S_{Y_n}, A \to Y_1 \cdots Y_n \in P$ erfüllt, gilt $\mathit{First}_k(X) \subseteq S_Y$.

Das Problem, auf das wir gestoßen sind, ist nun, für eine gegebene Menge von Bedingungen eine kleinste Lösung zu finden. Nachfolgend werden wir, zum besseren Verständnis, parallel zum allgemeinen Problem ein konkretes Beispiel betrachten.

Allgemeine Konstruktion	**Spezielles Beispiel**
	Sei $k = 2$. Betrachte Grammatik G: $E \to E + T \mid T$, $T \to T * F \mid F$, $F \to (E) \mid \texttt{int}$
Sei *Var* eine Menge von Variablen	Variablen: $Var = \{S_Y \mid Y \in \{E, T, F, (,), +, *, \texttt{int}\}\}$
Sei $(D, \sqsubseteq)$ partielle Ordnung	$(D, \sqsubseteq) = (2^{T^{\leq 2}}, \subseteq)$
Sei S eine endliche Menge von Constraints der Form $x \sqsupseteq t, x \in Var$.	Constraintmenge $\mathcal{S}$: $S_Y \supseteq \{Y\}\ \forall Y \in \{(,), +, *, \texttt{int}\}$ $S_F \supseteq S_{\texttt{int}}$ $S_F \supseteq S_(\oplus_2 S_E \oplus_2 S_)$ $S_T \supseteq S_F$ $S_T \supseteq S_T \oplus_2 S_* \oplus_2 S_F$ $S_E \supseteq S_T$ $S_E \supseteq S_E \oplus_2 S_+ \oplus_2 S_T$
Für die rechten Seiten t wird eine Funktion $[t] : (Var \to D) \to D$ definiert, die jeder Variablenbelegung aus $Var \to D$ einen Wert aus D zuweist.	Variablenbelegung: Sei $\mu :$ $S_(\mapsto \{(\}$, $S_) \mapsto \{)\}$, $S_E \mapsto \{\texttt{int}\}$ Für $t = S_(\oplus_2 S_E \oplus_2 S_)$ erhalten wir $[t](\mu) = \{(\} \oplus_2 \{\texttt{int}\} \oplus_2 \{)\} = \{(\texttt{int}\}$
Gesucht: (kleinste) Variablenbelegung $\mu : Var \to D$, die alle Constraints erfüllt, d. h. $\mu(x) \sqsupseteq [t](\mu) \quad \forall\, x \sqsupseteq t$. <u>Idee:</u> Worklist–Algorithmus	
(1) Sukzessive Approximation der Lösung. Setze zu Beginn $\mu(x) = \bot$ (kleinster Wert in D, sofern existent).	Kleinster Wert in $D = 2^{T^{\leq 2}}$ ist $\emptyset$.
(2) Wann immer $\mu(x) \not\sqsupseteq [t](\mu)$, dann setze $\mu(x) = \mu(x) \sqcup [t](\mu)$.	Kleinste obere Schranke von $d_1,\ d_2 \in D$ ist $d_1 \cup d_2$ mit $d_1, d_2 \in D$.
(3) Wann immer der Wert für eine Variable y sich ändert fügen wir alle Constraints $x \sqsupseteq t$ in die Worklist ein, deren rechte Seite von y "abhängt". D. h. wir benötigen sogenannte Influenz–Mengen $\mathit{infl}(y) = \{x \sqsupseteq t \mid [t]$ benötigt $y\}$.	

Damit ergibt sich folgender Algorithmus:

Algorithmus:

```
SetOfConstraints W = ∅ ;

void solve (Constraint x⊒t) {
D new;
  new = [t](μ);  // rechte Seite auswerten
  if (μ(x) ⋣ new) {       //Constraint nicht erfüllt
    μ(x) = μ(x) ⊔ new;
    W = W ∪ infl(x);
  }
}
void main (void) {
Constraint r;
  forall (x ∈ Var)
    μ(x) = ⊥;
  forall (r ∈ S)
    solve(r);        //erstes Auswerten sämtlicher Constraints
  while (W ≠ ∅ ) {
    r = extract(W);
    solve(r);
  }
}
```

Bemerkung: Einige Optimierungen sind möglich

- Ändern der Anordnung der Elemente in der Worklist
- Es genügt, Neuberechnungen von Constraints mit *neuen* Elementen durchzuführen

Beispiel:

Wir nummerieren die Elemente der Constraintmenge durch und bilden die Influenzmengen:

0:	$S_($	$\supseteq \{(\}$	$\mathit{infl}(S_()$	$=$	$\{6\}$
1:	$S_)$	$\supseteq \{)\}$	$\mathit{infl}(S_))$	$=$	$\{6\}$
2:	S_+	$\supseteq \{+\}$	$\mathit{infl}(S_+)$	$=$	$\{10\}$
3:	S_*	$\supseteq \{*\}$	$\mathit{infl}(S_*)$	$=$	$\{8\}$
4:	$S_{\texttt{int}}$	$\supseteq \{\texttt{int}\}$	$\mathit{infl}(S_{\texttt{int}})$	$=$	$\{5\}$
5:	S_F	$\supseteq S_{\texttt{int}}$	$\mathit{infl}(S_F)$	$=$	$\{7,8\}$
6:	S_F	$\supseteq S_(\oplus_2 S_E \oplus_2 S_)$			
7:	S_T	$\supseteq S_F$	$\mathit{infl}(S_T)$	$=$	$\{8,9,10\}$
8:	S_T	$\supseteq S_T \oplus_2 S_* \oplus_2 S_F$			
9:	S_E	$\supseteq S_T$	$\mathit{infl}(S_E)$	$=$	$\{6,10\}$
10:	S_E	$\supseteq S_E \oplus_2 S_+ \oplus_2 S_T$			

Der Worklist–Algorithmus liefert nun folgende Werte:

r	S_F	S_T	S_E	$S_(\dots S_{\texttt{int}}$	Worklist
`forall` :					
$0\dots4$	–	–	–	(... `int`	$5,6,8,10$
5	`int`	–	–	–	$5,6,\underline{7},8,10$
6	–	–	–	–	
7	–	`int`	–	–	$5,6,7,8,\underline{9},10$
8	–	`int*`	–	–	
9	–	–	`int`, `int*`	–	
10	–	–	`int+`	–	
`while` :					
5	–	–	–	–	$6,7,8,9,10$
6	`(int`	–	–	–	$7,8,9,10$
7	–	`(int`	–	–	$8,9,10$
8	–	–	–	–	$9,10$
9	–	–	`(int`	–	$\underline{6},10$
6	`((`	–	–	–	$\underline{7},\underline{8},10$
7	–	`((`	–	–	$8,\underline{9},10$
8	–	–	–	–	$9,10$
9	–	–	`((`	–	$\underline{6},10$
6	–	–	–	–	10
10	–	–	–	–	

Wir erhalten als First_2–Mengen der Nichtterminale:

$\mathit{First}_2(F) = S_F = \{\texttt{int}, (\texttt{int}, ((\,\}$,

$\mathit{First}_2(T) = S_T = \{\texttt{int}, (\texttt{int}, \texttt{int}*, ((\,\}$,

$\mathit{First}_2(E) = S_E = \{\texttt{int}, (\texttt{int}, \texttt{int}*, \texttt{int}+, ((\,\}$

Zur Korrektheit des Algorithmus:

Sei μ_i der Wert von μ nach dem i–ten Aufruf von **solve**, analog W_i für W. Dann gilt:

1) $\mu_i(x) \sqsubseteq \mu_{i+1}(x), x \in \mathit{Var}$

2) Für $i \geq \sharp S$ gilt: $\mu_i(x) \not\sqsupseteq [t](\mu) \implies x \sqsupseteq t \in W_i$

Sei h die Höhe von D, d. h. die maximale Länge einer echt aufsteigenden Kette in D.

Für eine endliche Menge A ist beispielsweise die Höhe h von $D = 2^A$ $h = \sharp A$.

Sei $|S| \geq \sum_{x \in \mathit{Var}} \sharp \mathit{infl}(x)$. Dann folgt aus 1)

3) Die Anzahl der Einfügungen von Constraints in W ist $\leq \sum_{x \in \mathit{Var}} h \cdot \sharp \mathit{infl}(x)$

$= h \cdot \sum_{x \in \mathit{Var}} \sharp \mathit{infl}(x) \leq h \cdot |S|$

Wir folgern:

3) $\implies$ Der Algorithmus terminiert nach maximal $\sharp S + h \cdot |S| \leq (h+1) \cdot |S|$ Aufrufen von

solve
2) $\Longrightarrow$ Am Ende sind alle Constraints erfüllt, d. h. μ liefert eine Lösung.

Frage: Liefert der Algorithmus auch die *kleinste* Lösung?

Zusätzliche Annahme:

Die rechten Seiten $[t]$ sind monoton, d. h. für $\mu_1, \mu_2 : Var \to D$ mit $\mu_1(x) \sqsubseteq \mu_2(x), x \in Var$ gilt stets auch $[t](\mu_1) \sqsubseteq [t](\mu_2)$. Diese zusätzliche Annahme ist bei uns natürlicherweise erfüllt (siehe Übungen). Wir erkennen mit dieser Annahme

4) Sei σ irgendeine Lösung für S. Dann gilt: $\mu_i(x) \sqsubseteq \sigma(x)\ \forall i, x \in Var$

4) $\Longrightarrow$ Die Lösung, die der Algorithmus liefert, ist auch die *kleinste* Lösung.

4.3.2 $Follow_k$

Definition: $Follow_k(A) = \{First_k(w) \mid S \xrightarrow{*} \alpha A w\}$

Sei $G = (N, T, P, S)$ eine Grammatik.
Mit dieser Definition erhalten wir folgende Bedingungen zur Berechnung der $Follow_k(X)$-Mengen der erweiterten Grammatik $G = (N \cup \{S'\}, T, P \cup \{S' \to S\}, S')$, $(S' \notin N)$:

- $Follow_k(S') \supseteq \{\varepsilon\}$
- $Follow_k(A) \supseteq First_k(X_1) \oplus_k \cdots \oplus_k First_k(X_n) \oplus_k Follow_k(B)$ für $B \to \alpha A X_1 \cdots X_n \in P$

Es gilt, daß $Follow_k(A), A \in N$ die kleinste Lösung des obigen Constraint–Systems ist. Die Voraussetzungen zur Anwendung unseres Algorithmus' sind damit erfüllt.

Beispiel:

Wir erweitern zur Berechnung der $Follow_2$–Mengen unsere Beispielgrammatik um die Produktion $S' \to E$. Für $First_2(X)$ schreiben wir abkürzend $First_X$.
Wir erhalten analog zur Berechnung der $First_2$–Mengen folgendes Constraintsystem:

0:	$S_{S'} \supseteq \{\varepsilon\}$	$infl(S_{S'})$	$=$	$\{1\}$
1:	$S_E \supseteq S_{S'}$	$infl(S_E)$	$=$	$\{2, 4, 5\}$
2:	$S_E \supseteq First_+ \oplus_2 First_T \oplus_2 S_E = \{+\texttt{int}, +(\} \oplus_2 S_E$			
3:	$S_E \supseteq First_) \oplus_2 S_F = \{)\} \oplus_2 S_F$			
4:	$S_T \supseteq S_E$	$infl(S_T)$	$=$	$\{6, 7, 8\}$
5:	$S_T \supseteq S_E$			
6:	$S_T \supseteq First_* \oplus_2 First_F \oplus_2 S_T = \{*\texttt{int}, *(\} \oplus_2 S_T$			
7:	$S_F \supseteq S_T$	$infl(S_F)$	$=$	$\{3\}$
8:	$S_F \supseteq S_T$			

Der Worklist–Algorithmus liefert für dieses System folgende Werte:

r	$S_{S'}$	S_E	S_T	S_F	Worklist
`forall` :					
0	ε	–	–	–	$\underline{1}$
1	–	ε	–	–	$1, \underline{2}, \underline{4}, \underline{5}$
2	–	$+\texttt{int}, +($	–	–	
3	–	–	–	–	
4	–	–	$\varepsilon, +\texttt{int}, +($	–	$1, 2, 4, 5, \underline{6}, \underline{7}, \underline{8}$
5	–	–	–	–	
6	–	–	$*\texttt{int}, *($	–	
7	–	–	–	$\varepsilon, +\texttt{int}, +(, *\texttt{int}, *($	$1, 2, \underline{3}, 4, 5, 6, 7, 8$
8	–	–	–	–	
`while` :					
1	–	–	–	–	$2, 3, 4, 5, 6, 7, 8$
2	–	–	–	–	$3, 4, 5, 6, 7, 8$
3	–	$),)+,)*$	–	–	$\underline{2}, 4, 5, 6, 7, 8$
2	–	–	–	–	$4, 5, 6, 7, 8$
4	–	–	$),)+,)*$	–	$5, 6, 7, 8$
5	–	–	–	–	$6, 7, 8$
6	–	–	–	–	$7, 8$
7	–	–	–	$),)+,)*$	$\underline{3}, 8$
3	–	$))$	–	–	$\underline{2}, \underline{4}, \underline{5}, 8$
2	–	–	–	–	$4, 5, 8$
4	–	–	$))$	–	$5, \underline{6}, \underline{7}, 8$
5	–	–	–	–	$6, 7, 8$
6	–	–	–	–	$7, 8$
7	–	–	–	$))$	$\underline{3}, 8$
3	–	–	–	–	8
8	–	–	–	–	

Wir erhalten als Follow_2–Mengen der Nichtterminale:

$\mathit{Follow}_2(S') = S_{S'} = \{\varepsilon\}$,
$\mathit{Follow}_2(F) = S_F = \{\varepsilon, +\texttt{int}, +(, *\texttt{int}, *(,),)+,)*,))\}$,
$\mathit{Follow}_2(T) = \mathit{Follow}_2(F)$,
$\mathit{Follow}_2(E) = S_E = \{\varepsilon, +\texttt{int}, +(,),)+,)*,))\}$

Bemerkung:

Der Algorithmus läßt sich noch vereinfachen: Wenn man zu Beginn alle Constraints in die Worklist einfügt, dann kann man die `forall`–Schleife weglassen.

4.3.3 Schnelle Verfahren für Spezialfälle von Constraint–Systemen

Constraint–Systeme über *2*

Wir betrachten im folgenden den Verband $(D, \sqsubseteq) = (\{0,1\}, <) =: \mathit{2}$ mit $\sqcup \equiv \vee$ und Constraint–Systeme über *2* mit Bedingungen der Form $x \geq x_1 \wedge \cdots \wedge x_n$.

Unter der Annahme, daß die rechten Seiten $x_1 \wedge \cdots \wedge x_n$ in konstanter Zeit ausgewertet werden können, läuft unser Verfahren für diesen Fall in linearer Zeit, da die Höhe h des Verbandes 2 ist.
Für Variablen S_X, $X \in N \cup T$ mit $S_X = 1 \;:\Longleftrightarrow\; (X$ ist produktiv oder $X \in T \cup \{\varepsilon\})$, entspricht das Lösen solcher Constraint–Systeme $S_A \geq S_{X_1} \wedge \cdots \wedge S_{X_n}$ für $A \to X_1 \cdots X_n \in P$, dem Berechnen der produktiven Nichtterminale für eine Grammatik $G = (N, T, P, S)$.
Wir können daher den Worklist–Algorithmus auch auf Constraint–Systeme über $\mathbf{2}$ anwenden und erhalten — unabhängig von der Länge der rechten Seiten — ein Verfahren zur Berechnung der produktiven Nichtterminale mit linearer Laufzeit.

Berechnung von *First*$_1$ und *Follow*$_1$

Beobachtung:

Sei T eine Menge von Terminalsymbolen.
Seien $L_1, L_2 \subseteq T \cup \{\varepsilon\}$ mit $L_1 \neq \emptyset \neq L_2$. Dann gilt: $L_1 \oplus_1 L_2 = \begin{cases} L_1, \text{ falls } \varepsilon \notin L_1 \\ (L_1 \setminus \{\varepsilon\}) \cup L_2 \text{ sonst} \end{cases}$

Idee: Wir verwalten die Eigenschaft, daß ε in einer Menge enthalten ist, separat.
Wir definieren dazu die Menge $F_\varepsilon(X) = \mathit{First}_1(X) \setminus \{\varepsilon\}$ und die Eigenschaft $E(X) \in \mathbf{2}$ mit $E(X) = 1 \;:\Longleftrightarrow\; \varepsilon \in \mathit{First}_1(X)$. Dies reduziert die Berechnung von $\mathit{First}_1(X)$ auf die Lösung eines Constraint–Systems über $\mathbf{2}$ für $E(X)$ und eines Constraint–Systems für $F_\varepsilon(X)$.
Diese Systeme enthalten nun keine $\oplus_1$–Operationen mehr.
$F_\varepsilon(X)$ heißt auch *ε–freie $First_1$–Menge für X*.

Das Constraint–System S_ε

Das Contraint–System S_ε besteht aus Regeln der Form:

- $S_a \geq 0$ für $a \in T$,
- $S_A \geq 1$ für $A \to \varepsilon \in P$,
- $S_A \geq S_{X_1} \wedge \cdots \wedge S_{X_n}$ für die Regeln $A \to X_1 \cdots X_n \in P$.

Hierbei sind wiederum $S_X, X \in V$ die Constraint–Variablen für Symbole aus $V = (N \cup T)$. Im Unterschied zum Constraint–System zur Berechnung der produktiven Nichtterminale fallen in S_ε alle Constraint–Variablen $S_a \geq 1, a \in T$ weg.
Nun gilt: Die kleinste Lösung des Constraint–Systems S_ε ist gerade $E(X), X \in V$.

Das Constraint–System *S1*

Das Contraint–System *S1* besteht aus Regeln der Form:

- $S_a \supseteq \{a\}$ für $a \in T$,
- $S_A \supseteq S_{X_j}$, falls gilt: $A \to X_1 \cdots X_n \in P$ und $\forall\, i < j : E(X_i) = 1$.

Es gilt: $F_\varepsilon(X), X \in V$ ist gerade die kleinste Lösung von *S1*.

Bei der Berechnung von *Follow*$_1$ treten die *Follow*–Mengen stets im 2. Argument von $\oplus_1$ auf, d. h., wir brauchen keine ε–freien *Follow*–Mengen zu betrachten.

Das Constraint–System *S2*

Das Constraint–System *S2* besteht aus Regeln der Form:

- $S_{S'} \supseteq \{\varepsilon\}$,
- $S_B \supseteq \mathit{First}_1(X_j)$, sofern $A \to \alpha B X_1 \cdots X_n \in P$ und $\forall\, i < j : E(X_i) = 1$,
- $S_B \supseteq S_A$, sofern $A \to \alpha B X_1 \cdots X_n \in P$ und $\forall\, 1 \le i \le n : E(X_i) = 1$.

Es gilt: $\mathit{Follow}_1(X)$, $X \in (V \cup \{S'\})$ ist die kleinste Lösung von *S2*.

Diese Vereinfachte Berechnung der First_1 –/ Follow_1–Mengen ist allerdings nur korrekt, sofern alle Nichtterminale der betrachteten KFG **produktiv** sind !

Beispiel:

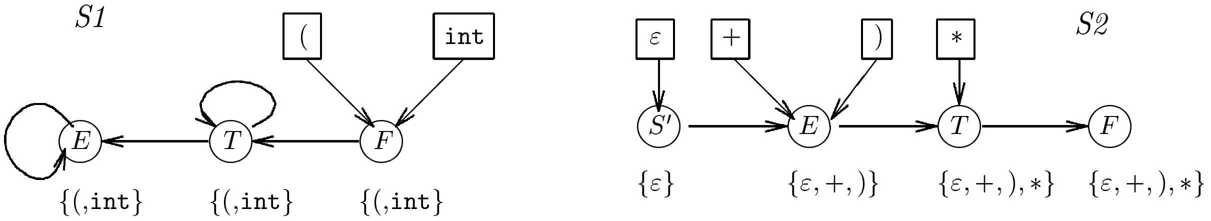
Betrachte folgende Grammatik G:
$$\begin{aligned} S' &\to E \\ E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid \texttt{int} \end{aligned}$$

Offensichtlich gilt für alle $X \in V$: $E(X) = 0$.

Constraint–System *S1*:
$$\begin{array}{ll} S_a \supseteq \{a\} & \forall\, a \in \{+, *, (,), \texttt{int}\} \\ S_E \supseteq S_E & S_E \supseteq S_T \\ S_T \supseteq S_T & S_T \supseteq S_F \\ S_F \supseteq S_(& S_F \supseteq S_{\texttt{int}} \end{array}$$

Constraint–System *S2*:
$$\begin{array}{ll} S_{S'} \supseteq \{\varepsilon\} & S_E \supseteq S_{S'} \\ S_E \supseteq \{+\} & S_E \supseteq \{)\} \\ S_T \supseteq \{*\} & S_T \supseteq S_E \\ S_F \supseteq S_T & \end{array}$$

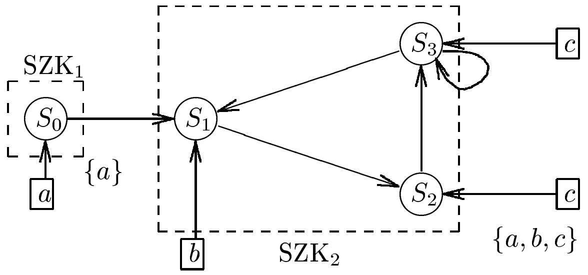
Als Abhängigkeitsgraphen der Constraint–Systeme erhalten wir:



Derartige Constraint–Systeme wie *S1* und *S2* enthalten stets Constraints der Form $S_A \supseteq S_B$ bzw. $S_A \supseteq \{a\}$. Für diese Constraint–Systeme können wir die Berechnung der kleinsten Lösung sehr effizient gestalten, indem wir die Werte $\{a\}$ durch den Abhängigkeitsgraphen propagieren.

Beispiel:

$S_0 \supseteq \{a\}$
$S_1 \supseteq \{b\}$
$S_1 \supseteq S_0$
$S_1 \supseteq S_3$
$S_2 \supseteq \{c\}$
$S_2 \supseteq S_1$
$S_3 \supseteq \{c\}$
$S_3 \supseteq S_2$
$S_3 \supseteq S_3$



Vorgehensweise bei der Berechnung:

1. Berechne in linearer Zeit starke Zusammenhangskomponenten (SZK) für die Variablen.
2. Ordne die starken Zusammenhangskomponenten topologisch (im Bsp. : $\mathrm{SZK}_1 < \mathrm{SZK}_2$).
3. Propagiere eingehende Werte $\{a\}$ jeweils in der kleinsten SZK zu allen Knoten (im Bsp. findet man für SZK_1 den Wert $\{a\}$).
4. Propagiere diese Menge zu allen nachfolgenden starken Zusammenhangskomponenten (im Bsp. findet man für SZK_2 die Menge $\{a\} \cup \{b, c\} = \{a, b, c\}$).

Der Wert einer Variablen X ist dann die Menge, die für die starke Zusammenhangskomponente berechnet wurde, in der X liegt.

Aufwand:	Aufbau des Graphen	$\mathrm{O}(\lvert S\rvert)$
	Berechnung der SZK	$\mathrm{O}(\lvert S\rvert)$
	Propagierung	$\mathrm{O}(\lvert S\rvert) + \mathrm{O}(\sharp\text{Kanten}) \leq \mathrm{O}(\lvert S\rvert)$

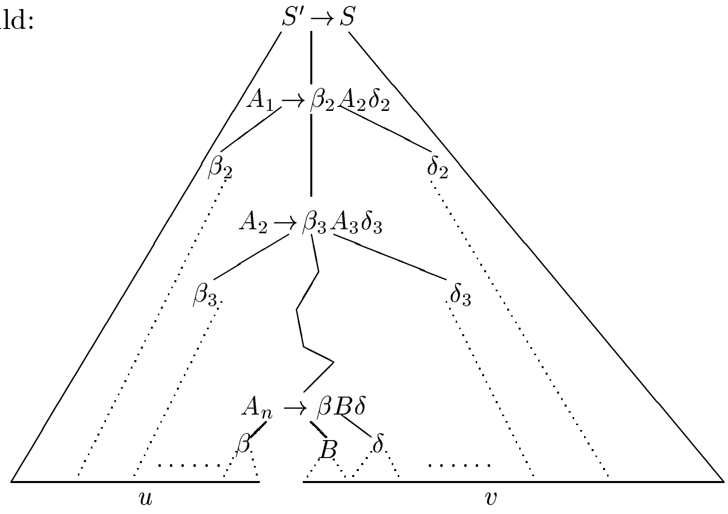
Die hier auftretenden Mengen implementieren wir als Bitvektoren, da die Anzahl der Tokens klein ist (i. a. höchstens 100). Die Vereinigung zweier Mengen geschieht dann in konstanter Zeit. Daraus ergibt sich eine lineare Gesamtkomplexität.

4.4 Top–Down–Syntaxanalyse

Sei $([S' \to .S], uv) \overset{*}{\vdash} ([A \to \beta.B\delta]\rho, v)$, wobei $\rho = [A_{n-1} \to \beta_n.A_n\delta_n] \cdots [A_0 \to \beta_1.A_1\delta_1]$ mit $[A_0 \to \beta_1.A_1\delta_1] = [S' \to .S]$ und $A_n = A \quad (\star)$.

Wir zeigten bereits (s. Seite 45): $([S' \to .S], w) \vdash^*_{\mathcal{A}} (\rho, \varepsilon) \implies H(\rho) \xrightarrow{*}_G w$.
Für $(\star)$ gilt dann: $H(\rho)\beta \xrightarrow{*} u$ mit $H(\rho) = \beta_2 \cdots \beta_n$.
Analog findet man: $B\,\delta\,F(\rho) \xrightarrow{*} v$ mit $F(\rho) = \delta_n \cdots \delta_2$.

Im Bild:



Seien $B \to \alpha_1 \mid \ldots \mid \alpha_n$ die Alternativen für B in P.
Bei jedem Expansionsübergang stellt sich das Problem, die richtige Alternative zu finden.
Wir benutzen zur Auswahl die nächsten k Eingabesymbole, d. h. $\mathit{First}_k(v)$.

Definition: Für $k \geq 1$ heißt eine kontextfreie Grammatik G $LL(k)$–*Grammatik*, falls für alle $B\beta \in NV^*$ mit $S \xrightarrow{*} uB\beta$ gilt:

Für $B\beta \longrightarrow_L \alpha\beta \xrightarrow{*} v$
und $B\beta \longrightarrow_L \alpha'\beta \xrightarrow{*} v'$ mit $\mathit{First}_k(v) = \mathit{First}_k(v')$ gilt stets $\alpha = \alpha'$.

Beispiel: $S \to \texttt{if}\ (E)\ S\ \texttt{else}\ S \mid \texttt{while}\ (E)\ S \mid E;$
$E \to id$

ist eine $LL(1)$–Grammatik.
Fügen wir jedoch die Regel $S \to \texttt{if}\ (E)\ S$ hinzu, ist die Grammatik nicht $LL(k)$ für beliebige $k \geq 1$ (siehe Übung).

Problem: Wie stellt man fest, ob eine Grammatik G eine $LL(k)$–Grammatik ist?

Zur Erinnerung: Für $\gamma = X_1 \cdots X_n \in V^*$
ist $\mathit{First}_k(\gamma) = \{\mathit{First}_k(w) \mid \gamma \xrightarrow{*} w\} = \mathit{First}_k(X_1) \oplus_k \cdots \oplus_k \mathit{First}_k(X_n)$ (s. Seite 47).

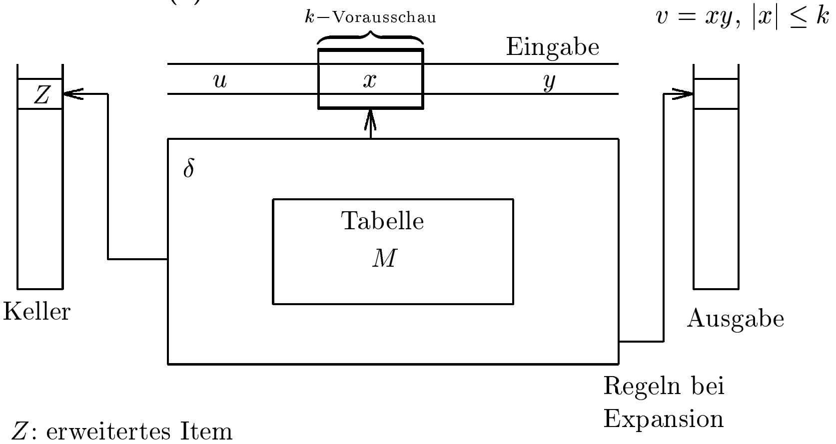
Satz:
Eine reduzierte KFG G ist genau dann $LL(k)$, wenn für alle $B \to \alpha, B \to \alpha' \in P$ mit $\alpha \neq \alpha'$ und alle $\beta \in V^*$ mit $S \xrightarrow{*} uB\beta$ gilt: $\mathit{First}_k(\alpha\beta) \cap \mathit{First}_k(\alpha'\beta) = \emptyset$ □

Da $\mathit{First}_k(\alpha\beta) = \mathit{First}_k(\alpha) \oplus_k \mathit{First}_k(\beta)$ gilt, schließen wir daraus, daß uns die β's nur im Hinblick darauf interessieren, welche First_k–Mengen sie liefern.

Diese Beobachtung ermöglicht es uns, den $LL(k)$–Parser herzustellen.

4.4.1 Der $LL(k)$-Parser

Schema des $LL(k)$-Parsers:



Z: erweitertes Item
$M[\langle B, L\rangle, x]$ liefert die Alternative für die Expansion

Aufbau der Parsertabelle M:

- In der ersten Komponente wird die Parsertabelle M jeweils mit Paaren $\langle B, L\rangle$ indiziert, wobei $B \in N$ und $L = \mathit{First}_k(\beta) \subseteq \mathit{Follow}_k(B) \subseteq T^{\leq k}$ für eine Ableitung $S \xrightarrow{*} uB\beta$.
- Die zweite Komponente von M wird mit Wörtern $x \in T^{\leq k}$, $k > 0$ indiziert.
  Für einen Tabelleneintrag gilt dann:
  $M[\langle B, L\rangle, x] = B \to \gamma \iff x \in \mathit{First}_k(\gamma) \oplus_k \mathit{First}_k(\delta) \oplus_k L$ für $A \to \alpha B\delta \in P$.

Berechnungsschritte des Parsers:

Die Berechnung der Mengen $L = \mathit{First}_k(\beta)$ erfolgt "online" mithilfe des Kellers.
Dazu erweitern wir Items durch Mengen $L, L' \subseteq T^{\leq k}$ wie folgt:

- Start–Item: $[S' \to .S,\ \{\varepsilon\}]$
- Shift: $([A \to \alpha_1.a\alpha_2,\ L],\ a,\ [A \to \alpha_1 a.\alpha_2,\ L])$
- Expansion: $([A \to \alpha_1.B\alpha_2,\ L'],\ \varepsilon,\ [B \to .\gamma,\ L][A \to \alpha_1.B\alpha_2,\ L'])$,
  wobei $L = \mathit{First}_k(\alpha_2) \oplus_k L'$
- Reduktion: $([B \to \gamma.,\ L'][A \to \alpha_1.B\alpha_2,\ L],\ \varepsilon,\ [A \to \alpha_1 B.\alpha_2,\ L])$

Man kann zeigen: $([S' \to .S,\ \{\varepsilon\}],\ u) \overset{*}{\vdash} ([A \to \alpha.\beta,\ L]\rho,\ \varepsilon) \implies L = \mathit{First}_k(F(\rho))$.

Nachteil dieser etwas trickreichen Konstruktion ist der aufwendige Aufbau einer (eventuell sehr großen) Tabelle !

Wir betrachten daher den Fall $k = 1$, um zu sehen, ob sich Vereinfachungen ergeben.

Satz:

Eine reduzierte KFG G ist genau dann $LL(1)$, wenn für alle Regeln $B \to \alpha$, $B \to \alpha' \in P$ mit $\alpha \neq \alpha'$ gilt: $(\mathit{First}_1(\alpha) \oplus_1 \mathit{Follow}_1(B)) \cap (\mathit{First}_1(\alpha') \oplus_1 \mathit{Follow}_1(B)) = \emptyset$.

Beweis:

"$\Leftarrow$" Sei $S \xrightarrow{*}_L uB\beta$. Da $First_1(\beta) \subseteq Follow_1(B)$ folgt aus der Monotonie von $\oplus_k$:

$$First_1(\alpha) \oplus_1 First_1(\beta) \subseteq First_1(\alpha) \oplus_1 Follow_1(B)$$
und
$$First_1(\alpha') \oplus_1 First_1(\beta) \subseteq First_1(\alpha') \oplus_1 Follow_1(B)$$

Daher gilt: $First_1(\alpha\beta) \cap First_1(\alpha'\beta) = \emptyset$.
Nach dem letzten Satz gilt nun (für $k = 1$), daß G eine $LL(1)$–Grammatik ist.

"$\Rightarrow$" Fall 1: $\varepsilon \notin First_1(\alpha)$
Dann ist $First_1(\alpha) \oplus_1 Follow_1(B) = First_1(\alpha) = First_1(\alpha\beta')$ für beliebige $\beta' \in V^*$.
Sei $S \xrightarrow{*}_L uB\beta$ mit $B \to \alpha, B \to \alpha' \in P$, dann gilt:

$$\begin{aligned}
&First_1(\alpha) \oplus_1 Follow_1(B) \;\cap\; First_1(\alpha') \oplus_1 Follow_1(B)\\
&= First_1(\alpha) \cap First_1(\alpha') \oplus_1 Follow_1(B)\\
&= First_1(\alpha) \cap First_1(\alpha') \oplus_1 \bigcup_{S \xrightarrow{*}_L uB\beta} First_1(\beta)\\
&= First_1(\alpha) \cap \bigcup_{S \xrightarrow{*}_L uB\beta} First_1(\alpha') \oplus_1 First_1(\beta)\\
&= First_1(\alpha) \cap \bigcup_{S \xrightarrow{*}_L uB\beta} First_1(\alpha'\beta)\\
&= \bigcup_{S \xrightarrow{*}_L uB\beta} First_1(\alpha) \cap First_1(\alpha'\beta) \;=\; \bigcup_{S \xrightarrow{*}_L uB\beta} First_1(\alpha\beta) \cap First_1(\alpha'\beta)
\end{aligned}$$

Aus dem letzten Satz folgt

$$\bigcup_{S \xrightarrow{*}_L uB\beta} First_1(\alpha\beta) \cap First_1(\alpha'\beta) \;=\; \bigcup_{S \xrightarrow{*}_L uB\beta} \emptyset \;=\; \emptyset.$$

Fall 2: $\varepsilon \notin First_1(\alpha')$ analog zu 1.

Fall 3: $\varepsilon \in First_1(\alpha)$ und $\varepsilon \in First_1(\alpha')$
$\Longrightarrow$ G ist **nicht** $LL(1)$, da $First_1(\beta) \subseteq First_1(\alpha\beta) \cap First_1(\alpha'\beta)$.

□

Beachte: Für $k > 1$ gilt nur die Rückrichtung des Beweises.

Definition: Eine Grammatik heißt *stark–$LL(k)$*, falls für alle $B \to \alpha$, $B \to \alpha' \in P$ mit $\alpha \neq \alpha'$ gilt: $(First_k(\alpha) \oplus_k Follow_k(B)) \cap (First_k(\alpha') \oplus_k Follow_k(B)) = \emptyset$.

Eine Grammatik, die stark–$LL(k)$ ist, ist somit auch $LL(k)$.
Eine Grammatik ist genau dann stark–$LL(1)$, wenn sie auch $LL(1)$ ist.

Der Parsers für stark–$LL(k)$–Grammatiken vereinfacht sich, da die Erweiterung der Items überflüssig wird.
Die Parsertabelle M wird in der ersten Komponente nur mit Nichtterminalen B indiziert.
Dabei sieht die Tabelle wie folgt aus: $M[B, x] = B \to \alpha \iff x \in First_k(\alpha) \oplus_k Follow_k(B)$.
Vorteil: Wir erhalten viel kleinere Tabellen und einfachere Items.

Beispiel: Die folgende Grammatik ist eine $LL(1)$–Grammatik:

$S \rightarrow$ `if` (E) S `else` S | $S \rightarrow$ `while` (E) S | $S \rightarrow E;$
$E \rightarrow$ `id`

M	`if`	`while`	`id`	`else`	(	)	;
S	$S \rightarrow$ `if` (E) S `else` S	$S \rightarrow$ `while` (E) S	$S \rightarrow E;$	-	-	-	-
E	-	-	$E \rightarrow$ `id`	-	-	-	-

Diese Tabelle enthält für jedes Token eine Spalte und für jedes Nichtterminal eine Zeile. Bearbeiten wir gerade ein Nichtterminal und lesen das nächste Token aus der Eingabe, so steht in der Tabelle in der Zeile des Nichtterminals und der Spalte des Tokens die Regel, die wir anwenden müssen.

Beispiel: Die folgende Grammatik ist eine $LL(2)$–Grammatik:

$S \rightarrow aAaa \mid bAba$
$A \rightarrow b \mid \varepsilon$

M	aa	ab	ba	bb	a	b	ε
$\langle S, \{\varepsilon\}\rangle$	$S \rightarrow aAaa$	$S \rightarrow aAaa$	–	$S \rightarrow bAba$	–	–	–
$\langle A, \{aa\}\rangle$	$A \rightarrow \varepsilon$	–	$A \rightarrow b$	–	–	–	–
$\langle A, \{ba\}\rangle$	–	–	$A \rightarrow \varepsilon$	$A \rightarrow b$	–	–	–

Diese Grammatik ist nicht stark–$LL(2)$, da
$\mathit{First}_2(b) \oplus_2 \mathit{Follow}_2(A) \cap \mathit{First}_2(\varepsilon) \oplus_2 \mathit{Follow}_2(A) = \{ba, bb\} \cap \{aa, ba\} \neq \emptyset$.

Definition: Eine kontextfreie Grammatik G heißt *linksrekursiv*, falls für ein $\beta \in V^*$ und ein $A \in N$ gilt: $A \xrightarrow{+}_L A\beta$.

Bemerkung:

1. Zu jeder reduzierten $LL(k)$–Grammatik G gibt es eine kontextfreie Grammatik G', die stark–$LL(k)$ ist, und für die gilt: $L(G) = L(G')$.
   Konstruktionsidee: Ersetze die Nichtterminale $A \in N$ durch Nichtterminale $\langle A, L\rangle$, $L \subseteq T^{\leq k}$, so daß $L = \mathit{First}_k(\beta)$ für das aktuelle β ist, d. h. wir machen A informativer.

2. Ist die kontextfreie Grammatik G linksrekursiv und reduziert, dann ist G mit Sicherheit **nicht** $LL(k)$ für beliebige $k \geq 1$.

   Beweis:

   Zur Vereinfachung nehmen wir an $A \rightarrow A\beta \in P$.
   Da A erreichbar ist, gilt: $S \xrightarrow{*}_L uA\gamma \xrightarrow{*}_L uA\beta^n\gamma$ für jedes $n \geq 0$.
   Da A produktiv ist, folgt: $\exists A \rightarrow \alpha \in P$ mit $\alpha \neq A\beta$.

   Annahme: Sei G $LL(k)$–Grammatik für ein beliebiges $k \geq 1$.
   Dann ist $\mathit{First}_k(\alpha\beta^n\gamma) \cap \mathit{First}_k(A\beta^{n+1}\gamma) = \emptyset$ (s. Satz Seite 56).
   Wegen $\mathit{First}_k(\alpha\beta^{n+1}\gamma) \subseteq \mathit{First}_k(A\beta^{n+1}\gamma)$, folgt $\mathit{First}_k(\alpha\beta^n\gamma) \cap \mathit{First}_k(\alpha\beta^{n+1}\gamma) = \emptyset$ für beliebige $n \geq 0$.
   Für $\beta \xrightarrow{*} w$ gibt es nun zwei Möglichkeiten:
   $w = \varepsilon$: $\mathit{First}_k(\alpha\beta^k\gamma) \supseteq \mathit{First}_k(\alpha\gamma) \subseteq \mathit{First}_k(\alpha\beta^{k+1}\gamma)$
   $\Longrightarrow$ Widerspruch!

$w \neq \varepsilon$: Wähle $n = k$!
$\Longrightarrow$ $First_k(\alpha\beta^k\gamma) \supseteq First_k(\alpha) \oplus_k First_k(w^k) \subseteq First_k(\alpha\beta^{k+1}\gamma)$
$\Longrightarrow$ Widerspruch! □

Leider ist es nicht immer möglich linksrekursive kontextfreie Grammatiken zu vermeiden. Ein Ausweg wäre, die Linksrekursivität zu beseitigen, was aber u. U. sehr teuer sein kann! Eine andere Möglichkeit wäre die Wahl eines anderen Parse–Verfahrens.
Im folgenden stellen wir ein weiteres Parse–Verfahren vor.

4.5 Bottom–Up–Syntaxanalyse

Das in diesem Abschnitt vorgestellte Bottom-Up-Verfahren ist die sogenannte $LR(k)$-Analyse. Sie deckt eine viel größere Klasse von kontextfreien Grammatik ab, als die $LL(k)$–Methode. Prinzipiell wird dabei eine Folge von Reduktionen gesucht, die eine Zeichenfolge auf das Startsymbol zurückführt

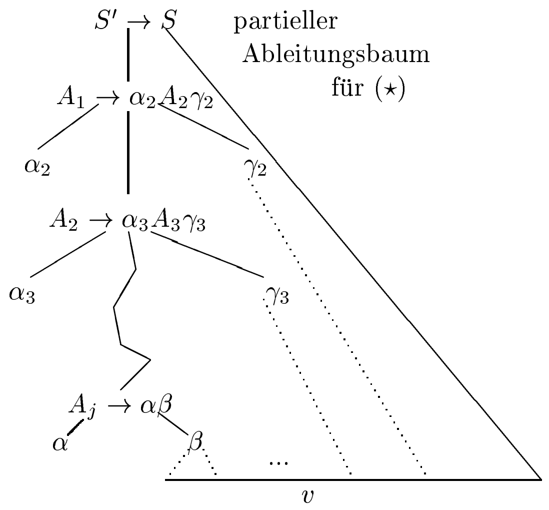
Drei Prinzipien zur $LR(k)$-Analyse sollen hier vorgestellt werden:

- Allgemeine (kanonische) $LR(k)$-Parser: Dies ist das mächtigste hier besprochene Verfahren. Allerdings ist die Implementierung sehr aufwendig.
- Simple $LR(k)$-Parser (*SLR(k)*): Die am einfachsten zu implementierende Methode unter diesen Verfahren. Sie funktioniert aber nur für eine kleine Klasse von Grammatiken.
- Lookahead $LR(k)$-Parser (*LALR(k)*): Dieses Verfahren liegt etwa zwischen den beiden anderen Verfahren (im Parser-Generator `yacc` verwendet).

Sei $([S' \to .S], uv) \overset{*}{\vdash} ([A_j \to \alpha.\beta]\rho, v)$, wobei $\rho = [A_{j-1} \to \alpha_j.A_j\gamma_j] \cdots [A_0 \to \alpha_1.A_1\gamma_1]$ mit $[A_0 \to \alpha_1.A_1\gamma_1] = [S' \to .S] \quad (\star)$.
Die Geschichte von ρ ist $H(\rho) = \alpha_2 \cdots \alpha_j$.

Das Wort $H(\rho)\alpha$ heißt *zuverlässiges Präfix* von G und das Item $[A \to \alpha.\beta]$ nennen wir *gültiges Item* für das zuverlässige Präfix $H(\rho)\alpha$.

Mit Hilfe des sogenannten charakteristischen ε–NFA's zu G versuchen wir nun ein zuverlässiges Präfix zu erkennen, welches in einem vollständigen Item (d. h. einem Item der Form $[A \to \alpha.]$) endet.



4.5.1 Der $LR(G)$–Automat

Zur Verwaltung der zuverlässigen Präfixe verwenden wir, ausgehend vom Itemkellerautomaten für die Grammatik G (s. Seite 44), den charakteristischen ε–NFA $c(G)$.

<u>Definition:</u> Der *charakteristischer ε–Automat* $c(G)$ für eine Grammatik G ist definiert als Quintupel $c(G) = (Q, V, \delta, q_0, F)$ mit:

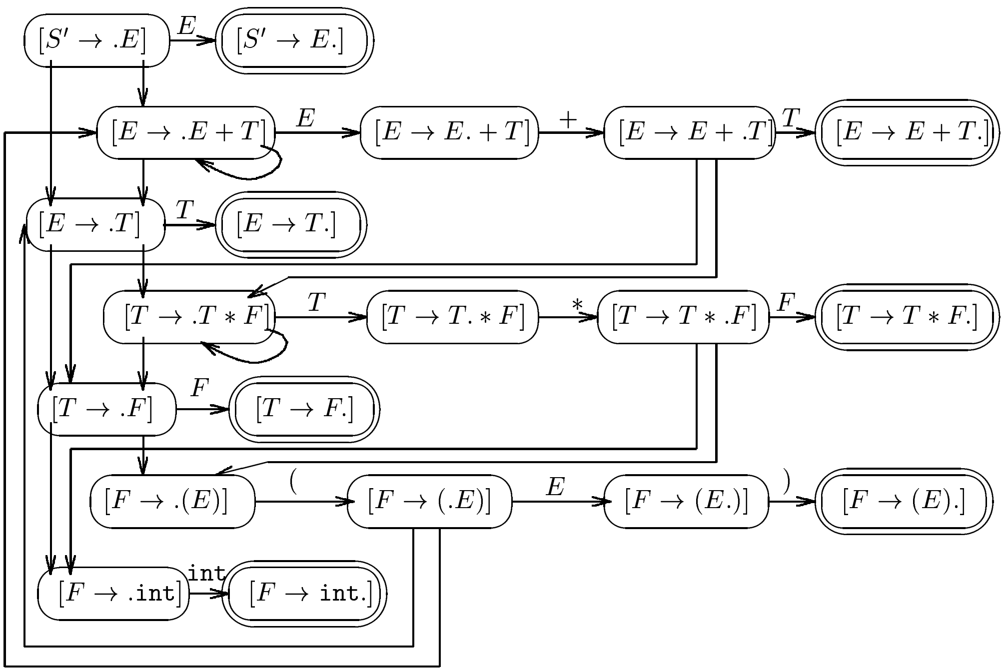
- $Q = I_{G'}$ (Menge der Items zu $G' = (N \cup \{S'\}, T, P \cup \{S' \to S\}, S')$, $S' \notin N$)
- $q_0 = [S' \to .S] \in Q$
- $F = \{[A \to \alpha.] \mid A \to \alpha \in P \cup \{S' \to S\}\}$
- δ besteht aus den Übergängen
  $([A \to \alpha.X\beta], X, [A \to \alpha X.\beta])$ mit $[A \to \alpha.X\beta] \in Q$, $X \in V$
  und $([A \to \alpha.B\beta], \varepsilon, [B \to .\gamma])$ für $[A \to \alpha.B\beta] \in Q$, $B \to \gamma \in P$.

<u>Beispiel:</u> Betrachte die folgende Grammatik:

$S' \to E$
$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid \texttt{int}$

Der charakteristische ε–NFA für diese Grammatik hat die folgende Gestalt:

Unmarkierte Pfeile im Diagramm symbolisieren ε–Übergänge.



<u>Satz:</u> $(q_0, \gamma, q) \in \delta^* \iff \gamma$ ist zuverlässiges Präfix und q ist gültiges Item für γ. □

Der kanonische $LR(0)$–Automat $LR(G)$ zu einer Grammatik G, ist der DFA, den man erhält, indem man den zu $c(G)$ äquivalenten DFA konstruiert. Bei der Kontruktion von $LR(G)$ werden **nach** dem Lesen eines Symbols $X \in V$ noch alle ε–Übergänge ausgeführt.
Fassen wir dies durch die folgenden Definitionen formaler.

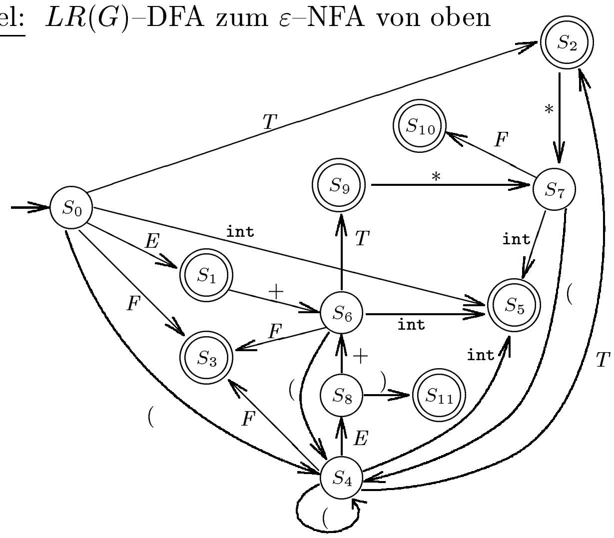
<u>Definition:</u> Für eine Menge $S \subseteq Q$ ist $\delta_\varepsilon^*(S)$ die kleinste Menge T für die gilt:
$S \subseteq T$ und $\forall B \to \gamma \in P : [A \to \alpha.B\beta] \in T \implies [B \to .\gamma] \in T$.

Definition: Der *kanonische $LR(0)$–Automat* $LR(G)$ zu einer Grammatik G, ist ein Quintupel $LR(G) = (Q', V, \delta', q_0', F')$ mit

- $Q' = 2^Q$
- $q_0' = \delta_\varepsilon^*(\{[S' \to .S]\})$
- $F' = \{S \subseteq Q \mid [A \to \gamma.] \in S\}$.
- Für $S \subseteq Q$, $X \in V$ ist die Übergangsfunktion δ' definiert durch
  $\delta'(S, X) = \delta_\varepsilon^*(\{[A \to \alpha X.\beta] \mid [A \to \alpha.X\beta] \in S\})$.

Beachte: Alle Items eines Zustands $\delta'(S, X)$ haben entweder den Punkt hinter ein und demselben $X \in N \cup T$, oder aber ganz am Anfang.

Beispiel: $LR(G)$–DFA zum ε–NFA von oben



Zustände des Automaten:

$$
\begin{aligned}
S_0 &= \{\ [S' \to .E],\ [E \to .E+T],\ [E \to .T],\ [T \to .T*F],\ [T \to .F],\\
&\qquad [F \to .(E)],\ [F \to .\mathtt{int}]\ \}\\
S_1 &= \{\ [S' \to E.],\ [E \to E.+T]\ \}\\
S_2 &= \{\ [E \to T.],\ [T \to T.*F]\ \}\\
S_3 &= \{\ [T \to F.]\ \}\\
S_4 &= \{\ [F \to (.E)],\ [E \to .E+T],\ [E \to .T],\ [T \to .T*F],\ [T \to .F],\\
&\qquad [F \to .(E)],\ [F \to .\mathtt{int}]\ \}\\
S_5 &= \{\ [F \to \mathtt{int}.]\ \}\\
S_6 &= \{\ [E \to E+.T],\ [T \to .T*F],\ [T \to .F],\ [F \to .(E)],\ [F \to .\mathtt{int}]\ \}\\
S_7 &= \{\ [T \to T*.F],\ [F \to .(E)],\ [F \to .\mathtt{int}]\ \}\\
S_8 &= \{\ [F \to (E.)],\ [E \to E.+T]\ \}\\
S_9 &= \{\ [E \to E+T.],\ [T \to T.*F]\ \}\\
S_{10} &= \{\ [T \to T*F.]\ \}\\
S_{11} &= \{\ [F \to (E).]\ \}
\end{aligned}
$$

Wir konstruieren nun aus $LR(G)$ den Kellerautomaten $\bar{\mathcal{A}}_G$. Die Idee hierbei ist folgende:

- $\bar{\mathcal{A}}_G$ rekonstruiert den Ableitungsbaum bottom-up (entspricht einer reversen Rechtsableitung).

- Im Keller stehen zuverlässige Präfixe (codiert).
- $\bar{\mathcal{A}}_G$ erkennt mittels $LR(G)$ Reduktionen.
- Um den Zustand von $LR(G)$ auch nach der Reduktion für das aktuelle zuverlässige Präfix zu berechnen, kellern wir nicht nur dieses Präfix, sondern auch für jedes Symbol den erreichten Zustand in $LR(G)$.

Dieser (i. a. nichtdeterministische) geeignet konstruierte Kellerautomat $\bar{\mathcal{A}}_G$ kann nun, durch Auswerten von Transitionen im kanonischen $LR(0)$-Automaten, Worte der Grammatik G erkennen.
Dabei kellert er Zustände des $LR(0)$-Automaten.

<u>Definition:</u> Der Kellerautomat $\bar{\mathcal{A}}_G = (\bar{Q}, T, \bar{\delta}, \bar{q_0}, \{f\})$ ist definiert durch:

- $\bar{Q} = Q' \cup \{f\}$, wobei f ein neuer Zustand ist.
- $\bar{q_0} = q_0'$
- $\bar{\delta}$ besteht aus den Übergängen:

  Shift: $(q,\, a,\, \delta'(q,a)q)$, $a \in T$, (d. h. $[B \to \alpha.a\beta] \in q$)
  Reduce: $(q_m \cdots q_1 q,\, \varepsilon,\, \delta'(q,B)q)$, falls $[B \to X_1 \cdots X_m.] \in q_m$ $(\in F')$,
  Stop: $(\delta'(q_0, S')q_0,\, \varepsilon,\, f)$.

- **Shift**-Übergang:
  Übergänge mit Terminalsymbolen im kanonischen $LR(0)$-Automaten zeigen an, daß in einer Produktion der Grammatik ein Terminalsymbol a abgearbeitet wurde (Übergang von einem Item $[A \to \alpha.a\beta]$ zu einem Item $[A \to \alpha a.\beta]$ im Itemkellerautomaten).
  Der Kellerautomat $\bar{\mathcal{A}}_G$ "kellert" nun, ausgehend vom obersten Zustand q seines Kellers, auch den Nachfolgezustand $\delta'(q,a)$ (Transition im $LR(0)$-Automaten) und konsumiert das Terminal a der Eingabe.

- **Reduce**-Übergang:
  Um die Bedeutung von Übergängen mit Nichtterminalsymbolen für den Kellerautomaten zu erklären, müssen wir zunächst die Endzustände des kanonischen $LR(0)$-Automaten betrachten. In einem Endzustand des kanonischen $LR(0)$-Automaten wissen wir, daß eine Produktion der Grammatik ganz "abgearbeitet" wurde (der Endzustand enthält ein Item der Form $[B \to X_1 \cdots X_m.]$).
  Für B wurde folglich schon ein Wort u mit $B \overset{*}{\to} u$ hergeleitet (und damit in der Eingabe konsumiert).
  $\bar{\mathcal{A}}_G$ kann nun die zur Herleitung von u verwendeten Informationen (Zustände $q_m, \ldots, q_1$) verwerfen. Dabei bleibt als oberster Zustand des Kellers der Zustand q zurück, von dem aus die Abarbeitung des Wortes u ausgehend vom Nichtterminal B ursprünglich startete (d. h. $[A \to \alpha.B\beta] \in q$).
  Dies rechtfertigt nun einen Übergang vom Zustand q im $LR(0)$-Automaten durch Lesen des Nichtterminals B in den Zustand $p = \delta'(q,B)$ mit $[A \to \alpha B.\beta] \in p$. Dieser Zustand p wird nun von $\bar{\mathcal{A}}_G$ "gekellert".

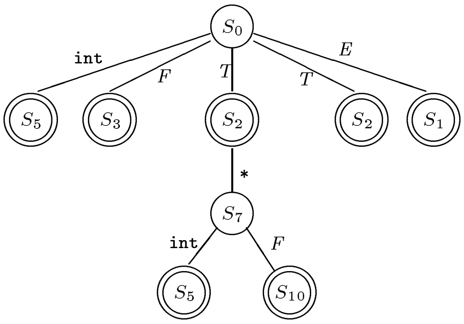
Fazit:
Der Kellerautomat $\bar{\mathcal{A}}_G$ führt zuerst einmal solange Shift-Übergänge durch Lesen von Terminalsymbolen $a_1 \cdots a_k = u$ durch, bis er einen Endzustand des $LR(0)$-Automaten erreicht hat (dies entspricht dem Auffinden einer Produktion im Ableitungsbaum, aus der ein Blatt b des Ableitungsbaums hergeleitet werden kann).
Durch einen Reduktionsschritt "findet" $\bar{\mathcal{A}}_G$ zu einem Zustand im $LR(0)$-Automaten zurück, an dem die Ableitung von u begann und legt sofort durch Lesen eines Nichtterminals B einen Zustand auf den Keller (dies entspricht einem Aufstieg im Ableitungsbaum von Blatt b aus zu einer Produktion $A \to \alpha B \beta$, wobei der Teilableitungsbaum zu B (der der Ableitung von u diente) schon untersucht wurde).
Von hier aus "sucht" $\bar{\mathcal{A}}_G$ wieder einen weiteren Endzustand durch Lesen von Terminalsymbolen im $LR(0)$-Automaten usw.

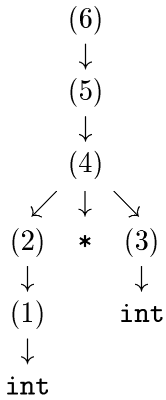
Im folgenden Beispiel werden die Kellerinhalte von $\mathcal{A}_G$ und $\bar{\mathcal{A}}_G$ für eine akzeptierende Berechnung des Wortes `int*int` gegenübergestellt.
X bezeichnet das vom $LR(G)$–DFA gelesene Zeichen.
Bei jeder Reduktion geben wir die entsprechende Produktion $p \equiv B \to X_1 \cdots X_m$ aus dem Endzustand an, für welchen die Reduktion ausgeführt wird.

Beispiel:

Keller v. $\mathcal{A}_G$	X	Keller v. $\bar{\mathcal{A}}_G$	p
$[S' \to .E]$		S_0	
$[E \to .T][S' \to .E]$		S_0	
$[T \to .T * F][E \to .T][S' \to .E]$		S_0	
$[T \to .F][T \to .T * F][E \to .T][S' \to .E]$		S_0	
$[F \to .\mathtt{int}][T \to .F][T \to .T * F][E \to .T][S' \to .E]$		S_0	
$[F \to \mathtt{int}.][T \to .F][T \to .T * F][E \to .T][S' \to .E]$	`int`	S_5S_0	$F \to \mathtt{int}$ (1)
$[T \to F.][T \to .T * F][E \to .T][S' \to .E]$	F	S_3S_0	$T \to F$ (2)
$[T \to T. * F][E \to .T][S' \to .E]$	T	S_2S_0	
$[T \to T * .F][E \to .T][S' \to .E]$	$*$	$S_7S_2S_0$	
$[F \to .\mathtt{int}][T \to T * .F][E \to .T][S' \to .E]$		$S_7S_2S_0$	
$[F \to \mathtt{int}.][T \to T * .F][E \to .T][S' \to .E]$	`int`	$S_5S_7S_2S_0$	$F \to \mathtt{int}$ (3)
$[T \to T * F.][E \to .T][S' \to .E]$	F	$S_{10}S_7S_2S_0$	$T \to T * F$ (4)
$[E \to T.][S' \to .E]$	T	S_2S_0	$E \to T$ (5)
$[S' \to E.]$	E	S_1S_0	$S' \to E$ (6)
		f	

Veranschaulicht man die Art und Weise, in der der Kellerautomat die Zustände des $LR(0)$-Automaten besucht, so entspricht dies einem Links-Rechts-DFS-Durchlauf des unten links abgebildeten Graphen.
Man erkennt, daß der Zustand S_2 eine Ursache für das nichtdeterministische Verhalten von $\bar{\mathcal{A}}_G$ ist: einerseits ist S_2 Endzustand, d. h. $\bar{\mathcal{A}}_G$ kann eine Reduktion durchführen, andererseits kann von hier aus ein Shift-Übergang durch Lesen von `*` stattfinden (dies gilt übrigens auch für die Zustände S_1 und S_9).





Die Ausgabereihenfolge der Produktionen p entspricht einem links-rechts Postorderdurchlauf (ohne Blätter) im oben rechts abgebildeten Ableitungsbaum für das Wort `int*int`.

Korrektheit: Wir betrachten im folgenden die Abbildungen

$$\begin{array}{rl} & H(istory)\text{: Kellerinhalt} \longrightarrow \text{zuverlässiges Präfix} \\ \text{und} & K(eller)\text{: zuverlässiges Präfix} \longrightarrow \text{Kellerinhalt.} \end{array}$$

$H : Q^+ \to V^*$ ist partiell definiert durch $< \begin{array}{l} H(q_0) = \varepsilon \\ H(q\pi) = H(\pi)X, \text{ falls } [A \to \alpha X.\beta] \in q \end{array}$.

$K : V^* \to Q^+$ ist definiert durch $< \begin{array}{l} K(\varepsilon) = q_0 \\ K(\alpha X) = \delta'(q, X)K(\alpha), \text{ falls } K(\alpha) = q\pi \end{array}$.

Die Korrektheit der Konstruktion folgt mit anschließendem Satz und Korollar.

<u>Satz:</u>

(1) Aus $(q_0, u) \overset{*}{\vdash}_{\bar{\mathcal{A}}_G} (\rho, \varepsilon)$ folgt $H(\rho) \overset{*}{\longrightarrow} u$.

(2) Ist $\delta'(q, X)$ definiert und gilt $X \overset{*}{\longrightarrow} w$, dann gilt auch $(q, w) \overset{*}{\vdash}_{\bar{\mathcal{A}}_G} (\delta'(q, X)q, \varepsilon)$.

<u>Beweis:</u>

(1) Induktion nach der Länge n der Berechnung:
$n = 0 : \rho = q_0,\ u = \varepsilon \ \surd$
$n \geq 1 :$ Sei $(q_0, u) \overset{*}{\vdash}_{\bar{\mathcal{A}}_G} (\rho', x) \vdash_{\bar{\mathcal{A}}_G} (\rho, \varepsilon), u = vx$
Falls der letzte Übergang **Shift** war, gilt:
$x \in T,\ \rho' = q\rho'',\ \rho = \delta'(q, x)\rho' \implies H(\rho) = H(\rho')x.$
$\overset{\text{IV}}{\implies} H(\rho') \overset{*}{\longrightarrow} v \implies H(\rho) \overset{*}{\longrightarrow} vx = u \implies$ Beh.
Falls der letzte Übergang **Reduce** war, gilt:
$x = \varepsilon,\ \rho' = q_m \cdots q_1 q\rho''$ mit $[B \to X_1 \cdots X_m.] \in q_m,\ \rho = \delta'(q, B)q\rho''.$
$\overset{\text{IV}}{\implies} H(\rho') \overset{*}{\longrightarrow} v = u.$
Da $H(\rho') = H(q\rho'')X_1 \cdots X_m$ gilt, folgt
$H(\rho) = H(q\rho'')B \longrightarrow_R H(q\rho'')X_1 \cdots X_m \overset{*}{\longrightarrow} u. \implies$ Beh.

(2) Induktion nach der Länge n der Ableitung:
$n = 0 : X = w = a \in T \implies$ **Shift**–Übergang: $(q, a) \vdash_{\bar{\mathcal{A}}_G} (\delta'(q, a)q, \varepsilon) \ \surd$

$n \geq 1$: Sei $X \to X_1 \cdots X_m \xrightarrow{*}_L w = w_1 \cdots w_m$ mit Ableitungen $X_j \xrightarrow{*} w_j$ für $j = 1, \ldots, m$.
$\overset{\text{IV}}{\Longrightarrow}$ Wann immer $\delta'(p, X_j)$ definiert ist, folgt $(p, w_j) \overset{*}{\vdash}_{\bar{\mathcal{A}}_G} (\delta'(p, X_j)p, \varepsilon)$.
Konstruiere eine Folge $p_0, \ldots, p_m \in Q$ wie folgt:
$p_0 = q$ und $p_j = \delta'(p_{j-1}, X_j)$ für $j > 0$. Diese Folge ist wohldefiniert, da:
$\delta'(p_0, X)$ definiert $\Longrightarrow \exists\, [A \to \alpha.X\beta] \in p_0 \Longrightarrow [X \to .X_1 \cdots X_m] \in p_0$
$\Longrightarrow [X \to X_1 \cdots X_{j-1}.X_j \cdots X_m] \in p_{j-1}$
Damit folgt:
$(q, w) = (p_0, w_1 \cdots w_m) \overset{*}{\vdash}_{\bar{\mathcal{A}}_G} (p_1 p_0, w_2 \cdots w_m) \overset{*}{\vdash}_{\bar{\mathcal{A}}_G} \cdots \overset{*}{\vdash}_{\bar{\mathcal{A}}_G} (p_m \cdots p_0, \varepsilon)$
Reduce–Übergang: $(p_m \cdots p_0, \varepsilon) \vdash_{\bar{\mathcal{A}}_G} (\delta'(q, X)q, \varepsilon) \Longrightarrow$ Beh. □

<u>Korollar:</u> Für eine KFG G gilt $L(G) = L(\bar{\mathcal{A}}_G)$.

<u>Beweis:</u>

"$\supseteq$": Sei $u \in L(\bar{\mathcal{A}}_G)$. Dann gilt $(q_0, u) \overset{*}{\vdash}_{\bar{\mathcal{A}}_G} (\delta'(q_0, S)q_0, \varepsilon) \overset{(1)}{\Longrightarrow} H(\delta'(q_0, S)q_0) \xrightarrow{*}_G u$.
Da $H(\delta'(q_0, S)q_0) = S$, können wir offensichtlich u aus S ableiten. $\Longrightarrow$ Beh.

"$\subseteq$": Sei $u \in L(G)$. Dann gilt $S \xrightarrow{*} u$.
Da $\delta'(q_0, S)$ definiert ist, folgt aus obigem Satz, Teil (2):
$(q_0, u) \overset{*}{\vdash}_{\bar{\mathcal{A}}_G} (\delta'(q_0, S)q_0, \varepsilon) \vdash_{\bar{\mathcal{A}}_G} (f, \varepsilon)$. $\Longrightarrow$ Beh. □

Wir erwähnten bereits, daß der Kellerautomat $\bar{\mathcal{A}}_G$ i. a. leider nichtdeterministisch ist.

In den beiden folgenden Fällen gibt es mehrere Möglichkeiten für Übergänge in $\bar{\mathcal{A}}_G$:

1. Wir haben die Auswahl zwischen mehreren Reduktionen.
   D. h. es gibt Zustände in $LR(G)$ mit $[A \to \gamma.]$, $[A' \to \gamma'.] \in q$ für $A \to \gamma \neq A' \to \gamma'$.
   Wir bezeichnen dies als Reduce-Reduce-Konflikt.

2. Wir haben die Auswahl zwischen Shift-Übergängen und Reduktionen.
   D. h. es gibt Zustände in $LR(G)$ mit $[A \to \gamma.]$, $[A' \to \alpha.a\beta] \in q$
   Wir bezeichnen dies als Shift-Reduce-Konflikt.

Solche Zustände heißen *ungeeignet* (in unserem Beispiel die Shift-Reduce-Konflikte S_1, S_2, S_9). Wir benutzen die k-Vorausschau, um diese Konflikte aufzulösen. Falls keine Vorausschau nötig ist (d. h. $k = 0$), dann ist die Grammatik eine $LR(0)$-Grammatik.

4.5.2 $LR(k)$–Grammatiken

<u>Definition:</u> Eine kontextfreie Grammatik G heißt *$LR(k)$–Grammatik*, falls gilt:
Für $S' \xrightarrow{*}_R \alpha A w \longrightarrow_R \alpha\beta w$
und $S' \xrightarrow{*}_R \gamma B u \longrightarrow_R \alpha\beta v$ mit $First_k(w) = First_k(v)$ gilt $\alpha = \gamma$, $A = B$ und $u = v$.
Dabei ist G' die um die zusätzliche Regel $S' \to S$ erweiterte Grammatik G.

Beispiele:

1) Sei G_1 die Grammatik mit Produktionen $S \to A \mid B$
$A \to aAb \mid 0$
$B \to aBbb \mid 1$

G_1 ist nicht $LL(k)$ für alle $k \geq 0$, denn es existieren Wörter mit mehr als k Symbolen a als Präfix, von denen wir nicht wissen, mit welcher der beiden Regeln das a erzeugt wurde. G_1 ist aber $LR(0)$.

2) Sei G_2 die Grammatik mit Produktionen $S \to aAc$
$A \to Abb \mid b$

G_2 ist nicht $LL(k)$ für alle $k \geq 0$, denn G_2 ist linksrekursiv.
G_2 ist aber $LR(0)$, denn $\alpha\underline{\beta}$ ist von einer der Formen $a\underline{b}$, $a\underline{Abb}$, $\underline{aAc}$.

3) Sei G_3 die Grammatik mit Produktionen $S \to aAc$
$A \to bbA \mid b$

G_3 ist $LR(1)$, denn sei $S \overset{*}{\longrightarrow}_R \alpha X w \longrightarrow_R \alpha\beta w$ mit $First_1(w) = y$. Dann ist $\alpha\underline{\beta}y$ von einer der Formen $ab^{2k}\underline{b}c$, $ab^{2k}\underline{bbA}c$, $\underline{aAc}\varepsilon$.
Wir können sozusagen mit einer Vorausschau von einem Zeichen das letzte Vorkommen von b bestimmen.

4) Sei G_4 die Grammatik mit Produktionen $S \to aAc$
$A \to bAb \mid b$

G_4 ist nicht $LR(k)$ für alle $k \geq 0$. Denn:

Annahme: G_4 ist $LR(k)$ für ein beliebiges, aber festes k.
Betrachte die Ableitungen
$S \overset{*}{\longrightarrow}_R \underbrace{ab^kAb^kc}_{\alpha A w} \longrightarrow_R \underbrace{ab^k\underline{b}b^kc}_{\alpha\underline{\beta} w}$ und $S \overset{*}{\longrightarrow}_R \underbrace{ab^{k+1}Ab^{k+1}c}_{\gamma A u} \longrightarrow_R \underbrace{ab^k\underline{b}b^{k+2}c}_{\alpha\underline{\beta} v}$.
Offensichtlich ist $u \neq v$ ein Widerspruch zur Annahme der $LR(k)$–Eigenschaft.

Bei dieser Ableitung ist man nicht in der Lage, das mittlere b zu bestimmen und eine passende Reduktion zu finden.

Satz: Eine reduzierte kontextfreie Grammatik G ist genau dann $LR(0)$, wenn $LR(G)$ keine ungeeigneten Zustände enthält.

Beweis:

" $\Rightarrow$ ": Annahme: $LR(G)$ enthält einen ungeeigneten Zustand q.

Fall 1: Sei $[A \to \gamma.]$, $[A' \to \gamma'.] \in q$.
Dann existieren zwei verschiedene Berechnungen eines nichtdeterministischen Automaten für dasselbe zuverlässige Präfix, d. h.
$S' \overset{*}{\longrightarrow}_R \alpha A w \longrightarrow_R \alpha\gamma w$
und $S' \overset{*}{\longrightarrow}_R \alpha' A' w' \longrightarrow_R \alpha'\gamma' w'$ mit $\alpha\gamma = \alpha'\gamma'$ und $A \to \gamma \neq A' \to \gamma'$.
Dies ist ein Widerspruch zur $LR(0)$–Eigenschaft von G.

Fall 2: Sei $[A \to \gamma.]$, $[A' \to \beta.a\delta] \in q$.
Dann existieren zwei Ableitungen
$$S' \xrightarrow{*}_R \alpha A w \longrightarrow_R \alpha\gamma w$$
und $S' \xrightarrow{*}_R \alpha' A' u \longrightarrow_R \alpha'\beta a\delta u \xrightarrow{*}_R \alpha'\beta avu$ mit $\alpha\gamma = \alpha'\beta$.
Betrachte nun den letzten Rechtsableitungsschritt der zweiten Ableitung:
Gilt $\delta = v$, dann ist $u \neq a\delta u$. $\Longrightarrow$ Widerspruch zur $LR(0)$–Eigenschaft.
Ist $\delta \neq v$, dann betrachte $\delta \xrightarrow{*}_R v_1 X v_2 \longrightarrow_R v$.
Nun gilt $S' \xrightarrow{*}_R \alpha'\beta av_1Xv_2u = \alpha\gamma av_1Xv_2u \longrightarrow_R \alpha\gamma avu$ mit $v_2u \neq avu$.
$\Longrightarrow$ Widerspruch zur $LR(0)$–Eigenschaft.

" $\Leftarrow$ ": $LR(G)$ enthalte keine ungeeigneten Zustände.
Betrachte nun die beiden Ableitungen $S' \xrightarrow{*}_R \alpha A w \longrightarrow_R \alpha\beta w$ und
$S' \xrightarrow{*}_R \gamma B u \longrightarrow_R \alpha\beta v$.
Bleibt die $LR(0)$–Eigenschaft zu zeigen, d. h. $\alpha = \gamma,\ A = B,\ v = u$.
Offenbar ist $\alpha\beta$ zuverlässiges Präfix.
Daher gibt es einen Zustand $p = \delta'^*(q_0, \alpha\beta)$, der nicht ungeeignet ist und daher genau ein Reduce–Item $[A \to \beta.]$ enthält und kein Shift–Item.
Wäre nun $(\alpha, A, v) \neq (\gamma, B, u)$, dann enthielte p ein weiteres vollständiges Item oder Shift–Item. Also gilt $\alpha = \gamma,\ A = B,\ v = u$.
$\Longrightarrow$ G ist $LR(0)$. □

4.5.3 Der $LR(G, k)$–Automat

Für den Fall $k > 0$ müssen wir unseren Kellerautomaten modifizieren. Die Idee dazu ist, die Items mit k–Vorausschaumengen auszustatten. Wir repräsentieren hierbei ein erweitertes Item der Form $[A \to \alpha.\beta, L]$ durch die Items $[A \to \alpha.\beta, x],\ x \in L$.

Definition: Die Menge $I_{G,k}$ der $LR(k)$–Items für G besteht aus allen Paaren $i = [A \to \alpha.\beta, x]$ mit $A \to \alpha\beta \in P$ und $x \in \mathit{Follow}_k(A)$. $A \to \alpha.\beta$ heißt *Kern* des Items i.
Gilt $S' \xrightarrow{*}_R \gamma A w \longrightarrow_R \gamma\alpha\beta w$ mit $x \in \mathit{First}_k(w)$, dann heißt $\gamma\alpha$ *zuverlässiges Präfix mit i als gültigem $LR(k)$–Item*.

Beispiel: $S' \to E, \quad E \to E + T \mid T, \quad T \to T * F \mid F, \quad F \to (E) \mid \texttt{int}$

$S' \xrightarrow{*}_R (E) \longrightarrow_R (E + T) \xrightarrow{*}_R (E + \texttt{int}) \longrightarrow_R (E + T + \texttt{int})$
Das zuverlässige Präfix $(E+$ hat als gültiges $LR(1)$–Item $[E \to E + .T, +]$

Definition: Der *charakteristische ε–Automat $c(G,k) = (Q, V, \delta, q_0, F)$ zu G* wird definiert durch:

- $Q = I_{G,k}$ Menge der $LR(k)$–Items
- $q_0 = [S' \to .S, \varepsilon]$
- $F = \{[A \to \gamma., x] \mid A \to \gamma \in P,\ x \in \mathit{Follow}_k(A)\}$
- δ wird definiert durch:
  $([A \to \alpha.Y\beta, x], Y, [A \to \alpha Y.\beta, x])$
  und $([A \to \alpha.B\beta, x], \varepsilon, [B \to .\gamma, y])$ mit $B \to \gamma \in P$ und $y \in \mathit{First}_k(\beta x)$.

Bemerkung: Die zweite Komponente der erweiterten Items ändert sich nur bei ε–Übergängen, d. h. Expansionen.

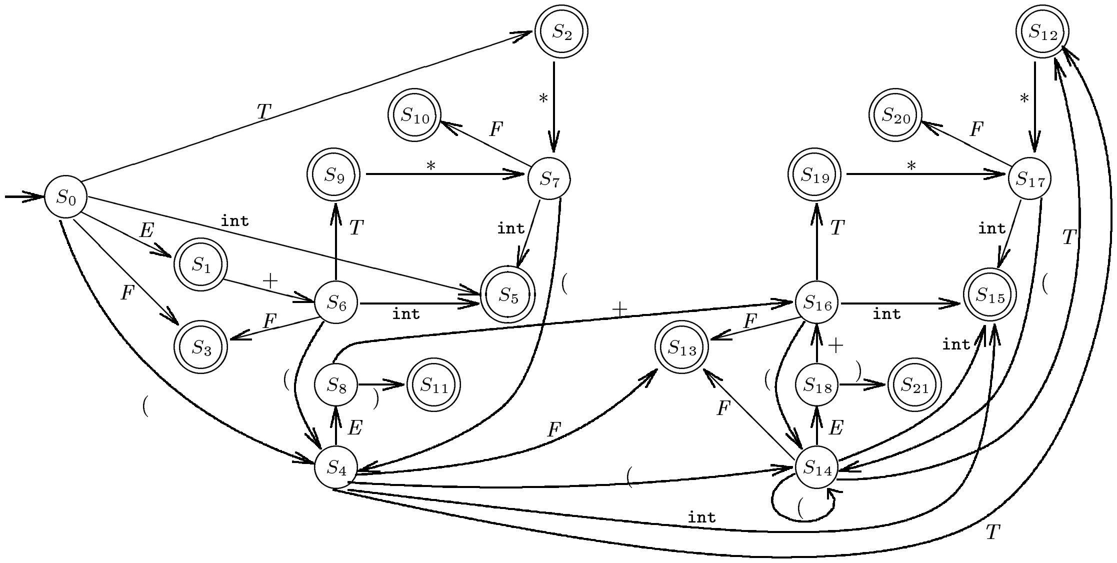
Definition: Der *kanonische $LR(k)$–Automat zu G* ist der DFA $LR(G,k)$ zu $c(G,k)$

Bemerkung: $c(G)$ entspricht $c(G,0)$ und $LR(G)$ entspricht $LR(G,0)$.
Bei der DFA–Konstruktion führen wir beim Übergang $\delta'(q,X)$ zuerst alle möglichen Shift–Übergänge unter X auf den Items in q durch. Dann erst fügen wir alle weiteren Items hinzu, die durch ε–Übergänge (Expansionen) erreicht werden können.
Der Übersichtlichkeit halber fassen wir alle Items eines Zustands mit gleichem Kern $A \to \alpha.\beta$ zusammen. D. h.: $[A \to \alpha.\beta, L] \in q \iff \{[A \to \alpha.\beta, x] \mid x \in L\} \subseteq q$.

Beispiel: $S' \to E, \quad E \to E + T \mid T, \quad T \to T * F \mid F, \quad F \to (E) \mid \texttt{int}$

$$
\begin{aligned}
S_0 &= \{\, [S' \to .E,\ \{\varepsilon\}\,],\ [E \to .E+T,\ \{\varepsilon,+\}\,],\ [E \to .T,\ \{\varepsilon,+\}\,],\\
&\quad\ [E \to .T*F,\ \{\varepsilon,+,*\}\,],\ [T \to .F,\ \{\varepsilon,+,*\}\,],\ [F \to .(E),\ \{\varepsilon,+,*\}\,],\\
&\quad\ [F \to .\texttt{int},\ \{\varepsilon,+,*\}\,]\,\}\\
S_1 &= \{\, [S' \to E.,\ \{\varepsilon\}\,],\ [E \to E.+T,\ \{\varepsilon,+\}\,]\,\}\\
S_2 &= \{\, [E \to T.,\ \{\varepsilon,+\}\,],\ [T \to T.*F,\ \{\varepsilon,+,*\}\,]\,\}\\
S_3 &= \{\, [T \to F.,\ \{\varepsilon,+,*\}\,]\,\}\\
S_4 &= \{\, [F \to (.E),\ \{\varepsilon,+,*\}\,],\ [E \to .E+T,\ \{),+\}\,],\ [E \to .T,\ \{),+\}\,],\\
&\quad\ [T \to .T*F,\ \{),+,*\}\,],\ [T \to .F,\ \{),+,*\}\,],\ [F \to .(E),\ \{),+,*\}\,],\\
&\quad\ [F \to .\texttt{int},\ \{),+,*\}\,]\,\}\\
S_5 &= \{\, [F \to \texttt{int}.,\ \{\varepsilon,+,*\}\,]\,\}\\
S_6 &= \{\, [E \to E+.T,\ \{\varepsilon,+\}\,],\ [T \to .T*F,\ \{\varepsilon,+,*\}\,],\ [T \to .F,\ \{\varepsilon,+,*\}\,],\\
&\quad\ [F \to .(E),\ \{\varepsilon,+,*\}\,],\ [F \to .\texttt{int},\ \{\varepsilon,+,*\}\,]\,\}\\
S_7 &= \{\, [T \to T*.F,\ \{\varepsilon,+,*\}\,],\ [F \to .(E),\ \{\varepsilon,+,*\}\,],\ [F \to .\texttt{int},\ \{\varepsilon,+,*\}\,]\,\}\\
S_8 &= \{\, [F \to (E.),\ \{\varepsilon,+,*\}\,],\ [E \to E.+T,\ \{),+\}\,]\,\}\\
S_9 &= \{\, [E \to E+T.,\ \{\varepsilon,+\}\,],\ [T \to T.*F,\ \{\varepsilon,+,*\}\,]\,\}\\
S_{10} &= \{\, [T \to T*F.,\ \{\varepsilon,+,*\}\,]\,\}\\
S_{11} &= \{\, [F \to (E).,\ \{\varepsilon,+,*\}\,]\,\}\\
S_{12} &= \{\, [E \to T.,\ \{),+\}\,],\ [T \to T.*F,\ \{),+,*\}\,]\,\}\\
S_{13} &= \{\, [T \to F.,\ \{),+,*\}\,]\,\}\\
S_{14} &= \{\, [F \to (.E),\ \{),+,*\}\,],\ [E \to .E+T,\ \{),+\}\,],\ [E \to .T,\ \{),+\}\,],\\
&\quad\ [T \to .T*F,\ \{),+,*\}\,],\ [T \to .F,\ \{),+,*\}\,],\ [F \to .(E),\ \{),+,*\}\,],\\
&\quad\ [F \to .\texttt{int},\ \{),+,*\}\,]\,\}\\
S_{15} &= \{\, [F \to \texttt{int}.,\ \{),+,*\}\,]\,\}\\
S_{16} &= \{\, [E \to E+.T,\ \{),+\}\,],\ [T \to .T*F,\ \{),+,*\}\,],\ [T \to .F,\ \{),+,*\}\,],\\
&\quad\ [F \to .(E),\ \{),+,*\}\,],\ [F \to .\texttt{int},\ \{),+,*\}\,]\,\}\\
S_{17} &= \{\, [T \to T*.F,\ \{),+,*\}\,],\ [F \to .(E),\ \{),+,*\}\,],\ [F \to .\texttt{int},\ \{),+,*\}\,]\,\}\\
S_{18} &= \{\, [F \to (E.),\ \{),+,*\}\,],\ [E \to E.+T,\ \{),+\}\,]\,\}\\
S_{19} &= \{\, [E \to E+T.,\ \{),+\}\,],\ [T \to T.*F,\ \{),+,*\}\,]\,\}\\
S_{20} &= \{\, [T \to T*F.,\ \{),+,*\}\,]\,\}\\
S_{21} &= \{\, [F \to (E).,\ \{),+,*\}\,]\,\}
\end{aligned}
$$

Beispiel: $LR(G,1)$–DFA für $S' \to E, \quad E \to E + T \mid T,$
$T \to T * F \mid F, \quad F \to (E) \mid \texttt{int}$



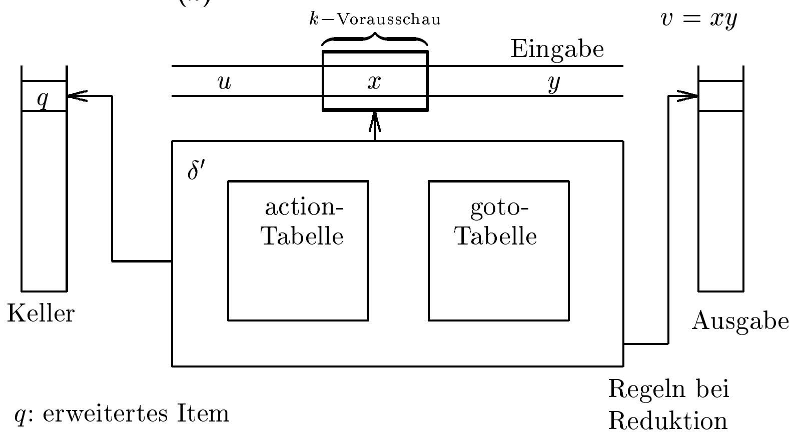
Definition: Ein Zustand q des Automaten $LR(G,k)$ heißt $LR(k)$–*ungeeignet*, falls gilt: $[A \to \gamma., x]$, $[B \to \alpha.\beta, y] \in q$ mit $A \to \gamma. \neq B \to \alpha.\beta$ und $x \in \mathit{First}_k(\beta y)$.

Im Beispiel zum $LR(0)$–Automaten waren die Zustände S_1, S_2 und S_9 ungeeignet. Im $LR(1)$–Automaten dazu sind sie allerdings nicht $LR(1)$–ungeeignet, d. h. die Vorausschau ermöglicht es uns, Konflikte aufzulösen. Mögliche Shift–Reduce–Konflikte in den Zuständen S_{12} und S_{19} werden ebenfalls durch die Vorausschau beseitigt.

4.5.4 $LR(k)$–Parser

Mit Kenntnis des $LR(G,k)$-Automaten und des Kellerautomaten $\bar{\mathcal{A}}_G$ können wir jetzt einen $LR(k)$–Parser konstruieren.

Schema des $LR(k)$-Parsers:



Die action–Tabelle hat dabei folgende Form:

	$x \in T^{\leq k}$
$q \in Q$	actions

Die action–Tabelle bestimmt die möglichen Shift– bzw. Reduce–Aktionen des $LR(G,k)$–Automaten durch Vorausschau von k Zeichen:

Shift (Sh): Lesen des nächsten Eingabesymbols im Zustand q.

Reduce ($A \to \alpha$): Reduziere gemäß der Regel $A \to \alpha$ bei Vorausschau eines Wortes $x \in T^{\leq k}$, falls $[A \to \alpha., x] \in q$. Gib inbesondere $A \to \alpha$ aus.

Accept (Ac): Leere den Keller und beende die Ausführung (d. h. $[S' \to S., \varepsilon] \in q$).

Error (-): Abbruch wegen Fehler (in der Praxis ist ein Abbruch natürlich zu vermeiden!).

Die goto–Tabelle beschreibt die Übergänge des $LR(G,k)$–Automaten:

	$X \in V$
$q \in Q$	$\delta'(q,X)$

Satz: (ohne Beweis) Eine reduzierte kontextfreie Grammatik G ist genau dann $LR(k)$, wenn $LR(G,k)$ keinen $LR(k)$–ungeeigneten Zustand enthält.

Beispiel: action–Tabelle für $S' \to E, \quad E \to E+T \mid T,$
$T \to T*F \mid F, \quad F \to (E) \mid \texttt{int}$

	`int`	(	)	$*$	$+$	ε
S_0	Sh	Sh	–	–	–	–
S_1	–	–	–	–	Sh	Ac
S_2	–	–	–	Sh	$E \to T$	$E \to T$
S_3	–	–	–	$T \to F$	$T \to F$	$T \to F$
S_4	Sh	Sh	–	–	–	–
S_5	–	–	–	$F \to \texttt{int}$	$F \to \texttt{int}$	$F \to \texttt{int}$
S_6	Sh	Sh	–	–	–	–
S_7	Sh	Sh	–	–	–	–
S_8	–	–	Sh	–	Sh	–
S_9	–	–	–	Sh	$E \to E+T$	$E \to E+T$
S_{10}	–	–	–	$T \to T*F$	$T \to T*F$	$T \to T*F$
S_{11}	–	–	–	$F \to (E)$	$F \to (E)$	$F \to (E)$
S_{12}	–	–	$E \to T$	Sh	$E \to T$	–
S_{13}	–	–	$T \to F$	$T \to F$	$T \to F$	–
S_{14}	Sh	Sh	–	–	–	–
S_{15}	–	–	$F \to \texttt{int}$	$F \to \texttt{int}$	$F \to \texttt{int}$	–
S_{16}	Sh	Sh	–	–	–	–
S_{17}	Sh	Sh	–	–	–	–
S_{18}	–	–	Sh	–	Sh	–
S_{19}	–	–	$E \to E+T$	Sh	$E \to E+T$	–
S_{20}	–	–	$T \to T*F$	$T \to T*F$	$T \to T*F$	–
S_{21}	–	–	$F \to (E)$	$F \to (E)$	$F \to (E)$	–

Die Konstruktion des $LR(k)$–Parsers ist *erschöpfend*, d. h., falls die Reduktionen eindeutig mit k–Vorausschau festgelegt sind, funktioniert unsere Parser–Konstruktion.
Allerdings ist der $LR(G,k)$–Automat i. a. sehr viel größer als der zugehörige $LR(G)$–Automat und damit ineffizient. Deshalb betrachtet man Teilklassen, zu denen viel kleinere Parser konstruiert werden können.
Die Idee hierbei ist, unter Verwendung des $LR(G)$–Automaten zu versuchen, die Konflikte der ungeeigneten Zustände mittels Vorausschau aufzulösen. Dazu stattet man die Items in den Zuständen des $LR(G)$–Automaten **nachträglich** mit Vorausschau–Mengen aus. Dies führt uns zu den Simple–$LR(k)$– und Lookahead–$LR(k)$–Parsern ($SLR(k)$– bzw. $LALR(k)$–Parser).

4.5.5 *SLR*(k)–Parser

Hier ist die Zuordnung von Vorausschau–Mengen zu Zuständen unabhängig vom Kontext, d. h. den Items $[A \to \alpha.\beta]$ wird in allen Zuständen jeweils die Vorausschau–Menge $Follow_k(A)$ zugeordnet.

Definition: Ein Zustand q des $LR(G)$–Automaten heißt genau dann $SLR(k)$*–ungeeignet*, wenn für zwei Zustände $[A \to \gamma.]$, $[B \to \alpha.\beta] \in q$ gilt:
$A \to \gamma. \neq B \to \alpha.\beta$ und $Follow_k(A) \cap (First_k(\beta) \oplus_k Follow_k(B)) \neq \emptyset$.
Eine Grammatik heißt $SLR(k)$, falls sie keine $SLR(k)$–ungeeigneten Zustände enthält.

Falls alle Zustände $SLR(k)$–geeignet sind, können wir auf folgende Weise einen Parser bauen:
Die goto–Tabelle ist die goto–Tabelle des $LR(0)$–Parsers, d. h. sie enthält die Zustandsübergangsfunktion δ' des $LR(G)$–Automaten.
Die action–Tabelle ist konstruierbar, da alle Konflikte gelöst werden können. Sie wird mit den Zuständen des $LR(G)$–Automaten und Worten aus $T^{\leq k}$ indiziert.

Beispiel: $S' \to E, \quad E \to E+T \mid T, \quad T \to T*F \mid F, \quad F \to (E) \mid \texttt{int}$

Die $Follow_1$–Mengen zu dieser Grammatik wurden auf Seite 54 berechnet.
Untersuchen wir die ungeigneten Zustände des $LR(G)$–Automaten von Seite 62:
$S_1 = \{[S' \to E.], [E \to E.+T]\}$:
$Follow_1(S') \cap First_1(+) \oplus_1 Follow_1(E) = \{\varepsilon\} \cap \{+\} = \emptyset$
$S_2 = \{[E \to T.], [T \to T.*F]\}$ und $S_9 = \{[E \to E+T.], [T \to T.*F]\}$:
$Follow_1(E) \cap First_1(*) \oplus_1 Follow_1(T) = \{\varepsilon, +,)\} \cap \{*\} = \emptyset$
Die Zustände S_1, S_2 und S_9 sind offenbar nicht $SLR(1)$–ungeignet.
Damit ist unsere Beispiel–Grammatik eine $SLR(1)$–Grammatik.

Bemerkung: Die Programmiersprache C ist keine $SLR(k)$–Sprache!

4.5.6 *LALR*(k)–Parser

Hier ist die Zuordnung von Vorausschau–Mengen zu Zuständen abhängig vom Kontext, d. h. ein Item erhält in verschiedenen Zuständen eventuell verschiedene Vorausschau–Mengen.

Definition: Für einen Zustand q von $LR(G)$ mit einem Item $[A \to \alpha.\beta] \in q$ definieren wir die Menge $\lambda(q, [A \to \alpha.\beta]) = \{First_k(w) \mid S' \xrightarrow{*}_R \gamma A w \text{ und } \delta'^*(q_0, \gamma\alpha) = q\}$.

Beachte: $\lambda(q, [A \to \alpha.\beta]) \subseteq \mathit{Follow}_k(A)$.

Definition: Ein Zustand q des $LR(G)$–Automaten heißt genau dann $LALR(k)$–*ungeeignet*, wenn q Items $[A \to \gamma.]$ und $[B \to \alpha.\beta]$ mit $A \to \gamma. \neq B \to \alpha.\beta$ enthält, für die gilt:
$\lambda(q, [A \to \gamma.]) \cap (\mathit{First}_k(\beta) \oplus_k \lambda(q, [B \to \alpha.\beta])) \neq \emptyset$.
Eine Grammatik heißt $LALR(k)$, falls sie keine $LALR(k)$–ungeeigneten Zustände enthält.

Der $LR(0)$–Automat kann also mit einer entsprechenden action–Tabelle zum $LALR(k)$–Parser erweitert werden.

Berechnung der Mengen $\lambda(q, [A \to \alpha.\beta])$:

Wir stellen das folgende Constaint–System *LA1* auf:

$$\begin{array}{lll}
\lambda(q_0, [S' \to .S]) & \supseteq & \{\varepsilon\}, \\
\lambda(q, [A \to \alpha X.\beta]) & \supseteq & \lambda(p, [A \to \alpha.X\beta]) \text{ für } \delta'(p, X) = q, \\
\lambda(q, [A \to .\gamma]) & \supseteq & \mathit{First}_k(\beta) \oplus_k \lambda(q, [B \to \alpha.A\beta]) \text{ für } [B \to \alpha.A\beta] \in q
\end{array}$$

Folgende Beobachtung gestattet es uns, das Constrait–System *LA1* zu vereinfachen:

- $\lambda(q, [A \to .\gamma])$ ist unabhängig von γ.
- $\lambda(q, [A \to \alpha.\beta])$ ist "leicht" zu berechnen, wenn man alle Werte $\lambda(p, [A \to .\alpha\beta])$ kennt.

  Es gilt $\lambda(q, [A \to \alpha.\beta]) = \bigcup_{\delta'^*(p,\alpha)=q} \lambda(p, [A \to .\alpha\beta])$.

Dies gibt uns die Möglichkeit, das Constraint-System *LA1* auf ein Constraint-System *LA2* für $\hat{\lambda}(q, A) = \lambda(q, [A \to .\gamma])$ $(A \to \gamma \in P)$ zurückzuführen.

Wir erhalten das Constraint–System *LA2*:

$$\begin{array}{lll}
\hat{\lambda}(q_0, S') & \supseteq & \{\varepsilon\} \\
\hat{\lambda}(q, A) & \supseteq & \mathit{First}_k(\beta) \oplus_k \hat{\lambda}(p, B) \quad \text{für ein} \quad [B \to \alpha.A\beta] \in q,\ \delta'^*(p, \alpha) = q
\end{array}$$

Zwar müssen wir für das Constraint–Systems *LA2* weniger Variablen berechnen, allerdings besteht das Constraint–System aus eventuell sehr viel mehr Bedingungen als *LA1*.

Der Parser–Generator `yacc` arbeitet beispielsweise auf $LALR(1)$–Grammatiken.

$\boldsymbol{LALR}$(1)-Parser

Für diesen Spezialfall können wir die $\oplus_1$-Konkatenation durch die Vereinigung $\cup$ ersetzen. Sei $E(\beta) = 1 :\Longleftrightarrow \beta \xrightarrow{*} \varepsilon$ und $F_\varepsilon(\beta) = \mathit{First}_1(\beta) \setminus \{\varepsilon\}$ die ε–freie First_1–Menge zu β.

Wir erhalten wir das Constraint–System *LA3*:

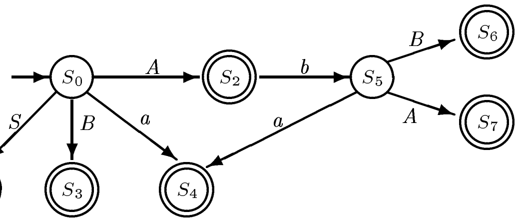
$$\begin{array}{lll}
\hat{\lambda}(q_0, S) & \supseteq & \{\varepsilon\} \\
\hat{\lambda}(q, A) & \supseteq & F_\varepsilon(\beta) \text{ für } [B \to \alpha.A\beta] \in q \\
\hat{\lambda}(q, A) & \supseteq & \hat{\lambda}(p, B), \text{ falls } [B \to \alpha.A\beta] \in q,\ \delta'^*(p, \alpha) = q \text{ und } E(\beta) = 1
\end{array}$$

Beispiel: Sei G die Grammatik mit Produktionen $S \to AbB \mid B, \quad A \to a, \quad B \to A.$

Der $LR(0)$–Automat zu G:

$$\begin{array}{lclcl}
S_0 & = & \delta'^*_\varepsilon([S' \to .S]) & = & \{[S' \to .S],[S \to .AbB],[S \to .B],[A \to .a],[B \to .A]\} \\
S_1 & = & \delta'(S_0,S) & = & \{[S' \to S.]\} \\
S_2 & = & \delta'(S_0,A) & = & \{[S \to A.bB],[B \to A.]\} \\
S_3 & = & \delta'(S_0,B) & = & \{[S \to B.]\} \\
S_4 & = & \delta'(S_0,a) & = & \{[A \to a.]\} \\
S_5 & = & \delta'(S_2,b) & = & \{[S \to Ab.B],[B \to .A],[A \to .a]\} \\
S_6 & = & \delta'(S_5,B) & = & \{[S \to AbB.]\} \\
S_7 & = & \delta'(S_5,A) & = & \{[B \to A.]\} \\
 & & \delta'(S_5,a) & = & S_4
\end{array}$$

Im Bild:



Nur der Zustand S_2 ist ungeeignet.

Wir müssen also lediglich $\lambda(S_2,[S \to A.bB])$ und $\lambda(S_2,[B \to A.])$ berechnen:

$\lambda(S_2,[S \to A.bB]) = \lambda(S_0,[S \to .AbB]) = \hat\lambda(S_0,S) = \{\varepsilon\}$

$\lambda(S_2,[B \to A.]) = \lambda(S_0,[B \to .A]) = \hat\lambda(S_0,B) = \{\varepsilon\}$

Es gilt: $\lambda(S_2,[B \to A.]) \cap (First_1(bB) \oplus_1 \lambda(S_2,[S \to A.bB])) = \{\varepsilon\} \cap \{b\} = \emptyset.$
G ist also eine $LALR(1)$–Grammatik.
Der Konflikt in S_2 wird durch die 1–Vorausschau aufgelöst: Falls wir als nächstes Symbol ε sehen, müssen wir einen Shift–Übergang ausführen, falls wir jedoch b sehen, müssen wir mit $B \to A$ reduzieren.

Kapitel 5

Semantische Analyse

Bis jetzt haben wir mittels Scanner und Parser einen Syntaxbaum für unser Quellprogramm erzeugt. Die nächste Phase der Compilierung besteht in der Überprüfung der nicht kontextfreien Eigenschaften des Programms ("statische Semantik").
Solche Eigenschaften sind beispielsweise:

1. Übereinstimmung der Namen von Kopf und Ende eines Modula-2 Modules,
2. Überprüfung, ob alle Variablen vor ihrer Benutzung deklariert wurden,
3. Typüberprüfungen.

Hierbei können die Punkte 1. und 2. i. a. noch durch geschickte Erweiterung des Syntaxbaumes überprüft werden.

Typen werden u. a. aus folgenden Gründen verwendet:

- Angabe des "Verwendungszwecks" eines Objektes und damit die Vermeidung fehlerhafter Benutzung des Objektes.
- Der Verwendungszweck muß nicht als sogenanntes *Tag* zur Laufzeit ständig mitgeführt werden. Damit wird die Programmierung stark erleichtert.
- Während der Laufzeit können Tests auf die Korrektheit der Tags entfallen. Dies führt zu effizienteren Programmen.

Im folgenden Abschnitt werden wir uns mit Typüberprüfungen auseinandersetzen.

5.1 Explizit getypte Sprachen

In imperativen Sprachen erhalten die Variablen und Funktionen bzw. Prozeduren bei ihrer Deklaration einen Typ zugewiesen. Dies ist insbesondere für die Speicherallokation wichtig. Typen werden durch Typausdrücke beschrieben, wie etwa:

- *Basistypen:* Boolean, Integer, Real, Character etc.,

- *Typkonstruktoren*, die auf andere Typen angewandt werden, wie etwa:
  - **Felder** $array(I, t)$ mit einem Indextyp I und einem Elementtyp t.
  - **Verbunde** $[a_1 : t_1, \ldots, a_n : t_n]$ mit den Selektoren a_i und den Komponententypen t_i.
  - **Zeiger** $\uparrow t$
  - **Funktionen** $t_1 \rightarrow t_2$ und **Prozeduren** $t_1 \rightarrow$ `void`.
    Um in Funktionen mehrere Argumente zuzulassen, betrachtet man Funktionen als einstellig mit einem Tupel $(t_1 \times \cdots \times t_n)$ als Argument.
- *Typnamen* A, die durch Typdefinitionen $A = t$ eingeführt werden. Diese werden zur Abkürzung, Übersichtlichkeit und zur Konstruktion rekursiver Typen eingesetzt.

Die folgenden Beispiele geben wir jeweils als Pascal–Codefragmente an.

<u>Beispiel:</u> Rekursiver Typ in Pascal

```
Type list = Record
                info : Integer;
                next : ↑list;
            End;
```

<u>Problem:</u> Gegeben ist eine Menge von Variablen–Deklarationen $\Gamma = \{x_1 : t_1, \ldots, x_n : t_n\}$. Überprüfe, ob ein Ausdruck e bezüglich Γ die Variablen **korrekt** verwendet, und berechne gleichzeitig den Ergebnistyp t für e.

Falls wir aus Γ den Typ für e berechnen (inferieren) können, so schreiben wir auch $\Gamma \vdash e : t$.

<u>Idee:</u> Wir geben Regeln an, wie sich der Typ eines Ausdrucks zu den Typen seiner Komponenten verhält.

Eine solche Regel besteht aus einem Namen, einer Voraussetzung und einem Schluß, den wir aus der Voraussetzung ziehen können.
Wir notieren dies folgendermaßen: Name $\frac{\text{Voraussetzung}}{\text{Schluß}}$

Es gibt folgende Axiome und Regeln:

Axiome: $\Gamma \vdash c : t$, falls c eine Konstante vom Typ t ist
$\Gamma \vdash x : t$, falls $x : t \in \Gamma$

Regeln:

SEL $\dfrac{\Gamma \vdash [a_1 : t_1, \ldots, a_n : t_n], \quad \Gamma \vdash e : t}{\Gamma \vdash e.a_i : t_i}$ (Selection)

REF $\dfrac{\Gamma \vdash e : \uparrow t}{\Gamma \vdash e\uparrow : t}$ (Reference)

APP $\dfrac{\Gamma \vdash e_1 : t_1 \rightarrow t_2, \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1(e_2) : t_2}$ (Application)

ARR $\dfrac{\Gamma \vdash e_1 : array(I, t), \quad \Gamma \vdash e_2 : I}{\Gamma \vdash e_1[e_2] : t}$ (Array)

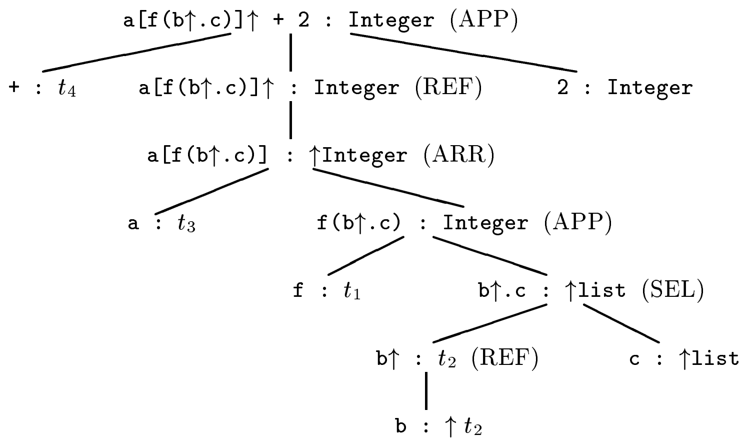
Mit diesen Regeln kann aus den Axiomen bottom–up ein Typausdruck für einen Ausdruck e über der Struktur von e berechnet werden.

Beispiel:

```
Function f(x: ↑list): Integer;      (Typ t1)
Begin
   ⋮
End;
Var b:↑Record                        (Typ ↑t2)
          c: ↑list;
        End;
    a: Array[0..10] of ↑Integer;    (Typ t3)
```

Wir wenden die Regeln auf den Ausdruck $e \equiv$ `a[f(b↑.c)]↑ + 2`, von den Blättern ausgehend, bottom–up an.
`+` ist hier eine Funktion vom Typ $t_4 =$ `Integer` $\times$ `Integer` $\rightarrow$ `Integer`.

Wir stellen die Ableitung mit den Regeln von oben als Baum dar, wobei an jedem Knoten in Klammern die verwendete Regel vermerkt ist, und erhalten:



Bei dieser Vorgehensweise treten folgende Probleme auf:

1. Konstanten können von verschiedenem Typ sein.
   So kann etwa `2` verschieden große `Integer`–Typen haben (etwa 2 oder 4 Byte). Eine Lösung wäre hier einen Defaulttyp vorzugeben. Beispielsweise könnte immer der kleinste passende Typ genommen werden.

2. Operatoren können mehrere Typen haben.
   Beispielsweise ist der Operator `+` entweder vom Typ `Integer` $\times$ `Integer` $\rightarrow$ `Integer` oder vom Typ `Real` $\times$ `Real` $\rightarrow$ `Real`. Man spricht in diesem Zusammenhang auch von *Überladung*. Eine Lösung ist hier die implizite Typkonversion, mit der die Typen der Argumente "passend" gemacht werden. Eine andere Lösung besteht in der Auswahl des "richtigen" Operators aufgrund der Argument–Typen.

3. Wir müssen zwei — eventuell rekursive — Typen vergleichen können, da bei Regelanwendungen ein Gleichheitstest auf Typen erforderlich wird.
   Für den Gleichheitstest gibt es verschiedene Strategien:

   - **Namensäquivalenz:** Hier findet ein rein textueller Vergleich statt.
     Definiert man etwa den Typ `test` als `Integer`, dann ist `test` $\neq$ `Integer`.

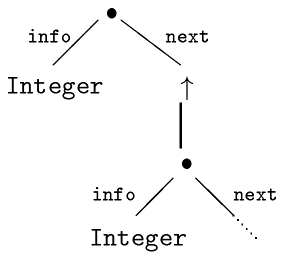
- **Strukturäquivalenz:** Hier repräsentiert ein Typname den Typausdruck, für den er eingeführt wurde. Man sagt auch, die Namen sind *transparent*.
  In diesem Fall ist `test` = `Integer`.

Achtung: Ein Typname kann auch einen unendlichen Typ–Ausdruck repräsentieren und ein unendlicher Typausdruck besitzt evtl. mehrere Darstellungen als Graph.

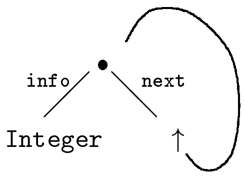
Beispiel: Ein Rekursiver Typ mit drei seiner Graphdarstellungen

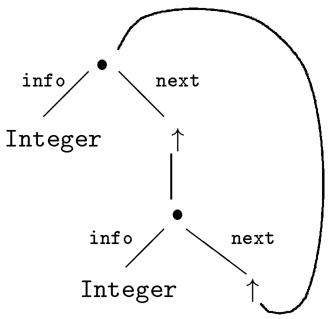
```
Type list = Record
                  info: Integer;
                  next: ↑list;
            End;
```

Dieser Typausdruck wird durch folgenden Graphen repräsentiert:



Es gibt aber noch weitere Repräsentationen dieses Ausdrucks, wie etwa:





Ein unendlicher Typausdruck t wird eindeutig charakterisiert durch die Menge der Pfade, die an der Wurzel starten. Für unser Beispiel ergibt sich:
$\langle \bullet, \texttt{info} \rangle$ `Integer`, $\langle \bullet, \texttt{next} \rangle \uparrow \langle \bullet, \texttt{info} \rangle$ `Integer`, ... (Pfade zu den Blättern)
$\bullet$, $\langle \bullet, \texttt{next} \rangle \uparrow$, $\langle \bullet, \texttt{next} \rangle \uparrow \bullet$, ... (Pfade zu den inneren Knoten)

Die Menge der Pfade ist regulär und daher durch einen minimalen DFA eindeutig charakterisiert. Der Gleichheitstest für zwei Typen kann damit durch einen Isomorphietest zweier DFAs gelöst werden, was äquivalent ist zu einem Isomorphietest von gerichteten geordneten endlichen Graphen.

Eine andere Strategie ist ein direkter Äquivalenztest ohne Minimierung des DFA. Ausgehend von einem System S von rekursiven Typ–Definitionen $A_1 = e_1; \ldots A_n = e_n;$ suchen wir für zwei Typ–Ausdrücke t_1, t_2 die Antwort auf die Frage: $t_1 \stackrel{?}{=} t_2$.
Für den direkten Äquivalenztest konstruieren wir einen Beweisbaum.

Verfahren:
Die Knoten des Beweisbaums beinhalten die zu überprüfenden Gleichheiten $s_1 \stackrel{?}{=} s_2$ und die Wurzel ist gerade $t_1 \stackrel{?}{=} t_2$. Ausgehend von der Wurzel tragen wir weitere Knoten gemäß den folgenden Regeln ein:

$$A_i \stackrel{?}{=} s \implies \begin{array}{c} A_i \stackrel{?}{=} s \\ | \\ e_i \stackrel{?}{=} s \end{array} \qquad\qquad s \stackrel{?}{=} A_i \implies \begin{array}{c} s \stackrel{?}{=} A_i \\ | \\ e_i \stackrel{?}{=} s \end{array}$$

$$[a_i : s_i] \stackrel{?}{=} [a_i : s_i'] \implies \begin{array}{ccc} & [a_i : s_i] \stackrel{?}{=} [a_i : s_i'] & \\ / & & \backslash \\ s_1 \stackrel{?}{=} s_1' & \cdots & s_n \stackrel{?}{=} s_n' \end{array}$$

$$s_1 \to s_2 \stackrel{?}{=} s_1' \to s_2' \implies \begin{array}{ccc} & s_1 \to s_2 \stackrel{?}{=} s_1' \to s_2' & \\ / & & \backslash \\ s_1 \stackrel{?}{=} s_1' & & s_2 \stackrel{?}{=} s_2' \end{array}$$

$$s \stackrel{?}{=} s \implies \begin{array}{c} s \stackrel{?}{=} s \\ | \\ \underline{s \stackrel{?}{=} s} \end{array}$$

Wir fügen einen Knoten nur dann in den Baum ein, falls er bisher nur einmal im Baum auftaucht. Knoten, die gleich einem ihrer Vorgänger sind, werden als akzeptierend markiert (hier unterstrichen) und nicht weiter expandiert. Falls ein Knoten noch nicht akzeptierend ist, aber auch keine Regel mehr anwendbar ist, sind die beiden Typen verschieden.
Wir stellen fest, daß in einer Komponente i ($i \in \{1, 2\}$) nur Teilausdrücke von t_i bzw. von den rechten Seiten e_j in den Typdefinitionen vorkommen können, d. h. die Anzahl der Knoten ist beschränkt durch $t_i + \sum_j |e_j|$. Damit ist die Tiefe des Baums beschränkt durch $(|t_1| + N) \cdot (|t_2| + N)$. Da der Verzweigungsgrad endlich ist, ist auch der Beweisbaum stets endlich. Daraus folgt, daß das Verfahren immer terminiert!

Algorithmus:

```
SetOfNodes L = {t₁ =? t₂};   // Worklist der Blätter

boolean equiv(void) {
SetOfNodes I = ∅;   // Menge der inneren Knoten des Entscheidungsbaums
Node       v;

  while (L != ∅) {
    v = extract(L);
    if (v ∉ I) {
      I = I ∪ {v};
      if (! expand(v)) return False;
      }
    }
  return True;
}
```

```
boolean expand(Node v) {
  switch(v) {
    case t ≟ t: return True;
    case Aᵢ = s: L = L ∪ {eᵢ ≟ s }; return True;
    case s = Aᵢ: L = L ∪ {s ≟ eᵢ }; return True;
    case s₁ → s₂ ≟ s'₁ → s'₂:
           L = L ∪ {s₁ ≟ s'₁, s₂ ≟ s'₂ }; return True;
    case [a₁ : s₁, ..., aₙ : sₙ] ≟ [a₁ : s'₁, ..., aₙ : s'ₙ]:
           L = L ∪ {sᵢ ≟ s'ᵢ | 1 ≤ i ≤ n}; return True;
    // weitere case für weitere Typkonstruktoren
    otherwise : return False;
    }
}
```

<u>Beispiel:</u> (`Int` steht abkürzend für `Integer`)

$A = [a{:}A;\ \ b{:}A \to \texttt{Int};\ \ c{:}\texttt{Int}]$
$B = [a{:}A;\ \ b{:}B;\ \ c{:}\texttt{Int}] \to \texttt{Int}$
Frage: $A \stackrel{?}{=} [a{:}A;\ \ b{:}B;\ \ c{:}\texttt{Int}]$

- $A \stackrel{?}{=} [a{:}A;\ \ b{:}B;\ \ c{:}\texttt{Int}]$
  - $[a{:}A;\ \ b{:}A \to \texttt{Int};\ \ c{:}\texttt{Int}] \stackrel{?}{=} [a{:}A;\ \ b{:}B;\ \ c{:}\texttt{Int}]$
    - $A \stackrel{?}{=} A$
      - $\underline{A \stackrel{?}{=} A}$
    - $A \to \texttt{Int} \stackrel{?}{=} B$
      - $A \to \texttt{Int} \stackrel{?}{=} [a{:}A;\ \ b{:}B;\ \ c{:}\texttt{Int}]$
        - $\underline{A \stackrel{?}{=} [a{:}A;\ \ b{:}B;\ \ c{:}\texttt{Int}]}$
        - $\texttt{Int} \stackrel{?}{=} \texttt{Int}$
          - $\underline{\texttt{Int} \stackrel{?}{=} \texttt{Int}}$
    - $\texttt{Int} \stackrel{?}{=} \texttt{Int}$
      - $\underline{\texttt{Int} \stackrel{?}{=} \texttt{Int}}$

Da alle Blätter akzeptieren, sind beide Typen gleich.

<u>Achtung:</u> Der Beweisbaum kann (theoretisch) exponentiell groß werden. Um ein polynomielles Verfahren zu erhalten, wird man versuchen isomorphe Teilbäume zu identifizieren. Es genügt nämlich offenbar zum Beweis einer Typanfrage $s \stackrel{?}{=} s'$, daß nur **irgendein** Vorgänger gleiche Beschriftung aufweist.

5.2 Subtypen

Subtypen sind ein wichtiges Konzept der objektorientierten Programmiersprachen. Die Idee hierbei ist es, bei Funktionen nicht den genauen Typ des Argumentes zu spezifizieren, sondern nur eine Mindestanforderung an den Typ zu stellen.

<u>Beispiel:</u>

```
Type t = Record
           size:Integer;
           info:String
         End;

Function ExtractInfo(Var x:t):String
Begin
   Return x.info
End;
```

Die Funktion `ExtractInfo` macht hier nicht nur für den Typ `t` Sinn, sondern auch für Argumente, die Verbunde sind, und eine Komponente `info` enthalten.

<u>Idee:</u> Spezifiziere die Funktion für den allgemeinsten Typ und gestatte ihre Anwendung auf alle Argumente, die einen Subtyp dieses Typs besitzen. Dabei sind Verbunde mit mehr Selektoren Subtypen von Verbunden mit weniger Selektoren!
Wir betrachten im weiteren die Menge T der Typausdrücke der folgenden Form:

$$t ::= \texttt{Boolean} \mid \texttt{Integer} \mid \texttt{Real} \mid \texttt{Void} \mid \ldots \mid t \times t \mid t \to t \mid \uparrow t \mid [a_1 : t_1, \ldots, a_n : t_n] \mid \ldots$$

Zur Konstruktion einer partiellen Ordnung auf T benutzen wir einerseits eine partielle Ordung auf den Basistypen (z. B.: `Boolean` $\leq$ `Integer` $\leq$ `Real`) und unterscheiden andererseits Verbunde mit mehr oder weniger Elementen.
Dies setzen wir jetzt auf beliebigen Typausdrücken in T fort:

$$\frac{t_1 \leq s_1 \quad t_2 \leq s_2}{t_1 \times t_2 \leq s_1 \times s_2}$$

$$\frac{t \leq s}{\uparrow t \leq \uparrow s}$$

$$\frac{\forall j \in \{1, \ldots, m\} \ \exists i : a_i \equiv b_j \ \wedge \ t_i \leq s_j}{[a_1 : t_1, \ldots, a_n : t_n] \leq [b_1 : s_1, \ldots, b_m : s_m]}$$

$$\frac{s_1 \leq t_1 \quad t_2 \leq s_2}{t_1 \to t_2 \leq s_1 \to s_2}$$

<u>Beispiel:</u>

$[\texttt{a:Integer, b:Boolean}] \leq [\texttt{a:Real}]$
`Integer` $\to$ `Integer` $\leq$ `Integer` $\to$ `Real`
`Real` $\to$ `Integer` $\leq$ `Integer` $\to$ `Integer`
Aber: `Real` $\to$ `Real` $\not\leq$ `Integer` $\to$ `Integer`
Damit ergibt sich das folgende Hasse–Diagramm:

$$\begin{tikzcd}[column sep=small]
 & \texttt{Integer} \to \texttt{Real} \arrow[dl, dash] \arrow[dr, dash] & \\
\texttt{Integer} \to \texttt{Integer} \arrow[dr, dash] & & \texttt{Real} \to \texttt{Real} \arrow[dl, dash] \\
 & \texttt{Real} \to \texttt{Integer} &
\end{tikzcd}$$

Zur Fortsetzung auf rekursiven Typen verallgemeinern wir die Idee zur Entscheidung der Gleichheit zweier Typen. Konzeptuell baut dieses Verfahren einen potentiell unendlichen Entscheidungsbaum mit Knoten der Form $s \leq t$ auf und akzeptiert, wenn sich kein Widerspruch zu den Subtyp-Regeln konstruieren läßt.

Beispiel:

$R_1 = [\texttt{a:Integer}, \texttt{f} : R_2 \to R_1]$
$R_2 = [\texttt{a:Integer}, \texttt{f} : S_2 \to R_2]$
$S_1 = [\texttt{a:Integer}, \texttt{b:Integer}, \texttt{f} : S_1 \to S_1]$
$S_2 = [\texttt{a:Integer}, \texttt{b:Integer}, \texttt{f} : R_2 \to S_2]$
Frage: $S_2 \leq S_1$?

$$
\begin{tikzcd}[arrows=dash]
 & & S_2 \leq S_1 \arrow[d] & \\
 & & R_2 \to S_2 \leq S_1 \to S_1 \arrow[dl] \arrow[dr] & \\
 & S_1 \leq R_2 \arrow[d] & & S_2 \leq S_1 \\
 & S_1 \to S_1 \leq S_2 \to R_2 \arrow[dl] \arrow[dr] & & \\
S_2 \leq S_1 & & S_1 \leq R_2 &
\end{tikzcd}
$$

Wir haben damit $S_2 \leq S_1$ gezeigt.
Weitere Beweise ergeben folgendes Hasse–Diagramm:

$$
\begin{tikzcd}[arrows=dash]
 & R_2 \arrow[dl] \arrow[dr] & \\
S_1 \arrow[dr] & & R_1 \arrow[dl] \\
 & S_2 &
\end{tikzcd}
$$

5.3 Implizite Typisierung und Überladung

5.3.1 Implizite Typisierung

Idee: Spezifiziere Datentypen (etwa Listen) durch Angabe der Konstruktion und erschließe dann die Typen der Variablen aus ihrer Verwendung.

Beispiel: Betrachte folgende funktionalen Programme:

```
data list = Nil
          | Cons int list

length Nil = 0
length (Cons x l) = 1 + length l

⇝ length: list → int
```

```
data list a = Nil
            | Cons a (list a)

length Nil = 0
length (Cons x l) = 1 + length l

⇝ length: (list a) → int
```

Vorteilhaft bei der impliziten Typisierung ist, daß der Programmierer weniger deklarieren muß. Im obigen Beispiel wird für jedes `a` ein “allgemeinster” Typ inferiert. Außerdem kann

man Funktionen und Datenstrukturen definieren, die für alle Typen geeignet sind (z. B. Listen beliebiger Elemente).
Diese sogenannte *Polymorphie* wurde etwa 1979 in ML implementiert.

5.3.2 Überladung

Idee: Nachempfindung der späten Bindung in objektorientierten Sprachen.
Ziel der Überladung ist es, eine Auswahl zwischen mehreren Funktions–Definitionen anhand des Typs des aktuellen Argumentes zu treffen.

Beispiel:

```
datastructure tree a = L a | N (tree a) (tree a)

map f [] = []
map f (x:xs) = f x : map f xs

map f (L a) = L (f a)
map f (N t1 t2) = N (map f t1) (map f t2)
```

Hierbei wird die Auswahl zwischen den beiden Definitionen von `map` anhand des zweiten Arguments getroffen. Überladung wurde etwa 1990 in Haskell implementiert.

Definition: Eine *Typklasse* ist eine Menge von Typen, für die eine bestimmte Funktion definiert ist. Eine *Instanz*-Deklaration eines Typs liefert eine Definition für die geforderte Funktion.

In unserem Beispiel sind etwa `list a` und `tree a` beide in der Typklasse von `map`.