Übersetzerbau

Mitschrift von www.kuertz.name

Hinweis: Dies ist **kein offizielles Script**, sondern nur eine private Mitschrift. Die Mitschriften sind teweilse **unvollständig, falsch oder inaktuell**, da sie aus dem Zeitraum 2001–2005 stammen. Falls jemand einen Fehler entdeckt, so freue ich mich dennoch über einen kurzen Hinweis per E-Mail – vielen Dank!

Klaas Ole Kürtz (klaasole@kuertz.net)

Inhaltsverzeichnis

1 Einführung

Literaturhinweise siehe Webseite, insbesondere A. Appel: „Modern compiler construction in ML“ und der Klassiker Aho, Sethi, Ullman: „Compilers – Principles, Techniques, Tools“.

1.1 Was ist ein Übersetzer?

Ein **Übersetzer/Compiler** ist selbst ein Programm, das jedem Programmiersprachen-Programm (PS-Programm, Quellprogramm) ein semantisch äquivalentes Maschinensprachen-Programm (MS-Programm, Ziel-Programm) zuordnet. Die zu übersetzende Programmiersprache ist meist eine **höhere Programmiersprache**:

- **imperative** Programmiersprachen (Inhalt dieser Vorlesung): Wertzuweisung, Kontroll- und Datenstrukturen, Module, Klassen
- **deklarative** Programmiersprachen: funktionale, logische Sprachen
- **nebenläufige** Programmiersprachen: parallele, verteilte Prozesse

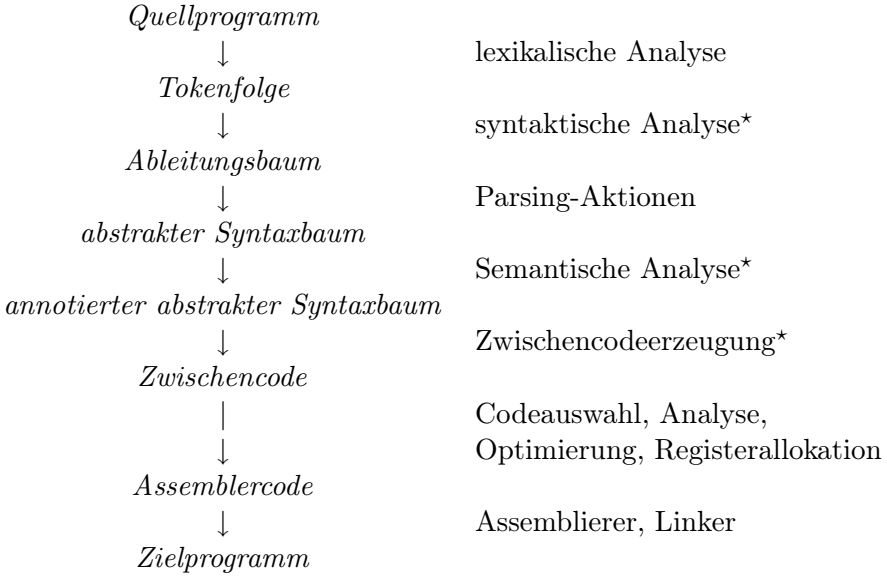
Die Maschinensprache (im allgemeinen für einen von-Neumann-Rechner) ist ausgelegt z.B. auf:

- **CISC**-Rechner (complex instruction set computers) stellen relativ viele Befehle zur Verfügung
- **RISC**-Architekturen (reduced instruction set computers) haben einen kleineren, einfacheren Befehlssatz, dafür werden die Befehle aber z.T. schneller abgearbeitet (u.a. durch Pipelining)
- **Parallel-Rechner**, beispielsweise SIMD (single instruction, multiple data) oder MIMD (multiple instruction, multiple data)

Die **Aufgaben** eines Übersetzers bestehen aus dem Erkennen der Syntax eines Programms und dem Erkennen der Semantik (Übersetzung in semantisch äquivalentes Programm). Übersetzer sind komplexe Programme, die hohe **Korrektheitsanforderungen** erfüllen müssen (beispielsweise bei sicherheitskritischen Systemen). Daher werden Übersetzer realisiert durch eine geeignete Struktur und geeignete Spezifizierungen.

1.2 Struktur von Übersetzern

Die typische Struktur eines Übersetzers:



Phasenübergreifend existiert bei $\star$ die **Symboltabelle**, eine zentrale Datenstruktur des Compilers. Die einzelnen **Phasen** des Übersetzungsvorgangs:

- **lexikalische Analyse**: Zweck ist die Erkennung von zusammengehörigen Symbolen, Kommentaren etc. Wekzeug: reguläre Ausdrücke ($\rightarrow$ endliche Automaten)
- **syntaktische Analyse**: dient der Erkennung der Programmstruktur, Werkzeug sind kontextfreie Grammatiken ($\rightarrow$ Kellerautomat)
- **Parsing-Aktionen**: zur Vereinfachung der Programmstruktur (alles entfernen, was semantisch egal ist)
- **semantische Analyse**: Erkennen von Kontextabhängigkeiten (Zuordnung zwischen Definition und Verwendung, Typanalyse)

  *Ab hier wird ein Programm als **korrekt** akzeptiert!*

- **Zwischencodeerzeugung**: vereinfachter, maschinenunabhängiger Code (Portabilität!)

  *Bis hier ist die Prozedur **maschinenunabhängig!***

- **Codeerzeugung**: Erzeugung konkreten Maschinencodes

- **Assembler:** Auflösung von Adressen

Als **Frontend** bezeichnet man den maschinenunabhängigen Teil des Compilers bis zum Zwischencode, alles weitere wird als **Backend** bezeichnet. Häufig existiert ein Übersetzer in Form eines Frontends mit mehreren Backends für unterschiedliche Architekturen. Die Begriffe **One-Pass-** oder ***n*-Pass-Compiler** geben an, wie oft der Compiler über den Quellcode läuft (bzw. heute über den Ableitungsbaum), früher wurde One-Pass bevorzugt (z.B. Pascal mittels Vorwärtsreferenzen `FORWARD`).

Die Ziele der Vorlesung sind:

- Grundlegende Techniken/Methoden des Übersetzerbaus kennenlernen
- Anwendung von Theorie (z.B. Theoretische Informatik) in der Praxis
- Übersetzer als Beispiel eines guten Softwareentwurfs

Implementierungssprache in dieser Vorlesung ist die funktionale Sprache Haskell – dabei wird der Haskell-98-Standard verwendet, als Interpreter wird Hugs benutzt (siehe auch die Einführung zu Haskell im Anhang B).

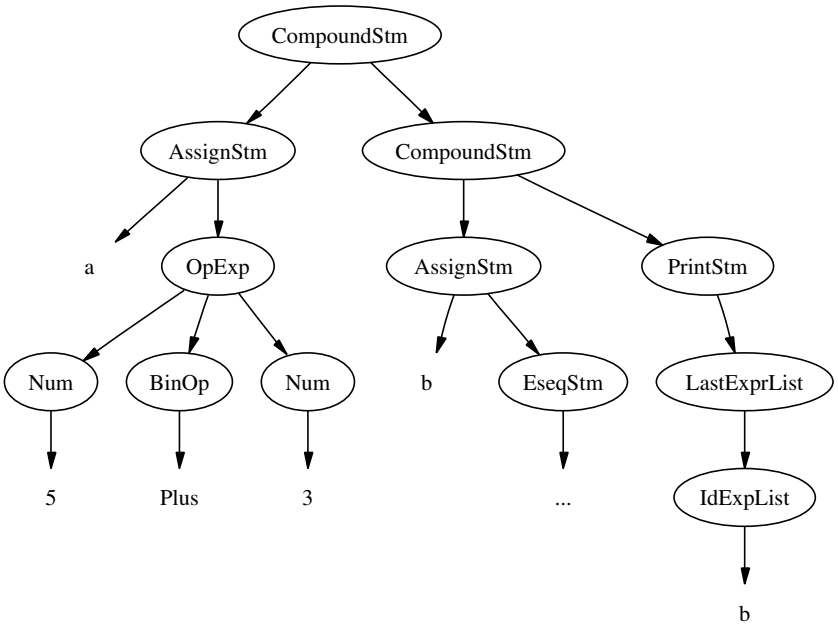
Als einfach zu kompilierende Beispielsprache verwenden wir Simple, die nur Zuweisungen, einfache Ausdrücke (mit Seiteneffekten) und Anweisungssequenzen erlaubt und definiert ist durch eine kontextfreie Grammatik (ohne Zahlen; Bezeichner sind Terminalsymbole):

Regel			Name
Stm	$\longrightarrow$	*Stm*; *Stm*	CompoundStm
Stm	$\longrightarrow$	*Id* `:=` *Exp*	AssignStm
Stm	$\longrightarrow$	`print(`*ExpList*`)`	PrintStm
Exp	$\longrightarrow$	*Id*	IdExp
Exp	$\longrightarrow$	*Num*	NumExp
Exp	$\longrightarrow$	*Exp BinOp Exp*	OpExp
Exp	$\longrightarrow$	`(`*Stm*`,`*Exp*`)`	EseqExp
ExpList	$\longrightarrow$	*Exp*`,` *Explist*	PairExpList
ExpList	$\longrightarrow$	*Exp*	LastExpList
BinOp	$\longrightarrow$	`+`	Plus
BinOp	$\longrightarrow$	`*`	Times
BinOp	$\longrightarrow$	`-`	Minus
BinOp	$\longrightarrow$	`/`	Div

Beispiel: Folgendes ist ein einfaches Simple-Programm:

```
a := 5+3; b := (print(a, a-1), 10*a); print(b)
```

Das Programm ergibt die Ausgabe `8 7 80`. Darstellung im Ableitungsbaum:



Diese Baumstrukturen kann man in Haskell oder SML mittels folgender Datentypen darstellen:

```
type Id = String
data BinOp = Plus|Minus|Times|Div
data Stm = CompoundStm Stm Stm|AssignStm Id Exp|
           PrintStm [Exp]
data Exp = IdExp Id | NumExp Int |
           OpExp Exp BinOp Exp | EseqExp Stm Exp
```

Der Beispielausdruck von oben ergibt sich dann als

```
prog = CompoundStm(AssignStm "a"
                             (OpExp (NumExp 5)
                                    Plus
                                    (NumExp 3)))
                  (...)
```

Die Lexikalische/Syntaktische Analyse ist dann eine Funktion *String* → *Stm*.

2 Programmiersprachen, Interpreter, Übersetzer

2.1 Interpreter

Programmiersprachen basieren auf einem Zeichenvorrat wie ASCII, wir definieren dann A als die Menge aller Zeichenketten über diesem Vorrat, z.B. $A = \text{ASCII}^*$ oder $A = \text{UNICODE}^*$.

Definition: Eine *Programmiersprache* L ist eine partielle Funktion[1] $L\colon A \dashrightarrow (A^* \dashrightarrow A)$. Der Definitionsbereich einer Programmiersprache ist die *Menge der syntaktisch zulässigen L-Programme* L-Prog. Sei $l \in L$-Prog. Dann ist $L(l)$ (im folgenden einfach Ll) eine Funktion $Ll\colon A^* \dashrightarrow A$ die *Ein-/Ausgabefunktion*, die *Semantik* von L.

Die syntaktische bzw. semantische Analyse dient dann der Erkennung, ob $l \in L$-Prog gilt. Üblich sind folgende Forderungen:

1. L-Prog $\subseteq A$ ist eine entscheidbare Teilmenge (d.h. die syntaktische und die semantische Analyse ist berechenbar)
2. $Ll\colon A^* \dashrightarrow A$ partiell rekursiv (Semantik von l implementierbar)
3. falls L eine Datenbankanfragesprache (z.B. $L = \text{SQL}$), dann muß meist auch Ll entscheidbar sein für alle $l \in L$-Prog.

Beachte: $l \in A$ alleine reine Syntax, d.h. $L_1 l \neq L_2 l$ für verschiedene Sprachen L_1, L_2.

Definition: Sei $l \in L$-Prog. Dann sei der Definitionsbereich

$$L\text{-Eingabe}(l) = \{\,(x_1, \ldots, x_n) \in A^* \mid Ll(x_1, \ldots, x_n) \in A\}$$

Weiterhin ist die Ausgabe definiert durch

$$\begin{aligned} L\text{-Ausgabe}(l) &= Ll(L\text{-Eingabe}(l)) \\ &= \{\,x \in A \mid \exists\,(x_1, \ldots, x_n) \in A^*\colon Ll(x_1, \ldots, x_n) = x\} \end{aligned}$$

Definition: Ein *Interpreter* i für eine Sprache L, geschrieben in Sprache I, ist ein Programm $i \in I$-Prog, was für alle $l \in L$-Prog und alle $(x_1, \ldots, x_n) \in L$-Eingabe(l) folgendes erfüllt:

$$Ii(l, x_1, \ldots, x_n) = Ll(x_1, \ldots, x_n)$$

[1] partielle Funktionen sind hier notiert mit „$\dashrightarrow$“

Als Symbol[2] sei $[\frac{L}{I}]$ die Menge aller Interpreter für L geschrieben in I. Obige Definition ist strikt, d.h. offen, falls Ll undefiniert oder Eingaben unzulässig ist. Im Beispiel aus Kapitel 1 ist $L = \mathsf{Simple}$ und $I = \mathsf{Haskell}$ (siehe Übung).

Generelle **Vorteile und Nachteile von Interpretern**:

- Interpreter sind einfacher als Übersetzer, damit ist auch die Korrektheit einfacher nachweisbar (keine Zielsprache!)
- Interpreter meistens langsam, dafür aber geringe Übersetzungs-/Startzeit – daher gut zum Testen

Wünschenswert sind daher für eine Programmiersprache sowohl Interpreter als auch Übersetzer. Dabei eignen sich funktionale Sprachen gut für die Implementierung von Interpretern.

2.2 Übersetzer

Definition: Gegeben sei eine Quellsprache Q und eine Zielsprache Z. Ein C-Programm c heißt *Übersetzer von Q nach Z*, falls:

1. C-Eingabe$(c) = Q$-Prog
2. C-Ausgabe$(c) \subseteq Z$-Prog
3. Für alle $q \in Q$-Prog, $(x_1, \ldots, x_n) \in Q$-Eingabe(q)

$$Qq(x_1, \ldots, x_n) = Z(Cc(q))(x_1, \ldots, x_n)$$

Als Symbol[3] wird hier $\left[\begin{smallmatrix} Q \to Z \\ C \end{smallmatrix}\right]$ verwendet. **Beispiele**: $Q = \mathsf{Simple}$, $Z = \mathsf{C}$ (oder Maschinencode), $C = \mathsf{Haskell}$. Oder ein C++-Compiler: $Q = \mathsf{C{+}{+}}$, $Z = \mathsf{8086}$ und $C = \mathsf{8086}$. Es ist auch eine **Komposition** möglich, d.h. beispielsweise

$$\left[\begin{array}{c|c} \mathsf{Simple} \to \mathsf{C} & \mathsf{C} \to \mathsf{8086} \\ \mathsf{Haskell} & \mathsf{8086} \end{array}\right]$$

Zum **Unterschied** zwischen Interpretern und Übersetzern:

- Der Übersetzer „kennt“ das Quellprogramm, beispielsweise ist einem Simple-Übersetzer die Menge aller Identifier bekannt und kann ein Feld statt einer Liste für die Umgebung verwenden, er übersetzt dann die Zuweisung durch indizierte Feldmodifikation (konstante Laufzeit!).
- Ein Compiler hat die Möglichkeit, Optimierungen im Programmablauf vorzunehmen.

[2] Original-Symbol siehe Anhang A
[3] Original-Symbol siehe Anhang A

2.3 Konstruktion von Übersetzern aus Interpretern

Eine Methode zur Konstruktion von Compilern ist die **partielle Auswertung** des Interpreters. Gegeben sei also $i \in [{}^{L}_{I}]$ (d.h. ein L-Interpreter in I). Gesucht ist $c \in [{}^{L}_{C}\!\!\rightarrow^{I}]$, d.h. ein L-Übersetzer nach I, der in C geschrieben ist. **Idee**:

$$Ii\colon (l, x_1, \ldots, x_n) \mapsto Ll(x_1, \ldots, x_n)$$

D.h. für jedes $l \in L$-Prog wird i in ein Programm Restprog$(i, l) \in I$-Prog transformiert mit

$$I(\text{Restprog}(i, l))(x_1, \ldots, x_n) = Ii(l, x_1, \ldots, x_n)$$

Transformationsmethode: Setze in i konkreten Parameter l ein und werde dann i überall aus, wo es möglich ist (d.h. wo es von l abhängt), d.h. man wertet i an einigen Stellen aus (daher partielle Auswertung). Dann gilt: $Cc\colon l \mapsto \text{Restprog}(i, l)$. Mögliche Auswertungen:

- Ausrechnen von bekannten Werten: `3+2` $\rightarrow$ 5, `3==3` $\rightarrow$ `true`
- Bedingungen vereinfachen:

  $$\texttt{if true then } e_1 \texttt{ else } e_2 \;\rightarrow\; e_1$$
  $$\texttt{if false then } e_1 \texttt{ else } e_2 \;\rightarrow\; e_2$$

- „Auffalten“: Falls `f` x_1 `...`x_n `=` `b` ist:

  $$\texttt{f } e_1 \texttt{ ...}e_n \rightarrow \texttt{b[}x_1 \mapsto e_1\texttt{, ...,} x_n \mapsto e_n\texttt{]}$$

- Spezialisierung:

  $$\texttt{f } c_1 \texttt{ ...}c_k\ e_1 \texttt{ ...}e_n \rightarrow \texttt{f-c1-...-ck } e_1 \texttt{ ...}e_n$$

  mit einer neuen Funktion `f-c1-...-ck` x_1 `...`x_n `=` `b[`$y_1 \mapsto c_1$`, ...,` $y_k \mapsto c_k$`]` wobei `f` y_1 `...`y_k x_1 `...`x_n `=` `b`.

Beispiel: Seien folgende Funktionen definiert:

```
f x y = if x > 0 then 3*x+y else y
g z = f 4 z
```

Dann ergibt eine partielle Auswertung (hier durch Auffalten):

```
g z = if 4 > 0 then 3*4+z else z
    = if true then 3*4+z else z
    = 3*4+z
    = 12+z
```

Definition: Ein Programm $PEVAL \in E$-Prog
heißt *partieller Auswerter* für I-Programme, falls

1. E-Eingabe$(PEVAL) = I$-Prog $\times A$
2. $EPEVAL(i, x_1) = i_1 \in I$-Prog
3. $Ii_1(x_2, \ldots, x_n) = Ii(x_1, x_2, \ldots, x_n)$ für alle $(x_2, \ldots, x_n)$ mit $(x_1, \ldots, x_n) \in I$-Eingabe(i)

Damit gilt: $PEVAL$ angesetzt auf $i \in [{}^{L}_{I}]$ und $l \in l$-Prog liefert Zielprogramm in I:

$$\begin{aligned} I(EPEVAL(i,l))(x_1, \ldots, x_n) &= Ii(l, x_1, \ldots, x_n) \\ &= Ll(x_1, \ldots, x_n) \end{aligned}$$

Somit ist $EPEVAL(i)$ ein Übersetzer von L nach I in E. Praktische Durchführung von Kahn/Carlsson 1984 mit Ergebnis $\left[{}^{\mathsf{Prolog}\to\mathsf{Lisp}}_{\mathsf{Lisp}}\right]$, allerdings ein sehr langsamer Übersetzer (etwa Faktor 100 langsamer als ein handgeschriebener Übersetzer).

Verbesserung: Falls $E = I$ ist, so setze $PEVAL$ auf sich selbst und den Interpreter i an. Sei nun $c := IPEVAL(PEVAL, i)$. Dann ist $c \in I$-Prog, es sei $Ic(l) =: z \in I$-Prog mit

$$\begin{aligned} Iz(x_1, \ldots, x_n) &= I(Ic(l))(x_1, \ldots, x_n) \\ &= I(I(IPEVAL(PEVAL, i)(l)))(x_1, \ldots, x_n) \\ &= I(IPEVAL(i, l))(x_1, \ldots, x_n) \\ &= Ii(l, x_1, \ldots, x_n) \\ &= Ll(x_1, \ldots, x_n) \end{aligned}$$

Damit ist $c \in [{}^{L}_{I}\!\!\to^{I}]$. Nun wird der recht aufwendige partielle Auswerter nur noch für den Interpreter aufgerufen. In der Übung wird gezeigt, daß man einen *Compilergenerator* erhält durch:

$$cc := IPEVAL(PEVAL, PEVAL)$$

2.4 Kombination von Interpretern und Übersetzern

Häufig geschieht keine direkte Übersetzung in Maschinencode, sondern eine Kombination aus Interpretern und Übersetzern. Vorteile sind:

- Übersetzungskomplexität geringer

- Wiederverwendung
- mehr Portabilität

Beispiel: Haskell-Implementierung auf einer Maschine M durch Übersetzung nach C, dann:

$$\left[\begin{array}{c|c} \mathsf{Haskell} \rightarrow \mathsf{C} & \mathsf{C} \rightarrow \mathrm{M} \\ \mathrm{M} & \mathrm{M} \end{array}\right]$$

Das Problem ist es hier, einen Haskell-Compiler in M zu schreiben. Daher wird zum Beispiel ein Haskell-Complier in C geschrieben und dann der Compiler nach M übersetzt.

Eine weitere Kombination ist eine **abstrakte Maschine**, die eine Zwischenschicht der Übersetzung darstellt. Beispiele hier:

- WAM: Warren Abstract Machine (Prolog)
- STGM: Spinless Tagless G-Machine (Haskell)
- JVM: Java Virtual Machine (Java)

Eigenschaften sind die Orientierung an der Quellsprache, aber auch die Effizienz der Implementierung auf existierenden Maschinen (etwa durch Interpretation/Emulation oder durch einen weiteren Übersetzer). Der Ablauf beispielsweise bei Java: Der Code wird zu Byte-Code übersetzt auf JVM, danach dann Interpretiert oder mit einem „Just-in-time"-Compiler übersetzt. Vorteil ist, daß der übersetzte JVM-Code portabel ist – Nachteil ist wiederum die Effizienz. Schema hier:

$$\left[\begin{smallmatrix}\mathsf{Java}\rightarrow\mathsf{JVM}\\ \mathsf{JVM}\end{smallmatrix}\right] \quad \left[\begin{smallmatrix}\mathsf{JVM}\\ M\end{smallmatrix}\right]$$
$$\left[\begin{smallmatrix}\mathsf{JVM}\\ M\end{smallmatrix}\right]$$

2.5 Bootstrapping

Ziel ist ein Übersetzer $\left[\begin{smallmatrix}Q\rightarrow M\\ M\end{smallmatrix}\right]$. Problem ist, daß Q eine Hochsprache ist und M niedrig, d.h. ungeeignet für komplexe Software. Wünschenswert wäre es, den Übersetzer in Q zu schreiben, um die „Eleganz" von Q für die Implementierung zu nutzen und auch gleich die erste große Anwendung für den neuen Übersetzer. Realisierung durch **Bootstrapping** in mehreren Schritten:

1. Wähle eine **Teilsprache** T-$Q \subseteq Q$, die für einfachen Übersetzerbau geeignet ist und keine schwierig zu implementierenden Konstrukte enthält.

2. Schreibe einen Übersetzer $C_1 \in \left[{}^{T\text{-}Q\to M}_{A}\right]$ mit einem beliebigen ausführbaren A (d.h. es existiert ein Übersetzer $[{}^{A\to M}_{M}]$ oder einen Interpreter $[{}^{A}_{M}]$). C_1 muß korrekt, aber nicht effizient sein oder Fehlerbehandlung unterstützen etc.

3. Erhalte den T-Q-Übersetzer $\left[{}^{T\text{-}Q\to M}_{M}\right]$ durch Ansetzen von $[{}^{A\to M}_{M}]$ auf $\left[{}^{T\text{-}Q\to M}_{A}\right]$ (eventuell mittels Übersetzergenerierung, siehe oben)

4. Schreibe den „eigentlichen“ Übersetzer C_2 in T-Q.

5. Implementiere nun C_2 durch Ansetzen von $\left[{}^{T\text{-}Q\to M}_{M}\right]$ auf $\left[{}^{Q\to M}_{T\text{-}Q}\right]$. Dieser erzeugt **guten Code**, ist selbst jedoch **ineffizient**.

6. Wende nun C_2 auf den Sourcecode von C_2 selbst an und erhalte einen **effizienten Übersetzer** C_3, der auch effizienten Code erzeugt.

7. Eine weitere Selbstanwendung bringt nichts, aber trotzdem nützlich als **Praxistest**: Implementiere C_2 durch Anwendung von C_3 auf den Sourcecode von C_2 (bzw. C_3). Dieser erhaltene Übersetzer C_4 sollte **textuell identisch** sein zu C_3.

Man benötigt also zwei Dinge zum Bootstrap: Einen einfachen Übersetzer in einer existierenden Sprache und dann einen „guten“ Übersetzer für die Sprache Q in T-Q. **Vorteile:**

1. Bei allen Verbesserungen und Erweiterungen von C_2 ist nun der volle Sprachumfang von Q verwendbar, d.h. C_2' ist aus $\left[{}^{Q\to M}_{Q}\right]$.

2. Jede Verbesserung in der Codeerzeugung von C_2 kommt C_2 selbst zugute: Ein verbesserter Compiler aus $\left[{}^{Q\to M}_{M}\right]$ ist erzeugbar durch Anwendung des alten implementierten Compilers auf den verbesserten Compiler.

3. Der Compiler ist leichter wartbar, da er selbst in der Hochsprache Q geschrieben ist.

4. Die Methoden erlauben *Querübersetzer (Crosscompiler)* etwa für neue Rechnertypen.

3 Lexikalische Analyse

3.1 Ziel der lexikalischen Analyse

Beispiel: Betrachte folgendes Programm:

```
int abs(int x) /* compute absolute value */
{
   if (x > 0) return (x);
   else return (-x);
}
```

Das Einlesen des Programms erfolgt nun in zwei Phasen:

1. Die **lexikalische Analyse**: Ziel ist das Erkennen zusammengehöriger „Worte“ wie z.B. `int`, `abs`, `0` etc., aber auch das Überlesen von Kommentaren, Leerzeichen, Zeilenvorschüben etc.

2. Die **syntaktische Analyse**: Erkennung der Programmstruktur (Ausdrücke, `if`-Anweisungen, ...)

Ziel des Kapitels ist also die Zerlegung einer Zeichenfolge (Quellprogramm) in eine Folge von *Lexemen* (lexikalische atomare Einheiten), auch *Tokens* genannt. Zudem werden die Lexeme in *Symbolklassen* eingeteilt, d.h. in Mengen gleichwertiger Lexeme. Ein konkretes Lexem wird bei einer Symbolklasse mit mehreren Elementen als *Attribut* dargestellt.

Klasse	Elemente
Id	x, y, abc, zaehler1
Num	42, 0, 99
Real	0.3, 3.14, 2e-2
Int	int
If	if

Klasse	Elemente
LParen	(
RParen	)
LBrace	{
RBrace	}

Attribute sind darstellbar als Argumente der Klasse, z.B. `Id "abs"`, `Num 99` etc. Häufig gibt es weitere (implizite) Attribute wie Zeilen- und Spaltennummer des ersten Lexemzeichens. Ein Programm für die lexikalische Analyse heißt *Scanner (Zerhacker)*, formal: scan: $\Sigma^* \to T^*$ mit Eingabealphabet Σ und Tokenklassentyp T.

Beispiel: Der Ausdruck scan ⟨*obiger Programmstring*⟩ ergibt:

$$[\texttt{Int}, \texttt{Id "abs"}, \texttt{LParen}, \texttt{Int}, \texttt{Id "x"}, \texttt{RParen}, \texttt{LBrace}, \ldots]$$

Ein Scanner überliest Kommentare, Leerzeichen, ...(nicht bei javadoc, lint etc.) und unterscheidet zwischen *Schlüsselwörtern* wie `if`, `else`, ...und *Bezeichnern* (Klasse `Id`). Konsequenz ist, daß Schlüsselworte nicht als Bezeichner

verwendbar sind, beispielsweise ist eine Anweisung wie `int int; int = 3;` meist unzulässig.

3.2 Reguläre Ausdrücke

Beobachtung für die Implementierung: Die Struktur der Tokenklassen ist regulär, damit kann ein Scanner durch reguläre Ausdrücke und somit durch endliche Automaten beschrieben werden.

Definition: Sei Σ ein Alphabet. Die Menge $\mathrm{RA}(\Sigma)$ aller regulären Ausdrücke über Σ ist die kleinste Menge mit folgenen Eigenschaften:

- $\Sigma \cup \{\varepsilon, \lambda\} \subseteq \mathrm{RA}(\Sigma)$
- $\alpha, \beta \in \mathrm{RA}(\Sigma) \Longrightarrow \alpha|\beta \in \mathrm{RA}(\Sigma), \alpha \cdot \beta \in \mathrm{RA}(\Sigma), \alpha^* \in \mathrm{RA}(\Sigma)$

Die Semantik der Regulären Ausdrücke ist nun eine Funktion $[\![]\!]\colon \mathrm{RA}(\Sigma) \to 2^{\Sigma^*}$, definiert durch:

$$\begin{aligned}
[\![\lambda]\!] &= \emptyset \\
[\![\varepsilon]\!] &= \{\varepsilon\} \\
[\![a]\!] &= \{a\} \\
[\![\alpha|\beta]\!] &= [\![\alpha]\!] \cup [\![\beta]\!] \\
[\![\alpha \cdot \beta]\!] &= \{\, A \cdot B \mid A \in [\![\alpha]\!], B \in [\![\beta]\!] \} \\
[\![\alpha^*]\!] &= \{\, A_1; \ldots; A_n \mid A_1, \ldots, A_n \in [\![\alpha]\!], n \geq 0 \}
\end{aligned}$$

Weiter Abkürzungen, die in vielen RegularExpressions-Werkzeugen verwendet werden können:

- $[a - z]$ ist die Menge aller Zeichen zwischen a und z, also $a|b|c|\ldots$
- α^+ bedeutet $\alpha \cdot \alpha^*$

Beispiel:

$$\begin{aligned}
\texttt{Id} &= [a - z]([a - z]|[0 - 9])^* \\
\texttt{Num} &= [0 - 9]^+ \\
\texttt{If} &= i{\cdot}f
\end{aligned}$$

3.3 Implementierung

3.3.1 Übersetzung in einen endlichen Automaten

Definition: Ein *nichtdeterministischer endlicher Automat (NEA, NFA)* ist ein Tupel $A = (Q, \Sigma, \delta, q_0, F)$ mit Zuständen Q, Alphabet Σ, Startzustand $q_0 \in Q$, Endzustände $F \subseteq Q$ und einer Übergangsfunktion $\delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$. Probleme, die hier auftauchen:

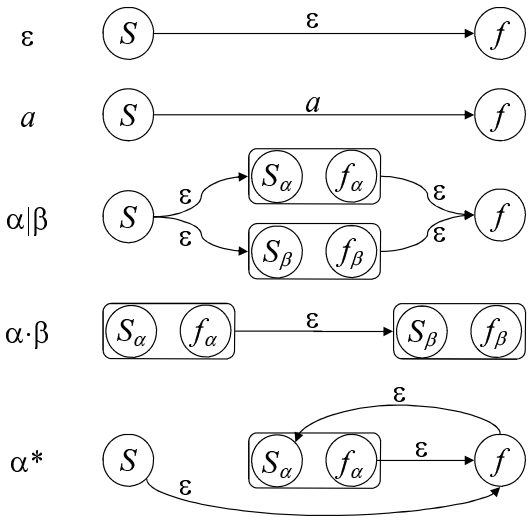
1. Ist `if` ein `Id` oder ein `If`?
2. Ist `ifxy` ein Lexem (`Id "ifxy"`) oder zwei (`If`, `Id "xy"`)?

Üblich:

zu 2. Benutze das *principle of longest match*: Der längste Präfix, der zu einer Symbolklasse gehört, ist das nächste Token. Damit ist `ifxy` ein Token, `if(` sind zwei Tokens.

zu 1. Die erste Symbolklasse, zu der ein longest-match-Token gehört, ist die Klasse dieses Tokens. Damit ist die Reihenfolge der Symbolklassen relevant, üblicherweise werden die Schlüsselwörter vor den Identifiern genannt.

Voraussetzung hier: Jeder NFA hat einen Start- und einen Endzustand (möglich mittels ε-Transitionen), übersetze dann die regulären Ausdrücke folgendermaßen in NFAs:



Problem: Ein Rechner kann nicht „raten“, daher ist ein NFA nicht zur Implementierung geeignet.

3.3.2 Übersetzung eines NFA in einen DFA

Lösung: Konvertiere den NFA in einen DFA (wie NFA, aber mit eindeutiger (partieller) Übergangsfunktion $\delta\colon Q \dashrightarrow Q$). Es fehlen jedoch die ε-Übergänge, repräsentiere eine Menge von NFA-Zuständen durch einen DFA-Zustand.

Definition: Sei ein NFA $(Q, \Sigma, \delta, q_0, F)$ gegeben und $T \subseteq Q$. Der ε-Abschluß $\hat{\varepsilon}(T)$ von T ist die kleinste Menge mit $T \subseteq \hat{\varepsilon}(T)$ und $q \in \hat{\varepsilon}(T) \Rightarrow \delta(q, \varepsilon) \subseteq \hat{\varepsilon}(T)$.

Algorithmus zum Berechnen des ε-Abschlusses:

```
S := T;
repeat
    S' := S;
    S := S ∪ ⋃_{q∈S}, δ(q, ε)
until S == S'
return S;
```

Der Algorithmus terminiert, da Q endlich ist und immer nur Zustände hinzugefügt werden. Nun kann man einen NFA deterministisch simulieren:

```
T := ε̂({q_0});
while not final(T)
    read c;
    T := ε̂(⋃_{q∈T} δ(q, c))
```

Hierbei ist `final(`T`) = (`$T \cap F \neq \emptyset$`)`. Die Berechnung des ε-Abschlußes ist jedoch aufwendig, besser ist eine Vorberechnung bei der Konstruktion des Übersetzers durch die *Potenzmengenkonstruktion*: Definiere zu einem NFA $(Q, \Sigma, \delta, q_0, F)$ einen DFA $(Q', \Sigma, \delta', q_0', F')$ durch

- $Q' = 2^Q$ (kann später reduziert werden)
- $F' = \{T \subseteq Q \mid T \cap F \neq \emptyset\}$
- $q_0' = \hat{\varepsilon}(\{q_0\})$
- $\delta'\colon Q' \times \Sigma \to Q'$ sei definiert durch $\delta'(T, a) = \hat{\varepsilon}\left(\bigcup_{q \in T} \delta(q, a)\right)$

Der konstruierte Automat enthält zu viele Zustände – diese Anzahl kann reduziert werden, indem nur die von q_0' erreichbaren Zustände aufgenommen werden und äquivalente Zustände verschmolzen werden (siehe Automatenminimierung).

3.3.3 Praktische Aspekte der Scannerimplementierung

- Schreibe an Endzustände erkannte Tokenklasse zur Scannerausgabe (bei mehreren Tokenklassen: die textuell erste, siehe oben)!
- Umsetzung des „longest match“: Merke letzten erfolgreichen Endzustand und Position; bei Nichtendzuständen ohne weiteren Übergang setze auf den letzten (gemerkten) Endzustand zurück.
- Bei vielen Schlüsselwörtern entstehen viele Zustände – Reduktion: Erkenne Schlüsselwörter genauso wie Bezeichner, falls der Scanner dann `Id(n)` erkennt, so prüfe, ob `n` ein Schlüsselwort ist: wenn ja, dann gib das entsprechende Token zurück, andernfalls `Id(n)`. Dieses Vorgehen ist vor allem empfehlenswert bei handgeschriebenen Scannern.
- Zur Implementierung des DFA: Entweder durch eine Tabelle (δ') und einen „Interpreter“ oder durch Übersetzung in `case`-Ausdrücke.
- Die **Scannerimplementierung** ist damit voll automatisierbar, entsprechend existieren fertige **Scanner-Generatoren** (z.B. lex), die reguläre Ausdrücke und zugehörige Symbolklassen übersetzen in ein Scanner-Programm.

4 Syntaktische Analyse

Aufgabe ist hier die Erkennung der Programmstruktur aus der Tokenfolge, wir brauchen also einen *Parser* für die syntaktische Analyse. Ausdrücke sind definiert durch:

```
Expr = num | id | Expr Op Expr | "(" Expr ")"
Op   = "+" | "*"
```

Bei Ausdrücken müssen z.B. die Klammerpaare ausgeglichen sein – daher muß der Parser „Klammern zählen“, das können DFAs nicht erreichen, d.h. Scanner sind nicht ausreichend. Daher erfolgt die Syntaxbeschreibung durch *kontextfreie Grammatiken (CFG)* und die Implementierung durch eine *Stackmaschine*.

Definition: Eine kontextfreie Grammatik $G = (N, T, P, S)$ besteht aus

- einer Menge N von Nichtterminalsymbolen $(A, B, C, \ldots)$
- einer Menge T von Terminalsymbolen $(a, b, c, \ldots)$
- einem Startsymbol $S \in N$
- einer Produktion P der Form $A \to \alpha_1\alpha_2 \ldots \alpha_n$ mit $n \geq 0, A \in N, \alpha_i \in N \cup T =: V$. Dabei heißt $\alpha_1\alpha_2 \ldots \alpha_n$ *Satzform*.

Die *Ableitungsrelation* ist nun eine Teilmenge von $V \times V$ mit

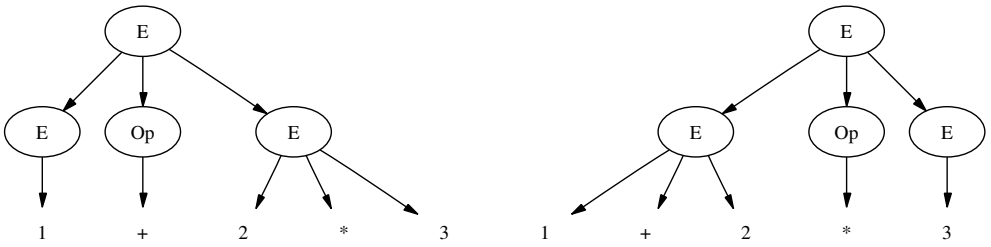
$$\alpha \to \beta \;:\Longleftrightarrow \alpha = \alpha_1 A \alpha_2, \beta = \alpha_1 \gamma \alpha_2, A \longrightarrow \gamma \in P$$

Falls $\alpha_1 \in T^*$, so nennt man die Ableitung eine *Linksableitung* (schreibe $\alpha \overset{L}{\Rightarrow} \beta$), von einer Rechtsableitung spricht man im Fall $\alpha_2 \in T^*$ (schreibe $\alpha \overset{R}{\Rightarrow} \beta$). Der reflexive, transitive Abschluß ist $\Rightarrow^*$. Die erzeugte Sprache ist nun $L(G) = \{ w \in T^* \mid S \Rightarrow^* w \}$.

Beispiel: Definiere Ausdrücke durch ein Startsymbol E und durch

$$\begin{aligned} T &= \{\texttt{num}, \texttt{id}, \texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}\} \\ N &= \{E, Op\} \\ P &= \{E \to \texttt{num}, E \to \texttt{id}, E \to E\,Op\,E, E \to \texttt{(}E\texttt{)}, Op \to \texttt{+}, Op \to \texttt{*}\} \end{aligned}$$

Stelle Ableitungen in der Grammatik nun dar durch einen *Ableitungsbaum*: Dieser repräsentiert die syntaktische Struktur eines Wortes. Die Wurzel enthält das Startsymbol, die Blätter nur Terminalsymbole, innere Knoten und Söhne entsprechen den angewendeten Produktionen. Beispiel für Ableitungsbäume:



Beide Bäume sind strukturell verschieden, das gleiche Wort `1+2*3` ist auf zwei Weisen darstellbar – welche Darstellung soll der Übersetzer verwenden?

Forderung: Das darf nicht vorkommen!

Definition: G ist *eindeutig*, wenn es für jedes $w \in L(G)$ genau einen Ableitungsbaum gibt, sonst: *mehrdeutig*.

Vermeidung durch Codierung von Bindungsstärken:

$$\begin{array}{lll} E \to E + T & T \to T * F & F \to \texttt{id} \\ E \to T & T \to F & F \to \texttt{num} \\ & & F \to \texttt{(}E\texttt{)} \end{array}$$

4.1 Recursive Descent Parsing

Für beliebige kontextfreie Grammatiken benötigt der (sehr gute!) CYK-Algorithmus (Coche, Younger, Kasami) schon $\mathcal{O}(n^2)$ Platz und $\mathcal{O}(n^3)$ Zeit. Für spezielle kontextfreie Grammatiken sind *recursive-descent parser (Parser mit rekusivem Abstieg)* besser, Idee:

- Der Parser arbeitet die Liste der Token von vorne ab.
- Terminalsymbole konsumieren von dieser Liste das entsprechende Token.
- Nichtterminale werden durch entsprechende Prozeduren repräsentiert, die dann die entsprechende Token konsumieren oder andere Funktionen aufrufen etc.
- Produktionen sind Fälle zur Definition der Nichtterminal-Prozeduren.

Beispiel: Grammatik für einfache Anweisungsfolgen:

$$\begin{array}{lcl} S & \to & id = num \\ S & \to & \texttt{print}\ id \\ S & \to & \texttt{begin}\ S\ L \\ L & \to & \texttt{end} \\ L & \to & \texttt{;}\ S\ L \end{array}$$

Annahme: Wir haben einen Scanner `scanner::String->[Token]` zur Verfügung, z.B. mit

```
data Token = Id | Eq | Num | Print | Begin | End | Semi
```

Der Parser ist nun eine Funktion, die als Eingabe eine Tokenliste benutzt (entspricht dem Eingabeprogramm) und eine Liste der restlichen Tokens (nicht zum gelesenen Satz gehörend) ausgibt. Damit gilt:

```
type Parser = [Token]->[Token]
```

Das Erkennen eines Terminalsymbols erfolgt nun durch

```
terminal::Token->Parser
terminal tok (t:ts) =
   if tok == t     -- erstes Token gleich Terminalsymbol?
      then ts      -- Resttoken als Resultat
      else error ("ERROR: wrong token ...")

parse_S::Parser    -- Parser für Nichtterminalsymbol S
parse_S (t:ts) = case t of
   Id    -> let ts1 = terminal Eq ts
                ts2 = terminal Num ts1
            in ts2
   Print -> terminal Id ts
   Begin -> let ts1 = parse_S ts
                ts2 = parse_L ts1
            in ts2

parse_L::Parser    -- Parser für Nichtterminalsymbol L
parse_L (t:ts) = case t of
   End   -> ts
   Semi  -> let ts1 = parse_S ts
                ts2 = parse_L ts1
            in ts2
```

Das Parsen eines Programmtextes `p::String` ist nun:

```
parse_S (scanner p) == []
```

Eine Folge von Symbolen wird in obigen Definitionen immer als Schachtelung von `let`s implementiert, das läßt sich verbessern durch einen *Sequenzoperator*: Durch `infixr 4 <*>` wird ein rechtsassoziativer (daher infix*r*) Infixoperator

der Bindungsstärke 4 eingeführt, der als Eingabe zwei Parser p_1 und p_2 erhält und einen Parser zurückgibt, der einen Parser zurückgibt, der die Sequenz von p_1 und p_2 darstellt.

```
infixr 4 <*>
(<*>)::Parser->Parser->Parser
p1 <*> p2 = \ts -> p2 (p1 ts)
```

Neue Definition der Parser von oben:

```
parse_S (t:ts) = case t of
   Id    -> (terminal Eq <*> terminal Num) ts
   Print -> terminal Id ts
   Begin -> (parse_S <*> parse_L) ts

parse_L (t:ts) = case t of
   End   -> ts
   Semi  -> (parse_S <*> parse_L) ts1
```

Dies entspricht genau den Grammatikregeln! Leider klappt diese einfache Entsprechung nicht immer, definiere beispielsweise Ausdrücke durch $E \to E + T$ und $E \to T$, dann entsteht nach obigem Schema

```
parse_E ts = case (head ts) of
    ? = (parse_E <*> terminal Plus <*> parse_T) ts
    ? = parse_T ts
```

Problem ist nun also die Entscheidung, welche Produktion angewendet werden muß! Wunsch ist also, daß diese Entscheidung nach dem Ansehen des ersten Tokens zu treffen ist, dann kann man einen einfachen Parser (von Hand) konstruieren. Andernfalls müssen mächtigere Parsergeneratoren (*LR-Parsing*, später) eingesetzt werden oder eine veränderte Grammatik verwendet werden.

4.1.1 LL(1)-Parser

LL(1) steht für *l*eft-to-right parsing with *l*eftmost derivation (Linksableitung) and *1*-symbol-lookahaed, entsprechend sind LL(k)-Parser mit $k > 1$ denkbar.

Idee ist: Berechne für alle rechten Seiten der Produktionen eines Nichtterminalsymbols die Menge möglicher erster Tokens: Falls diese Mengen disjunkt sind, liefert uns dies eine Entscheidungsgrundlage.

Definition: Sei $w \in T^*$ ein Wort. Dann sei

$$\text{start}_k(w) = \begin{cases} w & \text{falls} \quad |w| < k \\ u & \text{falls} \quad w = uv \text{ mit } |u| = k \end{cases}$$

Definition: Eine kontextfreie Grammatik $G = (N, T, P, S)$ heißt *LL(k)-Grammatik*, falls gilt: Sind die folgenden Linksableitungen mit $\text{start}_k(v) = \text{start}_k(w)$ möglich, so ist $\beta_1 = \beta_2$:

$$\begin{aligned} S &\xrightarrow{L}{}^{*} uA\alpha \xrightarrow{L} u\beta_1\alpha \longrightarrow^{*} uv \\ S &\xrightarrow{L}{}^{*} uA\alpha \xrightarrow{L} u\beta_2\alpha \longrightarrow^{*} uw \end{aligned}$$

Intuitiv: Falls u gelesen wurde, dann ist aufgrund der nächsten k Zeichen entscheidbar, welche Produktion anwendbar ist. Probleme sind aber:

- Die Entscheidung sollte unabhängig vom Kontext sein!
- Das Verfahren ist nicht konstruktiv: Wie entscheidet man effizient?

Weitere Einschränkung:

Definition: Eine kontextfreie Grammatik $G = (N, T, P, S)$ heißt *starke LL(k)-Grammatik*, falls gilt: Sind die folgenden Linksableitungen mit $\text{start}_k(v) = \text{start}_k(w)$ möglich, so ist $\beta_1 = \beta_2$:

$$\begin{aligned} S &\xrightarrow{L}{}^{*} u_1A\alpha_1 \xrightarrow{L} u_1\beta_1\alpha_1 \longrightarrow^{*} u_1v \\ S &\xrightarrow{L}{}^{*} u_2A\alpha_2 \xrightarrow{L} u_2\beta_2\alpha_2 \longrightarrow^{*} u_2w \end{aligned}$$

Unterschied zu *LL(k)*: Der Kontext spielt bei der Entscheidung keine Rolle!

Definition: Sei $G = (N, T, P, S)$ nun eine kontextfreie Grammatik, $\alpha \in (N \cup T)^*$ eine Satzform und $k > 0$. Dann sei folgendes die Menge aller aus α ableitbaren Anfangsfolgen von Terminalzeichen:

$$\text{FIRST}_k(\alpha) = \{\text{start}_k(w) \mid \alpha \longrightarrow^{*} w \in T^*\}$$

Falls diese zu kurz sind, dann schaue nach einem weiterem Token!

Definition: Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik, $A \in N$ und $k > 0$. Dann definiere folgende Menge, die alles enthält, was hinter A bei Ableitungen folgen kann:

$$\text{FOLLOW}_k(A) = \{w \in T^* \mid S \longrightarrow^{*} uAv \text{ und } w \in \text{FIRST}_k(v)\}$$

Definition: Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik, $(A \longrightarrow \alpha) \in P$. Führe folgende *Steuermenge (director set)* ein:

$$\text{D}_k(A \longrightarrow \alpha) = \text{start}_k(\text{FIRST}_k(\alpha) \cdot \text{FOLLOW}_k(A))$$

SATZ: Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik. Dann ist G eine starke LL(k)-Grammatik genau dann, wenn für zwei Produktionen $A \to \beta$ und $A \to \gamma \in P$ mit $\beta \neq \gamma$ gilt: $\mathrm{D}_k(A \to \beta) \cap \mathrm{D}_k(A \to \gamma) = \emptyset$.

Beweis: siehe [GÜTING, ERWIG 99, S. 60]

Somit sind zur Konstruktion eines Parsers folgende Schritte nötig:

- berechne die Steuermengen der Produktionen
- prüfe die Disjunktheit der Steuermengen
- beim Parsen: prüfe die nächsten k Zeichen auf Zugehörigkeit zu einer Steuermenge

In der Praxis beschränkt man sich häufig auf $k = 1$, um die Steuermengen nicht zu groß werden zu lassen. Verwende entsprechend in Zukunft $k = 1$ für FIRST_k und FOLLOW_k etc. Für $k = 1$ gilt zudem: Jede LL(1)-Grammatik ist auch starke LL(1)-Grammatik!

Kriterium für die Entscheidung, ob eine Grammatik LL(1)-Grammatik ist: Seien $A \to \beta$ und $A \to \gamma \in P$ mit $\beta \neq \gamma$. Es liegt eine LL(1)-Grammatik vor, falls

1. $\mathrm{FIRST}(\beta) \cap \mathrm{FIRST}(\gamma) = \emptyset$
2. Falls $\varepsilon \in \mathrm{FIRST}(\beta)$, dann $\mathrm{FOLLOW}(A) \cap \mathrm{FIRST}(\gamma) = \emptyset$

Damit können die Steuermengen einfacher definiert werden:

$$D(A \to \beta) = \begin{cases} \mathrm{FIRST}(\beta) & \text{falls} \quad \varepsilon \notin \mathrm{FIRST}(\beta) \\ (\mathrm{FIRST}(\beta) \setminus \varepsilon) \cup \mathrm{FOLLOW}(A) & \text{falls} \quad \varepsilon \in \mathrm{FIRST}(\beta) \end{cases}$$

Nun können wir einen Algorithmus zur Berechnung von FIRST und FOLLOW angeben, der in einer Iteration über die Regeln funktioniert; berechne zusätz-

lich $\text{leer}(A) \Leftrightarrow A \to^* \varepsilon$.

```
for each x ∈ N ∪ T
    FIRST(x) := ∅;
    FOLLOW(x) := ∅;
    leer(x) := false;

for each z ∈ T
    FIRST(z) = {z};

repeat
    for each A → α1 ... αk        (αi ∈ N ∪ T)
        if  leer(α1) ∧ ... ∧ leer(αk)
            leer(A) := true;
        for i = 1, ..., k
            if  leer(α1) ∧ ... ∧ leer(αi−1)
                FIRST(A) := FIRST(A) ∪ FIRST(αi);
            if  leer(αi+1) ∧ ... ∧ leer(αk)
                FOLLOW(αi) := FOLLOW(αi) ∪ FOLLOW(A);
            for j = i + 1, ..., k
                if  leer(αi+1) ∧ ... ∧ leer(αj−1)
                    FOLLOW(αi) := FOLLOW(αi) ∪ FIRST(αj);
until unchanged( FIRST, FOLLOW, leer )
```

Beispielgrammatik:

$$\begin{array}{lcl} Z & \to & d \\ Z & \to & XYZ \end{array} \qquad \begin{array}{lcl} Y & \to & \\ Y & \to & c \end{array} \qquad \begin{array}{lcl} X & \to & Y \\ X & \to & a \end{array}$$

0. Initialisierung:

0	leer	FIRST	FOLLOW
X	`false`	$\emptyset$	$\emptyset$
Y	`false`	$\emptyset$	$\emptyset$
Z	`false`	$\emptyset$	$\emptyset$

1. Iteration:

1	leer	FIRST	FOLLOW
X	`false`	$\{a\}$	$\{c, d\}$
Y	`true`	$\{c\}$	$\{d\}$
Z	`false`	$\{d\}$	$\emptyset$

2. Iteration

2	leer	FIRST	FOLLOW
X	`true`	$\{a, c\}$	$\{a, c, d\}$
Y	`true`	$\{c\}$	$\{a, c, d\}$
Z	`false`	$\{a, c, d\}$	$\emptyset$

Weitere Iterationen ändern an dieser Tabelle nichts mehr, der Algorithmus terminiert.

Nun können wir eine Parsing-Tabelle konstruieren. Unser RD-Parser entscheidet nach dem aktuellen Nichtterminal und dem nächsten Zeichen, welche Regel angewendet wird. Implementiert wird dies durch eine Tabelle $N \times T \to P$. Als Vorüberlegung gilt:

$$\text{FIRST}(\alpha_1 \dots \alpha_n) = \begin{cases} \text{FIRST}(\alpha_1) & \text{falls} \quad \neg\,\text{leer}(\alpha_1) \vee \alpha_1 \in T \\ \text{FIRST}(\alpha_1) \cup \text{FIRST}(\alpha_2, \dots, \alpha_n) & \text{falls} \quad \text{leer}(\alpha_1) \end{cases}$$

Nun zur Konstruktion dieser Tabelle: Sei $A \to \beta \in P$. Füge dann nach folgenden Regeln ein:

- Falls $t \in \text{FIRST}(\beta)$ ist, so füge $A \to \beta$ an (A, t) in die Tabelle ein.
- Falls $\text{leer}(\beta)$ gilt und $t \in \text{FOLLOW}(A)$, so füge $A \to \beta$ an (A, t) ein.

Sind in diesem Prozess keine Mehrfacheinträge in der Tabelle vorgenommen worden, so ist die Grammatik eine LL(1)-Grammatik und der Parser ist fertig! Im obigen Beispiel ergibt sich folgende Tabelle:

	a	c	d
X	$X \to a$ $X \to Y$	$X \to Y$	$X \to Y$
Y	$Y \to$	$Y \to$ $Y \to c$	$Y \to$
Z	$Z \to XYZ$	$Z \to XYZ$	$Z \to XYZ$ $T \to d$

Die Beispielgrammatik ist also keine LL(1)-Grammatik, sie ist mehrdeutig: Es gibt die Ableitung Regel $Z \to d$ und $Z \to XYZ \to YZ \to Z \to d$. Die Lösung besteht in einer Modifikation der Grammatik.

4.1.2 Modifikation zu LL-Grammatiken

- Elimination von **Linksrekursion**: Seien Regeln der Form $E \to E + T$ und $E \to T$ vorhanden. Ist $t \in \text{FIRST}(T)$, dann gilt auch $t \in \text{FIRST}(E+T)$, was sofort zu Mehrfacheinträgen in der Tabelle führt. Ursache ist die linksrekursive Regel, da die Regelanwendungen nicht mit beschränkter Vorausschau entscheidbar ist.

  Lösung: Transformation von Links- in Rechtsrekursion, im Beispiel ergeben sich die Regeln $E \to TE'$, $E' \to +TE'$ und $E' \to$.

  Allgemein: Gegen sei $X \to X\beta$ und $X \to \alpha$ (mit $\alpha \neq X$). Somit ist

$X \to^* \alpha\beta^*$, wobei die rechte Seite regulär ist und damit erzeugbar durch Rechtsrekursion, verwende dann Regeln $X \to \alpha X'$, $X' \to \beta X'$ und $X' \to$.

- Die LL-Analyse benötigt immer eine Vorausschau – das ergibt ein Problem, falls zwei Regeln mit gleichem Präfix beginnen, beispielsweise:

$$\begin{array}{lcl} S & \to & \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \\ S & \to & \texttt{if}\ E\ \texttt{then}\ S \end{array}$$

  Wende hier **Linksfaktorisierung** an: Ziehe den gemeinsamen Präfix heraus in eine Regel und führe ein neues Nichtterminal ein:

$$\begin{array}{lcl} S & \to & \texttt{if}\ E\ \texttt{then}\ SX \\ X & \to & \texttt{else}\ S \\ X & \to & \end{array}$$

4.1.3 Fehlerbehandlung in RD-Parsern

Bisher wurden beliebige kontextfreie Grammatiken (mehrdeutig) in eine eindeutige Grammatik verwandelt und zudem Linksrekursion verhindert. Neue kontextfreie Ausdrucks-Grammatik: Startsymbol S und ein „end-of-file"-marker \$:

$$\begin{array}{lcllcllcl} S & \to & E & T & \to & FT' & F & \to & \texttt{id} \\ E' & \to & TE' & T' & \to & *FT' & F & \to & \texttt{num} \\ E' & \to & +TE' & T' & \to & & F & \to & \texttt{(}E\texttt{)} \\ E' & \to & & & & & & & \end{array}$$

Dies ist eine LL(1)-Grammatik wegen folgender Parsing-Tabelle:

	+	*	id(num)	(	)	\$
S			$S \to E\$$	$S \to E\$$		
E			$E \to TE'$	$E \to TE'$		
E'	$E' \to +TE'$				$E' \to$	$E' \to$
T			$T \to FT'$	$T \to FT'$		
T'	$T' \to$	$T' \to *FT'$			$T' \to$	$T' \to$
F			$F \to$ id(num)	$F \to (E)$		

```
parse_T  ts = case (head ts) of
    Id     -> (parse_F <*> parse_T') ts
    Num    -> (parse_F <*> parse_T') ts
```

```
    LParen -> (parse_F <*> parse_T') ts
    _      -> error "T"

parse_T' ts = case (head ts) of
    Plus   -> ts
    Times  -> (terminal Times <*> parse_F <*> parse_T') ts
    RParen -> ts
    EOF    -> ts
    _      -> error "T'"
```

Der Aufruf `error` weist auf einen Syntaxfehler hin – aber was soll man an diesen Stellen machen?

- Fehlermeldung und Abbruch ist benutzerunfreundlich!
- Fehlermeldung, Reparatur des Fehler und weitermachen – durch Änderung der Eingabe zu korrektem Satz durch
  - Löschen eines Tokens
  - Ersetzen eines Tokens durch ein anders
  - Einfügen eines neuen Tokens

Beispiel: Bei `(error "T")` `Id`- oder `Num`-Token einfügen: nicht real, sondern „virtuell“ durch entsprechende Aktionen. Hier: verlasse `parse_T`, ohne weiterzulesen:

```
(error "T") -> trace "Expected id, num or left paren" ts
```

Einfügen ist jedoch gefährlich: eventuell terminiert die Fehlerbehandlung nicht! Sicherer ist es, Token zu überspringen (Löschen) bis zu einem sinnvollen Punkt, d.h. bis das nächste Token zur FOLLOW-Menge des Nichtterminals gehört.

Beispiel: Es ist FOLLOW$(T') = \{), +, \$\}$, benutze nun

```
error "T" -> trace
   "expected +, *, right paren or end-of-file"
   (skipto [RParen, Plus, EOF] ts)

skipto stopset (t:ts) =
   if t `elem` stopset then t:ts
                       else skipto stopset ts
```

Problem bei der Fehlerkorrektur ist das Erzeugen von Folgefehlern, die durch die Korrektur entstehen – es kann zu Kaskaden von Fehler kommen nur durch ein falsches Token.

4.2 Bottom-Up-Analyse

4.2.1 LR(k)-Parser

Idee: Konstruiere Ableitungsbaum von unten nach oben, am weitesten verbreitet[4] ist die *LR(k)-Analyse* (*l*eft-to-right parsing with *r*ightmost derivation with k symbols look-ahead). Dies ist die mächtigere Parsing-Technik, sie ist aber auch komplexer – eingesetzt wird sie u.a. in Parsergeneratoren wie z.B. Yacc.

Idee: Der Parser liest sowohl von der Eingabe als auch von einem Stack, letzterer dient der Speicherung von Eingabeteilen, die noch nicht verarbeitet sind. Aus den nächsten k Eingabezeichen und dem Stackinhalt entscheidet der Parser, ob er

- *shift* ausführt: nächstes Eingabezeichen auf den Stack legen oder
- *reduce* ausführt: den Stackinhalt vereinfachen

Daher heißen diese Parser auch *shift-reduce-Parser*. Die Reduzierung des Stacks geschieht in mehreren Schritten:

- Wähle eine Regel $X \to \alpha\beta\gamma$ aus.
- Lösche γ, β und α (in dieser Reihenfolge nacheinander) vom Stack.
- Speichere X auf den Stack.

Damit besteht der Stackinhalt im Wesentlichen aus Nichtterminal- und Terminalzeichen.

Beispiel: Betrachte die eindeutige, aber nicht umgewandelte Ausdrucksgrammatik:

$$\begin{array}{lll} E \to E + T & T \to T * F & F \to \texttt{id} \\ E \to T & T \to F & F \to \texttt{num} \\ & & F \to \texttt{(}E\texttt{)} \end{array}$$

Eingabe sei z.B. `id + id * id`, dann ist der Ablauf folgendermaßen:

[4] alternativ: Operator-Präzedenz-Analyse

Stack	Eingabe	Aktion
	`id + id * id`	*shift*
`id`	`+ id * id`	*reduce*
F	`+ id * id`	*reduce*
T	`+ id * id`	*reduce*
E	`+ id * id`	*shift*
E+	`id * id`	*shift*
E+`id`	`* id`	*reduce*
E+F	`* id`	*reduce*
E+T	`* id`	*shift*
E+T*	`id`	*shift*
E+T* `id`	ε	*reduce*
E+T*F	ε	*reduce*
E+T	ε	*reduce*
E	ε	*accept*

Der Vorgang *reduce* entspricht also einer Regelanwendung. Der Parser wendet diese Regeln bottom-up an, d.h. die Umkehrung der *reduce*-Aktionen ergibt folgende **Rechtsableitung**:

$$E \longrightarrow E + T \longrightarrow E + T * F \longrightarrow E + T * \texttt{id} \longrightarrow E + F * \texttt{id} \longrightarrow^* \texttt{id} + \texttt{id} * \texttt{id}$$

Das **Problem** ist nun, wie der Parser seine Aktion (*shift* oder *reduce*) aussucht. **Lösung:** Lese den gesamten Stackinhalt mit einem endlichen Automaten und entscheide je nach Zustand des Automaten am Ende des Lesens. Somit hat der Stack die Form $S_0\alpha_1S_1\alpha_2S_2 \ldots S_{n-1}\alpha_nS_n$ mit $\alpha_i \in N \cup T$ und S_i als Zustände des Automaten (eigentlich sind die α_i nun überflüssig, aber wir nehmen sie zur bessern Übersicht auf).

Die Aktionen des Parsers werden durch eine *LR-Parsing-Tabelle* mit folgenden Einträgen definiert:

- $s\,n$ – *shift* und gehe in Zustand n
- $g\,n$ – gehe in Zustand n
- $r\,k$ – *reduce* mit Regel k
- a – *accept*, d.h. die Eingabe wurde vollständig erfolgreich erkannt
- ε (leerer Eintrag) – Syntaxfehler

Dabei entsprechen die `s`/`g`-Einträge den Kanten des Automaten: `s` bei Terminalsymbolen und `g` bei Nichtterminalen.

- *Beispiel:* Einfache Grammatik (wobei das neue Startsymbol nur auf das alte Startsymbol und den End-of-file-Marker \$ verweist):

$$\begin{array}{llll} (0) & S' & \to & S\$ \\ (1) & S & \to & (L) \\ (2) & S & \to & x \end{array} \qquad \begin{array}{llll} (3) & L & \to & S \\ (4) & L & \to & L, S \end{array}$$

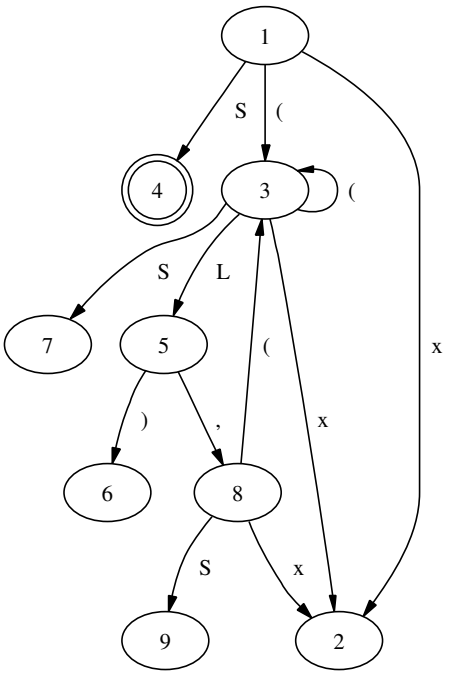
Beachte: Die Grammatik ist linkrsrekursiv und damit nicht LL(k)! Die LR-Parsing-Tabelle hat die Form Zustand $\times$ $(N \cup T \cup \{\$\}) \rightarrow$ Aktion:

	shift/reduce					*goto*	
	(	)	x	,	\$	S	L
1	$s3$		$s2$			$g4$	
2	$r2$	$r2$	$r2$	$r2$	$r2$		
3	$s3$		$s2$			$g7$	$g5$
4					a		
5		$s6$		$s8$			
6	$r1$	$r1$	$r1$	$r1$	$r1$		
7	$r3$	$r3$	$r3$	$r3$	$r3$		
8	$s3$		$s2$			$g9$	
9	$r4$	$r4$	$r4$	$r4$	$r4$		

Der Startzustand entspricht dem ersten Stackelement. Aktionen bei *reduce*:

- Lösche oberste Stackelemente (entsprechend rechter Regelseiten)
- lege linkes Nichtterminalsymbole auf den stack
- Bereche neuen Zustand aus oberstem Zustand (nach entfernen) mit goto-Tabelle

Die Tabelle entspricht dabei dem folgenden Automaten:



Beachte: Zustände ohne Übergänge (Senken) sind entweder der Endzustand oder *reduce*-Zustände. Mit der obigen Tabelle arbeitet der Parser wie folgt auf der Eingabe „`(x,x)$`“:

Stack	Eingabe	Aktion
1	`(x,x)$`	$s3$
1`(`3	`x,x)$`	$s2$
1`(`3`x`2	`,x)$`	$r2$
1`(`3S7	`,x)$`	$r3$
1`(`3L5	`,x)$`	$s8$
1`(`3L5`,`8	`x)$`	$s2$
1`(`3L5`,`8`x`2	`)$`	$r2$
1`(`3L5`,`8S9	`)$`	$r4$
1`(`3L5	`)$`	$s6$
1`(`3L5`)`6	`$`	$r1$
1S4	`$`	a

Dies entspricht der Ableitung

$$S \longrightarrow (L) \longrightarrow (L, S) \longrightarrow (L, x) \longrightarrow (S, x) \longrightarrow (x, x)$$

Frage: Wie erhält man die LR-Parsing-Tabelle? Konstruiere einen Automaten mit k Symbolen Vorausschau:

- Für $k \geq 2$ entstehen sehr große Parsingtabellen, praktisch nicht relevant.
- In der Praxis sind alle sinnvollen Programmiersprachen mit $k = 1$ beschreibbar durch LR(1)-Grammatiken.
- Ein verständliches, aber sehr eingeschränktes Prinzip ist $k = 0$.

Zunächst betrachten wir hier die Konstruktion von LR(0)-Parsern.

4.2.2 LR(0)-Parser

Im obigen Beispiel: Stack und Eingabe entspricht immer einer Satzform der Rechtsableitung. **Idee:** Jeder Zustand entspricht Regeln mit Markierung, wie weit schon gelesen wurde. Produktionen mit solchen Positionsmarkierungen heißen LR(0)-*Elemente* (*items*).

Definition: Sei $A \to \beta\gamma \in P$ eine Produktion mit $\beta\gamma \in (N \cup T)^*$. Dann heißt $A \to \beta.\gamma$ ein LR(0)-*Element* der Grammatik.

Beachte: $\beta = \varepsilon$ und $\gamma = \varepsilon$ sind zulässig. Somit gehören zur Regel $S \to (L)$ die vier Elemente $S \to .(L)$, $S \to (.L)$, $S \to (L.)$ und $S \to (L).$. Die

Automatenzustände sind nun Mengen von LR(0)-Elementen, die die Ableitungsinformation beschreiben. Der Startzustand enthält $S' \to .S\$$ („noch nichts gelesen“). Die passende Eingabe dazu ist alles, was aus $S\$$ ableitbar ist, d.h. alles, was aus S ableitbar ist, ist auch möglicher Anfang der Eingabe. Nehme daher alle S-Regeln zum Zustand hinzu, der Startzustand (1) entspricht daher

$$S' \to .S\$ \qquad S \to .(L) \qquad S \to .x$$

Definition:

1. Sei M eine Menge von LR(0)-Elementen. Dann ist der *Abschluß* (*closure*) von M die kleinste Menge mit
   (a) $M \subseteq \text{closure}(M)$
   (b) Ist $A \to \alpha.B\beta \in \text{closure}(M)$ und $B \to \gamma$ Produktion, so ist $B \to .\gamma \in \text{closure}(M)$
2. Ein LR(0)-*Zustand* ist eine abgeschlossene Menge von LR(0)-Elementen.

Die Berechnung ist trivial mittels Iteration. Der Startzustand ist der Abschluß von $S' \to S\$$. Die Berechnung der Zustandsübergänge erfolgt nun durch Verschieben der Markierung um ein Symbol nach rechts. Der Übergang von Zustand (1) unter Symbol x ist der Abschluß von $S \to x.$, dies ergibt Zustand (2). Der Zustand (3) entsteht durch Übergang von (1) unter (:

$$\begin{array}{lll} S \to (.L) & L \to .S & S \to .(L) \\ & L \to .L, S & S \to .x \end{array}$$

Definition: Sei M ein LR(0)-Zustand und $X \in N \cup T$. Dann ist

$$\text{goto}(M, X) := \text{closure}(\{\, A \to \alpha X.\beta \mid A \to \alpha.X\beta \in M\})$$

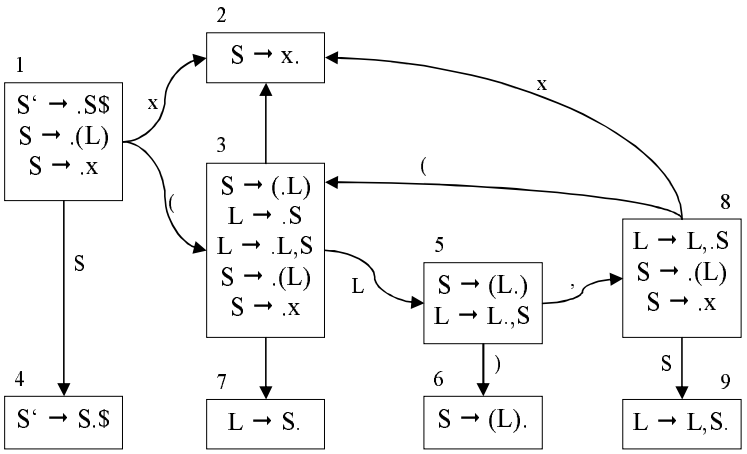
Daraus läßt sich die Parsingtabelle konstruieren: Die *reduce*-Zustände erhält man aus LR(0)-Zuständen, die abgearbeiteten Produktionen entsprechen, d.h. Elemente der Form $A \to \beta.$ enthalten.

Berechnung des LR(0)-Automaten (mit Zuständen/Knoten T und beschrifte-

ten Kanten E)

$T := \{\text{closure}(\{S' \to .S\$\})\};$
$E := \emptyset$
`repeat`
  `for each` $M \in T$
    `for each` $A \to \alpha.X\beta \in M$
      $N := \text{goto}(M, X)$
      $T := T \cup \{N\}$
      $E := E \cup \{M \xrightarrow{X} N\}$
`until` T, E `unchanged`

Der für die obige Grammatik berechnete Automat ergibt sich als (wobei vom Zustand 4 aus das $ nicht weiter verfolgt wird):



Nun läßt sich die LR(0)-Parsing-Tabelle aufstellen: Numeriere alle LR(0)-Zustände des Automaten (z.B. 1 für Startzustand) und stelle die Parsing-Tabelle auf:

$$\text{Zustaende} \times (N \cup T \cup \{\$\}) \longrightarrow \text{Aktion}$$

- $i \xrightarrow{X} j$ mit $X \in T$ – *shift*-Aktion: Trage Aktion „$s\,j$“ an Stelle (i, X) ein.
- $i \xrightarrow{X} j$ mit $X \in N$ – *goto*-Aktion: Trage Aktion „$g\,j$“ an Stelle (i, X) ein.
- Sei i ein Zustand mit Element $A \to \beta.$ (wobei $A \to \beta$ die n-te Produktion sei): Trage „$r\,n$“ an allen Stellen (i, X) mit $X \in T$ ein.
- Sei i ein Zustand mit Element $S' \to S.\$$: Trage „a“ an Stelle $(i, \$)$ ein.

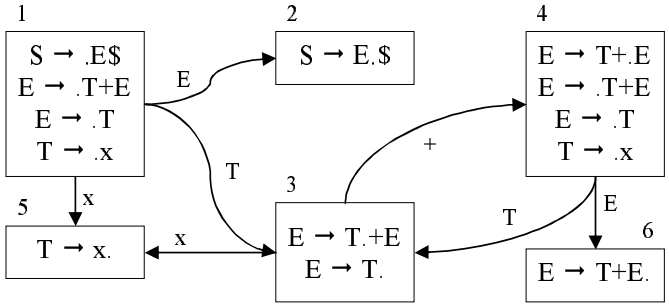
Die Parsing-Tabelle für obiges Beispiel ist oben schon angegeben. Falls die Tabelle keine mehrfachen Einträge enthält, so liegt eine *LR(0)-Grammatik* vor.

4.2.3 SLR(1)-Parser

Bei LR(0) benutzt man keine Vorausschau bei der Konstruktion: *reduce* ist unabhängig vom nächsten Zeichen! Dadurch entstehen manchmal vermeidbare Konflikte, betrachte beispielsweise folgende Grammatik:

$$\begin{array}{llllll} (0) & S & \to & E\$ & (1) & E \to T+E \\ (2) & E & \to & T & (3) & T \to x \end{array}$$

Der LR(0)-Automat ergibt sich als:



Die LR(0)-Tabelle enthält nun einen *shift-reduce-Konflikt* bei $(3, +)$:

	x	$+$	\$	E	T
1	$s5$			$g2$	$g3$
2			a		
3	$r2$	**s4/r2**	$r2$		
4	$s5$			$g6$	$g3$
5	$r3$	$r3$	$r3$		
6	$r1$	$r1$	$r1$		

Eine **Verbesserung** liefert die *SLR(k)-Konstruktion (Simple LR)*: Die Konstruktion erfolgt wie bei LR(0), jedoch wird bei *reduce*-Einträgen nur reduziert, falls die nächsten k Eingabezeichen hinter den reduzierten Nichtterminal tatsächlich vorkommen können – verwende wiederum die FOLLOW-Mengen! Hier verwenden wir $k = 1$, die SLR-Parsing-Tabelle ist dann wie $LR(0)$ mit der folgenden Änderung:

- Sei i ein Zustand mit Element $A \to \beta$. (wobei $A \to \beta$ die n-te Produktion sei): Trage „$r\,n$“ an allen Stellen (i, X) mit $X \in \text{FOLLOW}(A)$ ein.

Falls die Tabelle nun konfliktfrei ist, so liegt eine *SLR(1)-Grammatik* vor. Beispielsweise ist hier

$$\text{FOLLOW}(E) = \{\$\} \qquad \text{FOLLOW}(T) = \{+, \$\}$$

Damit ergibt sich die folgende konfliktfreie SLR(1)-Tabelle:

	x	$+$	\$	E	T
1	$s5$			$g2$	$g3$
2			a		
3		$s4$	$r2$		
4	$s5$			$g6$	$g3$
5		$r3$	$r3$		
6			$r1$		

Vorteile von SLR-Parsern:

- kleine Zustandsmengen (damit auch kleine Tabellen, vor allem im Vergleich zu LR(1))
- viele Programmiersprachen sind SLR(1)

4.2.4 LR(1)-Parser

Ein LR(0)-Zustand enthält eventuell Elemente (und damit auch Konflikte), die in realen Ableitungen nicht vorkommen können. Verbesserung ist hier, die nächsten k Eingabezeichen hinter der rechten Seite einer Produktion.

Definition: Ein *LR(1)-Element* hat die Form $(A \to \beta.\gamma, x)$ mit $A \to \beta\gamma \in P$ und $x \in T \cup \{\$\}$. Dabei wird $A \to \beta.\gamma$ als *Kern* des Elements bezeichnet. Intuitiv: Ein Element $(A \to \beta.\gamma, x)$ entspricht der Situation, daß β oben auf dem Stack ist und γx ableitbar ist zum Anfang der Resteingabe.

Parser der Form LR(k) mit $k > 1$ werden praktisch nicht benutzt, da die Zustandsmengen dann zu groß werden und die meisten Programmiersprachen eine LR(1)-Grammatik haben. Die **Konstruktion** eines LR(1)-Parsers erfolgt wie bei LR(0), nur erfolgt die Berechnung mit LR(1)-Elementen.

Definition: Sei M eine Menge von LR(1)-Elementen. Dann ist der *Abschluß* (*closure*) von M die kleinste Menge mit

1. $M \subseteq \text{closure}(M)$
2. Ist $(A \to \alpha.B\beta, x) \in \text{closure}(M)$ und $B \to \gamma$ Produktion sowie $w \in \text{FIRST}(\beta x)$, so ist $(B \to .\gamma, w) \in \text{closure}(M)$

Neu ist die Berücksichtigung möglicher Eingabezeichen hinter Produktionen.

Definition: Sei M eine Menge von LR(1)-Elementen und $X \in N \cup T$. Definiere dann folgende Menge:

$$\text{goto}(M, X) := \text{closure}(\{ (A \to \alpha X.\beta, w) \mid (A \to \alpha.X\beta, w) \in M\})$$

Der neue Startzustand des LR(1)-Automaten ist nun closure($\{(S' \to .S\$, ?)\}$), wobei das ? beliebig ist, da man nie über das \$ liest. Somit kann nun der LR(1)-Automat berechnet werden wie der LR(0)-Automat, aber mit LR(1)-Zuständen. Aus diesem Automaten wird dann – wie bei LR(0) – die LR(1)-Parsing-Tabelle erzeugt, einen Unterschied gibt es nur bei *reduce*-Einträgen:

- Sei i ein LR(1)-Zustand mit Element $(A \to \beta., x)$ (wobei $A \to \beta$ die n-te Produktion sei): Trage „$r\,n$“ an der Stelle (i, x) ein.

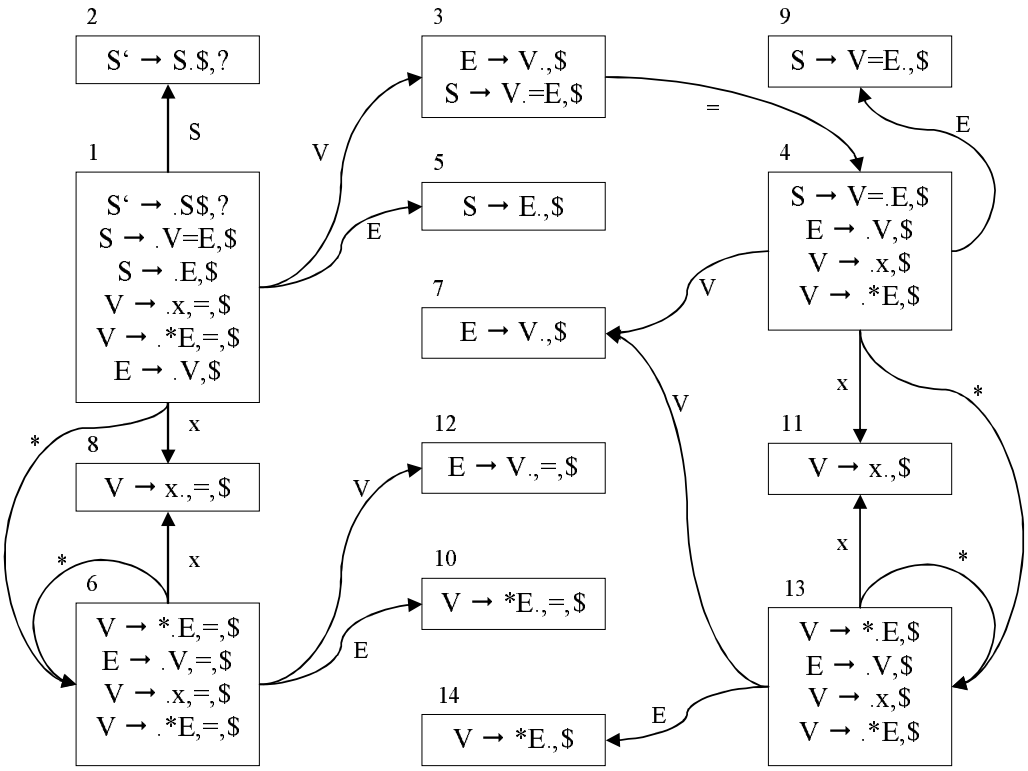
Es ergeben sich hieraus nun präzisere *reduce*-Aktionen, d.h. weniger *shift-reduce*- und *reduce-reduce*-Konflikte. Eine Grammatik wird nun als *LR(1)-Grammatik* bezeichnet, wenn die LR(1)-Tabelle konfliktfrei ist.

Beispiel: Eine Grammatik für Ausdrücke und Dereferenzierungen wie in C:

$$\begin{array}{lcllcl} S' & \to & S\$ & E & \to & V \\ S & \to & V = E & V & \to & x \\ S & \to & E & V & \to & *E \end{array}$$

Nun ergibt sich der folgende LR(1)-Automat:[5]

[5] Zum Vereinfachen der Schreibweise werden LR(1)-Elemente mit gleichem Kern zusammengefaßt!



Daraus erhalten wir die folgende LR(1)-Parsingtabelle (analog zu LR(0)), nur bei *recude*-Aktion Berücksichtigung der Vorausschau:

	x	$*$	$=$	\$	S	E	V
1	$s8$	$s6$			$g2$	$g5$	$g3$
2				a			
3			$s4$	$r3$			
4	$s11$	$s13$				$g9$	$g7$
5				$r2$			
6	$s8$	$s6$		$r3$		$g10$	$g12$
7			$r4$	$r4$			
8				$r1$			
9			$r5$	$r5$			
10				$r4$			
11			$r3$	$r3$			
12	$s11$	$s13$					
13						$g14$	$g7$
14				$r5$			

Schon bei dieser kleinen Grammatik haben wir 14 LR(1)-Zustände, in der Regel können LR(1)-Tabellen sehr groß werden durch die Kombination von

Regeln und Vorausschau. Daher versucht man in der Praxis, die Tabelle zu verkleinern. Methode hierzu ist *Look-Ahead-LR(1) (LALR(1))*.

4.2.5 LALR(1)-Parser

Idee: Verschmelze Zustände, deren Elemente sich nur in den Vorausschau-Zeichen, aber nicht im LR(0)-Kern unterscheiden. Im obigen Beispiel lassen sich folgende Zustandspaare gleiche Kernmengen und lassen sich daher zusammenfassen: (6, 13), (7, 12), (8, 11) und (10, 14). Bilde nun die Vereinigung der Zustände, daraus ergibt sich direkt ein verkleinerter Automat, da alle *goto*-Übergänge eines vereinigten Zustands in einen vereinigten Zustand führen. Konstruiere hieraus eine Parsertabelle wie bei LR(1).

Anmerkung: Hierdurch können im Vergleich zu der LR(1)-Tabelle nur neue *reduce-reduce*-Konflikte entstehen, dieses tritt in der Praxis aber selten auf. Eine Grammatik heißt nun LALR(1) genau dann, wenn die Tabelle nur eindeutige Einträge hat.

Die LALR(1)-Tabelle für obiges Beispiel sieht wie folgt aus:

	x	$*$	$=$	\$	S	E	V
1	$s8$	$s6$			$g2$	$g5$	$g3$
2				a			
3			$s4$	$r3$			
4	$s8$	$s6$				$g9$	$g7$
5				$r2$			
6	$s8$	$s6$				$g10$	$g7$
7			$r3$	$r3$			
8			$r4$	$r4$			
9				$r1$			
10			$r5$	$r5$			

Diese Grammatik ist also LALR(1) und die verkleinerte Tabelle reicht aus. In der Regel wird bei LR-Parsern nur die LALR-Methode verwendet, da sonst die Tabellen zu groß werden.

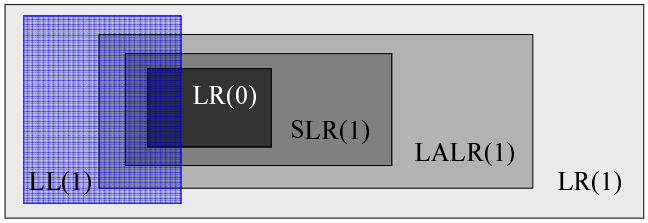
4.3 Klassifikation der Grammatiken und Sprachen

Um einen Eindruck der Mächtigkeit der verschiedenen Methoden zu erhalten, setzen wir einige Ergebnisse über die Klasse von Grammatiken an. Im folgenden Satz bezeichnen wir mit den Typnamen die Menge aller Grammatiken von diesem Typ:

SATZ: Es gilt:

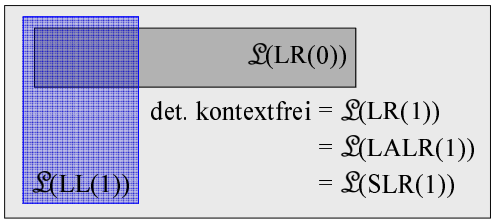
1. $\forall\, k \geq 0$ ist $\mathrm{LL}(k) \subsetneq \mathrm{LR}(k)$
2. $\mathrm{LR}(0) \subsetneq \mathrm{SLR}(1)$
3. $\mathrm{SLR}(1) \subsetneq \mathrm{LALR}(1)$
4. $\mathrm{LALR}(1) \subsetneq \mathrm{LR}(1)$

Graphisch:



Interessant ist auch folgende Aussage zur Mächtigkeit der LR(1)-Klasse:

Satz: Zu jeder deterministisch kontextfreien Sprache, d.h. eine, die mit einem deterministischen Kellerautomaten erkannt werden kann, existiert eine LR(1)-Grammatik, die diese erzeugt.



4.4 Parsergeneratoren

Die Konstruktion von Parsingtabellen ist einfach, aber aufwendig – d.h. gut automatisierbar. Nutze daher Parser-Generatoren, die als Eingabe eine Grammatik erhalten (plus „semantische Aktion“, später) und als Ausgabe ein Parser-Programm liefern.

Das Vorgehen eines Generators für LALR(1) ist nun: Berechne zunächst die Zustände (entweder LR(1) + Verschmelzung, vgl. Übung; oder auch durch direkte Berechnung, was effizienter, aber konzeptuell aufwendiger ist), berechne dann die Parsing-Tabelle und generiere daraus den Parser entweder als „Interpreter“ für die Parsingtabelle oder als spezielles Programm (case-Anweisungen über alle Zustände).

Es gibt sowohl Parsergeneratoren basierend auf $\mathrm{LL}(k)$ als auch für $\mathrm{LALR}(k)$,

meist nur $k = 1$. Im Folgenden verwenden wir Happy, eine Haskell-Variante des klassischen Yacc-Parsers für Unix; dieser basiert auf LALR(1). Eingabe ist eine Grammatikbeschreibung der Form:

```
<Modulkopf>
<Parserdeklaration>
%%
<Grammatik-Regeln>
<Modulende>
```

Hierbei sind Modulkopf und -ende optionaler Haskellcode, welcher um den generierten Parser gesetzt wird, z.B. der Code für einen Scanner, semantische Aktionen etc.). Der Modulkopf ist wie folgt aufgebaut:

```
{
module ...
import ...
}
```

Die Parserdeklaration beinhaltet allgemeine Deklarationen für den Parser, wie Liste der Terminalsymbole, Hauptparserfunktionen u.ä. Die Grammatikregeln haben die Form

```
A: B1 ... Bn {<Aktion>}
 | C1 ... Cm \ldots
```

Hierbei ist `A` ein Nichtterminalsymbol und die $\texttt{B}_i$s Nichtterminal- oder Terminalsymbole und `Aktion` eine semantische Aktion, die bei der *reduce*-Aktion dieser Regel ausgeführt wird.

Beispiel: Unsere Grammatik für Ausdrücke wird mit Happy wie folgt spezifiziert (z.B. in `expr.y`):

```
%name calc                -- name of created function
%tokentype { Token }      -- type of accepted tokens
                          --  => calc :: [Token]->t with t
                          --     attribute of first rule
%token '+' { PLUS }       -- terminals of grammar
       '*' { MULT }
       '(' { LPAREN }
       ')' { RPAREN }
       id  { ID }
       num { NUM }
%%
prg    : exp              { }
```

```
exp    : exp '+' term    { }
       | term            { }
term   : term '*' factor { }
       | faktor          { }
factor : id              { }
       | num             { }
       | '(' exp ')'     { }
{
happyError :: [Token]->a  -- always has to be defined
happyError _ = error "Parse error"
data Token = ID | NUM | PLUS | MULT | LPAREN | RPAREN
}
```

Token allgemein lassen sich angeben mittels Pattern, diese sind Haskell-Pattern des Tokentyps:

```
%token <name1> { <Pattern1> }
       <name2> { <Pattern2> }
```

Die Argumente dienen dazu, Werte des Scanners (z.B. Wert einer Zahl, Namen eines Identifiers) an die semantische Aktion zu übergeben. In diesem Fall wird die Notation `num { Num $$ }`. Nach Übersetzen mit Happy erhält man `exp.hs`, welche in der Funktion `calc` den Parser enthält. Dabei können folgende Konflikte auftreten:

- *shift-reduce*-Konflikt: Happy bevorzugt *shift* vor *reduce*
- *reduce-reduce*-Konflikt: Happy bevorzugt die textuell frühere Regel

Trotzdem kann man Konflikte durch Änderung der Grammatik eliminieren, betrachte Beispielsweise die Grammatik

$$\begin{aligned} S &\rightarrow \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \\ S &\rightarrow \texttt{if}\ E\ \texttt{then}\ S \\ S &\rightarrow \texttt{assign} \end{aligned}$$

Mehrdeutig ist die Ableitung von

$$\texttt{if}\ a\ \texttt{then}\ \underbrace{\texttt{if}\ b\ \texttt{then}\ S_1}\ \texttt{else}\ S_2$$

Üblich ist, daß `else` zum nächstmöglichen `then` gehört. Die LR-Parsingtabelle enthält jedoch zunächst einen *shift-reduce*-Konflikt:

$$\begin{aligned} S \rightarrow \texttt{if}\ E\ \texttt{then}\ S.\ \texttt{else}\ S \quad &\leftarrow \quad \textit{shift} \\ S \rightarrow \texttt{if}\ E\ \texttt{then}\ S. \quad &\rightarrow \quad \textit{reduce} \end{aligned}$$

Happy bevorzugt nun *shift*, was vernünftig ist, da es zur obigen gängigen Programmiersprachenkonvention passt. Besser ist eine modifizierte Grammatik: U entspricht *unmatched* (ohne `else`), während M *matched* entspricht. Die Grammatik lautet dann:

$$\begin{aligned} S &\rightarrow M \\ S &\rightarrow U \\ M &\rightarrow \texttt{if } E \texttt{ then } M \texttt{ else } M \\ M &\rightarrow \texttt{assign} \\ U &\rightarrow \texttt{if } E \texttt{ then } S \\ U &\rightarrow \texttt{if } E \texttt{ then } M \texttt{ else } U \end{aligned}$$

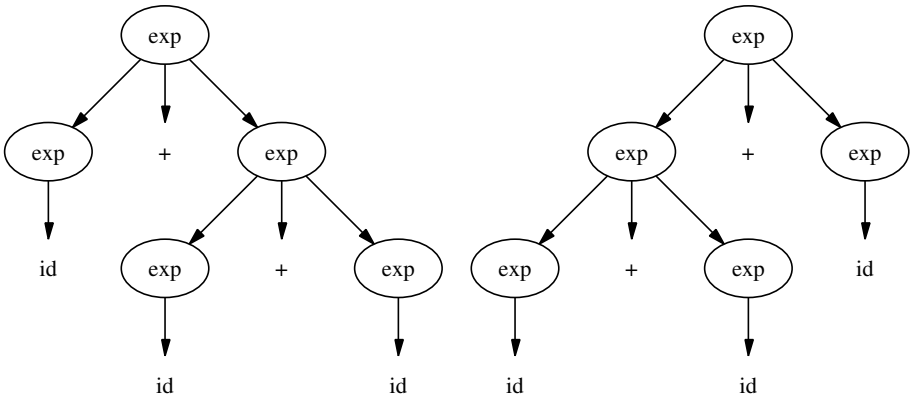
Beispiel: Betrachte eine mehrdeutige Grammatik für Ausdrücke:

```
exp : exp '+' exp  {}
    | id           {}
```

Nun kann es zu einem Konflikt kommen:

```
exp : exp. '+' exp  ←  shift
exp : exp '+' exp.  ←  reduce
```

Betrachte die Ableitungsbäume für `id + id + id`:



Links wird *shift* vor *reduce* benutzt, rechts *reduce* vor *shift*. Hier ist die Heuristik von Happy unpassend! Lösung: Entweder wieder die Grammatik modifizieren oder eine neue Parserdeklaration einfügen:

```
%left '+'
```

Dadurch wird das Terminalsymbol + linksassoziativ, d.h. der Konflikt wird durch ein *reduce* vor *shift* gelöst. Analog kann Potenzieren rechtsassoziativ gemacht werden, d.h. der Konflikt wird durch ein *shift* vor *reduce* gelöst.

```
%right '^'
```

Desweiteren kann man Operatoren als nicht-assoziativ deklarieren, dadurch wird ein Fehler ausgegeben, falls der Fall auftritt.

```
%nonassoc '='
```

Weitere Konflikte sind Prioritäten, betrache beispielsweise folgende Grammatik:

```
exp : exp '+' exp {}
    | exp '*' exp {}
    | id
```

Dann ist der Ausdruch `id + id * id` mehrdeutig! Intuitiv bindet `*` stärker als `+`, deklariere dies durch eine Deklarationssequenz, wobei die späteren Deklarationen stärker binden, insgesamt:

```
%nonassoc '='
%left '+'
%left '*'
%right '^'
...
exp : exp '+' exp {}
    | exp '*' exp {}
    | exp '=' exp {}
    | exp '^' exp {}
    | '(' exp ')' {}
    | id          {}
```

Nun meldet Happy keine Konflikte mehr!

5 Semantische Aktionen und Abstrakte Syntax

Nun beschäftigen wir uns mit der Codeerzeugung entsprechend der erkannten Syntax. Der erste Schritt dazu ist das Einfügen semantischer Aktionen in die Syntaxanalyse.

5.1 Semantische Aktionen

Idee ist, daß jedes Grammatiksymbol eine Bedeutung trägt. Bei den Terminalsymbolen wird hier der konkrete Wert der Symbolklassen benutzt (`Id`: String, `Num`: Zahlenwert), bei Nichtterminalsymbolen die Bedeutung des abgeleiteten Sprachkonstrukts – z.B. bei Ausdrücken der Wert, bei Anweisungen beispielsweise eine Transformation auf dem Speicher.

Definition: Eine *semantische Aktion* ist eine Anweisung oder ein Ausdruck (je nach imperativer oder funktionaler Sprache), die einer Syntaxregel zugeordnet wird und bei Anwendung den Wert des linken Nichttermzeichens berechnet.

5.1.1 Recursive Descent Parser

Ein rekursiver Abstiegsparser mit semantischen Aktionen arbeitet nun wie folgt:

- Erweitere die Terminalzeichen um Komponente mit Bedeutung:

  ```
  data Token = ID String | Num Int | Plus | Minus | ...
  ```

- Nichtterminale werden ja durch Funktionen repräsentiert, der Bedeutungswert entspricht dem Ergebnis der Funktion

Somit werden Nichtterminalfunktionen implementiert durch Aufruf der Symbole der rechten Seite und Berechnung der Bedeutung.

Beispiel: Berechnung des Werts von Binärzahlen, benutze Grammatik Grammatik

$$\mathit{digits} \rightarrow \mathit{digits}\,0 \mid \mathit{digits}\,1 \mid 0 \mid 1$$

Diese ist linksrekursiv, nach Transformation:

$$\mathit{digits} \rightarrow 0\,\mathit{digits} \mid 1\,\mathit{digits} \mid 0 \mid 1$$

Der semantische Wert soll nun der zugehörige Integer-Wert sein, implementiere nun einen RD-Parser mit semantischen Aktionen:

```
data Token = Bit0 | Bit1 | EOF
     deriving (Eq, Show)

exp2 n = if n == 0 then 1 else 2 \cdot exp2 (n-1)

-- semantischer Wert: Paar
--   (Anzahl digits, Wert der schon gelesenen digits)

digits (t:ts) = case t of
    Bit0 -> let (ts1,(l,  v         )) = digits ts
            in  (ts1,(l+1,v         ))
    Bit1 -> let (ts1,(l,  v         )) = digits ts
            in  (ts1,(l+1,v+(exp2 l)))
    EOF  -> (ts,(0,0))

parse ts = value where (_,(_,value)) = digits ts
```

5.1.2 Bottom-Up-Parser

Bei Bottom-Up-Parsern werden Terminalzeichen wie oben erweitert, aber auch die Nichtterminalzeichen erhalten eine Komponente für den semantischen Wert – speichere diese auf dem Stack, so daß dieser Symbole und semantische Werte enthält. Die Semantischen Aktionen werden also am Ende jeder Regel angegeben, siehe beispielsweise Happy:

```
%tokentype {Token}
%token '0' {Bit0}
       '1' {Bit1}
%%
digits : digits '0' { 2*$1   }
       | digits '1' { 2*$1+1 }
       | '0'        {      0 }
       | '1'        {      1 }
```

Semantische Aktionen werden in LR-Parsern bei **Reduktion** ausgeführt; dabei sind alle Symbole der rechten Seite dann auf dem Stack, d.h. deren Werte stehen zur Verfügung und mit der Notation `$`i kann auf den semantischen Wert des i-ten Symbols der rechten Seite zugegriffen werden. Dann wird der Wert des reduzierten Nichtterminals in den Klammern `{...}` definiert.

Die *reduce*-Aktion mit semantischen Aktionen wird nun erweitert zu:

1. Ersetze im Ausdruck der semantischen Aktion alle Symbole `$`i durch entsprechende Werte

2. Reduziere den Stack (wie früher)

3. Werte die semantische Aktion aus und speichere das Ergebnis bei oberstem Stackelement

Beispiel: Schreibe einen DeskCalculator: Berechnung von Ausdrücken in der schon früher angegebenen Grammatik; als semantischer Wert wird nun der Integer-Wert des Ausdrucks benutzt[6]

```
{ data Token = ID String | Num Int | PLUS | TIMES | LPAREN | RPAREN }
% token id {ID $$}
        num {Num $$}
        '+' {PLUS}
        ...
%%
prog   : exp             { $1    }
exp    : exp '+' term    { $1+$3 }
       | term            { $1    }
term   : term '*' factor { $1*$3 }
       | factor          { $1    }
factor : num             { $1    }
       | '(' exp ')'     { $2    }
```

Entsprechendes mit einem RD-Parser umzusetzen wäre schwieriger wegen der Aufblähung der Regeln z.B. zu $T \to FT'$ und $T' \to *FT'$ und $T' \to$. Reiche nun den semantischen Wert des ersten Arguments in T' hinein:

```
t (t:ts) = let (fres, ts1) = f (t:ts)
               in case t' of
                   NUM _   -> t' fres ts1
                   LPAREN  -> t' fres ts1
                   _       -> error ...
t' lval (t:ts) = case t of
                   PLUS    -> (lval, t:ts)
                   TIMES   -> let (frest, ts1) = f ts
                                  in t' (lval * fres) ts1
                   RPAREN  -> (lval, t:ts)
                   EOF     -> (lval, t:ts)
```

Somit: konkrete Syntax (insbesondere LL(1)) häufig nicht direkt geeignet für semantische Berechnung, daher geht man über zu einer **abstrakten Syntax** (siehe unten).

Die semantischen Aktionen reichen im Prinzip schon aus für alle Übersetzungsaufgaben, daher bezeichnet man Yacc, Happy etc. auch schon als Compilergeneratoren. In der Praxis wird dies jedoch bei größeren Projekten versucht

[6] `$$` bedeutet, daß der semantische Wert von `id` das `ID`-Argument des Scanners ist.

zu modularisieren, siehe unten.

Man kann jedoch auch einen einfachen Interpreter mittels semantischen Aktionen konstruieren, beispielsweise für Simple-Programme. Hauptdatenstruktur eines Interpreters ist eine **Umgebung**: Abbildung `String` → `Int`, die Identifiern Werte zuweist. Notwendig ist eine Möglichkeit zur Änderung eines Eintrags und eine leere Umgebung. Der „Wert“ eines Ausdrucks ist dann eine Abbildung `Umgebung` → `Int`, der „Wert“ einer Zuweisung ist eine Abbildung `Umgebung` → `Umgebung`:

```
{
update :: (String->Int)->String->Int->(String->Int)
update e n v = \m -> if m == n then v else e m

empty_env n = error("Access to undefined var " ++ n)
}
%name interpreter
%tokentype Token
%token ';' {SEMICOLON}
       '=' {ASSIGN}
       '+' {PLUS}
       '*' {TIMES}
       '(' {LPAREN}
       ')' {RPAREN}
       id  {ID $$}
       num {Num $$}
%%
progrm : stmts '(' exp ')'    { $3 ($1 empty_env)        }
stmts  : stm ';' stmts        { \e -> $3 ($1 e)          }
       | stm                  { $1                       }
stm    : id '=' exp           { \e -> update e $1 ($3 e) }
exp    : exp '+' term         { \e -> ($1 e + $3 e)::Int }
       | term                 { $1                       }
term   : term '*' factor      { \e -> ($1 e * $3 e)::Int }
       | factor               { $1                       }
factor : id                   { \e -> e $1               }
       | num                  { \_ -> $1                 }
       | '(' exp ')'          { $2                       }
{
data Token = ID String | NUM Int | MULT | ...
     deriving Show
}
```

Anmerkungen:

- Die Umgebung wird immer durchgereicht (sie ist nicht global, sondern lokal veränderbar).

- Idee der Denotationellen Semantik: Syntaxkonstrukte entsprechen semantischen Funktionen, dies ist hiermit direkt implementierbar!
- Was noch fehlt, sind Fallunterscheidungen, Schleifen, ...– siehe Übung :o). Zudem sind noch Ausgaben nötig – führe hier bei der Semantik einen Ausgabestring mit, so daß der Wert einer Anweisung eine Transformation (alte Umgebung, bisherige Ausgabe)->(neue Umgebung, neue Ausgabe). Eine Zuweisung verändert nun die erste Komponente, während eine `print`-Anweisung die zweite Komponente modifiziert.

Der neue Interpreter (verkürzt):

```
progrm : stmts             { snd ($1 (empty_env, ""))           }
stmts  : stm ';' stmts     { $3 . $1                            }
       | stm               { $1                                 }
stm    : id '=' exp        { \(e,o) -> (update e $1 ($3 e), o)  }
       | print '(' exp ')' { \(e,o) -> (e, o++show($3 e :: Int)) }
```

5.2 Attributierte Grammatiken

Semantische Aktionen entsprechen einem **Datenfluß von unten nach oben** im Ableitungsbaum, das ist jedoch nicht ausreichend z.B. bei:

- Schachtelungstiefe bei Blöcken: Datenfluß von oben nach unten
- Aufbau einer Symboltabelle: Datenfluß oben → unten → links → rechts → oben

Die Verallgemeinerung sind daher **Attributierte Grammatiken**: für jedes Symbol sind mehrere Attribute mit flexibleren Abhängigkeiten möglich, unterteilt sind die Attribute in zwei Klassen: Bei *synthetisierten Attributen* erfolgt die Berechnung von unten nach oben, bei *vererbten Attributen* erfolgt die Berechnung von oben nach unten.

Definition: Sei eine kontextfreie Grammatik $G = (N, T, P, S)$ gegeben. Eine *attributierte Grammatik* ordnet jedem $X \in N \cup T$ eine Menge von *synthetisierten Attributen* $\mathrm{Syn}(X)$ und eine Menge von *vererbten Attributen* $\mathrm{Inh}(X)$ zu (wobei $\mathrm{Inh}(a) = \emptyset \; \forall \; a \in T$ gilt) und enthält für $A_0 \to A_1 \ldots A_n \in P$ eine Menge von *semantischen Regeln (Attributgleichungen)* der Form $i.a = f(j_1.a_1, \ldots, j_k.a_k)$ mit

- $a \in \mathrm{Syn}(A_i)$ falls $i = 0$
- $a \in \mathrm{Inh}(A_i)$ falls $i > 0$

- für alle $1 \leq i \leq k$ gilt $a_i \in \mathrm{Syn}(A_{j_i})$, falls $j_i > 0$; für $j_i = 0$ gilt $a_i \in \mathrm{Inh}(A_{j_i})$

Intuition: $j.a$ ist ein Attribut a von A_j; die linke Gleichungsseite darf nur synthetisierte Attribute der linken Seite der Produktion oder vererbte Attribute der rechten Seite; für die rechte Gleichungsseite entsprechend umgekehrt – dadurch vermeidet man zyklische Abhängigkeiten in einer Regel, z.B. $0.a = 0.a + 0.a$.

Die *Attributierung* eines Ableitungsbaums ist nun eine Zuordnung von Knoten zu Attributwerten, so daß bei jeder Produktion im Ableitungsbaum alle Attributgleichungen erfüllt sind.

Beispiel: Berechnung der größten Schachtelungstiefe in blockstruckturierter Sprache in folgender vereinfachten Grammatik (mehrdeutig):

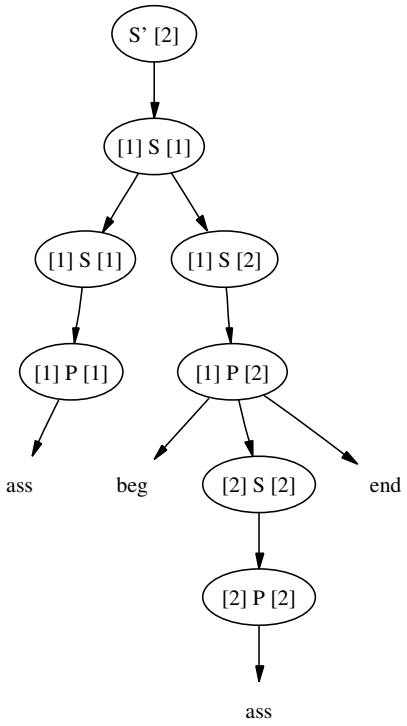
$$\begin{array}{llll} S' \rightarrow S & S \rightarrow SS & P \rightarrow \texttt{assign} \\ & S \rightarrow P & P \rightarrow \texttt{begin}\ S\ \texttt{end} \end{array}$$

Attribute: Zum einen die Tiefe eines Blockes $\mathrm{Inh}(S) = \mathrm{Inh}(P) = \{d\}$ (depth of block), zum anderen das Maximum $\mathrm{Syn}(S) = \mathrm{Syn}(P) = \mathrm{Syn}(S') = \{m\}$ (maximum depth)

$$\begin{array}{lll} S' \rightarrow S & 1.d = 1 \\ & 0.m = 1.m \\ S \rightarrow SS & 1.d = 0.d \\ & 2.d = 0.d \\ & 0.m = \max(1.m, 2.m) \\ S \rightarrow P & 1.d = 0.d \\ & 0.m = 1.m \\ P \rightarrow \texttt{assign} & 0.m = 0.d \\ P \rightarrow \texttt{begin}\ S\ \texttt{end} & 2.d = 0.d + 1 \\ & 0.m = 2.m \end{array}$$

Der Beispielsatz `assign begin assign end` führt zu der Ableitung

$$S' \rightarrow S \rightarrow \left\{ \begin{array}{l} S \rightarrow P \rightarrow \left\{ \begin{array}{l} \texttt{begin} \\ S \rightarrow P \rightarrow \texttt{assign} \\ \texttt{end} \end{array} \right. \\ S \rightarrow P \rightarrow \texttt{assign} \end{array} \right.$$



Hiermit sind alle Übersetzungsprobleme formal definierbar als Berechnung im Syntaxbaum! *Beispiele:*

- Aufbau und das Durchreichen einer Symboltabelle (Aufbau entspricht einem vererbten Symboltabellenattribut und Durchreichen entspricht einem synthetisierten Attribut)
- bei der Codegenerierung: Zusammenbau von Codestücken
- Aufruftiefe etc.

Viele Compilergeneratoren basieren häufig auf attributierten Grammatiken, Eingabe ist dann eine attributierte Grammatik, Ausgabe ist ein Parser und ein Attributauswerter.

Problem: Wie berechnet man Attributierungen? Ideal wäre eine Attributierung schon während der Syntaxanalyse.

Definition: Eine attributierte Grammatik heißt *S-attributiert*, falls $\mathrm{Inh}(X) = \emptyset$ für alle $X \in N$.

Dies läßt sich so auffassen, als ob jedes Symbol ein Attribut (bzw. ein Tupel) hat, das bei Anwendung der Syntaxregel berechnet werden kann. Dieses ist

leicht implementierbar sowohl in Recursive-Descent-Parsern als auch in *shift-reduce*-Parsern. Die semantischen Aktionen erlauben dann auch Seiteneffekte (imperative Sprachen). Compilergeneratoren wie Yacc oder Happy erlauben nur S-attributierte Grammatiken.

Definition: Eine Attributierte Grammatik heißt L-attributiert, falls jedes vererbte Attribut eines Symbols A_j auf der rechten Seite einer Produktion $A_0 \to A_1, \ldots, A_m$ nur abhängt von den (synthetisierten) Attributen der Symbole $A_i, \ldots, A_{j-1}$ und den vererbten Attributen von A_0. D.h. für alle Attributgleichungen $j.a = f(\ldots, i.a', \ldots)$ mit $a \in \mathrm{Inh}(A_j)$ gilt: $i < j$.

Dann bestehen Abhängigkeiten im Baum nur von links nach rechts! Damit ist die Berechnung einem top-down-links-rechts-Durchlauf möglich, d.h. die Implementierung ist in RD-Parsern einfach: Vererbte Attribute sind Parameter der Nichtterminalprozeduren, während die synthetisierten Attribute die Ergebnisse der Nichtterminalprozeduren sind.

Beispiel: Die vorhin verwendete Grammatik für Blockschachtelung ist L-attributiert!

```
data Token = Assign | Begin | End | Seq

parse :: [Token]->(Int,[Token])
parse ts = stm 1 ts

stm :: Int->[Token]->(Int,[Token])
stm d (t:ts) = case t of
    Seq -> let (m1, ts1) = stm d ts
               (m2, ts2) = stm d ts1
           in (max m1 m2, ts2)
    _   -> block d (t:ts)

block :: Int->[Token]->(Int,[Token])
block d (t:ts) = case t of
    Assign -> (d,ts)
    Begin  -> let (m,End:ts1) = stm (d+1) ts
              in (m,ts1)

> parse [Seq,Assign,Begin,Assign,End]
(2,[])
```

Bemerkungen: S/L-attributierte Grammatiken sind relevant bei Ressourcen-Beschränkungen (one-pass-Compiler), der Nachteil besteht jedoch im Mangel

an Modularität, da semantische Analyse und Codeerzeugung etc. als Attribute geschrieben werden müßten. Verwende dann mehrere Attributgrammatiken oder Funktionen auf Syntaxbäumen: abstrakte Syntax!

5.3 Abstrakte Sytax

Nachteil einer konkreten Syntax ist, daß diese häufig sehr komplex und ungeeignet für Übersetzungsaufgaben ist (z.B. Ausdrücke in LR(1)- oder LL(1)-Form). Einfache und klare Grammatik für Ausdrücke:

$$\begin{aligned} exp &\rightarrow \texttt{ID} \mid \texttt{NUM} \mid exp\ binop\ exp \\ binop &\rightarrow \texttt{PLUS} \mid \texttt{TIMES} \mid \ldots \end{aligned}$$

Nachteil dieser Grammatik ist, daß sie **mehrdeutig** ist. Dies ist ein Problem für den Parser, jedoch kein problem für spätere Phasen, da diese dann auf einem konkreten Ableitungsbaum basieren.

Eine **abstrakte Syntax** enthält daher die gleichen Strukturinformationen wie eine konkrete Syntax, d.h. die Semantik eines Programms muß aus der abstrakten Syntax ableitbar sein; sie verzichtet jedoch auf überflüssige Elemente.

Aufgabe eines Parsers ist also, das Programm zu lesen und einen zugehörigen abstrakten Syntaxbaum zu generieren (d.h. einen Term bezüglich der abstrakten Syntax). Bei der abstrakten Syntax wird alles weggelassen, was **semantisch irrelevant** ist, beispielsweise

- Terminalsymbole ohne semantische Information (`ASSIGN`, `SEMICOLON`)
- „Kettenregeln“ (Weiterreichen von semantischen Werten, z.B. `exp`→`term` und `term`→`factor`)

Anmerkungen: Frühe Compiler waren häufig one-pass-Compiler und benutzten daher nicht das Konzept der abstrakten Syntax. In modernen Compilern jedoch wird dies eingesetzt, die abstrakte Syntax bildet eine Schnittstelle zwischen syntaktischer Analyse und semantischer Analyse, dies trägt zur Modularität bei.

Realisierung in Haskell: In Haskell entspricht die abstrakte Syntax einem Datentyp, damit ist ein abstrakter Syntaxbaum ein Term dieses Typs.

Beispiel: Abstrakte Syntax für Simple:

$$\begin{aligned} Stm &\rightarrow Stm; Stm \mid \texttt{ID} := \texttt{exp} \mid \texttt{PRINT}\ exp \\ exp &\rightarrow \texttt{ID} \mid \texttt{NUM} \mid exp\ binop\ exp \\ binop &\rightarrow \texttt{PLUS} \mid \texttt{TIMES} \end{aligned}$$

Als Datentyp in Haskell (wobei die Terminalsymbole wie `PRINT` überflüssig sind, dafür aber eine Kennzeichnung der jeweiligen Alternativen sind):

```
data Binop = PLUS | TIMES
data Exp = IdExp String | NumExp Int
         | OpExp Exp Binop Exp
data Stm = SeqStm Stm Stm | AssignStm String Exp
         | PrintStm Exp
```

Änderung des Parsers zur Erzeugung des abstrakten Syntaxbaums:

```
...
%%
program : stmts              { $1               }
stmts   : stm ';' stmts      { SeqStm $1 $3     }
        | stm                { $1               }
stm     : id '=' exp         { AssignStm $1 $3  }
        | print '(' exp ')'  { PrintStm $3       }
exp     : exp '+' term       { OpExp $1 PLUS $3 }
        | term               { $1               }
term    : term '*' factor    { OpExp $1 TIMES $3 }
        | factor             { $1               }
factor  : id                 { IdExp $1         }
        | num                { NumExp $1        }
        | '(' exp ')'        { $2               }
```

Beispiel: Die Eingabe `x=2; print(3+x*5)` setzt sich um zu

```
(SeqStm (AssignStm "x" (NumExp 2))
        (PrintStm (OpExp (NumExp 3) PLUS
                         (OpExp (IdExp "x") TIMES
                                (NumExp 5)))))
```

Anmerkungen:

- Eine ähnliche Implementierung ist für RD-Parser möglich, hier ist in der Regel die Verwendung von abstrakter Syntax noch wichtiger, da die LL(1)-Syntax meist sehr umständlich ist.
- Ein Nachteil der abstrakten Syntax ist bei der Fehlerausgabe (z.B. Typfehler) fehlt der Bezug zur konkreten Syntax, insbesondere fehlen Positionsangaben im Quelltext! Lösung: diese Informationen werden in den abstrakten Syntaxbaum eingebungen:

1. Füge einen Typ der Positionsangaben hinzu, z.B. `type Pos = Int` (oder `(Int, Int)`)
2. Erweitere abstrakte Syntax, wo es eventuell notwendig ist:

   ```
   data Exp = IdExp String Pos | NumExp Int | OpExp Exp BinOp Exp Pos
   ```

3. Initialisiere Werte für Positionsangaben in semantischen Aktionen, dazu ist es notwendig, daß z.B. der Scanner für jedes Token eine Position als Wert anfügen muß.

6 Semantische Analyse

Die **semantische Analyse** dient der Vorbereitung der Codeerzeugung. Da bisher eine kontextfreie Syntax verwendet wurde, müssen nun syntaktische Einschränkungen überprüft werden, die nicht kontextfrei ausdrückbar sind. Genaue Aufgaben:

- Zuordnung von Deklarationen zu Anwendungen
- Überprüfung von Sichtbarkeitsregeln (Blockstruktur, Prozeduren, `let`, ...)
- Typprüfung (Überprüfung typkorrekter Anwendungen, Annotation von Ausdrücken mit Typangaben)

Realisiert wird dies mit einer **Symboltabelle**, die prinzipiell eine Abbildung von Bezeichnern auf Bezeichner-Infos (Typ, Blocktiefe, Speicherplatz, ...) ist. Dies ist die zentrale Datenstruktur für weitere Phasen (z.B. Codeauswahl abhängig vom Typ, Speicherplatz). Sie muß Techniken zur Verwaltung von Scope-Konstrukten bereitstellen. Ein **Scope** ist ein Programmteil, in dem eine Menge von Bezeichnern sichtbar ist, beispielsweise:

- Blöcke (Algol, Pascal, Modula, ...)
- Prozedur- und Funktionsrümpfe
- Methoden, Records, Klassen
- `let`, `letrec`, `where`, ...

Die Symboltabelle muß folgende Operationen bereitstellen:

- *emptytable* erschafft eine neue, leere Symboltabelle
- *enter_scope* eröffnet einen neuen Scope
- *exit_scope* verläßt den aktuellen Scope und geht in den Zustand vor dem zugehörigen *enter_scope*
- *insert(id, info)* erstellt einen neuen Eintrag `id` mit `info`-Informationen (z.B. Typangabe) im aktuellen Scope (kann z.B. einen Fehler erzeugen, falls `id` schon vorhanden ist)
- *lookup(id)* sucht den `info`-Eintrag für `id` von innen nach außen (und eventuell Fehlermeldung, falls nicht vorhanden)

Problem ist, daß die Suche schnell sein muß!

Implementierungsmöglichkeiten:

1. Kellerartige Verwaltung: *insert* entspricht `push`, *enter_scope* entspricht `push` mit einer Scopemarkierung, *exit_scope* ist `pop` bis zur letzten Scopemarkierung und *lookup* ist eine Suche von `top` nach unten im Stack.

   Diese Implementierung ist sehr einfach, aber sehr ineffizient, da das lookup linear ist in der Anzahl der Deklarationen

2. Kellerartig mit schnellerer Suche (wie oben, aber jeden Scope als Suchbaum verwalten, d.h. logarithmische Suche in jedem Scope)

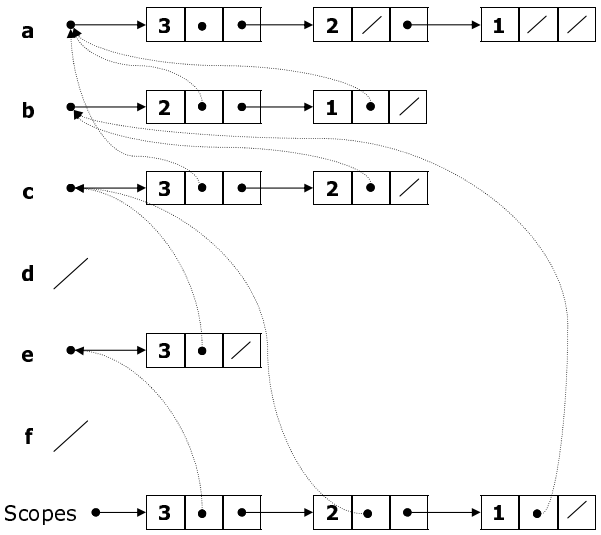
3. Verwaltung mit (nahezu) konstanter Suchzeit:

   - Verwalte ein Feld mit Bezeichnern als Indexmenge, z.B. durch Hashing; dann ist der Zugriff in nahezu konstanter Zeit möglich
   - Jedes Feldelement verweist auf eine Liste von Einträgen, wobei der erste Eintrag die innerste Deklaration und der letzte Eintrag die äußerste Deklaration ist; bei keinem Eintrag ist der Bezeichner nicht sichtbar
   - zusätzliche Verkettung aller Bezeichner eines Scopes zum effizienten Löschen eines Scopes
   - Verwaltung der Köpfe der Scope-Listen als Stack

Beispiel: Betrachte das folgende Programm an der Stelle `(*)`:

```
begin (1)
   declare a, b
   begin (2)
      declare a, b, c
      begin (3)
         declare a, c, e
         (*)
      end
   end
end
```

Die Symboltabelle sieht nun folgendermaßen aus:



Die Operationen darauf lassen sich dann wie folgt implementieren:

- *insert*: neuer Listenkopf einfügen und verketten, als erstes Element in die Eintragsliste einfügen
- *lookup*: zugehörigen Listenkopf suchen (konstante Zeit!)
- *enter_scope*: neuen Scope-Kopf einfügen
- *exit_scope*: Lösche alle Listenköpfe der Eintragslisten

Nun sind die folgenden Schritte realisierbar (z.B. mit Attributgrammatiken oder Prozeduren auf dem abstrakten Syntaxbaum):

- Zuordnung zwischen Deklaration und Anwendung: Deklaration entspricht *insert*, Anwendung entspricht *lookup*
- Typüberprüfung: *insert* mit deklariertem Typ und Übersetzung von Ausdrücken in typannotierte Ausdrücke

7 Code-Erzeugung

7.1 Laufzeitspeicherorganisation

Zur Laufzeit soll der Zugriff auf Variablen über ihre **Adresse im Zielprogramm** erfolgen. Zu klären ist, wo die Objekte abgelegt werden und wie man auf die Objekte zugreift. Betrachte folgendes Beispielprogramm:

```
proc p
begin
   int x;
   proc q
   begin
      int y;
      y := x; (*)
      ...
   end;
end;
```

Bei `(*)` erfolge ein Zugriff auf x und y, aber beide sind nicht global (p und q können rekursiv sein!) und haben daher keinen festen Speicherplatz. Zudem ist x nicht lokal in q deklariert: wie findet man x?

Allgemein: Welche Werte müssen wo und wie lange gespeichert werden? Das hängt von der verwendeten Sprache ab!

Programmglobale Variablen haben dieselbe Lebensdauer wie das Programm, prozedurlokale Variablen die Lebensdauer der Prozedurabarbeitung. Ausnahmen treten auf bei **Funktionen höherer Ordnung**:

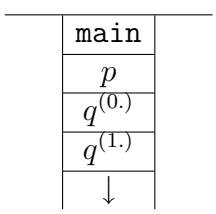
```
f x = let g y = x + y
       in g
h = f 0
g = f 3
> (h 2) + (g 5)
10
```

Die Funktion f wird zwei mal aufgerufen und beendet, allerdings muß der Wert von x noch gespeichert werden! Im Folgenden betrachten wir jedoch zunächst **keine Funktionen höherer Ordnung**, d.h. alle lokalen Variablen leben bis Prozedurende.

Damit können die Prozedur-Variablen kellerartig verwaltet werden: Beim Prozedureintritt kann Speicherplatz für lokale Variablen reserviert werden auf dem Prozedurkeller für lokale Variablen; bei Prozedurende wird der Speicher wieder freigegeben. Als Skizze:

```
main
   proc p
      proc q
         ... q ...;
      end
      ... q ...;
   end
   ... p ...;
end
```

auf dem Keller:



Manche Programmiersprachen erlauben lokale Deklarationen in **Blöcken**, Lebensdauer ist dann die Blockabarbeitung. **Implementierung** im ersten Ansatz wie Prozeduren (bei Blockeintritt: reserviere Speicherplatz auf dem Stack), es geht aber effizienter, da Blöcke nicht rekursiv sind (reserviere Speicherplatz bei Prozedurbeginn für alle Blöcke in dieser Prozedur). Beachte dabei, daß der Speicherplatz bei geschachtelten Blöcken addiert wird, bei disjunkten Blöcken ist der Maximalwert der einzelnen Blöcke wichtig:

```
proc p
begin (* Block A *)
   int x
   ...
   begin (* Block B *)
      int y
      ...
   end
   ...
   begin (* Block C *)
      int z
      ...
   end
end
```

Speicherplatz für die Prozedur p ist nun gegeben durch:

$$\text{size}(p) = \text{size}(A) + \max(\text{size}(B), \text{size}(C))$$

Hierbei gibt size die Summe des Speicherplatzes für alle unmittelbaren Deklarationen in einem Block an. Das gleiche Schema kann man auch für **nicht-rekursive Prozeduren** anwenden (z.B. Fortran).

Dynamische Datenstrukturen sind dadurch gekennzeichnet, daß die Größe erst zur Laufzeit bekannt ist, beispielsweise dynamische Arrays: Indexgrenzen werden erst zur Laufzeit bekannt:

```
proc p(int n)
   int a[n]
   ...
```

Realisierung entweder durch Festlegung des Speicherplatzes für a nach Eintritt in p auf dem Stack; oder durch Anforderung von Speicherplatz für a zur Laufzeit in separatem dynamischen Speicherbereich (**Heap**, siehe unten).

Bei **dynamischen Records** (Objekten) wird Speicherplatz durch explizite Operationen („new“) bereitgestellt, die Lebensdauer ist dann häufig über das Prozedurende hinaus, d.h. keine Speicherung im Stack!

Daher muß ein neuer Speicherbereich für dynamische (nicht kellerartige) Speicherverwaltung verwendet werden: ein **Heap**. Eine Speicheranforderung geschieht explizit durch „new“, die Freigabe passiert dann ebenfalls explizit („dispose“), dies birgt die Gefahr von **„falschen“ Freigaben** („dangling pointers“). Alternativ geschieht eine implizite Freigabe des Speicherplatzes durch **garbage collection** (sicherer, aber algorithmisch schwieriger und weniger effizient).

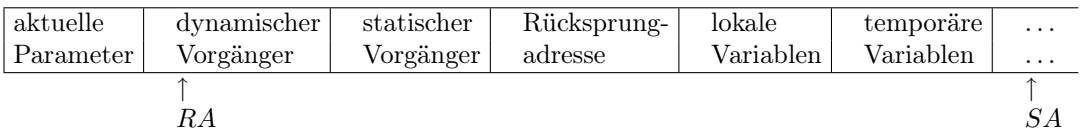
Typische Laufzeitspeicheraufteilung:

Der Code ist das übersetzte Programm, im statischen Speicher werden globale Variablen, Konstanten (Strings) etc. (ersetzbar durch Prozedur „main“). Auf dem Stack sind Infos zu Prozeduraufrufen (siehe unten) gespeichert, auf dem Heap werden dynamische Datenstrukturen verwaltet. Ein **Stack/Heap-Overflow** tritt auf, wenn Stack- oder Heapgrenzen zusammenstoßen, hier wird dann beispielsweise eine garbage collection angestoßen.

7.1.1 Aufbau des Stacks

Für jeden Prozeduraufruf liegt auf dem Stack ein Eintrag („stack frame“ oder „Prozedurrahmen“), der alle für den Aufruf der Prozedur notwendigen Informationen enthält. Jeder Prozeduraufruf erzeugt einen neuen Rahmen, bei Prozedurende wird der Rahmen gelöscht und der Zustand vor dem Aufruf wiederhergestellt. Der Rahmen enthält die Rückkehradresse, lokale Variablen, aktuelle Parameter etc.

Vorschlag für einen Rahmenaufbau



- SA ist die Adresse des nächsten freien Stackelements („top of stack“).
- RA ist die Rahmenadresse, d.h. die Basisadresse für den aktuellen Rahmen.
- Die aktuellen Parameter werden bei Prozeduraufruf vom Aufrufer auf den Stack gelegt (vor einem Sprung zum Prozedurcode), hier können aber eventuell auch Ergebnisparameter (bzw. die Adressen der Ergebnisse) abgelegt werden.
- Der **dynamische Vorgänger** ist der Wert von RA der aufrufenden Prozedur (dient zur Wiederherstellung des Zustandes nach Prozedurabarbeitung); bei Prozedureintritt: $M[SA] := RA; RA := SA; SA := SA + 1;$.
- Im **statischen Vorgänger** wird der RA des letzten Aufrufs der textuell umfassenden Prozedur gespeichert (notwendig zur Adressierung nicht-lokaler Variablen; nicht notwendig bei Programmiersprachen ohne textuell geschachtelte Prozeduren, z.B. in C).
- Die **Rücksprungadresse** ist die Codeadresse, bei der nach Prozedurende weitergemacht wird. Viele klassische Maschinenarchitektueren haben eine *call*-Instruktion, die automatisch die Rücksprungadresse auf den Stack speichert. Diese Technik kann jedoch wieder den Nachteil haben, daß immer ein Zugriff auf den Hauptspeicher nötig ist bei jedem Prozeduraufruf.

  Viele moderne Architekturen lassen die Stackverwaltung offen, d.h. ein Prozeduraufruf erfolgt nicht mittels einer *call*-Instruktion, sondern:

  - Speichere die Rückkehradresse in ein spezielles Register (RET).
  - Springe zum Code der Prozedur.
  - Springe am Prozedurende einer Prozedur ohne weiteren Prozeduraufruf zu RET.
  - In einer Prozedur mit weiterem Prozeduraufruf: Speichere den Inhalt von RET im Prozedurrahmen und schreibe ihn nach Aufruf oder bei Prozedurende zurück.

  Damit läßt sich eine Zeitersparnis bei einfachen Prozeduren erreichen, bei komplexeren Prozeduren ergeben sich kellerbasierte Aufrufe.

Allgemeines Prinzip: Versuche, Werte zunächst in Registern zu halten, bevor sie in den Stack geschrieben werden.

Beispiel: In der Praxis haben Prozeduren selten mehr als vier Parameter, damit wird es z.T. möglich, die ersten vier Parameter in Registern zu übergeben und weitere Parameter auf den Stack zu legen. Falls ein weiterer Aufruf erfolgt und diese Parameter danach noch benötigt werden, sichere diese vor dem Aufruf auf dem Stack.

Damit können elementare Prozeduren und sogar endrekursive Prozeduren völlig ohne Stack abgearbeitet werden. Dies ist eine wichtige Optimierung für objekt-orientierte, logische und funktionale Sprachen.

7.1.2 Dynamische und statische Vorgänger

Beispiel:

```
proc main
   int x
   ...
   proc p
      int y
      ...
      proc q
        int z
        ...
        x := y + z (*)
...
```

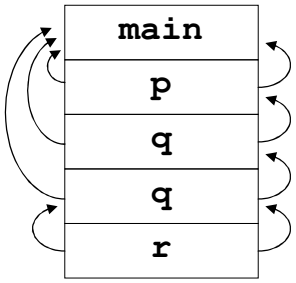
Wie erfolgt nun der Zugriff auf x und y in $(*)$?

- Der statische Vorgänger von q ist die RA des letzten Aufrufs von p, der statische Vorgänger von p ist die RA des letzten Aufrufs von `main`.
- Annahme: x hat Relativadresse 2 in main, y hat Relativadresse 3 in p
- Adresse von y in $(*)$ als $M[RA + 1] + 3$
- Adresse von x in $(*)$ als $M[M[RA + 1] + 1] + 2$

Wichtig ist, daß der statische Vorgänger nicht gleich dem dynamischen Vorgänger sei muß! Der dynamischer Vorgänger ist die RA der dynamisch vorigen Prozedur, der statische Vorgänger ist die RA des dynamisch letzten Aufrufs der statisch umfassenden Prozedur.
Beispiel: Betrachte in folgender Definition die Aufrufreihenfolge `main` $\rightarrow$ `p` $\rightarrow$ `q` $\rightarrow$ `q` $\rightarrow$ `r`:

```
proc main
   ...
   proc p
   ...
   proc q
      ...
      proc r
```

Die Stackstruktur sieht dann wie folgt aus:

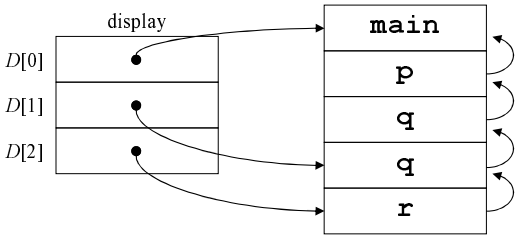


Beim Aufruf einer Prozedur werden nun folgende Aktionen ausgeführt:

- Die Übergabe des statischen Vorgängers erfolgt im Register SV, die Berechnung des SV kann berechnet werden durch die Differenz der Schachtelungstiefe zwischen der aufrufenden und der aufgerufenen Prozedur (geschieht in der semantischen Analyse):
  - Ist die Differenz gleich 0, so liegt ein direkt rekursiver Aufruf vor, d.h. wir übergeben $SV := M[RA + 1]$
  - Ist die Differenz größer 0, so müssen wir entsprechend lange der statischen Vorgängerkette folgen (erzeuge entsprechenden Code).
- Bei Prozedureintritt wird dann $M[RA + 1] := SV; SA := SA + 1;$ ausgeführt.

Ein Nachteil ist, daß bei tiefen Schachtelungen ein Zugriff auf globalere Variablen dem Durchlauf durch die statische Vorgängerkette entspricht. Alternativen:

1. **Display-Technik**: Die statische Vorgängerliste wird ersetzt durch ein Feld D („display“): $D[i]$ ist der statische Vorgänger der Schachtelungstiefe i. Obiges Beispiel mit Display-Technik:



Ein Zugriff auf prozedurlokale Variablen erfolgt nun über $M[RA + \textit{relative Adresse}]$, auf globale Variablen in Tiefe j erfolgt durch $M[D[j] + \textit{relative Adresse}]$.

Im obigen Beispiel ist damit der Zugriff innerhalb von r auf Variablen aus q mittels $D[1]$ möglich, auf Variablen aus *main* mittels $D[0]$.

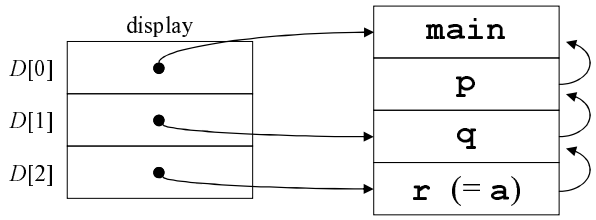
Implementierung:

- Im Prozedurrahmen ersetzen wir den statischen Vorgänger durch ein Display, d.h. aber, daß beim Aufruf einer Prozedur das Display des statischen Vorgängers komplett kopiert werden muß!
- Wir verwalten das Display als globales Feld (die Maximalgröße läßt sich zur Compilezeit bestimmen); bei Prozeduraufruf der Tiefe i wird $D[i]$ im aktuellen Prozedurrahmen gesichert und $D[i] := RA$ gesetzt; bei Prozedurende muß der gesicherte Wert wieder zurückgeschrieben werden. Dies funktioniert jedoch nicht, falls Prozeduren als Parameter auftreten:

  Beispiel:

```
main
   int x;
   proc p
      int x;
      proc a
         x = 1;
      end
      call q(a);
   end
   proc q(proc r)
      call r;
   end
call ps;
end
```

Dann hat a die Tiefe 2, q die Tiefe 1. Das Display sieht nun wie folgt aus:



Der statische Vorgänger von $r(= a)$ müßte nun p sein, doch p ist nicht über das Display adressierbar! Ausweg: speichere bei Prozedurparametern außer der Codeadresse auch den statischen Vorgänger; d.h. der Prozedurrahmen muß das Display enthalten (und bei Aufruf kopieren).

2. **Lambda-Lifting** (aus funktionalen Sprachen): Idee ist, daß jede nichtlokale Variable, auf die ein Zugriff in einer Prozedur erfolgt, als Argument hinzugefügt wird:

```
proc main
   int x;
   proc p
      int y;
         proc q
            int z;
            x := y + z;
         end
      call q;
   end
call p;
end
```

Das λ-Lifting ergibt jetzt folgendes Programm:

```
proc main
   int x;
   call p(x);
end
proc p(ref x)
   int y;
   call q(x, y);
```

```
end
proc q(ref x, ref y);
   int z;
   x := y + z;
end
```

Potentieller Nachteil: Statische Vorgängerketten werden ersetzt durch Verweisketten, zudem werden eventuell viele Prozedurparameter benötigt. Demgegenüber stehen als **Vorteile**, daß kein Speicher für Displays und statische Vorgänger benötigt wird sowie keine Verwaltung zur Laufzeit (Zeitersparnis!) und auch eine problemlose Verwendung von Prozeduren als Parameter.

7.1.3 Speicherorganisation für spezielle Datentypen

Im Prozedurrahmen ist ein Speicherbereich für lokale Variablen reserviert – doch wie ist dieser genau aufgebaut? Idee ist prinzipiell eine sequentielle Speicherung der lokal deklarierten Objekte.

Beispiel: Betrachte eine Prozedur p mit Variablen $v_1, \ldots, v_n$ vom Typ $\tau_1, \ldots, \tau_n$. Bei einer sequentiellen Speicherung ergibt sich dann folgender Aufbau:

Speicher für v_1
Speicher für v_2
$\ldots$
Speicher für v_n

Der Zugriff auf v_i erfolgt jetzt unter der Relativadresse

$$RA + 3 + \sum_{j=1}^{i-1} \text{size}(\tau_j)$$

Dabei ist $\text{size}(\tau_j)$ der benötigte Speicherplatz für Objekte vom Typ τ_j, dies sollte berechenbar sein durch den Compiler, da so auch die Relativadresse durch den Compiler berechenbar ist.

Beispielsweise sind bei primitiven Datentypen einige Standardgrößen üblich:

$$\begin{aligned} \text{size}(\texttt{int}) &= 4 \text{ Bytes} \\ \text{size}(\texttt{double}) &= 8 \text{ Bytes} \\ \text{size}(\texttt{bool}) &= 1 \text{ Byte} \\ \text{size}(\texttt{reftype}) &= \text{architekturabhängig} \end{aligned}$$

Prinzipiell könnte dabei `bool` auch in einem Bit gespeichert werden, dies ist bei üblichen Architekturen jedoch nicht direkt adressierbar (oder 4 Bytes bei 32-Bit-Architekturen). Zu komplexeren Datentypen:

1. **records:**

   $$\tau = \texttt{record}\ c_1 : \tau_1, \ldots, c_k : \tau_k\ \texttt{end}$$

   Dann würden c_1 bis c_k wieder sequentiell gespeichert mit

   $$\text{size}(\tau) = \sum_{i=1}^{k} \text{size}(\tau_i)$$

   Die Relativadresse (relativ zur Basisadresse des Records) für Komponente c_j ist dann

   $$\text{reladr}(c_j) = \sum_{i=1}^{j-1} \text{size}(\tau_i)$$

2. **Felder:**

   $$\tau = \texttt{array}[u_1..o_1, u_2..o_2, \ldots, u_k..o_k]\ \texttt{of}\ \tau'$$

   Falls alle u_i, o_i zur Übersetzungszeit bekannt sind, spricht man von einem *statischen Feld*: Sei $d_i = o_i - u_i + 1$.

   $$\text{size}(\tau) = \left(\prod_{i=1}^{k} d_i \right) \cdot \text{size}(\tau')$$

   Die Speicherung des Feldes kann nun (z.B. bei einem zweidimensionalen Feld) zeilenweise oder spaltenweise erfolgen:

   - Bei der zeilenweisen Speicherung:

11	12	13
21	22	23
31	32	33

     $\Longrightarrow$

11	12	13	21	22	23	31	32	33

     Dann ist die relative Adresse reladr($\texttt{element}[i_1, \ldots, i_k]$) gleich

     $\text{size}(\tau') \cdot (d_k \cdot \ldots \cdot d_2(i_1 - u_1) + d_k \cdot \ldots \cdot d_3(i_2 - u_2) + \ldots + (i_k - u_k))$

   - Bei der spaltenweisen Speicherung:

11	12	13
21	22	23
31	32	33

     $\Longrightarrow$

11	21	31	12	22	32	13	23	33

     Dann ist die relative Adresse reladr($\texttt{element}[i_1, \ldots, i_k]$) gleich

     $\text{size}(\tau') \cdot ((i_1 - u_1) + (i_2 - u_2) \cdot d_1 + \ldots + (i_k - u_k) \cdot d_1 \cdot \ldots \cdot d_{k-1})$

Von ein dynamischen Feld spricht man, falls einige u_i oder o_i zur Compile-Zeit unbekannt sind. Im lokalen Speicher wird nun ein sogenannter *dope vector (DV)* für das Feld gespeichert:

AA
u_1
o_1
$\ldots$
u_k
o_k

Damit ergibt sich

$$\text{size}(\tau) = 2 \cdot \text{size}(\texttt{int}) \cdot k + \text{size}(\texttt{reftype})$$

Damit werden also nur die dynamisch relevanten Parameter gespeichert, die eigentlichen Daten werden im Heap abgelegt. Bei Bekanntwerden der Grenzen wird dann folgendes zur Laufzeit ausgeführt:

```
for i = 1...k
  DV[size(reftype) + 2(i-1)·size(int)] := u_i
  DV[size(reftype) + (2i-1)·size(int)] := o_i
DV[0] := SA // erste freie Speicheradresse im Stack
asize := size(τ') · d_1 · d_2 · ... · d_k
if SA + asize > stackmax then
  error "Stack overflow!"
else
  SA := SA + asize
// Einstellen der fiktiven [0,0,...,0]-Adresse:
DV[0] := DV[0] − (d_k · ... · d_2 · u_1 + d_k · ... · d_3 · u_2 + ... + u_k)
```

Nun können Feldelemente wie folgt adressiert werden (mit r als Relativadresse des DV): Die Adresse address($element[i_1, \ldots, i_k]$) ist gleich

$$M[RA + r] + \text{size}(\tau') \cdot (d_k \cdot \ldots \cdot d_2 \cdot i_1 + d_k \cdot \ldots \cdot d_3 \cdot i_2 + \ldots + i_k)$$

Um die Anzahl der Multiplikationen zu reduzieren, kann man folgende Umformung anwenden:

$$M[RA + r] + \text{size}(\tau') \cdot ((\ldots((i_1 \cdot d_2 + i_2) \cdot d_3 + i_3) \ldots + i_{k-1}) \cdot d_k + i_k)$$

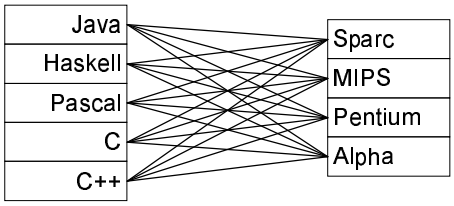
Dies alles gilt für zeilenweises Speichern, analog für spaltenweises abspeichern!

3. **Vereinigungstypen**: $\tau = \tau_1 + \tau_2$; verwende hier überlappendes Speichern: $\text{size}(\tau) = \max(\text{size}(\tau_1), \text{size}(\tau_2))$.

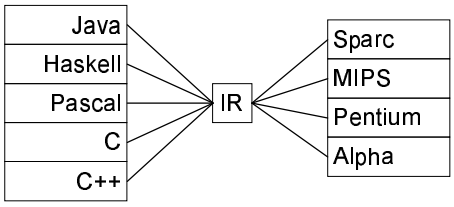
4. **Variante Records**: Kombination von Records und Vereinigungstypen.

5. **Funktionen/Prozeduren**: Speicherung der Codeadresse und des statischen Vorgängers (siehe oben).

7.2 Zwischencodeerzeugung



Eine direkte Übersetzung in Maschinencode für m Sprachen auf n Maschinen würde $m \cdot n$ verschiedene Übersetzer benötigen. Besser ist es, Optimierungen für verschiedene Zielarchitekturen wiederzuverwenden. Daher geschieht eine Übersetzung in eine gemeinsame **Zwischensprache (IR, intermediate representation)**:



Nun müssen nur noch m *front ends* und n *back ends* geschrieben werden:

- Das front end muß keine komplexen maschinenspezifischen Details beachten, und
- das back end muß keine komplexen Quellsprachendetails beachten.

Anforderungen an die Zwischensprache:

- einfach generierbar durch semantische Analyse
- einfach übersetzbar in reale Zielmaschinen
- IR-Konstrukte: einfache Semantik (für Optimierung und Transformation)

Es gibt nun viele IR-Ansätze:

1. **abstrakte Ausdrucksbäume** – siehe später

2. **Stackmaschinencode**: Einfache Stackmaschine, alle Berechnungen finden auf dem Stack statt. So wird beispielsweise $x := a + 3 \cdot b$ übersetzt in

```
lvar x    // lade die Adresse von x auf den Stack
push a    // lade den Wert von a auf den Stack
const 3   // lade Konstante 3 auf den Stack
push b    // lade Wert von b auf den Stack
apply *   // verknüpfe die beiden obersten Elemente
          // mit * und lege Ergebnis auf den Stack
apply +   // analog
assign    // weise oberstes Element an zweitoberste
          // Adresse zu
```

3. **Drei-Adresscode**: einfache Assemblersprache, allerdings können die Befehle bis zu drei Operanden verknüpfen; Beispiele für Befehle:

```
x := y op z        // mit op aus {+, -, *, /, ... }
x := op y          // mit unärem Operator, z.B. not
x := y             // mit y Konstanten oder Variablen
goto L             // Sprung
if x cop y goto L  // mit cop aus {=, <, >, ...}
x := y[i]          // Feldindizierung
x[i] := y          // Feldindizierung
```

   Beispielübersetzung von $x := a + 3 \cdot b$ mit temporären Variablen v, w

```
v := 3
w := v * b
x := a + w
```

   Vorteil gegenüber Stackcode: Umordnungen und Optimierungen sind leichter möglich, da hier explizite Variablen (später eventuell abgebildet auf Register) verwendet. Abstrakte Ausdrucksbäume sind später leicht übersetzbar in 3-Adresscode.

7.2.1 Abstrakte Ausdrucksbäume

Es gibt zwei Objektarten in abstrakten Ausdrucksbäumen:

- **Ausdrücke** liefern ein Ergebnis, haben aber auch Seiteneffekte.
- **Anweisungen** haben kein Ergebnis, sondern nur Seiteneffekte.

Definiert werden die Bäume als abstrakte Assemblersprache als Haskell-Datentypen:

```
data Exp = CONST Int -- auch Float, ...
         | NAME Label -- Sprungziel als Wert
         | TEMP Temp
         | BINOP BinOp Exp Exp
         | MEM Exp
         | CALL Exp [Exp]
         | ESEQ Stm Exp
data Stm = MOVE Exp Exp
         | EXP Exp
         | JUMP Exp
         | CJUMP RelOp Exp Exp Label Label
         | SEQ Stm Stm
         | LABEL Label
data BinOp = PLUS | MINUS | MULT | DIV |
             AND | OR | LSHIFT | RSHIFT
data RelOp = EQ | NE | LT | GE
```

Bedeutung:

- `CONST` i – Konstante i
- `NAME` l – Symbolischer Name (l entspricht einer Assemblermarke)
- `TEMP` t – temporärer Wert mit Namen t (Maschinenregister) wird zurückgegeben
- `BINOP` *op* e_1 e_2 – Auswertung zunächst von e_1, danach von e_2 und dann Anwendung von *op* auf die berechneten Werte und Rückgabe des Ergebnisses
- `MEM` e – Auswertung von e und Rückgabe des Speicherinhalts an der Adresse e; hier Vereinfachung durch die Annahme, alle Speicherinhalte haben die gleiche Länge; sonst müßten noch spezialisierte Operationen angegeben werden wie `MEM` e n mit n als Anzahl an Bytes, die gelesen werden.

- `CALL` f l – Auswertung von f zu einem Namen und Auswertung von l von rechts nach Links, dann Aufruf der Prozedur mit den ausgewerteten Argumenten
- `ESEQ` s e – Auswertung von s, Auswertung von e und Rückgabe des Ergebnisses von e
- `MOVE` (`TEMP` t) e – Ergebnis der Auswertung von e speichern in t
- `MOVE` (`MEM` e_1) e_2 – werte e_1 aus und erhalte Adresse a, werte danach e_2 aus und schreibe dies in a
- `EXP` e – werte e aus und ignoriere das Ergebnis
- `JUMP` e – werte e aus und springe zum Ergebnis
- `CJUMP` *cop* e_1 e_2 t f – werte erst e_1 aus, dann e_2, dann *cop* von e_1 und e_2; falls letzteres wahr ist, so springe zu t, andernfalls zu f
- `SEQ` s_1 s_2 – führe s_1 aus, dann s_2
- `LABEL` l – nur Sprungmarke

Die genaue **Semantik** ist nun leicht definierbar, z.B. durch einen Interpreter, der Speicher durch eine Abbildung `mem :: Int -> Int` simuliert und die Register durch `reg :: Temp -> Int`.

Bisher sind nur Prozedurrümpfe darstellbar; die Hinzunahme von Prozeduren/Funktionen geschieht durch

```
data Proc = PROC Label [Temp] Stm
          | FUNC Label [Temp] Stm Temp
```

Dabei ist `Stm` der Rumpf und das letzte `Temp` der Funktion das Register, in dem das Ergebnis abgelegt wird.

`LABEL` und `TEMP` haben beliebig viele Werte, und wir haben eine Operation zur Erzeugung neuer Namen, da der Compiler bei der Übersetzung ständig eindeutige Namen benötigt:

```
data Label = Label String
data Temp  = Tepm  String
```

Die **Namensgenerierung** soll *on the fly* geschehen; möglich wäre eine globale „Funktion“ `newname`, dies ist jedoch nur mit Seiteneffekten möglich. Funktional ermöglicht man dies durch die Erweiterung von Basisnamen. Definiere hierzu eine Funktion `newlabel` für endgültige Namen und `extendLabel` für temporäre Erweiterung von Namen (für tiefere Codestrukturen).

```
newlabel :: Label -> Int -> Label
extendLabel :: Label -> Int -> Label

> newlabel (Label "xxx") 3
xxx_3
```

Entsprechend können auch `newtemp` und `extendtemp` definiert werden.

7.2.2 Übersetzung in Zwischencode

Wir nehmen nun an, daß die semantische Analyse durchgeführt wurde, daß die Speicherorganisation berechnet und Bezeichner entsprechend annotiert wurden.

Es wird nun eine feste temporäre Größe *RA* benutzt, die aktuelle **Rahmenadresse**, z.B. `(TEMP "RA")` Unsere Übersetzungsfunktion heißt `Trans`, die als Eingabe ein annotiertes Quellsprachenkonstrukt in abstrakter Syntax erwartet und Zwischencode ausgibt.

- **Konstanten**:

  ```
  Trans (NumExp i) = CONST i
  ```

- **Variablen**: Unterscheide zwischen Vorkommen auf der linken und rechten Seite einer Zuweisung:

  - *l-Wert*: Wert eines Ausdrucks, der links in einer Zuweisung auftaucht (eine Speicheradresse)
  - *r-Wert*: Wert eines Ausdrucks, der rechts in einer Zuweisung auftaucht, z.B. Inhalt einer Speicherzelle

  Dies ist auch relevant bei Prozedurparametern: *value*-Parameter entsprechen *r*-Werten, *var*-Parameter entsprechen *l*-Werten. Beachte: Konstanten haben einen *r*-Wert, aber keinen *l*-Wert! Für Variablen müssen wir nun unterschiedliche Trans-Funktionen angeben!

  - `TransL` liefert Code für *l*-Werte von Ausdrücken
  - `TransR` liefert Code für *r*-Werte von Ausdrücken
  - `TransS` liefert Code für Anweisungen

  Nun ergibt beispielsweise `TransR (NumExp i) = CONST i` (aber es erfolgt keine Definition von `TransL (NumExp i)`)!

Die Darstellung von Variablen erfolgt nun durch

(VarExp $\underbrace{\text{offset}}_{\substack{\text{relative Adresse} \\ im \text{Prozedur-rahmen}}}$ $\underbrace{\text{level}}_{\substack{\text{relative} \\ \text{Schachtelungs-tiefe}}}$)

```
TransL (VarExp offset 0) =
          BINOP PLUS (TEMP ra) (CONST offset)
TransL (VarExp offset l) =
  let deref level base =
        if level > 0
                then MEM (BINOP PLUS
                                (deref (level-1) base)
                                (CONST 1))
                else base
  in BINOP PLUS (deref l (TEMP ra)) (CONST offset)
```

Beispiel:

```
TransL (VarExp 15 2) = BINOP PLUS (MEM (BINOP PLUS
                         (MEM (BINOP PLUS (TEMP ra)
                                          (CONST 2)))
                         (CONST 15)))
```

Weiter kann `TransR` definiert werden:

```
TransR (VarExp offset l) = MEM (TransL (VarExp offset l))
TransR (OpExp e1 op e2) = BINOP op (TransR e1) (TransR e2)
TransR (FunExp f args) = CALL f (map TransR args)
```

Analog ist die Übersetzung von **Zeigervariablen** (bzw. *var*-Parametern), z.B. Dereferenzierung:

```
TransL (* exp) = TransR exp
TransR (* exp) = MEM (TransL (* exp))
```

Adreßoperator:

```
TransR (& exp) = TransL exp
```

Strukturierte Daten: Zugriff auf Recordkomponente (mit o_i Offset der Komponente c_i):

```
TransL (exp.c<i>) = BINOP PLUS (TransL exp) (CONST o<i>)
```

Analoger Code wird für **Felder** erzeugt, allerdings auch Code zur Überprüfung der Indexgrenzen! Dies ist sicher für einen sicheren Programmablauf! Alternative ist, daß eine Programmanalyse zeigt, daß die Indexgrenzen immer eingehalten werden!

- **Übersetzung von Anweisungen**: `TransS` hat die Parameter `bl` *(base label)* und `bt` *(base temp)* zur Generierung neuer Namen.

```
TransS bl bt (e1 := e2) = MOVE (MEM (TransL e1))
                               (TransR e2)
TransS bl bt (s1; s2) = SEQ (TransS bl1 bt1 s1)
                            (TransS bl2 bt2 s2)
              where bl1 = newlabel bl 1
                    bl2 = newlabel bl 2
                    bt1 = newtemp bt 1
                    bt2 = newtemp bt 2
TransS bl bt (if b then s1 else s2) =
       let t = newlabel bl 1
           f = newlabel bl 2
           e = newlabel bl 3
        in seq[ CJUMP NE (TransR b) (CONST 0) t j,
                LABEL t, TransS bl1 bt1 s1,
                         JUMP (NAME e),
                LABEL f, TransS bl2 bt2 s2,
                LABEL e]
           where bl1 = extendlabel bl 1
                 bl2 = extendlabel bl 2
                 bt1 = extendtemp bt 1
                 bt2 = extendtemp bt 2
TransS bl bt (while b do s) =
      let start = newlabel bl 1
          body = newlabel bl 2
          stop = newlabel bl 3
       in seq[ LABEL start,
               CJUMP EQ (TransR b) (CONST 0) stop body
               LABEL body, TransS bl1 bt s,
                           JUMP (NAME start),
               LABEL stop]
          where bl1 = extendlabel bl 1
```

Hierbei sei

$$\texttt{seq}[a_1, \ldots, a_n] = \texttt{SEQ}\ \ a_1\ \ \texttt{(SEQ}\ \ a_2\ \ \ldots\texttt{(SEQ}\ \ a_{n-1}\ \ a_n\texttt{)}\ \ \ldots\texttt{)}$$

Prozeduraufrufe können nun sowohl mit *value*-Parametern als auch mit *var*-Parametern übersetzt werden:

```
TransS bl bt (p(args)) = EXP
            (CALL p (map TransR args)) -- value
TransS bl bt (p(args)) = EXP
            (CALL p (map TransL args)) -- var
```

- **Übersetzung von Prozeduren**: Aktionen am Anfang und Ende einer Prozedur siehe oben (dynamischer/statischer Vorgänger setzen, ...) – dies ist einfach in Zwischencode übersetzbar; der Prozedurrumpf wird wie oben angegeben übersetzt.

- **Konstrollstrukturen für boolsche Ausdrücke**: Wir repräsentieren `false` durch 0 und `true` durch 1. Boolsche Ausdrücke können nun analog zu anderen Ausdrücken übersetzt werden, aber: häufig werden diese nur partiell ausgewertet, falls das ausreicht: Beispielsweise wird in C vom Ausdruck $e_1 \wedge e_2$ der Ausdruck e_2 nur ausgewertet, falls e_1 schon `true` ergibt. D.h. die Implementierung erfolgt durch Sprünge statt durch `BINOP`s:

$$e_1\ \texttt{\&\&}\ e_2 \longrightarrow (\texttt{if}\ e_1\ \texttt{then}\ r := 0\ \texttt{else}\ r := e_2, r)$$

Realisierung in Zwischencode:

```
let r = newtemp bt 1
    t = newlabel bl 1
    f = newlabel bl 2
    e = newlabel bl 3
 in ESEQ (seq [CJUMP EQ (TransR e_{1}) (CONST 0) t f,
               LABEL t, MOVE (TEMP r) (CONST 0),
                        JUMP (NAME e),
               LABEL f, MOVE (TEMP r) (TransR e_{2}),
               LABEL e])
         (TEMP r)
```

7.2.3 Basisblöcke

Es treten mit den allgemeinen Ausdrucksbäumen einige Probleme auf realen Architekturen auf:

- **Optimierung, um weniger Register zu verwenden:** Hierzu kann man den Code umordnen zur Berechnung von Teilausdrücken. Die ist schwierig wegen der Audrücke mit Seiteneffekten (`ESEQ`, `CALL`)!
- `CJUMP` **mit zwei Sprungzielen**: Reale Maschinenarchitekturen haben nur ein Sprungziel!
- **Prozeduraufrufe**: Die Übergabe von Argumenten in festen Registern stellt ein Problem bei geschachtelten `CALL`s dar (z.B. `CALL f [CALL g [CONST 1]]`)!

Lösung: transformiere Ausdrucksbäume in die kanonische Form der **Liste von Basisblöcken**! Dies geschieht in drei Phasen:

1. linearisiere (eliminiere `ESEQ`, `CALL`)
2. gruppiere Anweisungen in Basisblöcke
3. ordne die Basisblöcke so um, daß `CJUMP`-Sprungziele direkt folgen

7.2.4 Phase 1: Linearisierung

Ziel ist die **Eliminierung aller Seiteneffekte** in Ausdrücken, indem `ESEQ` und `CALL` in Ausdrücken nach oben gezogen werden und mit der umgebenden Anweisung vereinigt werden. Die Ausdrucksbäume sind Terme, daher können wir die Transformation durch **Termersetzungsregeln** beschreiben. Die `ESEQ`-Ausdrücke werden wie folgt behandelt:

1. Zusammenfassung von `ESEQ`s:

   `ESEQ` s_1 `(ESEQ` s_2 e`)` $\rightarrow$ `ESEQ (SEQ` s_1 s_2`)` e

2. Herausziehen von `ESEQ`s in linken Argumenten:

`BINOP` op `(ESEQ` s e_1`)` e_2	$\rightarrow$	`ESEQ` s `(BINOP` op e_1 e_2`)`
`MEM (ESEQ` s e`)`	$\rightarrow$	`ESEQ` s `(MEM` e`)`
`JUMP (ESEQ` s e`)`	$\rightarrow$	`ESEQ` s `(JUMP` e`)`
`CJUMP` op `(ESEQ` s e_1`)` e_2 l_1 l_2	$\rightarrow$	`SEQ` s `(CJUMP` op e_1 e_2 l_1 l_2`)`

3. `ESEQ` in rechten Argumenten: beachte Seiteneffekt von s auf e_1 (mit einem neuen temporären Wert t):

```
BINOP op e1 (ESEQ s e2)          →  ESEQ (MOVE (TEMP t) e1)
                                      (ESEQ s (BINOP op (TEMP t) e2))
CJUMP op e1 (ESEQ s e2) l1 l2    →  SEQ (MOVE (TEMP t) e1)
                                      (SEQ s (CJUMP op (TEMP t) e2 l1 l2))
```

Später wird diese rechte Seite dann noch vereinfacht. Falls die Auswertung von s den Ausdruck e_1 nicht beeinflußt (d.h. falls $\text{indep}(s, e_1)$):

```
BINOP op e1 (ESEQ s e2)          →  ESEQ s (BINOP op e1 e2)
CJUMP op e1 (ESEQ s e2) l1 l2    →  SEQ s (CJUMP op e1 e2 l1 l2)
```

Wichtig ist nun eine gute Approximation von indep (Verfeinerung ist möglich!):

$$\text{indep}(s,e) = \begin{cases} \texttt{true} & \text{falls} \quad \begin{cases} & e = (\texttt{NAME}\ldots) \\ \vee & e = (\texttt{CONST}\ldots) \\ \vee & s = \texttt{EXP}(\texttt{CONST}\ldots) \end{cases} \\ \texttt{false} & \text{sonst} \end{cases}$$

Bei folgendem Beispiel würde das Herausziehen von `MOVE` den Wert von `(MEM x)` beeinflussen:

```
BINOP PLUS (MEM x) (ESEQ (MOVE (MEM x) y) (CONST 1))
```

4. Weitere Regeln:

   ```
   MOVE (TEMP t) (ESEQ s e)  →  SEQ s (MOVE (TEMP t) e)
   EXP (ESEQ s e)            →  SEQ s (EXP e)
   ```
   Diese Liste ist noch zu ergänzen!

Die iterierte Anwendung der Regeln erzeugt `ESEQ`-freie Ausdrücke, da alle `ESEQ`s zunächst an die Wurzel von Ausdrücken gebracht wurden und dann alle `ESEQ`s an der Wurzel durch `SEQ` ersetzt wurden.

Nun muß noch die **Elimination geschachtelter** `CALL`s erfolgen: Ist zum Beispiel ein festes Register für den Rückgabewert von `CALL` vorgesehen, so entstehen Konflikte bei geschachtelten `CALL`s:

```
BINOP PLUS (CALL ...) (CALL ...)
```

Lösung mit einem neuen temporären Wert t:

`CALL` f $args$ → `ESEQ (MOVE (TEMP` t`) (CALL` f `args)) (TEMP` t`)` Dabei darf das neu erzeugte Vorkommen von `CALL` nicht erneut transformiert werden! Eliminiere nun `ESEQ` wie oben. Später erfolgt eine Elimination von t durch geschickte Registerallokation.

Nun sind alle Ausdrücke **seiteneffektfrei**, und alle `SEQ`-Operationen stehen oben. Es erfolgt daher eine Umwandlung der Bäume in eine **Liste von Anweisungen** (lineare Struktur)! Zunächst erfolgt noch die Umsetzung

$$\texttt{SEQ (SEQ } s_1\ s_2\texttt{)}\ s_3 \rightarrow \texttt{SEQ } s_1\ \texttt{(SEQ } s_2\ s_3\texttt{)}$$

Abschließend kann nun umgeformt werden:

$$\texttt{SEQ } s_1\ \texttt{(SEQ } s_2\ \texttt{...(SEQ } s_{n-1}\ s_n\texttt{)}\ \texttt{...)} \rightarrow [s_1, s_2, \ldots, s_{n-1}, s_n]$$

7.2.5 Phase 2: Gruppierung zu Basisblöcken

Wir wollen später Sprünge vereinfachen bzw. eliminieren. Notwendig ist dazu eine Unterteilung in `JUMP`/`LABEL` und `MOVE`-Folgen.

Ein **Basisblock** ist nun eine Instruktionsfolge, die immer am Anfang betreten und am Ende verlassen wird, d.h. die erste Instuktion ist ein `LABEL`, die letzte Instruktion ist ein `JUMP` oder ein `CJUMP`, alle inneren Instruktionen erhalten kein `LABEL` und kein (`C`)`JUMP`.

Diese Umsetzung ist nun einfach realisierbar durch eine Funktion

```
basicBlocks :: [Stm] -> [[Stm]]
```

Anmerkung: Bei Prozeduraufrufen erfolgt ein impliziter Sprung (`MOVE` ..., `CALL` ...). Dies könnte prinzipiell auch das Ende eines Basisblocks darstellen, das wird jedoch erst notwendig bei weiteren Optimierungen (Registerallokation).

7.2.6 Phase 3: Umordnung von Basisblöcken

Beobachtung: Die Anordnung der Basisblöcke ist beliebig! Wähle also eine Ordnung mit **möglichst wenigen Sprüngen**; ordne dazu die Basisblöcke zu (möglichst wenigen) größeren **Traces (Spuren)** um. Dabei ist eine Spur eine Instruktionsfolge, die fortlaufend ausgeführt werden kann, d.h. innerhalb kommen keine `JUMP`s, aber `CJUMP`s mit einer Zielmarke, die als nächste Instruktion nach dem `CJUMP` kommt.

Einfache Methode zur Anordnung der Basisblöcke zu Traces: Wähle Basisblock b und füge dahinter den Basisblock, zu dem verzweigt wird (auch *Nachfolger* von b, d.h. für $b = [\ldots, \texttt{JUMP } l]$ oder $b = [\ldots, \texttt{CJUMP } l\ l']$ sind Blöcke mit Label l oder l' Nachfolger von b).

Beispiel: für einen kompletten Trace:

$$[\texttt{LABEL } l_1, \ldots, \texttt{JUMP } l_3] +\!+ [\texttt{LABEL } l_3, \ldots, \texttt{CJUMP } l_2\ l_4] +\!+ [\texttt{LABEL } l_2, \ldots, \texttt{JUMP } l_1]$$

Ein Trace endet also, falls die Nachfolger bereits in einem Trace enthalten sind; markiere dazu die Basisblöcke entsprechend. Algorithmus:

$B :=$ ⟨Liste der Basisblöcke⟩ ;
`while` $B \neq [\,]$
  $T := [\,]$;
  $b :=$ `head`(B);
  $B :=$ `tail`(B);
  `while` $\neg$`marked`(b)
    `mark`(b);
    $T = T +\!+ b$;
    `if` $\exists c \in B$: $\neg$`marked`(c)
      $b := c$;
      $B := B \setminus c$;
  ⟨beende Trace T⟩

Auf vielen realen Maschinen sind bedingte Sprünge nur mit `true`-Label möglich, passe daher den Sprungcode an und modifiziere `CJUMP`, so daß dahinter immer das `false`-Label folgt:

- Hinter `CJUMP` folgt `true`-Label: negiere Bedingung und vertausche Labels!
- Hinter `CJUMP` folgt weder `true`- noch `false`-Label: wähle ein neues Label l und ersetze `CJUMP` op e_1 e_2 t f durch

  `CJUMP` op e_1 e_2 t l`; LABEL` l`; JUMP (NAME` f`)`

- Falls hinter `JUMP` l direkt `LABEL` l folgt, so lösche das `JUMP`.

Der obige Algorithmus versucht, möglichst lange Traces zu erzeugen – das ist nicht unbedingt am besten: *Beispiel:*

```
JUMP (NAME test)              JUMP (NAME test)
LABEL test                    -----------
CJUMP > i N done body         LABEL body
LABEL body                    <loop body>
<loop body>                   JUMP (Name test)
JUMP (NAME test)              LABEL test
-----------                   CJUMP > <= i N done body done
LABEL done                    LABEL done
...                           ...
```

Hier entsteht also in der rechten Version schnellerer Code, da kein unbedingter Sprung bei jedem Durchlauf benutzt wird.

7.3 Zielcodeauswahl

Jeder Knoten eines abstrakten Ausdrucksbaums enstpricht einer einfachen Operation (Speicherzugriff, Addition). Reale Maschinen bieten jedoch komplexe Instruktionen, die mehreren Knoten im Baum entsprechen, z.B. $M[r_i + c]$ kann als Ausdrucksbaum nur realisiert werden durch

`MEM (BINOP PLUS` v_i `(CONST` c`))`

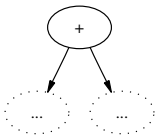
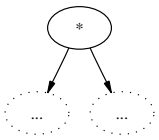
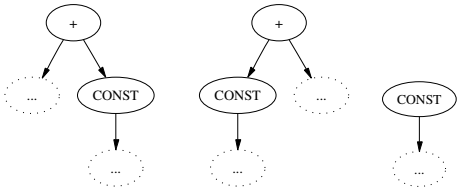
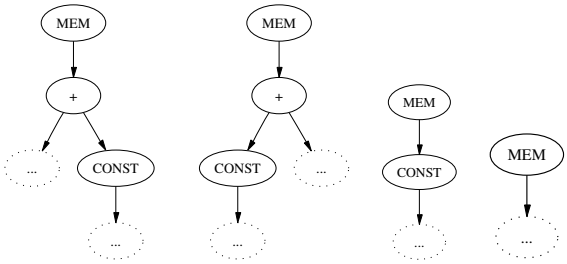
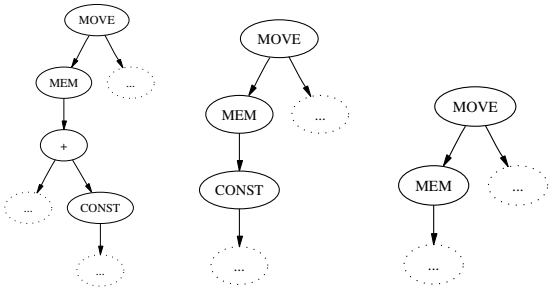
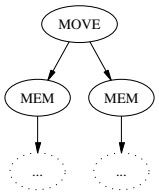
Die Aufgabe der Zielcodeauswahl ist nun, eine Maschineninstruktionsfolge zu finden, die den abstrakten Ausdrucksbaum implementiert. Die Lösung ist nicht eindeutig – wir suchen die **optimale** Folge. Stelle hierzu die Maschineninstruktionen als Teilbäume, sogenannte **Baummuster**, dar. Die Zielcodeauswahl entspricht dann der (optimalen) vollständigen Überdeckung des Ausdrucksbaums mit Baummustern.

Betrachte hierzu eine vereinfachte Maschine, reduziere dazu verschiedene Varianten, indem wir annehmen, daß das Register r_0 immer den Wert 0 enthält.

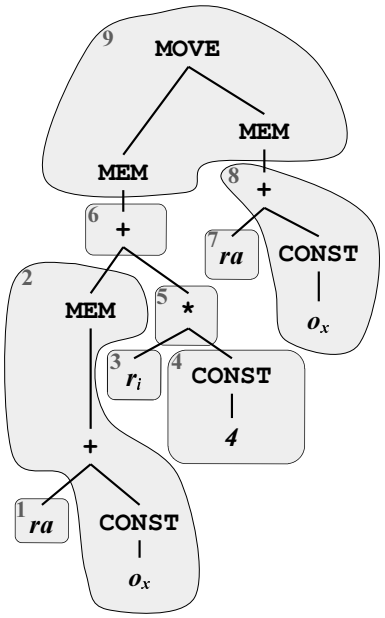
Pro Instruktion liegen hier mehrere Baummuster vor (siehe nächste Seite). Dies liegt zum einen an kommutativen Operatoren, zum anderen am speziellen Register (`ADDI`, `LOAD`, `STORE` benutzen r_0).

Die Ergebnisse von `ADD(I)`, `MUL`, `LOAD` etc. werden in Registern abgelegt. Wir benötigen neue Register bei der Übersetzung komplexer Bäume – durch Wiederverwendung vorhandener Register kann die Anzahl aber später in der Registerallokation reduziert werden.

Da Baummuster für alle Situationen notwendig sind, wird beispielsweise auch ein Baummuster für den Zugriff auf ein Register eingeführt: `TEMP` r_i entspricht einfach dem Register r_i (keine echte Maschineninstruktion); analog für Sprünge.

Name	Wirkung	Baummuster
`ADD`	$r_i \leftarrow r_j + r_k$	
`MUL`	$r_i \leftarrow r_j \cdot r_k$	
`ADDI`	$r_i \leftarrow r_j + c$	
`LOAD`	$r_i \leftarrow M[r_j + c]$	
`STORE`	$M[r_j + c] \leftarrow r_i$	
`MOVEM`	$M[r_i] \leftarrow M[R_j]$	

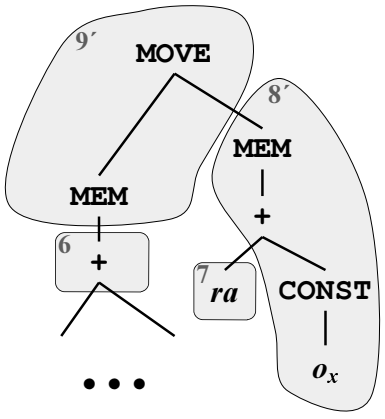
Beispiel: Quellprogramm sei $a[i] := x$; Annahme: Elemente von a haben eine Größe von vier Bytes; $RA + o_a$ sei die Anfangsadresse von a und $RA + o_x$ die Anfangsadresse von x; zudem erhalte r_i das Ergebnis i. Es ist folgende Überdeckung möglich:



Die Zwischencodeerzeugung ergibt also folgenden Code:

1	`LOAD`	$r_1 \leftarrow M[ra + o_a]$
4	`ADDI`	$r_2 \leftarrow r_0 + 4$
5	`MUL`	$r_2 \leftarrow r_i \cdot r_2$
6	`ADD`	$r_1 \leftarrow r_1 + r_2$
8	`ADDI`	$r_2 \leftarrow r_a + o_x$
9	`MOVEM`	$M[r_1] \leftarrow M[r_2]$

Es ist auch noch eine andere Überdeckung möglich:



1	`LOAD`	$r_1 \leftarrow M[ra + o_a]$
4	`ADDI`	$r_2 \leftarrow r_0 + 4$
5	`MUL`	$r_2 \leftarrow r_i \cdot r_2$
6	`ADD`	$r_1 \leftarrow r_1 + r_2$
8'	`LOAD`	$r_2 \leftarrow M[ra + o_x]$
9'	`STORE`	$M[r_1] \leftarrow r_2$

Es ist auch ein trivialer Code möglich, d.h. jeder Knoten entspricht in etwa einer Maschineninstruktion:

`ADDI`	$r_1 \leftarrow r_0 + o_a$
`ADD`	$r_1 \leftarrow r_a + r_1$
`LOAD`	$r_1 \leftarrow M[r_1 + 0]$
`ADDI`	$r_2 \leftarrow r_0 + 4$
`MUL`	$r_2 \leftarrow r_i \cdot r_2$
`ADD`	$r_1 \leftarrow r_1 + r_2$
`ADDI`	$r_2 \leftarrow r_0 + o_x$
`ADD`	$r_2 \leftarrow ra + r_2$
`LOAD`	$r_2 \leftarrow M[r_2 + 0]$
`STORE`	$M[r_1] \leftarrow r_2$

Dieser Code ist jedoch länger und wahrscheinlich auch langsamer! Es ist also wichtig, **guten Code** zu erzeugen – doch was ist guter Code und wie erzeugt man den? Wir wollen Code mit minimalen **Kosten** erzeugen, Maß dafür beispielsweise:

- Länge aller Instruktionen (z.B. für kleinen Code von Applets, die über ein (langsames) Netz geladen werden)
- Ausführungszeit aller Instruktionen

Definition: Die Überdeckung eines Baumes mit Baummustern heißt

- *lokal optimal*, falls zwei benachbarte Muster nicht zu einem einzigen Muster mit kleineren Kosten kombiniert werden können;
- *optimal*, falls es keine Überdeckung mit kleineren Kosten gibt.

Beispiel: Jede Instruktion möge hier vereinfacht 2 Einheiten kosten bis auf `MOVEM`, die m Einheiten kostet. Falls $m > 2$ ist, ist der zweite Zielcode mit $8'$ und $9'$ optimal. Falls $m < 2$ ist, ist der erste Zielcode mit 8 und 9 optimal. Falls $m = 2$ ist, so sind beide optimal. Unabhängig von m sind beide lokal optimal.

7.3.1 Algorithmen für (lokal) optimale Zielcodes

Intuitiv ist lokal optimaler Code einfacher zu finden als optimaler Code. Unterscheide vor allem zwischen zwei wesentlichen Architekturen:

- **CISC** (complex instruction set computer): komplexe Operationen, die unterschiedliche Kosten haben – daher wesentliche Unterschiede zwischen lokal optimalem und optimalem Code
- **RISC** (reduced instruction set computer): einfache Operationen, keine großen Kostenunterschiede – daher kaum Unterschiede zwischen lokal optimalem und optimalem Code

Der Algorithmus *Maximal Munch (maximales Mampfen)* erzeugt lokal optimalen Zielcode:

- Vorbereitung: Falls ein Baummuster existiert, das in zwei kleinere Baummuster mit geringeren kombinierten Kosten aufteilbar ist, dann lösche dieses Muster
- Überdecke dann die Wurzel mit dem größten passenden Muster (d.h. mit den meisten Knoten) und wende den Algorithmus rekursiv auf alle nichtüberdeckten Teilbäume an.

Wähle also im obigen Beispiel `MOVEM` als größtes Muster für die Wurzel.

Implementierung in Haskell: Überdeckungen entsprechen pattern matching; die maximale Überdeckung erreicht man durch Angabe des größten Musters zuerst.

```
munchStm :: Stm -> [String]
munchStm (MOVE (MEM (BINOP PLUS e1 (CONST i))) e2) =
    munchExp e1 ++ munchExp e2 ++ codegen "STORE"
munchStm (MOVE (MEM (BINOP PLUS (CONST i) e1)) e2) =
    munchExp e1 ++ munchExp e2 ++ codegen "STORE"
munchStm (MOVE (MEM e1) (MEM e2)) =
    munchExp e1 ++ munchExp e2 ++ codegen "MOVEM"
munchStm (MOVE (MEM (CONST i)) e) =
    munchExp e ++ codegen "STORE"
...
munchExp :: Exp -> [String]
munchExp (MEM (BINOP PLUS e (CONST i))) =
    munchExp e ++ codegen "LOAD"
munchExp (MEM (BINOP PLUS (CONST i) e)) =
```

```
    munchExp e ++ codegen "LOAD"
munchExp (MEM (CONST i)) = codegen "LOAD"
munchExp (MEM e) = munchExp e ++ codegen "LOAD"
munchExp (BINOP PLUS e (CONST i)) =
    munchExp e ++ codegen "ADDI"
munchExp (BINOP PLUS (CONST i) e) =
    munchExp e ++ codegen "ADDI"
munchExp (BINOP PLUS e1 e2) =
    munchExp e1 ++ munchExp e2 ++ codegen "ADD"
munchExp (CONST i) = codegen "ADDI"
munchExp (TEMP t) = []

codegen c = [c] -- vereinfacht!
```

Anmerkungen:

- Hier ist nur Pattern Matching angewandt worden, notwendig ist noch die Registerauswahl für das Ergebnis (in `munchExp`).
- Die Instruktionen werden in umgekehrter Reihenfolge generiert!
- Der Algorithmus *MaximalMunch* terminiert immer erfolgreich, falls es für jeden Einzelknotentypen ein passendes Baummuster gibt.
- Der Algorithmus *MaximalMunch* findet immer eine lokal optimale Überdeckung.

7.3.2 Berechnung optimaler Zielcodes

Wir nutzen das Prinzip der **dynamischen Programmierung**: Finde optimale Lösungen durch Finden von optimalen Lösungen für Teilprobleme.

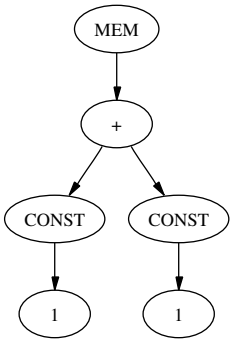
Idee dabei: Jeder Knoten enthält bestimmte Kosten für beste Instruktionsfolge, die diesen Teilbaum überdeckt. Bottum-Up-Algorithmus: Berechne zunächst die Kosten für die Blätter, dann die darüberliegende Schicht usw.

Die Kostenberechnung für einen Knoten läuft folgendermaßen ab:

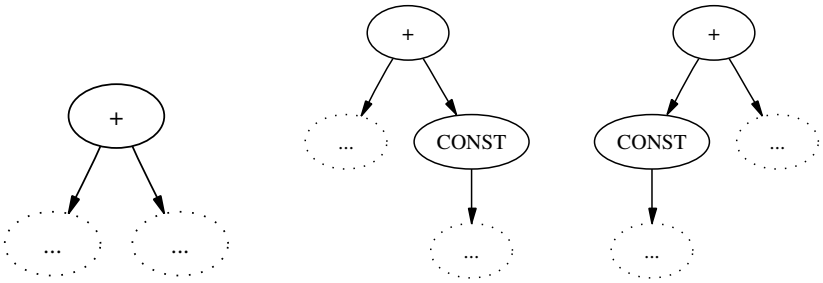
- Betrachte ein passendes Baummuster für diesen Knoten, dieses hat Kosten c.
- Die Blätter dieses Muster entsprechen Knoten im Baum, deren Kosten $c_1, c_2, \ldots, c_n$ bereits berechnet sind; die Gesamtkosten für dieses Muster ergeben sich daher als $c + \sum_{i=1}^{n} c_i$.

- Prüfe alle passenden Baummuster und wähle das mit den geringsten Gesamtkosten aus, das ergibt die Kosten für den betrachteten Knoten.

Beispiel: Betrachte folgenden Teilbaum:

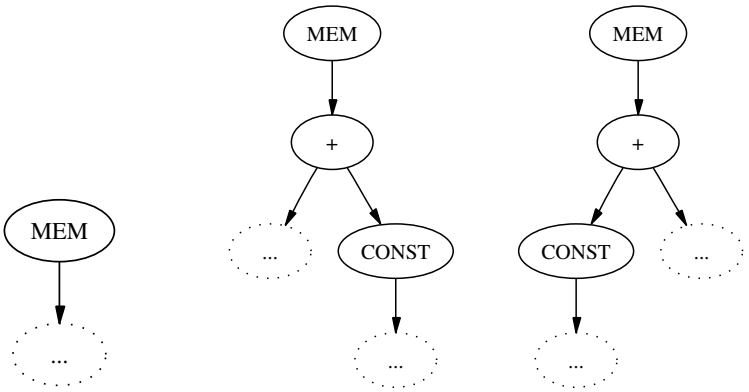


Jede Instruktion soll zwei Einheiten kosten. Die `CONST`-Muster entsprechen `ADDI` und kosten entsprechend zwei Instruktionen. Möglichkeiten für eine Überdeckung des Teilbaums:



`ADD`: $2 + 2 + 2 = 6$ `ADDI`: $2 + 2 = 4$ `ADDI`: $2 + 2 = 4$

Damit hat der `+`-Knoten die Kosten 4. Der `MEM`-Knoten kann folgendermaßen überdeckt werden:



`LOAD`: $4 + 2 = 6$ `LOAD`: $2 + 2 = 4$ `LOAD`: $2 + 2 = 4$

Somit hat auch der `MEM`-Knoten Kosten von 4. Der kostengünstigste Gesamtcode ist also

$$\begin{array}{ll} \texttt{ADDI} & r_1 \leftarrow r_0 + 1 \\ \texttt{LOAD} & r_1 \leftarrow M[r_1 + 2] \end{array}$$

7.4 Programmanalyse

Nach der bisherigen Annahme haben wir beliebig viele temporäre Werte (d.h. Register) zur Verfügung. In der Praxis stehen weniger Register zur Verfügung – ein Register muß also für mehrere temporäre Werte benutzt werden, falls die temporären Werte nicht gleichzeitig benötigt werden. Der Compiler muß die **Lebendigkeit** von temporären Werten analysieren!

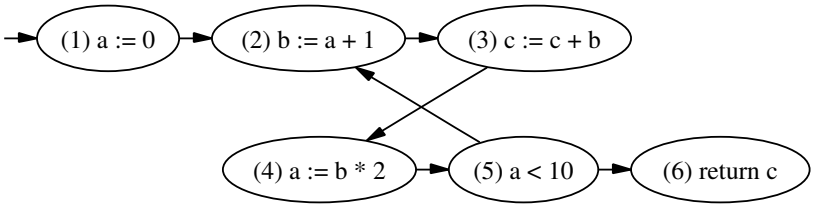
Variablen sind **lebendig** (an einer Programmstelle), wenn deren Wert später noch verwendet wird, daher führen wir eine **Lebendigkeitsanalyse (liveness analysis)** in Form einer „Rückwärtsanalyse“ durch:

- Bei Programmende ist keine Variable lebendig, und die LA arbeitet dann vom Ende zum Anfang.
- Geeignete Datenstruktur ist ein **Kontrollflußgraph**, die Knoten entsprechen den Anweisungen im Code, und Kanten von x nach y besagen, daß die Anweisung y direkt nach Anweisung x ausgeführt werden kann.

Beispiel:

```
    a := 0
L1: b := a + 1
    c := c + b
    a := b * 2
    if a < 10 goto L1
    return c
```

Der zugehörige Kontrollflußgraph:



Betrachte nun die einzelnen Variablen:

- Variable b:
  - benutzt in 4, daher lebendig in $3 \rightarrow 4$
  - nicht überschrieben in 3, daher lebendig in $2 \rightarrow 3$
  - überschrieben in 2, daher tot in $1 \rightarrow 2$
- Variable a:
  - lebendig in $1 \rightarrow 2$, $5 \rightarrow 2$, $4 \rightarrow 5$
  - tot in $3 \rightarrow 4$, $2 \rightarrow 3$ (obwohl sie einen Wert hat!)
- Variable c: lebendig im gesamten Programm, daher auch lebendig am Anfang: nicht initialisierte Variable (zum Beispiel Compilerwarnung oder Initialisierungsanweisung) falls c lokale Variable ist (kein Prozedurparameter)

Damit sind a und b nicht gleichzeitig lebendig, benutze also das gleiche Register für a und b, d.h. zwei Register reichen aus.

In dieser Analyse „fließen“ Informationen durch den Graphen, man bezeichnet diese Klasse von Algorithmen auch als **Datenflussanalyse**. Weitere Analyseprobleme:

- **Konstantenpropagation**: Welche Variablen oder Ausdrücke haben konstante Werte?
- **Sharing-Analyse:** Welche Variablen und Datenstrukturen haben gemeinsame Teile?
- **Dead Code**: Welcher Code ist unerreichbar bzw. wird nicht benutzt?

Diese Techniken sind im Prinzip ähnlich, im Folgenden wird daher nur die Lebendigkeitsanalyse behandelt.

Definitionen:

- Knoten n hat *out*-Kanten (führen zu Nachfolgeknoten $\text{succ}(n)$) und *in*-Kanten (führen zu Vorgängerknoten $\text{prec}(n)$)

  Beispiel: Knoten 5 hat *out*-Kanten $5 \rightarrow 6$ und $5 \rightarrow 2$ und *in*-Kante $4 \rightarrow 5$.
- Eine Zuweisung im Knoten n an Variable v *definiert* v, damit $\text{def}(n) = \{v\}$.

  Beispiel: $\text{def}(3) = \{c\}$, $\text{def}(5) = \emptyset$

- Die Benutzung von Variable v *benutzt* v, damit sei $v \in \text{use}(n)$.
  Beispiel: $\text{use}(3) = \{b, c\}$, $\text{use}(5) = \{a\}$
- Eine Variable v ist *lebendig* in der Kante e, wenn ein Pfad von e zu n existiert mit $v \in \text{use}(n)$, und es gilt $v \notin \text{def}(m)$ für alle Knoten $m \neq n$ auf dem Pfad.
- $\text{in}(n)$ ist die Menge der lebendigen Variablen in *in*-Kanten von n
  Beispiel: $\text{in}(1) = \{c\}$
- $\text{out}(n)$ ist die Menge der lebendigen Variablen in *out*-Kanten von n
  Beispiel: $\text{out}(1) = \{a, c\}$

Es ergeben sich nun folgende **Regeln für die Lebendigkeitsanalyse**:

- $v \in \text{use}(n) \Rightarrow v \in \text{in}(n)$
- $v \in \text{in}(n) \Rightarrow v \in \text{out}(n) \; \forall \; m \in \text{pred}(n)$
- $v \in \text{out}(n) \wedge v \notin \text{def}(n) \Rightarrow v \in \text{in}(n)$

Damit können wir ein **Gleichungssystem** für den Kontrollflußgraphen aufstellen; die Analyse muß das Gleichungssystem lösen! Für alle Knoten n gelten folgende Gleichungen:

$$\begin{aligned} \text{in}(n) &= \text{use}(n) \cup (\text{out}(n) \setminus \text{def}(n)) \\ \text{out}(n) &= \bigcup_{m \in \text{succ}(n)} \text{in}(m) \end{aligned}$$

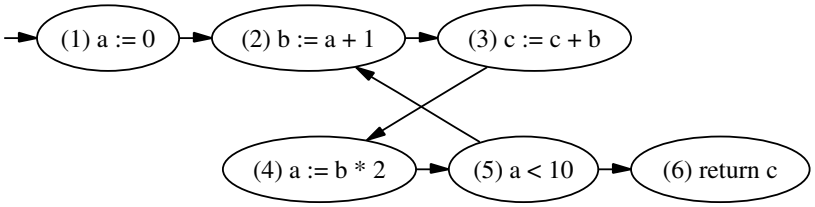
Variablen in diesem Gleichungssystem sind

$$\bigcup_{n\{\text{Knoten}\}} \{\text{in}(n), \text{out}(n)\}$$

Berechnung einer Lösung geschieht durch **Fixpunktiteration**:

```
for each n
  in(n) = {}
  out(n) = {}
repeat
  for each n
    in'(n) = in(n)
    out'(n) = out(n)
    in(n) = use(n) ∪ (out(n) \ def(n))
    out(n) = ⋃_{m∈succ(n)} in(m)
until  in(n) == in'(n) ∧ out(n) == out'(n)
```

Im Beispiel von oben:



Dann ergibt sich folgende Iteration:

			Runde 1		Runde 2		Runde 3		Runde 4		Runde 5		Runde 6	
n	use	def	in	out	in	out	in	out	in	out	in	out	in	out
1		a				a		a		a, c	c	a, c	c	a, c
2	a	b	a		a	b, c	a, c	b, c	a, c	b, c	a, c	b, c	a, c	b, c
3	b, c	c	b, c		b, c	b	b, c	b	b, c	b	b, c	b	b, c	b, c
4	b	a	b		b	a	b	a	b	a, c	b, c	a, c	b, c	a, c
5	a		a	a	a	a, c	a, c	a, c	a, c	a, c	a, c	a, c	a, c	a, c
6	c		c		c		c		c		c		c	

Die siebte Runde ergibt die gleichen Ergebnisse wie Runde 6, der Algorithmus terminiert daher.

Zur **Berechnungsstrategie**: Die Ordnung in der inneren „`for each`“-Schleife für Knoten ist beliebig. Wähle daher die Ordnung mit „schnellem“ Datenfluss, d.h. da die Abhängigkeit $\text{out}(n) = \bigcup_{m \in \text{succ}(n)} \text{in}(m)$ besteht, bestimmen die Nachfolger die Werte der Vorgänger. Berechne daher von den Nachfolgern zu den Vorgängern (d.h. hier in Knotenordnung 6, 5, 4, 3, 2, 1). In jeder Iteration wird zunächst out, dann in berechnet, damit ergibt sich folgende verkürzte Ausführung, bei der das Ergebnis der dritten Iteration schon dem der zweiten entspricht:

			Runde 1		Runde 2	
n	use	def	out	in	out	in
6	c			c		c
5	a		c	a, c	a, c	a, c
4	b	a	a, c	b, c	a, c	b, c
3	b, c	c	b, c	b, c	b, c	b, c
2	a	b	b, c	a, c	b, c	a, c
1		a	a, c	c	a, c	c

7.5 Registerallokation

Die bisherige Annahme war, daß beliebig viele Register zur Verfügung stehen. Real stehen jedoch nur endlich viele zu Verfügung, daher ist die Aufgabe der **Registerallokation** (falls möglich) die Zuordnung von temporären Werten zu konkreten Registern – und damit die Einsparung von `MOVE`/`LOAD`-Instruktionen durch die Wiederverwendung von Registern.

Generell geht man **sequentiell** durch den Code:

- Bei jedem Vorkommen eines temporären Wertes: Liegt dieser in einem Register? Falls nein, in welches legen? Annahme ist hier, daß alle Operationen auf Registern arbeiten (z.B. RISC).
- Falls ein temporärer Werte nicht im Register liegt, jedoch benötigt wird: Wähle ein Register aus und erzeuge eine entsprechende `LOAD`-Instruktion.
- Alle temporären Werte, die nicht in Registern gehalten werden können, werden im Prozedurspeicher abgelegt (temporäre Werte im Prozedurrahmen). Der Compiler muß dann die relative Adresse zuweisen.

Hier wird die Registerallokation für **Basisblöcke** vorgestellt; allgemein werden kompliziertere Techniken angewendet wie z.B. die Datenflußanalyse (wo werden welche temporären Werte benötigt?). Dies geschieht z.B. mit Hilfe von Konfliktgraphen (mit temporären Werten als Knoten und Kanten zwischen zwei Knoten, falls die beiden temporäre Werte gleichzeitig existieren) – die Registerzuordnung erfolgt dann durch Graphfärbung.

Die Menge der wirklich **freien Register** ist eine Teilmenge der Maschinenregister, da einzelne für die Rahmenadresse, den Stack- und Heap-Pointer etc. reserviert sind. Zu jeder Programmvariable und jedem temporären Wert v sei ein Speicherplatz M_v im Stack reserviert (dies wird für temporäre Werte evtl. nicht benötigt).

Allgemein verwaltet man folgende Informationen:

- **Registerinhalte**

  $$\mathrm{Con}\colon \mathrm{Register} \to 2^{\mathrm{Variablen}}$$

  Hierbei sind Variablen die Programmvariablen und die temporären Werte. Nun ist $\mathrm{Con}(R)$ die Menge der Variablen, die momentan im Register R gespeichert sind – zu Beginn eines Basisblocks ist $\mathrm{Con}(R) = \emptyset$ für alle Register R; und $\mathrm{Con}(R)$ kann eventuell mehrelementig sein bei Kopieroperationen.

- **Variablenposition**

  $$\text{Pos}\colon \text{Variablen} \to \text{Register} \cup \text{Speicher}$$

  Dann gibt $\text{Pos}(v)$ die Stelle an, wo sich der aktuelle Wert von v befindet (d.h. ein R oder M_v). Zu Beginn eines Basisblocks ist $\text{Pos}(v) = M_v$ für alle Programmvariablen; aber $\text{Pos}(v)$ kann undefiniert sein, falls nicht mehr benötigt.

- **Lebendigkeit von Variablen**:

  $$\text{Live}\colon \text{Variablen} \to \text{Bool}$$

  An einer Programmstelle ist $\text{Live}(v) = \texttt{true}$, falls v später noch benötigt wird. Zum Ende eines Basisblocks ist $\text{Live}(v) = \texttt{false}$ für alle temporären Werte. Die Berechnung erfolgt durch Rückwärtspropagation: Falls in der Instruktion I der Wert x überschrieben und der Wert y benutzt wird, so ist $\text{Live}(x) = \texttt{false}$ und $\text{Live}(y) = \texttt{true}$; alles andere ist unverändert für Live von I (Spezialisierung des vorigen Kapitels).

Zentrale **Hilfsoperation** für die Registerallokation ist eine Prozedur `getreg()`, die ein Register zur Speicherung einer Variablen bestimmt (eigentlich ist dies eine Funktion Con / Pos / Live $\to$ Con / Pos + Code).

Verwende folgenden **Algorithmus** für `getreg()`:

- Falls ein Register R mit $\text{Con}(R) = \{v\}$ und $\text{Live}(v) = \texttt{false}$ existiert, dann $\text{Con}(R) := \emptyset$ und `return` R (Wiederbenutzung eines nicht mehr benötigten Registers).
- Wähle sonst ein freies Register R (d.h. mit $\text{Con}(R) = \emptyset$) und gib R zurück.
- Falls keines frei ist: Wähle ein belegtes Register R: Erzeuge für alle $v \in \text{Con}(R)$ mit $\text{Pos}(v) \neq M_v$ einen Befehl `STORE` $M[M_v] \leftarrow R$ und setze $\text{Pos}(V) := M_v$. Setze dann $\text{Con}(R) := \emptyset$ und gib R zurück.

  Die Auswahl von R ist beliebig, da hier kein bestes Verfahren bekannt ist.

Hiermit ist dann realer Maschinencode generierbar: Wenn bisheriger Code ein Register benötigt, bestimme ein „reales“ Register mit `getreg()`.

Der **Algorithmus für die Registerallokation**:

- Betrachte nacheinander alle Instruktionen eines Basisblocks (in Ausführungsreihenfolge), sei instr die aktuelle Instruktion:

(a) Für jedes (virtuelle) Register r_i, das als Operand in instr vorkommt[7]:

1. Wähle R_i mit $r_i \in \mathrm{Con}(R_i)$, sonst $R_i :=$ `getreg()`.
2. Falls $\mathrm{Pos}(r_i) \neq R_i$ ist, so füge `LOAD` $R_i \leftarrow M[M_{r_i}]$ *vor* instr ein und setzte $\mathrm{Con}(R_i) = \{r_i\}$.

(b) Falls das Ergebnis von instr in (virtuelles) Register r_j gespeichert wird:

1. Falls oben ein neues Operandenregister für instr mit `getreg()` gewählt wurde, dann benutze dieses für das Ergebnis r_j.
2. Sonst setze $R :=$ `getreg()` (wie oben: eventuell `STORE`-Instruktion *vor* instr einfügen) und benutze R für das Ergebnis.

(c) Erzeuge die Maschineninstruktion für instr mit dem in (a) und (b) gewählten Registern.

(d) Aktualisiere Con und Pos entsprechend der Semantik von instr:

- `ADD` $r_i \leftarrow r_j + r_k$ und R für r_i gewählt: Dann setze $\mathrm{Con}(R) = \{r_i\}$ und $\mathrm{Pos}(r_i) = R$.
- `ADDI` $r_i \leftarrow r_j + c$ und R für r_i gewählt:
  Setze $\mathrm{Con}(R) = \{r_i\}$ und $\mathrm{Pos}(r_i) = R$.
- `ADDI` $r_i \leftarrow r_j + 0$ und R für r_j gewählt:
  Setze $\mathrm{Con}(R) = \{r_i, r_j\}$ und $\mathrm{Pos}(r_i) = R$.
- `LOAD` $r_i \leftarrow M[r_j + c]$ und R für r_i gewählt und $r_j + c$ Adresse der Variablen v:
  Setze $\mathrm{Con}(R) = \{r_i, v\}$ und $\mathrm{Pos}(r_i) = R$.
- `STORE` $M[r_i + c] \leftarrow r_j$ und $r_i + c$ Adresse der Variablen v:
  Setze $\mathrm{Pos}(v) = M_v$.
- ...

(e) Aktualisiere Con bezüglich Live-Information nach der Instruktion instr: Für alle Register S mit $x \in \mathrm{Con}(S)$ und $\mathrm{Live}(x) =$ `false` setze $\mathrm{Con}(S) = \mathrm{Con}(S) \setminus \{x\}$.

(f) Falls in (b) Ergebnisregister R für r_j gewählt wurde: Für alle Register $S \neq R$ mit $r_i \in \mathrm{Con}(S)$ setze $\mathrm{Con}(S) = \mathrm{Con}(S) \setminus \{r_i\}$.

- Erzeuge am Ende eines Basisblocks für alle Variablen v mit $\mathrm{Live}(v) =$ `true` und $\mathrm{Pos}(v) = R$ die Instruktion `STORE` $M[M_v] \leftarrow R$.

[7] Als Operand (d.h. nicht als Ziel) kommt ein Register in allen Fällen auf der rechten Seite vor (etwa `ADD`$r_i \leftarrow r_j + r_k$), aber auch als Adressregister in der linken Seite, d.h. bei `STORE`$M[r_i + c] \leftarrow r_j$ ist auch r_i ein Operand!

Zur Integration in die Codeauswahl: Bei der Übersetzung von Ausdrucksbäumen wird `getreg()` zur Auswahl von Registern für Zwischenergebnisse benutzt!

Beispiel: Annahme, es stehen Maschinenregister R, S und T zur Verfügung; im Programm wird die Variable w verwendet, alle anderen Variablen seien temporär. Betrachte folgende Instruktionsfolge (ohne Befehlsnamen):

Instruktion	Live(Out)
$y \leftarrow 1$	$\{w, y\}$
$z \leftarrow w$	$\{y, z\}$
$x \leftarrow y + z$	$\{x, y, z\}$
$v \leftarrow x + y$	$\{x, y, z, v\}$
$t \leftarrow z \cdot v$	$\{t, x, y\}$
$u \leftarrow y \cdot x$	$\{u, t\}$
$w \leftarrow u + t$	$\{w\}$

Dann ergibt sich:

Instruktion	Code	Con	Pos	Kommentar
$y \leftarrow 1$	$R \leftarrow 1$	$R \mapsto \{y\}$	$y \mapsto R$	`getreg()` $= R$
$z \leftarrow w$	$S \leftarrow M[M_w]$	$R \mapsto \{y\}$ $S \mapsto \{z\}$	$y \mapsto R$ $z \mapsto S$	`getreg()` $= S$
$x \leftarrow y + z$	$T \leftarrow R + S$	$R \mapsto \{y\}$ $S \mapsto \{z\}$ $T \mapsto \{x\}$	$y \mapsto R$ $z \mapsto S$ $x \mapsto T$	`getreg()` $= T$ (für x)
$v \leftarrow x + y$	$M[M_z] \leftarrow S$	$R \mapsto \{y\}$ $S \mapsto \{v\}$ $T \mapsto \{x\}$	$y \mapsto R$ $z \mapsto M_z$ $x \mapsto T$ $v \mapsto S$	`getreg()` $= S$ (für v)
$t \leftarrow z \cdot v$	$M[M_x] \leftarrow T$ $T \leftarrow M[M_z]$ $T \leftarrow T \cdot S$	$R \mapsto \{y\}$ $S \mapsto \emptyset$ $T \mapsto \{z\}$ $T \mapsto \{t\}$	$y \mapsto R$ $z \mapsto M_z$ $x \mapsto M_x$ $t \mapsto T$	`getreg()` $= S$ (für z) $M[M_x] \leftarrow T$
$u \leftarrow y \cdot x$	$S \leftarrow M[M_x]$ $S \leftarrow R \cdot S$	$R \mapsto \emptyset$ $S \mapsto \{u\}$ $T \mapsto \{t\}$	$y \mapsto R$ $z \mapsto M_z$ $x \mapsto M_x$ $t \mapsto T$ $u \mapsto S$	`getreg()` $= S$ (für x)
$w \leftarrow u + t$	$R \leftarrow S + T$ $M[M_w] \leftarrow R$	$R \mapsto \{w\}$ $S \mapsto \emptyset$ $T \mapsto \emptyset$	$w \mapsto R$	`getreg()` $= R$ (für w)

7.6 Techniken zur Code-Optimierung

Ziel ist die Erzeugung „möglichst guten“ Zielcodes (wobei „optimal“ nicht ganz richtig ist...): Zum einen schnelle Ausführung, zum anderen wenig Laufzeitspeicher; aber auch geringe Codegröße (z.B. bei applets oder eingebetteten Systemen wichtig).

Grundsätzliche Methoden sind:

- **Algebraische Optimierung**: Gesetze der Quellsprache werden ausgenutzt, um Transformationen auf Quellsprachebene vorzunehmen.
- **Partielle Auswertung**: Auswertung bekannter Teile im Quellprogramm (siehe 2.3)
- **Maschinenunabhängige Optimierung**: Transformationen auf Zwischensprachenebene
  - **lokale Optimierung**: betrachte kleinere Programmausschnitte (z.B. Basisblöcke)
  - **globale Optimierung**: betrachte größere Einheiten (z.B. Prozeduren, aber beispielsweise auch das gesamtes Programm)
    - **Datenflußanalyse**: liefert Infos zum Laufzeitverhalten durch Kontroll- und Datenflußanalyse des Ablaufgraphen
    - **abstrakte Interpretation**: liefert Infos zum Laufzeitverhalten durch „abstrakte“ Programmausführung, d.h. mit abstrakten Werten statt mit konkreten Werten (z.B. Vorzeichen statt konkreter Zahl)
- **maschinenabhängige Optimierung**: Ausnutzung spezieller Maschineneigenschaften, z.B. Adressierungsmodi oder Registerauwahl

7.6.1 Algebraische Optimierung

Diese Technik nutzt Gesetze der Quellsprache (beispielsweise $0 * x \rightsquigarrow 0$), was in imperativen Sprachen wegen der Seiteneffekte selten vorkommt. Relevant ist diese Technik hingegen bei deklarativen Sprachen, z.B. Haskell:

```
foldr :: (a->b->b)->b->[a]->b
fordr f e [] = e
foldr f e (x:xs) = f x (foldr f e xs)

map :: (a->b)->[a]->[b]
```

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

sum = foldr (+) 0
qu x = x*x
```

Sei nun $e =$ `sum (map qu (map qu` xs`))` ein Teilausdruck im Programm. Die Auswertung von e mit $xs = [1, 2, 3]$ erzeugt nun zwei **Zwischendatenstrukturen** $[1, 4, 9]$ und $[1, 16, 81]$; wobei diese Zwischendatenstrukturen kein Bestandteil des Endergebnisses sind – der Aufbau kostet jedoch Zeit und Speicher. Dies ist typisch für funktionalen/kompositionellen Programmierstil. Vermieden werden können solche ungünstigen Teile durch Quellsprachentransformation.

LEMMA: Für alle f, g, xs gilt: `map` f `(map` g xs`) = map (`$f \cdot g$`)` xs

Beweis: durch strukturelle Induktion über xs:

- Induktionsanfang: für $xs =$ `[]` gilt:

$$\begin{aligned} \texttt{map}\ f\ (\texttt{map}\ g\ []) &= \texttt{map}\ f\ [] \\ &= [] = \texttt{map}\ (f \cdot g)\ [] \end{aligned}$$

- Induktionsannahme: Das Lemma gelte für xs; zu zeigen ist die Behauptung für $x : xs$; es gilt:

$$\begin{aligned} \texttt{map}\ f\ (\texttt{map}\ g\ x : xs) &= \texttt{map}\ f\ (gx\ \texttt{:map}\ g\ xs) \\ &= (f\ (g\ x))\texttt{:(map}\ f\ (\texttt{map}\ g\ xs)) \\ \text{I.A.} &= (f\ (g\ x))\texttt{:(map}\ (f \cdot g)\ xs)) \\ &= (f \cdot g\ x)\texttt{:(map}\ (f \cdot g)\ xs)) \\ &= \texttt{map}\ (f \cdot g\ x)\ x : xs \end{aligned}$$

LEMMA: Für alle f, g, e, xs gilt:

`foldr` f e `(map` g xs`) = foldr (\` x y `->` f `(`g x`)` y`)` e xs

Der Beweis hierzu ist analog wie oben zu führen.

Nun läßt sich obiger Ausdruck e reduzieren zu:

$$\begin{aligned} e &= \texttt{foldr (+) 0 (map qu (map qu}\ xs\texttt{))} \\ &= \texttt{foldr (\x y -> qu (qu x) + y) 0}\ xs \end{aligned}$$

Hier wurde die **Seiteneffektfreiheit** ausgenutzt; diese Technik findet daher Anwendung bei funktionalen Sprachen; arithmetischen Ausdrücken in imperativen Sprachen; aber auch in relationalen Anfragesprachen (siehe Datenbanken).

7.6.2 Partielle Auswertung

Motivation ist, daß eine algebraische Auswertung stattfinden soll, dies jedoch automatisiert werden soll. Dazu wird das **Quellprogramm ausgewertet** – wo immer das möglich ist – und dadurch simplifiziert (siehe 2.3). **Problem** ist, daß die partielle Auswertung immer terminieren sollte, selbst bei Endlosschleifen (hier wird *loop checking* angewandt).

Beispiel:

```
  sum (map qu [1, 2])
= sum (qu 1 : map qu [2])
= sum (qu 1 : qu 2 : map [])
= sum (qu 1 : qu 2 : [])
= sum (1*1 : 2*2 : [])
= sum (1 : 4 : [])
= foldr (+) 0 (1:4:[])
= 1 + foldr (+) 0 (4:[])
= 1 + (4 + foldr (+) 0 [])
= 1 + (4 + 0)
= 1 + 4
= 5
```

Auch partielle Auswertung mit Variablen ist möglich: Beispielsweise kann man mögliche Werte für bestimmte Variablen testen:

```
e = sum (map qu ys)
  = foldr (+) 0 (map qu ys)
  = if ys == []   then 0
       ya == x:xs then foldr (+) 0 (map qu x:xs)
                       = foldr (+) 0 (qu x : map qu xs)
                       = qu x + foldr (+) 0 (map qu xs)
                       = x*x + foldr (+) 0 (map qu xs)
```

Weiter kann nicht ausgewertet werden, sonst läuft der Partielle Auswerter in eine Endlosschleife. Doch nun wird ein neue Funktion für den komplexen Ausdruck definiert: die Funktion `f` soll die Semantik `f ys = foldr (+) 0 (map qu ys)` haben, durch obige Berechnungen ergibt sich:

```
f    []  = 0
f (x:xs) = x*x + f xs

e = f ys
```

Diese Lösung erzeugt keine Zwischendatenstrukturen! D.h. wir haben schnelleren Code (`sum`, `qu`); auch weniger Code (`sum`, `qu`, `foldr`, `map` überflüssig – siehe *dead code elimination*). Zudem ist dieses Verfahren automatisch!

7.6.3 Maschinenunabhängige lokale Optimierung

Diese Optimierungen geschenen auf Zwischensprachebene und optimieren lokale Codestücke (z.B. Basisblöcke).

- **Konstantenpropagation** (constant propagation): Falls der Wert einer Variablen an einer Stelle bekannt ist, dann ersetze die Variable durch den Wert der Variablen; kombiniere dies mit der **Konstantenfaltung** (constant folding): Werte konstante Ausdrücke zur Compilezeit aus!

  Beispiel: Betrachte folgendes Programm:

  ```
  x := 3
  y := 4
  z := x*y
  if i < z then jump l1
  ```

  Dann ergibt sich zunächst aus der Konstantenpropagation:

  ```
  x := 3
  y := 4
  z := 3*4
  if i < z then jump l1
  ```

  Durch Konstantenfaltung:

  ```
  x := 3
  y := 4
  z := 12
  if i < z then jump l1
  ```

  Wiederum durch Propagation:

  ```
  x := 3
  y := 4
  z := 12
  if i < 12 then jump l1
  ```

Eventuell ist es später durch eine Lebendigkeitsanalyse möglich, auf die Variablen x, y, z ganz zu verzichten und das ganze zu reduzieren auf

```
if i < 12 then jump l1
```

- **Kopierpropagation** (copy propagation): Variablen werden hier durch identische Variablen ersetzt, zum Beispiel:

```
x := y
... // hier keine Veränderung von x
z := a*x
```

  Ersetze dies dann durch `z := a*y` – es kann dann eventuell auf `x` völlig verzichtet werden!

- **Reduktion der Stärke von Operationen**: Ersetze komplexe Operatoren durch einfache (auch auf Quell- und Zielsprachebene)

ersetze	`x**2`	durch	`x*x`
ersetze	`x*2`	durch	`x+x`
ersetze	`x*2`	durch	`lshift x`

- **in-line-expansion**: Ersetze „einfache“ Funktionsaufrufe (z.B. ohne Kontrollstrukturen) durch Funktionsrumpf (kann auch auf Quellsprachebene geschehen): Setze zum Beispiel `sq x = x*x` in Codestück `...(sq y)...` ein und erhalte `...(y*y)...`. Aber auch bei einfache Kontrollstrukturen: Falls im Programm z.B. durch Konstantenpropagation in der folgenden Situation ein `True` auftaucht, so kann die in-line-expansion eine Vereinfachung bringen:

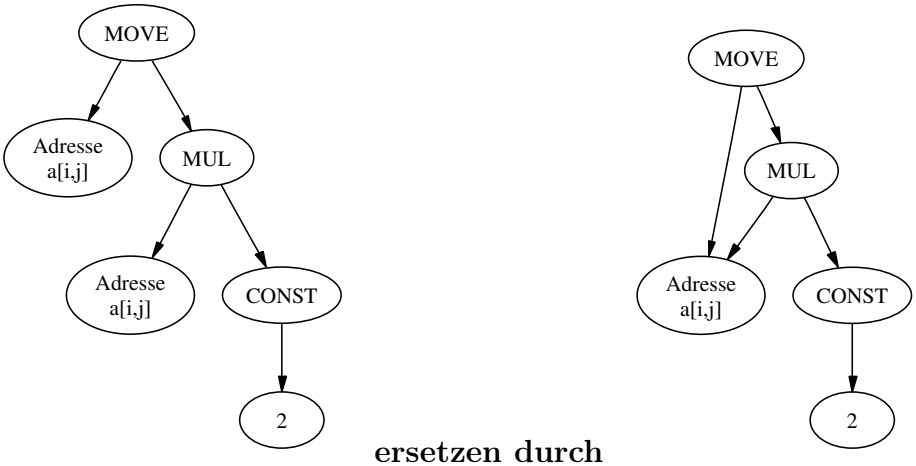
```
f b x y = if b then x else y

   ... (f True e1 e2) ...
=> ...         e1     ...
```

  Dies ist auch auf Zielsprachebene möglich: Wird beispielsweise `abs x` verwendet, so kann man auf der Zielsprachenebene zum Beispiel eine Instruktion „setze das Vorzeichen-Bit auf positiv“ ausführen.

- **Elimination redundanter Berechnungen**: mehrfache Teilberechnungen werden eliminiert!

  Beispiel: Im Code `a[i,j] := a[i,j]*2` kommt der Code zur Adreßberechnung von `a[i,j]` doppelt vor; eliminiere daher die zweite Berechnung (speichere das Ergebnis z.B. in einem Register zwischen).

  Als Technik setzt man statt der abstrakten Ausdrucksbäume dann DAGs (gerichtete azyklische Graphen) ein:



7.6.4 Schleifenoptimierungen

Die Optimierung von Schleifen ist eher global (im Kontrollflußgraphen) anwendbar, sie sind aber sehr lohnenswert, da heute ca. 90% der Laufzeit in Schleifen abläuft.

- **Verschiebung von Schleifeninvarianten**: Falls Operanden in einer Schleife nicht verändert werden und keine Seiteneffekte auftreten, so ziehe dies aus dem Schleifenrumpf heraus! *Beispiel:*

```
for i := 1 to 200 do
   for j := 1 to 100 do
      a[i,j] := sqrt(n) * (i + n) - j;
```

  Dies wird optimiert zu:

```
r := sqrt(n);
for i := 1 to 200 do
   s := r*(i + n);
   for j := 1 to 100 do
      a[i,j] := s - j;
```

Ersparnis ist hier 19.999 Berechnungen von `sqrt(n)` und 19.800 `r*(i+n)`-Berechnungen! Dies ist auch lohnenswert auf Zwischensprachenebene, z.B. bei der Adressberechnung bei Feldzugriffen!

- **Schleifenentfaltung**: Kleinere Schleifen haben durch Initialisierung, Inkrementierung und Prüfung der Abbruchbedingung einen großen Overhead. Entfalte daher kleinere Schleifen, *Beispiel:*

```
for i := 1 to 100 do
   for j := 1 to 2 do
      write(a[i,j]);
```

Dies wird ersetzt durch:

```
for i := 1 to 100 do
   write(a[i,1]);
   write(a[i,2]);
```

7.6.5 Globale Optimierungen

Hier wird mehr als ein Basisblock betrachtet, z.T. werden bekannte lokale Optimierungen ausgeweitet; zum Beispiel die Konstantenpropagation und -faltung.

- **dead code elimination:** Toter Code sind Anweisungen, die nie bei einer Programmausführung erreichbar sind; dies ist durch Konstantenpropagation erkennbar. *Beispiel:* Debugging-Code:

```
if DEBUG then
   writeln(...)
```

Falls die globale Konstante `DEBUG` auf `false` gesetzt ist, kann die obige Anweisung gelöscht werden. Lohnenswert ist dies vor allem auch

  - bei automatisch generiertem Code und
  - bei geladenen, aber nur teilweise genutzten Bibliotheken

7.6.6 Maschinenabhängige Optimierungen

Diese Optimierungen sind abhängig von der Zielmaschinenstruktur. Die Zielcodeauswahl und die Registerauswahl sind bereits diskutierte Beispiele; weitere Techniken:

- **Peephole Optimization**: Optimierung kleiner Befehlssequenzen am Ende der Zielcodeerzeugung („Guckloch"-Optimierung), die durch schematische Übersetzung Redundanzen aufweisen!

  Beispiel: Der Programmcode $x := y + 2;\ z := x \cdot 3$ ergibt den naiven[8] Zielcode

  $$\begin{array}{ll} & \texttt{LOAD}\ R \leftarrow M_y \\ & \texttt{ADDI}\ R \leftarrow R + 2 \\ (1) & \texttt{STORE}\ M_x \leftarrow R \\ (2) & \texttt{LOAD}\ R \leftarrow M_x \\ & \texttt{MUL}\ R \leftarrow R \cdot 3 \\ & \texttt{STORE}\ M_z \leftarrow R \end{array}$$

  Schiebe die „Schablone" über zwei benachbarte Instruktionen; im Beispiel läßt sich bei Betrachtung von (1) und (2) die Instruktion (2) sparen. Weiteres Beispiel:

  $$\begin{array}{ll} (3) & \texttt{JUMP}\ l_1 \\ & \ldots \\ & l_1 : \texttt{JUMP}\ l_2 \end{array}$$

  Ersetze im Beispiel die Instruktion (3) durch $\texttt{JUMP} l_2$.

7.6.7 Datenflußanalyse

Die Datenflußanalyse basiert auf Flußgraphen und liefert Informationen an Punkten im Flußgraph (z.B. Anfang/Ende von Basisblöcken). Diese gilt für **alle** möglichen Programmläufe und Voraussetzung für mächtige Optimierungen.

Klassifikation:

- **forward** oder **backward analysis** (Flußrichtung der Informationen)
- **all/any analysis** (Berücksichtigung aller/einiger Nachbarblöcke)

Bestandteile sind nun die Basisblöcke B als Knoten und Kanten, falls man im Kontrollfluß von einem Block zum nächsten verzweigen kann (definiere Mengen der Vorgänger- und Nachfolgerblöcke $\text{pred}(B)$ und $\text{succ}(B)$).

Folgende Informationen werden dann berechnet (der genaue Inhalt ist abhängig von der konkreten Analyse):

- $\text{in}(B)$, $\text{out}(B)$: gilt am Anfang bzw. Ende des Basisblocks B
- $\text{gen}(B)$, $\text{kill}(B)$: im Block B erzeugte bzw. gelöschte Informationen

[8] bei der hier vorgestellten Methode zur Registerallokation würde dies nicht auftauchen

Die Analyse geschieht dann in zwei Schritten:

- Aufstellen von Gleichungen über $\text{in}(B_i)$, $\text{out}(B_j)$, $\text{gen}(B_k)$, $\text{kill}(B_l)$
- Lösen der Gleichungen durch **Fixpunktiteration**

Beispiele:

- Bei der **UD-Verkettung** (use/definition chaining) wird zu jeder Variablenverwendung (als Operand, „use“) die Menge aller Definitionen, die dort gültig sind. Als Anwendung dieser Analyse ist die Konstantenpropagation zu sehen:

  $$x := 5;\ \ldots;\ \texttt{if}\ x < 10\ \texttt{then}\ldots$$

  Ist die Zuweisung $x := 5$ innerhalb der `if`-Abfrage gültig? Die Information ist hier die Menge der gültigen Definitionen (Zuweisungen), die Gleichungen ergeben sich folgendermaßen:

  - Für den Startblock B_1 gilt: $\text{in}(B_1) = \emptyset$ (keine gültige Definition)
  - Zu Beginn eines Block sind alle Definitionen irgendeines Vorgängers gültig:
    $$\text{in}(B_i) = \bigcup_{B_j \in \text{pred}(B_j)} \text{out}(B_j)$$
  - Am Ende eines Basisblocks sind alle Definitionen gültig, die in B_i generiert wurden und alle zu Beginn von B_i gültigen, die nicht in B_i neu definiert werden:
    $$\text{out}(B_i) = \text{gen}(B_i) \cup (\text{in}(B_i) \setminus \text{kill}(B_i))$$

  Diese Analyse ist also eine *forward-any-analysis*, da der Informationsfluß vom Beginn zum Ende des Programms erfolgt und die Gültigkeit in einem Block gilt, falls in irgendeinem Vorgängerblock gültig sind.

- **Lebendigkeitsanalyse**: Sei B_n der letzte Block.

  $$\begin{aligned} \text{out}(B_n) &= \emptyset \\ \text{out}(B_i) &= \bigcup_{b_j \in \text{succ}(B_i)} \text{in}(B_j) \\ \text{in}(B_i) &= \underbrace{\text{gen}(B_i)}_{\text{erstes Vorkommen als Operand in } B_i} \cup(\text{in}(B_i) \setminus \text{kill}(B_i)) \end{aligned}$$

  Somit ist die Lebendigkeitsanalyse eine *backward-any-analysis*.

7.6.8 Abstrakte Interpretation

Die abstrakte Interpretation (Cousot/Cousot 1977) orientiert sich an der operationalen Semantik (Interpreter); man rechnet mit abstrakten (statt konkreten) Werten – notwendig ist also eine abstrakte operationale Semantik. Beachte, daß die abstrakte Berechnung terminieren muß – dies ist entweder dadurch garantiert, daß man nur endlich viele abstrakte Werte zuläßt (und damit nur endlich viele Interpreterkonfigurationen); oder durch „Überspringen" von eventuell endlosen Berechnungsketten *(widening)*.

Eingesetze Techniken:

- Verbände als abstrakte Wertebereiche
- Fixpunktiteration
- abstrakte Werte approximieren konkrete Werte
- abstrakte Operationen approximieren konkrete Operationen

Hiermit erhalten wir eine semantisch konkrete Programmanalyse!

Beispiel: **Vorzeichenanalyse**: Welche Vorzeichen haben Variablen? Abstrakte Werte sind hier p (für Werte > 0), n (< 0) und z ($= 0$); abstrakte Operationen sind hier markiert mit $\bar{}$. Es gilt dann:

$$\begin{aligned} p\bar{+}p &= p \\ n\bar{+}n &= n \\ p\bar{+}z &= p \\ n\bar{+}z &= n \end{aligned}$$

Doch die Gleichung $p\bar{+}n$ bereitet Probleme – führe ein abstrakten Wert ? ein, der für beliebige Werte steht und definiere nun die Operation vollständig durch folgende Tabelle:

$\bar{+}$	p	z	n	?
p	p	p	?	?
z	p	z	n	?
n	?	n	n	?
?	?	?	?	?

$\bar{\cdot}$	p	z	n	?
p	p	z	n	?
z	z	z	z	z
n	n	z	p	?
?	?	z	?	?

Ersetze dann noch Konstanten durch abstrakte Werte und rechne dann mit abstrakten Werten bzw. Operationen, bis ein Fixpunkt erreicht ist; dies ist die abstrakte Interpretation!

Für jede Analyse ist zu zeigen, daß die abstrakten Operationen die konkreten Operationen approximieren; dann sind die Ergebnisse der abstrakten Interpretation automatisch korrekt.

Ausblick

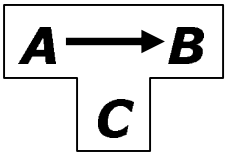
Nicht betrachtet wurde hier die Übersetzung spezieller Sprachen: funktionale, logische, objektorientierte Sprachen; die spezielle Implementierungstechniken benötigen für

- bedarfsgesteuerte Auswertung, Funktionen höherer Ordnung (z.B. Haskell)
- Backtracking, logische Variablen und Unifikation (z.B. Prolog)
- Methodensuche (objektorientierte Sprachen)
- Speicherverwaltung, garbage collection

A Symbole

- Interpreter für die Sprache A in der Sprache B geschrieben:

- In der Sprache C geschriebener Compiler, der A-Programme in die Sprache B übersetzt:



B Einführung in Haskell

Vorteile:

- hohes Programmierniveau, keine Manipulation von Speicherzellen
- keine Seiteneffekte (hilfreich bei Codeoptimierung, Verständlichkeit)
- Programmierung über Eigenschaften, nicht über zeitlichen Ablauf
- implizite Speicherverwaltung
- einfache Korrektheitsbeweise/Verifikation
- kompakte Sourcecodegröße, kürzere Entwicklungszeiten, lesbarere Programme
- modularer Programmaufbau, Polymorphie, Funktionen höherer Ordnung → Wiederverwendbarkeit

Nachteil ist die Geschwindigkeit. Strukturen eines Funktionalen Programmes sind:

- Variablen ≡ unbekannte Werte
- Programm ≡ Menge von Funktionsdefinitionen
- Speicher nicht explizit verwendbar, wird automatisch verwaltet
- Programmablauf ≡ Reduktion von Ausdrücken (mathematische Theorie des λ-Kalküls[9])

B.1 Funktionsdefinitionen

Funktionen werden definiert durch ein Anweisung wie:

$$\texttt{f x}_1 \texttt{ ... x}_n \texttt{ = e}$$

Im Rumpf können vorkommen:

- Zahlen: `3`, `3.1415`
- Basisoperationen: `3+4`, `5*7`
- Funktionsanwendungen: `(f` $\texttt{e}_1$ `...` $\texttt{e}_n$`)`

[9] Church (1941)

- bedingte Ausdrücke: `if b then` e_1 `else` e_2

Beispiele:

- Eine Quadratfunktion definiert man durch `square x = x * x`. Ablauf in Hugs:

```
>:l square
>square 3
9
>square (2+5)
49
```

- Fakultät: mehrere Möglichkeiten

  1. direkt:

     ```
     fac n = if n == 0 then 1 else n * fac (n-1)
     ```

  2. durch bedingte Gleichungen:

     ```
     fac n | n == 0 = 1
           | otherwise = n * fac (n-1)
     ```

  3. mit Pattern-Matching:

     ```
     fac 0 = 1
     fac (n+1) = (n+1) * fac n
     ```

- Fibonacci-Funktion (mit Pattern-Matching):

  ```
  fib1 0 = 0
  fib1 1 = 1
  fib1 (n+2) = fib1 n + fib1 (n+1)
  ```

  Problem ist hier die exponentielle Laufzeit in $\mathcal{O}(2^n)$. Besser: **Akkumulatortechnik**, d.h. Berechnung „von unten“, bis Wert erreicht ist. Hilfsfunktion:

  ```
  fib2 n = fib2' 0 1 n
  fib2' x y 0 = x
  fib2' x y (n+1) = fib2' y (x+y) n
  ```

  Bessere Strukturierung mittels lokaler Definitionen durch `where`:

```
fib2 n = fib2' 0 1 n
         where fib2' x y 0 = x
               fib2' x y (n+1) = fib2' y (x+y) n
```

Alternative ist die `let`-Anweisung, beispielsweise kann $f(x, y) = y(1 - y) + (1 + xy)(1 - y) + xy$ umgesetzt werden als

```
f x y = let a = 1 - y
            b = x * y
        in y * a + (1 + b) * a + b
```

B.2 Datenstrukturen

Haskell hat ein **strenges** Typenkonzept, alle Werte sind getypt (Zahlen, boolesche Werte). Auch **jeder zulässige Ausdruck** hat einen Typ, der den Wert beschreibt, der eventuell berechnet wird. Schreibweise dafür (hinter beliebigem Ausdruck möglich!): `ausdruck::typ`. Basistypen sind:

Typ	Konstanten	Funktionen
`Bool`	`True`, `False`	`&&`, `||`, `not`
`Int`	1, 2, ...	`+`, `-`, `*`, `div`, `mod`
`Float`	`0.3`, `1.5e-2`	`+`, `-`, `*`, `/`
`Char`	`'a'`, ..., `'z'`, ...	

Listen (Folgen) von Elementen vom Typ `t` haben den Typ `[t]`. Beispielsweise ist der Typ `String` identisch mit `[Char]`. Die leere Liste ist `[]`; für `x::t` und `l::[t]` ist `x:l::[t]`.

Die Liste mit den Elementen 1, 2, 3 läßt sich schreiben als `1:(2:(3:[]))`, einfacher geht's mit `1:2:3:[]`, durch `[1,2,3]` oder bei Listen von Zahlen: `[a..b]` sowie `[a..]`. Viele Operationen auf Listen sind vordefiniert in `Prelude` und `List`:

```
length [] = 0
length (x:xs) = 1 + length xs
```

Verkettung von Listen geschieht durch `++`:

```
[]++ys = ys
(x:xs)++ys = x:(xs++ys)
```

Beachte die **Infixnotation**, durch `(++)` ergibt sich aber eine Funktion für Präfixnotation aus dem Infix-Operator, d.h.

$$\texttt{[1,2]++[3,4]} \equiv \texttt{(++)[1,2][3,4]} \equiv \texttt{[1,2,3,4]}$$

Tupel sind beispielsweise `(1, 'a', True)::(Int, Char, Bool)`. Es gibt auch **funktionale Typen**: $\texttt{t}_1\texttt{->t}_2$ ist der Typ eine Funktion mit Argumenten vom Typ $\texttt{t}_1$ und Ergebnis vom Typ $\texttt{t}_2$, Beispiel:

```
square::Int->Int
square x = x*x
```

Konstanten des funktionalen Typs sind **λ-Abstraktionen**: anonyme Funktionen wie `\x -> 1`, beispielsweise die Inkrementfunktion

```
\x->x+1::Int->Int
```

In Haskell können Funktionen sowohl über Tupel als auch curryfiziert definiert werden[10], Beispiel:

```
add::(Int,Int)->Int
add(a,b) = a+b
add::Int->(Int->Int)
add a b = a+b
```

Es gilt mathematisch: $[A \times B \to C] \simeq [A \to (B \to C)]$, daher

$$\texttt{add 3 4} \equiv \texttt{((add 3) 4)} \implies \texttt{(add 1)::Int->Int}$$

B.3 Selbstdefinierte bzw. algebraische Datenstrukturen

Konstrukturen zum Aufbau algebraischer Datentypen sind frei interpretierte Funktionen (nicht reduzierbar, z.B. `[]` und `(:)`). Datendefinition:

$$\texttt{data t = c}_1\ \texttt{t}_{11}\ \texttt{...}\ \texttt{t}_{1n_1}\ \texttt{|}\ \texttt{...}\ \texttt{|}\ \texttt{c}_k\ \texttt{t}_{k1}\ \texttt{...}\ \texttt{t}_{kn_k}$$

Dies führt einen neuen Typ t ein mit k Konstruktoren mit $\texttt{c}_j\texttt{::t}_{j1}\texttt{->...->t}_{jn_j} \to \texttt{t}$. Sowohl Typen als auch Konstruktoren müssen mit Großbuchstaben beginnen. Beispiel:

```
data ListInt = Nil | Cons Int ListInt
```

Damit ist `Nil::ListInt` und `Cons::Int->ListInt->ListInt`. Listen allgemein: `[t]` sind polymorph bezüglich der Elemente:

```
data [t] = [] | t:[t]
```

[10] nach Schönfinkel und Curry

Dabei ist `t` eine Typ-Variable. Vorteil dieser Polymorphie: Der Code wird wiederverwendbar, wir benötigen keine einzelnen `++Char`, `++Int`, `++[Int]` ...Beispielsweise funktionieren auf allen Listen:

```
length::[a]->Int
(++)::[a]->[a]->[a]
```

Beachte den Unterschied:

```
last::[a]->a
last [x] = x
last (_:x:xs) = last (x:xs)
```

Beispiel: Binäre Bäume mit Elementen beliebigen Typs:

```
data Tree a = Empty | Node (Tree a) a (Tree a)

Node::Tree a->a->Tree a->Tree a

treeToList::Tree a->[a]
treeToList Empty = []
treeToList (Node tl e tr) = treeToList tl ++ (e:treeToList tr)
```

Dann ist `treeToList (Node Empty 7 Empty)` gleich `[7]::Tree Int`. **Spezialfälle** sind:

- Aufzählungstypen:

  ```
  data Color = Red | Green | Blue
  ```

- Verbundstypen:

  ```
  data Complex = Complex Float Float
  ```

  Dabei hat `Complex` sowohl die Bedeutung als Typ als auch als Konstruktor. Die Addition:

  ```
  addc::Complex->Complex->Complex
  addc (Complex r1 i1) (Complex r2 i2)
         = Complex (r1 + r2) (i1 + i2)
  ```

Weiterer Polymorphismus: Bei Rechenoperationen sollte `(+)` nicht nur auf `Int` definiert sein, sondern auch auf `Float`, `Complex`, `Integer`, ...Hierzu gibt es *Typklassen*, in denen Typen zusammengefaßt werden, auf denen z.B. gerechnet werden kann: `Num`

```
(+)::Num a => a->a->a
```

Spezialfälle sind `Int->Int->Int` und `Float->Float->Float`. Außerdem wichtige Klassen:

- In `Eq` sind Vergleich mit `(==)` und `(/=)` möglich.
- In `Ord` sind Vergleiche mit `(==)`, `(<=)`, `(>)` etc. möglich.
- In `Show` sind Anzeigemöglichkeiten definiert, um beispielsweise Ausdrücke auf der Konsole ausgeben zu lassen.

Fügt man hinter einem Datentyp `deriving(Show, Eq)` ein, so läßt kann man Datentypen einer dieser Typklassen zuordnen und für die Funktionen von `Show` und `Eq` automatisch Code erzeugen zu lassen.

B.4 Pattern Matching

Komfortabler Programmierstil, linke Seiten entsprechen „Prototypen“ des Funktionsaufruf:

- `x` (Variable) matcht immer, bindet x an aktuellen Ausdruck
- `(p1:p2)`, `(Node p1 p2 p3)` matcht, falls der Konstruktor auf den obersten Konstruktor des Aufrufs und Argumente passen
- `(p1, ..., pm)` matcht auf Tupel
- `(n+k)` mit einer Zahl k matcht auf Argumente $\geq k$ und bindet n an das Argument $-k$
- `x@p` matcht, falls `p` paßt und x wird an entpsrechenden Teilterm gebunden, z.B. matcht `x:xs@(_:_)` auf Listen mit mindestens zwei Elementen
- `_` paßt auf alles, es wird aber nichts gebunden
- Zahlen werden wie nullstellige Konstruktoren behandelt

Beispiel:

```
fac 0 = 1
fac nP1@(n+1) = nP1*fac n
```

Nicht erlaubt sind Pattern mit mehrfach auftretenden Variablen, also z.B. `eq x x = True`, da nicht unbedingt ein Vergleich existiert. (Prinzipielles) Problem beim Pattern Matching ist außerdem, daß die Ausführungsreihenfolge bei überlappenden Pattern Einfluß auf das Ergebnis hat, dies sollte man also vermeiden. In Hugs ist die Auswertung von links nach rechts und von oben nach unten festgelegt, Beispiel für eine schlechte Definition:

```
fac 0 = 1
fac n = n*fac(n-1)
```

Besser wäre es, in der letzten Zeile das `(n+k)`-Pattern zu verwenden.

B.5 Funktionen höherer Ordnung

Funktionen sind in funktionalen Sprachen oft „Bürger erster Klasse“, d.h. sie dürfen als Parameter oder Rückgabewert verwendet werden und auch in Datenstrukturen eingebunden werden. Dies wird in der generischen Programmierung verwendet und in Programmschemata, man erhält dadurch eine höhere Wiederverwendbarkeit und Modularität.

Beispiel: Die Ableitung ist eine Funktion, welche aus einer Funktion eine neue Funktion ableitet. Die numerische Berechnung $f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$ läßt sich umsetzen zu

```
ableitung::(Float->Float)->(Float->Float)
ableitung f = f'
  where f' x = (f(x+dx)-f x)/dx
        dx   = 0.0001

> ableitung sin 0.0
1.0
> ableitung (\x->x*x) 1.0
2.00023
```

Beispiel:

- Inkrementiere alle Elemente einer Liste:

  ```
  inclist :: Num a => [a]->[a]
  inclist [] = []
  inclist (x:xs) = x+1:inclist xs
  ```

- Chiffrieren von Strings durch zyklisches Verändern von Zeichen

```
code :: Char->Char
code c | c == 'Z' = 'A'
       | c == 'z' = 'a'
       | otherwise chr(ord(c+1))
codeString :: String->String
codeString "" = ""
codeString (c:cs) = (code c):codeString cs
```

! Beachte: inclist und codeString sind ähnlich in der Struktur. Unterschied ist nur `code` statt `inc`, praktisch: das als Parameter übergeben

```
map :: (a->b)->[a]->[b]
map _ [] = []
map f (x:xs) = f x:map f xs

inclist xs = map (+1) xs
codeString s = map code s
```

Noch einfacher: mit partieller Applikation:

```
inclist = map (+1)
codeString = map code
```

Vorteil der curryfizierten Darstellung `f:: t1->(t2->(t3->(...)))`: Der Ausdruck `f x` ist partiell applizierbar, falls `x::t1`. Sonderfall bei Infixoperatoren (+, -, *, /, ...): Folgende vier Ausdrücke sind äquivalent:

```
a/b
(a/) b
(/) a b
(/b) a
```

Andere Listenoperationen: Auffalten

- Summe einer Liste

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + (sum xs)
```

- Eingabekontrolle durch Summe der ASCII-Werte

```
checkSum :: String -> Int
checkSum [] = -1003
checkSum (c:cs) = (ord c) + (checksum cs)
> checksum "Informatik"
42
```

! Beachte: beide sind wieder gleich von der Struktur, Zusammenfassung:

```
foldr :: (a->b->b)->b->[a]->b
fordr f e [] = e
foldr f e (x:xs) = f x (foldr f e xs)

sum = foldr (+) 0
checkSum = foldr (\c -> ((ord c) +))
         = foldr (\c s -> (ord c) + s)
         = foldr ((+).ord)
```

Allgemeines Vorgehen: Suche allgemeines Schema und realisiere es durch eine Funktion höherer Ordnung, z.B. Filter:

```
filter :: (a->Bool)->[a]->[a]
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise filter p xs

filter p = foldr (\x ys -> if p x then x:ys else ys) []
         = foldr (\x if p x then (x:) else id) []
```

Umwandlung von einer Liste in eine Menge (doppelte Entfernen)

```
nub::Eq a=>[a]->[a]
nub [] = []
nub (x:xs) = x:nub(filter (/=) xs)
```

Anwendung: Sortieren Listen mittels Quicksort

```
qsort [] = []
qsort (x:xs) =    qsort (filter (<=x) xs)
               ++ [x]
               ++ qsort (filter (>x)  xs)
```

Besser noch mit `split::(a->Bool)->[a]->([a],[a])`:

```
qsort (x:xs) = qsort le++[x]++ge
      where (le,ge) = split (<=x) xs
```

Umsetzung von Kontrollstrukturen *Beispiel:* `while`-Schleife

```
x = 1;
while (x < 100) do
   x = 2*x;
```

Umsetzung in Haskell:

```
while::(a->Bool)->(a->a)->a->a
while p f x | p x = while p f (f x)
            | otherwise = x

> while (<100) (2*) 1
128
```

B.6 Funktionen als Datenstrukturen

Datenstrukturen: Objekte mit Operationen zur

- Konstruktion (z.B. `[]`, `(:)` bei Listen)
- Selektion (z.B. `head` und `tail` bei Listen)
- Verknüfungen (z.B. `++` bei Listen)

Wichtig ist die Funktionalität (Schnittstelle) und nicht die Implementierung – entspricht abstrakten Datentypen, d.h. Datenstrukturen entsprechen einem Satz von Funktionen. *Beispiel:* Arrays (Felder mit Elementen vom Typ `a`):

- Konstruktion:

  ```
  emptyArray :: Array a
  putIndex :: Int->a->Array a->Array a
  ```

- Selektion

  ```
  getIndex :: Int->Array a->a
  ```

Implementierung durch Funktion höherer Ordnung: Array als Abbildung von Indizes auf Werte

```
type Array a = Int->a
emptyArray i = error "Feld nicht initialisiert"
getIndex i a = a i
putIndex i v a = a'
    where a' j | i == j v
               | otherwise = a j
>getIndex 2 (putindex 2 'b' emptyArray)
'b'
```

Vorteil ist die konzeptuelle Klarheit (entspricht der Spezifikation), Nachteil ist die Zugriffszeit.

B.7 Lazy Evaluation

Betrachte folgendes Haskell-Programm:

```
g x = 1
h = h
> g h
1
```

Berechnungen: `g h` kann ausgewertet werden zu 1 oder zu `g h` usw. Es gibt zwei ausgezeichnete Reduktionen:

- *left-most-innermost (call-by-value)* entspricht *strikten* Sprachen: ML, Erlang, imperative Sprachen
- *left-most-outermost (call-by-name)* entspricht *nicht-strikten* Sprachen: Haskell, ...

Eigenschaften von *left-most-outermost*:

- Alles, was durch irgendeine Reduktion berechnet werden kann, wird berechnet.
- Vermeidung überflüssiger Berechnungen (z.B. $h \to h \to h \to h \to \ldots$)
- Rechnen mit unendlichen Datenstrukturen möglich
- zur Effizienz: betrachte $\texttt{double}(x) = x + x$

$$\begin{aligned} \texttt{double}(3+4) &= (3+4)+(3+4) = 7+(3+4) = 7+7 = 14 \\ \texttt{double}(3+4) &= \texttt{double}(7) = 7+7 = 14 \end{aligned}$$

Optimierung durch *Sharing*:

$$\texttt{double}(3+4) = \underbrace{\cdot + \cdot}_{(3+4)} + \underbrace{\cdot + \cdot}_{(3+4)} = 14$$

Beispiele fürs Rechnen mit unendlichen Datenstrukturen:

```
from :: a -> [a]
from n = n::from (n+1)

take :: Int->[a]->[a]
take 0 _ = []
take (n+1) (x:xs) = x:take n xs

> take 3 (from 1)
[1,2,3]
```

Vorteil ist hier die Trennung von Kontrolle (`take 3`) und Daten (`from 1`). Primzahlberechnung mit dem Sieb des Eratostenes:

```
sieve :: [Int]->[Int]
sieve (x:xs) = x:sieve (filter (\y -> y `mod´ x > 0) xs)
primes = sieve (from 2)
```

Ersten zehn Primzahlen: `take 10 primes`

B.8 Monaden

Problem: Ein-/Ausgabe als Seiteneffekt! In ML liefert `output("hu"); output("hu")` das Wort `huhu` als Seiteneffekt und `()` als Ergebnis (`Unit` in ML oder `()` in Haskell).

Aber:

```
let val x = output("hu") in
  x; x
end
```

Dies liefert Ergebnis `hu` und Wert `()`! In Haskell wegen Lazy-Auswertung noch unverständlicher, wann die Ausgabe erfolgt! **Lösung:** *Monaden* (IO-Monade): Ein-/Ausgabeoperationen protokollieren und auf „Top-Level“ ausführen. Das Protokollieren geschieht in der IO-Monade.

```
putChar :: Char->IO()
```

Dann liefert `putChar 'a'` die Aktion „gib ein `a` aus!“. Aktionen, die bei `main` ankommen, werden ausgeführt. Damit gibt das folgende Programm `a` auf dem Bildschirm aus:

```
main::IO ()
main = putChar 'a'
```

Beachte: `putChar` bewirkt keine Ausgabe! Hintereinanderausführung mehrerer Ausgaben:

```
(>>):IO()->IO()->IO()
main = let p = putChar 'a' in
  p>>p
```

Dann liefert `main` die Ausgabe `aa`, genau wie `putChar 'a' >> putChar 'a'`. Erweiterung:

```
putStr :: String -> IO()

putStr "" = return ()
putStr (c:cs) = putChar c >> putStr cs
```

Alternativ:

```
putStr = foldr (\c -> putChar c >>) (return ())
putStr = foldr ((>>) . putChar) (return ())
```

Weiteres Beispiel:

```
head[putStr "Hallo",putStr"Ihr"]
```

In der IO-Monade gelten folgende Gesetze:

```
done >> m = m
m >> done = m
m >> (n >> o) = (m >> n) >> o
```

Damit ist `done` ein neutrales Element und `(>>)` assoziativ. Damit formen beide ein *Monoid*. Analog für return und `(>>=)`:

```
return v >>= \x -> m = m[x/v]
m >>= \x -> return x = m
m >>= \x -> (n >>= \y -> o)
   = (m >>= \x -> n) >>= \y -> o
  (falls x nicht frei in o)
```

Diese Struktur heißt nach LEIBNITZ *Monade*.

In Haskell: Eine Monade ist ein Typkonstruktor m zusammen mit zwei Operationen `return` und `(>>=)`, welche obige Gesetze erfüllen.

```
class Monade m where
   (>>=) :: m a -> (a -> m b) -> mb
   return :: a -> m a
   (>>) :: m a -> m a -> ma
   p >> q = p >>= \_ -> q
```

Beispiel: die Maybe-Monade:

```
data Maybe a = Just a | Nothing
```

Auswertung arithmetischer Ausdrücke:

```
data Expr = Expr :+: Expr
          | Expr :/: Expr
          | Num Float
```

Problem: Vermeidung von Laufzeitfehlern (hier Division durch null), Ziel:

```
> eval (Num 3 :+: Num 4)
Just 7.0
> eval (Num 3 :+: (Num 2 :/: Num 0))
Nothing
```

nun:

```
eval :: Expr -> Maybe Float
eval (Num n) = Just n
eval (e1 :/: e2) = case eval e1 of
     Nothing -> Nothing
     Just m1 -> case eval e2 of
                    Nothing -> Nothing
                    Just 0 -> Nothing
                    Just n2 -> Just n1/n2
```

Schöner wird's mit 'ner Monade, Idee: Nothing schlägt durch!

```
instance Monad Maybe where
  Nothing >>= k = Nothing
  Just x >>= k  = k x
  return        = Just
```

Dann benutzen:

```
   eval (e1 :/: e2) = do
       n1 <- eval e1
       n2 <- eval e2
       if n2 == 0 then Nothing
                  else return (n1/n2)
   eval (Num n) = return n
```

Ähnliche Instanz:

```
data Either a b = Left a | Right b
```

Listen formen ebenfalls Monaden: Betrachte Listen hierzu als Container, welcher Werte aufnimmt:

```
instance Monad [] where
   return x = [x]
   (x:xs) >>= f = (f x) ++ (xs >>= f)
   [] >>= = []
```

Benutzung:

```
> [1,2,3] >>= \x -> [4,5] >>= \y -> return (x, y)
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Mit der `do`-Notation:

```
do x <- [1,2,3]
   y <- [4,5]
   return (x, y)

[(x, y) | x <- [1,2,3], y <- [4, 5]]
```

Analysiert man die `[]`-Monade genauer, so erkennt man zusätzliche Eigenschaften. Die Leere Liste ist eine Null der Monadenstruktur:

```
m >>= \x -> [] = []
[] >>= m = []
```

Deshalb:

```
class (Monad m) = MonadZero m where
   zero :: m a
```

Für Listen:

```
instance MonadZero [] where
   zero = []
```

Außerdem gibt es die ausgezeichnete Funktion `(++)`, welche zwei Listen konkateniert

```
class (MonadZero m) => MonadPlus m where
  (++) :: m a -> m a -> m a
```

Ab Haskell 98 sind `MonadPlus` und `MonadZero` zusammen.

Vorteil: Es ergeben sich viele Funktionen, die allgemein für `MonadPlus` definiert sind:

```
mapM :: Monad m => (a -> b) -> m a -> m b
mapM f l = l >>= return f

map :: (a -> b) -> [a] -> [b]
map = mapM

guard :: MonadPlus m => Bool -> m ()
guard b = if b then return () else zero

do x <- [1..10]
   guard (x < 5)
   return x
```

List Comprehension:

$$\{(i,j) \mid i \in \{1,\ldots,3\}, j \in \{2,\ldots,4\}, i \neq j\}$$

wird umgesetzt in

```
[(i,j) | i <- [1..3], j <- [2..4], i != j]
```

das ergibt in Haskell `[(1,2),(1,3),(1,4),(2,3),(2,4),(3,2),(3,4)]`.