

Helmut Seidl

Compilerbau

TU München

Sommersemester 2005

Inhaltsverzeichnis

0	Einführung	6
0.1	Prinzip eines Interpreters:	6
0.2	Prinzip eines Übersetzers:	6
0.3	Aufbau eines Übersetzers:	7
0.4	Aufgaben der Code-Erzeugung:	7
1	Die Architektur der CMa	9
2	Einfache Ausdrücke und Wertzuweisungen	10
3	Anweisungen und Anweisungsfolgen	14
4	Bedingte und iterative Anweisungen	15
4.1	Bedingte Anweisung, einseitig	16
4.2	Zweiseitiges if	16
4.3	while-Schleifen	17
4.4	for-Schleifen	18
4.5	Das switch-Statement	18
5	Speicherbelegung für Variablen	20
5.1	Felder	21
5.2	Strukturen	22
6	Zeiger und dynamische Speicherverwaltung	23
7	Zusammenfassung	26
8	Freigabe von Speicherplatz	28
9	Funktionen	29
9.1	Speicherorganisation für Funktionen	30
9.2	Bestimmung der Adress-Umgebung	31
9.3	Betreten und Verlassen von Funktionen	32
9.4	Zugriff auf Variablen, formale Parameter und Rückgabe von Werten	37
10	Übersetzung ganzer Programme	39

1	Die lexikalische Analyse	41
1.1	Grundlagen: Reguläre Ausdrücke	43
1.2	Grundlagen: Endliche Automaten	47
1.3	Design eines Scanners	61
1.4	Implementierung von DFAs	64
2	Die syntaktische Analyse	70
2.1	Grundlagen: Kontextfreie Grammatiken	71
2.2	Grundlagen: Kellerautomaten	83
2.3	Vorausschau-Mengen	92
2.4	Topdown Parsing	105
2.5	Schnelle Berechnung von Vorausschau-Mengen	114
2.6	Bottom-up Analyse	117
2.7	Spezielle Bottom-up-Verfahren mit <i>LR(G)</i>	133
3	Semantische Analyse	138
3.1	Symbol-Tabellen	139
3.2	Typ-Überprüfung	147
3.3	Inferieren von Typen	157
3.4	Attributierte Grammatiken	185
4	Die Optimierungsphase	194
5	Perspektiven	199
5.1	Hardware	199
5.2	Programmiersprachen	200
5.3	Programmierumgebungen	200
5.4	Neue Anforderungen	200
6	Instruktions-Selektion	201

Organisatorisches

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

Er findet zu den Compilerbauterminen statt :-)

Zeiten:

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

Einordnung:

Diplom-Studierende:

Compilerbau: Wahlpflichtveranstaltung
Abstrakte Maschinen: Vertiefende Vorlesung

Bachelor-Studierende:

Compilerbau: 8 ETCS-Punkte
Abstrakte Maschinen: nicht anrechenbar

Scheinerwerb:

- Diplom-Studierende:**
- 50% der Punkte;
 - zweimal Vorrechnen :-)
- Bachelor-Studierende:**
- Klausur
 - Erfolgreiches Lösen der Aufgaben wird zu 20% angerechnet :-))

Material:

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

Weitere Veranstaltungen:

- **Seminar** Theorem Beweisen — Di., 14:00-16:00 Uhr
- **Wahlpflicht-Praktikum:** Konstruktion eines Compilers
(Michael Petter)
- SEPs ...

0 Einführung

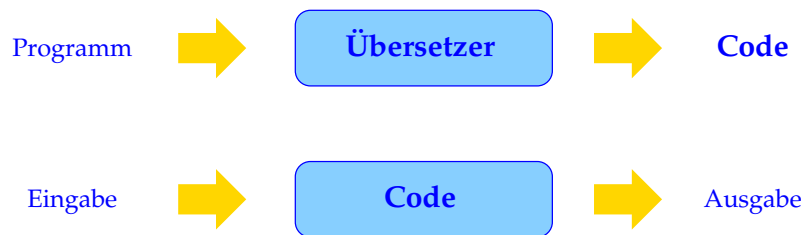
0.1 Prinzip eines Interpreters:



Vorteil: Keine Vorberechnung auf dem Programmtext erforderlich
=> keine/geringe Startup-Zeit :-)

Nachteil: Während der Ausführung werden die Programm-Bestandteile immer wieder analysiert => längere Laufzeit :-(

0.2 Prinzip eines Übersetzters:



Zwei Phasen:

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

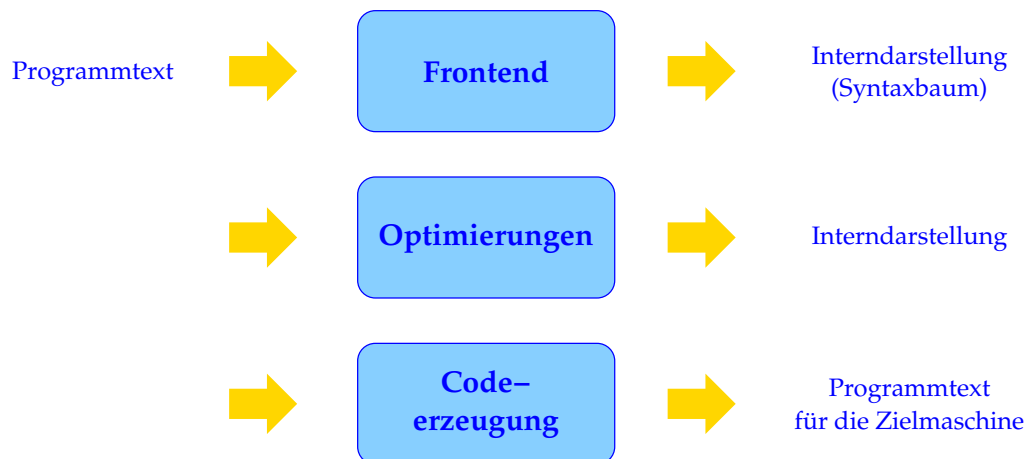
Eine Vorberechnung auf dem Programm gestattet u.a.

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

Nachteil: Die Übersetzung selbst dauert einige Zeit :-(

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

0.3 Aufbau eines Übersetzers:



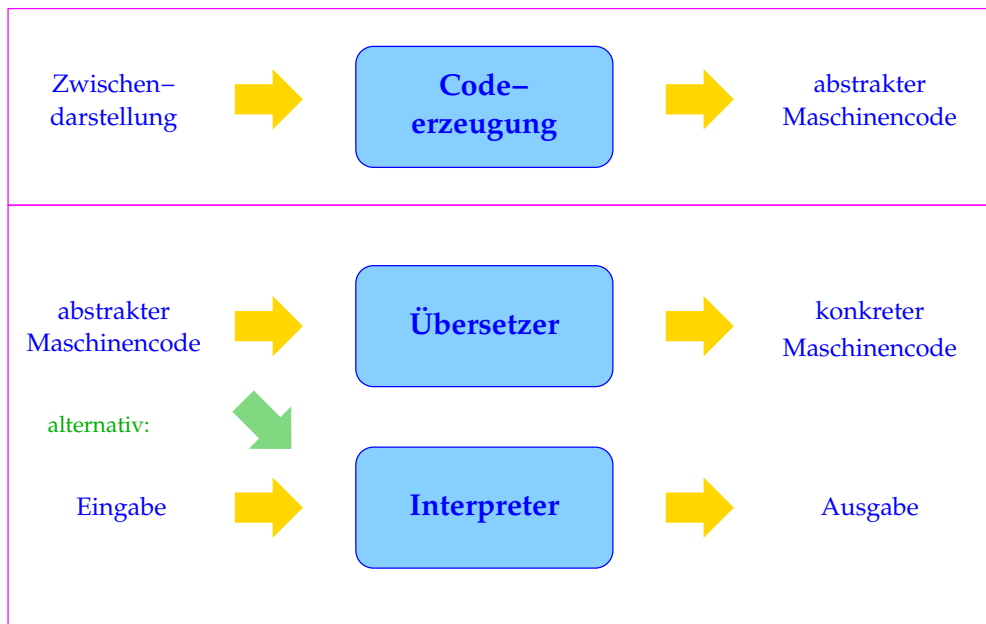
0.4 Aufgaben der Code-Erzeugung:

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

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

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

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



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

Vorteile:

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

Programmiersprachen, deren Übersetzungen auf abstrakten Maschinen beruhen:

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

Hier werden folgende Sprachen und abstrakte Maschinen betrachtet:

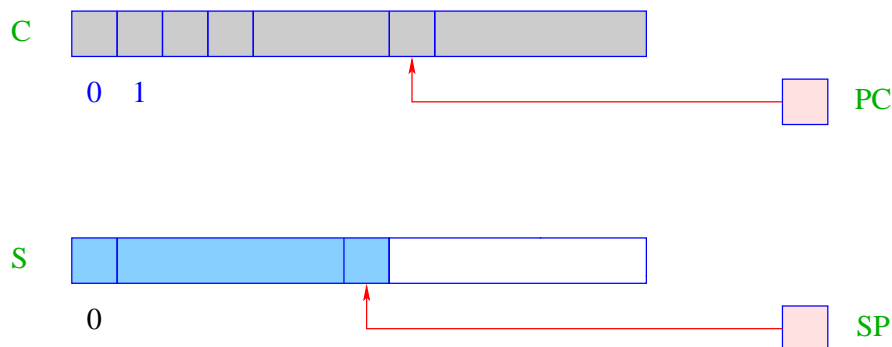
C	→	CMa	//	<i>imperativ</i>
PuF	→	MaMa	//	<i>funktional</i>
PuP	→	WiM	//	<i>logikbasiert</i>

Die Übersetzung von C

1 Die Architektur der CMa

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

Für die CMa benötigen wir:



- **S** ist der (Daten-)Speicher, auf dem nach dem LIFO-Prinzip neue Zellen allokiert werden können \implies **Keller/Stack**.
- **SP** ($\hat{=}$ **Stack Pointer**) ist ein Register, das die Adresse der obersten belegten Zelle enthält.
Vereinfachung: Alle Daten passen jeweils in eine Zelle von **S**.
- **C** ist der Code-Speicher, der das Programm enthält.
Jede Zelle des Felds **C** kann exakt einen abstrakten Befehl aufnehmen.
- **PC** ($\hat{=}$ **Program Counter**) ist ein Register, das die Adresse des **nächsten** auszuführenden Befehls enthält.
- Vor Programmausführung enthält der **PC** die Adresse 0
 \implies **C[0]** enthält den ersten auszuführenden Befehl.

Die Ausführung von Programmen:

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

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

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

2 Einfache Ausdrücke und Wertzuweisungen

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

Das heißt: erzeuge eine Instruktionsfolge, die

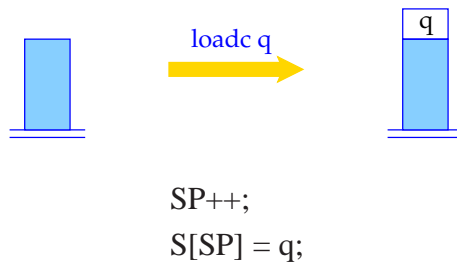
- den Wert des Ausdrucks ermittelt und dann
- oben auf dem Keller ablegt...

Idee:

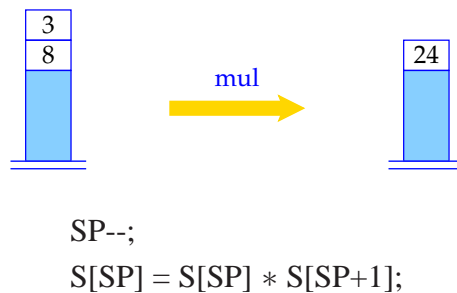
- berechne erst die Werte für die Teilausdrücke;
- merke diese Zwischenergebnisse oben auf dem Keller;
- wende dann den Operator an!

Generelles Prinzip:

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



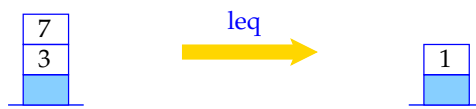
Die Instruktion `loadc q` benötigt keine Argumente, legt dafür aber als Wert die Konstante `q` oben auf dem Stack ab.



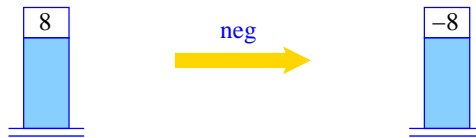
`mul` erwartet zwei Argumente oben auf dem Stack, konsumiert sie und legt sein Ergebnis oben auf dem Stack ab.

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

Beispiel: Der Operator `leq`



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



$$S[SP] = -S[SP];$$

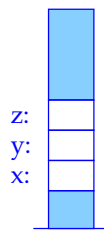
Beispiel: Code für $1 + 7$:

loadc 1 loadc 7 add

Ausführung dieses Codes:



Variablen ordnen wir Speicherzellen in **S** zu:



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

Variablen können auf zwei Weisen verwendet werden.

Beispiel: $x = y + 1$

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

L-Wert von x = Adresse von x
R-Wert von x = Inhalt von x

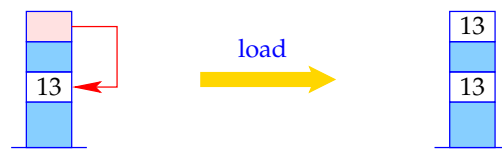
$\text{code}_R e \rho$	liefert den Code zur Berechnung des R-Werts von e in der Adress-Umgebung ρ
$\text{code}_L e \rho$	analog für den L-Wert

Achtung:

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

$$\begin{aligned} \text{code}_R (e_1 + e_2) \rho &= \text{code}_R e_1 \rho \\ &\quad \text{code}_R e_2 \rho \\ &\quad \text{add} \\ &\quad \dots \text{ analog für die anderen binären Operatoren} \\ \text{code}_R (-e) \rho &= \text{code}_R e \rho \\ &\quad \text{neg} \\ &\quad \dots \text{ analog für andere unäre Operatoren} \\ \text{code}_R q \rho &= \text{loadc } q \\ \text{code}_L x \rho &= \text{loadc } (\rho x) \\ &\quad \dots \\ \text{code}_R x \rho &= \text{code}_L x \rho \\ &\quad \text{load} \end{aligned}$$

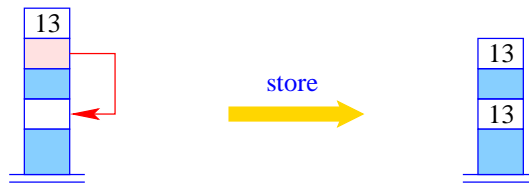
Die Instruktion `load` lädt den Wert der Speicherzelle, deren Adresse oben auf dem Stack liegt.



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

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

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



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

Beispiel: Code für $e \equiv x = y - 1$ mit $\rho = \{x \mapsto 4, y \mapsto 7\}$.
 Dann liefert `codeR` e ρ :



Optimierungen:

Einführung von Spezialbefehlen für häufige Befehlsfolgen, hier etwa:

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

3 Anweisungen und Anweisungsfolgen

Ist e ein Ausdruck, dann ist $e;$ eine Anweisung (Statement). Anweisungen liefern keinen Wert zurück. Folglich muss der **SP** vor und nach der Ausführung des erzeugten Codes gleich sein:

`code` $e; \rho$ = `codeR` $e \rho$
 `pop`

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



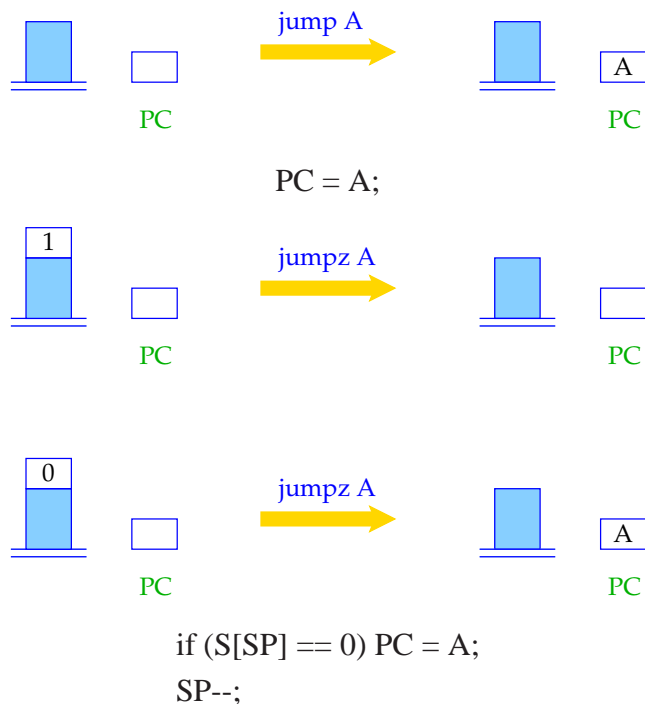
`SP--;`

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

$$\begin{aligned} \text{code } (s \text{ ss}) \rho &= \text{code } s \rho \\ &\quad \text{code } \text{ss } \rho \\ \text{code } \varepsilon \rho &= \quad \quad \quad // \text{ leere Folge von Befehlen} \end{aligned}$$

4 Bedingte und iterative Anweisungen

Um von linearer Ausführungsreihenfolge abzuweichen, benötigen wir Sprünge:



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

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

Vorteile:

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

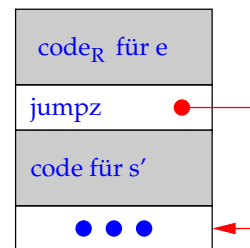
4.1 Bedingte Anweisung, einseitig

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

Idee:

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

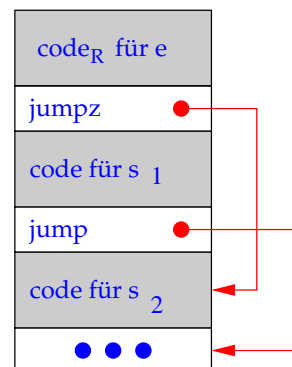
code $s \rho$ = code_R $e \rho$
 jumpz A
 code $s' \rho$
 A: ...



4.2 Zweiseitiges if

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

code $s \rho$ = code_R $e \rho$
 jumpz A
 code $s_1 \rho$
 jump B
 A: code $s_2 \rho$
 B: ...



Beispiel:

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

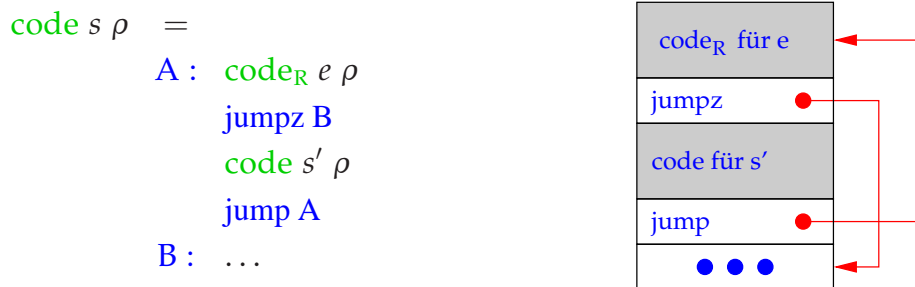
$s \equiv \mathbf{if}(x > y)$ (i)
 $x = x - y;$ (ii)
 $\mathbf{else} y = y - x;$ (iii)

Dann liefert $\text{code } s \rho$:

<pre>loada 4 loada 7 gr jumpz A</pre> <p style="text-align: center; color: red;">(i)</p>	<pre>loada 4 loada 7 sub storea 4 pop jump B</pre> <p style="text-align: center; color: red;">(ii)</p>	<pre>A: loada 7 loada 4 sub storea 7 pop B: ...</pre> <p style="text-align: center; color: red;">(iii)</p>
--	--	--

4.3 while-Schleifen

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



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

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

Dann liefert $\text{code } s \rho$ die Folge:

<pre>A: loada 7 loadc 0 gr jumpz B</pre>	<pre>loada 9 loadc 1 add storea 9 pop</pre>	<pre>loada 7 loada 8 sub storea 7 pop jump A</pre>	<pre>B: ...</pre>
--	---	--	-------------------

4.4 for-Schleifen

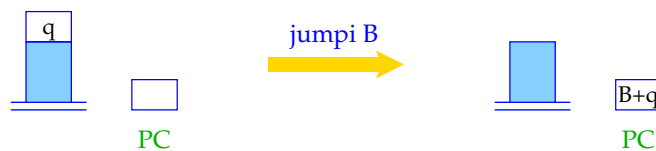
Die **for**-Schleife $s \equiv \mathbf{for} (e_1; e_2; e_3) s'$ ist äquivalent zu der Statementfolge $e_1; \mathbf{while} (e_2) \{s' e_3; \}$ – sofern s' keine **continue**-Anweisung enthält. Darum übersetzen wir:

```
code s ρ = codeR e1
           pop
           A : codeR e2 ρ
               jumpz B
               code s' ρ
               codeR e3 ρ
               pop
               jump A
           B : ...
```

4.5 Das switch-Statement

Idee:

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



```
PC = B + S[SP];
SP--;
```

Vereinfachung:

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

$$s \equiv \text{switch } (e) \{$$

$$\quad \text{case 0: } ss_0 \text{ break;}$$

$$\quad \text{case 1: } ss_1 \text{ break;}$$

$$\quad \vdots$$

$$\quad \text{case } k-1: ss_{k-1} \text{ break;}$$

$$\quad \text{default: } ss_k$$

$$\}$$

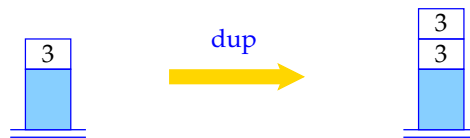
Dann ergibt sich für s die Instruktionsfolge:

<code>code</code> $s \rho$	=	<code>code</code> _R $e \rho$	C_0 :	<code>code</code> $ss_0 \rho$	B :	<code>jump</code> C_0
		<code>check</code> $0 k B$		<code>jump</code> D		...
				...		<code>jump</code> C_k
			C_k :	<code>code</code> $ss_k \rho$	D :	...
				<code>jump</code> D		

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

<code>check</code> $0 k B$	=	<code>dup</code>	<code>dup</code>	<code>jumpi</code> B
		<code>loadc</code> 0	<code>loadc</code> k	A : <code>pop</code>
		<code>geq</code>	<code>leq</code>	<code>loadc</code> k
		<code>jumpz</code> A	<code>jumpz</code> A	<code>jumpi</code> B

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



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

Achtung:

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

5 Speicherbelegung für Variablen

Ziel:

Ordne jeder Variablen x **statisch**, d. h. zur Übersetzungszeit, eine feste (Relativ-)Adresse ρx zu!

Annahmen:

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

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

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

5.1 Felder

Beispiel: `int [11] a;`

Das Feld `a` enthält 11 Elemente und benötigt darum 11 Zellen.
 ρa ist die Adresse des Elements `a[0]`.



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

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

Dann ergibt sich für die Deklaration $d \equiv t_1 x_1; \dots t_k x_k$:

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

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

Aufgabe:

Erweitere `codeL` und `codeR` auf Ausdrücke mit indizierten Feldzugriffen.

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

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

$$\begin{aligned} \text{code}_L a[e] \rho &= \text{loadc } (\rho a) \\ &\quad \text{code}_R e \rho \\ &\quad \text{loadc } |t| \\ &\quad \text{mul} \\ &\quad \text{add} \end{aligned}$$

... oder allgemeiner:

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

Bemerkung:

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

$$2[a] \quad a[2] \quad a + 2$$

5.2 Strukturen

In **Modula** heißen Strukturen **Records**.

Vereinfachung:

Komponenten-Namen werden nicht anderweitig verwandt.

Alternativ könnte man zu jedem Struktur-Typ st eine separate Komponenten-Umgebung ρ_{st} verwalten :-)

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

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

Sei allgemein $t \equiv \text{struct } \{t_1 c_1; \dots t_k c_k\}$. Dann ist

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

Damit erhalten wir:

$$\text{code}_L(e.c) \rho = \begin{array}{l} \text{code}_L e \rho \\ \text{loadc}(\rho c) \\ \text{add} \end{array}$$

Beispiel:

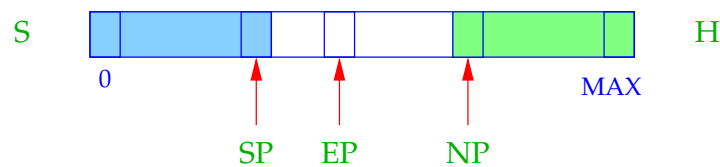
Sei `struct { int a; int b; } x;` mit $\rho = \{x \mapsto 13, a \mapsto 0, b \mapsto 1\}$.
Dann ist

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

6 Zeiger und dynamische Speicherverwaltung

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

⇒ Wir benötigen eine weitere potentiell beliebig große Datenstruktur **H**
– den **Heap** (bzw. die **Halde**):



NP $\hat{=}$ **New Pointer**; zeigt auf unterste belegte Halde-
zelle.

EP $\hat{=}$ **Extreme Pointer**; zeigt auf die Zelle, auf die der
SP maximal zeigen kann

(innerhalb der aktuellen Funktion). $\hat{=}$

Idee dabei:

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

- Die Überprüfungen bei Heap-Allokationen bleiben erhalten :-).

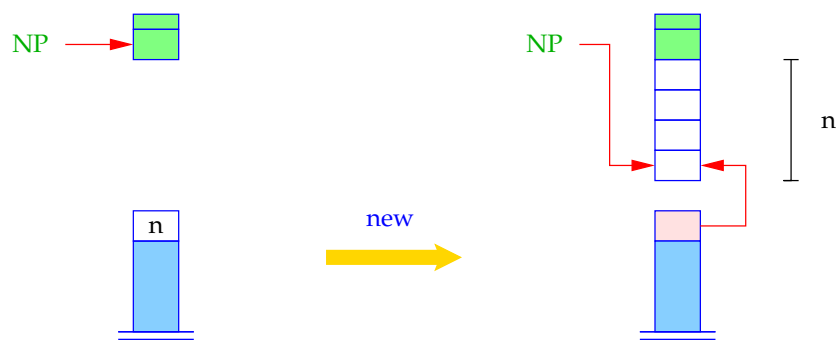
Mit Zeiger (-Werten) rechnen, heißt in der Lage zu sein,

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

Es gibt zwei Arten, Zeiger zu erzeugen:

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

$$\text{code}_R \text{ malloc } (e) \rho = \text{code}_R e \rho \text{ new}$$



```

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

```

- NULL ist eine spezielle Zeigerkonstante (etwa 0 :-)
 - Im Falle einer Kollision von Stack und Heap wird der NULL-Zeiger zurückgeliefert.
- (2) Die Anwendung des Adressoperators & liefert einen Zeiger auf eine Variable, d. h. deren Adresse ($\hat{=}$ L-Wert). Deshalb:

$$\text{code}_R (\&e) \rho = \text{code}_L e \rho$$

Dereferenzieren von Zeigern:

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

$$\text{code}_L (*e) \rho = \text{code}_R e \rho$$

Beispiel:

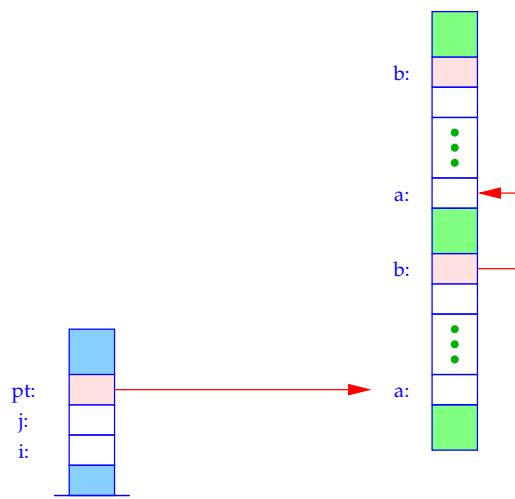
Betrachte für
`struct t { int a[7]; struct t *b; };`
`int i, j;`
`struct t *pt;`

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

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

$$\text{code}_L (e \rightarrow a) \rho = \text{code}_R e \rho$$

`loadc (ρa)`
`add`



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

$$\begin{aligned} \text{code}_L e \rho &= \text{code}_R ((pt \rightarrow b) \rightarrow a) \rho &= \text{code}_R ((pt \rightarrow b) \rightarrow a) \rho \\ &\text{code}_R (i + 1) \rho &\text{loada 1} \\ &\text{loadc 1} &\text{loadc 1} \\ &\text{mul} &\text{add} \\ &\text{add} &\text{loadc 1} \\ & &\text{mul} \\ & &\text{add} \end{aligned}$$

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

$$\text{code}_R((pt \rightarrow b) \rightarrow a) \rho = \text{code}_R(pt \rightarrow b) \rho = \begin{array}{l} \text{loada } 3 \\ \text{loadc } 0 \\ \text{add} \\ \text{load} \\ \text{loadc } 0 \\ \text{add} \end{array}$$

Damit ergibt sich insgesamt die Folge:

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

7 Zusammenfassung

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

$$\text{code}_L(e_1[e_2]) \rho = \begin{array}{l} \text{code}_R e_1 \rho \\ \text{code}_R e_2 \rho \\ \text{loadc } |t| \\ \text{mul} \\ \text{add} \end{array} \quad \text{sofern } e_1 \text{ Typ } t [] \text{ hat}$$

$$\text{code}_L(e.a) \rho = \begin{array}{l} \text{code}_L e \rho \\ \text{loadc } (\rho a) \\ \text{add} \end{array}$$

$$\text{code}_L (*e) \rho = \text{code}_R e \rho$$

$$\text{code}_L x \rho = \text{loadc} (\rho x)$$

$$\text{code}_R (\&e) \rho = \text{code}_L e \rho$$

$$\text{code}_R (\text{malloc} (e)) \rho = \text{code}_R e \rho \\ \text{new}$$

$$\text{code}_R e \rho = \text{code}_L e \rho \quad \text{falls } e \text{ ein Feld ist}$$

$$\text{code}_R (e_1 \square e_2) \rho = \text{code}_R e_1 \rho \\ \text{code}_R e_2 \rho \\ \text{op} \quad \text{op Befehl zu Operator '}\square\text{'}$$

$$\text{code}_R q \rho = \text{loadc} q \quad q \text{ Konstante}$$

$$\text{code}_R (e_1 = e_2) \rho = \text{code}_L e_1 \rho \\ \text{code}_R e_2 \rho \\ \text{store}$$

$$\text{code}_R e \rho = \text{code}_L e \rho \\ \text{load} \quad \text{sonst}$$

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

Betrachte das Statement: `s1 ≡ *a = 5;`

Dann ist:

$$\text{code}_L (*a) \rho = \text{code}_R a \rho = \text{code}_L a \rho = \text{loadc} 7 \\ \text{code} s_1 \rho = \text{loadc} 7 \\ \text{loadc} 5 \\ \text{store} \\ \text{pop}$$

Zur Übung übersetzen wir auch noch:

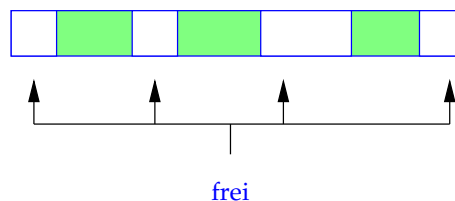
$$s_2 \equiv b = (\text{int} *) \&a + 2; \quad \text{und} \quad s_3 \equiv *(b + 3) = 5;$$

<code>code</code>	$(s_2 s_3)$	ρ	=	loadc 17		loadc 17
				loadc 7		load
				loadc 2		loadc 3
				loadc 1	// Skalierung	loadc 1 // Skalierung
				mul		mul
				add		add
				store		loadc 5
				pop	// Ende von s_2	store
						pop // Ende von s_3

8 Freigabe von Speicherplatz

Probleme:

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



Mögliche Auswege:

- Nimm an, der Programmierer weiß, was er tut. Verwalte dann die freien Abschnitte (etwa sortiert nach Größe) in einer speziellen Datenstruktur;
 \implies **malloc** wird teuer :-)
- Tue nichts, d.h.:

`code free` (e); ρ = `codeR` e ρ
`pop`

\implies einfach und (i.a.) effizient :-)

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

9 Funktionen

Die Definition einer Funktion besteht aus

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

In **C** gilt:

$\text{code}_R f \rho = \text{load } c_f = \text{Anfangsadresse des Codes für } f$

⇒ Auch Funktions-Namen müssen in der Adress-Umgebung verwaltet werden!

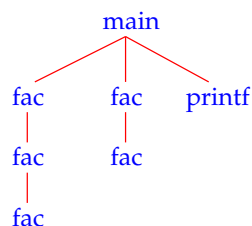
Beispiel:

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

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

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

Der Rekursionsbaum im Beispiel:



Wir schließen:

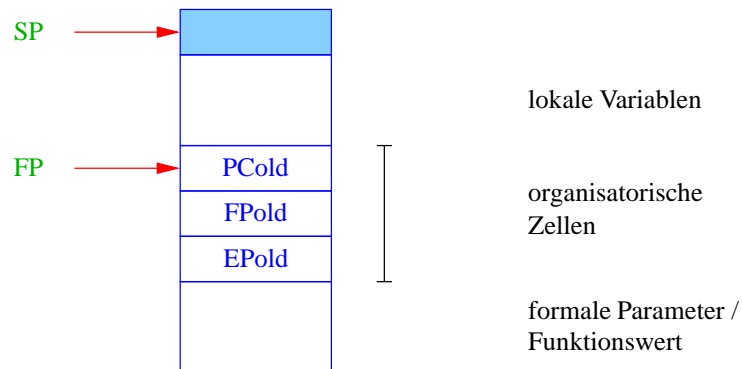
Die **formalen Parameter** und **lokalen Variablen** der verschiedenen Aufrufe der selben Funktion (**Instanzen**) müssen auseinander gehalten werden.

Idee:

Lege einen speziellen Speicherbereich für jeden Aufruf einer Funktion an.

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

9.1 Speicherorganisation für Funktionen



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

Achtung:

- Die lokalen Variablen erhalten Relativadressen $+1, +2, \dots$
- Die formalen Parameter liegen **unterhalb** der organisatorischen Zellen und haben deshalb **negative** Adressen relativ zu **FP** :-)
- Diese Organisation ist besonders geeignet für Funktionsaufrufe mit **variabler** Argument-Anzahl wie z.B. `printf`.
- Den Speicherbereich für die Parameter recyceln wir zur Speicherung des Rückgabewerts der Funktion :-))

Vereinfachung: Der Rückgabewert passt in eine einzige Zelle.

Die organisatorischen Zellen speichern die Register-Inhalte, die nach dem Funktions-Aufruf **restauriert** werden müssen. Bei einem Funktions-Aufruf muss der **FP** in eine **organisatorische Zelle** gerettet werden.

Unsere Übersetzungsaufgaben für Funktionen:

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

9.2 Bestimmung der Adress-Umgebung

Wir müssen zwei Arten von Variablen unterscheiden:

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

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

Achtung:

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

Beispiel:

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

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

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

Vorkommende Adress-Umgebungen in dem Programm:

0 Außerhalb der Funktions-Definitionen:

```
 $\rho_0 :$ 
     $i \mapsto (G, 1)$ 
     $l \mapsto (G, 2)$ 
     $ith \mapsto (G, \_ith)$ 
     $main \mapsto (G, \_main)$ 
    ...
```

1 Innerhalb von `ith`:

$$\begin{aligned} \rho_1 : \quad & i \mapsto (L, -4) \\ & x \mapsto (L, -3) \\ & l \mapsto (G, 2) \\ & \text{ith} \mapsto (G, \text{ith}) \\ & \text{main} \mapsto (G, \text{main}) \\ & \dots \end{aligned}$$

Achtung:

- Die formalen Parameter werden von **rechts** nach **links** ausgewertet !!
- Der erste Parameter liegt direkt unterhalb der organisatorischen Zellen :-)
- Für einen Prototypen $\tau f(\tau_1 x_1, \dots, \tau_k x_k)$ setzen wir:

$$x_1 \mapsto (L, -2 - |\tau_1|) \quad x_i \mapsto (L, -2 - |\tau_1| - \dots - |\tau_i|)$$

2 Innerhalb von `main`:

$$\begin{aligned} \rho_2 : \quad & i \mapsto (G, 1) \\ & l \mapsto (G, 2) \\ & k \mapsto (L, 1) \\ & \text{ith} \mapsto (G, \text{ith}) \\ & \text{main} \mapsto (G, \text{main}) \\ & \dots \end{aligned}$$

9.3 Betreten und Verlassen von Funktionen

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

Der Code für den Aufruf muss auf den Caller und den Callee verteilt werden.

Die Aufteilung kann nur so erfolgen, dass der Teil, der von Informationen des Callers abhängt, auch dort erzeugt wird (analog für den Callee).

Achtung:

Den Platz für die aktuellen Parameter kennt nur der Caller ...

Aktionen beim Betreten von g :

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

Aktionen bei Beenden des Aufrufs:

- | | | |
|--|---|--------|
| 1. Abspeichern des Rückgabewerts | } | return |
| 2. Rücksetzen der Register FP, EP, SP | | |
| 3. Rücksprung in den Code von f , d. h.
Restauration des PC | | |
| 4. Aufräumen des Stack | } | slide |

Damit erhalten wir für einen Aufruf für eine Funktion mit mindestens einem Parameter und einem Rückgabewert:

$$\begin{aligned}
 \text{code}_R g(e_1, \dots, e_n) \rho &= \text{code}_R e_n \rho \\
 &\dots \\
 &\text{code}_R e_1 \rho \\
 &\text{mark} \\
 &\text{code}_R g \rho \\
 &\text{call} \\
 &\text{slide } (m - 1)
 \end{aligned}$$

wobei m der Platz für die aktuellen Parameter ist .

Beachte:

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

- Ähnlich deklarierten Feldern, werden Funktions-Namen als **konstante Zeiger** auf Funktionen aufgefasst. Dabei ist der R-Wert dieses Zeigers gleich der Anfangs-Adresse der Funktion.

- **Achtung!** Für eine Variable `int (*g)()`; sind die beiden Aufrufe

`(*g)()` und `g()`

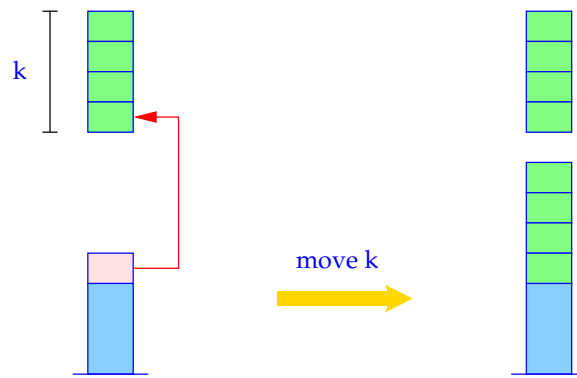
äquivalent! Per **Normalisierung**, muss man sich hier vorstellen, werden Dereferenzierungen eines Funktions-Zeigers ignoriert :-)

- Bei der Parameter-Übergabe von Strukturen werden diese kopiert.

Folglich:

<code>code_R f ρ</code>	<code>= loadc (ρ f)</code>	<code>f</code> ein Funktions-Name
<code>code_R (*e) ρ</code>	<code>= code_R e ρ</code>	<code>e</code> ein Funktions-Zeiger
<code>code_R e ρ</code>	<code>= code_L e ρ</code>	
	<code>move k</code>	<code>e</code> eine Struktur der Größe <code>k</code>

Dabei ist:



```

for (i = k-1; i ≥ 0; i--)
    S[SP+i] = S[S[SP]+i];
SP = SP+k-1;

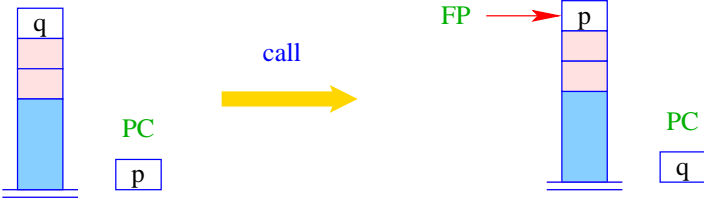
```

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



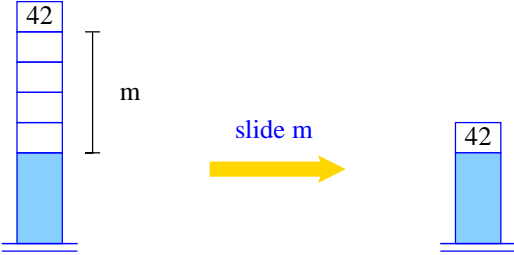
```
S[SP+1] = EP;
S[SP+2] = FP;
SP = SP + 2;
```

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



```
tmp = S[SP];
S[SP] = PC;
FP = SP;
PC = tmp;
```

Der Befehl **slide** kopiert den Rückgabewert an die korrekte Stelle:



```

tmp = S[SP];
SP = SP-m;
S[SP] = tmp;

```

Entsprechend übersetzen wir eine Funktions-Definition:

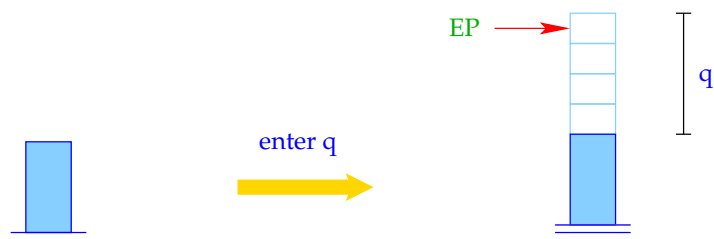
```

code t f (specs){V_defs ss} ρ =
    _f: enter q      // setzen des EP
        alloc k     // Anlegen der lokalen Variablen
    code ss ρ_f
        return      // Verlassen der Funktion

```

wobei $q = max + k$ wobei
 max = maximale Länge des lokalen Kellers
 k = Platz für die lokalen Variablen
 ρ_f = Adress-Umgebung für f
// berücksichtigt $specs, V_defs$ und ρ

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

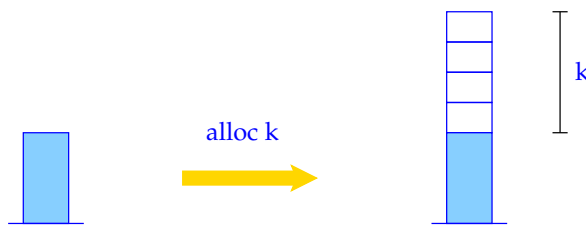


```

EP = SP + q;
if (EP ≥ NP)
    Error ("Stack Overflow");

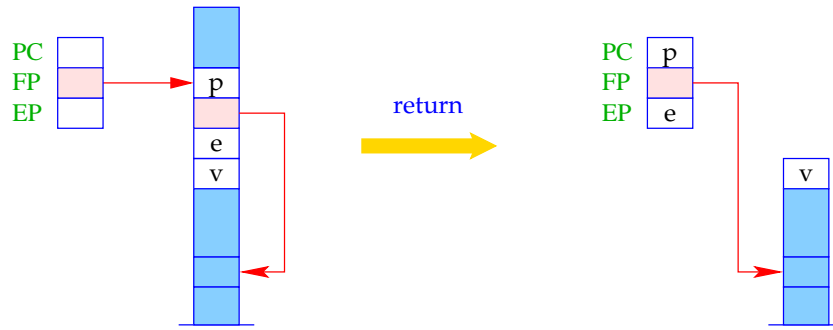
```

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



$$SP = SP + k;$$

Der Befehl `return` gibt den aktuellen Keller-Rahmen auf.
D.h. er restauriert die Register `PC`, `FP` und `EP`:



$PC = S[FP]; EP = S[FP-2];$
 if ($EP \geq NP$) Error ("Stack Overflow");
 $SP = FP-3; FP = S[SP+2];$

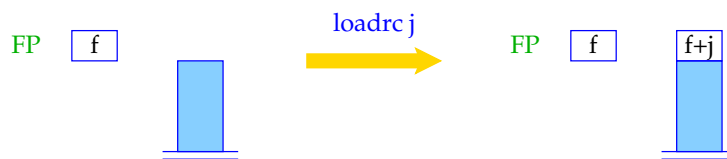
9.4 Zugriff auf Variablen, formale Parameter und Rückgabe von Werten

Zugriffe auf lokale Variablen oder formale Parameter erfolgen relativ zum aktuellen `FP`.
Daher modifizieren wir `codeL` für Variablen-Namen.

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

$$code_L x \rho = \begin{cases} loadc j & tag = G \\ loadrc j & tag = L \end{cases}$$

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



$SP++;$
 $S[SP] = FP+j;$

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

loadr j = loadrc j
load

bla; storer j = loadrc j; bla
store

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

code **return e**; ρ = code_R e ρ
storer -3
return

Beispiel: Für die Funktion

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

erzeugen wir:

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

Dabei ist $\rho_{\text{fac}} : x \mapsto (L, -3)$ und $q = 1 + 4 = 5$.

10 Übersetzung ganzer Programme

Vor der Programmausführung gilt:

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

Sei $p \equiv V_defs \ F_def_1 \dots F_def_n$, ein Programm,
wobei F_def_i eine Funktion f_i definiert, von denen eine `main` heißt.

Der Code für das Programm p enthält:

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

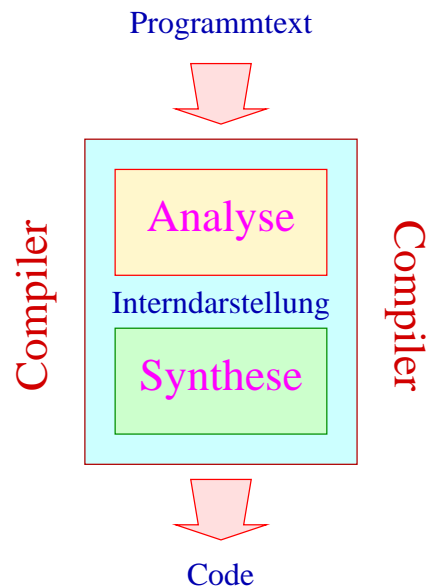
Dann definieren wir:

```
code p  $\emptyset$  =      enter (k + 6)
                   alloc (k + 1)
                   loadc 0
                   mark
                   loadc _main
                   call
                   slide (k + 1)
                   halt
                   _f1: code F_def1  $\rho$ 
                   :
                   _fn: code F_defn  $\rho$ 
```

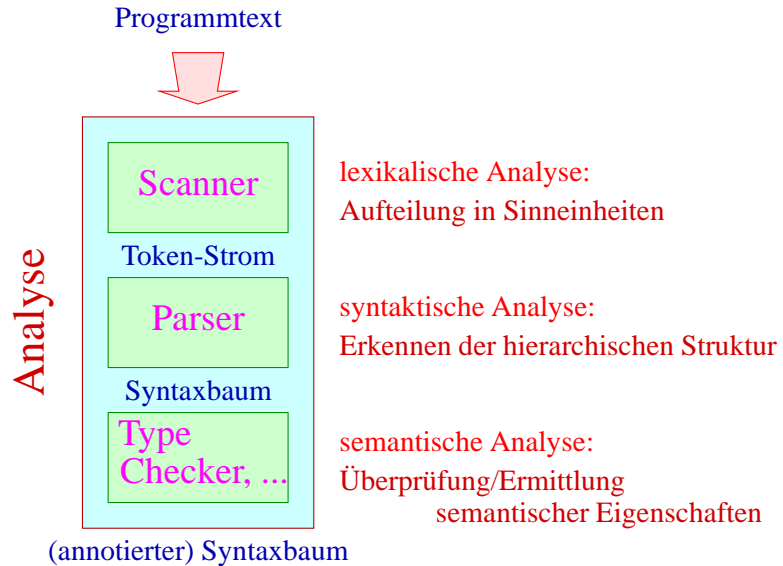
wobei $\emptyset \hat{=}$ leere Adress-Umgebung;
 $\rho \hat{=}$ globale Adress-Umgebung;
 $k \hat{=}$ Platz für globale Variablen

Die Analyse-Phase

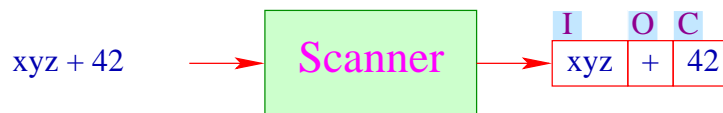
Orientierung:



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



1 Die lexikalische Analyse



- Ein **Token** ist eine Folge von Zeichen, die zusammen eine Einheit bilden.
- Tokens werden in **Klassen** zusammen gefasst. Zum Beispiel:
 - **Namen (Identifier)** wie `xyz`, `pi`, ...
 - **Konstanten** wie `42`, `3.14`, `"abc"`, ...
 - **Operatoren** wie `+`, ...
 - **reservierte Worte** wie `if`, `int`, ...

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

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

⇒ **Sieber**

Diskussion:

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



Vorteile:

Produktivität:

Die Komponente lässt sich **schneller** herstellen :-)

Korrektheit:

Die Komponente realisiert (beweisbar :-) die Spezifikation.

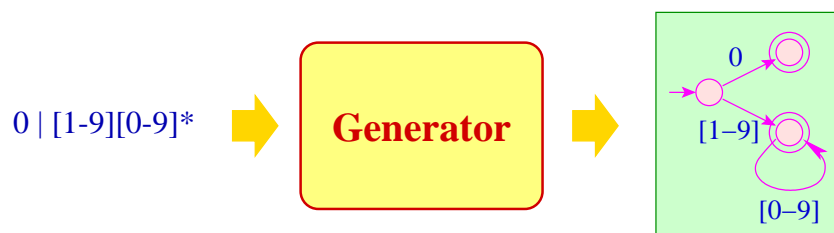
Effizienz:

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

Einschränkungen:

- Spezifizieren ist auch **Programmieren** — nur eventuell einfacher :-)
- Generierung statt Implementierung lohnt sich nur für **Routine-Aufgaben** ... und ist nur für Probleme möglich, die **sehr gut verstanden** sind :-)

... in unserem Fall:



Spezifikation von Token-Klassen: Reguläre Ausdrücke;

Generierte Implementierung: Endliche Automaten + X :-)

1.1 Grundlagen: Reguläre Ausdrücke

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

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

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



Stephen Kleene, Madison Wisconsin, 1909-1994

Beispiele:

$$\begin{aligned} &((a \cdot b^*) \cdot a) \\ &(a \mid b) \\ &((a \cdot b) \cdot (a \cdot b)) \end{aligned}$$

Achtung:

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

$$* > \cdot > |$$

und lassen “.” weg :-)

- Reale Spezifikations-Sprachen bieten zusätzliche Konstrukte wie:

$$\begin{aligned} e? &\equiv (\epsilon \mid e) \\ e^+ &\equiv (e \cdot e^*) \end{aligned}$$

und verzichten auf “ ϵ ” :-)

Spezifikationen benötigen eine **Semantik** :-)

Im Beispiel:

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

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

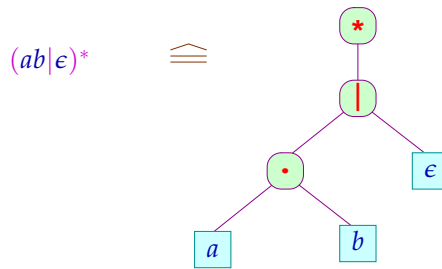
$$\begin{aligned} \llbracket \epsilon \rrbracket &= \{\epsilon\} \\ \llbracket a \rrbracket &= \{a\} \\ \llbracket e^* \rrbracket &= (\llbracket e \rrbracket)^* \\ \llbracket e_1 \mid e_2 \rrbracket &= \llbracket e_1 \rrbracket \cup \llbracket e_2 \rrbracket \\ \llbracket e_1 \cdot e_2 \rrbracket &= \llbracket e_1 \rrbracket \cdot \llbracket e_2 \rrbracket \end{aligned}$$

Beachte:

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

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

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



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

Finger-Übung:

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

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

Konstruktion:

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

$$\begin{aligned}
 \mathcal{T}[\epsilon] &= \epsilon \\
 \mathcal{T}[a] &= a \\
 \mathcal{T}[e_1|e_2] &= \text{case } (\mathcal{T}[e_1], \mathcal{T}[e_2]) \text{ of } (\epsilon, \epsilon) : \epsilon \\
 &\quad | (e'_1, \epsilon) : e'_1? \\
 &\quad | (\epsilon, e'_2) : e'_2? \\
 &\quad | (e'_1, e'_2) : (e'_1 | e'_2) \\
 \mathcal{T}[e_1 \cdot e_2] &= \text{case } (\mathcal{T}[e_1], \mathcal{T}[e_2]) \text{ of } (\epsilon, \epsilon) : \epsilon \\
 &\quad | (e'_1, \epsilon) : e'_1 \\
 &\quad | (\epsilon, e'_2) : e'_2 \\
 &\quad | (e'_1, e'_2) : (e'_1 \cdot e'_2) \\
 \mathcal{T}[e^*] &= \text{case } \mathcal{T}[e] \text{ of } \epsilon : \epsilon \\
 &\quad | e_1 : e_1^* \\
 \mathcal{T}[e?] &= \text{case } \mathcal{T}[e] \text{ of } \epsilon : \epsilon \\
 &\quad | e_1 : e_1?
 \end{aligned}$$

Unsere Anwendung:

Identifizier in Java:

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

Gleitkommazahlen:

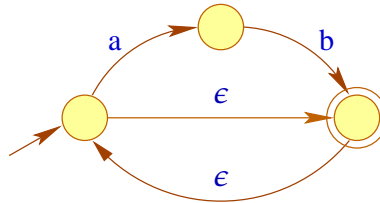
```
Float = {di}* (\.{di}|{di}\.) {di}* ((e|E)(\+|\-)?{di}+)?
```

Bemerkungen:

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

1.2 Grundlagen: Endliche Automaten

Beispiel:



Knoten: Zustände;

Kanten: Übergänge;

Beschriftungen: konsumierter input :-)



Michael O. Rabin, Stanford University



Dana S. Scott, Carnegie Mellon University, Pittsburgh

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

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

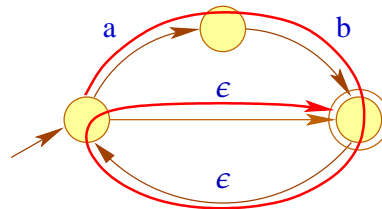
Für ϵ -NFAs ist:

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

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

Akzeptierung

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



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

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

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

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

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

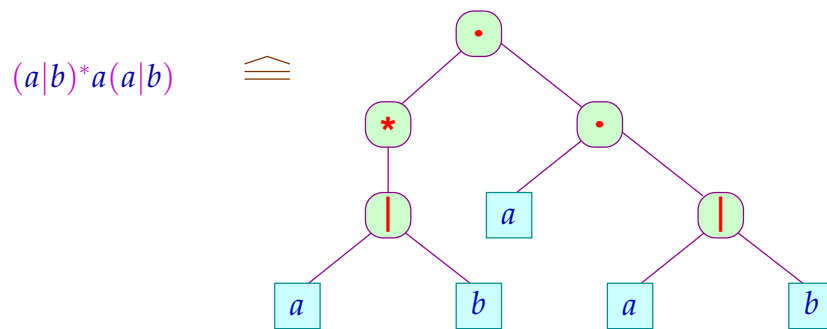
Satz:

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

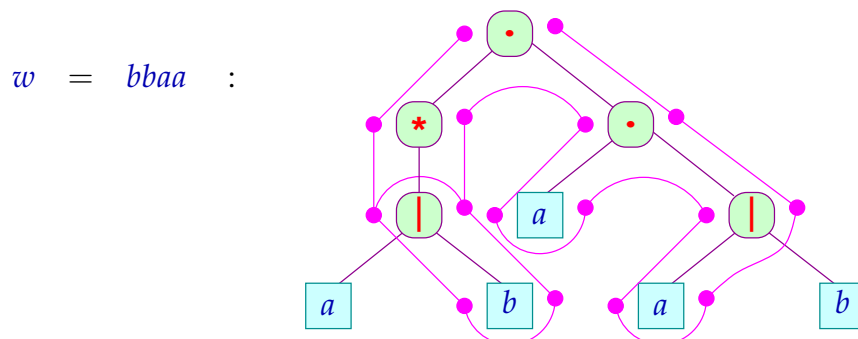
Idee:

Der Automat verfolgt (konzeptionell mithilfe einer Marke “•”), wohin man in e mit der Eingabe w gelangen kann.

Beispiel:



Beispiel:

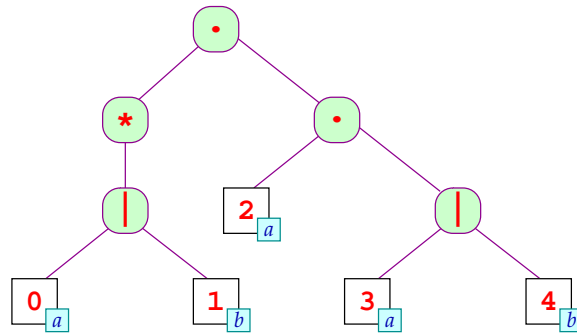


Beachte:

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

\implies Wir numerieren die Blätter durch ...

... im Beispiel:



Die Konstruktion:

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

Anfangszustand: $\bullet e$;

Endzustand: $e \bullet$;

Übergangsrelation:

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

Die übrigen Übergänge sind:

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

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

Diskussion:

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

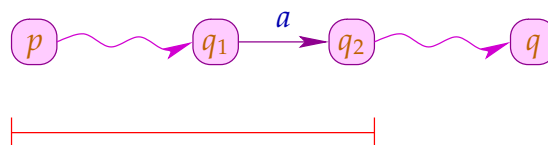


Strategie:

- (1) Beseitigung der ϵ -Übergänge;
- (2) Beseitigung des Nichtdeterminismus :-)

Beseitigung von ϵ -Übergängen:

Zwei einfache Ansätze:

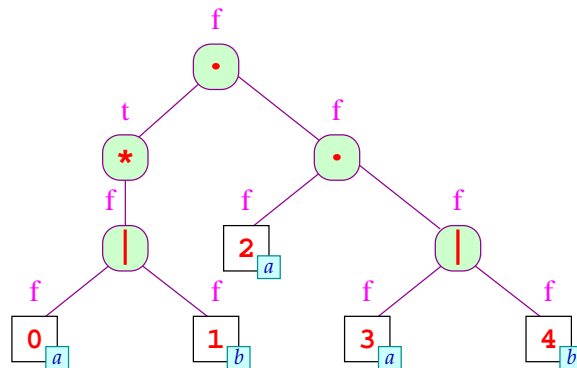


Wir benutzen hier den zweiten Ansatz.

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

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

... im Beispiel:



Implementierung:

DFS post-order Traversierung

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

Andernfalls:

$$\text{empty}[r_1 \mid r_2] = \text{empty}[r_1] \vee \text{empty}[r_2]$$

$$\text{empty}[r_1 \cdot r_2] = \text{empty}[r_1] \wedge \text{empty}[r_2]$$

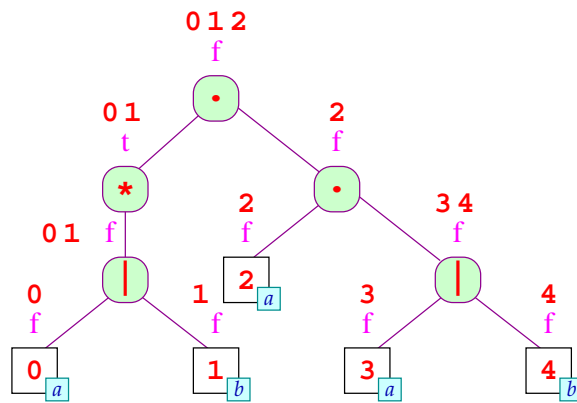
$$\text{empty}[r_1^*] = t$$

$$\text{empty}[r_1?] = t$$

2. Schritt:

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

... im Beispiel:



Implementierung:

DFS post-order Traversierung

Für Blätter $r \equiv \boxed{i \mid x}$ ist $\text{first}[r] = \{i \mid x \neq \epsilon\}$.

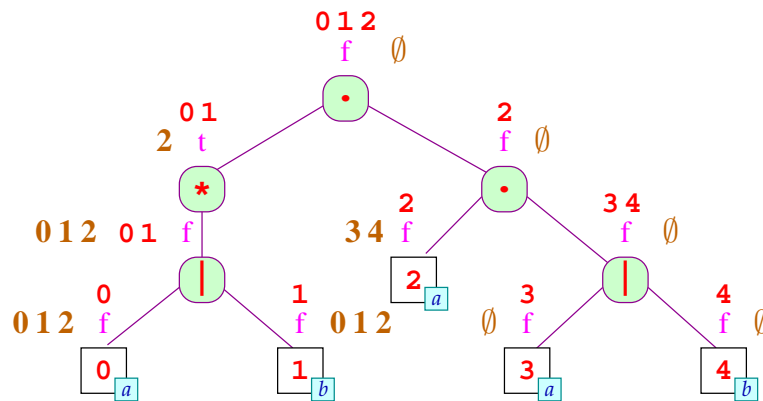
Andernfalls:

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

3. Schritt:

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

... im Beispiel:



Implementierung:

DFS pre-order Traversierung ;-)

Für die Wurzel haben wir:

$$next[e] = \emptyset$$

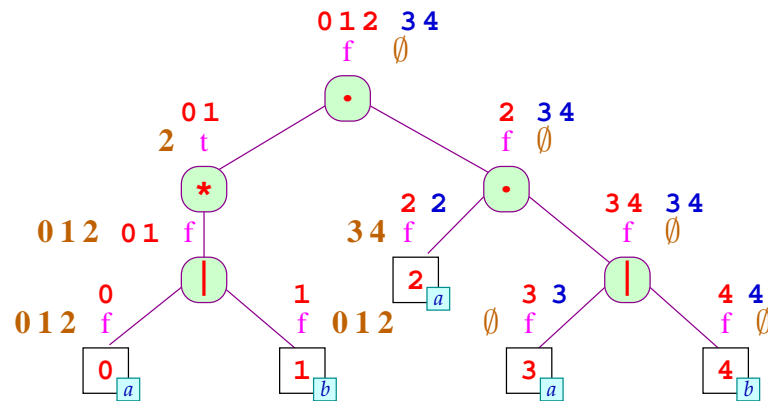
Ansonsten machen wir eine Fallunterscheidung über den **Kontext**:

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

4. Schritt:

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

... im Beispiel:



Implementierung:

DFS **post-order** Traversierung :-)

Für Blätter $r \equiv \boxed{i \mid x}$ ist $\text{last}[r] = \{i \mid x \neq \epsilon\}$.

Andernfalls:

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

Integration:

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

Startzustand: $\bullet e$

Endzustände:

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

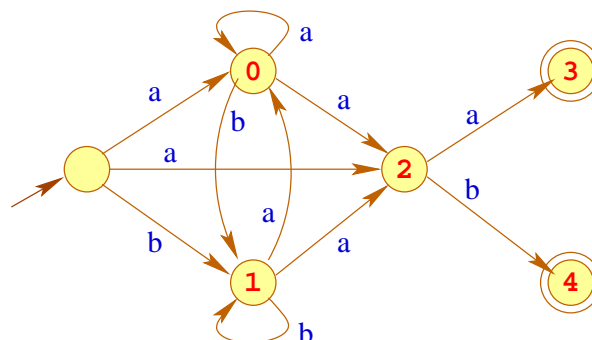
Übergänge:

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

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

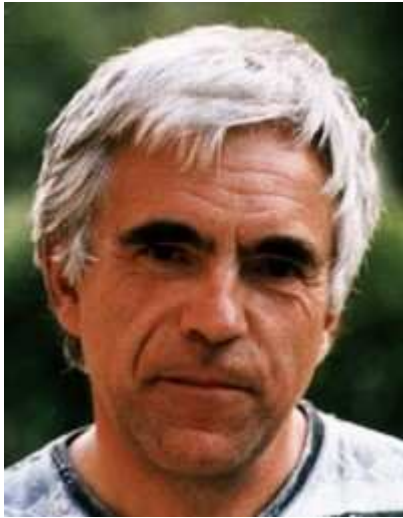
Den resultierenden Automaten bezeichnen wir mit A_e .

... im Beispiel:



Bemerkung:

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

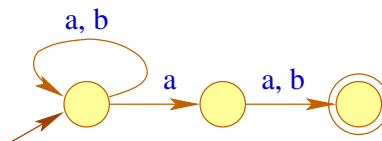


Gerard Berry, Esterel Technologies



Ravi Sethi, Research VR, Lucent Technologies

Der erwartete Automat:

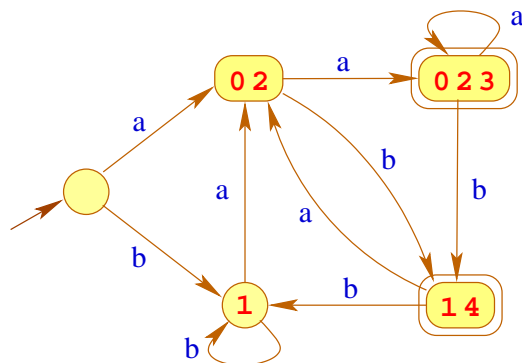
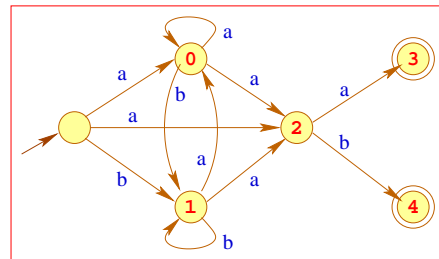


Bemerkung:

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

⇒ Teilmengen-Konstruktion

... im Beispiel:



Satz:

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

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

Konstruktion:

Zustände: Teilmengen von Q ;

Anfangszustände: $\{I\}$;

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

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

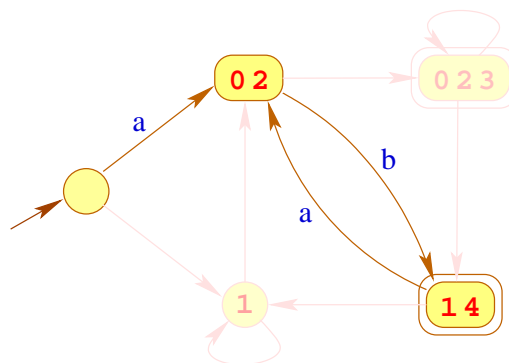
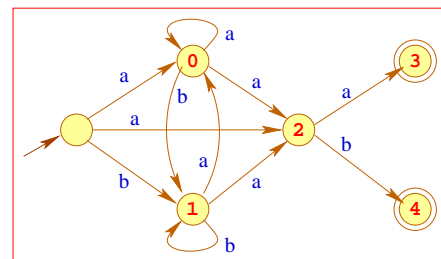
Achtung:

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

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

... im Beispiel:

a b a b



Bemerkungen:

- Bei einem Eingabewort der Länge n werden maximal $\mathcal{O}(n)$ Mengen konstruiert :-)
- Ist eine Menge bzw. eine Kante des **DFA** einmal konstruiert, heben wir sie in einer **Hash-Tabelle** auf.

- Bevor wir einen neuen Übergang konstruieren, sehen wir erst nach, ob wir diesen nicht schon haben :-)

Zusammenfassend finden wir:

Satz

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

$$\mathcal{L}(A) = \llbracket e \rrbracket$$

1.3 Design eines Scanners

Eingabe (vereinfacht): eine Menge von Regeln:

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

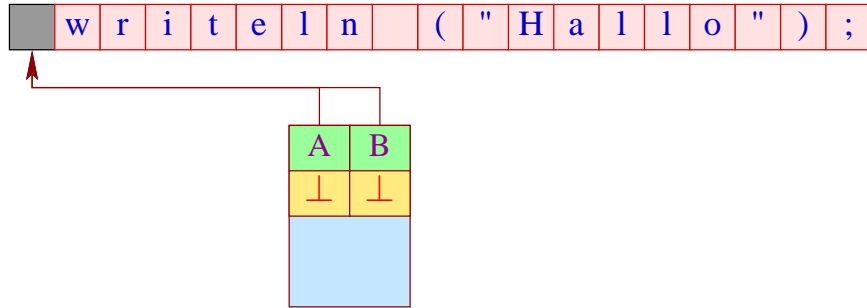
Ausgabe: ein Programm, das

- ... von der Eingabe ein **maximales Präfix** w liest, das $e_1 \mid \dots \mid e_k$ erfüllt;
- ... das **minimale** i ermittelt, so dass $w \in \llbracket e_i \rrbracket$;
- ... für w action_i ausführt.

Implementierung:

Idee:

- Konstruiere den DFA $\mathcal{P}(A_e) = (Q, \Sigma, \delta, \{q_0\}, F)$ zu dem Ausdruck $e = (e_1 \mid \dots \mid e_k)$;
- Definiere die Mengen:
$$\begin{array}{ll} F_1 & = \{q \in F \mid q \cap \text{last}[e_1] \neq \emptyset\} \\ F_2 & = \{q \in (F \setminus F_1) \mid q \cap \text{last}[e_2] \neq \emptyset\} \\ & \dots \\ F_k & = \{q \in (F \setminus (F_1 \cup \dots \cup F_{k-1})) \mid q \cap \text{last}[e_k] \neq \emptyset\} \end{array}$$
- Für Eingabe w gilt: $\delta^*(q_0, w) \in F_i$ genau dann wenn der Scanner für w action_i ausführen soll :-)
- Der Scanner verwaltet zwei Zeiger $\langle A, B \rangle$ und die zugehörigen Zustände $\langle q_A, q_B \rangle \dots$
- Der Zeiger A merkt sich die letzte Position in der Eingabe, nach der ein Zustand $q_A \in F$ erreicht wurde;
- Der Zeiger B verfolgt die aktuelle Position.

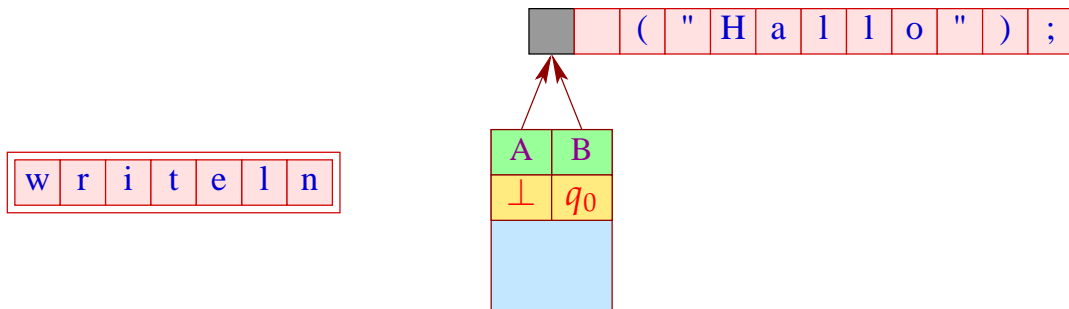


Idee (Fortsetzung):

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

$B := A; \quad A := \perp;$

$q_B := q_0; \quad q_A := \perp$



Erweiterung: Zustände

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

Beispiel: Kommentare

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

Eingabe (verallgemeinert): eine Menge von Regeln:

```
<state> { e1 { action1 yybegin(state1); }  
          e2 { action2 yybegin(state2); }  
          ...  
          ek { actionk yybegin(statek); }  
        }
```

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

... im Beispiel:

```
<YYINITIAL>  "/*" { yybegin(COMMENT); }  
<COMMENT>   { "*/" { yybegin(YYINITIAL); }  
              . | \n { }  
            }
```

Bemerkungen:

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

1.4 Implementierung von DFAs

Aufgaben:

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

Probleme:

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

Reduzierung der Anzahl der Zustände

Idee: Minimierung

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

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

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

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

Problem: Wie berechnet man \equiv_r ?

Idee:

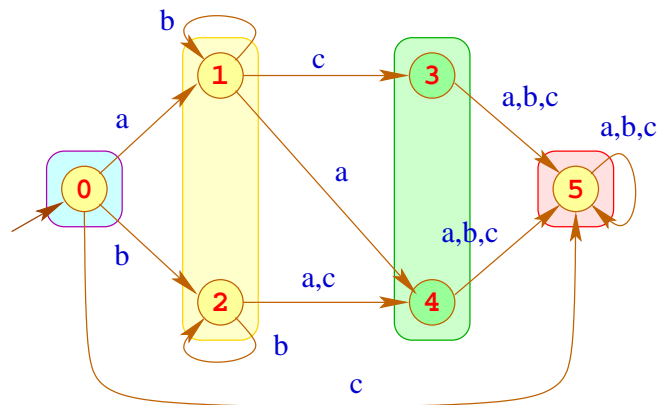
- Wir nehmen an, **maximal viel** sei äquivalent :-)

Wir starten mit der Partition:

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

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

Beispiel:



Bemerkungen:

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

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

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

- Der Ergebnis-Automat ist der **eindeutig bestimmte minimale Automat** für $\mathcal{L}(A)$;-)
- Eine naive Implementierung erfordert Laufzeit $\mathcal{O}(n^2)$.
Eine raffinierte Verwaltung der Partition liefert ein Verfahren mit Laufzeit $\mathcal{O}(n \cdot \log(n))$.



Anil Nerode , Cornell University, Iitaca



John E. Hopcroft, Cornell University, Iitaca

Reduzierung der Tabellengröße

Problem:

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

1. Idee:

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

Beispiel:

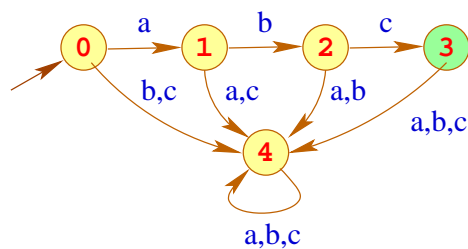
```
le = [a-zA-Z_\$]
ledi = [a-zA-Z_\$0-9]
Id = {le} {ledi}*
```

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

2. Idee:

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

Beispiel:



... die zugehörige Tabelle (transponiert):

	0	1	2	3	4
<i>a</i>	1	4	4	4	4
<i>b</i>	4	2	4	4	4
<i>c</i>	4	4	3	4	4

Beobachtung:

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

... im Beispiel:

	0	1	2
<i>A</i>	1	2	3
<i>valid</i>	<i>a</i>	<i>b</i>	<i>c</i>

- Feld **valid** teilt mit, für welches Element aus Σ der Eintrag gilt :-)
- **Achtung:** I.a. werden die Spalten nicht so perfekt übereinander passen!
Dann verschieben wir sie so lange, bis die jeweils nächste in die bisherigen Löcher hineinpasst.
- Darum müssen wir ein zusätzliches Feld **displacement** verwalten, in dem wir uns die Verschiebung merken ;-)

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

```

 $\delta(j, a) =$  let d = displacement[a]
              in if (valid[d + j]  $\equiv$  a)
                  then A[d + j]
                  else Default
              end
    
```

Diskussion:

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

2 Die syntaktische Analyse



- Die syntaktische Analyse versucht, Tokens zu größeren Programmeinheiten zusammen zu fassen.
- Solche Einheiten können sein:
 - Ausdrücke;
 - Statements;
 - bedingte Verzweigungen;
 - Schleifen; ...

Diskussion:

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



Spezifikation der hierarchischen Struktur: kontextfreie Grammatiken;

Generierte Implementierung: Kellerautomaten + X :-)

2.1 Grundlagen: Kontextfreie Grammatiken

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

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

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



Noam Chomsky, MIT (Guru)



John Backus, IBM (Erfinder von Fortran)

Die Regeln kontextfreier Grammatiken sind von der Form:

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

Beispiel:

$$\begin{aligned} S &\rightarrow a S b \\ S &\rightarrow \epsilon \end{aligned}$$

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

Konventionen:

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

Weitere Beispiele:

$$\begin{aligned} S &\rightarrow \langle \text{stmt} \rangle \\ \langle \text{stmt} \rangle &\rightarrow \langle \text{if} \rangle \mid \langle \text{while} \rangle \mid \langle \text{rexp} \rangle ; \\ \langle \text{if} \rangle &\rightarrow \text{if} (\langle \text{rexp} \rangle) \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle \\ \langle \text{while} \rangle &\rightarrow \text{while} (\langle \text{rexp} \rangle) \langle \text{stmt} \rangle \\ \langle \text{rexp} \rangle &\rightarrow \text{int} \mid \langle \text{lexp} \rangle \mid \langle \text{lexp} \rangle = \langle \text{rexp} \rangle \mid \dots \\ \langle \text{lexp} \rangle &\rightarrow \text{name} \mid \dots \end{aligned}$$

Weitere Konventionen:

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

Weitere Grammatiken:

$E \rightarrow E + E^0 \mid E * E^1 \mid (E)^2 \mid \text{name}^3 \mid \text{int}^4$
$E \rightarrow E + T^0 \mid T^1$
$T \rightarrow T * F^0 \mid F^1$
$F \rightarrow (E)^0 \mid \text{name}^1 \mid \text{int}^2$

Die beiden Grammatiken beschreiben die **gleiche Sprache** ;-)

Grammatiken sind **Wortersetzungssysteme**.

Die Regeln geben die möglichen Ersetzungsschritte an.

Eine Folge solcher Ersetzungsschritte heißt auch **Ableitung**.

... im letzten Beispiel:

$$\begin{aligned} \underline{E} &\rightarrow \underline{E} + T \\ &\rightarrow \underline{T} + T \\ &\rightarrow T * \underline{E} + T \\ &\rightarrow \underline{T} * \text{int} + T \\ &\rightarrow \underline{E} * \text{int} + T \\ &\rightarrow \text{name} * \text{int} + \underline{T} \\ &\rightarrow \text{name} * \text{int} + \underline{E} \\ &\rightarrow \text{name} * \text{int} + \text{int} \end{aligned}$$

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

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

Den **reflexiven** und **transitiven** Abschluss von \rightarrow schreiben wir: $\rightarrow^* \text{ :-}$)

Bemerkungen:

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

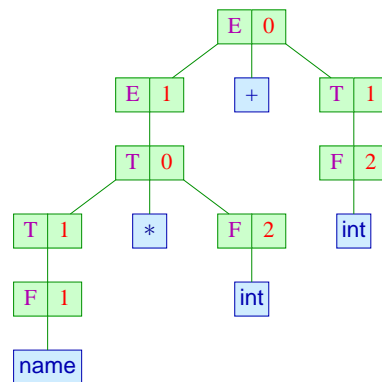
$$\mathcal{L}(G) = \{w \in T^* \mid S \rightarrow^* w\}$$

Achtung:

Die Reihenfolge, in der disjunkte Teile abgeleitet werden, ist unerheblich :-)
Ableitungen eines Symbols stellt man als **Ableitungsbaum** dar :-)

... im Beispiel:

$\underline{E} \rightarrow^0 E + T$
 $\rightarrow^1 \underline{T} + T$
 $\rightarrow^0 T * \underline{F} + T$
 $\rightarrow^2 \underline{T} * \text{int} + T$
 $\rightarrow^1 \underline{F} * \text{int} + T$
 $\rightarrow^1 \text{name} * \text{int} + \underline{T}$
 $\rightarrow^1 \text{name} * \text{int} + \underline{F}$
 $\rightarrow^2 \text{name} * \text{int} + \text{int}$



Ein Ableitungsbaum für $A \in N$:

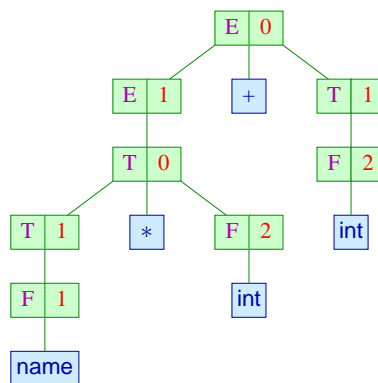
innere Knoten: Regel-Anwendungen;
Wurzel: Regel-Anwendung für A ;
Blätter: Terminale oder ϵ ;

Die Nachfolger von (B, i) entsprechen der rechten Seite der Regel \rightarrow

Beachte:

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

... im Beispiel:



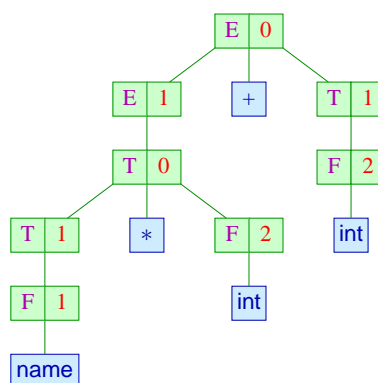
Links-Ableitung: $(E, 0) (E, 1) (T, 0) (T, 1) (F, 1) (F, 2) (T, 1) (F, 2)$

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

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

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

... im Beispiel:



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

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

... unsere beiden Grammatiken:

$E \rightarrow E+E^0$	$ $	$E * E^1$	$ $	$(E)^2$	$ $	name^3	$ $	int^4
$E \rightarrow E+T^0$	$ $	T^1						
$T \rightarrow T * F^0$	$ $	F^1						
$F \rightarrow (E)^0$	$ $	name^1	$ $	int^2				

Die zweite ist eindeutig, die erste nicht :-)

Fazit:

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

Fingerübung:

überflüssige Nichtterminale und Regeln

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

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

Beispiel:

$S \rightarrow a B B \mid b D$

$A \rightarrow B c$

$B \rightarrow S d \mid C$

$C \rightarrow a$

$D \rightarrow B D$

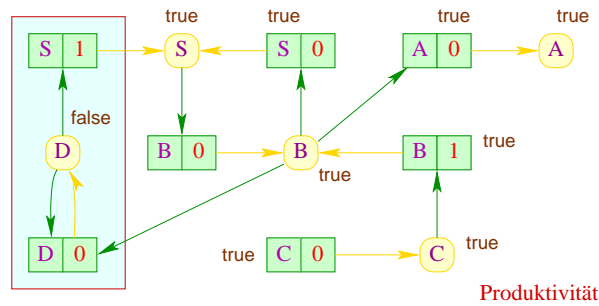
Produktive Nichtterminale: S, A, B, C

Erreichbare Nichtterminale: S, B, C, D

Idee für Produktivität:

And-Or-Graph für die Grammatik

... hier:



And-Knoten: Regeln

Or-Knoten: Nichtterminale

Kanten: $((B, i), B)$ für alle Regeln (B, i)
 $(A, (B, i))$ falls $(B, i) \equiv B \rightarrow \alpha_1 A \alpha_2$

Algorithmus:

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

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

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

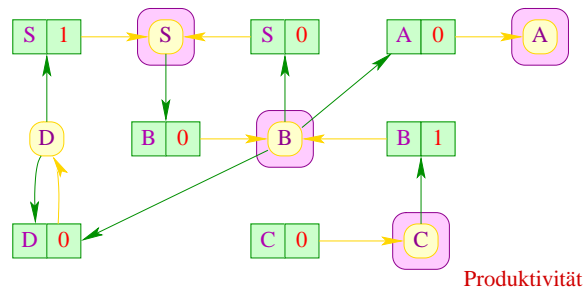
```

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

```

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

... im Beispiel:



Laufzeit:

- Die Initialisierung der Datenstrukturen erfordert lineare Laufzeit.
 - Jede Regel wird maximal einmal in `W` eingefügt.
 - Jedes `A` wird maximal einmal in `result` eingefügt.
- ⇒ Der Gesamtaufwand ist **linear** in der Größe der Grammatik :-)

Korrektheit:

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

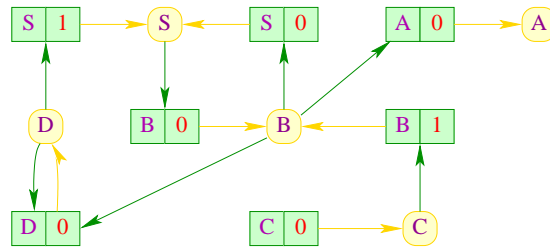
Diskussion:

- Um den Test $(A \in result)$ einfach zu machen, repräsentiert man die Menge $result$ durch ein **Array**.
- W wie auch die Mengen $rhs[A]$ wird man dagegen als **Listen** repräsentieren :-)
- Der Algorithmus funktioniert auch, um **kleinste** Lösungen von **Booleschen** Ungleichungssystemen zu bestimmen :-)
- Die Ermittlung der produktiven Nichtterminale kann benutzt werden, um festzustellen, ob $\mathcal{L}(G) \neq \emptyset$ ist (\rightarrow **Leerheitsproblem**)

Idee für Erreichbarkeit:

Abhängigkeits-Graph

... hier:



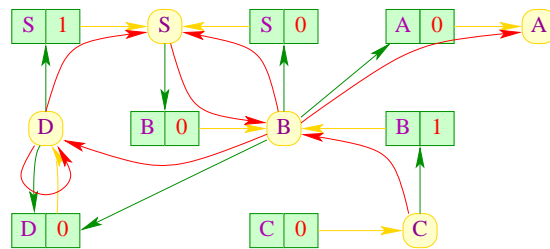
Knoten: Nichtterminale

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

Idee für Erreichbarkeit:

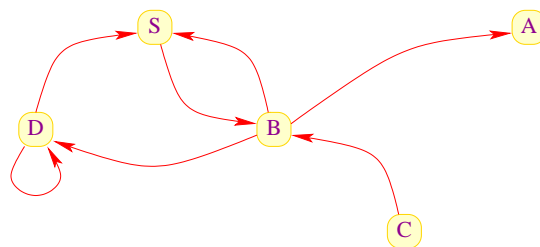
Abhängigkeits-Graph

... hier:

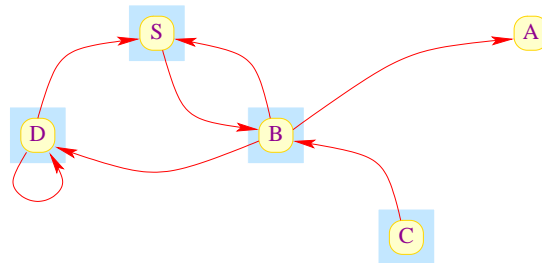


Knoten: Nichtterminale

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



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



Fazit:

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

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

Satz

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

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

Konstruktion:

1. Schritt:

Berechne die Teilmenge $N' \subseteq N$

aller produktiven und erreichbaren Nichtterminale von G .

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

2. Schritt:

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

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

... im Beispiel:

$S \rightarrow aBB \mid bD$

$A \rightarrow Bc$

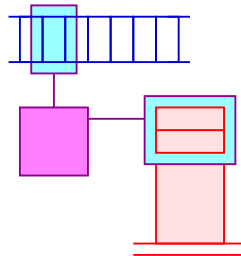
$B \rightarrow Sd \mid C$

$C \rightarrow a$

$D \rightarrow BD$

2.2 Grundlagen: Kellerautomaten

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



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

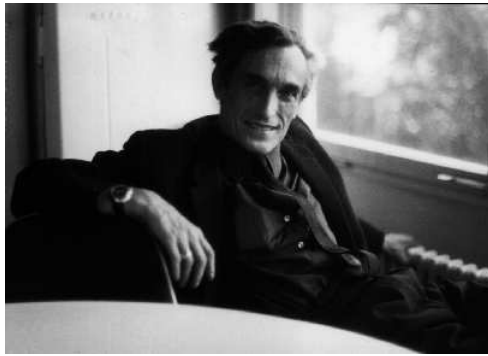


Friedrich L. Bauer, TUM



Klaus Samelson, TUM

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



Marcel-Paul Schützenberger
(1920-1996), Paris



Antony G. Öttinger, Präsident der
ACM 1966-68

Beispiel:

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

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

Achtung:

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

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

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

Mithilfe der Übergänge definieren wir **Berechnungen** von Kellerautomaten :-)

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

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

bestehend aus dem **Kellerinhalt** und dem **noch zu lesenden input figures/**.

... im Beispiel:

Zustände:	0, 1, 2	0	a	11
Anfangszustand:	0	1	a	11
Endzustände:	0, 2	11	b	2
		12	b	2

$$\begin{aligned}
 (0, a a a b b b) &\vdash (11, a a b b b) \\
 &\vdash (111, a b b b) \\
 &\vdash (1111, b b b) \\
 &\vdash (112, b b) \\
 &\vdash (12, b) \\
 &\vdash (2, \epsilon)
 \end{aligned}$$

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

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

Bemerkungen:

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

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

Wir akzeptieren also mit **Endzustand** und leerem Keller :-)

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

Das ist genau dann der Fall wenn für verschiedene Übergänge $(\gamma_1, x, \gamma_2), (\gamma'_1, x', \gamma'_2) \in \delta$ gilt:

Ist γ_1 ein Suffix von γ'_1 , dann muss $x \neq x' \wedge x \neq \epsilon \neq x'$ sein.

... im Beispiel:

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

ist das natürlich der Fall :-))

Satz

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

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

Konstruktion 1: Shift-Reduce-Parser

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

Beispiel:

$S \rightarrow AB$
 $A \rightarrow a$
 $B \rightarrow b$

Der Kellerautomat:

Zustände: q_0, f, a, b, A, B, S ;
Anfangszustand: q_0
Endzustand: f

q_0	a	$q_0 a$
a	ϵ	A
A	b	Ab
b	ϵ	B
AB	ϵ	S
$q_0 S$	ϵ	f

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

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

$$\delta = \{(q, x, qx) \mid q \in Q, x \in T\} \cup // \text{ Shift-Übergänge}$$

$$\{(q \alpha, \epsilon, q A) \mid q \in Q, A \rightarrow \alpha \in P\} \cup // \text{ Reduce-Übergänge}$$

$$\{(q_0 S, \epsilon, f)\} // \text{ Abschluss :-)}$$

Eine Beispiel-Berechnung:

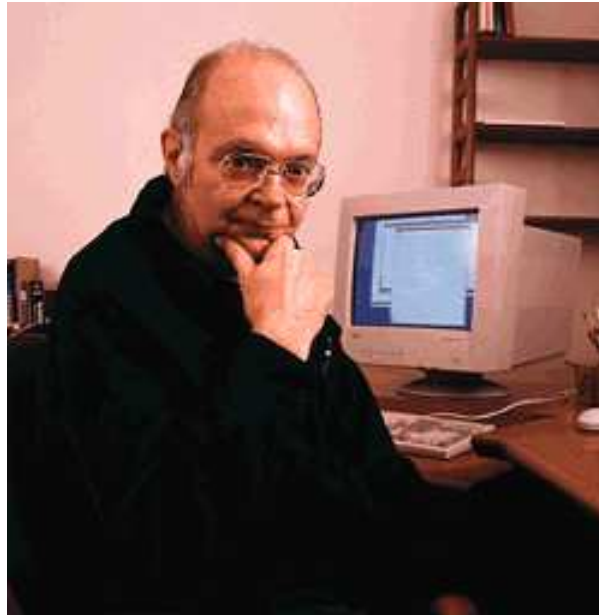
$$\begin{array}{l} (q_0, ab) \vdash (q_0 \boxed{a}, b) \vdash (q_0 A, b) \\ \vdash (q_0 A \boxed{b}, \epsilon) \vdash (q_0 \boxed{AB}, \epsilon) \\ \vdash (q_0 S, \epsilon) \vdash (f, \epsilon) \end{array}$$

Offenbar gilt:

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

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

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



Donald E. Knuth, Stanford

Konstruktion 2: Item-Kellerautomat

- Rekonstruiere eine **Linksableitung**.
- Expandiere Nichtterminale mithilfe einer Regel.
- Verifiziere sukzessive, dass die gewählte Regel mit der Eingabe übereinstimmt.
⇒ Die Zustände sind jetzt **Items**.
- Ein Item ist eine Regel mit **Punkt**:

$$[A \rightarrow \alpha \bullet \beta], \quad A \rightarrow \alpha \beta \in P$$

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

Unser Beispiel:

$$S \rightarrow AB \quad A \rightarrow a \quad B \rightarrow b$$

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

Dann konstruieren wir:

Anfangszustand: $[S' \rightarrow \bullet S]$
 Endzustand: $[S' \rightarrow S \bullet]$

$[S' \rightarrow \bullet S]$	ϵ	$[S' \rightarrow \bullet S] [S \rightarrow \bullet AB]$
$[S \rightarrow \bullet AB]$	ϵ	$[S \rightarrow \bullet AB] [A \rightarrow \bullet a]$
$[A \rightarrow \bullet a]$	a	$[A \rightarrow a \bullet]$
$[S \rightarrow \bullet AB] [A \rightarrow a \bullet]$	ϵ	$[S \rightarrow A \bullet B]$
$[S \rightarrow A \bullet B]$	ϵ	$[S \rightarrow A \bullet B] [B \rightarrow \bullet b]$
$[B \rightarrow \bullet b]$	b	$[B \rightarrow b \bullet]$
$[S \rightarrow A \bullet B] [B \rightarrow b \bullet]$	ϵ	$[S \rightarrow AB \bullet]$
$[S' \rightarrow \bullet S] [S \rightarrow AB \bullet]$	ϵ	$[S' \rightarrow S \bullet]$

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

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

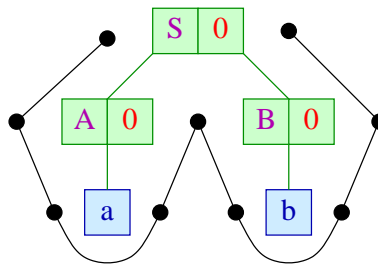
Shifts: $([A \rightarrow \alpha \bullet a \beta], a, [A \rightarrow \alpha a \bullet \beta])$ für $A \rightarrow \alpha a \beta \in P$

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

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

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

... im Beispiel:



Diskussion:

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

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

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

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



Philip M. Lewis, SUNY



Richard E. Stearns, SUNY

Beispiel: $S \rightarrow \epsilon \mid a S b$

Die Übergänge des zugehörigen Item-Kellerautomat:

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

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

2.3 Vorausschau-Mengen

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

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

Beispiel:

ϵ
ab
aa
aa

die Präfixe der Länge 2 :-)

Rechenregeln:

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

$$\begin{aligned} \text{First}_k(\emptyset) &= \emptyset \\ \text{First}_k(L_1 \cup L_2) &= \text{First}_k(L_1) \cup \text{First}_k(L_2) \\ \text{First}_k(L_1 \cdot L_2) &= \text{First}_k(\text{First}_k(L_1) \cdot \text{First}_k(L_2)) \\ &:= \text{First}_k(L_1) \odot \text{First}_k(L_2) \end{aligned}$$

k – Konkatenation

Beachte:

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

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

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

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

Für $k \geq 1$ gilt:

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

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

Idee: Stelle ein **Ungleichungssystem** auf!

Beispiel: $k = 2$

$$\begin{array}{l} E \rightarrow E + T \quad | \quad T \\ T \rightarrow T * F \quad | \quad F \\ F \rightarrow (E) \quad | \quad \text{name} \quad | \quad \text{int} \end{array}$$

Jede Regel gibt Anlass zu einer Inklusionsbeziehung:

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

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

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

Insgesamt erhalten wir das Ungleichungssystem:

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

Allgemein:

$$\begin{array}{l} \text{First}_k(A) \supseteq \text{First}_k(X_1) \odot \dots \odot \text{First}_k(X_m) \\ \text{für jede Regel } A \rightarrow X_1 \dots X_m \in P \text{ mit } X_i \in T \cup N. \end{array}$$

Gesucht:

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

... im Beispiel:

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

... hat die Lösung:

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

Beobachtung:

- Die Menge \mathbb{D}_k der möglichen Werte für $\text{First}_k(A)$ bilden einen **vollständigen Verband** :-)
- Die Operatoren auf den rechten Seiten der Ungleichungen sind **monoton**, d.h. verträglich mit " \subseteq " :-)

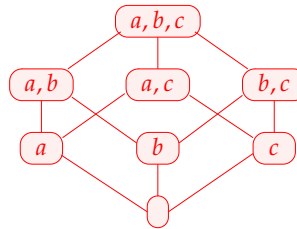
Exkurs: Vollständige Verbände

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

$$\begin{aligned} a \sqsubseteq a & \qquad \qquad \qquad \text{Reflexivität} \\ a \sqsubseteq b \wedge b \sqsubseteq a & \implies a = b & \text{Anti - Symmetrie} \\ a \sqsubseteq b \wedge b \sqsubseteq c & \implies a \sqsubseteq c & \text{Transitivität} \end{aligned}$$

Beispiele:

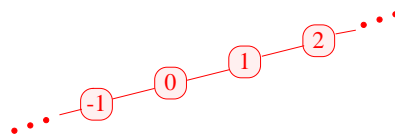
1. $\mathbb{D} = 2^{\{a,b,c\}}$ mit der Relation " \sqsubseteq ":



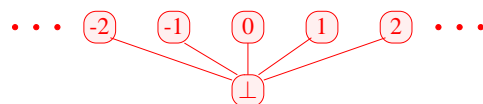
3. \mathbb{Z} mit der Relation " $=$ ":



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



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



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

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

d heißt **kleinste obere Schranke (lub)** falls

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

Achtung:

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

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

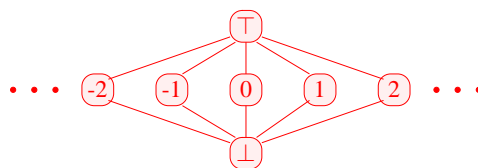
Beachte:

Jeder vollständige Verband besitzt

- ein **kleinstes** Element $\perp = \bigsqcup \emptyset \in \mathbb{D}$;
- ein **größtes** Element $\top = \bigsqcup \mathbb{D} \in \mathbb{D}$.

Beispiele:

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



Es gilt:

Satz:

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

Beweis:

Konstruiere $U = \{u \in \mathbb{D} \mid \forall x \in X : u \sqsubseteq x\}$.
// die Menge der unteren Schranken von X :-)

Setze: $g := \sqcup U$

Behauptung: $g = \sqcap X$

(1) g ist eine **untere Schranke** von X :

Für $x \in X$ gilt:

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

$$\implies x \text{ ist obere Schranke von } U$$

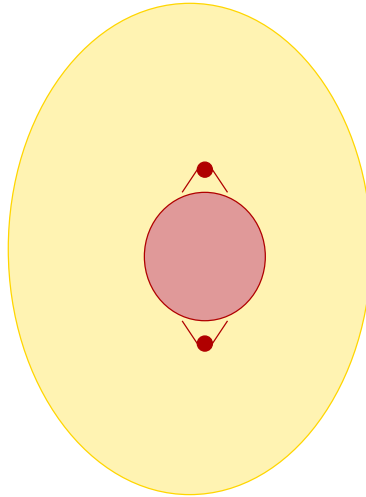
$$\implies g \sqsubseteq x \quad \text{:-)}$$

(2) g ist **größte untere Schranke** von X :

Für jede untere Schranke u von X gilt:

$$u \in U$$

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



Wir suchen **Lösungen** für Ungleichungssysteme der Form:

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

wobei:

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

Ungleichung für $\text{First}_k(A)$:

$$\text{First}_k(A) \sqsupseteq \bigcup \{ \text{First}_k(X_1) \odot \dots \odot \text{First}_k(X_m) \mid A \rightarrow X_1 \dots X_m \in P \}$$

Denn:

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

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

Beispiele:

(1) $\mathbb{D}_1 = \mathbb{D}_2 = 2^U$ für eine Menge U und $f x = (x \cap a) \cup b$.
Offensichtlich ist jedes solche f monoton :-)

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

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

Gesucht: möglichst **kleine** Lösung für:

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

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

Idee:

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

$$F(x_1, \dots, x_n) = (y_1, \dots, y_n) \quad \text{wobei} \quad y_i = f_i(x_1, \dots, x_n).$$
- Sind alle f_i monoton, dann auch F :-)
- Wir **approximieren** sukzessive eine Lösung. Wir konstruieren:

$$\perp, \quad F \perp, \quad F^2 \perp, \quad F^3 \perp, \quad \dots$$

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

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

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

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

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

Die Iteration:

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

Offenbar gilt:

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

$$\underline{\perp} \sqsubseteq F \underline{\perp} \sqsubseteq F^2 \underline{\perp} \sqsubseteq \dots$$

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

Die zweite Aussage folgt mit **vollständiger Induktion**:

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

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

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

da F monoton ist :-)

Fazit:

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

Fragen:

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

Satz

Kleene

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

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

Bemerkung:

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

Beweis:

$$\begin{aligned} (1) \quad f d_0 = d_0 : \quad f d_0 &= f(\bigsqcup_{m \geq 0} (f^m \perp)) \\ &= \bigsqcup_{m \geq 0} (f^{m+1} \perp) \quad \text{wegen Stetigkeit :-)} \\ &= \perp \sqcup (\bigsqcup_{m \geq 0} (f^{m+1} \perp)) \\ &= \bigsqcup_{m \geq 0} (f^m \perp) \\ &= d_0 \end{aligned}$$

(2) d_0 ist **kleinster** Fixpunkt:

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

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

Bemerkung:

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

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

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

$\implies d_0$ ist der kleinste Wert x mit $x \sqsupseteq f x$

Anwendung:

Sei $x_i \sqsupseteq f_i(x_1, \dots, x_n)$, $i = 1, \dots, n$ (*)
ein **Ungleichungssystem**, wobei alle $f_i : \mathbb{D}^n \rightarrow \mathbb{D}$ **monoton** sind.

\implies kleinste Lösung von (*) \equiv kleinster Fixpunkt von F :-)

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

Satz

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

$$\text{First}_k(A) \sqsupseteq \text{First}_k(X_1) \odot \dots \odot \text{First}_k(X_m), \quad A \rightarrow X_1 \dots X_m \in P$$

Beweis-Idee:

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

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

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

Fazit:

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

Achtung: Naive Fixpunkt-Iteration ist ziemlich **ineffizient** :-)

Idee: Round Robin Iteration

Benutze bei der Iteration nicht die Werte der letzten Iteration, sondern die jeweils **aktuellen** :-)

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

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

Die Round-Robin-Iteration:

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

Der Code für Round Robin Iteration sieht in Java so aus:

```
for (i = 1; i ≤ n; i++) xi = ⊥;
do {
    finished = true;
    for (i = 1; i ≤ n; i++) {
        new = fi(x1, ..., xn);
        if (!(xi ⊇ new)) {
            finished = false;
            xi = xi ⊔ new;
        }
    }
} while (!finished);
```

Zur Korrektheit:

Sei $y_i^{(d)}$ die i -te Komponente von $F^d \perp$.

Sei $x_i^{(d)}$ der Wert von x_i nach der i -ten RR-Iteration.

Man zeigt:

(1) $y_i^{(d)} \sqsubseteq x_i^{(d)}$:-)

(2) $x_i^{(d)} \sqsubseteq z_i$ für jede Lösung (z_1, \dots, z_n) :-)

- (3) Terminiert RR-Iteration nach d Runden, ist $(x_1^{(d)}, \dots, x_n^{(d)})$ eine Lösung :-))

Unsere Anwendung:

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

Die RR-Iteration:

First ₂	1	2	3
F	name, int	(name, (int	((
T	name, int	(name, (int, name *, int *	((
E	name, int	(name, (int, name *, int *, name +, int +	((

Der Einfachheit halber haben wir in jeder Iteration nur die **neuen** Elemente vermerkt :-)

Diskussion:

- Die Länge h der längsten echt aufsteigenden Kette nennen wir auch **Höhe** von \mathbb{D}
...
- Im Falle von First_k ist die Höhe des Verbands **exponentiell** in k :-)
- Die Anzahl der Runden von RR-Iteration ist beschränkt durch $\mathcal{O}(n \cdot h)$ (n die Anzahl der Variablen)
- Die **praktische** Effizienz von RR-Iteration hängt allerdings auch von der **Anordnung** der Variablen ab :-)
- Anstelle von RR-Iteration gibt es auch schnellere Fixpunkt-Verfahren, die aber im schlimmsten Fall immer noch exponentiell sind :-((
 \implies Man beschränkt sich i.a. auf **kleine** k !!!

2.4 Topdown Parsing

Idee:

- Benutze den Item-Kellerautomaten.
- Benutze die nächsten k Zeichen, um die Regeln für die Expansionen zu bestimmen ;-)
- Eine Grammatik heißt $LL(k)$, falls dies immer eindeutig möglich ist.

Wir definieren:

Eine reduzierte Grammatik heißt dann $LL(k)$, falls für je zwei verschiedene Regeln $A \rightarrow \alpha$, $A \rightarrow \alpha' \in P$ und jede Ableitung $S \xrightarrow{*}_L u A \beta$ mit $u \in T^*$ gilt:

$$\text{First}_k(\alpha \beta) \cap \text{First}_k(\alpha' \beta) = \emptyset$$

Beispiel 1:

```
S → if ( E ) S else S |  
    while ( E ) S |  
    E ;  
E → id
```

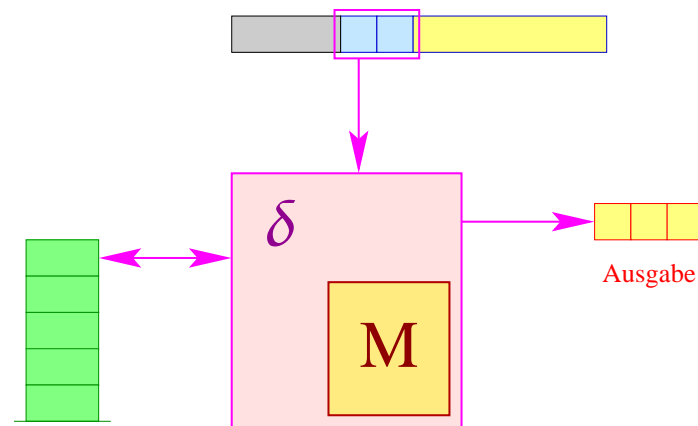
ist $LL(1)$, da $\text{First}_k(E) = \{\text{id}\}$:-)

Beispiel 2:

```
S → if ( E ) S else S |  
    if ( E ) S |  
    while ( E ) S |  
    E ;  
E → id
```

... ist nicht $LL(k)$ für jedes $k > 0$.

Struktur des $LL(k)$ -Parsers:



- Der Parser sieht ein Fenster der Länge k der Eingabe;
- er realisiert im Wesentlichen den Item-Kellerautomaten;
- die Tabelle $M[q, w]$ enthält die jeweils zuwählende Regel :-)

... im Beispiel:

$$\begin{aligned}
 S &\rightarrow \text{if } (E) S \text{ else } S^0 \mid \\
 &\quad \text{while } (E) S^1 \mid \\
 &\quad E;^2 \\
 E &\rightarrow \text{id}^0
 \end{aligned}$$

Zustände: Items

Tabelle:

	if	while	id
$[\dots \rightarrow \dots \bullet S \dots]$	0	1	2
$[\dots \rightarrow \dots \bullet E \dots]$	—	—	0

Im Allgemeinen ...

- ist die Menge der möglichen nächsten k Zeichen gegeben durch:

$$\text{First}_k(\alpha\beta) = \text{First}_k(\alpha) \odot \text{First}_k(\beta)$$

wobei:

- (1) α die rechte Seite der passenden Regel;
- (2) β ein möglicher rechter Kontext von A ist :-)
- $\text{First}_k(\beta)$ müssen wir dynamisch akkumulieren.

\implies Wir erweitern Items um Vorausschau-Mengen ...

Ein **erweitertes** Item ist ein Paar: $[A \rightarrow \alpha \bullet \gamma, L]$ ($A \rightarrow \alpha \gamma \in P, L \subseteq T^{\leq k}$)

Die Menge L benutzen wir, um $\text{First}_k(\beta)$ für den rechten Kontext β von A zu repräsentieren :-)

Konstruktion:

Zustände: erweiterte Items

Anfangszustand: $[S' \rightarrow \bullet S, \{\epsilon\}]$

Endzustand: $[S' \rightarrow S \bullet, \{\epsilon\}]$

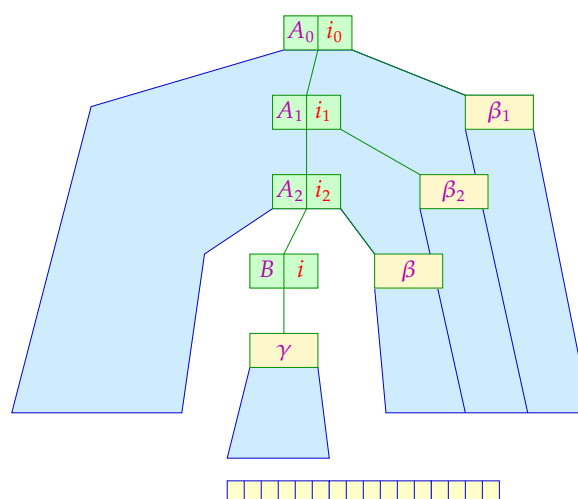
Expansionen: $([A \rightarrow \alpha \bullet B \beta, L], \epsilon, [A \rightarrow \alpha \bullet B \beta, L] [B \rightarrow \bullet \gamma, \text{First}_k(\beta) \odot L])$

für $A \rightarrow \alpha B \beta, B \rightarrow \gamma \in P$

Übergänge: **Shifts:** $([A \rightarrow \alpha \bullet a \beta, L], a, [A \rightarrow \alpha a \bullet \beta, L])$ für $A \rightarrow \alpha a \beta \in P$

Reduce: $([A \rightarrow \alpha \bullet B \beta, L] [B \rightarrow \gamma \bullet, L'], \epsilon, [A \rightarrow \alpha B \bullet \beta, L])$

für $A \rightarrow \alpha B \beta, B \rightarrow \gamma \in P$

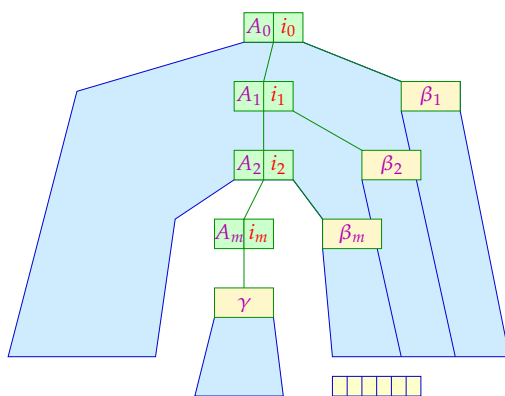


Die Vorausschau-Tabelle:

Wir setzen $M[A \rightarrow \alpha \bullet B \beta, L, w] = i$ genau dann wenn (B, i) die Regel $B \rightarrow \gamma$ ist und: $w \in \text{First}_k(\gamma) \odot \text{First}_k(\beta) \odot L$

$([A_0 \rightarrow \bullet \alpha_1 A_1 \beta_1, L_1], uv) \vdash^* ([A_0 \rightarrow \alpha_1 \bullet A_1 \beta_1, L_1] \dots [A_{m-1} \rightarrow \alpha_m \bullet A_m \beta_m, L_m], v)$
 $\vdash^* ([A_0 \rightarrow \alpha_1 A_1 \beta_1 \bullet, L_1], \epsilon) \dots$ gilt genau dann wenn:

- (1) $\alpha_1 \dots \alpha_m \rightarrow^* u$
- (2) $A_m \beta_m \dots \beta_1 \rightarrow^* v$
- (3) $L_m = \text{First}_k(\beta_{m-1}) \odot \dots \odot \text{First}_k(\beta_1) \odot L_1$



Satz

Die reduzierte kontextfreie Grammatik G ist $LL(k)$ genau dann wenn die k -Vorausschau-Tabelle für alle benötigten erweiterten Items wohl-definiert ist.

Diskussion:

- Der erweiterte Item-Kellerautomat zusammen mit einer k -Vorausschau-Tabelle erlaubt die deterministische Rekonstruktion einer Links-Ableitung :-)
- Die Anzahl der Vorausschau-Mengen L kann sehr groß sein :-)
- ...

Beispiel: $S \rightarrow \epsilon \mid a S b$

Die Übergänge des erweiterten Item-Kellerautomat ($k = 1$):

0	$[S' \rightarrow \bullet S, \{\epsilon\}]$	ϵ	$[S' \rightarrow \bullet S, \{\epsilon\}] [S \rightarrow \bullet, \{\epsilon\}]$
1	$[S' \rightarrow \bullet S, \{\epsilon\}]$	ϵ	$[S' \rightarrow \bullet S, \{\epsilon\}] [S \rightarrow \bullet a S b, \{\epsilon\}]$
2	$[S \rightarrow \bullet a S b, \{\epsilon\}]$ $[S \rightarrow \bullet a S b, \{b\}]$	a	$[S \rightarrow a \bullet S b, \{\epsilon\}]$ $[S \rightarrow a \bullet S b, \{b\}]$
3	$[S \rightarrow a \bullet S b, \{\epsilon\}]$ $[S \rightarrow a \bullet S b, \{b\}]$	ϵ	$[S \rightarrow a \bullet S b, \{\epsilon\}] [S \rightarrow \bullet, \{b\}]$ $[S \rightarrow a \bullet S b, \{b\}] [S \rightarrow \bullet, \{b\}]$
4	$[S \rightarrow a \bullet S b, \{\epsilon\}]$ $[S \rightarrow a \bullet S b, \{b\}]$	ϵ	$[S \rightarrow a \bullet S b, \{\epsilon\}] [S \rightarrow \bullet a S b, \{b\}]$ $[S \rightarrow a \bullet S b, \{b\}] [S \rightarrow \bullet a S b, \{b\}]$
5	$[S \rightarrow a \bullet S b, \{\epsilon\}] [S \rightarrow \bullet, \{b\}]$ $[S \rightarrow a \bullet S b, \{b\}] [S \rightarrow \bullet, \{b\}]$	ϵ	$[S \rightarrow a S \bullet b, \{\epsilon\}]$ $[S \rightarrow a S \bullet b, \{b\}]$

6	$[S \rightarrow a \bullet S b, \{\epsilon\}] [S \rightarrow a S b \bullet, \{b\}]$ $[S \rightarrow a \bullet S b, \{b\}] [S \rightarrow a S b \bullet, \{b\}]$	ϵ	$[S \rightarrow a S \bullet b, \{\epsilon\}]$ $[S \rightarrow a S \bullet b, \{b\}]$
7	$[S \rightarrow a S \bullet b, \{\epsilon\}]$ $[S \rightarrow a S \bullet b, \{b\}]$	b	$[S \rightarrow a S b \bullet, \{\epsilon\}]$ $[S \rightarrow a S b \bullet, \{b\}]$
8	$[S' \rightarrow \bullet S, \{\epsilon\}] [S \rightarrow \bullet, \{\epsilon\}]$	ϵ	$[S' \rightarrow S \bullet, \{\epsilon\}]$
9	$[S' \rightarrow \bullet S, \{\epsilon\}] [S \rightarrow a S b \bullet, \{\epsilon\}]$	ϵ	$[S' \rightarrow S \bullet, \{\epsilon\}]$

Die Vorausschau-Tabelle:

	ϵ	a	b
$[S' \rightarrow \bullet S, \{\epsilon\}]$	0	1	-
$[S \rightarrow a \bullet S b, \{\epsilon\}]$	-	1	0
$[S \rightarrow a \bullet S b, \{b\}]$	-	1	0

Beobachtung:

- Die auszuwählende Regel hängt hier ja gar nicht von den Erweiterungen der Items ab !!!
- Unter dieser Voraussetzung können wir den Item-Kellerautomaten **ohne** Erweiterung benutzen :-)
- Hängt die auszuwählende Regel **nur** von der aktuellen Vorausschau w ab, nennen wir G auch **stark LL(k)** ...

Wir definieren: $\text{Follow}_k(A) = \cup \{ \text{First}_k(\beta) \mid S \xrightarrow{*}_L u A \beta \}.$

Die reduzierte kontextfreie Grammatik G heißt **stark $LL(k)$** , falls für je zwei verschiedene $A \rightarrow \alpha, A \rightarrow \alpha' \in P$:

$$\text{First}_k(\alpha) \odot \text{Follow}_k(A) \cap \text{First}_k(\alpha') \odot \text{Follow}_k(A) = \emptyset$$

... im Beispiel: $S \rightarrow \epsilon \mid a S b$

$$\text{Follow}_1(S) = \{ \epsilon, b \}$$

$$\text{First}_1(\epsilon) \odot \text{Follow}_1(S) = \{ \epsilon \} \odot \{ \epsilon, b \} = \{ \epsilon, b \}$$

$$\text{First}_1(a S b) \odot \text{Follow}_1(S) = \{ a \} \odot \{ \epsilon, b \} = \{ a \}$$

Wir schließen: Die Grammatik ist in der Tat **stark $LL(1)$** :-)

Ist G eine **starke $LL(k)$** -Grammatik, können wir die Vorausschau-Tabelle statt mit (erweiterten) Items mit Nichtterminalen indizieren :-)

Wir setzen $M[B, w] = i$ genau dann wenn (B, i) die Regel $B \rightarrow \gamma$ ist und: $w \in \text{First}_k(\gamma) \odot \text{Follow}_k(B).$

... im Beispiel: $S \rightarrow \epsilon \mid a S b$

	ϵ	a	b
S	0	1	0

Satz

- Jede **starke $LL(k)$** -Grammatik ist auch **$LL(k)$** :-)
- Jede **$LL(1)$** -Grammatik ist bereits **stark $LL(1)$** :-))

Beweis:

Sei G **stark $LL(k)$** .

Betrachte eine Ableitung $S \xrightarrow{*}_L u A \beta$ und Regeln $A \rightarrow \alpha, A \rightarrow \alpha' \in P.$

Dann haben wir:

$$\begin{aligned}
\text{First}_k(\alpha\beta) \cap \text{First}_k(\alpha'\beta) &= \text{First}_k(\alpha) \odot \text{First}_k(\beta) \cap \text{First}_k(\alpha') \odot \text{First}_k(\beta) \\
&\subseteq \text{First}_k(\alpha) \odot \text{Follow}_k(A) \cap \text{First}_k(\alpha') \odot \text{Follow}_k(A) \\
&= \emptyset
\end{aligned}$$

Folglich ist G auch $LL(k)$:-)

Sei G $LL(1)$.

Betrachte zwei verschiedene Regeln $A \rightarrow \alpha, A \rightarrow \alpha' \in P$.

Fall 1: $\epsilon \in \text{First}_1(\alpha) \cap \text{First}_1(\alpha')$.

Dann kann G nicht $LL(1)$ sein :-)

Fall 2: $\epsilon \notin \text{First}_1(\alpha) \cup \text{First}_1(\alpha')$.

Sei $S \xrightarrow{*}_L u A \beta$. Da G $LL(1)$ ist, gilt:

$$\begin{aligned}
&\text{First}_1(\alpha) \odot \text{Follow}_1(A) \cap \text{First}_1(\alpha') \odot \text{Follow}_1(A) \\
&= \text{First}_1(\alpha) \cap \text{First}_1(\alpha') \\
&= \text{First}_1(\alpha) \odot \text{First}_1(\beta) \cap \text{First}_1(\alpha') \odot \text{First}_1(\beta) \\
&= \emptyset
\end{aligned}$$

Fall 3: $\epsilon \in \text{First}_1(\alpha)$ und $\epsilon \notin \text{First}_1(\alpha')$.

Dann gilt:

$$\begin{aligned}
&\text{First}_1(\alpha) \odot \text{Follow}_1(A) \cap \text{First}_1(\alpha') \odot \text{Follow}_1(A) \\
&= \text{First}_1(\alpha) \odot \text{Follow}_1(A) \cap \text{First}_1(\alpha') \\
&= \text{First}_1(\alpha) \odot (\cup\{\text{First}_1(\beta) \mid S \xrightarrow{*}_L u A \beta\}) \cap \text{First}_1(\alpha') \\
&= (\cup\{\text{First}_1(\alpha) \odot \text{First}_1(\beta) \mid S \xrightarrow{*}_L u A \beta\}) \cap \text{First}_1(\alpha') \\
&= \cup\{\text{First}_1(\alpha) \odot \text{First}_1(\beta) \cap \text{First}_1(\alpha') \mid S \xrightarrow{*}_L u A \beta\} \\
&= \cup\{\emptyset \mid S \xrightarrow{*}_L u A \beta\} \\
&= \emptyset
\end{aligned}$$

Fall 4: $\epsilon \notin \text{First}_1(\alpha)$ und $\epsilon \in \text{First}_1(\alpha')$: analog :-)

Beispiel:

$$S \rightarrow a A a a^0 \mid b A b a^1$$

$$A \rightarrow b^0 \mid \epsilon^1$$

Offenbar ist die Grammatik $LL(2)$:-) Andererseits gilt:

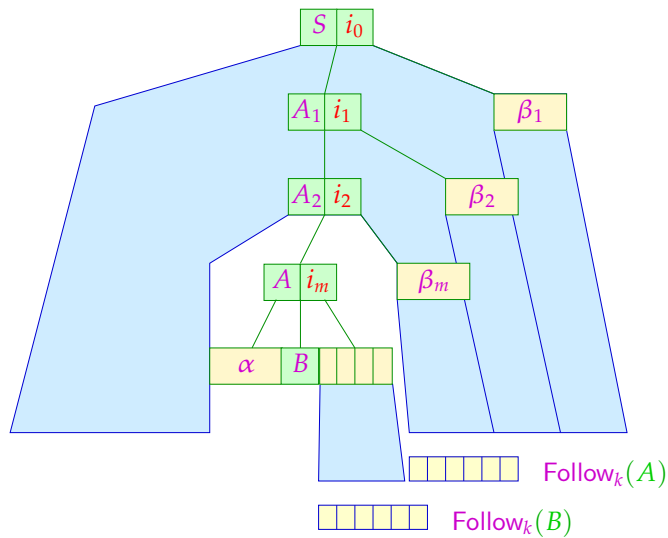
$$\begin{aligned} & \text{First}_2(b) \odot \text{Follow}_2(A) \cap \text{First}_2(\epsilon) \odot \text{Follow}_2(A) \\ &= \{b\} \odot \{aa, ba\} \cap \{\epsilon\} \odot \{aa, ba\} \\ &= \{ba, bb\} \cap \{aa, ba\} \\ &\neq \emptyset \end{aligned}$$

Folglich ist die Grammatik **nicht** stark $LL(2)$:-)

Wir schließen:

- Für $k > 1$ ist nicht jede $LL(k)$ -Grammatik automatisch stark $LL(k)$.
- Zu jeder $LL(k)$ -Grammatik kann jedoch eine äquivalente starke $LL(k)$ -Grammatik konstruiert werden \implies Übung!

Berechnung von $\text{Follow}_k(B)$:



Berechnung von $\text{Follow}_k(B)$:

Idee:

- Wir stellen ein Ungleichungssystem auf :-)
- ϵ ist ein möglicher rechter Kontext von S :-)
- Mögliche rechte Kontexte der linken Seite einer Regel propagieren wir an das Ende jeder rechten Seite ...

... im Beispiel:

$$S \rightarrow \epsilon \mid a S b$$

$$\text{Follow}_k(S) \supseteq \{\epsilon\}$$

$$\text{Follow}_k(S) \supseteq \{b\} \odot \text{Follow}_k(S)$$

Allgemein:

$$\text{Follow}_k(S) \supseteq \{\epsilon\}$$

$$\text{Follow}_k(B) \supseteq \text{First}_k(X_1) \odot \dots \odot \text{First}_k(X_m) \odot \text{Follow}_k(A)$$

$$\text{für } A \rightarrow \alpha B X_1 \dots X_m \in P$$

Diskussion:

- Man überzeugt sich, dass die kleinste Lösung dieses Ungleichungssystems tatsächlich die Mengen $\text{Follow}_k(B)$ liefert :-)
- Die Größe der auftretenden Mengen steigt mit k rapide :-)
- In praktischen Systemen wird darum meist nur der Fall $k = 1$ implementiert ...

2.5 Schnelle Berechnung von Vorausschau-Mengen

Im Fall $k = 1$ lassen sich **First**, **Follow** besonders effizient berechnen :-)

Beobachtung:

Seien $L_1, L_2 \subseteq T \cup \{\epsilon\}$ mit $L_1 \neq \emptyset \neq L_2$. Dann ist:

$$L_1 \odot L_2 = \begin{cases} L_1 & \text{falls } \epsilon \notin L_1 \\ (L_1 \setminus \{\epsilon\}) \cup L_2 & \text{sonst} \end{cases}$$

Ist G reduziert, sind alle Mengen $\text{First}_1(A)$ nichtleer :-)

Idee:

- Behandle ϵ separat!
Sei $\text{empty}(X) = \text{true}$ gdw. $X \rightarrow^* \epsilon$.
- Definiere die ϵ -freien **First**₁-Mengen

$$\begin{aligned} F_\epsilon(a) &= \{a\} && \text{für } a \in T \\ F_\epsilon(A) &= \text{First}_1(A) \setminus \{\epsilon\} && \text{für } A \in N \end{aligned}$$

- Konstruiere direkt ein Ungleichungssystem für $F_\epsilon(A)$:

$$F_\epsilon(A) \supseteq F_\epsilon(X_j) \quad \text{falls } A \rightarrow X_1 \dots X_m \in P, \\ \text{empty}(X_1) \wedge \dots \wedge \text{empty}(X_{j-1})$$

... im Beispiel:

$$\begin{array}{lcl} E & \rightarrow & E + T \quad | \quad T \\ T & \rightarrow & T * F \quad | \quad F \\ F & \rightarrow & (E) \quad | \quad \text{name} \quad | \quad \text{int} \end{array}$$

wobei $\text{empty}(E) = \text{empty}(T) = \text{empty}(F) = \text{false}$.

Deshalb erhalten wir:

$$\begin{array}{lcl}
 F_\epsilon(S') \supseteq F_\epsilon(E) & F_\epsilon(E) \supseteq F_\epsilon(E) \\
 F_\epsilon(E) \supseteq F_\epsilon(T) & F_\epsilon(T) \supseteq F_\epsilon(T) \\
 F_\epsilon(T) \supseteq F_\epsilon(F) & F_\epsilon(F) \supseteq \{ (, \text{name}, \text{int} \}
 \end{array}$$

Entsprechend konstruieren wir zur Berechnung von Follow_1 :

$$\begin{array}{l}
 \text{Follow}_1(S) \supseteq \{\epsilon\} \\
 \text{Follow}_1(B) \supseteq F_\epsilon(X_j) \quad \text{falls } A \rightarrow \alpha B X_1 \dots X_m \in P, \\
 \qquad \qquad \qquad \text{empty}(X_1) \wedge \dots \wedge \text{empty}(X_{j-1}) \\
 \text{Follow}_1(B) \supseteq \text{Follow}_1(A) \quad \text{falls } A \rightarrow \alpha B X_1 \dots X_m \in P, \\
 \qquad \qquad \qquad \text{empty}(X_1) \wedge \dots \wedge \text{empty}(X_m)
 \end{array}$$

... im Beispiel:

$$\begin{array}{lcl}
 E \rightarrow E + T & | & T \\
 T \rightarrow T * F & | & F \\
 F \rightarrow (E) & | & \text{name} \quad | \quad \text{int}
 \end{array}$$

... erhalten wir:

$$\begin{array}{lcl}
 \text{Follow}_1(S') \supseteq \{\epsilon\} & \text{Follow}_1(E) \supseteq \text{Follow}_1(S') \\
 \text{Follow}_1(E) \supseteq \{+,)\} & \text{Follow}_1(T) \supseteq \{*\} \\
 \text{Follow}_1(T) \supseteq \text{Follow}_1(E) & \text{Follow}_1(F) \supseteq \text{Follow}_1(T)
 \end{array}$$

Diskussion:

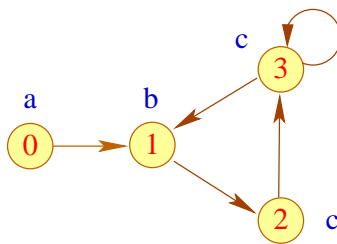
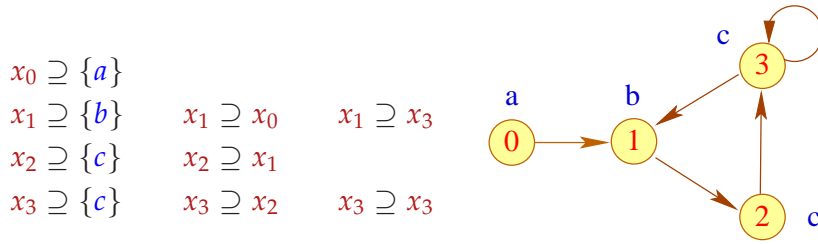
- Diese Ungleichungssysteme bestehen aus Ungleichungen der Form:

$$x \supseteq y \quad \text{bzw.} \quad x \supseteq d$$

für Variablen x, y und $d \in \mathbb{D}$.

- Solche Ungleichungssysteme heißen **reine Vereinigungs-Probleme** :-)
- Diese Probleme können mit **linearem** Aufwand gelöst werden ...

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

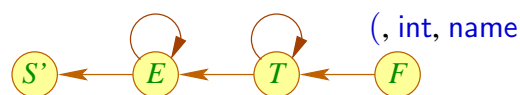


Vorgehen:

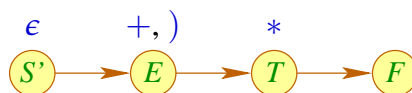
- Konstruiere den **Variablen-Abhängigkeitsgraph** zum Ungleichungssystem.
- Innerhalb einer **starken Zusammenhangskomponente** haben alle Variablen den gleichen Wert :-)
- Hat eine SZK keine eingehenden Kanten, erhält man ihren Wert, indem man die kleinste obere Schranke aller Werte in der SZK berechnet :-)
- Gibt es eingehende Kanten, muss man zusätzlich die Werte an deren Startknoten hinzufügen :-)

... für unsere Beispiel-Grammatik:

First₁ :



Follow₁ :



2.6 Bottom-up Analyse

Achtung:

- Viele Grammatiken sind nicht $LL(k)$:-)
- Eine Grund ist Links-Rekursivität ...
- Die Grammatik G heißt links-rekursiv, falls

$$A \rightarrow^+ A \beta \quad \text{für ein } A \in N, \beta \in (T \cup N)^*$$

Beispiel:

$$\begin{array}{l} E \rightarrow E + T \quad | \quad T \\ T \rightarrow T * F \quad | \quad F \\ F \rightarrow (E) \quad | \quad \text{name} \quad | \quad \text{int} \end{array}$$

... ist links-rekursiv :-)

Satz

Ist die Grammatik G reduziert und links-rekursiv, dann ist G nicht $LL(k)$ für jedes k .

Beweis: Vereinfachung: $A \rightarrow A \beta \in P$

$$A \text{ erreichbar} \implies S \xrightarrow{*}_L u A \gamma \xrightarrow{*}_L u A \beta^n \gamma \quad \text{für jedes } n \geq 0.$$

$$A \text{ produktiv} \implies \exists A \rightarrow \alpha : \alpha \neq A \beta.$$

Annahme: G ist $LL(k)$:-) Dann gilt für alle $n \geq 0$:

$$\text{First}_k(\alpha \beta^n \gamma) \cap \text{First}_k(A \beta \beta^n \gamma) = \emptyset$$

$$\text{Weil } \text{First}_k(\alpha \beta^{n+1} \gamma) \subseteq \text{First}_k(A \beta^{n+1} \gamma)$$

$$\text{folgt: } \text{First}_k(\alpha \beta^n \gamma) \cap \text{First}_k(\alpha \beta^{n+1} \gamma) = \emptyset$$

Fall 1: $\beta \rightarrow^* \epsilon$ — Widerspruch !!!

Fall 2: $\beta \rightarrow^* w \neq \epsilon \implies \text{First}_k(\alpha \beta^k \gamma) \cap \text{First}_k(\alpha \beta^{k+1} \gamma) \neq \emptyset$:-)

Bottom-up Parsing:

Wir rekonstruieren reverse Rechtsableitungen :-)

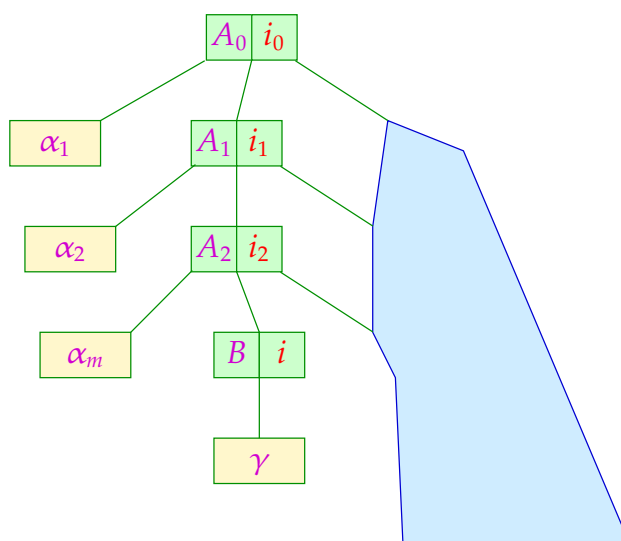
Dazu versuchen wir, für den Shift-Reduce-Parser $M_G^{(1)}$ die Reduktionsstellen zu identifizieren ...

Betrachte eine Berechnung dieses Kellerautomaten:

$$(q_0 \alpha \gamma, v) \vdash (q_0 \alpha B, v) \vdash^* (q_0 S, \epsilon)$$

$\alpha \gamma$ nennen wir **zuverlässiges Präfix** für das vollständige Item $[B \rightarrow \gamma \bullet]$.

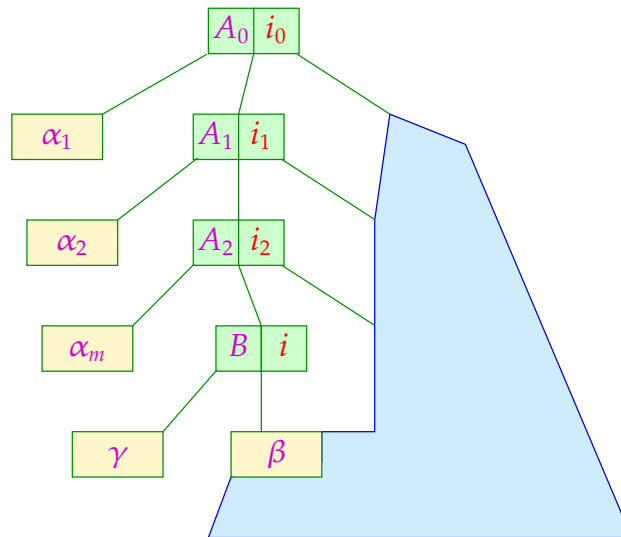
Dann ist $\alpha \gamma$ zuverlässig für $[B \rightarrow \gamma \bullet]$ gdw. $S \xrightarrow*_R \alpha B v$:-)



... wobei $\alpha = \alpha_1 \dots \alpha_m$:-)

Umgekehrt können wir zu jedem möglichen Wort α' die Menge aller möglicherweise später passenden Regeln ermitteln ...

Das Item $[B \rightarrow \gamma \bullet \beta]$ heißt **gültig** für α' gdw. $S \xrightarrow*_R \alpha B v$ mit $\alpha' = \alpha \gamma$:



... wobei $\alpha = \alpha_1 \dots \alpha_m$:-)

Beobachtung:

Die Menge der zuverlässigen Präfixe aus $(N \cup T)^*$ für (vollständige) Items kann mithilfe eines endlichen Automaten berechnet werden :-)

Zustände: Items :-)

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

Endzustände: $\{[B \rightarrow \gamma \bullet] \mid B \rightarrow \gamma \in P\}$

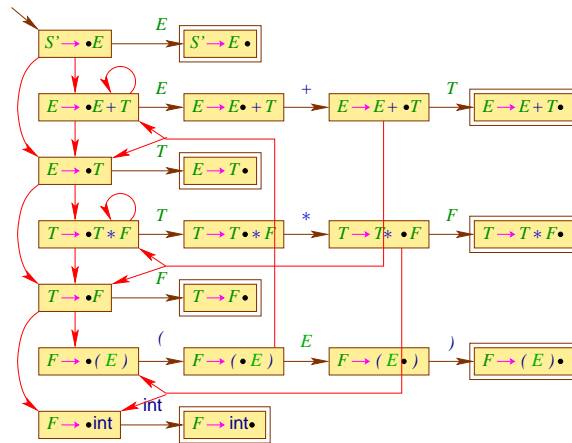
Übergänge:

- (1) $([A \rightarrow \alpha \bullet X \beta], X, [A \rightarrow \alpha X \bullet \beta]), \quad X \in (N \cup T), A \rightarrow \alpha X \beta \in P;$
- (2) $([A \rightarrow \alpha \bullet B \beta], \epsilon, [B \rightarrow \bullet \gamma]), \quad A \rightarrow \alpha B \beta, B \rightarrow \gamma \in P;$

Den Automaten $c(G)$ nennen wir **charakteristischen Automaten** für G .

Beispiel:

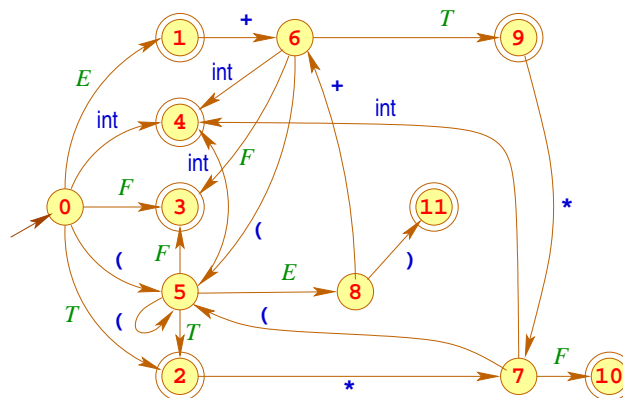
$$\begin{array}{lcl}
 E & \rightarrow & E + T \quad | \quad T \\
 T & \rightarrow & T * F \quad | \quad F \\
 F & \rightarrow & (E) \quad | \quad \text{int}
 \end{array}$$



Den kanonischen $LR(0)$ -Automaten $LR(G)$ erhalten wir aus $c(G)$, indem wir:

- (1) nach jedem lesenden Übergang beliebig viele ϵ -Übergänge einschieben (unsere Konstruktion 1 zur Beseitigung von ϵ -Übergängen :-)
- (2) die Teilmengenkonstruktion anwenden.

... im Beispiel:



Dazu konstruieren wir:

$$q_0 = \{[S' \rightarrow \bullet E], [E \rightarrow \bullet E + T], [E \rightarrow \bullet T], [T \rightarrow \bullet T * F], [T \rightarrow \bullet F], [F \rightarrow \bullet (E)], [F \rightarrow \bullet \text{int}]\}$$

$$q_1 = \delta(q_0, E) = \{[S' \rightarrow E \bullet], [E \rightarrow E \bullet + T]\}$$

$$q_2 = \delta(q_0, T) = \{[E \rightarrow T \bullet], [T \rightarrow T \bullet * F]\}$$

$$q_3 = \delta(q_0, F) = \{[T \rightarrow F \bullet]\}$$

$$q_4 = \delta(q_0, \text{int}) = \{[F \rightarrow \text{int} \bullet]\}$$

$$q_5 = \delta(q_0, () = \{[F \rightarrow (\bullet E)], [E \rightarrow \bullet E + T], [E \rightarrow \bullet T], [T \rightarrow \bullet T * F], [T \rightarrow \bullet F], [F \rightarrow \bullet (E)], [F \rightarrow \bullet \text{int}]\}$$

$$q_7 = \delta(q_2, *) = \{[T \rightarrow T * \bullet F], [F \rightarrow \bullet (E)], [F \rightarrow \bullet \text{int}]\}$$

$$q_8 = \delta(q_5, E) = \{[F \rightarrow (E \bullet)], [E \rightarrow E \bullet + T]\}$$

$$q_6 = \delta(q_1, +) = \{[E \rightarrow E + \bullet T], [T \rightarrow \bullet T * F], [T \rightarrow \bullet F], [F \rightarrow \bullet (E)], [F \rightarrow \bullet \text{int}]\}$$

$$q_9 = \delta(q_6, T) = \{[E \rightarrow E + T \bullet], [T \rightarrow T \bullet * F]\}$$

$$q_{10} = \delta(q_7, F) = \{[T \rightarrow T * F \bullet]\}$$

$$q_{11} = \delta(q_8,) = \{[F \rightarrow (E) \bullet]\}$$

Beachte:

Der kanonische $LR(0)$ -Automat kann auch **direkt** aus der Grammatik konstruiert werden :-)
Man benötigt die Hilfsfunktion:

$$\delta_e^*(q) = q \cup \{[B \rightarrow \bullet \gamma] \mid \exists [A \rightarrow \alpha \bullet B' \beta'] \in q, \beta \in (N \cup T)^* : B' \xrightarrow{*} B \beta\}$$

Dann definiert man:

Zustände: Mengen von Items;

Anfangszustand: $\delta_\epsilon^* \{[S' \rightarrow \bullet S]\}$

Endzustände: $\{q \mid \exists A \rightarrow \alpha \in P : [A \rightarrow \alpha \bullet] \in q\}$

Übergänge: $\delta(q, X) = \delta_\epsilon^* \{[A \rightarrow \alpha X \bullet \beta] \mid [A \rightarrow \alpha \bullet X \beta] \in q\}$

Idee zu einem Parser:

- Der Parser verwaltet ein zuverlässiges Präfix $\alpha = X_1 \dots X_m$ auf dem Keller und benutzt $LR(G)$, um Reduktionsstellen zu entdecken.
- Er kann mit einer Regel $A \rightarrow \gamma$ reduzieren, falls $[A \rightarrow \gamma \bullet]$ für α gültig ist :-)
- Damit der Automat nicht immer wieder neu über den Kellerinhalt laufen muss, kellern wir anstelle der X_i jeweils die **Zustände !!!**

Achtung:

Dieser Parser ist nur dann **deterministisch**, wenn jeder Endzustand des kanonischen $LR(0)$ -Automaten keine **Konflikte** enthält ...

... im Beispiel:

$$q_1 = \{[S' \rightarrow E \bullet], [E \rightarrow E \bullet + T]\}$$

$$q_2 = \{[E \rightarrow T \bullet], [T \rightarrow T \bullet * F]\} \quad q_9 = \{[E \rightarrow E + T \bullet], [T \rightarrow T \bullet * F]\}$$

$$q_3 = \{[T \rightarrow F \bullet]\} \quad q_{10} = \{[T \rightarrow T * F \bullet]\}$$

$$q_4 = \{[F \rightarrow \text{int} \bullet]\} \quad q_{11} = \{[F \rightarrow (E) \bullet]\}$$

Die Endzustände q_1, q_2, q_9 enthalten mehr als ein Item :-)
Aber wir haben ja auch noch nicht **Vorausschau** eingesetzt :-)

Die Konstruktion des $LR(0)$ -Parsers:

Zustände: $Q \cup \{f\}$ (f neu :-)

Anfangszustand: q_0

Endzustand: f

Übergänge:

Shift:	(p, a, pq)	falls	$q = \delta(p, a) \neq \emptyset$
Reduce:	$(p q_1 \dots q_m, \epsilon, pq)$	falls	$[A \rightarrow X_1 \dots X_m \bullet] \in q_m,$ $q = \delta(p, A)$
Finish:	$(q_0 p, \epsilon, f)$	falls	$[S' \rightarrow S \bullet] \in p$

wobei $LR(G) = (Q, T, \delta, q_0, F)$.

Zur Korrektheit:

Man zeigt:

Die akzeptierenden Berechnungen des $LR(0)$ -Parsers stehen in eins-zu-eins Beziehung zu denen des Shift-Reduce-Parsers $M_G^{(1)}$.

Wir folgern:

- \implies Die akzeptierte Sprache ist genau $\mathcal{L}(G)$:-)
- \implies Die Folge der Reduktionen einer akzeptierenden Berechnung für ein Wort $w \in T$ liefert eine **reverse Rechts-Ableitung** von G für w :-)

Leider ist der $LR(0)$ -Parser i.a. nicht-deterministisch :-)

Wir identifizieren zwei Gründe:

Reduce-Reduce-Konflikt:

$$[A \rightarrow \gamma \bullet], [A' \rightarrow \gamma' \bullet] \in q \text{ mit } A \neq A' \vee \gamma \neq \gamma'$$

Shift-Reduce-Konflikt:

$$[A \rightarrow \gamma \bullet], [A' \rightarrow \alpha \bullet a \beta] \in q \text{ mit } a \in T \\ \text{für einen Zustand } q \in Q.$$

Solche Zustände nennen wir **ungeeignet**.

Idee:

Benutze k -Vorausschau, um Konflikte zu lösen.

Wir definieren:

Die reduzierte kontextfreie Grammatik G heißt $LR(k)$ -Grammatik, falls für $\text{First}_k(w) = \text{First}_k(x)$ aus:

$$\left. \begin{array}{l} S \xrightarrow{*}_R \alpha A w \rightarrow \alpha \beta w \\ S \xrightarrow{*}_R \alpha' A' w' \rightarrow \alpha \beta x \end{array} \right\} \text{folgt: } \alpha = \alpha' \wedge A = A' \wedge w' = x$$

Beispiele:

$$(1) \quad S \rightarrow A \mid B \quad A \rightarrow a A b \mid 0 \quad B \rightarrow a B b b \mid 1$$

... ist nicht $LL(k)$ für jedes k — aber $LR(0)$:

Sei $S \xrightarrow{*}_R \alpha X w \rightarrow \alpha \beta w$. Dann ist $\alpha \underline{\beta}$ von einer der Formen:

$$\underline{A}, \underline{B}, a^n \underline{a A b}, a^n \underline{a B b b}, a^n \underline{0}, a^n \underline{1} \quad (n \geq 0)$$

$$(2) \quad S \rightarrow a A c \quad A \rightarrow A b b \mid b$$

... ist ebenfalls $LR(0)$:

Sei $S \xrightarrow{*}_R \alpha X w \rightarrow \alpha \beta w$. Dann ist $\alpha \underline{\beta}$ von einer der Formen:

$$a \underline{b}, a \underline{A b b}, a \underline{A c}$$

$$(3) \quad S \rightarrow a A c \quad A \rightarrow b b A \mid b \quad \dots \text{ ist nicht } LR(0), \text{ aber } LR(1) :$$

Für $S \xrightarrow{*}_R \alpha X w \rightarrow \alpha \beta w$ mit $\{y\} = \text{First}_k(w)$ ist $\alpha \underline{\beta} y$ von einer der Formen:

$$a b^{2n} \underline{b} c, a b^{2n} \underline{b b A} c, \underline{a A} c$$

(4) $S \rightarrow a A c \quad A \rightarrow b A b \mid b \quad \dots$ ist nicht $LR(k)$ für jedes $k \geq 0$:

Betrachte einfach die Rechtsableitungen:

$$S \xrightarrow{*}_R a b^n A b^n c \rightarrow a b^n \underline{b} b^n c$$

In der Tat gilt:

Satz:

Die reduzierte Grammatik G ist genau dann $LR(0)$ wenn der kanonische $LR(0)$ -Automat $LR(G)$ keine ungeeigneten Zustände enthält.

Beweis:

Enthalte G einen ungeeigneten Zustand q .

Fall 1: $[A \rightarrow \gamma \bullet], [A' \rightarrow \gamma' \bullet] \in q$ mit $A \rightarrow \gamma \neq A' \rightarrow \gamma'$

Dann gibt es ein zuverlässiges Präfix $\alpha \gamma = \alpha' \gamma'$ mit

$$S \xrightarrow{*}_R \alpha A w \rightarrow \alpha \gamma w \quad \wedge \quad S \xrightarrow{*}_R \alpha' A' x \rightarrow \alpha' \gamma' x \\ \implies G \text{ ist nicht } LR(0) \text{ :-)}$$

Fall 2: $[A \rightarrow \gamma \bullet], [A' \rightarrow \beta \bullet a \beta'] \in q$

Dann gibt es ein zuverlässiges Präfix $\alpha \gamma = \alpha' \beta$ mit

$$S \xrightarrow{*}_R \alpha A w \rightarrow \alpha \gamma w \quad \wedge \quad S \xrightarrow{*}_R \alpha' A' x \rightarrow \alpha' \beta a \beta' x$$

Fall 2: $[A \rightarrow \gamma \bullet], [A' \rightarrow \beta \bullet a \beta'] \in q$

Dann gibt es ein zuverlässiges Präfix $\alpha \gamma = \alpha' \beta$ mit

$$S \xrightarrow{*}_R \alpha A w \rightarrow \alpha \gamma w \quad \wedge \quad S \xrightarrow{*}_R \alpha' A' x \rightarrow \alpha' \beta a \beta' x$$

Ist $\beta' \in T^*$, dann ist G nicht $LR(0)$:-)

Andernfalls $\beta' \xrightarrow*_R v_1 X v_2 \rightarrow v_1 u v_2$. Damit erhalten wir:

$$\begin{aligned} S &\xrightarrow*_R \alpha' \beta a v_1 X v_2 x \rightarrow \alpha' \beta a v_1 u v_2 x \\ \implies G &\text{ ist nicht LR(0) :-)} \end{aligned}$$

Enthalte $LR(G)$ keine ungeeigneten Zustände. Betrachte:

$$S \xrightarrow*_R \alpha A w \rightarrow \alpha \gamma w \qquad S \xrightarrow*_R \alpha' A' w' \rightarrow \alpha' \gamma' x$$

Sei $\delta(q_0, \alpha \gamma) = q$. Insbesondere ist $[A \rightarrow \gamma \bullet] \in q$. **Annahme:** $(\alpha, A, w') \neq (\alpha', A', x)$.

Fall 1: $w' = x$. Dann muss q $[A' \rightarrow \gamma' \bullet]$ enthalten :-)

Fall 2: $w' \neq x$. Weitere Fallunterscheidung :-))

Sei $k > 0$.

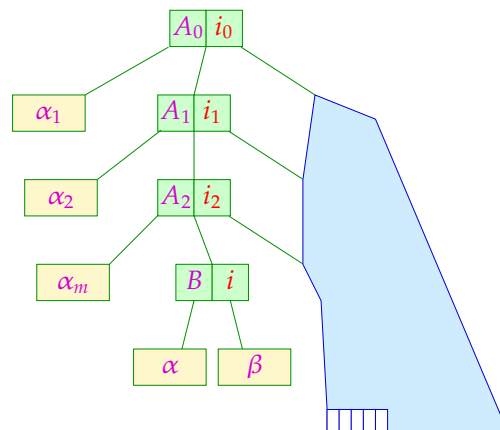
Idee: Wir staten Items mit k -Vorausschau aus :-)

Ein $LR(k)$ -Item ist dann ein Paar:

$$[B \rightarrow \alpha \bullet \beta, x], \quad x \in \text{Follow}_k(B)$$

Dieses Item ist gültig für $\gamma \alpha$ falls:

$$S \xrightarrow*_R \gamma B w \quad \text{mit} \quad \{x\} = \text{First}_k(w)$$



... wobei $\alpha_1 \dots \alpha_m = \gamma$

Die Menge der gültigen $LR(k)$ -Items für zuverlässige Präfixe berechnen wir wieder mithilfe eines endlichen Automaten :-)

Der Automat $c(G, k)$:

Zustände: $LR(k)$ -Items :-)

Anfangszustand: $[S' \rightarrow \bullet S, \epsilon]$

Endzustände: $\{[B \rightarrow \gamma \bullet, x] \mid B \rightarrow \gamma \in P, x \in \text{Follow}_k(B)\}$

Übergänge:

- (1) $([A \rightarrow \alpha \bullet X \beta, x], X, [A \rightarrow \alpha X \bullet \beta, x]), \quad X \in (N \cup T)$
- (2) $([A \rightarrow \alpha \bullet B \beta, x], \epsilon, [B \rightarrow \bullet \gamma, x']),$
 $A \rightarrow \alpha B \beta, \quad B \rightarrow \gamma \in P, x' \in \text{First}_k(\beta) \odot \{x\};$

Dieser Automat arbeitet wie $c(G)$

— verwaltet aber zusätzlich ein k -Präfix aus dem Follow_k der linken Seiten.

Den kanonischen $LR(k)$ -Automaten $LR(G, k)$ erhält man aus $c(G, k)$, indem man nach jedem Übergang beliebig viele ϵ liest und dann den Automaten **deterministisch** macht ...

Man kann ihn aber auch **direkt** aus der Grammatik konstruieren werden :-)

Wie bei $LR(0)$ benötigt man eine Hilfsfunktion:

$$\delta_\epsilon^*(q) = q \cup \{[B \rightarrow \bullet \gamma, x] \mid \exists [A \rightarrow \alpha \bullet B' \beta', x'] \in q, \\ \beta \in (N \cup T)^* : B' \xrightarrow{*} B \beta\} \wedge \\ x \in \text{First}_k(\beta \beta') \odot \{x'\}$$

Dann definiert man:

Zustände: Mengen von $LR(k)$ -Items;

Anfangszustand: $\delta_\epsilon^* \{[S' \rightarrow \bullet S, \epsilon]\}$

Endzustände: $\{q \mid \exists A \rightarrow \alpha \in P : [A \rightarrow \alpha \bullet, x] \in q\}$

Übergänge: $\delta(q, X) = \delta_\epsilon^* \{[A \rightarrow \alpha X \bullet \beta, x] \mid [A \rightarrow \alpha \bullet X \beta, x] \in q\}$

Im Beispiel:

$$q_0 = \{[S' \rightarrow \bullet E, \{\epsilon\}], \\ [E \rightarrow \bullet E + T, \{\epsilon, +\}], \\ [E \rightarrow \bullet T, \{\epsilon, +\}], \\ [T \rightarrow \bullet T * F, \{\epsilon, +, *\}], \\ [T \rightarrow \bullet F, \{\epsilon, +, *\}], \\ [F \rightarrow \bullet (E), \{\epsilon, +, *\}], \\ [F \rightarrow \bullet \text{int}, \{\epsilon, +, *\}]\}$$

$$q_1 = \delta(q_0, E) = \{[S' \rightarrow E \bullet, \{\epsilon\}], \\ [E \rightarrow E \bullet + T, \{\epsilon, +\}]\}$$

$$q_2 = \delta(q_0, T) = \{[E \rightarrow T \bullet, \{\epsilon, +\}], \\ [T \rightarrow T \bullet * F, \{\epsilon, +, *\}]\}$$

$$q_3 = \delta(q_0, F) = \{[T \rightarrow F \bullet, \{\epsilon, +, *\}]\}$$

$$q_4 = \delta(q_0, \text{int}) = \{[F \rightarrow \text{int} \bullet, \{\epsilon, +, *\}]\}$$

$$q_5 = \delta(q_0, () = \{[F \rightarrow (\bullet E), \{\epsilon, +, *\}], \\ [E \rightarrow \bullet E + T, \{\}, +\}], \\ [E \rightarrow \bullet T, \{\}, +\}], \\ [T \rightarrow \bullet T * F, \{\}, +, *], \\ [T \rightarrow \bullet F, \{\}, +, *], \\ [F \rightarrow \bullet (E), \{\}, +, *], \\ [F \rightarrow \bullet \text{int}, \{\}, +, *]\}$$

$$q'_5 = \delta(q_5, () = \{[F \rightarrow (\bullet E), \{\}, +, *], \\ [E \rightarrow \bullet E + T, \{\}, +\}], \\ [E \rightarrow \bullet T, \{\}, +\}], \\ [T \rightarrow \bullet T * F, \{\}, +, *], \\ [T \rightarrow \bullet F, \{\}, +, *], \\ [F \rightarrow \bullet (E), \{\}, +, *], \\ [F \rightarrow \bullet \text{int}, \{\}, +, *]\}$$

$$q_6 = \delta(q_1, +) = \{[E \rightarrow E + \bullet T, \{\epsilon, +\}], \\ [T \rightarrow \bullet T * F, \{\epsilon, +, *\}], \\ [T \rightarrow \bullet F, \{\epsilon, +, *\}], \\ [F \rightarrow \bullet (E), \{\epsilon, +, *\}], \\ [F \rightarrow \bullet \text{int}, \{\epsilon, +, *\}]\}$$

$$q_7 = \delta(q_2, *) = \{[T \rightarrow T * \bullet F, \{\epsilon, +, *\}], \\ [F \rightarrow \bullet (E), \{\epsilon, +, *\}], \\ [F \rightarrow \bullet \text{int}, \{\epsilon, +, *\}]\}$$

$$q_8 = \delta(q_5, E) = \{[F \rightarrow (E \bullet), \{\epsilon, +, *\}], \\ [E \rightarrow E \bullet + T, \{\}, +\}]\}$$

$$q_9 = \delta(q_6, T) = \{[E \rightarrow E + T \bullet, \{\epsilon, +\}], \\ [T \rightarrow T \bullet * F, \{\epsilon, +, *\}]\}$$

$$q_{10} = \delta(q_7, F) = \{[T \rightarrow T * F \bullet, \{\epsilon, +, *\}]\}$$

$$q_{11} = \delta(q_8,) = \{[F \rightarrow (E) \bullet, \{\epsilon, +, *\}]\}$$

$$q'_2 = \delta(q'_5, T) = \{[E \rightarrow T \bullet, \{ \}, +], [T \rightarrow T \bullet * F, \{ \}, +, *]\}$$

$$q'_7 = \delta(q_9, *) = \{[T \rightarrow T * \bullet F, \{ \}, +, *], [F \rightarrow \bullet (E), \{ \}, +, *], [F \rightarrow \bullet \text{int}, \{ \}, +, *]\}$$

$$q'_3 = \delta(q'_5, F) = \{[F \rightarrow F \bullet, \{ \}, +, *]\}$$

$$q'_8 = \delta(q'_5, E) = \{[F \rightarrow (E \bullet), \{ \}, +, *], [E \rightarrow E \bullet + T, \{ \}, +]\}$$

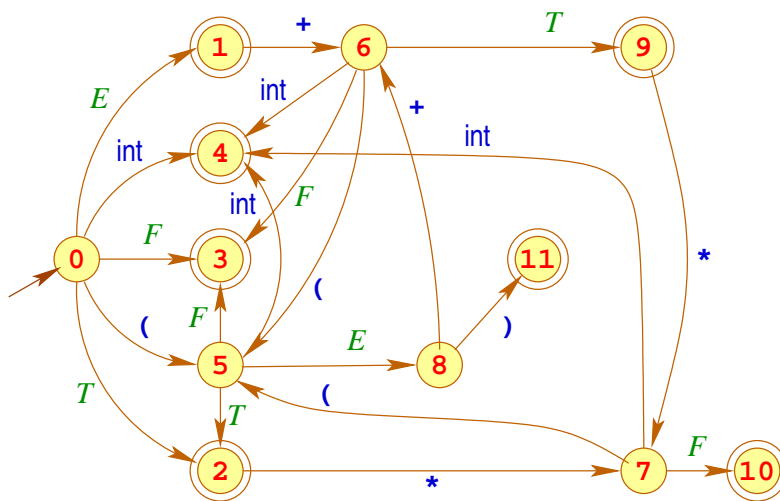
$$q'_4 = \delta(q'_5, \text{int}) = \{[F \rightarrow \text{int} \bullet, \{ \}, +, *]\}$$

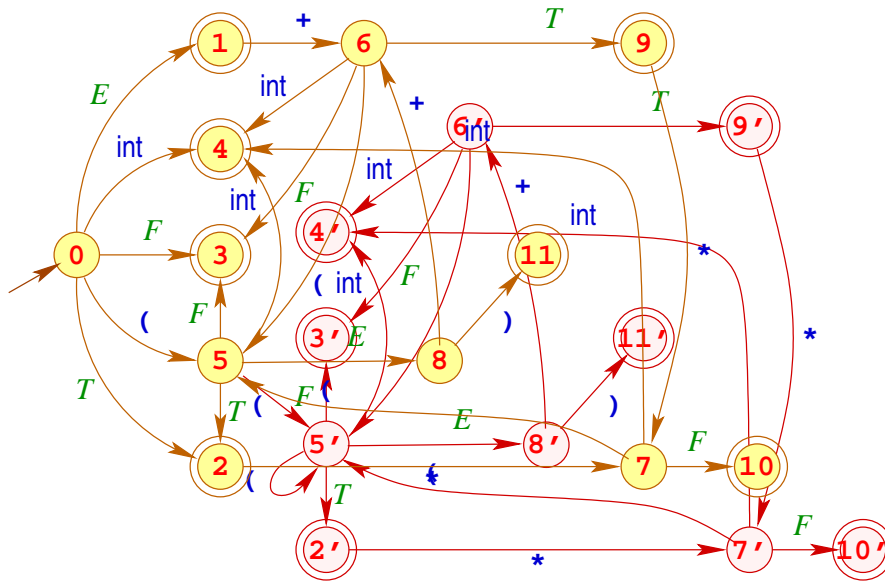
$$q'_6 = \delta(q_8, +) = \{[E \rightarrow E + \bullet T, \{ \}, +], [T \rightarrow \bullet T * F, \{ \}, +, *], [T \rightarrow \bullet F, \{ \}, +, *], [F \rightarrow \bullet (E), \{ \}, +, *], [F \rightarrow \bullet \text{int}, \{ \}, +, *]\}$$

$$q'_9 = \delta(q'_6, T) = \{[E \rightarrow E + T \bullet, \{ \}, +], [T \rightarrow T \bullet * F, \{ \}, +, *]\}$$

$$q'_{10} = \delta(q'_7, F) = \{[T \rightarrow T * F \bullet, \{ \}, +, *]\}$$

$$q'_{11} = \delta(q'_8,) = \{[F \rightarrow (E) \bullet, \{ \}, +, *]\}$$





Diskussion:

- Im Beispiel hat sich die Anzahl der Zustände fast verdoppelt :-)
- Es kann noch schlimmer kommen :-)
- Die Konflikte in den Zuständen q_1, q_2, q_9 sind nun aufgelöst ...
Z.B. haben wir für:

$$q_9 = \{ [E \rightarrow E + T \bullet, \{\epsilon, +\}], [T \rightarrow T \bullet * F, \{\epsilon, +, *\}] \}$$

mit:

$$\{\epsilon, +\} \cap (\text{First}_1(*F) \odot \{\epsilon, +, *\}) = \{\epsilon, +\} \cap \{*\} = \emptyset$$

Allgemein:

Wir identifizieren zwei Konflikte:

Reduce-Reduce-Konflikt:

$$[A \rightarrow \gamma \bullet, x], [A' \rightarrow \gamma' \bullet, x] \in q \text{ mit } A \neq A' \vee \gamma \neq \gamma'$$

Shift-Reduce-Konflikt:

$$[A \rightarrow \gamma \bullet, x], [A' \rightarrow \alpha \bullet a \beta, y] \in q \text{ mit } a \in T \text{ und}$$

$$x \in \{a\} \odot \text{First}_k(\beta) \odot \{y\}.$$

für einen Zustand $q \in Q$.

Solche Zustände nennen wir jetzt $LR(k)$ -ungeeignet :-)

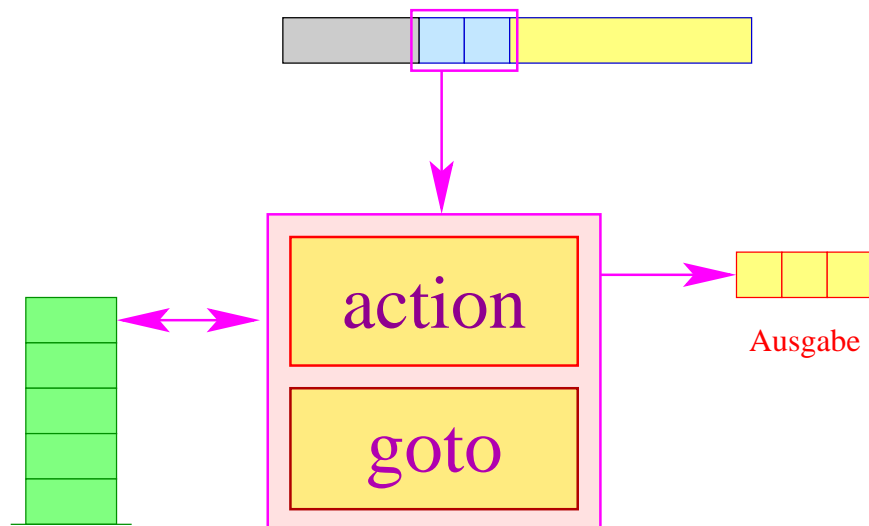
Satz

Eine reduzierte kontextfreie Grammatik G ist genau dann $LR(k)$ wenn der kanonische $LR(k)$ -Automat $LR(G, k)$ keine $LR(k)$ -ungeeigneten Zustände besitzt.

Diskussion:

- Unser Beispiel ist offenbar $LR(1)$:-)
- Im Allgemeinen hat der kanonische $LR(k)$ -Automat sehr viel mehr Zustände als $LR(G) = LR(G, 0)$:-)
- Man betrachtet darum i.a. **Teilklassen** von $LR(k)$ -Grammatiken, bei denen man nur $LR(G)$ benutzt ...
- Zur Konflikt-Auflösung ordnet man den Items in den Zuständen Vorausschau-Mengen zu:
 - (1) Die Zuordnung ist unabhängig vom Zustand \implies Simple $LR(k)$
 - (2) Die Zuordnung hängt vom Zustand ab \implies $LALR(k)$

Der $LR(k)$ -Parser:



Erläuterung:

- Die goto-Tabelle kodiert die Zustandsübergänge:

$$\text{goto}[q, X] = \delta(q, X) \in Q$$

- Die action-Tabelle beschreibt für jeden Zustand q und möglichen Look-ahead w die erforderliche Aktion.

Diese sind:

shift // Shift-Operation
reduce ($A \rightarrow \gamma$) // Reduktion mit Ausgabe
error // Fehler

... im Beispiel:

$E \rightarrow E + T^0 \mid T^1$
 $T \rightarrow T * F^0 \mid F^1$
 $F \rightarrow (E)^0 \mid \text{int}^1$

action	ϵ	int	()	+	*
q_1	$S', 0$					s
q_2	$E, 1$					s
q'_2				$E, 1$		s
q_3	$T, 1$				$T, 1$	$T, 1$
q'_3				$T, 1$	$T, 1$	$T, 1$
q_4	$F, 1$				$F, 1$	$F, 1$
q'_4				$F, 1$	$F, 1$	$F, 1$
q_9	$E, 0$				$E, 0$	s
q'_9				$E, 0$	$E, 0$	s
q_{10}	$T, 0$				$T, 0$	$T, 0$
q'_{10}				$T, 0$	$T, 0$	$T, 0$
q_{11}	$F, 0$				$F, 0$	$F, 0$
q'_{11}				$F, 0$	$F, 0$	$F, 0$

2.7 Spezielle Bottom-up-Verfahren mit $LR(G)$

Idee 1: Benutze $Follow_k$ -Mengen zur Konflikt-Lösung ...

Reduce-Reduce-Konflikt:

Falls für $[A \rightarrow \gamma \bullet], [A' \rightarrow \gamma' \bullet] \in q$ mit $A \neq A' \vee \gamma \neq \gamma'$,

$$Follow_k(A) \cap Follow_k(A') \neq \emptyset$$

Shift-Reduce-Konflikt:

Falls für $[A \rightarrow \gamma \bullet], [A' \rightarrow \alpha \bullet a \beta] \in q$ mit $a \in T$,

$$Follow_k(A) \cap (\{a\} \odot First_k(\beta) \odot Follow_k(A')) \neq \emptyset$$

für einen Zustand $q \in Q$.

Dann nennen wir den Zustand q $SLR(k)$ -ungeeignet :-)

Die reduzierte Grammatik G nennen wir $SLR(k)$ (simple $LR(k)$:-), falls der kanonische $LR(0)$ -Automat $LR(G)$ keine $SLR(k)$ -ungeeigneten Zustände enthält :-)

... im Beispiel:

Bei unserer Beispiel-Grammatik treten Konflikte möglicherweise in den Zuständen q_1, q_2, q_9 auf:

$$q_1 = \{[S' \rightarrow E \bullet], \quad Follow_1(S') \cap \{+\} \odot \{\dots\} = \{\epsilon\} \cap \{+\} \\ [E \rightarrow E \bullet + T]\} \quad = \emptyset$$

$$q_2 = \{[E \rightarrow T \bullet], \quad Follow_1(E) \cap \{*\} \odot \{\dots\} = \{\epsilon, +,)\} \cap \{*\} \\ [T \rightarrow T \bullet * F]\} \quad = \emptyset$$

$$q_9 = \{[E \rightarrow E + T \bullet], \quad Follow_1(E) \cap \{*\} \odot \{\dots\} = \{\epsilon, +,)\} \cap \{*\} \\ [T \rightarrow T \bullet * F]\} \quad = \emptyset$$

Idee 2: Berechne für jeden Zustand q Follow-Mengen :-)

Für $[A \rightarrow \alpha \bullet \beta] \in q$ definieren wir:

$$\begin{aligned} \Lambda_k(q, [A \rightarrow \alpha \bullet \beta]) &= \{ \text{First}_k(w) \mid S' \xrightarrow{*}_R \gamma A w \wedge \\ &\quad \delta(q_0, \gamma \alpha) = q \} \\ // &\subseteq \text{Follow}_k(A) \end{aligned}$$

Reduce-Reduce-Konflikt:

$[A \rightarrow \gamma \bullet], [A' \rightarrow \gamma' \bullet] \in q$ mit $A \neq A' \vee \gamma \neq \gamma'$ wobei:

$$\Lambda_k(q, [A \rightarrow \gamma \bullet]) \cap \Lambda_k(q, [A' \rightarrow \gamma' \bullet]) \neq \emptyset$$

Shift-Reduce-Konflikt:

$[A \rightarrow \gamma \bullet], [A' \rightarrow \alpha \bullet a \beta] \in q$ mit $a \in T$ wobei:

$$\Lambda_k(q, [A \rightarrow \gamma \bullet]) \cap (\{a\} \odot \text{First}_k(\beta) \odot \Lambda_k(q, [A' \rightarrow \alpha \bullet a \beta])) \neq \emptyset$$

Solche Zustände nennen wir jetzt **LALR(k)-ungeeignet :-)**

Die reduzierte Grammatik G nennen wir **LALR(k)**, falls der kanonische **LR(0)**-Automat $LR(G)$ keine **LALR(k)**-ungeeigneten Zustände enthält :-)

Bevor wir Beispiele betrachten, überlegen wir erst, wie die Mengen $\Lambda_k(q, [A \rightarrow \alpha \bullet \beta])$ berechnet werden können :-)

Idee: Stelle ein Ungleichungssystem auf !!!

$$\begin{aligned} \Lambda_k(q_0, [S' \rightarrow \bullet S]) &\supseteq \{\epsilon\} \\ \Lambda_k(q, [A \rightarrow \alpha X \bullet \beta]) &\supseteq \Lambda_k(p, [A \rightarrow \alpha \bullet X \beta]) && \text{falls } \delta(p, X) = q \\ \Lambda_k(q, [A \rightarrow \bullet \gamma]) &\supseteq \text{First}_k(\beta) \odot \Lambda_k(q, [B \rightarrow \alpha \bullet A \beta]) && \text{falls } [B \rightarrow \alpha \bullet A \beta] \in q \end{aligned}$$

Beispiel:

$$\begin{aligned} S &\rightarrow A b B \mid B \\ A &\rightarrow a \mid b B \\ B &\rightarrow A \end{aligned}$$

Der kanonische **LR(0)**-Automat hat dann die folgenden Zustände:

$$\begin{aligned}
q_0 = & \{[S' \rightarrow \bullet S], & q_2 = \delta(q_0, a) = & \{[A \rightarrow a \bullet]\} \\
& \{[S \rightarrow \bullet A b B], & & \\
& \{[A \rightarrow \bullet a], & q_3 = \delta(q_0, b) = & \{[A \rightarrow b \bullet B], \\
& \{[A \rightarrow \bullet b B], & & \{[B \rightarrow \bullet A], \\
& \{[S \rightarrow \bullet B], & & \{[A \rightarrow \bullet a], \\
& \{[B \rightarrow \bullet A]\} & & \{[A \rightarrow \bullet b B]\}
\end{aligned}$$

$$q_1 = \delta(q_0, S) = \{[S' \rightarrow S \bullet]\} \qquad q_4 = \delta(q_0, B) = \{[S \rightarrow B \bullet]\}$$

$$q_5 = \delta(q_0, A) = \{[S \rightarrow A \bullet b B], \qquad q_8 = \delta(q_5, b) = \{[S \rightarrow A b \bullet B], \\
\{[B \rightarrow A \bullet]\} \qquad \qquad \qquad \{[B \rightarrow \bullet A],$$

$$q_6 = \delta(q_3, A) = \{[B \rightarrow A \bullet]\} \qquad \{[A \rightarrow \bullet a], \\
\{[A \rightarrow \bullet b B]\}$$

$$q_7 = \delta(q_3, B) = \{[A \rightarrow b B \bullet]\} \qquad q_9 = \delta(q_8, B) = \{[S \rightarrow A b B \bullet]\}$$

Shift-Reduce-Konflikt: $q_5 = \{[S \rightarrow A \bullet b B], \\ \{[B \rightarrow A \bullet]\}$

Dabei ist: $\text{Follow}_1(B) \cap \{b\} \odot \{\dots\} = \{\epsilon, b\} \cap \{b\} \neq \emptyset$

Ausschnitt des Ungleichungssystems:

$$\begin{aligned}
\Lambda_1(q_1, [B \rightarrow A \bullet]) &\supseteq \Lambda_1(q_0, [B \rightarrow \bullet A]) & \Lambda_1(q_0, [B \rightarrow \bullet A]) &\supseteq \Lambda_1(q_0, [S \rightarrow \bullet B]) \\
& & \Lambda_1(q_0, [S \rightarrow \bullet B]) &\supseteq \Lambda_1(q_0, [S' \rightarrow \bullet S]) \\
& & \Lambda_1(q_0, [S' \rightarrow \bullet S]) &\supseteq \{\epsilon\}
\end{aligned}$$

Folglich: $\Lambda_1(q_5, [B \rightarrow A \bullet]) = \{\epsilon\}$

Diskussion:

- Das Beispiel ist folglich **nicht** *SLR(1)*, aber *LALR(1)* :-)

- Das Beispiel ist nicht so an den Haaren herbei gezogen, wie es scheint ...
- Umbenennung: $A \Rightarrow L \quad B \Rightarrow R \quad a \Rightarrow \text{id} \quad b \Rightarrow * / =$ liefert:

$$\begin{aligned} S &\rightarrow L = R \mid R \\ L &\rightarrow \text{id} \mid * R \\ R &\rightarrow L \end{aligned}$$

... d.h. ein Fragment der Grammatik für C-Ausdrücke :-)

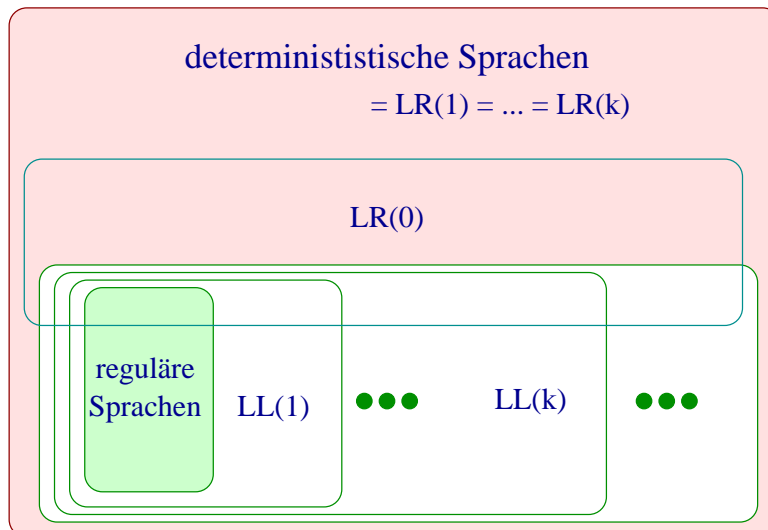
Für $k = 1$ lassen sich die Mengen $\Lambda_k(q, [A \rightarrow \alpha \bullet \beta])$ wieder effizient berechnen :-)

Das verbesserte Ungleichungssystem:

$$\begin{aligned} \Lambda_1(q_0, [S' \rightarrow \bullet S]) &\supseteq \{\epsilon\} \\ \Lambda_1(q, [A \rightarrow \alpha X \bullet \beta]) &\supseteq \Lambda_1(p, [A \rightarrow \alpha \bullet X \beta]) && \text{falls } \delta(p, X) = q \\ \Lambda_1(q, [A \rightarrow \bullet \gamma]) &\supseteq F_\epsilon(X_j) && \text{falls } [B \rightarrow \alpha \bullet A X_1 \dots X_m] \in q \\ &&& \text{und } \text{empty}(X_1) \wedge \dots \wedge \text{empty}(X_{j-1}) \\ \Lambda_1(q, [A \rightarrow \bullet \gamma]) &\supseteq \Lambda_1(q, [B \rightarrow \alpha \bullet A X_1 \dots X_m]) && \text{falls } [B \rightarrow \alpha \bullet A X_1 \dots X_m] \in q \\ &&& \text{und } \text{empty}(X_1) \wedge \dots \wedge \text{empty}(X_m) \end{aligned}$$

\implies ein reines Vereinigungsproblem :-))

Übersicht über die Sprachklassen:



Diskussion:

- Alle kontextfreien Sprachen, die sich mit einem deterministischen Kellerautomaten parsen lassen, können durch eine **LR(1)**-Grammatik beschrieben werden.
- Durch **LR(0)**-Grammatiken lassen sich alle **präfixfreien** deterministisch kontextfreien Sprachen beschreiben :-)
- Die Sprachklassen zu **LL(k)**-Grammatiken bilden dagegen eine **Hierarchie** innerhalb der deterministisch kontextfreien Sprachen.
- Da zu jeder **LL(k)**-Grammatik eine **äquivalente starke LL(k)**-Grammatik konstruiert werden kann, sind letztere nicht in der Übersicht vermerkt.

3 Semantische Analyse

- Lexikalisch und syntaktisch korrekte Programme können trotzdem fehlerhaft sein ;-(
- Einige von diesen Fehlern werden bereits durch die Sprachdefinition ausgeschlossen und müssen vom Compiler überprüft werden :-)
- Weitere Analysen sind erforderlich, um:
 - Bezeichner eindeutig zu machen;
 - die Typen von Variablen zu ermitteln;
 - Möglichkeiten zur Programm-Optimierung zu finden.

3.1 Symbol-Tabellen

Beispiel:

```
void foo() {  
    int A;  
    void fee() {  
        double A;  
        A = 0.5;  
        write(A);  
    }  
    A = 2;  
    fee();  
    write(A);  
}
```

Diskussion:

- Innerhalb des Rumpfs von `fee` wird die Definition von `A` durch die **lokale Definition** verdeckt :-)
- Für die Code-Erzeugung benötigen wir für jede Benutzung eines Bezeichners die zugehörige **Definitionsstelle**.
- **Statische Bindung** bedeutet, dass die Definition eines Namens `A` an allen Programmpunkten innerhalb ihres gesamten Blocks **gültig** ist.
- **Sichtbar** ist sie aber nur außerhalb derjenigen Teilbereiche, in denen eine weitere Definition von `A` gültig ist :-)

... im Beispiel:

```
void foo() {  
    int A;  
    void fee() {  
        double A;  
        A = 0.5;  
        write(A);  
    }  
    A = 2;  
    fee();  
    write(A);  
}
```

Kompliziertere Regeln der Sichtbarkeit gibt es in objektorientierten Programmiersprachen wie Java ...

Beispiel:

```
public class Foo {  
    protected int x = 17;  
    protected int y = 5;  
    private int z = 42;  
    public int b() { return 1; }  
}  
class Fee extends Foo {  
    protected double y = .5;  
    public int b(int a) { return a; }  
}
```

Diskussion:

- **private** Members sind nur innerhalb der aktuellen Klasse gültig :-)
- **protected** Members sind innerhalb der Klasse, in den Unterklassen sowie innerhalb des gesamten **package** gültig :-)
- Methoden **b** gleichen Namens sind stets verschieden, wenn ihre Argument-Typen verschieden sind !!!

- Bei Aufrufen einer Methode wird **dynamisch** entschieden, welche Definition gemeint ist
...

Beispiel:

```

public class Foo {
    protected int foo() { return 1; }
}
class Fee extends Foo {
    protected int foo() { return 2; }
    public int test(boolean b) {
        Foo x = (b) ? new Foo() : new Fee();
        return x.foo();
    }
}

```

Aufgabe: Finde zu jeder Benutzung eines Bezeichners die zugehörige Definition

1. Schritt: Ersetze Bezeichner durch **eindeutige** Nummern !

Input: Folge von Strings

Output: (1) Folge von Nummern
(2) Tabelle, die zu Nummern die Strings auflistet

Beispiel:

das	schwein	ist	dem	schwein	was	...
-----	---------	-----	-----	---------	-----	-----

...	das	schwein	dem	menschen	ist	wurst
-----	-----	---------	-----	----------	-----	-------

... liefert:

0	1	2	3	1	4	0	1	3	5	2	6
---	---	---	---	---	---	---	---	---	---	---	---

0	das
1	schwein
2	ist
3	dem
4	was
5	menschen
6	wurst

Implementierung 1:

Wir benutzen eine partielle Abbildung: $S : \text{String} \rightarrow \text{int}$ verwaltet :-)

Wir verwalten einen Zähler $\text{int count} = 0$; für die Anzahl der bereits gefundenen Wörter :-)

Damit definieren wir eine Funktion: $\text{int getIndex}(\text{String } w) :$

```

int getIndex(String w) {
    if (S(w) ≡ undefined) {
        S = S ⊕ {w ↦ count};
        return count++;
    } else return S(w);
}

```

Implementierung 2: Partielle Abbildungen

Ideen:

- Liste von Paaren $(w, i) \in \text{String} \times \text{int} :$
 - Einfügen: $\mathcal{O}(1)$
 - Finden: $\mathcal{O}(n) \implies$ zu teuer :-)
- balancierte Bäume :
 - Einfügen: $\mathcal{O}(\log(n))$
 - Finden: $\mathcal{O}(\log(n)) \implies$ zu teuer :-)
- Hash Tables :
 - Einfügen: $\mathcal{O}(1)$
 - Finden: $\mathcal{O}(1)$... zumindest im Mittel :-)

... im Beispiel:

- Wir legen ein Feld M von hinreichender Größe m an :-)
- Wir wählen eine Hash-Funktion $H : \text{String} \rightarrow [0, m - 1]$ mit den Eigenschaften:
 - $H(w)$ ist leicht zu berechnen :-)
 - H streut die vorkommenden Wörter gleichmäßig über $[0, m - 1]$:-)

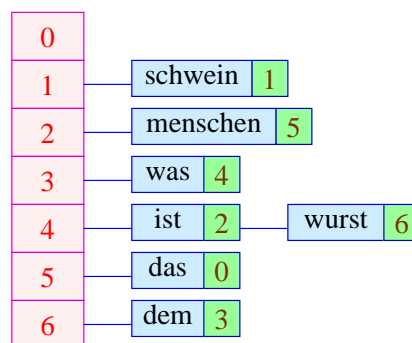
Mögliche Wahlen:

$$\begin{aligned} H_0(x_0 \dots x_{r-1}) &= (x_0 + x_{r-1}) \% m \\ H_1(x_0 \dots x_{r-1}) &= (\sum_{i=0}^{r-1} x_i \cdot p^i) \% m \\ &= (x_0 + p \cdot (x_1 + p \cdot (\dots + p \cdot x_{r-1} \dots))) \% m \end{aligned}$$

für eine Primzahl p (z.B. 31 :-)

- Das Argument-Wert-Paar (w, i) legen wir dann in $M[H(w)]$ ab :-)

Mit $m = 7$ und H_0 erhalten wir:



Um den Wert des Worts w zu finden, müssen wir w mit allen Worten x vergleichen, für die $H(w) = H(x)$:-)

2. Schritt: Symboltabellen

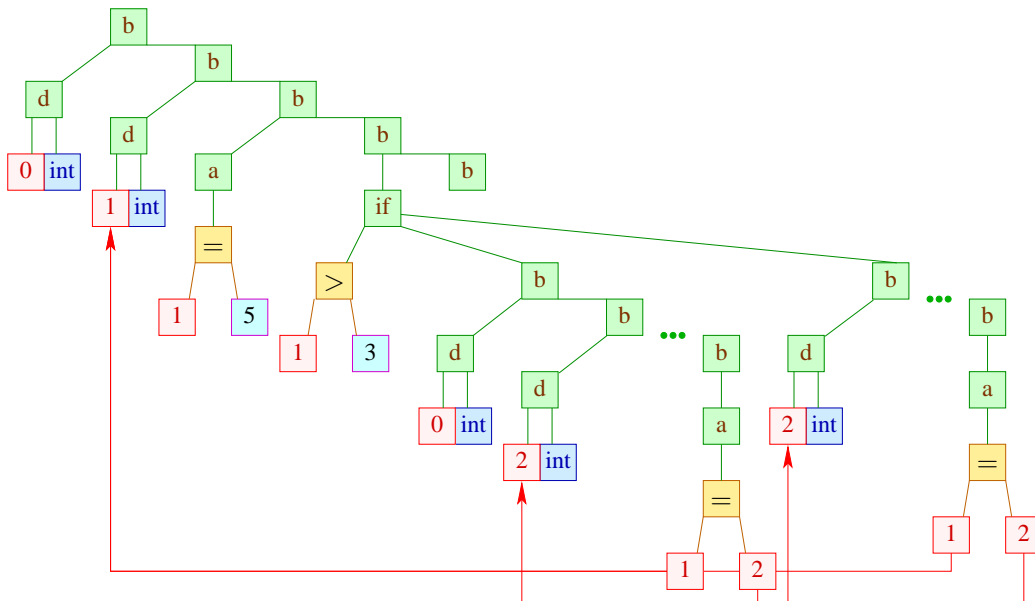
- Durchmustere den Syntaxbaum in einer geeigneten Reihenfolge, die
 - jede Definition **vor** ihren Benutzungen besucht :-)
 - die jeweils aktuell sichtbare Definition zuletzt besucht :-)
- Für jeden Bezeichner verwaltet man einen **Keller** der gültigen Definitionen.
- Trifft man bei der Durchmusterung auf eine Definition eines Bezeichners, schiebt man sie auf den Keller.
- Verlässt man den Gültigkeitsbereich, muss man sie wieder vom Keller werfen :-)
- Trifft man bei der Durchmusterung auf eine Benutzung, schlägt man die letzte Definition auf dem Keller nach ...
- Findet man keine Definition, haben wir einen Fehler gefunden :-)

Beispiel:

```
{ int a, b;
  a = 5;
  if (a > 3) {
    int a, c;
    a = 3;
    c = a + 1;
    b = c;
  }
} else {
  int c;
  c = a + 1;
  b = c;
}
```

0	a
1	b
2	c

Der zugehörige [Syntaxbaum](#) ...



Diskussion:

- Der Durchlauf ist hier einfach [links-rechts](#) DFS.
- Benutzt man eine Listen-Implementierung der Keller und eine rekursive Implementierung, kann man auf das Beseitigen der jeweils neuen Definitionen verzichten :-)
- Anstelle erst die Namen durch Nummern zu ersetzen und dann die Zuordnung von Benutzungen zu Definitionen vorzunehmen, kann man auch gleich eindeutige Nummern vergeben :-))

Achtung:

- Manche Programmiersprachen verbieten eine Mehrfach-Deklaration des selben Namens innerhalb eines Blocks :-)
- Dann muss man für jede Deklaration einen Pointer auf den Block verwalten, zu dem sie gehört.
- Gibt es eine weitere Deklaration des gleichen Namens mit dem selben Pointer, muss ein Fehler gemeldet werden :-))

Erweiterung:

- Hat man mehrere wechselseitig **rekursive Funktionsdefinitionen** in einem Block, müssen deren Namen vor Durchmustern der Rümpfe in die Tabelle eingetragen werden ...

```
fun odd 0 = false
| odd 1 = true
| odd x = even (x - 1)
and even 0 = true
| even 1 = false
| even x = odd (x - 1)
```

- Hat man eine objektorientierte Sprache mit Vererbung zwischen Klassen, sollte die übergeordnete Klasse vor der Unterklasse besucht werden :-)
- Bei Überladung muss simultan eine Typüberprüfung vorgenommen werden ...

3.2 Typ-Überprüfung

In modernen (imperativen / objektorientierten / funktionalen) Programmiersprachen besitzen Variablen und Funktionen einen **Typ**, z.B. **int**, **struct** { **int** *x*; **int** *y*; }.

Typen sind nützlich für:

- die **Speicherverwaltung**;
- die Vermeidung von **Laufzeit-Fehlern** :-)

In imperativen /objektorientierten Programmiersprachen muss der Typ bei der Deklaration spezifiziert und vom Compiler die typ-korrekte Verwendung überprüft werden :-)

Typen werden durch Typ-**Ausdrücke** beschrieben.

Die Menge T der Typausdrücke enthält:

- (1) **Basis-Typen**: **int**, **boolean**, **float**, **void** ...
- (2) **Typkonstruktoren**, angewendet auf Typen, z.B.:

- Verbunde: **struct** { t_1 *a*₁; ... t_k *a*_k; }
- Zeiger: t *
- Felder: t []

Achtung:

In **C** muss zusätzlich eine Größe spezifiziert werden;

die Variable muss dann zwischen t und [*n*] stehen :-)

- Funktionen: $t(t_1, \dots, t_k)$

Achtung:

In **C** muss die Variable zwischen t und (t_1, \dots, t_k) stehen.

In **SML** dagegen würde man diesen Typ anders herum schreiben: $t_1 * \dots * t_k \rightarrow t$:-)

Wir benutzen: (t_1, \dots, t_k) als Tupel-Typen.

- (3) **Typ-Namen**.

Typ-Namen sind nützlich:

- als Abkürzung :-)
In **C** kann man diese mithilfe von **typedef** einführen:

typedef t *x* ;

- zur Konstruktion rekursiver Typen ...

Beispiel:

```

struct list0 {
    int info;
    struct list1 * next;
};

struct list1 {
    int info;
    struct list0 * next;
};

```

Aufgabe:

Gegeben: eine Menge von Typ-Deklarationen $\Gamma = \{t_1 x_1; \dots; t_m x_m\}$
Überprüfe: Kann ein Ausdruck e mit dem Typ t versehen werden?

Beispiel:

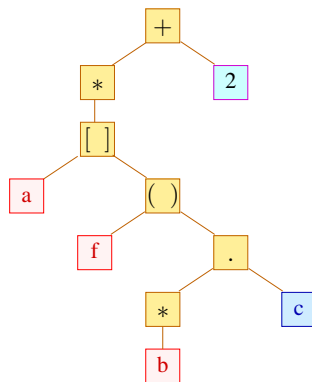
```

struct list {int info;struct list * next;};
int f(struct list * l) {return 1;};
struct {struct list * c;} * b;
int * a[11];

```

Betrachte den Ausdruck:

$*a[f(b \rightarrow c)] + 2;$



Idee:

- Traversiere den Syntaxbaum **bottom-up**.
- Für Bezeichner sagt uns Γ den richtigen Typ **;-)**
- Konstanten wie `2` oder `0.5` sehen wir den Typ direkt an **;-)**
- Die Typen für die inneren Knoten erschließen wir mithilfe von **Typ-Regeln**.

Formal betrachten wir **Aussagen** der Form:

$$\Gamma \vdash e : t$$

// (In der Typ-Umgebung Γ hat e den Typ t)

Axiome:

Const: $\Gamma \vdash c : t_c$ (t_c Typ der Konstante c)

Var: $\Gamma \vdash x : \Gamma(x)$ (x Variable)

Regeln:

$$\text{Ref: } \frac{\Gamma \vdash e : t}{\Gamma \vdash \&e : t*} \qquad \text{Deref: } \frac{\Gamma \vdash e : t*}{\Gamma \vdash *e : t}$$

$$\text{Array: } \frac{\Gamma \vdash e_1 : t* \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1[e_2] : t}$$

$$\text{Array: } \frac{\Gamma \vdash e_1 : t[] \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1[e_2] : t}$$

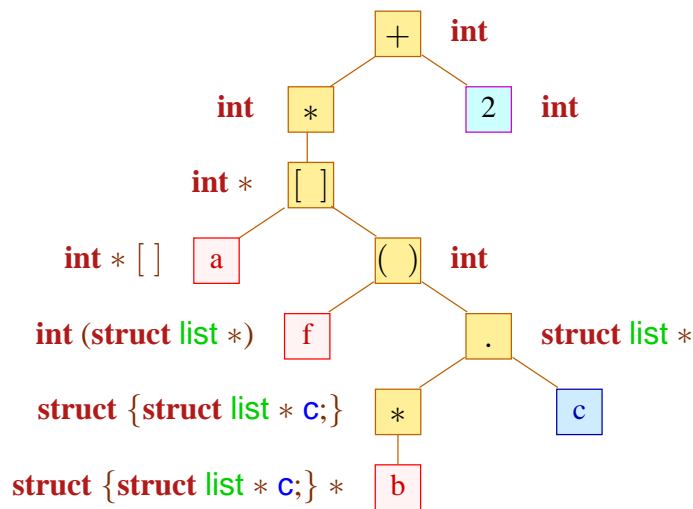
$$\text{Struct: } \frac{\Gamma \vdash e : \mathbf{struct} \{t_1 a_1; \dots t_m a_m\}}{\Gamma \vdash e.a_i : t_i}$$

$$\text{App: } \frac{\Gamma \vdash e : t(t_1, \dots, t_m) \quad \Gamma \vdash e_1 : t_1 \dots \Gamma \vdash e_m : t_m}{\Gamma \vdash e(e_1, \dots, e_m) : t}$$

$$\text{Op: } \frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}}$$

$$\text{Cast: } \frac{\Gamma \vdash e : t_1 \quad t_1 \text{ in } t_2 \text{ konvertierbar}}{\Gamma \vdash (t_2) e : t_2}$$

... im Beispiel:



Diskussion:

- Welche Regel an einem Knoten angewendet werden muss, ergibt sich aus den Typen für die bereits bearbeiteten Kinderknoten :-)
- Dazu muss die Gleichheit von Typen festgestellt werden.

Achtung:

`struct A {}` und `struct B {}` werden als verschieden betrachtet !!

Nach:

`typedef int C;`

bezeichnen `C` und `int` immer noch den gleichen Typ :-)

- ...
- ...
- Manche Operatoren wie z.B. `+` sind überladen: sie besitzen mehrere verschiedene Bedeutungen.
- Welche Bedeutung ausgewählt werden soll, entscheidet sich aufgrund der Argument-Typen. Der Operator `+` kann zum Beispiel bedeuten:
 - Addition auf `short`, `int`, `long`, `float` oder `double` :-)
 - Pointer-Arithmetik :-))
- Ist die Bedeutung ermittelt, wird (in bestimmten Fällen) für das Argument, das noch nicht den richtigen Typ hat, eine Typ-Konvertierung eingefügt.

Strukturelle Typ-Gleichheit:

Semantisch können wir zwei rekursive Typen t_1, t_2 als **gleich** betrachten, falls sie die gleiche Menge von Pfaden zulassen.

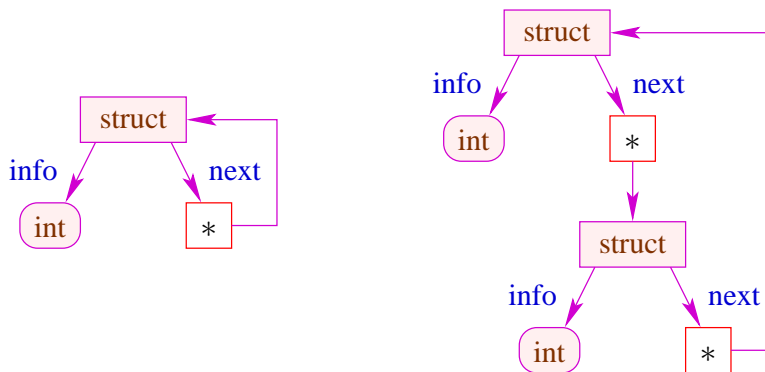
Beispiel:

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

```
struct list1 {  
    int info;  
    struct {  
        int info;  
        struct list1 * next;  
    } * next;  
}
```

Rekursive Typen können wir als **gerichtete Graphen** darstellen.

... im Beispiel:



Beobachtung:

- Hat ein Knoten mehr als einen Nachfolger, tragen die ausgehenden Kanten **unterschiedliche** Beschriftungen :-)
- Das kann man auch für Funktions-Knoten erreichen :-)
- Der Typgraph kann damit als **deterministischer endlicher Automat** aufgefasst werden, der alle Pfade durch den Typ akzeptiert :-))
- Zwei Typen können wir dann als äquivalent auffassen, wenn ihre Typgraphen, aufgefasst als **DFA**s äquivalent sind.

- Insbesondere gibt es stets einen eindeutig bestimmten **minimalen** Typgraphen für jeden Typ :-)
- Strukturelle Äquivalenz rekursiver Typen ist deshalb schnell entscheidbar !!!

Alternativer Algorithmus:

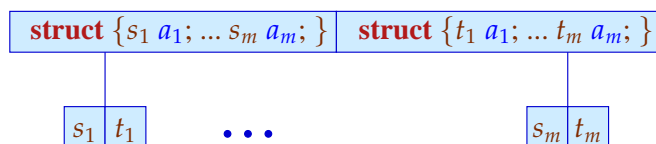
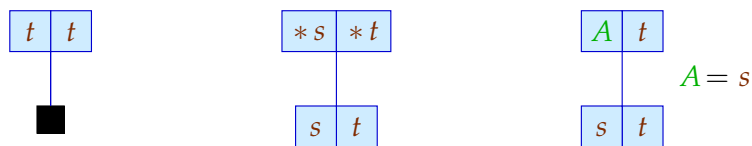
Idee:

- Verwalte Äquivalenz-Anfragen für je zwei Typausdrücke ...
- Sind die beiden Ausdrücke **syntaktisch** gleich, ist alles gut :-)
- Andernfalls reduziere die Äquivalenz-Anfrage zwischen Äquivalenz-Anfragen zwischen (hoffentlich **einfacheren** anderen Typausdrücken :-)

Nehmen wir an, rekursive Typen würden mithilfe von Typ-Gleichungen der Form:

$$A = t$$

eingeführt ...



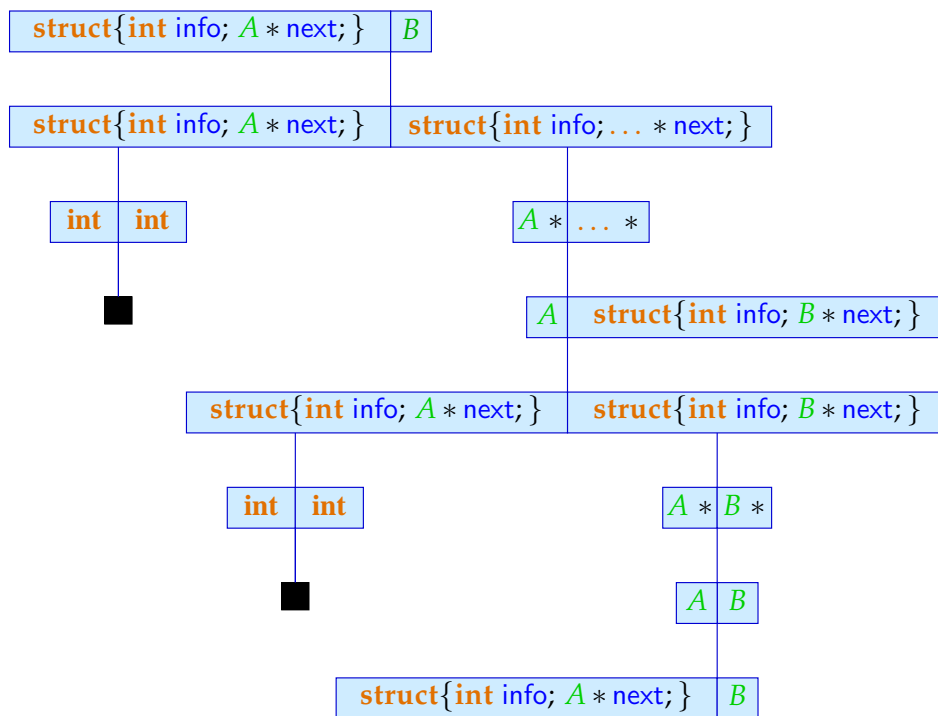
... im Beispiel:

$$\begin{aligned}
 A &= \text{struct } \{ \text{int info}; A * \text{next}; \} \\
 B &= \text{struct } \{ \text{int info}; \\
 &\quad \text{struct } \{ \text{int info}; B * \text{next}; \} * \text{next}; \}
 \end{aligned}$$

Wir fragen uns etwa, ob gilt:

$$\text{struct } \{ \text{int info}; A * \text{next}; \} = B$$

Dazu konstruieren wir:



Diskussion:

- Stoßen wir bei der Konstruktion des Beweisbaums auf eine Äquivalenz-Anfrage, auf die keine Regel anwendbar ist, gibt es einen Widerspruch !!!
- Die Konstruktion des Beweisbaums kann dazu führen, dass die gleiche Äquivalenz-Anfrage ein weiteres Mal auftritt ...
- Taucht eine Äquivalenz-Anfrage ein weiteres Mal auf, können wir hier abbrechen ;-)

⇒ die Anzahl zu betrachtender Anfragen ist endlich :-)

⇒ das Verfahren terminiert :-))

Teiltypen

- Auf den arithmetischen Basistypen **char, int, long, ...** gibt es i.a. eine reichhaltige Teiltypen-Beziehungen.
- Dabei bedeutet $t_1 \leq t_2$, dass die Menge der Werte vom Typ t_1
 - (1) eine **Teilmenge** der Werte vom Typ t_2 sind :-)
 - (2) in einen Wert vom Typ t_2 konvertiert werden können :-)
 - (3) die Anforderungen an Werte vom Typ t_2 erfüllen ...

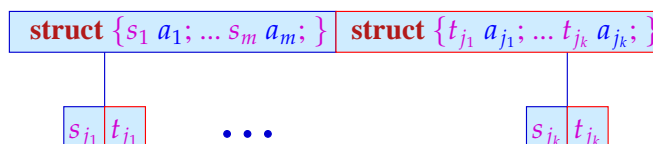
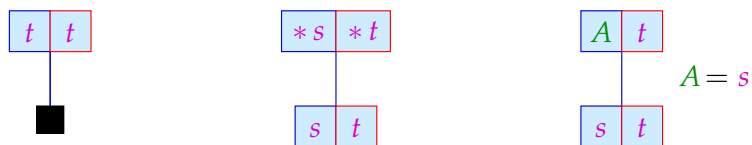


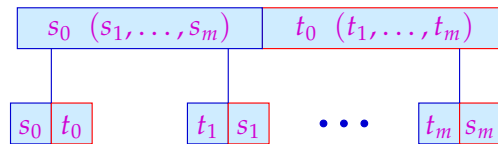
Erweitere Teiltypen-Beziehungen der Basistypen auf komplexe Typen :-)

Beispiel:

```
string extractInfo(struct { string info; } x) {
    return x.info;
}
```

- Offenkundig funktioniert **extractInfo** für alle Argument-Strukturen, die eine Komponente **string info** besitzen :-)
- Die Idee ist vergleichbar zur Anwendbarkeit auf Unterklassen (aber allgemeiner :-)
- Wann $t_1 \leq t_2$ gelten soll, beschreiben wir durch Regeln ...





Beispiele:

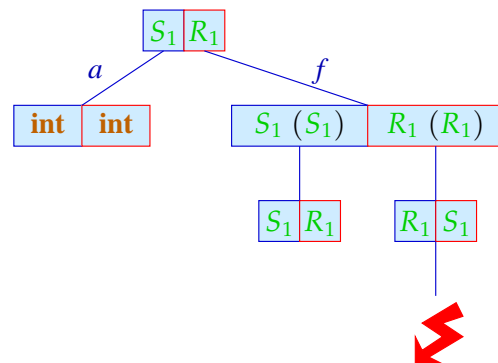
$\text{struct } \{\text{int } a; \text{int } b;\} \leq \text{struct } \{\text{float } a;\}$
 $\text{int } (\text{int}) \not\leq \text{float } (\text{float})$

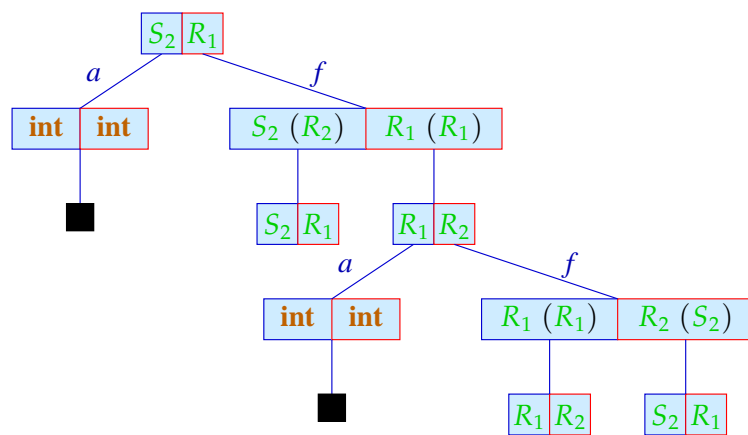
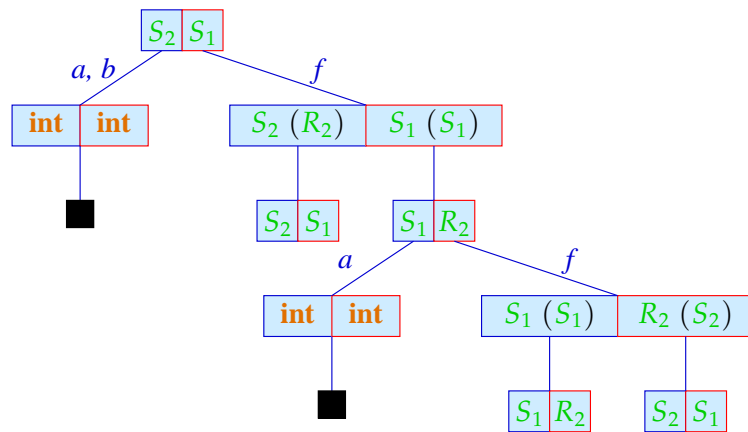
Achtung:

- Bei den Argumenten dreht sich die Anordnung der Typen gerade um !!!
- Diese Regeln können wir direkt benutzen, um auch für **rekursive** Typen die Teiltyp-Relation zu entscheiden :-)

Beispiel:

$R_1 = \text{struct } \{\text{int } a; R_1 (R_1) f;\}$
 $S_1 = \text{struct } \{\text{int } a; \text{int } b; S_1 (S_1) f;\}$
 $R_2 = \text{struct } \{\text{int } a; R_2 (S_2) f;\}$
 $S_2 = \text{struct } \{\text{int } a; \text{int } b; S_2 (R_2) f;\}$





Diskussion:

- Um die Beweisbäume nicht in den Himmel wachsen zu lassen, wurden einige Zwischenknoten ausgelassen :-)
- Strukturelle Teiltypen sind sehr mächtig und deshalb nicht ganz leicht zu durchschauen.
- Java verallgemeinert Strukturen zu **Objekten / Klassen**.
- Teiltyp-Beziehungen zwischen Klassen müssen **explizit deklariert** werden :-)
- Durch Vererbung wird sichergestellt, dass Unterklassen über die (sichtbaren) Komponenten der Oberklasse verfügen :-))
- Überschreiben einer Komponente mit einem **spezielleren** Typ ist möglich — aber nur, wenn diese keine Methode ist :-)

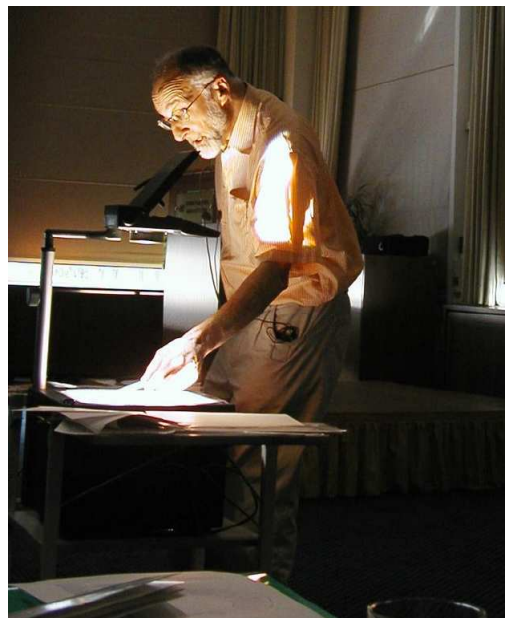
3.3 Inferieren von Typen

- Im Gegensatz zu imperativen Sprachen kann in **funktionalen** Programmiersprachen der Typ von Bezeichnern (i.a.) weggelassen werden.
- Diese werden dann **automatisch** hergeleitet :-)

Beispiel:

```
fun fac x = if x ≤ 0 then 1
           else x · fac (x - 1)
```

Dafür findet der **SML**-Compiler: `fac : int → int`



Robin (Dumbledore) Milner, Edinburgh

Idee:

J.R. Hindley, R. Milner

Stelle Axiome und Regeln auf, die den Typ eines Ausdrucks in Beziehung setzen zu den Typen seiner Teilausdrücke :-)

Der Einfachheit halber betrachten wir nur eine funktionale **Kernsprache** ...

$$\begin{aligned}
e ::= & b \mid x \mid (\square_1 e) \mid (e_1 \square_2 e_2) \\
& \mid (\text{if } e_0 \text{ then } e_1 \text{ else } e_2) \\
& \mid (e_1, \dots, e_k) \mid [] \mid (e_1 : e_2) \\
& \mid (\text{case } e_0 \text{ of } [] \rightarrow e_1; h : t \rightarrow e_2) \\
& \mid (e_1 e_2) \mid (\text{fn } (x_1, \dots, x_m) \Rightarrow e) \\
& \mid (\text{letrec } x_1 = e_1; \dots; x_n = e_n \text{ in } e_0) \\
& \mid (\text{let } x_1 = e_1; \dots; x_n = e_n \text{ in } e_0)
\end{aligned}$$

Beispiel:

$$\begin{aligned}
\text{letrec } \text{rev} &= \text{fn } x \Rightarrow r \ x \ []; \\
r &= \text{fn } x \Rightarrow \text{fn } y \Rightarrow \text{case } x \text{ of} \\
&\quad [] \rightarrow y; \\
&\quad h : t \rightarrow r \ t \ (h : y) \\
\text{in } \text{rev} &(1 : 2 : 3 : [])
\end{aligned}$$

Wir benutzen die üblichen Präzedenz-Regeln und Assoziativitäten, um hässliche Klammern zu sparen :-)

Als einzige Datenstrukturen betrachten wir **Tupel** und **List** :-))

Wir benutzen eine Syntax von Typen, die an **SML** angelehnt ist ...

$$t ::= \text{int} \mid \text{bool} \mid (t_1, \dots, t_m) \mid \text{list } t \mid t_1 \rightarrow t_2$$

Wir betrachten wieder Typ-Aussagen der Form:

$$\Gamma \vdash e : t$$

Axiome:

Const:	$\Gamma \vdash c : t_c$	(t_c Typ der Konstante c)
Nil:	$\Gamma \vdash [] : \text{list } t$	(t beliebig)
Var:	$\Gamma \vdash x : \Gamma(x)$	(x Variable)

Regeln:

$$\begin{array}{l} \text{Op:} \quad \frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}} \\ \text{If:} \quad \frac{\Gamma \vdash e_0 : \mathbf{bool} \quad \Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash (\mathbf{if } e_0 \mathbf{ then } e_1 \mathbf{ else } e_2) : t} \\ \text{Tupel:} \quad \frac{\Gamma \vdash e_1 : t_1 \quad \dots \quad \Gamma \vdash e_m : t_m}{\Gamma \vdash (e_1, \dots, e_m) : (t_1, \dots, t_m)} \\ \text{App:} \quad \frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1 e_2) : t_2} \\ \text{Fun:} \quad \frac{\Gamma \oplus \{x_1 \mapsto t_1, \dots, x_m \mapsto t_m\} \vdash e : t}{\Gamma \vdash \mathbf{fn } (x_1, \dots, x_m) \Rightarrow e : (t_1, \dots, t_m) \rightarrow t} \end{array}$$

...

$$\begin{array}{l} \dots \\ \text{Cons:} \quad \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : \mathbf{list } t}{\Gamma \vdash (e_1 : e_2) : \mathbf{list } t} \\ \text{Case:} \quad \frac{\Gamma \vdash e_0 : \mathbf{list } t_1 \quad \Gamma \vdash e_1 : t \quad \Gamma \oplus \{x \mapsto t_1, y \mapsto \mathbf{list } t_1\} \vdash e_2 : t}{\Gamma \vdash (\mathbf{case } e_0 \mathbf{ of } [] \rightarrow e_1; x : y \rightarrow e_2) : t} \\ \text{Letrec:} \quad \frac{\Gamma' \vdash e_1 : t_1 \quad \dots \quad \Gamma' \vdash e_m : t_m \quad \Gamma' \vdash e_0 : t}{\Gamma \vdash (\mathbf{letrec } x_1 = e_1; \dots; x_m = e_m \mathbf{ in } e_0) : t} \end{array}$$

wobei $\Gamma' = \Gamma \oplus \{x_1 \mapsto t_1, \dots, x_m \mapsto t_m\}$

Könnten wir die Typen für alle Variablen-Vorkommen **raten**,
ließe sich mithilfe der Regeln überprüfen, dass unsere Wahl korrekt war :-)

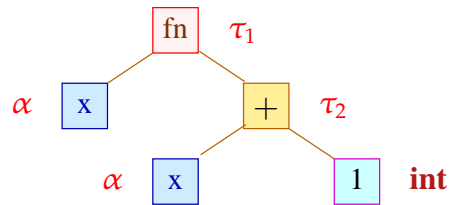
Wie raten wir die Typen der Variablen ???

Idee:

- Mache die Namen der verschiedenen Variablen eindeutig.
- Führe **Typ-Variablen** für die unbekannt Typen der Variablen und Teilausdrücke ein.
- Sammle die Gleichungen, die notwendigerweise zwischen den Typ-Variablen gelten müssen.
- Finde für diese Gleichungen Lösungen :-)

Beispiel:

fn $x \Rightarrow x + 1$



Gleichungen:

$$\tau_1 = \alpha \rightarrow \tau_2$$

$$\tau_2 = \mathbf{int}$$

$$\alpha = \mathbf{int}$$

Wir schließen:

$$\tau_1 = \mathbf{int} \rightarrow \mathbf{int}$$

Für jede Programm-Variable x und für jedes Vorkommen eines Teilausdrucks e führen wir die Typ-Variable $\alpha[x]$ bzw. $\tau[e]$ ein.

Jede Regel-Anwendung gibt dann Anlass zu einigen Gleichungen ...

Const:	$e \equiv c$	$\implies \tau[e] = \tau_c$
Nil:	$e \equiv []$	$\implies \tau[e] = \mathbf{list} \alpha \quad (\alpha \text{ neu})$
Var:	$e \equiv x$	$\implies \tau[e] = \alpha[x]$
Op:	$e \equiv e_1 + e_2$	$\implies \tau[e] = \tau[e_1] = \tau[e_2] = \mathbf{int}$
Tupel:	$e \equiv (e_1, \dots, e_m)$	$\implies \tau[e] = (\tau[e_1], \dots, \tau[e_m])$
Cons:	$e \equiv e_1 : e_2$	$\implies \tau[e] = \tau[e_2] = \mathbf{list} \tau[e_1]$
	...	

...

If:	$e \equiv \text{if } e_0 \text{ then } e_1 \text{ else } e_2$	$\implies \tau[e_0] = \text{bool}$ $\tau[e] = \tau[e_1] = \tau[e_2]$
Case:	$e \equiv \text{case } e_0 \text{ of } [] \rightarrow e_1; x : y \rightarrow e_2$	$\implies \tau[e_0] = \alpha[y] = \text{list } \alpha[x]$ $\tau[e] = \tau[e_1] = \tau[e_2]$
Fun:	$e \equiv \text{fn } (x_1, \dots, x_m) \Rightarrow e_1$	$\implies \tau[e] = (\alpha[x_1], \dots, \alpha[x_m]) \rightarrow \tau[e_1]$
App:	$e \equiv e_1 e_2$	$\implies \tau[e_1] = \tau[e_2] \rightarrow \tau[e]$
Letrec:	$e \equiv \text{letrec } x_1 = e_1; \dots; x_m = e_m \text{ in } e_0$	$\implies \alpha[x_1] = \tau[e_1] \dots$ $\alpha[x_m] = \tau[e_m]$ $\tau[e] = \tau[e_0]$

Bemerkung:

- Die möglichen Typ-Zuordnungen an Variablen und Programm-Ausdrücke erhalten wir als **Lösung** eines Gleichungssystems über Typ-Termen :-)
- Das Lösen von Systemen von Term-Gleichungen nennt man auch **Unifikation** :-)

Beispiel:

$$g(z, f(x)) = g(f(x), f(a))$$

Eine Lösung dieser Gleichung ist die **Substitution** $\{x \mapsto a, z \mapsto f(a)\}$
In dem Fall ist das offenbar die **einzige** :-)

Satz:

Jedes System von Term-Gleichungen:

$$s_i = t_i \quad i = 1, \dots, m$$

hat entweder **keine Lösung** oder eine **allgemeinste** Lösung.

Eine **allgemeinste Lösung** ist eine Substitution σ mit den Eigenschaften:

- σ ist eine Lösung, d.h. $\sigma(s_i) = \sigma(t_i)$ für alle i .
- σ ist allgemeinst, d.h. für jede andere Lösung τ gilt: $\tau = \tau' \circ \sigma$ für eine Substitution τ' :-)

Beispiele:

- (1) $f(a) = g(x)$ — hat keine Lösung :-)
- (2) $x = f(x)$ — hat ebenfalls keine Lösung ;-)
- (3) $f(x) = f(a)$ — hat genau eine Lösung:-)
- (4) $f(x) = f(g(y))$ — hat **unendlich** viele Lösungen :-)
- (5) $x_0 = f(x_1, x_1), \dots, x_{n-1} = f(x_n, x_n)$ —
hat mindestens **exponentiell große** Lösungen !!!

Bemerkungen:

- Es gibt genau eine Lösung, falls die allgemeinste Lösung keine Variablen enthält, d.h. **ground** ist :-)
- Gibt es zwei verschiedene Lösungen, dann bereits unendlich viele ;-)
- **Achtung:** Es kann mehrere allgemeinste Lösungen geben !!!

Beispiel: $x = y$

Allgemeinste Lösungen sind : $\{x \mapsto y\}$ oder $\{y \mapsto x\}$

Diese sind allerdings nicht **sehr** verschieden :-)

- Eine allgemeinste Lösung kann immer **idempotent** gewählt werden, d.h. $\sigma = \sigma \circ \sigma$.

Beispiel: $x = x \quad y = y$

Nicht idempotente Lösung: $\{x \mapsto y, y \mapsto x\}$

Idempotente Lösung: $\{x \mapsto x, y \mapsto y\}$

Berechnung einer allgemeinsten Lösung:

```
fun occurs (x,t) = case t
  of x          → true
   | f(t1,...,tk) → occurs (x,t1) ∨ ... ∨ occurs (x,tk)
   | _          → false
fun unify (s,t) θ = if θ s ≡ θ t then θ
  else case (θ s, θ t)
    of (x,t) → if occurs (x,t) then Fail
           else {x ↦ t} ∘ θ
       | (t,x) → if occurs (x,t) then Fail
           else {x ↦ t} ∘ θ
       | (f(s1,...,sk), f(t1,...,tk)) → unifyList [(s1,t1), ..., (sk,tk)] θ
       | _ → Fail
  ...
and unifyList list θ = case list
  of [] → θ
   | ((s,t)::rest) → let val θ = unify (s,t) θ
                       in if θ = Fail then Fail
                           else unifyList rest θ
                       end
```

Diskussion:

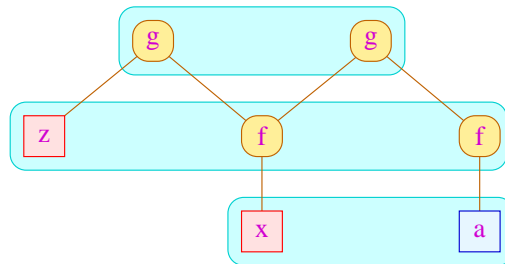
- Der Algorithmus startet mit `unifyList [(s1, t1), ..., (sm, tm)] { } ...`
- Der Algorithmus liefert sogar eine idempotente allgemeinste Lösung :-)
- Leider hat er möglicherweise exponentielle Laufzeit :-)
- Lässt sich das verbessern ???

Idee:

- Wir repräsentieren die Terme der Gleichungen als Graphen.
- Dabei identifizieren wir bereits isomorphe Teilterme ;-)
- ...

... im Beispiel:

$$g(z, f(x)) = g(f(x), f(a))$$



- ...
- Wir berechnen eine **Äquivalenz-Relation** \equiv auf den Knoten mit den folgenden Eigenschaften:
 - $s \equiv t$ für jede Gleichung unseres Gleichungssystems;
 - $s \equiv t$ nur, falls entweder s oder t eine Variable ist oder beide den gleichen Top-Konstruktor haben.
 - Falls $s \equiv t$ und $s = f(s_1, \dots, s_k), t = f(t_1, \dots, t_k)$ dann auch $s_1 \equiv t_1, \dots, s_k \equiv t_k$.
- Falls keine solche Äquivalenz-Relation existiert, ist das System unlösbar.
- Falls eine solche Äquivalenz-Relation gilt, müssen wir überprüfen, dass der Graph modulo der Äquivalenz-Relation **azyklisch** ist.
- Ist er azyklisch, können wir aus der Äquivalenzklasse jeder Variable eine **allgemeinste Lösung** ablesen ...

Implementierung:

- Wir verwalten eine **Partition** der Knoten;
- Wann immer zwei Knoten äquivalent sein sollen, vereinigen wir ihre Äquivalenzklassen und fahren mit den Söhnen entsprechend fort.
- Notwendige Operationen auf der Datenstruktur π für eine Partition:
 - **init**(Nodes) liefert eine Repräsentation für die Partition $\pi_0 = \{\{v\} \mid v \in \text{Nodes}\}$
 - **find**(π, u) liefert einen Repräsentanten der Äquivalenzklasse — der wann immer möglich keine Variable sein soll :-)
 - **union**(π, u_1, u_2) vereinigt die Äquivalenzklassen von u_1, u_2 :-)
- Der Algorithmus startet mit einer Liste

$$W = [(u_1, v_1), \dots, (u_m, v_m)]$$

der Paare von Wurzelknoten der zu unifizierenden Terme ...

```
 $\pi = \text{init}(\text{Nodes});$ 
while ( $W \neq \emptyset$ ) {
  ( $u, v$ ) = Extract( $W$ );
   $u = \text{find}(\pi, u); v = \text{find}(\pi, v);$ 
  if ( $u \neq v$ ) {
     $\pi = \text{union}(\pi, u, v);$ 
    if ( $u \notin \text{Vars} \wedge v \notin \text{Vars}$ )
      if ( $\text{label}(u) \neq \text{label}(v)$ ) return Fail
    else {
      ( $u_1, \dots, u_k$ ) = Successors( $u$ );
      ( $v_1, \dots, v_k$ ) = Successors( $v$ );
       $W = (u_1, v_1) :: \dots :: (u_k, v_k) :: W;$ 
    }
  }
}
```

Komplexität:

$\mathcal{O}(\# \text{Knoten})$ Aufrufe von **union**

$\mathcal{O}(\# \text{Kanten} + \# \text{Gleichungen})$ Aufrufe von **find**

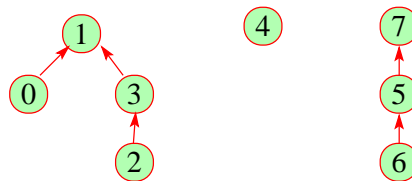
\implies Wir benötigen effiziente **Union-Find-Datenstruktur** :-)

Idee:

Repräsentiere Partition von U als gerichteten Wald:

- Zu $u \in U$ verwalten wir einen Vater-Verweis $F[u]$.
- Elemente u mit $F[u] = u$ sind Wurzeln.

Einzelne Bäume sind Äquivalenzklassen.
Ihre Wurzeln sind die Repräsentanten ...

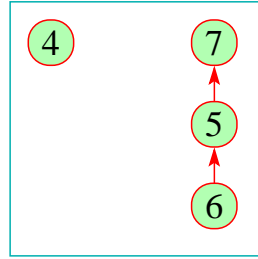
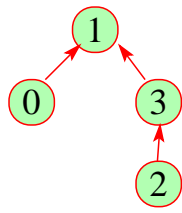


0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

1	1	3	1	4	7	5	7
---	---	---	---	---	---	---	---

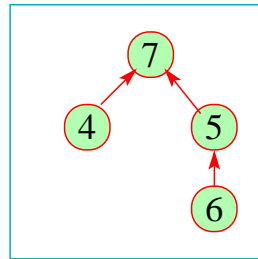
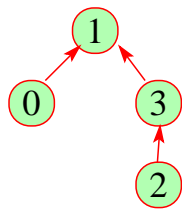
\rightarrow **find** (π, u) folgt den Vater-Verweisen :-)

\rightarrow **union** (π, u_1, u_2) hängt den Vater-Verweis eines u_i um ...



0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

1	1	3	1	4	7	5	7
---	---	---	---	---	---	---	---



0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

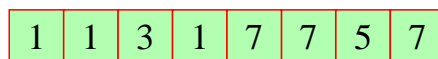
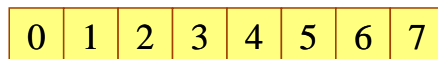
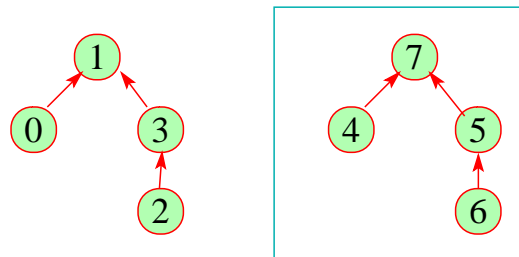
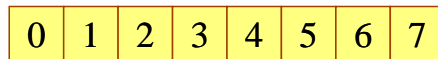
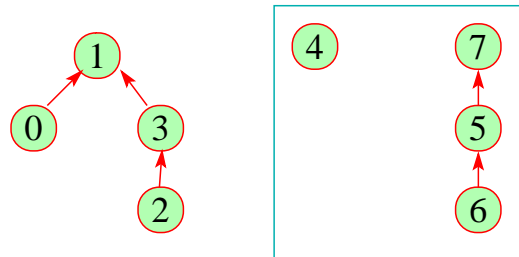
1	1	3	1	7	7	5	7
---	---	---	---	---	---	---	---

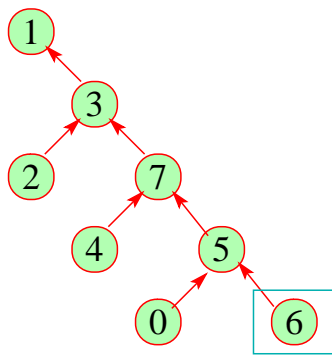
Die Kosten:

union : $\mathcal{O}(1)$:-)
 find : $\mathcal{O}(\text{depth}(\pi))$:-)

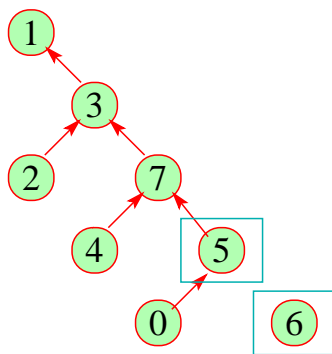
Strategie zur Vermeidung tiefer Bäume:

- Hänge den **kleineren** Baum unter den **größeren** !
- Benutze **find** , um Pfade zu komprimieren ...

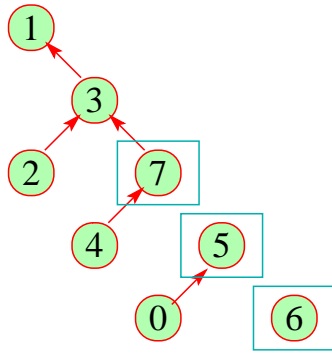




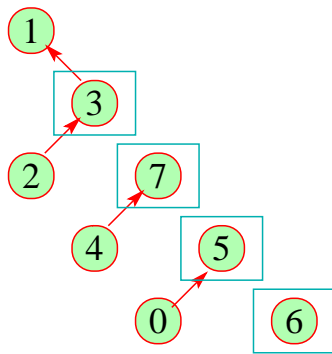
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



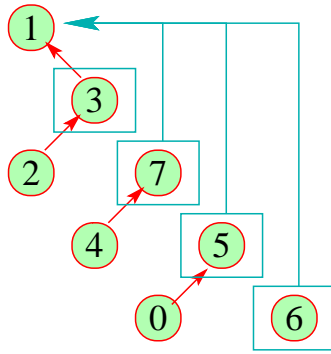
0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	7	7	5	3



0	1	2	3	4	5	6	7
5	1	3	1	1	7	1	1



Robert Endre Tarjan, Princeton

Beachte:

- Mit dieser Datenstruktur dauern n union- und m find-Operationen $\mathcal{O}(n + m \cdot \alpha(n, n))$
// α die inverse Ackermann-Funktion :-)

- Für unsere Anwendung müssen wir **union** nur so modifizieren, dass an den Wurzeln **nach Möglichkeit** keine Variablen stehen.
- Diese Modifikation vergrößert die asymptotische Laufzeit nicht :-)

Fazit:

- Wenn Typ-Gleichungen für ein Programm lösbar sind, dann gibt es eine **allgemeinste** Zuordnung von Programm-Variablen und Teil-Ausdrücken zu Typen, die alle Regeln erfüllen :-)
- Eine solche **allgemeinste Typisierung** können wir in (fast) linearer Zeit berechnen :-)

Achtung:

In der berechneten Typisierung können Typ-Variablen vorkommen !!!

Beispiel: $e \equiv \mathbf{fn} (f, x) \Rightarrow f x$

Mit $\alpha \equiv \alpha[x]$ und $\beta \equiv \tau[f x]$ finden wir:

$$\begin{aligned}\alpha[f] &= \alpha \rightarrow \beta \\ \tau[e] &= (\alpha \rightarrow \beta, \alpha) \rightarrow \beta\end{aligned}$$

Diskussion:

- Die Typ-Variablen bedeuten offenbar, dass die Funktionsdefinition für jede mögliche Instantiierung funktioniert \implies **Polymorphie**
Wir kommen darauf zurück :-)
- Das bisherige Verfahren, um Typisierungen zu berechnen, hat den Nachteil, dass es nicht **syntax-gerichtet** ist ...
- Wenn das Gleichungssystem zu einem Programm keine Lösung besitzt, erhalten wir **keine Information**, wo der Fehler stecken könnte :-)

⇒ Wir benötigen ein syntax-gerichtetes Verfahren !!!

... auch wenn es möglicherweise ineffizienter ist :-)

Der Algorithmus \mathcal{W} :

```
fun  $\mathcal{W} e (\Gamma, \theta) = \text{case } e$ 
  of  $c$            →  $(t_c, \theta)$ 
    |  $[]$         → let val  $\alpha = \text{new}()$ 
                    in  $(\text{list } \alpha, \theta)$ 
                    end
    |  $x$          →  $(\Gamma(x), \theta)$ 
    |  $(e_1, \dots, e_m)$  → let val  $(t_1, \theta) = \mathcal{W} e_1 (\Gamma, \theta)$ 
                                ...
                                val  $(t_m, \theta) = \mathcal{W} e_m (\Gamma, \theta)$ 
                                in  $((t_1, \dots, t_m), \theta)$ 
                                end
    ...
    |  $(e_1 : e_2)$  → let val  $(t_1, \theta) = \mathcal{W} e_1 (\Gamma, \theta)$ 
                    val  $(t_2, \theta) = \mathcal{W} e_2 (\Gamma, \theta)$ 
                    val  $\theta = \text{unify}(\text{list } t_1, t_2) \theta$ 
                    in  $(t_2, \theta)$ 
                    end
    |  $(e_1 e_2)$  → let val  $(t_1, \theta) = \mathcal{W} e_1 (\Gamma, \theta)$ 
                    val  $(t_2, \theta) = \mathcal{W} e_2 (\Gamma, \theta)$ 
                    val  $\alpha = \text{new}()$ 
                    val  $\theta = \text{unify}(t_1, t_2 \rightarrow \alpha) \theta$ 
                    in  $(\alpha, \theta)$ 
                    end
    ...
```

```

| (if e0 then e1 else e2) → let val (t0, θ) =  $\mathcal{W} e_0 (\Gamma, \theta)$ 
    val θ = unify (bool, t0) θ
    val (t1, θ) =  $\mathcal{W} e_1 (\Gamma, \theta)$ 
    val (t2, θ) =  $\mathcal{W} e_2 (\Gamma, \theta)$ 
    val θ = unify (t1, t2) θ
  in (t1, θ)
end

...

| (case e0 of [] → e1; (x : y) → e2)
  → let val (t0, θ) =  $\mathcal{W} e_0 (\Gamma, \theta)$ 
    val α = new()
    val θ = unify (list α, t0) θ
    val (t1, θ) =  $\mathcal{W} e_1 (\Gamma, \theta)$ 
    val (t2, θ) =  $\mathcal{W} e_2 (\Gamma \oplus \{x \mapsto \alpha, y \mapsto \text{list } \alpha\}, \theta)$ 
    val θ = unify (t1, t2) θ
  in (t1, θ)
end

...

| fn (x1, ..., xm) ⇒ e
  → let val α1 = new()
    ...
    val αm = new()
    val (t, θ) =  $\mathcal{W} e (\Gamma \oplus \{x_1 \mapsto \alpha_1, \dots, x_m \mapsto \alpha_m\}, \theta)$ 
  in ((α1, ..., αm) → t, θ)
end

...

```

```

| (letrec x1 = e1; ...; xm = em in e0)
  → let val α1 = new()
      ...
      val αm = new()
      val Γ = Γ ⊕ {x1 ↦ α1, ..., xm ↦ αm}
      val (t1, θ) = W e1 (Γ, θ)
      val θ = unify(α1, t1) θ
      ...
      val (tm, θ) = W em (Γ, θ)
      val θ = unify(αm, tm) θ
      val (t0, θ) = W e0 (Γ, θ)
  in (t0, θ)
  end

```

...

```

| (let x1 = e1; ...; xm = em in e0)
  → let val (t1, θ) = W e1 (Γ, θ)
      val Γ = Γ ⊕ {x1 ↦ t1}
      ...
      val (tm, θ) = W em (Γ, θ)
      val Γ = Γ ⊕ {xm ↦ tm}
      val (t0, θ) = W e0 (Γ, θ)
  in (t0, θ)
  end

```

...

Bemerkungen:

- Am Anfang ist $\Gamma = \emptyset$ und $\theta = \emptyset$:-)
- Der Algorithmus unifiziert nach und nach die Typ-Gleichungen :-)
- Der Algorithmus liefert bei jedem Aufruf einen Typ t zusammen mit einer Substitution θ zurück.
- Der inferierte allgemeinste Typ ergibt sich als $\theta(t)$.
- Die Hilfsfunktion `new()` liefert jeweils eine neue Typvariable :-)
- Bei jedem Aufruf von `unify()` kann die Typinferenz fehlschlagen ...

- Bei Fehlschlag sollte die Stelle, wo der Fehler auftrat gemeldet werden, die Typ-Inferenz aber mit **plausiblen Werten** fortgesetzt werden `:-}`

Beispiel:

```

let dup    = fn f ⇒ fn x ⇒ (f x);
      inc    = fn y ⇒ y + 1;
      single = fn y ⇒ y : []
in dup single (dup inc 1)
end

```

Wir finden:

```

α[dup]   = (α → β) → α → β
α[inc]   = int → int
α[single] = γ → list γ

```

- Durch die Anwendung: `dup single` erhalten wir:

```

α      = γ
β      = list γ
α[dup] = (γ → list γ) → γ → list γ

```

- Durch die Anwendung: `dup inc` erhalten wir:

```

α      = int
β      = int
α[dup] = (int → int) → int → int

```

- \implies **Typ-Fehler ???**

Idee 1: Kopiere jede Definition für jede Benutzung ...

... im Beispiel:

```
let dup = fn f => fn x => f(f x);
    inc = fn y => y + 1;
    single = fn y => y : []
in ((fn f => fn x => f(f x)) single
   ((fn f => fn x => f(f x)) inc 1)
end
```

- + Die beiden Teilausdrücke $(\text{fn } f \Rightarrow \text{fn } x \Rightarrow (f\ x))$ erhalten jeweils einen **eigenen** Typ mit **unabhängigen** Typ-Variablen :-)
- + Das expandierte Programm ist typbar :-))
- Das expandierte Programm kann **seeehr** groß werden :-((
- Typ-Checking ist nicht mehr **modular** :-(((

Idee 2: Kopiere die Typen für jede Benutzung ...

- Wir erweitern Typen zu **Typ-Schemata**:

$$t ::= \alpha \mid \text{bool} \mid \text{int} \mid (t_1, \dots, t_m) \mid \text{list } t \mid t_1 \rightarrow t_2$$
$$\sigma ::= t \mid \forall \alpha_1, \dots, \alpha_k. t$$

- **Achtung:** Der Operator \forall erscheint nur auf dem Top-Level !!!
- Typ-Schemata werden für **let**-definierte Variablen eingeführt.
- Bei deren Benutzung wird der Typ im Schema mit **frischen** Typ-Variablen instantiiert ...

Neue Regeln:

$$\text{Inst: } \frac{\Gamma(x) = \forall \alpha_1, \dots, \alpha_k. t}{\Gamma \vdash x : t[t_1/\alpha_1, \dots, t_k/\alpha_k]} \quad (t_1, \dots, t_k \text{ beliebig})$$

$$\text{Let: } \frac{\begin{array}{l} \Gamma_0 \vdash e_1 : t_1 \quad \Gamma_1 = \Gamma_0 \oplus \{x_1 \mapsto \text{close } t_1 \Gamma_0\} \\ \dots \\ \Gamma_{m-1} \vdash e_m : t_m \quad \Gamma_m = \Gamma_{m-1} \oplus \{x_m \mapsto \text{close } t_m \Gamma_{m-1}\} \\ \Gamma_m \vdash e_0 : t_0 \end{array}}{\Gamma_0 \vdash (\text{let } x_1 = e_1; \dots; x_m = e_m \text{ in } e_0) : t_0}$$

Der Aufruf `close t Γ` macht alle Typ-Variablen in `t` generisch (d.h. instantierbar), die nicht auch in `Γ` vorkommen ...

```

fun close t Γ = let
  val α1, ..., αk = free (t) \ free (Γ)
in ∀ α1, ..., αk. t
end

```

Eine Instantiierung mit `frischen` Typ-Variablen leistet die Funktion:

```

fun inst σ = let
  val ∀ α1, ..., αk. t = σ
  val β1 = new() ... val βk = new()
in t[β1/α1, ..., βk/αk]
end

```

Der Algorithmus \mathcal{W} (erweitert):

```

...
| x      →      inst (Γ(x))
| (let x1 = e1; ...; xm = em in e0)
      →      let val (t1, θ) =  $\mathcal{W}$  e1 (Γ, θ)
              val σ1 = close (θ t1) (θ Γ)
              val Γ = Γ ⊕ {x1 ↦ σ1}
              ...
              val (tm, θ) =  $\mathcal{W}$  em (Γ, θ)
              val σm = close (θ tm) (θ Γ)
              val Γ = Γ ⊕ {xm ↦ σm}
              val (t0, θ) =  $\mathcal{W}$  e0 (Γ, θ)
      in (t0, θ)
      end

```

Beispiel:

```

let dup   = fn f ⇒ fn x ⇒ (f x);
    inc   = fn y ⇒ y + 1;
    single = fn y ⇒ y : []
in dup single (dup inc 1)
end

```

Wir finden:

```

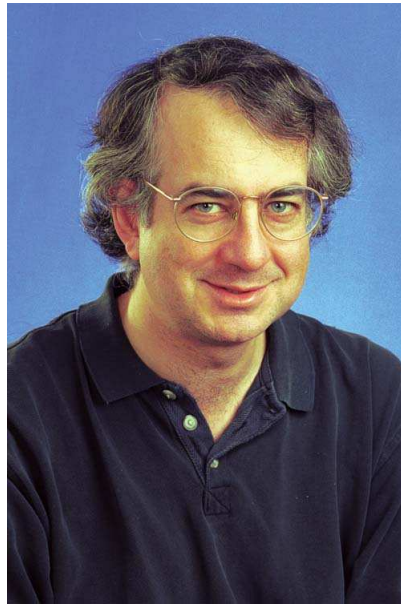
α[dup]   = ∀ α, β. (α → β) → α → β
α[inc]   = int → int
α[single] = ∀ γ. γ → list γ

```

Bemerkungen:

- Der erweiterte Algorithmus berechnet nach wie vor **allgemeinste** Typen :-)
- Instantiierung von Typ-Schemata bei jeder Benutzung ermöglicht **polymorphe Funktionen** sowie **modulare Typ-Inferenz** :-))

- Die Möglichkeit der Instantiierung erlaubt die Codierung von **DEXPTIME**-schwierigen Problemen in die Typ-Inferenz ??
... ein in der **Praxis** eher marginales Problem :-)
- Die Einführung von Typ-Schemata ist nur für **nicht-rekursive** Definitionen möglich: die Ermittlung eines allgemeinsten Typ-Schemas für rekursive Definitionen ist **nicht berechenbar !!!**



Harry Mairson, Brandeis University

Seiteneffekte

- Für ein elegantes Programmieren sind gelegentlich Variablen, deren Wert geändert werden kann, ganz **nützlich** :-)
- Darum erweitern wir unsere kleine Programmiersprache um **Referenzen**:

$$e ::= \dots \mid \mathbf{ref} \ e \mid !e \mid e_1 := e_2$$

Beispiel:

```
let count = ref 0;
new = fn () => let
    ret = !count;
    _ = count := ret + 1
    in ret
in new() + new()
```

Als neuen Typ benötigen wir:

$$t ::= \dots \text{ref } t \dots$$

Neue Regeln:

Ref:
$$\frac{\Gamma \vdash e : t}{\Gamma \vdash (\text{ref } e) : \text{ref } t}$$

Deref:
$$\frac{\Gamma \vdash e : \text{ref } t}{\Gamma \vdash (!e) : t}$$

Assign:
$$\frac{\Gamma \vdash e_1 : \text{ref } t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash (e_1 := e_2) : ()}$$

Achtung:

Diese Regeln vertragen sich nicht mit Polymorphie !!!

Beispiel:

```
let y = ref [];
_ = y := 1 : (!y);
_ = y := true : (!y)
in 1
```

Für y erhalten wir den Typ: $\forall \alpha. \text{ref } (\text{list } \alpha)$

\implies Die Typ-Inferenz liefert keinen Fehler

\implies Zur Laufzeit entsteht eine Liste mit **int** und **bool** :-)

Ausweg: Die Value-Restriction

- Generalisiere nur solche Typen, die **Werte** repräsentieren, d.h. keine **Verweise** auf Speicherstellen enthalten :-)
- Die Menge der **Value**-Typen lässt sich einfach beschreiben:

$$v ::= \text{bool} \mid \text{int} \mid \text{list } v \mid (v_1, \dots, v_m) \mid t \rightarrow t$$

... im Beispiel:

Der Typ: **ref** (**list** α) ist **kein** Value-Typ.

Darum darf er nicht generalisiert werden \implies Problem gelöst :-)



Matthias Felleisen, Northeastern University

Schlussbemerkung:

- **Polymorphie** ist ein sehr nützliches Hilfsmittel bei der Programmierung :-)
- In Form von **Templates** hält es in **Java 1.5** Einzug.
- In der Programmiersprache **Haskell** hat man Polymorphie in Richtung **bedingter** Polymorphie weiter entwickelt ...

Beispiel:

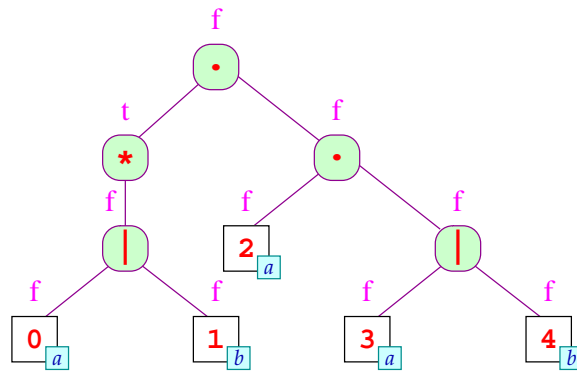
```
fun member x list = case list
  of [] → false
   | h::t → if x = h then true
             else member x t
```

`member` hat den Typ: $\alpha' \rightarrow \text{list } \alpha' \rightarrow \text{bool}$ für jedes α' mit Gleichheit !!

3.4 Attributierte Grammatiken

- Viele Berechnungen der semantischen Analyse wie während der Code-Generierung arbeiten auf dem Syntaxbaum.
- An jedem Knoten greifen sie auf bereits berechnete Informationen zu und berechnen daraus neue Informationen :-)
- Was lokal zu tun ist, hängt nur von der Sorte des Knotens !!!
- Damit die zu lesenden Werte an jedem Knoten bei jedem Lesen bereits vorliegen, müssen die Knoten des Syntaxbaums in einer bestimmten Reihenfolge durchlaufen werden ...

Beispiel: Berechnung des Prädikats $\text{empty}[r]$



Idee zur Implementierung:

- Für jeden Knoten führen wir ein Attribut empty ein.
- Die Attribute werden in einer DFS **post-order** Traversierung berechnet:
 - An einem Blatt lässt sich der Wert des Attributs unmittelbar ermitteln :-)
 - Das Attribut an einem inneren Knoten hängt darum nur von den Attributen der Nachfolger ab :-)
- Wie das Attribut lokal zu berechnen ist, ergibt sich aus dem Typ des Knotens ...

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

Andernfalls:

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

Diskussion:

- Wir benötigen einen einfachen und flexiblen Mechanismus, mit dem wir über die Attribute an einem Knoten und seinen Nachfolgern reden können.
- Der Einfachheit geben wir ihnen einen fortlaufenden Index:

$\text{empty}[0]$: das Attribut des Vater-Knotens
 $\text{empty}[i]$: das Attribut des i -ten Sohns ($i > 0$)

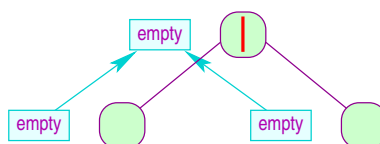
... im Beispiel:

$$\begin{aligned} \boxed{x} &: \text{empty}[0] := (x \equiv \epsilon) \\ \boxed{|} &: \text{empty}[0] := \text{empty}[1] \vee \text{empty}[2] \\ \boxed{\cdot} &: \text{empty}[0] := \text{empty}[1] \wedge \text{empty}[2] \\ \boxed{*} &: \text{empty}[0] := t \\ \boxed{?} &: \text{empty}[0] := t \end{aligned}$$

Diskussion:

- Die lokalen Berechnungen der Attributwerte müssen zu einem globalen Algorithmus zusammen gesetzt werden :-)
- Dazu benötigen wir:
 - (1) eine Besuchsreihenfolge der Knoten des Baums;
 - (2) lokale Berechnungsreihenfolgen ...
- Die Auswertungsstrategie sollte aber mit den **Attribut-Abhängigkeiten** kompatibel sein :-)

... im Beispiel:



Achtung:

- Zur Ermittlung einer Auswertungsstrategie reicht es nicht, sich die **lokalen** Attribut-Abhängigkeiten anzusehen.
- Es kommt auch darauf an, wie sie sich **global** zu einem Abhängigkeitsgraphen zusammen setzen !!!
- Im Beispiel sind die Abhängigkeiten stets von den Attributen der Söhne zu den Attributen des Vaters gerichtet.
 \implies Postorder-DFS-Traversierung
- Die Variablen-Abhängigkeiten können aber auch **komplizierter** sein ...

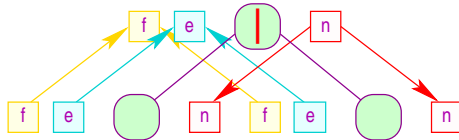
Beispiel: Simultane Berechnung von **empty**, **first**, **next** :

$$\boxed{x} : \quad \begin{aligned} \text{empty}[0] &:= (x \equiv \epsilon) \\ \text{first}[0] &:= \{x \mid x \neq \epsilon\} \\ &\quad // \text{ (keine Gleichung für next !!!) } \end{aligned}$$

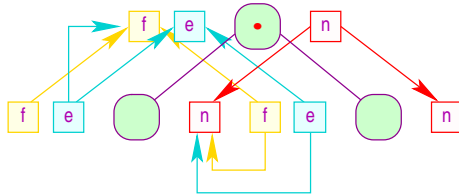
$$\boxed{\text{root:}} : \quad \begin{aligned} \text{empty}[0] &:= \text{empty}[1] \\ \text{first}[0] &:= \text{first}[1] \\ \text{next}[0] &:= \emptyset \\ \text{next}[1] &:= \text{next}[0] \end{aligned}$$



$$\boxed{|} : \quad \begin{aligned} \text{empty}[0] &:= \text{empty}[1] \vee \text{empty}[2] \\ \text{first}[0] &:= \text{first}[1] \cup \text{first}[2] \\ \text{next}[1] &:= \text{next}[0] \\ \text{next}[2] &:= \text{next}[0] \end{aligned}$$

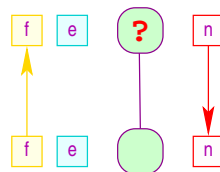
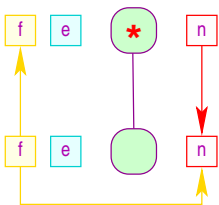


\square : $empty[0] := empty[1] \wedge empty[2]$
 $first[0] := first[1] \cup (empty[1]) ? first[2] : \emptyset$
 $next[1] := first[2] \cup (empty[2]) ? next[0]$
 $next[2] := next[0]$



\ast : $empty[0] := t$
 $first[0] := first[1]$
 $next[1] := first[1] \cup next[0]$

$?$: $empty[0] := t$
 $first[0] := first[1]$
 $next[1] := next[0]$



Problem:

- Eine Auswertungsstrategie kann es nur dann geben, wenn die Variablen-Abhängigkeiten in jedem attribuierten Baum **azyklisch** sind !!!
- Es ist **DEXPTIME**-vollständig, herauszufinden, ob keine zyklischen Variablenabhängigkeiten vorkommen können :-)

Ideen:

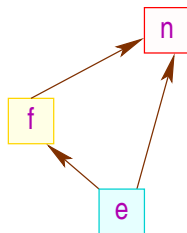
- (1) Die **Benutzerin** soll die Strategie spezifizieren ;-)
- (2) In der **Praxis** wirds schon nicht so schlimm kommen ;-}
- (3) Betrachte **Teilklassen** ...

Stark azyklische Attributierung:

Berechne eine **partielle Ordnung** auf den Attributen eines Knotens, die **kompatibel** mit den lokalen Attribut-Abhängigkeiten ist:

- Wir starten mit der trivialen Ordnung $\sqsubseteq = =$:-)
- Die aktuelle Ordnung setzen wir an den Sohn-Knoten in die lokalen Abhängigkeitsgraphen ein.
- Ergibt sich ein Kreis, geben wir auf :-))
- Andernfalls fügen wir alle Beziehungen $a \sqsubseteq b$ hinzu, für die es jetzt einen Pfad von $a[0]$ nach $b[0]$ gibt.
- Lässt sich \sqsubseteq nicht mehr vergrößern, hören wir auf ...

... im Beispiel:



Diskussion:

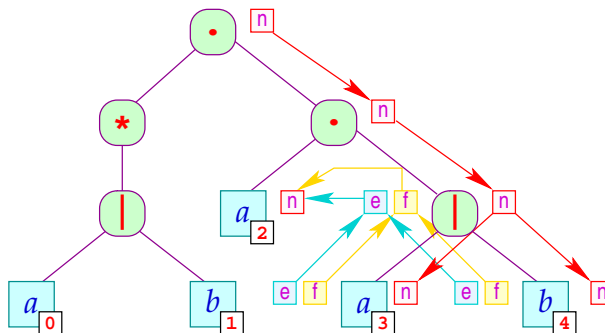
- Die Berechnung der partiellen Ordnung \sqsubseteq ist eine Fixpunkt-Berechnung :-)
- Die partielle Ordnung können wir in eine **lineare Ordnung** einbetten ...
- Die lineare Ordnung gibt uns an, in welcher Reihenfolge die Attribute berechnet werden müssen :-)
- Die lokalen Abhängigkeitsgraphen zusammen mit der linearen Ordnung erlauben die Berechnung einer Strategie ...

Mögliche Strategien:

(1) Bedarfsgetriebene Auswertung:

- Beginne mit der Berechnung eines Attributs.
- Sind die Argument-Attribute noch nicht berechnet, berechne rekursiv deren Werte :-)
- Besuche die Knoten des Baum **nach Bedarf...**

Beispiel, bedarfsgetrieben:



Diskussion:

- Die Reihenfolge hängt i.a. vom zu attributierenden Baum ab.
- Der Algorithmus muss sich merken, welche Attribute er bereits berechnet hat :-)
- Der Algorithmus besucht manche Knoten unnötig oft.
- Der Algorithmus ist **nicht-lokal** :-((

Mögliche Strategien (Forts.):

(2) Auswertung in Pässen:

- **Minimiere** die Anzahl der **Besuche** an jedem Knoten.
- Organisiere die Auswertung in **Durchläufe** durch den Baum.
- Berechne für jeden Pass eine **Besuchsstrategie** für die Knoten zusammen mit einer **lokalen Strategie** für jeden Knoten-Typ ...

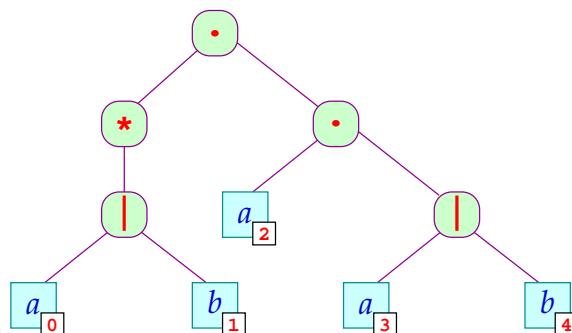
Achtung:

- Das minimale Attribut in der Anordnung für stark azyklische Attributierungen lässt sich stets in **einem Pass** berechnen :-)
- Man braucht folglich für stark azyklische Attributierungen maximal so viele Pässe, wie es Attribute gibt :-))
- Hat man einen Baum-Durchlauf zur Berechnung einiger Attribute, kann man überprüfen, ob er geeignet ist, gleichzeitig weitere Attribute auszuwerten
⇒ **Optimierungsproblem**

... im Beispiel:

empty und **first** lassen sich gemeinsam berechnen.
next muss in einem weiteren Pass berechnet werden :-)

Weiteres Beispiel: Nummerierung der Blätter eines Baums:



Idee:

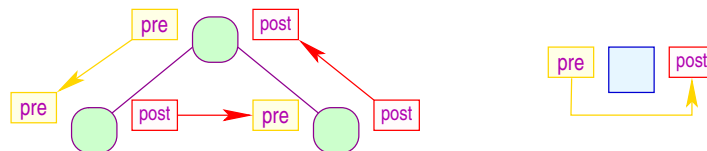
- Führe Hilfsattribute `pre` und `post` ein !
- Mit `pre` reichen wir einen Zählerstand nach unten
- Mit `post` reichen wir einen Zählerstand wieder nach oben ...

Root: `pre[0] := 0`
`pre[1] := pre[0]`
`post[0] := post[1]`

Node: `pre[1] := pre[0]`
`pre[2] := post[1]`
`post[0] := post[2]`

Leaf: `post[0] := pre[0] + 1`

... die lokalen Attribut-Abhängigkeiten:



- Die Attributierung ist offenbar stark azyklisch :-)
- Man kann alle Attribute in **einem Links-Rechts-Durchlauf** auswerten :-))
- So etwas nennen wir **L-Attributierung**.
- L-Attributierung liegt auch unseren **Query-Tools** zur Suche in XML-Dokumenten zugrunde \implies `fxgrep`

Praktische Erweiterungen:

- Symboltabellen, Typ-Überprüfung / Inferenz und (einfache) Codegenerierung können durch Attributierung berechnet werden :-)
- In diesen Anwendungen werden stets **Syntaxbäume** annotiert.
- Die Knoten-Beschriftungen entsprechen den Regeln einer kontextfreien Grammatik :-)
- Knotenbeschriftungen können in **Sorten** eingeteilt werden — entsprechend den **Nichtterminalen** auf der linken Seite ...
- Unterschiedliche Nichtterminale benötigen evt. **unterschiedliche Mengen von Attributen**.
- Eine **attributierte Grammatik** ist eine **CFG** erweitert um:
 - Attribute für jedes Nichtterminal;
 - lokale Attribut-Gleichungen.
- Damit können die syntaktische, Teile der semantischen Analyse wie der Codeerzeugung **generiert** werden :-)

4 Die Optimierungsphase

1. Vermeidung überflüssiger Berechnungen

- verfügbare Ausdrücke
- Konstantenpropagation/Array-Bound-Checks
- Code Motion

2. Ersetzen teurer Berechnungen durch billige

- Peep Hole Optimierung
- Inlining
- Reduction of Strength
- ...

3. Anpassung an Hardware

- Instruktions-Selektion
- Registerverteilung
- Scheduling
- Speicherverwaltung

Beobachtung 1: Intuitive Programme sind oft ineffizient.

Beispiel:

```
void swap (int i, int j) {
    int t;
    if (a[i] > a[j]) {
        t = a[j];
        a[j] = a[i];
        a[i] = t;
    }
}
```

Ineffizienzen:

- Adressen $a[i]$, $a[j]$ werden je dreimal berechnet :-)
- Werte $a[i]$, $a[j]$ werden zweimal geladen :-)

Verbesserung:

- Gehe mit Pointer durch das Feld a;
- speichere die Werte von a[i], a[j] zwischen!

```
void swap (int *p, int *q) {
    int t, ai, aj;
    ai = *p; aj = *q;
    if (ai > aj) {
        t = aj;
        *q = ai;
        *p = t;    // t kann auch noch
    }            // eingespart werden!
}
```

Beobachtung 2:

Höhere Programmiersprachen (sogar C :-)) abstrahieren von Hardware und Effizienz. Aufgabe des Compilers ist es, den natürlich erzeugten Code an die Hardware anzupassen.

Beispiele:

- ... Füllen von Delay-Slots;
- ... Einsatz von Spezialinstruktionen;
- ... Umorganisation der Speicherzugriffe für besseres Cache-Verhalten;
- ... Beseitigung (unnötiger) Tests auf Overflow/Range.

Beobachtung 3:

Programm-Verbesserungen sind nicht immer korrekt :-)

Beispiel:

$$y = f() + f(); \quad \implies \quad y = 2 * f();$$

Idee: Spare zweite Auswertung von f() ???

Problem: Die zweite Auswertung könnte ein anderes Ergebnis liefern als die erste (z.B. wenn f() aus der Eingabe liest :-)

Folgerungen:

- ⇒ Optimierungen haben **Voraussetzungen**.
- ⇒ Die **Voraussetzungen** muss man:
 - formalisieren,
 - überprüfen :-)
- ⇒ Man muss beweisen, dass die Optimierung **korrekt** ist, d.h. die **Semantik** erhält !!!

Beobachtung 4:

Optimierungs-Techniken hängen von der **Programmiersprache** ab:

- welche Ineffizienzen auftreten;
- wie gut sich Programme analysieren lassen;
- wie schwierig / unmöglich es ist, Korrektheit zu beweisen ...

Beispiel: Java

Unvermeidbare Ineffizienzen:

- * Array-Bound Checks;
- * dynamische Methoden-Auswahl;
- * bombastische Objekt-Organisation ...

Analysierbarkeit:

- + keine Pointer-Arithmetik;
- + keine Pointer in den Stack;
- dynamisches Klassenladen;
- Reflection, Exceptions, Threads, ...

Korrektheitsbeweise:

- + mehr oder weniger definierte Semantik;
- Features, Features, Features;
- Bibliotheken mit wechselndem Verhalten ...

Beispiel: Zwischendarstellung von `swap()`

```
0:  A1 = A0 + 1 * i;      //  A0 == &a
1:  R1 = M[A1];          //  R1 == a[i]
2:  A2 = A0 + 1 * j;
3:  R2 = M[A2];          //  R2 == a[j]
4:  if (R1 > R2) {
5:      A3 = A0 + 1 * j;
6:      t = M[A3];
7:      A4 = A0 + 1 * j;
8:      A5 = A0 + 1 * i;
9:      R3 = M[A5];
10:     M[A4] = R3;
11:     A6 = A0 + 1 * i;
12:     M[A6] = t;
    }
```

Optimierung 1: $1 * R \implies R$

Optimierung 2: Wiederbenutzung von Teilausdrücken

$$A_1 == A_5 == A_6$$
$$A_2 == A_3 == A_4$$

$$M[A_1] == M[A_5]$$
$$M[A_2] == M[A_3]$$

$$R_1 == R_3$$

Damit erhalten wir:

```

A1 = A0 + i;
R1 = M[A1];
A2 = A0 + j;
R2 = M[A2];
if (R1 > R2) {
    t = R2;
    M[A2] = R1;
    M[A1] = t;
}

```

Optimierung 3: Verkürzung von Zuweisungsketten :-)

Ersparnis:

	vorher	nachher
+	6	2
*	6	0
load	4	2
store	2	2
>	1	1
=	6	2

5 Perspektiven

Herausforderungen:

- neue Hardware;
- neue Programmiersprachen;
- neue Anwendungen für Compiler-Technologie :-)

5.1 Hardware

Die Code-Erzeugung soll die Möglichkeiten der Hardware **optimal** ausnutzen ...

Herausforderungen:

Neue Hardware:

- Speicher-Hierarchie mit unterschiedlich schnellen Caches für verschiedene Zwecke;
- On-Board Nebenläufigkeit mit Pipelines, mehreren ALUs, spekulativer Parallelität, ...
- Interaktion mit mächtigen Zusatzkomponenten wie Graphik-Karten ...

Eingeschränkte Hardware:

z.B. auf Chip-Karten, in Kühlschränken, Bremsanlagen, Steuerungen ...

⇒ ubiquitous Computing

- minimaler Energie-Verbrauch :-)
- minimaler Platz :-)
- Echtzeit-Anforderungen;
- Korrektheit;
- Fehler-Toleranz.

5.2 Programmiersprachen

Spezielle Features:

- mobiler Code;
- Nebenläufigkeit;
- graphische Benutzeroberflächen;
- Sicherheits-Komponenten;
- neue / bessere Typsysteme;
- Unterstützung für Unicode und XML.

Neue Programmiersprachen:

- XSLT;
- XQuery;
- Web-Services;
- anwendungs-spezifische Sprachen ...

5.3 Programmierumgebungen

Diverse **Programmierhilfsmittel** benutzen Compiler-Technologie ...

- syntax-gesteuerte Editoren;
- Programm-Visualisierung;
- automatische Programm-Dokumentation;
- **partielle** Codeerzeugung aus **UML**-Modellen;
- **UML**-Modell-Extraktion \implies **reverse engineering**
- Konsistenz-Überprüfungen, Fehlersuche;
- Portierung.

5.4 Neue Anforderungen

- Zuverlässigkeit
- Sicherheit

6 Instruktions-Selektion

Problem:

- unregelmäßige Instruktionssätze ...
- mehrere Adressierungsarten, die evt. mit arithmetischen Operationen kombiniert werden können;
- Register für unterschiedliche Verwendungen ...

Beispiel: Motorola MC68000

Dieser einfachste Prozessor der 680x0-Reihe besitzt

- 8 Daten- und 8 Adressregister;
- eine Vielzahl von Adressierungsarten ...

Notation	Beschreibung	Semantik
D_n	Datenregister direkt	D_n
A_n	Adressregister direkt	A_n
(A_n)	Adressregister indirekt	$M[A_n]$
$d(A_n)$	Adressregister indirekt mit Displacement	$M[A_n + d]$
$d(A_n, D_m)$	Adressregister indirekt mit Index und Displacement	$M[A_n + D_m + d]$
x	Absolut kurz	$M[x]$
x	Absolut lang	$M[x]$
$\#x$	Unmittelbar	x

- Der MC68000 ist eine **2-Adress-Maschine**, d.h. ein Befehl darf maximal 2 Adressierungen enthalten. Die Instruktion:

$\text{add } D_1 \ D_2$

addiert die Inhalte von D_1 und D_2 und speichert das Ergebnis nach und D_2 :-)

- Die meisten Befehle lassen sich auf Bytes, Wörter (2 Bytes) oder Doppelwörter (4 Bytes) anwenden.
Das unterscheiden wir durch Anhängen von **.B**, **.W**, **.D** (Default: **.W**)
- Die **Ausführungszeit** eines Befehls ergibt sich (i.a.) aus den Kosten der Operation plus den Kosten für die Adressierung der Operanden ...

	Adressierungsart	Byte / Wort	Doppelwort
D_n	Datenregister direkt	0	0
A_n	Adressregister direkt	0	0
(A_n)	Adressregister indirekt	4	8
$d(A_n)$	Adressregister indirekt mit Displacement	8	12
$d(A_n, D_m)$	Adressregister indirekt mit Index und Displacement	10	14
x	Absolut kurz	8	12
x	Absolut lang	12	16
$\#x$	Unmittelbar	4	8

Beispiel:

Die Instruktion: **move.B** $8(A_1, D_1.W), D_5$
benötigt: $4 + 10 + 0 = 14$ Zyklen

Alternativ könnten wir erzeugen:

adda $\#8, A_1$ Kosten: $8 + 8 + 0 = 16$
adda $D_1.W, A_1$ Kosten: $8 + 0 + 0 = 8$
move.B $(A_1), D_5$ Kosten: $4 + 4 + 0 = 8$

mit Gesamtkosten **32** oder:

adda $D_1.W, A_1$ Kosten: $8 + 0 + 0 = 8$
move.B $8(A_1), D_5$ Kosten: $4 + 8 + 0 = 12$

mit Gesamtkosten **20** :-)

Achtung:

- Die verschiedenen Code-Sequenzen sind im Hinblick auf den Speicher und das Ergebnis äquivalent !
- Sie unterscheiden sich im Hinblick auf den Wert des Registers A_1 sowie die gesetzten Bedingungs-Codes !!
- Ein schlauer Instruktions-Selektor muss solche Randbedingungen berücksichtigen :-)

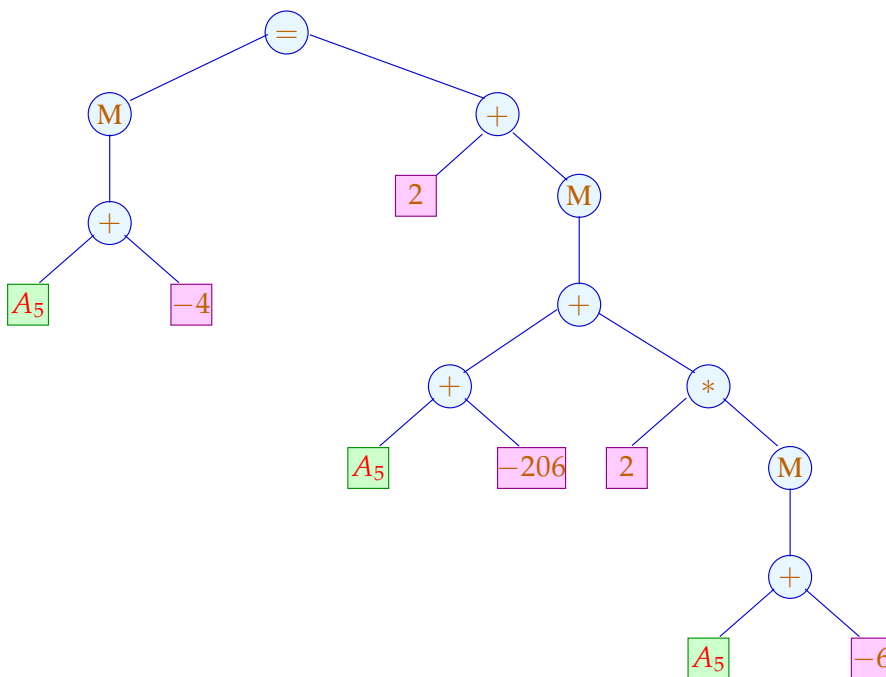
Etwas größeres Beispiel:

```
int b, i, a[100];  
b = 2 + a[i];
```

Nehmen wir an, die Variablen werden relativ zu einem **Framepointer** A_5 mit den Adressen $-4, -6, -206$ adressiert. Dann entspricht der Zuweisung das Stück Zwischen-Code:

$$M[A_5 - 4] = 2 + M[A_5 - 206 + 2 \cdot M[A_5 - 6]];$$

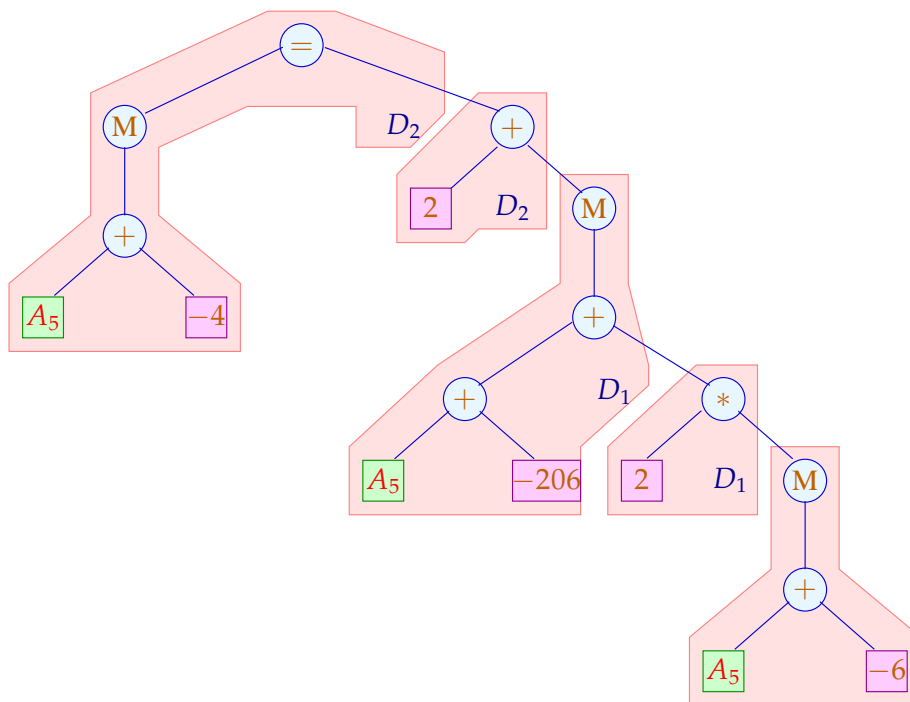
Das entspricht dem Syntaxbaum:



Eine mögliche Code-Sequenz:

move	$-6(A_5), D_1$	Kosten:	12
add	D_1, D_1	Kosten:	4
move	$-206(A_5, D_1), D_2$	Kosten:	14
addq	$\#2, D_2$	Kosten:	4
move	$D_2, -4(A_5)$	Kosten:	12

Gesamtkosten : 46



Eine alternative Code-Sequenz:

move.L	A_5, A_1	Kosten:	4
adda.L	$\#-6, A_1$	Kosten:	12
move	$(A_1), D_1$	Kosten:	8
mulu	$\#2, D_1$	Kosten:	44
move.L	A_5, A_2	Kosten:	4
adda.L	$\#-206, A_2$	Kosten:	12
adda.L	D_1, A_2	Kosten:	8
move	$(A_2), D_2$	Kosten:	8
addq	$\#2, D_2$	Kosten:	4
move.L	A_5, A_3	Kosten:	4
adda.L	$\#-4, A_3$	Kosten:	12
move	$D_2, (A_3)$	Kosten:	8
<i>Gesamtkosten :</i>			124

Diskussion:

- Die Folge **ohne komplexe Adressierungsarten** ist erheblich teurer :-)
- Sie benötigt auch mehr Hilfsregister :-)
- Die beiden Folgen sind nur äquivalent im Hinblick auf den Speicher — die Register haben anschließend verschiedene Inhalte ...
- Eine korrekte Folge von Instruktionen kann als eine **Pflasterung** des Syntaxbaums aufgefasst werden !!!

Genereller Ansatz:

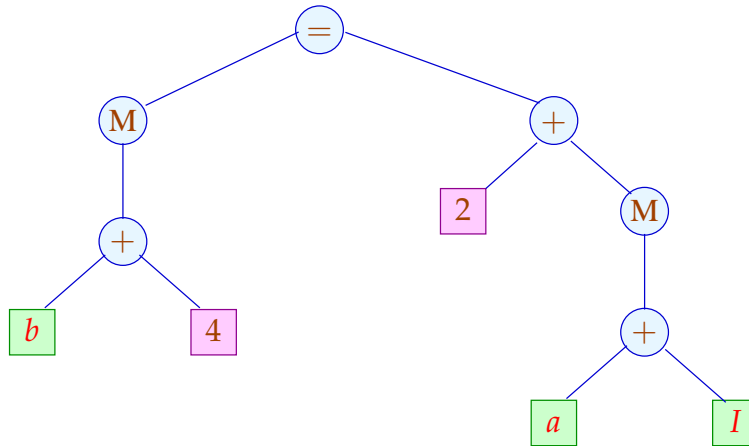
- Wir betrachten Basis-Blöcke **vor der Registerverteilung**:

$$\begin{aligned}A &= a + I; \\D_1 &= M[A]; \\D_2 &= D_1 + 2; \\B &= b + 4; \\M[B] &= D_2\end{aligned}$$

- Wir fassen diese als **Folge von Bäumen** auf. **Wurzeln**:
 - Werte, die mehrmals verwendet werden;

- Variablen, die am Ende des Blocks lebendig sind;
- Stores.

... im Beispiel:



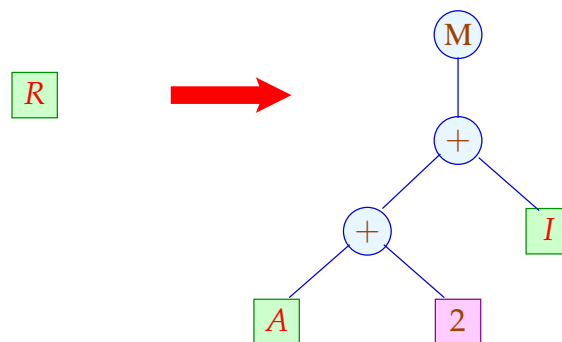
Die Hilfsvariablen A, B, D_1, D_2 sind vorerst verschwunden :-)

Idee:

Beschreibe den Effekt einer Instruktion als **Ersetzungsregel** auf Bäumen:

Die Instruktion: $R = M[A + 2 + D];$

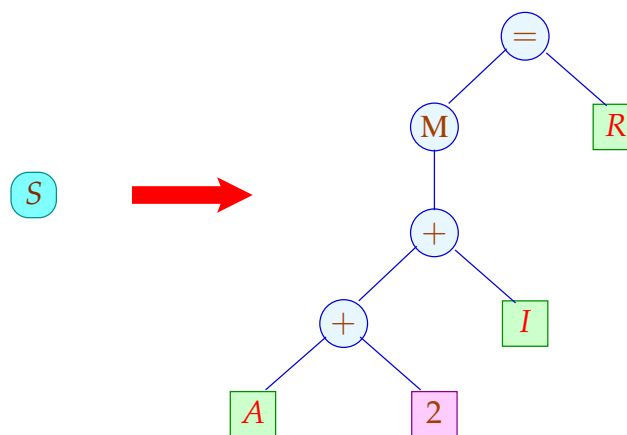
entspricht zum Beispiel:



linke Seite	Ergebnisregister(klasse)
rechte Seite	berechneter Wert für Ergebnisregister
innere Knoten	<ul style="list-style-type: none"> • Load M • Arithmetik
Blätter	<ul style="list-style-type: none"> • Argumentregister(klassen) • Konstanten(klasse)

Die Grundidee erweitern wir (evt.) um eine Store-Operation.

Für die Instruktion: $M[A + 2 + D] = R;$
erlauben wir uns:



Die linke Seite S kommt nicht in rechten Seiten vor :-)

Spezifikation des Instruktionssatzes:

- (1) verfügbare Registerklassen // Nichtterminale
- (2) Operatoren und Konstantenklassen // Terminale
- (3) Instruktionen // Regeln

\implies reguläre Baumgrammatik

Triviales Beispiel:

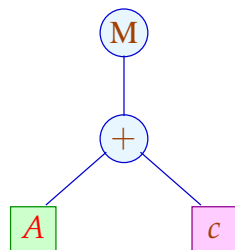
Loads :	Comps :	Moves :
$D \rightarrow M[A]$	$D \rightarrow c$	$D \rightarrow A$
$D \rightarrow M[A + A]$	$D \rightarrow D + D$	$A \rightarrow D$

- Registerklassen D (Data) und A (Address).
- Arithmetik wird nur für Daten unterstützt ...
- Laden nur für Adressen :-)
- Zwischen Daten- und Adressregistern gibt es Moves.

Target: $M[A + c]$

Aufgabe:

Finde Folge von Regelanwendungen, die das Target aus einem Nichtterminal erzeugt ...



Die **umgekehrte** Folge der Regelanwendungen liefert eine geeignete Instruktionsfolge :-)

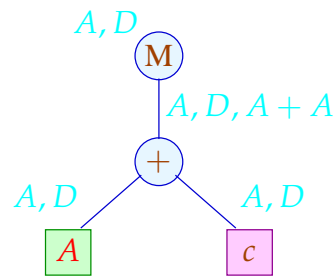
Verschiedene Ableitungen liefern verschiedene Folgen ...

Problem:

- Wie durchsuchen wir systematisch die Menge aller Ableitungen ?
- Wie finden wir die **beste** ??

Beobachtung:

- Nichtterminale stehen stets an den **Blättern**.
- Statt eine Ableitung für das Target topdown zu raten, sammeln wir sämtliche Möglichkeiten bottom-up auf
 ⇒ **Tree parsing**
- Dazu lesen wir die Regeln **von rechts nach links** ...



Für jeden Teilbaum t des Targets sammeln wir die Menge

$$Q(t) \subseteq \{S\} \cup \text{Reg} \cup \text{Term}$$

Reg die Menge der Registerklassen,

Term die Menge der Teilbäume rechter Seiten — auf mit:

$$Q(t) = \{s \mid s \Rightarrow^* t\}$$

Diese ergeben sich zu:

$$Q(R) = \text{Move}\{R\}$$

$$Q(c) = \text{Move}\{c\}$$

$$Q(a(t_1, \dots, t_k)) = \text{Move}\{s = a(s_1, \dots, s_k) \in \text{Term} \mid s_i \in Q(t_i)\}$$

// normalerweise $k \leq 2$:-)

Die Hilfsfunktion **Move** bildet den Abschluss unter Regelanwendungen:

$$\text{Move}(L) \supseteq L$$

$$\text{Move}(L) \supseteq \{R \in \text{Reg} \mid \exists s \in L : R \rightarrow s\}$$

Die kleinste Lösung dieses Constraint-Systems lässt sich aus der Grammatik in **linearer** Zeit berechnen :-)

// Im Beispiel haben wir in $Q(t)$ auf s verzichtet,

// falls s kein **echter** Teilterm einer rechten Seite ist :-)

Auswahlkriterien:

- Länge des Codes;
- Laufzeit der Ausführung;
- Parallelisierbarkeit;
- ...

Achtung:

Die Laufzeit von Instruktionen kann vom Kontext abhängen !!?

Vereinfachung:

Jede Instruktion r habe Kosten $c[r]$.

Die Kosten einer Instruktionsfolge sind **additiv**:

$$c[r_1 \dots r_k] = c[r_1] + \dots + c[r_k]$$

	c	Instruktion
0	3	$D \rightarrow M[A + A]$
1	2	$D \rightarrow M[A]$
2	1	$D \rightarrow D + D$
3	1	$D \rightarrow c$
4	1	$D \rightarrow A$
5	1	$A \rightarrow D$

Aufgabe:

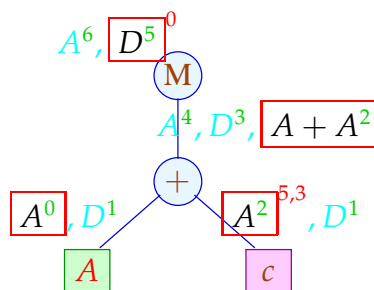
Wähle eine Instruktionsfolge mit minimalen Kosten !

Idee:

Samme Ableitungen bottom-up auf unter

- * Kostenkalkulation und
- * Auswahl.

... im Beispiel:



Kostenkalkulation:

$$\begin{aligned}c_t[s] &= c_{t_1}[s_1] + \dots + c_{t_k}[s_k] && \text{falls } s = a(s_1, \dots, s_k), t = a(t_1, \dots, t_k) \\c_t[R] &= \bigcap \{c[R, s] + c_t[s] \mid s \in Q(t)\} && \text{wobei}\end{aligned}$$

$$\begin{aligned}c[R, s] &\leq c[r] && \text{falls } r : R \rightarrow s \\c[R, s] &\leq c[r] + c[R', s] && \text{falls } r : R \rightarrow R'\end{aligned}$$

Das Constraint-System für $c[R, s]$ kann in Zeit $\mathcal{O}(n \cdot \log n)$ gelöst werden
— falls n die Anzahl der Paare R, s ist :-)

Für jedes R, s liefert die Fixpunkt-Berechnung eine Folge:

$$\pi[R, s] : R \Rightarrow R_1 \Rightarrow \dots \Rightarrow R_k \Rightarrow s$$

deren Kosten gerade $c[R, s]$ ist :-)

Mithilfe der $\pi[R, s]$ lässt sich eine billigste Ableitung topdown rekonstruieren :-)

Im Beispiel:

$$\begin{aligned}D_2 &= c; \\A_2 &= D_2; \\D_1 &= M[A_1 + A_2];\end{aligned}$$

mit Kosten 5. Die Alternative:

$$\begin{aligned}D_2 &= c; \\D_3 &= A_1; \\D_4 &= D_3 + D_2; \\A_2 &= D_4; \\D_1 &= M[A_2];\end{aligned}$$

hätte Kosten 7 :-)

Diskussion:

- Die Code-Erzeugung muss schnell gehn :-)
- Anstelle für jeden Knoten neu zu überprüfen, wie die Regeln zusammen passen, kann die Berechnung auch in einen **endlichen Automaten** kompiliert werden :-))

Ein deterministischer endlicher Baumautomat (DTA) A besteht aus:

- Q \equiv endliche Menge von Zuständen
- Σ \equiv Operatoren und Konstanten
- δ_a \equiv Übergangsfunktion für $a \in \Sigma$
- $F \subseteq Q$ \equiv akzeptierende Zustände

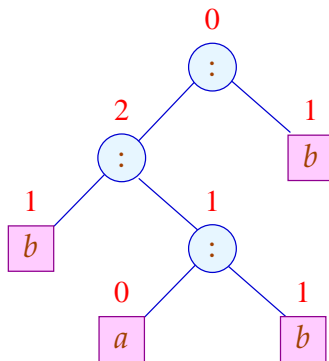
Dabei ist:

- $\delta_c : Q \rightarrow Q$ falls c Konstante
- $\delta_a : Q^k \rightarrow Q$ falls a k -stellig

Beispiel:

$$\begin{aligned}
 Q &= \{0, 1, 2\} & F &= \{0\} \\
 \Sigma &= \{a, b, :\} \\
 \delta_a &= 0 & \delta_b &= 1 \\
 \delta : (s_1, s_2) &= (s_1 + s_2) \% 3
 \end{aligned}$$

// akzeptiert alle Bäume mit $3 \cdot k$ b -Blättern



Der Zustand an einem Knoten a ergibt sich aus den Zuständen der Kinder mittels δ_a (-:

$$\begin{aligned}
 Q(c) &= \delta_c \\
 Q(a(t_1, \dots, t_k)) &= \delta_a(Q(t_1), \dots, Q(t_k))
 \end{aligned}$$

Die von A definierte Sprache (oder: Menge von Bäumen) ist:

$$\mathcal{L}(A) = \{t \mid Q(t) \in F\}$$

... in unserer Anwendung:

Q	\equiv	Teilmengen von $\text{Reg} \cup \text{Term} \cup \{S\}$
	//	I.a. werden nicht sämtliche Teilmengen benötigt :-)
F	\equiv	gewünschter Effekt
δ_R	\equiv	$\text{Move}\{R\}$
δ_c	\equiv	$\text{Move}\{c\}$
$\delta_a(Q_1, \dots, Q_k)$	\equiv	$\text{Move}\{s = a(s_1, \dots, s_k) \in \text{Term} \mid s_i \in Q_i\}$

... im Beispiel:

$$\begin{aligned}\delta_c &= \{A, D\} = q_0 \\ &= \delta_A \\ &= \delta_D \\ \delta_+(q_0, q_0) &= \{A, D, A + A\} = q_1 \\ &= \delta_+(q_0, -) \\ &= \delta_+(-, q_0) \\ \delta_M(q_0) &= \{A, D\} = q_0 \\ &= \delta_M(q_1)\end{aligned}$$

Um die Anzahl der Zustände zu reduzieren, haben wir die vollständigen rechten Seiten, die keine echten Teilmuster sind, in den Zuständen weggelassen :-)

Integration der Kostenberechnung:

Problem:

Kosten können (im Prinzip) beliebig groß werden ;-(
Unser FTA besitzt aber nur endlich viele Zustände :-((

Idee:

Pelegri-Lopart 1988

Betrachte nicht absolute Kosten — sondern relative !!!



Eduardo Pelegri-Llopart,
Sun Microsystems, Inc.

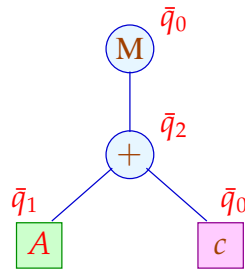
Beobachtung:

- In gängigen Prozessoren kann man Werte von jedem Register in jedes andere schieben
 \implies
 Die Kosten zwischen Registern differieren nur um eine Konstante :-)
- Komplexe rechte Seiten lassen sich i.a. mittels elementarerer Instruktionen simulieren
 \implies
 Die Kosten zwischen Teilausdrücken und Registern differieren nur um eine Konstante :-))
- Die Kostenberechnung ist additiv \implies
 Wir können statt mit absoluten Kosten-Angaben auch mit Kosten-Differenzen rechnen !!!
 Von diesen gibt es nur endlich viele :-)

... im Beispiel:

$$\begin{aligned}
 \delta_c &= \{A \mapsto 1, D \mapsto 0\} = \bar{q}_0 \\
 &= \delta_D \\
 \delta_A &= \{A \mapsto 0, D \mapsto 1\} = \bar{q}_1 \\
 \delta_+(\bar{q}_1, \bar{q}_0) &= \{A \mapsto 2, D \mapsto 1, A + A \mapsto 0\} = \bar{q}_2 \\
 \delta_+(\bar{q}_0, \bar{q}_0) &= \{A \mapsto 1, D \mapsto 0, A + A \mapsto 1\} = \bar{q}_3 \\
 \delta_+(\bar{q}_1, \bar{q}_1) &= \{A \mapsto 4, D \mapsto 3, A + A \mapsto 0\} = \bar{q}_4 \\
 &\dots \\
 \delta_M(\bar{q}_2) &= \{A \mapsto 1, D \mapsto 0\} = \bar{q}_0 \\
 &= \delta_M(\bar{q}_i) \quad , \quad i = 0, \dots, 4
 \end{aligned}$$

... das liefert die folgende Berechnung:



Für jede Konstanten-Klasse c und jedes Register R in δ_c tabellieren wir die zu wählende billigste Berechnung:

$$c : \{A \mapsto 5, 3, D \mapsto 3\}$$

Analog tabellieren wir für jeden Operator a , jedes $\tau \in \bar{Q}^k$ und jedes R in $\delta_a(\tau)$:

M	select_M
\bar{q}_0	$\{A \mapsto 5, 1, D \mapsto 1\}$
\bar{q}_1	$\{A \mapsto 5, 1, D \mapsto 1\}$
\bar{q}_2	$\{A \mapsto 5, 0, D \mapsto 0\}$
\bar{q}_3	$\{A \mapsto 5, 1, D \mapsto 1\}$
\bar{q}_4	$\{A \mapsto 5, 0, D \mapsto 0\}$

Für “+” ist die Tabelle besonders einfach:

+	\bar{q}_j
\bar{q}_i	$\{A \mapsto 5, 3, D \mapsto 3\}$

Problem:

- Für reale Instruktionssätze benötigt man leicht um die 1000 Zustände.
- Die Tabellen für mehrstellige Operatoren werden riesig :-)

\implies Wir benötigen Verfahren der Tabellen-Komprimierung ...

Tabellen-Kompression:

Die Tabelle für “+” sieht im Beispiel so aus:

+	\bar{q}_0	\bar{q}_1	\bar{q}_2	\bar{q}_3	\bar{q}_4
\bar{q}_0	\bar{q}_3	\bar{q}_2	\bar{q}_3	\bar{q}_3	\bar{q}_3
\bar{q}_1	\bar{q}_2	\bar{q}_4	\bar{q}_2	\bar{q}_2	\bar{q}_2
\bar{q}_2	\bar{q}_3	\bar{q}_2	\bar{q}_3	\bar{q}_3	\bar{q}_3
\bar{q}_3	\bar{q}_3	\bar{q}_2	\bar{q}_3	\bar{q}_3	\bar{q}_3
\bar{q}_4	\bar{q}_3	\bar{q}_2	\bar{q}_3	\bar{q}_3	\bar{q}_3

Die meisten Zeilen / Spalten sind offenbar ganz ähnlich ;-)

Idee 1: Äquivalenzklassen

Wir setzen $q \equiv_a q'$, genau dann wenn

$$\begin{aligned} \forall p : \quad & \delta_a(q, p) = \delta_a(q', p) \quad \wedge \quad \delta_a(p, q) = \delta_a(p, q') \\ & \wedge \text{select}_a(q, p) = \text{select}_a(q', p) \quad \wedge \quad \text{select}_a(p, q) = \text{select}_a(p, q') \end{aligned}$$

Im Beispiel:

$$\begin{aligned} Q_1 &= \{\bar{q}_0, \bar{q}_2, \bar{q}_3, \bar{q}_4\} \\ Q_2 &= \{\bar{q}_1\} \end{aligned}$$

mit:

+	Q_1	Q_2
Q_1	\bar{q}_3	\bar{q}_2
Q_2	\bar{q}_2	\bar{q}_4

Idee 2: Zeilenverschiebung

Sind viele Einträge gleich (im Beispiel etwa **default** = \bar{q}_3), genügt es, die übrigen Einträge zu speichern ;-)

Im Beispiel:

+	\bar{q}_0	\bar{q}_1	\bar{q}_2	\bar{q}_3	\bar{q}_4
\bar{q}_0		\bar{q}_2			
\bar{q}_1	\bar{q}_2	\bar{q}_4	\bar{q}_2	\bar{q}_2	\bar{q}_2
\bar{q}_2		\bar{q}_2			
\bar{q}_3		\bar{q}_2			
\bar{q}_4		\bar{q}_2			

Dann legen wir:

- (1) gleiche Zeilen übereinander;
- (2) verschiedene (Klassen von) Zeilen auf Lücke verschoben übereinander:

	\bar{q}_0	\bar{q}_1	\bar{q}_2	\bar{q}_3	\bar{q}_4		0	1	
class	0	1	0	0	0		disp	0	2

	0	1	2	3	4	5	6
A	\bar{q}_2	\bar{q}_2	\bar{q}_4	\bar{q}_2	\bar{q}_2	\bar{q}_2	\bar{q}_2
valid	0	0	1	1	1	1	1

Für jeden Eintrag im ein-dimensionalen Feld **A** vermerken wir in **valid**, zu welcher Zeile der Eintrag gehört ...

Ein Feld-Zugriff $\delta_+(\bar{q}_i, \bar{q}_j)$ wird dann so realisiert:

```

 $\delta_+(\bar{q}_i, \bar{q}_j) =$  let  $c = \text{class}[\bar{q}_i];$ 
                         $d = \text{disp}[c];$ 
in if ( $\text{valid}[d + j] \equiv c$ )
    then  $A[d + j]$ 
    else default
end

```



Reinhard Wilhelm, Saarbrücken

Diskussion:

- Die Tabellen werden i.a. erheblich kleiner.
- Dafür werden Tabellenzugriffe etwas teurer.
- Das Verfahren versagt in einigen (theoretischen) Fällen.
- Dann bleibt immer noch das **dynamische** Verfahren ...

möglicherweise mit **Caching** der einmal berechneten Werte,
um unnötige Mehrfachberechnungen zu vermeiden :-)