Notizen zur Vorlesung

Übersetzerbau

Wintersemester 2014/15

Prof. Dr. Michael Hanus

Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion
Christian-Albrechts-Universität zu Kiel

Version vom 5. Februar 2015

Vorwort

Die Übersetzung von Programmiersprachen ist eine wohldefinierte aber dennoch komplexe Aufgabe. Zur Beherrschung dieser Komplexität wurde eine Zerlegung in einzelne Übersetzungsaufgaben entwickelt, die heute in dieser oder ähnlicher Form in den meisten Übersetzern verwendet wird. In dieser Vorlesung werden die einzelnen Übersetzungsaufgaben und die bekannten Lösungsansätze dazu vorgestellt. Im Einzelnen werden behandelt:

- Programmiersprachen, Interpreter, Übersetzer
- Lexikalische Analyse
- Syntaktische Analyse
- Semantische Analyse
- Codeerzeugung

Dieses Skript ist eine überarbeitete Fassung einer Mitschrift, die ursprünglich von Klaas Ole Kürtz in LaTeX gesetzt wurde. Ich danke Herrn Kürtz für die LaTeX-Vorlage und seine Zustimmung, diese weiterzuverwenden, und Eike Schulz für eine weitere Mitschrift. Außerdem danke ich Stefan Junge, Fabian Reck und besonders Björn Peemöller für Hinweise und Korrekturen zu diesem Skript.
Noch ein wichtiger Hinweis: Dieses Skript soll nur einen Überblick über das geben, was in der Vorlesung gemacht wird. Es ersetzt nicht die Teilnahme an der Vorlesung, die zum Verständnis der Konzepte und Techniken des Übersetzerbaus wichtig ist. Ebenso wird für ein vertieftes Selbststudium empfohlen, sich die in der Vorlesung angegebenen Lehrbücher anzuschauen.

Kiel, Februar 2015 Michael Hanus

P.S.: Wer in diesem Skript keine Fehler findet, hat sehr unaufmerksam gelesen. Ich bin für alle Hinweise auf Fehler dankbar, die mir persönlich, schriftlich oder per E-Mail mitgeteilt werden.

Inhaltsverzeichnis

1. Einführung

Es existieren zahlreiche Lehrbücher zu dem Thema der Vorlesung, allerdings keines, was exakt die Vorlesungsinhalte abdeckt. Aus diesem Grund sind hier einige empfehlenswerte Bücher aufgeführt:

- A. Aho, R. Sethi, J.Ullman: *Compilerbau (Teil 1 + 2)*, Oldenbourg, 1999
- A. Aho, M. Lam, R. Sethi, J.Ullman: *Compilers: principles, techniques, and tools (2nd ed.)*, Pearson Education, 2007
- A. Appel: *Modern compiler implementation in ML*, Cambridge UP, 1997
- R.H. Güting, M. Erwig: *Übersetzerbau*, Springer 1999
- W. Waite, G. Goos: *Compiler Construction*, Springer, 1984
- R. Wilhelm, D. Maurer: *Übersetzerbau: Theorie, Konstruktion, Generierung*, Springer, 1992
- N. Wirth: *Grundlagen und Techniken des Compilerbaus*, Oldenbourg, 2008

Das Buch von Andrew Appel behandelt dabei am besten die Vorlesungsinhalte.

1.1. Was ist ein Übersetzer?

Ein **Übersetzer/Compiler** ist selbst ein Programm, das jedem Programmiersprachenprogramm (Quellprogramm) ein semantisch äquivalentes Maschinensprachenprogramm (Zielprogramm) zuordnet. Hierbei ist die zu übersetzende Programmiersprache PS meist eine **höhere Programmiersprache**, wie z.B.:

- **imperative** Programmiersprachen (dies bildet den wesentlichen Inhalt dieser Vorlesung) mit Konstrukten wie Wertzuweisung, Kontroll- und Datenstrukturen, Module, Klassen
- **deklarative** Programmiersprachen: funktionale, logische Sprachen
- **nebenläufige** Programmiersprachen: parallele, verteilte Prozesse

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

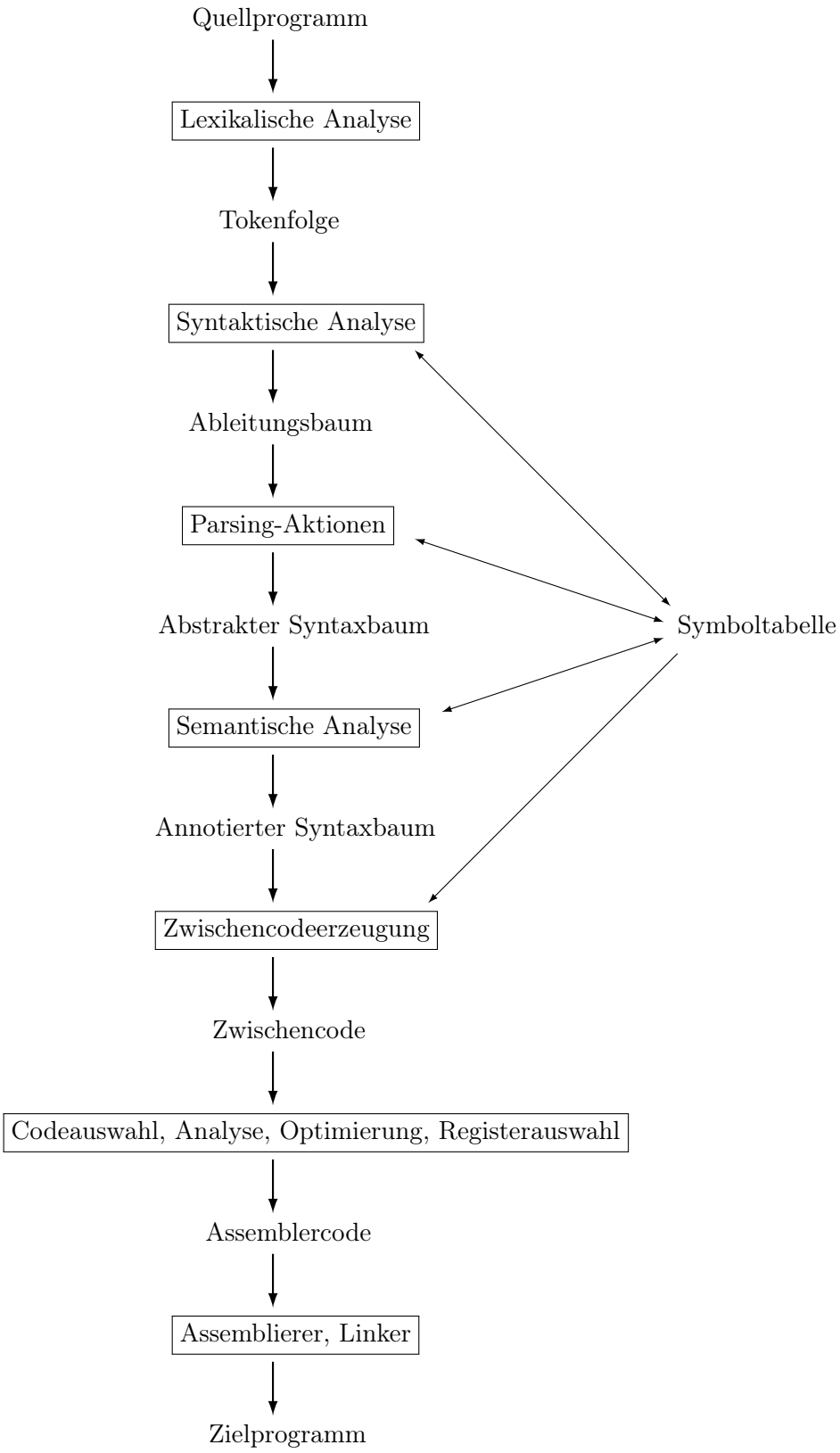
- **CISC**-Rechner (complex instruction set computers) stellen relativ viele Befehle zur Verfügung, wobei die einzelnen Befehle komplex sein können

- **RISC**-Architekturen (reduced instruction set computers) haben einen kleineren, einfacheren Befehlssatz, dafür werden die Befehle aber in der Regel 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 ein 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 Spezifikationsmethoden.

1.2. Struktur von Übersetzern

Die typische Struktur eines Übersetzers sieht wie folgt aus:


Quellprogramm
Lexikalische Analyse
Tokenfolge
Syntaktische Analyse
Ableitungsbaum
Parsing-Aktionen
Abstrakter Syntaxbaum
Symboltabelle
Semantische Analyse
Annotierter Syntaxbaum
Zwischencodeerzeugung
Zwischencode
Codeauswahl, Analyse, Optimierung, Registerauswahl
Assemblercode
Assemblierer, Linker
Zielprogramm

Bei der syntaktischen Analyse, den Parsing-Aktionen sowie der semantischen Analyse wird die Symboltabelle benutzt *und verändert* (angedeutet durch Doppelpfeile), bei der Zwischencodeerzeugung wird sie nur benutzt.

Die einzelnen **Phasen des Übersetzungsvorgangs** beinhalten folgende Aufgaben:

- **lexikalische Analyse**: Zweck ist die Erkennung von zusammengehörigen Symbolen, Kommentaren etc. Spezifikationsmethoden hierfür sind reguläre Ausdrücke, die mit Hilfe endlicher Automaten implementiert werden können.
- **syntaktische Analyse**: Dient der Erkennung der Programmstruktur, Spezifikationsmethoden sind kontextfreie Grammatiken, die durch Kellerautomaten implementiert werden können.
- **Parsing-Aktionen**: Zur Vereinfachung der Programmstruktur (Entfernen von semantisch irrelevanten Inhalten)
- **semantische Analyse**: Erkennen von Kontextabhängigkeiten (Zuordnung zwischen Definition und Verwendung, Typanalyse, etc.)

  *Ab hier wird ein Programm als **korrekt** akzeptiert.*
- **Zwischencodeerzeugung**: Erzeugung von vereinfachtem, maschinenunabhängigen Code (dies ist wichtig für die Portabilität)

  *Bis hier ist das Übersetzungsverfahren **maschinenunabhängig**.*
- **Codeerzeugung**: Erzeugung des 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 wurden One-Pass-Compiler bevorzugt, was durchaus Einfluss auf die Syntax der Sprache hatte (z.B. bei 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

Die Implementierungssprache in dieser Vorlesung ist die funktionale Sprache Haskell, denn funktionale Sprachen sind besonders gut für Übersetzungsaufgaben geeignet. Wir verwenden dabei den Haskell2010-Standard, als Haskell-Implementierung kann z.B. der Glasgow Haskell Compiler (GHC) benutzt werden, der für zahlreiche Systeme frei verfügbar ist (siehe auch die Einführung zu Haskell im Anhang A).

1.3. Eine einfache Beispielsprache: Simple

Als Beispiel für eine einfach zu übersetzende Beispielsprache verwenden wir Simple, die nur Zuweisungen, einfache Ausdrücke (mit Seiteneffekten) und Anweisungssequenzen erlaubt und definiert ist durch eine kontextfreie Grammatik (hierbei wird die konkrete Syntax von Zahlen und Bezeichner weggelassen, d.h. diese sind Terminalsymbole für diese Grammatik):

Regel			Regelname
Stm	$\longrightarrow$	*Stm*; *Stm*	CompoundStm
Stm	$\longrightarrow$	*Id* `:=` *Exp*	AssignStm
Stm	$\longrightarrow$	`print(`*ExpList*`)`	PrintStm
Exp	$\longrightarrow$	*Num*	NumExp
Exp	$\longrightarrow$	*Id*	IdExp
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

Die Semantik von Simple sollte intuitiv klar sein.

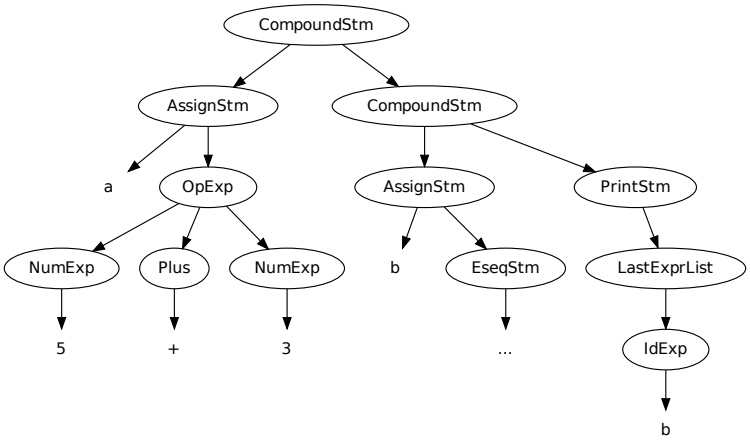
Beispiel: Die folgende Zeichenfolge ist ein einfaches Simple-Programm:

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

Ein Ablauf dieses Programms erzeugt die Ausgabe

```
8 7
80
```

Wenn dieses Programm in einem Übersetzer oder Interpreter verarbeitet werden soll, wird dessen Struktur als Ableitungsbaum dargestellt. In unserem Fall könnte dieser wie folgt aussehen (wobei wir in den inneren Knoten den Regelbezeichner schreiben):



Diese Baumstrukturen kann man in einer funktionalen Programmiersprache wie Haskell mittels folgender Datentypen darstellen:

```
type Id = String

data BinOp = Plus | Minus | Times | Div
  deriving (Eq, Show)

data Stm = CompoundStm Stm Stm
         | AssignStm   Id  Exp
         | PrintStm    [Exp]
  deriving (Eq, Show)

data Exp = NumExp  Int
         | IdExp   Id
         | OpExp   Exp BinOp Exp
         | EseqExp Stm Exp
  deriving (Eq, Show)
```

Das obige Beispielprogramm bzw. sein Ableitungsbaum könnte dann durch den folgenden Ausdruck/Datenterm dargestellt werden:

```
CompoundStm
  (AssignStm "a" (OpExp (NumExp 5) Plus (NumExp 3)))
  (CompoundStm
     (AssignStm
        "b"
        (EseqExp (PrintStm [IdExp "a",
                            OpExp (IdExp "a") Minus (NumExp 1)])
```

```
                (OpExp (NumExp 10) Times (IdExp "a"))))
      (PrintStm [IdExp "b"]))
```

Damit ist die Funktionalität der lexikalischen und syntaktischen Analyse leicht erklärbar: es ist eine Funktion vom Typ

`String` $\rightarrow$ `Stm`

Bevor wir zeigen, wie man eine solche Funktion implementieren kann, diskutieren wir die Begriffe von Programmiersprache, Interpreter und Übersetzer und deren Zusammenhänge.

2. Grundlagen

2.1. Programmiersprachen

Programmiersprachen basieren auf einem Zeichenvorrat wie ASCII, wir definieren daher 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]

$$L\colon A \multimap (A^* \multimap A)$$

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 geschrieben) eine Funktion $Ll\colon A^* \multimap A$, eine *Ein-/Ausgabefunktion*, die **Semantik** des Programms l in der Sprache L.

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

1. L-Prog $\subseteq A$ ist eine entscheidbare Teilmenge (d.h. die syntaktische und die semantische Analyse ist berechenbar)
2. $Ll\colon A^* \multimap A$ ist partiell rekursiv (turing-berechenbar, d.h. die Semantik von l ist implementierbar)

Falls L eine Datenbankanfragesprache ist (z.B. $L = \text{SQL}$), dann muss meist auch Ll entscheidbar sein für alle Anfrageprogramme $l \in L$-Prog, d.h. die Anfragen müssen auf allen Datenbankinstanzen terminieren.

Beachte: $l \in A$ alleine ist nur reine Syntax ohne Bedeutung, d.h. für verschiedene Sprachen L_1, L_2 kann durchaus $L_1 l \neq L_2 l$ gelten, was bedeutet, dass dasselbe Programm im Allgemeinen eine unterschiedliche Semantik bzgl. verschiedener Sprachen haben kann. Ein Beispiel kann z.B. das Pattern Matching in deklarativen Sprachen sein. Betrachten wir dazu das folgende Haskell-Programm:

```
f True False = True
f _    True  = False

loop = loop

test = f loop True
```

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

Die Auswertung von `test` terminiert in Haskell nicht. Dieses Programm ist jedoch auch ein syntaktisch zulässiges Curry-Programm[2] und in Curry wird `test` zu `False` ausgewertet.

DEFINITION: Sei $l \in L\text{-Prog}$. Dann sei der Definitionsbereich

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

Die L-Eingabe(l) umfasst also nur die Werte, für die das Programm l eine Ausgabe berechnet (und terminiert). Weiterhin ist die Ausgabe definiert durch

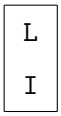
$$\begin{aligned} L\text{-Ausgabe}(l) &= Ll(L\text{-Eingabe}(l)) \\ &= \{x \in A \mid \exists\, (x_1, \ldots, x_n) \in A^* \colon Ll(x_1, \ldots, x_n) = x\} \end{aligned}$$

2.2. Interpreter

Interpreter sind selbst Programme, die in einer Sprache I geschrieben sind:
DEFINITION: Ein **Interpreter** i für eine Sprache L, geschrieben in der Sprache I, ist ein Programm $i \in I\text{-Prog}$, das für alle $l \in L\text{-Prog}$ und alle $(x_1, \ldots, x_n) \in L\text{-Eingabe}(l)$ folgendes erfüllt:

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

Das durch i interpretierte Programm $l \in L\text{-Prog}$ muss also dieselbe Semantik besitzen wie Ll, d.h. i muss die Semantik erhalten. Das Symbol



bezeichnet die Menge aller Interpreter für L geschrieben in I. Somit haben Interpreter L-Programme und deren Eingabe als Eingabe und berechnen die Ausgabe gemäß der L-Semantik.

Die obige Gleichung ist strikt, d.h. es ist nichts ausgesagt über die Fälle, bei denen Ll undefiniert oder Eingaben unzulässig sind. Im Beispiel aus Kapitel 1.3 ist $L = \mathsf{Simple}$ und $I = \mathsf{Haskell}$ (siehe Übung).

Generelle **Eigenschaften von Interpretern**:

- Funktionale Sprachen sind gut geeignet zur Implementierung von Interpretern.
- Interpreter sind einfacher als Übersetzer, damit ist auch die Korrektheit einfacher nachweisbar (wir benötigen keine Zielsprache und deren Semantik).
- Interpreter sind meistens langsam, benötigen dafür aber nur eine geringe Übersetzungs-/Startzeit – daher sind sie gut zum Testen geeignet.

Wünschenswert sind daher für eine Programmiersprache sowohl Interpreter als auch Übersetzer.

[2] http://www.curry-language.org

2.3. Übersetzer

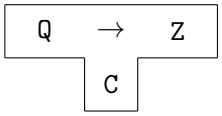
DEFINITION: Gegeben sei eine Quellsprache Q und eine Zielsprache Z. Ein C-Programm c heißt **Übersetzer** oder **Compiler** von Q nach Z falls:

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)

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

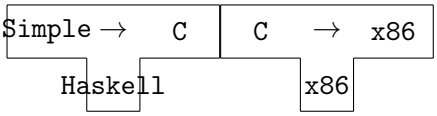
   Auch hier muss also die Semantik erhalten bleiben.

Als Symbol wird hier



verwendet, auch „T-Diagramm“ genannt. T-Diagramme [**?**, **?**] wurden schon frühzeitig zur graphischen Beschreibung komplexer Übersetzungsaufgaben verwendet.

Beispiele: $Q =$ Simple, $Z =$ C (oder Maschinencode), $C =$ Haskell. Oder für einen C-Compiler: $Q =$ C, $Z =$ 8086 und $C =$ 8086.
Es ist auch eine **Komposition** möglich, d.h. beispielsweise



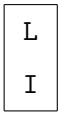
Zum **Unterschied** zwischen Interpretern und Übersetzern:

- Der Übersetzer „kennt“ das Quellprogramm, beispielsweise ist einem Übersetzer für Simple die Menge aller Identifier bekannt. So kann er ein Feld statt einer Liste für die Umgebung verwenden und dann die Zuweisung in eine indizierte Feldmodifikation übersetzen (konstante Laufzeit!).
- Ein Übersetzer hat im Allgemeinen durch die Kenntnis des Quellprogramms die Möglichkeit, viele Optimierungen im späteren Programmablauf vorzunehmen.

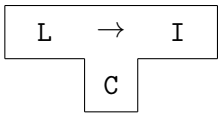
2.4. Konstruktion von Übersetzern aus Interpretern

Eine Methode zur Konstruktion von Compilern aus gegebenen Interpretern ist die **partielle Auswertung** des Interpreters. Damit hätte man Interpreter und Compiler für eine Programmiersprache zur Verfügung, sofern der Übersetzer per Konstruktion korrekt ist. Einen Korrektheitsnachweis müsste man dann nur noch für den Interpreter führen.

Gegeben sei also ein Interpreter i der Form



d.h. ein L-Interpreter in I. Gesucht ist ein Übersetzer c der Form



d.h. ein L-Übersetzer nach I, der in C geschrieben ist.

Idee: da i ein Interpreter ist, gilt:

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

Der Interpreter berechnet also für ein Programm l und eine Eingabe $(x_1, \ldots, x_n)$ die Ausgabe des Programms für diese Eingabe bzgl. der Semantik von L.
Nun transformieren wir für jedes $l \in L$-Prog das Programm i in ein neues Programm $\text{Restprog}(i, l) \in I$-Prog mit der Eigenschaft

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

Wir spezialisieren also i in Bezug auf das Programm l nach der folgenden **Transformationsmethode**: Setze in i den konkreten Parameter l ein und werte dann i überall da aus, wo es möglich ist (d.h. wo es von l abhängt), d.h. man wertet das Programm i an einigen Stellen aus (daher nennt man dies **partielle Auswertung** von i angewendet auf l). Ein partieller Auswerter c, der dies automatisch tut, ist dann ein Übersetzer:

$$Cc\colon l \mapsto \text{Restprog}(i, l)$$

wobei der partielle Auswerter c in der Sprache C geschrieben ist.

Beispiele für mögliche partielle Auswertungstransformationen:

- Ausrechnen von bekannten Werten:

  `3 + 2` $\rightsquigarrow$ `5`
  `3 == 3` $\rightsquigarrow$ `True`

- Bedingungen vereinfachen:

  `if True then` e_1 `else` e_2 $\rightsquigarrow$ e_1
  `if False then` e_1 `else` e_2 $\rightsquigarrow$ e_2

- „Auffalten“: Falls `f` definiert ist durch

  `f` $\mathtt{x}_1$ `...` $\mathtt{x}_n$ `= b`

dann ersetze einen Aufruf von `f`:

$$\texttt{f e}_1 \ldots \texttt{e}_n \quad \rightsquigarrow \quad \texttt{b[x}_1 \mapsto \texttt{e}_1\texttt{,...,x}_n \mapsto \texttt{e}_n\texttt{]}$$

- Spezialisierung: Falls `f` definiert ist durch

  $$\texttt{f y}_1 \ldots \texttt{y}_k \ \texttt{x}_1 \ldots \texttt{x}_n \texttt{ = b}$$

  und die ersten k Argumente (die Reihenfolge könnte natürlich auch anders sein) feste Werte $\texttt{c}_1, \ldots, \texttt{c}_k$ sind, dann erzeuge eine neue Funktion

  $$\texttt{f\_c}_1\texttt{\_...\_c}_k \ \texttt{x}_1 \ldots \texttt{x}_n \texttt{ = b[y}_1 \mapsto \texttt{c}_1\texttt{,...,y}_k \mapsto \texttt{c}_k\texttt{]}$$

  und ersetze

  $$\texttt{f c}_1 \ldots \texttt{c}_k \ \texttt{e}_1 \ldots \texttt{e}_n \quad \rightsquigarrow \quad \texttt{f\_c}_1\texttt{\_...\_c}_k \ \texttt{e}_1 \ldots \texttt{e}_n$$

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):

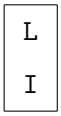
```
g z = f 4 z
    = if 4 > 0 then 3 * 4 + z else z
    = if True then 3 * 4 + z else z
    = 3 * 4 + z
    = 12 + z
```

Definition: Ein in einer beliebigen Sprache E geschriebenes 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)

Ein partieller Auswerter nimmt also ein Programm und einen Teil der Eingabe dieses Programms und wertet das Programm bezüglich dieser Eingabe aus. Das Resultat ist ein zweites Programm, das die restlichen Eingaben akzeptiert und sich so verhält wie das Originalprogramm mit allen Eingaben (Punkt 3. der obigen Definition).

Damit gilt: $PEVAL$ angesetzt auf einen Interpreter i der Form

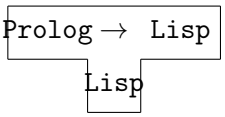


und ein Quellprogramm $l \in l$-Prog liefert ein Zielprogramm $i_1 = EPEVAL(i, l) \in I$-Prog, das in I geschrieben ist und folgende Eigenschaft hat:

$$\begin{aligned} I(EPEVAL(i,l))(x_1,\ldots,x_n) &= Ii_1(x_1,\ldots,x_n) \\ &= Ii(l,x_1,\ldots,x_n) \\ &= Ll(x_1,\ldots,x_n) \end{aligned}$$

Somit ist $EPEVAL(i)$ tatsächlich ein Übersetzer von L nach I in E. Man kann sich vorstellen, dass $EPEVAL(i)$ für ein Programm l durch Spezialisierung genau den I-Code erstellt, der bei der Interpretation von l durchlaufen werden könnte. Allerdings ist dieser Code häufig nicht sehr effizient, weil die partielle Auswertung sehr schematisch und nicht mit der „Intelligenz“ eines Übersetzers vorgeht.

Tatsächlich wurde eine Prolog-Implementierung auf dieser Basis erstellt [Kahn/Carlsson 84] mit $L = Prolog$ und $E = I = Lisp$, d.h. das Ergebnis ist



Allerdings ergab dies einen sehr langsamen Übersetzer, der etwa um den Faktor 100 langsamer war als ein handgeschriebener Übersetzer.

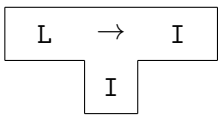
Verbesserung: Falls $E = I$ ist, d.h. der partielle Auswerter ist in der gleichen Sprache wie der Interpreter geschrieben, dann kann man $PEVAL$ auf sich selbst und den Interpreter i ansetzen, um einen schnelleren Übersetzer zu erhalten:

$$c := IPEVAL(PEVAL, i)$$

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 der Form



tatsächlich ein Übersetzer. Es wird also der recht aufwändige partielle Auswerter nur noch für den Interpreter aufgerufen, woraus man dann einen Übersetzer erhält. Durch die doppelte partielle Auswertung wird der Code der ersten partiellen Auswertung ebenfalls partiell ausgewertet und damit effizienter.

Beide Übersetzer sind dabei so entstanden, dass man einen gegebenen Interpreter partiell auswertet, wobei der Interpreter bekannt und damit festgelegt ist, aber das zu übersetzende Quellprogramm selbst erst später bekannt wird. Man kann aber noch eine Stufe weiter gehen und den Interpreter selbst erst später bekannt machen. Damit erhält man einen **Compilergenerator**, d.h. ein Programm, das aus einer Sprachbeschreibung einen Übersetzer generiert. In unserem Fall ist die Sprachbeschreibung ein Interpreter für diese Sprache. Somit wäre ein Compilergenerator ein Programm, das einen Interpreter als Eingabe erhält und dann einen Übersetzer als Ausgabe liefert. Tatsächlich ist dies mit partieller Auswertung möglich. In der Übung wird gezeigt, dass man einen solchen Compilergenerator *cc* durch folgende Konstruktion erhalten kann:

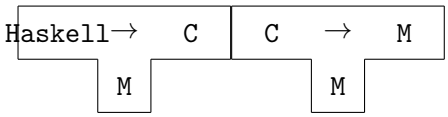
$$cc := IPEVAL(PEVAL, PEVAL)$$

2.5. Kombination von Interpretern und Übersetzern

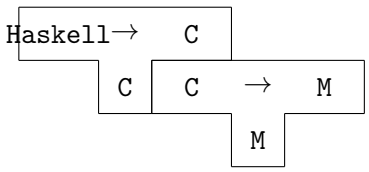
In der Praxis erfolgt häufig keine direkte Übersetzung eine Quellsprache in Maschinencode, sondern es wird eine Kombination aus Interpretern und Übersetzern eingesetzt. Die Vorteile hiervon sind:

- Die Komplexität der einzelnen Übersetzungsschritte ist geringer.
- Wiederverwendung existierender Übersetzer
- mehr Portabilität

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



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, d.h. den linken Compiler im vorigen Bild erhalten wir wie folgt:



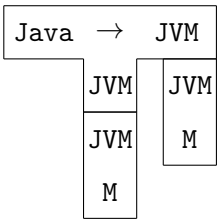
Eine weitere Möglichkeit zur Implementierung höherer Programmiersprachen ist die Verwendung **abstrakter Maschinen**, die eine Zwischenschicht der Übersetzung darstellen. Dies wird insbesondere dann gemacht, wenn eine direkte Abbildung von Quellsprachkonstrukten in Zielsprachkonstrukten schwierig ist, was gerade bei Hochsprachen wichtig ist, wo man von der von Neumann-Architektur weitgehend abstrahieren will.

Beispiele für abstrakte Maschinen sind:

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

Eigenschaften abstrakter Maschinen sind die Orientierung an den Konstrukten der Quellsprache (z.B. Unifikation in Prolog, unausgewertete Ausdrücke in Haskell), aber auch die Effizienz der Implementierung auf existierenden Maschinen (etwa durch Interpretation/Emulation oder durch einen weiteren Übersetzer).

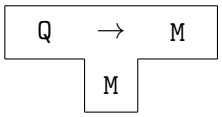
Der Ablauf beispielsweise bei Java: Der Code wird zu Byte-Code übersetzt auf JVM, danach dann interpretiert oder mit einem „Just-in-time"-Compiler übersetzt. Vorteil ist, dass der übersetzte JVM-Code portabel ist – Nachteil ist wiederum die Effizienz. Das Schema sieht hier wie folgt aus:



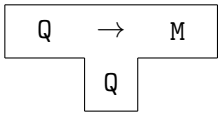
Ein weiterer Aspekt abstrakter Maschinen sind mögliche Analysen des Codes. Zum Beispiel kann der JVM-Code bzgl. Sicherheitsaspekten geprüft werden, d.h. die Überprüfung unberechtigter Zugriffe ist im JVM-Code einfacher als in reinem Maschinencode.

2.6. Bootstrapping

Das allgemeine Ziel bei der Konstruktion eines Übersetzers ist es, einen Übersetzer der Form



zu erzeugen. Das praktische Problem hierbei ist, dass Q eine Hochsprache ist und M in der Regel ein niedriges Programmierniveau hat, d.h. ungeeignet ist für die Erstellung komplexer 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 zu haben. Unser Ziel ist also die Entwicklung eines Übersetzers

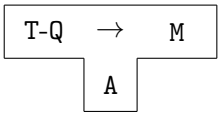


Leider ist dieser so nicht lauffähig!

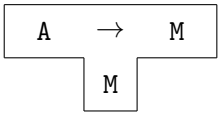
Wir können dies aber durch **Bootstrapping** in mehreren Schritten realisieren:

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

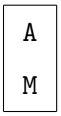
2. Schreibe einen Übersetzer C_1 der Form

   in einer beliebigen ausführbaren Sprache A, d.h. es existiert ein Übersetzer

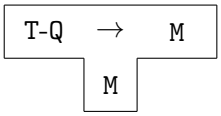
   

   oder ein Interpreter

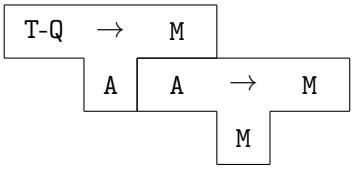
   

   Beispiele für A sind Assembler oder C.
   C_1 muß nur korrekt sein, aber kann eine schlechte Fehlerbehandlung haben oder nur ineffizienten Code erzeugen.

3. Erzeuge den `T-Q`-Übersetzer

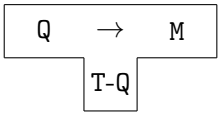
   

   durch

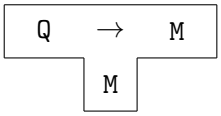
   Dieser Übersetzer kann auch aus einem einfachen Interpreter für `T-Q` automatisch generiert werden, siehe oben.

4. Schreibe den „eigentlichen“ Übersetzer C_2 für Q, aber noch nicht in Q sondern in der Sprache `T-Q`:

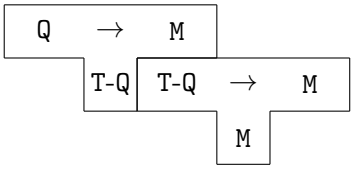


Dieser sollte den üblichen Qualitätsansprüchen bzgl. der Fehlerbehandlung und Codeeffizienz genügen.

5. Erzeuge nun eine ausführbare Version von C_2

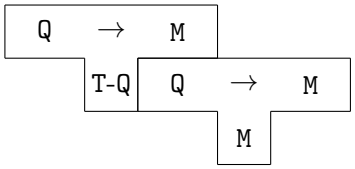


mittels (3), d.h. durch die Kombination



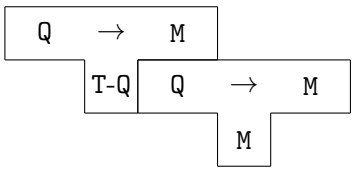
Dieser erzeugt **guten Code**, ist selbst jedoch **ineffizient**, falls C_1 ineffizienten Code erzeugt. Diesen Nachteil kann man nun allerdings durch **Selbstanwendung** beheben:

6. Erzeuge eine effizientere Implementierung von C_2 mit Hilfe von (5), d.h. wende C_2 aus (5) auf den Source-Code von C_2 selbst an und erhalte einen **effizienten Übersetzer** C_3:



Dieser erzeugt nun auch effizienten Code. Für diesen Bootstrap-Schritt ist es wichtig, dass `T-Q` eine Teilmenge von `Q` ist!

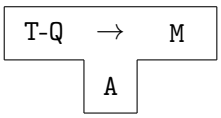
7. Eine weitere Selbstanwendung bringt nichts, ist aber trotzdem nützlich als **Praxistest**: Implementiere C_2 durch Anwendung von C_3 auf den Sourcecode von C_2 (bzw. C_3):



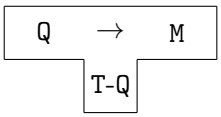
Dieser erhaltene Übersetzer C_4 sollte **textuell bzw. binär identisch** zu C_3 sein.

Man benötigt also zwei Dinge zum Bootstrapping:

1. Einen einfachen Übersetzer C_1 in einer existierenden Sprache:

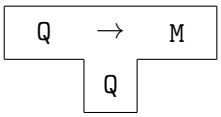


2. Einen „guten“ Übersetzer C_2 für die Sprache Q in T-Q:

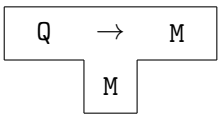


Vorteile:

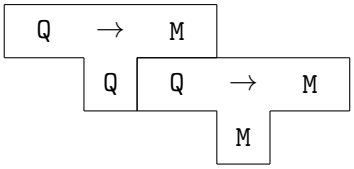
1. Bei allen Verbesserungen und Erweiterungen von C_2 in der Form C_2' ist nun der volle Sprachumfang von Q verwendbar, d.h. C_2' ist von der Form



2. Jede Verbesserung in der Codeerzeugung von C_2 kommt C_2 selbst zugute: Ein verbesserter Compiler der Form



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, indem man nur das Backend anpasst.

Diese Vorgehen wurde von Wirth am Beispiel der Implementierung von Pascal umgesetzt und ist heute eine wichtige Implementierungstechnik im Übersetzerbau.

3. Lexikalische Analyse

3.1. Ziel der lexikalischen Analyse

Beispiel: Betrachten wir den folgenden Programmausschnitt:

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

Der Übersetzer muss dieses Programm analysieren, um korrekten Code zu erzeugen. Diese Analyse erfolgt wie beim Lesen/Verstehen von Text in zwei Phasen:

1. **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. **Syntaktische Analyse**: Erkennen der Programmstruktur (Ausdrücke, bedingte Anweisungen, ...)

Ziel der lexikalischen Analyse ist also die Zerlegung einer Zeichenfolge (Quellprogramm) in eine Folge von **Lexemen** (lexikalische atomare Einheiten, von griechisch *lexis* „Wort“), auch **Token** genannt. Zudem werden die Lexeme in **Symbolklassen** eingeteilt, d.h. in Mengen von Lexemen, die für die syntaktische Analyse als gleichartig anzusehen sind. 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 (dies ist z.B. wichtig für Fehlermeldungen). Ein Programm für die lexikalische Analyse heißt **Scanner** (**Zerhacker**). Formal ist dies eine Abbildung

$$\text{scan}\colon \Sigma^* \to T^*$$

mit einem Eingabealphabet Σ und einem Tokenklassentyp T.

Beispiel: Die Auswertung des Ausdrucks

```
scan "obigerProgrammstring"
```

ergibt:

```
[Int, Id "abs", LParen, Int, Id "x", RParen, LBrace, If,...]
```

Ein Scanner überliest in der Regel Kommentare und Leerzeichen (nicht aber bei javadoc, lint etc.) und unterscheidet zwischen

- *Schlüsselwörtern* wie `if`, `else`, ... und
- *Bezeichnern* (Klasse `Id`).

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

- $\Sigma \cup \{\varnothing, \varepsilon\} \subseteq \mathrm{RA}(\Sigma)$ ($\varnothing$: leere Menge; ε: leere Zeichenfolge)
- $\alpha, \beta \in \mathrm{RA}(\Sigma) \Longrightarrow \alpha|\beta \in \mathrm{RA}(\Sigma), \alpha \cdot \beta \in \mathrm{RA}(\Sigma), \alpha^* \in \mathrm{RA}(\Sigma)$

Jeder reguläre Ausdruck beschreibt eine Menge von Zeichenketten (Sprache). Daher ist die Semantik der regulären Ausdrücke durch eine Funktion $[\![\cdot]\!]\colon \mathrm{RA}(\Sigma) \to 2^{\Sigma^*}$ definiert:

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

Weitere Abkürzungen, die in vielen Werkzeugen zur Verarbeitung regulärer Ausdrücke verwendet werden, sind z.B.

- $[a-z]$ ist die Menge aller Zeichen zwischen a und z, also $a|b|c|\cdots|z$
- $\alpha?$ bedeutet $\alpha|\varepsilon$ (Option)
- α^+ bedeutet $\alpha \cdot \alpha^*$ (Wiederholung, mindestens einmal)

Beispiel: In der konkreten Notation wird der Punkt "$\cdot$" bei der Konkatenation oft weggelassen:

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

3.3. Implementierung

Die Übersetzung eines regulären Ausdrucks in ein Scanner-Programm kann in mehreren Schritten erfolgen.

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 $q_0 \in Q$, Endzuständen $F \subseteq Q$ und einer Zustandsübergangsfunktion

$$\delta \colon Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$$

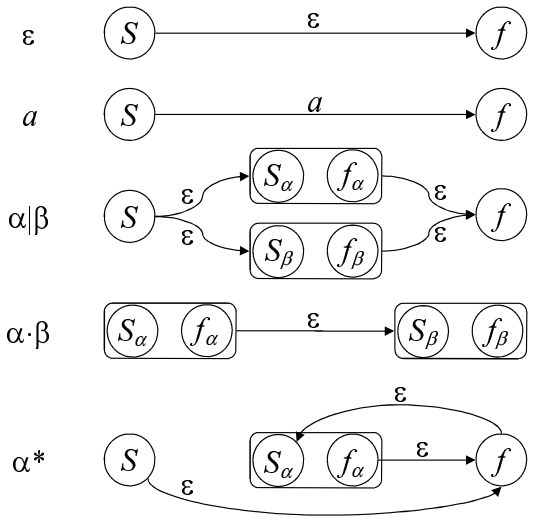
Beispiel: Ein Automat zur Erkennung von `Id` und `If` (siehe Vorlesung).
Hierbei können die folgenden Unklarheiten entstehen:

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

Übliche Festlegung:

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

Zur genauen Beschreibung der Übersetzung eines regulären Ausdrucks in einen NFA nehmen wir an, dass jeder NFA einen Start- und einen Endzustand hat (dies ist immer möglich mittels ε-Transitionen). Dann können wir reguläre Ausdrücke folgendermaßen in NFAs übersetzen:



Bei dieser Übersetzung kann man redundante Zustände gleich verschmelzen.

3.3.2. Übersetzung eines NFA in einen DFA

Problem der NFA-Übersetzung: Ein Rechner kann nicht „raten“, daher ist ein NFA nicht zur Implementierung geeignet.
Lösung: Konvertiere den NFA in einen **DFA** (wie NFA, aber mit eindeutiger (partieller) Übergangsfunktion $\delta\colon Q \times \Sigma \rightarrow\!\!\!\!\circ\ \ Q$). Es fehlen jedoch die ε-Übergänge, die ein NFA machen kann. Daher repräsentieren wir eine Menge von NFA-Zuständen als *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 in jeder Iteration immer nur Zustände hinzugefügt werden.

Damit kann man einen NFA deterministisch simulieren:

```
T := ε̂({q0});
while not final(T)
do
    read c;
    T := ε̂(⋃_{q∈T} δ(q, c))
done
```

Hierbei ist `final(T) = (T ∩ F ≠ ∅)` (Endzustände).

Die Berechnung des ε-Abschlusses ist jedoch aufwändig, 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 auf erreichbare Zustände reduziert werden)
- $F' = \{T \subseteq Q \mid T \cap F \neq \varnothing\}$
- $q_0' = \hat{\varepsilon}(\{q_0\})$
- $\delta' \colon Q' \times \Sigma \to Q'$ sei definiert durch $\delta'(T, a) = \hat{\varepsilon}\left(\bigcup_{q \in T} \delta(q, a)\right)$

Der konstruierte Automat enthält zu viele Zustände („Zustandsexplosion“). Diese Anzahl kann aber reduziert werden:

- Erzeuge nur die von q_0' erreichbaren Zustände, d.h. starte mit q_0' und berechne dann iterativ alle weiteren erreichbaren Zustande.
- Verschmelze „äquivalente“ Zustände, d.h. Zustände, von denen aus nur gleiche Zeichenfolgen erkannt werden können („Automatenminimierung“).

3.3.3. Praktische Aspekte der Scannerimplementierung

- Schreibe an die Endzustände die erkannte Symbolklasse zur Scannerausgabe (bei mehreren Tokenklassen: die textuell erste, siehe oben)!
- Umsetzung des „longest match“: Speichere den letzten erfolgreichen Endzustand und die Position; bei Nichtendzuständen ohne weiteren Übergang setze auf den letzten (gespeicherten) Endzustand zurück.
- Bei vielen Schlüsselwörtern entstehen viele Zustände – hier ist folgende Reduktion möglich: Erkenne Schlüsselwörter zunächst genauso wie Bezeichner. Falls der Scanner dann “`Id x`” erkennt, so prüfe, ob `x` ein Schlüsselwort ist: wenn ja, dann gib das entsprechende Token zurück, andernfalls “`Id x`”. Dieses Vorgehen ist vor allem empfehlenswert bei handgeschriebenen Scannern.
- Implementierung des DFA als Programm: Entweder durch eine Tabelle (δ') mit einen „Interpreter“ dafür oder durch Übersetzung in `case`-Ausdrücke.

- Die **Scannerimplementierung** ist damit voll automatisierbar. Daher existieren vollständige **Scanner-Generatoren** (z.B. lex oder alex), die reguläre Ausdrücke und zugehörige Symbolklassen in ein Scanner-Programm übersetzen.

4. Syntaktische Analyse

4.1. Einleitung

Die **Aufgabe** der syntaktischen Analyse ist die Erkennung der syntaktischen Einheiten einer Programmiersprache (Ausdrücke, Anweisungen, Prozeduren) aus der Folge der lexikalischen Einheiten (Token). Ein Programm, das diese Aufgabe erledigt, nennt man einen **Parser** für die syntaktische Analyse.

Beachte, dass Scanner für diese Aufgabe in der Regel nicht ausreichend sind, da viele Konstrukte in Programmiersprachen nicht regulär sind. Betrachten wir z.B. Ausdrücke einer Programmiersprache, deren Struktur wie folgt definiert ist:

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

Bei Ausdrücken müssen z.B. die Klammerpaare ausgeglichen sein – daher muss der Parser „Klammern zählen“, was mit DFAs nicht möglich ist (dies kann man formal mit dem Pumping-Lemma für reguläre Sprachen zeigen), d.h. Scanner sind nicht ausreichend. Daher erfolgt die Syntaxbeschreibung durch *kontextfreie Grammatiken (CFG)* und die Implementierung durch eine *Stackmaschine* („Zählen mit Hilfe eines Stacks“).

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** $(a, b, c, \ldots)$ mit $N \cap T = \varnothing$
- einem Startsymbol $S \in N$
- einer Menge P von Produktionen 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 eine Teilmenge von $V^* \times V^*$ mit

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

Falls $\alpha_1 \in T^*$, so nennt man die Ableitung eine **Linksableitung** (schreibe $\alpha \xrightarrow{L} \beta$), von einer **Rechtsableitung** spricht man im Fall $\alpha_2 \in T^*$ (schreibe $\alpha \xrightarrow{R} \beta$). Der reflexive, transitive Abschluss ist $\longrightarrow^*$. Die erzeugte **Sprache** ist dann definiert als

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

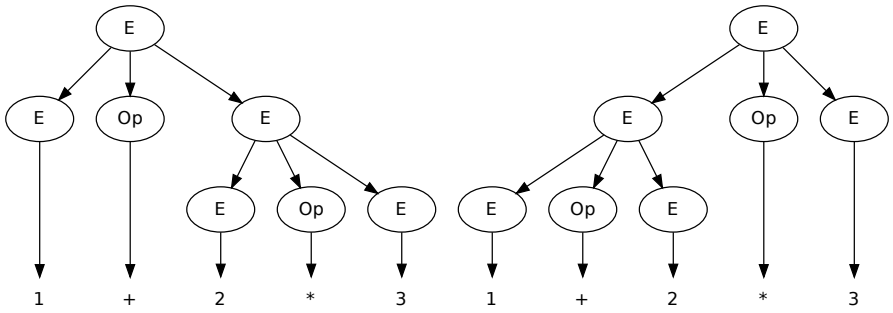
BEISPIEL: Eine kontextfreie Grammatik für Ausdrücke ist definiert durch das Startsymbol E und durch

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

(Die obige Notation ist nur eine besser lesbare Darstellung dieser Grammatik.)

Ableitungen in der Grammatik werden üblicherweise durch einen **Ableitungsbaum** dargestellt. Dieser repräsentiert die syntaktische Struktur eines Wortes. Die Wurzel enthält das Startsymbol, die Blätter nur Terminalsymbole, innere Knoten und deren Kindknoten entsprechen den angewendeten Produktionen.

Beispiel für Ableitungsbäume (hier schreiben wir konkrete Zahlen anstelle des Token `num`):



Beide Bäume sind strukturell verschieden, d.h. das gleiche Wort `1+2*3` ist auf zwei Weisen darstellbar – welche Darstellung soll der Übersetzer verwenden? Dies ist sehr relevant, denn das Berechnungsergebnis ist bei beiden Bäumen verschieden (7 bzw. 9).

Forderung: Das darf nicht vorkommen!

DEFINITION: Eine kontextfreie Grammatik G ist **eindeutig**, wenn es für jedes $w \in L(G)$ genau einen Ableitungsbaum gibt, sonst heißt sie **mehrdeutig**.

Vermeidung durch Codierung von Bindungsstärken:

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

Mit dieser Grammatik kann die Struktur des obigen rechten Ableitungsbaums nicht mehr erzeugt werden.

4.2. Parser mit rekursivem Abstieg (Recursive-Descent Parsing)

Beliebige kontextfreie Grammatiken können mit dem sogenannten CYK-Algorithmus (nach Cocke, Younger, Kasami) analysiert werden. Dieser Parsing-Algorithmus ist für allgemeine kontextfreie Grammatiken schon sehr gut, aber er benötigt im Allgemeinen $\mathcal{O}(n^2)$ Platz und $\mathcal{O}(n^3)$ Zeit. Da dies für die Analyse von Programmen in der Praxis zu langsam ist (hier sind lineare Verfahren wünschenswert), betrachtet man spezielle Verfahren und dazu passende spezielle kontextfreie Grammatiken.

Ein Verfahren, das besonders für das Schreiben von Parsern „von Hand“ geeignet ist, sind **recursive-descent parser** (**Parser mit rekusivem Abstieg**), die auf der folgenden Idee beruhen:

- Jedem Grammatiksymbol wird ein Parser bzw. eine Parserprozedur/-funktion zugeordnet.
- Jeder Parser konsumiert die Liste der Token von links nach rechts in der Eingabereihenfolge.
- Terminalsymbole konsumieren von dieser Liste das entsprechende Token.
- Nichtterminale werden durch Prozeduren/Funktionen repräsentiert, die dann wiederum die Parser für die rechten Seite einer Produktion aufrufen.
- Verschiedene Produktionen sind Fälle zur Definition der Nichtterminal-Prozeduren.

Beispiel: Betrachten wir die folgende 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

```
scan :: String → [Token]
```

zur Verfügung, z.B. mit

```
data Token = Id | Eq | Num | Print | Begin | End | Semi
  deriving (Eq, Show)
```

Der Parser ist nun eine Funktion, die als Eingabe eine Tokenliste erhält (diese entspricht dem Eingabeprogramm) und eine Liste der restlichen Token (die nicht vom Parser gelesen wurden) ausgibt. Damit gilt:

```
type Parser = [Token] → [Token]
```

Das Erkennen eines Terminalsymbols kann durch den folgenden einfachen Parser erfolgen, der über dem zu erkennenden Terminalsymbol parametrisiert ist:

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

Weiterhin benötigen wir zwei Parser für die beiden Nichtterminalsymbole:

```
-- Parser fuer das Nichtterminalsymbol S
parse_S :: Parser
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

-- Parser fuer das Nichtterminalsymbol L
parse_L :: Parser
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` kann nun durch folgenden Aufruf erfolgen:

```
parse_S (scan p)
```

Wenn das Ergebnis die leere Tokenliste `[]` ist, dann ist das Programm syntaktisch korrekt. Wenn wir uns die Struktur dieses Programms genauer ansehen, dann erkennen wir, dass eine Folge von Symbolen immer als Schachtelung von `lets` implementiert wird. Wir können die Lesbarkeit des Programms verbessern, wenn wir hierfür einen *Sequenzoperator* einführen: Durch die Haskell-Deklaration

```
infixr 4 <*>
```

wird ein rechtsassoziativer (daher infix*r*) Infixoperator der Bindungsstärke 4 eingeführt. Dieser hat als Eingabe zwei Parser `p1` und `p2` und liefert als Ergebnis einen Parser, der die Sequenz von `p1` und `p2` erkennt. Da dieser Operator zwei Parser kombiniert, wird er auch als **Parserkombinator** bezeichnet. Wir können diesen wie folgt definieren:

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

Damit können wir die Parser von oben nun wie folgt definieren:

```
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) ts
```

Dies entspricht genau den Grammatikregeln! Somit erhalten wir einen rekursiven Abstiegsparser relativ direkt durch Übersetzung der Grammatik in ein Programm bestehend aus rekursiven Funktionen, wobei durch ein `case` die verschiedenen Regeln für ein Nichtterminal unterschieden werden.
Leider klappt diese einfache Umsetzung nicht immer. Wenn wir beispielsweise Ausdrücke durch $E \to E + T$ und $E \to T$ definieren, dann entsteht nach obigem Schema das Programm

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

Das Problem ist nun also die Entscheidung, welche Produktion angewendet werden muss! Eine wünschenswerte Lösung wäre, dass diese Entscheidung nur durch Ansehen des ersten Token zu treffen ist, so dass wir die Grammatiken einfach in einen Parser umsetzen können.
Leider ist eine solche einfache Umsetzung nicht immer möglich, so dass wir später auch mächtigere Parsing-Techniken (*LR-Parsing*) kennen lernen werden. Alternativ kann man auch versuchen, die Grammatik so zu verändern, dass man einfach einen rekursiven Abstiegsparser verwenden kann. Auch damit werden wir uns später beschäftigen. Zunächst einmal schauen wir uns aber die Grundlagen der rekursiven Abstiegsparser genauer an, um eine Technik zu entwickeln, wie man die Auswahl zwischen verschiedenen Regelalternativen systematisch herleiten kann.

4.2.1. LL(1)-Parsing

LL(1) steht für ***l**eft-to-right parsing with **l**eftmost derivation (Linksableitung) and **1**-symbol-lookahead (Vorausschau)*. Dieselbe Idee kann man auch für eine Vorausschau mit $k > 1$ entwickeln, so dass man im Allgemeinen vom LL(k)-Parsing spricht, allerdings wird in der Praxis überwiegend mit $k = 1$ gearbeitet.

Die **Idee** hierbei ist es, für alle rechten Seiten der Produktionen eines Nichtterminalsymbols die Menge möglicher erster Token zu berechnen. Falls diese Mengen disjunkt sind, liefert uns dies eine Entscheidungsgrundlage für die anzuwendende Produktion.

DEFINITION: Sei $w \in T^*$ ein Wort. Dann ist

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

eine Funktion zur Berechnung der (maximal) k ersten Terminalsymbole von w. Hierbei bezeichnet $|w|$ die Länge von w.

DEFINITION: Eine kontextfreie Grammatik $G = (N, T, P, S)$ heißt **LL(k)-Grammatik**, falls gilt: Sind

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

zwei Linksableitungen mit $\text{start}_k(v) = \text{start}_k(w)$, so ist $\beta_1 = \beta_2$.

Intuitiv: Falls u gelesen wurde, dann ist aufgrund der nächsten k Zeichen eindeutig entscheidbar, welche Produktion angewendet werden muss.

Die Anwendung dieser Eigenschaft birgt aber noch Probleme:

- Die Entscheidung sollte unabhängig vom Kontext sein! Ansonsten muss man den Kontext (d.h. den gelesenen Teil u) in die Entscheidung einbeziehen.
- Das Verfahren ist nicht konstruktiv: Wie trifft man effizient die Entscheidung für eine bestimmte Regel?

Aus diesem Grund schränken wir die Klasse der Grammatiken weiter ein:

DEFINITION: Eine kontextfreie Grammatik $G = (N, T, P, S)$ heißt **starke LL(k)-Grammatik**, falls gilt: Sind

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

zwei Linksableitungen mit $\text{start}_k(v) = \text{start}_k(w)$, so ist $\beta_1 = \beta_2$.

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

Wie kann man in einer konkreten Situation die Entscheidung treffen? Berechne dazu für jede Satzform die Menge der daraus ableitbaren Terminalfolgen (bis zur Länge k):

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

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

Die Motivation hierfür ist, dass man versucht, die LL(k)-Entscheidung auf der Basis der FIRST_k-Mengen der rechten Regelseite für ein Nichtterminalsymbol zu treffen. Ein Problem ergibt sich allerdings, wenn diese kürzer als k sind. In diesem Fall ist der „nachfolgende Kontext“ des Nichtterminalsymbols relevant:

DEFINITION: Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik, $A \in N$ und $k > 0$. Dann definiere die folgende Menge:

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

Intuitiv: $\text{FOLLOW}_k(A)$ enthält alles, was hinter A bei Ableitungen folgen kann. Somit kann die Konkatenation von FIRST_k einer rechten Regelseite und FOLLOW_k für das Nichtterminalsymbol für die notwendige Entscheidung verwendet werden. Aus diesem Grund definieren wir für jede Regel eine Steuermenge:

DEFINITION: Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik und $(A \longrightarrow \alpha) \in P$ eine Produktion dieser Grammatik. Dann heißt

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

die **Steuermenge** (**director set**) für diese Produktion.

Mit Hilfe der Steuermengen können wir nun die LL(k)-Entscheidung treffen:

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

BEWEIS: siehe [Güting/Erwig 99, S. 60]

Somit sind zur Konstruktion eines Parsers folgende Schritte nötig:

- Berechne die Steuermengen der Produktionen (diese enthalten nur endlich viele Terminalsymbole und sind daher durch eine Fixpunktiteration berechenbar, s.u.).
- Prüfe die Disjunktheit der Steuermengen: falls diese disjunkt sind, dass ist die Grammatik eine starke LL(k)-Grammatik.
- Beim Parsen: prüfe die nächsten k Zeichen auf Zugehörigkeit zu einer Steuermenge und wähle die entsprechende Produktion aus.

In der Praxis beschränkt man sich häufig auf $k = 1$, um die Steuermengen nicht zu groß werden zu lassen. Wir verwenden entsprechend in Zukunft $k = 1$ und schreiben einfach FIRST für FIRST_1 und FOLLOW für FOLLOW_1.

Für $k = 1$ gilt außerdem: Jede LL(1)-Grammatik ist auch starke LL(1)-Grammatik!

Somit haben wir folgendes Kriterium für die Entscheidung, ob eine Grammatik eine 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. $\text{FIRST}(\beta) \cap \text{FIRST}(\gamma) = \varnothing$
2. Falls $\varepsilon \in \text{FIRST}(\beta)$, dann ist $\text{FOLLOW}(A) \cap \text{FIRST}(\gamma) = \varnothing$

Damit können die Steuermengen einfacher definiert werden:

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

Nun können wir einen Algorithmus zur Berechnung von FIRST und FOLLOW angeben, der eine Iteration über die Regeln durchführt und zusätzlich das Prädikat

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

berechnet.

4.2.2. Algorithmus zur Berechnung von FIRST und FOLLOW

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

for each z ∈ T
    FIRST(z) = {z};

repeat
    for each A → α_1 ... α_k      (α_i ∈ N ∪ T)
        if  leer(α_1) ∧ ... ∧ leer(α_k)
        then  leer(A) := true;
        for i = 1, ..., k
            if  leer(α_1) ∧ ... ∧ leer(α_{i-1})   (or i = 1)
            then  FIRST(A) := FIRST(A) ∪ FIRST(α_i);
            if  leer(α_{i+1}) ∧ ... ∧ leer(α_k)   (or i = k)
            then  FOLLOW(α_i) := FOLLOW(α_i) ∪ FOLLOW(A);
            for j = i + 1, ..., k
                if  leer(α_{i+1}) ∧ ... ∧ leer(α_{j-1})   (or i + 1 = j)
                then  FOLLOW(α_i) := FOLLOW(α_i) ∪ FIRST(α_j);
until  FIRST, FOLLOW, leer  are not changed
```

Als Beispiel betrachten wir die folgende Grammatik:

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

0. Initialisierung:

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

1. Iteration:

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

2. Iteration

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

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

4.2.3. Konstruktion einer Parsing-Tabelle

Mittels dieser berechneten Information können wir eine Parsing-Tabelle zur Steuerung des Recursive-Descent-Parsers (RD-Parser) konstruieren. Unser RD-Parser entscheidet nach dem aktuellen Nichtterminal und dem nächsten Zeichen, welche Regel angewendet wird. Implementiert wird dies durch eine Parsing-Tabelle vom Typ

$$N \times T \to P$$

Hierzu erweitern wir zunächst FIRST auf Folgen von Grammatiksymbolen:

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

Nun können wir die Tabelle wie folgt konstruieren: Sei $A \to \beta \in P$. Dann füge in die Tabelle Produktionen nach folgenden Regeln ein:

- Falls $t \in \text{FIRST}(\beta)$ ist, so füge $A \to \beta$ an der Stelle (A, t) in die Tabelle ein.
- Falls $\text{leer}(\beta)$ gilt (d.h. es gilt $\text{leer}(\alpha_i)$ für $i = 1, \dots, n$ wenn $\beta = \alpha_1 \dots \alpha_n$ ist) und $t \in \text{FOLLOW}(A)$, so füge $A \to \beta$ an der Stelle (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 die 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	$Z \to XYZ$	$Z \to XYZ$	$Z \to XYZ$ $Z \to d$

Die Beispielgrammatik ist also keine LL(1)-Grammatik. Die Ursache liegt in der Mehrdeutigkeit dieser Grammatik. Z.B. gibt es für die Ableitung von d mehrere Möglichkeiten:

$$Z \to d$$

$$Z \to XYZ \to YZ \to Z \to d$$

Ist eine Grammatik nicht LL(1), dann kann man häufig durch bestimmte Modifikationen eine äquivalente Grammatik erhalten, die die LL(1)-Eigenschaft hat. Zwei wichtige Methoden hierzu diskutieren wir im Folgenden.

4.2.4. Elimination von Linksrekursion

Ein häufiges Problem für die LL(1)-Analyse sind **linksrekursive Regeln**. Betrachten wir z.B. den folgenden Ausschnitt einer Grammatik für Ausdrücke:

$$\begin{aligned} E &\to E + T \\ E &\to T \end{aligned}$$

Ist $t \in \text{FIRST}(T)$, dann gilt auch $t \in \text{FIRST}(E + T)$, was sofort zu Mehrfacheinträgen in der Tabelle an der Stelle (E, t) führt, so dass diese Grammatik nicht LL(1) ist. Die Ursache ist die erste linksrekursive Regel, da damit die Auswahl der anzuwendenden Regel nicht mit beschränkter Vorausschau entscheidbar ist.

Lösung: Transformation von Links- in Rechtsrekursion, im Beispiel ergeben sich die Regeln

$$\begin{aligned} E &\to TE' \\ E' &\to +TE' \\ E' &\to \end{aligned}$$

Nun ist die Entscheidung zur Auswahl der E'-Regel durch Vorausschau möglich:

- Falls das erste Symbol '+' ist, dann wähle die Regel $E' \to +TE'$

- Sonst: wähle die Regel $E' \rightarrow$ (unter der Annahme, dass $+ \notin \text{FOLLOW}(E')$)

Allgemeines Transformationsschema: Gegen sei $X \rightarrow X\beta$ und $X \rightarrow \alpha$ (mit $\alpha \neq X \ldots$). Somit ist $X \rightarrow^* \alpha\beta^*$, wobei die rechte Seite regulär ist und damit erzeugbar durch Rechtsrekursion mit Hilfe der Regeln

$$\begin{aligned} X &\rightarrow \alpha X' \\ X' &\rightarrow \beta X' \\ X' &\rightarrow \end{aligned}$$

4.2.5. Linksfaktorisierung

Die LL-Analyse benötigt immer eine Vorausschau, um die anzuwendende Regel zu erkennen. Dies ergibt ein Problem, falls zwei Regeln mit einem gleichen Präfix beginnen, beispielsweise:

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

Die Lösung dieses Problems besteht in einer **Linksfaktorisierung**: Ziehe den gemeinsamen Präfix heraus in eine Regel und führe ein neues Nichtterminal für die unterschiedlichen Enden ein:

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

Übung: Funktioniert dies auch, wenn das `fi` fehlt?

4.2.6. Fehlerbehandlung in RD-Parsern

Bei der bisherigen Konstruktion eines Parsers für Ausdrücke sind wir in mehreren Schritten vorgegangen:

1. Der Ausgangspunkt war eine einfache, aber mehrdeutige Grammatik (z.B. mit der Regel $E \rightarrow E\ Op\ E$).
2. Diese wurde in eine eindeutige Grammatik durch Einführung neuer Nichtterminalsymbole überführt ($E \rightarrow E + T$, $E \rightarrow T$,...)
3. Anschließend wurde Linksrekursion durch Transformation der Grammatikregeln eliminiert.

Das Ergebnis ist eine Grammatik der folgenden Struktur (hier ist diese um ein Startsymbol S und einen „end-of-file“-marker \$ erweitert):

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

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)	$F \to (E)$		

Auf der Basis dieser Parsing-Tabelle können wir ein Parser-Programm konstruieren, von dem wir einen kleinen Ausschnitt zeigen:

```
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
    t      → error ("T: " ++ show t)

parse_T' ts = case (head ts) of
    Plus   → ts
    Times  → (terminal Times <*> parse_F <*> parse_T') ts
    RParen → ts
    EOF    → ts
    t      → error ("T': " ++ show t)
```

Der Aufruf `error` weist auf einen Syntaxfehler hin, d.h. das aktuelle Token wird nach der Grammatik nicht erwartet. Was soll man an diesen Stellen machen?

- Fehlermeldung und Abbruch ist benutzerunfreundlich!
- Besser: Fehlermeldung, Reparatur des Fehlers und weitermachen: Dies könnte durch Änderung der Eingabe zu einem korrekten Satz durch
  - Löschen eines Tokens
  - Ersetzen eines Tokens durch ein anders
  - Einfügen eines neuen Tokens

  erfolgen.

Beispiel: Bei `(error "T")` könnte man ein `Id`- oder `Num`-Token einfügen: dies muss nicht real gemacht werden, sondern „virtuell“, d.h. der Parser führt eine entsprechende

Aktion aus, als ob das Token vorhanden wäre. Hier: verlasse `parse_T`, ohne die Eingabe weiterzulesen:

```
error "T"     ≈   trace "Expected id, num or left paren" ts
```

Das (virtuelle) Einfügen von Token kann jedoch gefährlich sein, da dies weitere Fehler verursachen kann, so dass die Fehlerbehandlung eventuell nicht terminiert!

Sicherer ist es, Token zu überspringen (Löschen) bis zu einem „sinnvollen“ Punkt, z.B. bis das nächste Token zur FOLLOW-Menge des Nichtterminals gehört.

Beispiel: Es gilt FOLLOW$(T') = \{), +, \$\}$. Somit könnte man `error "T'"` durch den Code

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

ersetzen, wobei die Hilfsoperation `skipTo` wie folgt definiert ist:

```
skipTo stopset ts = dropWhile (`notElem` stopset) ts
```

Ein generelles Problem bei der Fehlerbehandlung ist das Erzeugen von Folgefehlern, die durch die Korrektur entstehen – es kann zu Kaskaden von Fehlern nur durch ein falsches Token kommen. Daher benötigt eine gute Fehlerbehandlung eine Menge „fine tuning“ von Hand, wobei typische Programmierfehler einer Sprache zu berücksichtigen sind.

4.3. Bottom-Up-Analyse

Wir haben gesehen, dass die LL(k)-Analyse mittels RD-Parsern leicht zu implementieren ist, aber einige Schwachpunkte (Linksrekursion, Regeln mit gleichem Präfix) hat. Die Ursache liegt darin, dass die LL(k)-Analyse auf der Basis der nächsten k Eingabesymbole erraten muss, mit welcher Produktion die nachfolgende Eingabe abgeleitet werden kann. Diese Schwäche kann man vermeiden, indem man zunächst die Eingabesymbole liest und danach versucht zu bestimmen, mit welcher Regel diese ableitbar sind. Dies führt zu einer **Bottom-Up-Analyse**, bei der der Ableitungsbaum von unten nach oben konstruiert wird. Im Folgenden beschäftigen wir uns konkret mit der LR(k)-Analyse.

4.3.1. LR(k)-Analyse

- am weitesten verbreitete Bottom-Up-Analysetechnik (eine andere Bottom-Up-Analysetechnik ist die Operator-Präzedenz-Analyse)
- *L*eft-to-right parsing (lese Eingabe von links nach rechts)
- *R*ightmost derivation (konstruiere eine Rechtsableitung)
- k Symbole Vorausschau

- mächtige Parsing-Technik, aber auch komplexere Konstruktion von Parsing-Tabellen, die von Hand schwer erstellbar sind
- automatische Generierungswerkzeuge (Parsergeneratoren wie z.B. Yacc) existieren

Grundidee der LR(k)-Analyse: 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 eine

- *shift*-Aktion ausführt: schiebe das nächste Eingabesymbol auf den Stack, oder eine
- *reduce*-Aktion ausführt: vereinfache den Stackinhalt.

Daher heißen diese Parser auch **shift-reduce-Parser**. Die Reduzierung des Stacks bei einer reduce-Aktion 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 folgende eindeutige Grammatik für Ausdrücke:

$E \to E + T$	$T \to T * F$	$F \to$ `id`
$E \to T$	$T \to F$	$F \to$ `num`
		$F \to$ `(`E`)`

Als Eingabe betrachten wir `id + id * id`. Dann ist der Ablauf eines shift-reduce-Parser folgendermaßen:

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*

An der Stelle *(*)* wäre im Prinzip auch eine reduce-Aktion möglich gewesen, allerdings gibt es keine Regel zur Ableitung von $E*\ldots$. Aus diesem Grund hat der Parser dort eine shift-Aktion ausgeführt.

Der Vorgang *reduce* entspricht einer Regelanwendung. Der Parser wendet also 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 wesentliche **Problem** ist nun, wie der Parser entscheidet, welche Aktion (*shift* oder *reduce*) im nächsten Schritt angewendet wird.

Lösung: Lese den gesamten Stackinhalt mit einem endlichen Automaten und entscheide auf Grund des Zustandes des Automaten am Ende des Lesens und der nächsten k Eingabezeichen, welche Aktion ausgeführt wird. Um zu vermeiden, dass wir bei jedem Token den gesamten Stack lesen, speichern wir die Automatenzustände auch auf dem Stack, so dass wir nur das oberste Element lesen müssen. Somit hat der Stack die Form

$$S_0\alpha_1 S_1\alpha_2 S_2 \ldots S_{n-1}\alpha_n S_n$$

mit $\alpha_i \in N \cup T$ und S_i sind Zustände des Automaten (eigentlich sind die α_i nun überflüssig, aber wir nehmen sie zur besseren Übersicht auf).

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

- $s\,n$ – *shift* und gehe in den Zustand n
- $g\,n$ – gehe in den 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: Wir betrachten eine einfache Grammatik für „Anweisungssequenzen“ (wobei das neue Startsymbol S', für das es nur eine Regel gibt, auf das alte Startsymbol und den End-of-file-Marker \$ verweist):

(0)	S'	$\to$	$S\$$	(3)	L	$\to$	S
(1)	S	$\to$	(L)	(4)	L	$\to$	L, S
(2)	S	$\to$	x				

Beachte: Die Grammatik ist linksrekursiv (wegen Regel 4) und damit nicht LL(k)!

Die LR-Parsing-Tabelle hat die Form Zustand $\times (N \cup T \cup \{\$\}) \rightarrow$ Aktion:

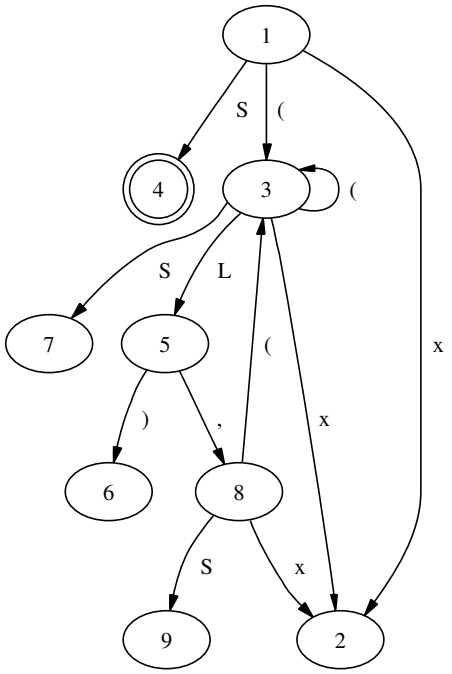
	shift/reduce					*goto*	
	(	)	x	,	\$	S	L
1	$s3$		$s2$			$g4$	
2	$r2$	$r2$	$r2$	$r2$	$r2$		
3	$s3$		$s2$			$g7$	$g5$
4					a		
5		$s6$		$s8$			
6	$r1$	$r1$	$r1$	$r1$	$r1$		
7	$r3$	$r3$	$r3$	$r3$	$r3$		
8	$s3$		$s2$			$g9$	
9	$r4$	$r4$	$r4$	$r4$	$r4$		

Der linke Teil der Tabelle heißt auch **shift/reduce-Tabelle** und der rechte Teil auch **goto-Tabelle**. Der Startzustand entspricht dem ersten Stackelement und ist der Zustand 1.

Aktionen bei *reduce*:

- Lösche oberste Stackelemente (entsprechend der rechten Regelseite)
- Lege linkes Nichtterminalsymbol auf den Stack
- Berechne neuen Zustand aus oberstem Zustand (nach dem Entfernen) mit der goto-Tabelle

Die Tabelle entspricht dabei dem folgenden Automaten:



Beachte: Zustände ohne Übergänge (Senken) sind entweder ein Endzustand (4) oder *reduce*-Zustände (2,6,7,9).

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 hierzu 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 $k = 0$.

Aus diesem Grund betrachten wir zunächst die Konstruktion von LR(0)-Parsern.

4.3.2. LR(0)-Parser

Beobachtung im obigen Beispiel: Stack und Eingabe entspricht immer einer Satzform der Rechtsableitung.

Idee: Jeder Zustand des Automaten enthält Regeln mit einer Markierung, wie weit schon gelesen wurde.

Produktionen mit solchen Positionsmarkierungen heißen **LR(0)-Elemente** (*LR(0) 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 (hier ist „.“ ein Symbol, das nicht in der Grammatik vorkommt).

Beachte: $\beta = \varepsilon$ und $\gamma = \varepsilon$ sind zulässig. Somit gehören z.B. zur Regel $S \to (L)$ die vier LR(0)-Elemente

$S \to .(L)$

$S \to (.L)$

$S \to (L.)$

$S \to (L).$

Die Automatenzustände sind nun Mengen von LR(0)-Elementen, die bestimmte Ableitungssituationen 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. Wir nehmen daher alle S-Regeln zum Zustand hinzu, d.h. der Startzustand (1) entspricht daher der Menge von LR(0)-Elementen

$$\{\ S' \to .S\$,\quad S \to .(L),\quad S \to .x\ \}$$

Dies führt uns zu folgender Definition:

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 $B \to \gamma$ eine Produktion, dann ist auch $B \to .\gamma \in \text{closure}(M)$
2. Ein **LR(0)-Zustand** ist eine abgeschlossene Menge von LR(0)-Elementen.

Die Berechnung von Abschlüssen ist trivial mittels Iteration.

Der Startzustand des Automaten ist $\text{closure}(\{S' \to .S\$\})$.

Die Berechnung der Zustandsübergänge erfolgt nun durch Verschieben der Markierung um ein Symbol nach rechts. Der Übergang von Zustand (1) unter dem Symbol x ist der Abschluss von $S \to x.$, was den Zustand (2) ergibt:

$$\{\ S \to x.\ \}$$

Der Zustand (3) entsteht durch Übergang von (1) unter ‘(’:

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

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

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

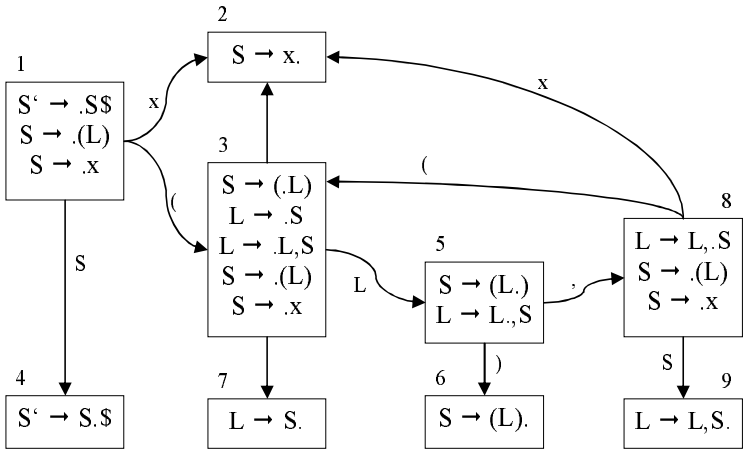
Daraus lässt sich nun der LR(0)-Automat konstruieren. Hierbei erhält man die *reduce*-Aktionen aus LR(0)-Zuständen, die abgearbeiteten Produktionen entsprechen, d.h. Elemente der Form $A \to \beta.$ enthalten.

Berechnung des LR(0)-Automaten

(mit Zuständen/Knoten T und Übergängen/beschrifteten Kanten E)

$T := \{\text{closure}(\{S' \to .S\$\})\};$
$E := \varnothing$
`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`

Zu beachten ist, dass das Symbol \$ nicht zur Grammatik gehört, d.h. es findet keine Berechnung von goto$(M, \$)$ statt, sondern dies resultiert in einer accept-Aktion (s.u.). Somit wird für die obige Grammatik der folgende Automat berechnet:



Berechnung der LR(0)-Parsing-Tabelle

Zunächst werden alle LR(0)-Zustände des Automaten durchnummeriert (z.B. mit 1 für den Startzustand). Im obigen Bild sind diese Nummern schon an die Zustände geschrieben. Stelle dann die Parsing-Tabelle auf, die den Typ

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

hat, indem die Tabelle nach folgenden Regeln gefüllt wird:

- $i \xrightarrow{X} j$ mit $X \in T$: Dies entspricht einer *shift*-Aktion: Trage Aktion „$s\,j$“ an der Stelle (i, X) ein.
- $i \xrightarrow{X} j$ mit $X \in N$: Dies entspricht einer *goto*-Aktion: Trage Aktion „$g\,j$“ an der Stelle (i, X) ein.

- Falls ein Zustand i ein Element $A \to \beta.$ enthält (wobei $A \to \beta$ die n-te Produktion sei): Dies entspricht einer *reduce*-Aktion: Trage „$r\,n$“ an allen Stellen (i, X) mit $X \in T$ ein.
- Falls ein Zustand i ein Element $S' \to S.\$$ enthält: Dies entspricht der *accept*-Aktion: Trage „a“ an der Stelle $(i, \$)$ ein.

Falls eine solche Tabelle keine mehrfachen Einträge enthält, dann heißt die Grammatik eine **LR(0)-Grammatik**.

Für unseren konstruierten Automaten erhalten wir damit genau unsere schon bekannte LR(0)-Parsing-Tabelle:

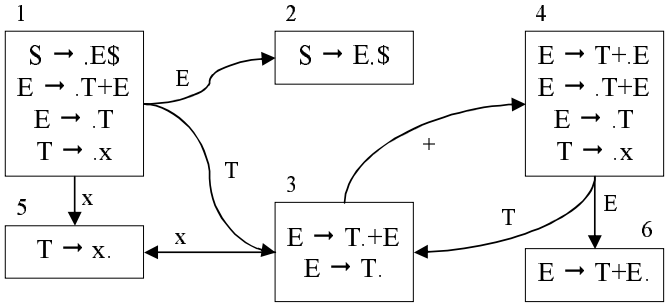
	shift/reduce					*goto*	
	(	)	x	,	\$	S	L
1	$s3$		$s2$			$g4$	
2	$r2$	$r2$	$r2$	$r2$	$r2$		
3	$s3$		$s2$			$g7$	$g5$
4					a		
5		$s6$		$s8$			
6	$r1$	$r1$	$r1$	$r1$	$r1$		
7	$r3$	$r3$	$r3$	$r3$	$r3$		
8	$s3$		$s2$			$g9$	
9	$r4$	$r4$	$r4$	$r4$	$r4$		

4.3.3. SLR(1)-Parser

Bei LR(0) benutzt man keine Vorausschau bei der Konstruktion des Automaten, d.h. die *reduce*-Einträge in der Tabelle sind unabhängig vom nächsten Eingabesymbol. Dadurch entstehen Konflikte, die manchmal durch eine genauere Vorausschau vermeidbar wären. Wir betrachten als Beispiel die 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 sieht wie folgt aus:



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 des Automaten erfolgt wie bei LR(0), jedoch werden *reduce*-Einträge nur erzeugt, falls die nächsten k Eingabezeichen hinter dem reduzierten Nichtterminal tatsächlich vorkommen können; hierzu können die FOLLOW-Mengen verwendet werden.

In unserem Beispiel verwenden wir $k = 1$. Die **SLR(1)-Parsing-Tabelle** ist dann wie eine LR(0)-Parsing-Tabelle konstruiert, aber mit der folgenden Änderung:

- Falls ein Zustand i ein Element $A \to \beta.$ enthält (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)

4.3.4. LR(1)-Parser

Ein LR(0)-Zustand enthält eventuell Elemente (und damit auch Konflikte), die in realen Ableitungen nicht vorkommen können. Um dieses Problem zu vermeiden, verfeinern wir

die Zustände durch Hinzunahme der nächsten k Eingabesymbole 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 statt mit LR(0)-Elementen. Zu diesem Zweck müssen die Definitionen von closure und goto auf LR(1)-Elemente angepasst werden.

DEFINITION: Sei M eine Menge von LR(1)-Elementen. Dann ist der *Abschluss* (*closure*) von M die kleinste Menge closure(M) 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 also die Berücksichtigung möglicher Eingabesymbole, die hinter der rechten Seite γ einer Produktion folgen können, was auf Grund des Kontextes nur FIRST(βx) sein kann.

Die goto-Funktion ist dagegen genauso wie im LR(0)-Fall definiert:

DEFINITION: Sei M eine Menge von LR(1)-Elementen und $X \in N \cup T$. Dann ist

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

Der Startzustand des LR(1)-Automaten ist wie üblich definiert als

$$\text{closure}(\{(S' \to .S\$, ?)\})$$

Hier ist ? beliebig, 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:

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

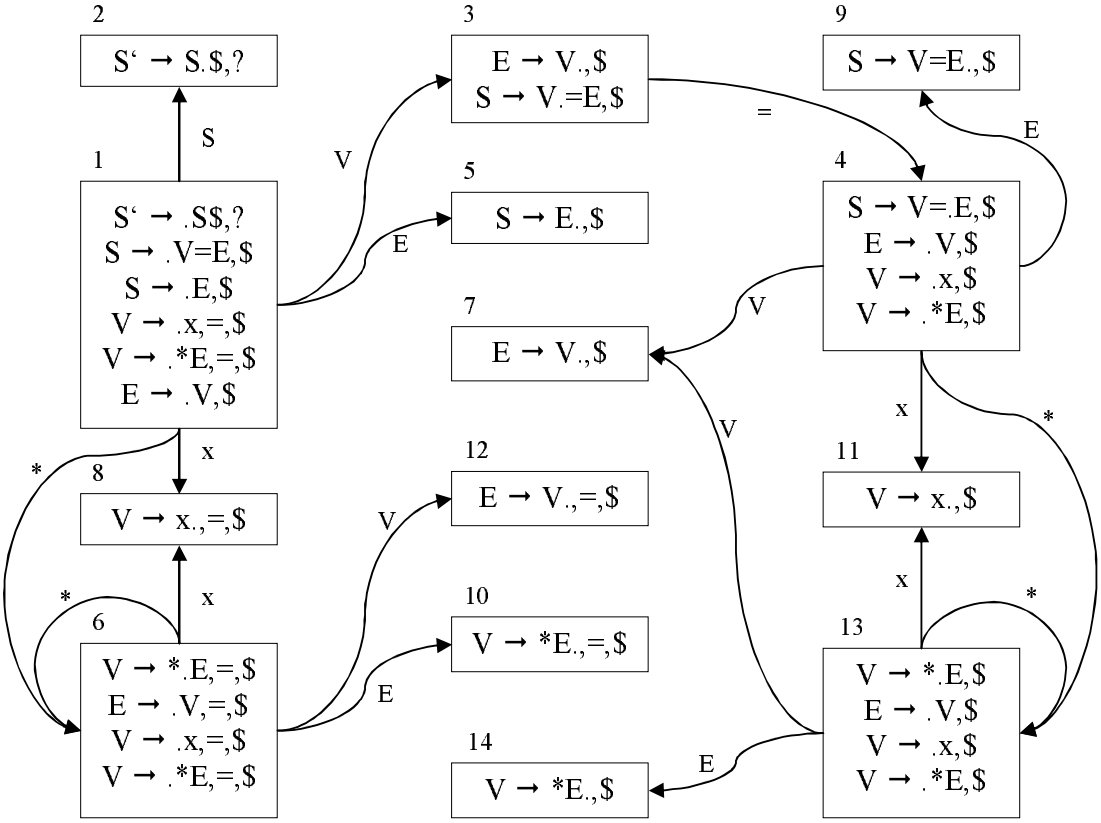
Es ergeben sich hieraus nun präzisere *reduce*-Aktionen, d.h. weniger *shift/reduce*- und *reduce/reduce*-Konflikte. Eine Grammatik wird als **LR(1)-Grammatik** bezeichnet, wenn die so konstruierte LR(1)-Tabelle konfliktfrei ist.

Beispiel: Betrachten wir eine Grammatik für Ausdrücke und Dereferenzierungen wie in C:

(0)	S'	$\to$	$S\$$	(3)	E	$\to$	V
(1)	S	$\to$	$V = E$	(4)	V	$\to$	x
(2)	S	$\to$	E	(5)	V	$\to$	$*E$

Diese Grammatik ist nicht SLR(1) ($\rightsquigarrow$ Übung).

Der LR(1)-Automat sieht wie folgt aus:[1]



[1] Zur Vereinfachung werden LR(1)-Elemente mit gleichem Kern zusammengefasst!

Daraus erhalten wir die folgende LR(1)-Parsingtabelle (analog zu LR(0), nur bei *reduce*-Aktionen mit 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	$s8$	$s6$				$g10$	$g12$
7				$r3$			
8			$r4$	$r4$			
9				$r1$			
10			$r5$	$r5$			
11				$r4$			
12			$r3$	$r3$			
13	$s11$	$s13$				$g14$	$g7$
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. Die Methode hierzu, die nachfolgend skizziert wird, heißt **Look-Ahead-LR(1)** (**LALR(1)**).

4.3.5. LALR(1)-Parser

Idee: Verschmelze Zustände, deren Elemente sich nur in den Vorausschausymbolen, aber nicht im LR(0)-Kern unterscheiden.

Im obigen Beispiel haben die 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)-Grammatik** genau dann, wenn die so konstruierte 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.

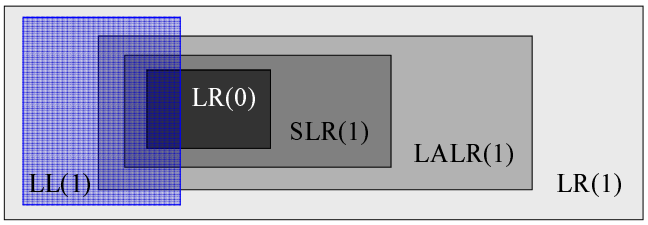
4.4. Klassifikation der Grammatiken und Sprachen

Um einen Eindruck der Mächtigkeit der verschiedenen Methoden zu erhalten, geben 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:



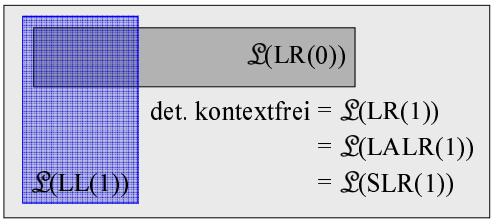
Interessant ist auch die folgende Aussage zur Mächtigkeit der LR(1)-Klasse:

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.

Somit sind also LR(1)-Grammatiken für die Syntaxanalyse im Prinzip ausreichend. Wenn wir mit mit $\mathcal{L}(L)$ die Menge aller Sprachen, die durch die Grammatikklasse L erzeugt werden können, bezeichnen, dann gelten die folgenden Beziehungen, die im nachfolgenden Bild visualisiert sind:

- $\mathcal{L}(LL(1)) \not\subset \mathcal{L}(LR(0))$
- $\mathcal{L}(LR(0)) \not\subset \mathcal{L}(LL(1))$
- $\mathcal{L}(LR(0)) \subsetneq \mathcal{L}(LR(1))$
- $\mathcal{L}(SLR(1)) = \mathcal{L}(LALR(1)) = \mathcal{L}(LR(1))$



4.5. Parser-Generatoren

Die Konstruktion von Parsingtabellen ist einfach, aber aufwändig – d.h. gut automatisierbar. Aus diesem Grund bietet sich die Verwendung von **Parser-Generatoren** an, die als Eingabe eine Grammatik erhalten (und auch „semantische Aktionen“, später) und als Ausgabe ein Parser-Programm liefern.

Das Vorgehen eines Generators ist z.B. für LALR(1): Berechne zunächst die Zustände (entweder LR(1) + Verschmelzung oder auch durch direkte Berechnung, was effizienter, aber konzeptuell aufwändiger 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 LL(k) als auch für 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). Die Eingabe ist hierbei eine Grammatikbeschreibung der Form:

```
<Modulkopf (optional)>
<Parser-Deklarationen>
%%
<Grammatik-Regeln>
<Modulende (optional)>
```

Hierbei sind Modulkopf und -ende optionaler Haskell-Code, welcher um den generierten Parser gesetzt wird. Dieser kann z.B. den Code für einen Scanner und für semantische Aktionen (s.u.) enthalten. Der Modulkopf ist wie folgt aufgebaut:

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

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

```
A: B_1 ... B_n { <Aktion> }
 | C_1 ... C_m { <Aktion> }
 ⋮
```

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

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

```
%name calc                -- name of created function
%tokentype { Token }      -- type of accepted tokens
                          -- Thus, 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
}
```

Hierbei definiert `%name` den Namen der generierten Parserfunktion. `%tokentype` definiert den Haskell-Datentyp der Token.

Durch `%token` werden alle Terminalsymbole, die in der Grammatik benutzt werden, festgelegt. Dabei steht in den geschweiften Klammern ein 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 $$ }`“ benutzt.

Weiterhin muss immer eine Funktion `happyError` mit dem entsprechenden Tokentyp definiert sein, die im Falle eines Syntaxfehlers aufgerufen wird.

Nach Übersetzen mit Happy erhält man Haskell-Programm `exp.hs`, welches eine Funktion `calc` definiert, die den Parser implementiert.

Konflikte

Bei der Konstruktion von shift/reduce-Parsern können folgende Konflikte auftreten, die nach einer festen Regel werden, um einen lauffähigen Parser zu erhalten (dies muss allerdings nicht die beabsichtigte Lösung sein!):

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

Auch wenn man durch diese Lösungsstrategie einen lauffähigen Parser erhält, sollte man Konflikte durch Änderung der Grammatik explizit eliminieren!

Beispiel: Betrachten wir z.B. die mehrdeutige(!) 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 des Satzes

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

Üblich ist, dass ein `else` zum nächstmöglichen `then` gehört. Die LR-Parsing-Tabelle für diese Grammatik enthält jedoch zunächst einen *shift-reduce*-Konflikt:

$$\begin{array}{lll} S \to \texttt{if}\ E\ \texttt{then}\ S.\ \texttt{else}\ S & \rightsquigarrow & \textit{shift} \\ S \to \texttt{if}\ E\ \texttt{then}\ S. & \rightsquigarrow & \textit{reduce} \end{array}$$

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

$$\begin{array}{rcl} S & \to & M \\ S & \to & U \\ M & \to & \texttt{if}\ E\ \texttt{then}\ M\ \texttt{else}\ M \\ M & \to & \texttt{assign} \\ U & \to & \texttt{if}\ E\ \texttt{then}\ S \\ U & \to & \texttt{if}\ E\ \texttt{then}\ M\ \texttt{else}\ U \end{array}$$

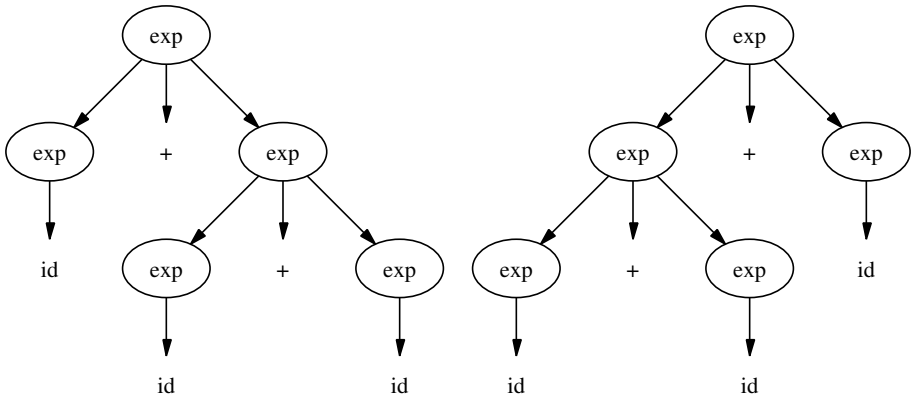
Beispiel: Betrachte eine mehrdeutige Grammatik für Ausdrücke:

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

Nun kann es zu einem Konflikt kommen:

$$\begin{array}{lll} \texttt{exp : exp . '+' exp} & \rightsquigarrow & \textit{shift} \\ \texttt{exp : exp '+' exp .} & \rightsquigarrow & \textit{reduce} \end{array}$$

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 wir modifizieren die Grammatik oder wir fügen eine explizite Parserdeklaration hinzu:

```
%left '+'
```

Dadurch wird spezifiziert, dass das Terminalsymbol + linksassoziativ ist, d.h. der Konflikt wird durch ein *reduce* vor *shift* gelöst.

Analog kann ein Operator zum Potenzieren wie mathematisch üblich als rechtsassoziativ spezifiziert werden, wodurch der Konflikt durch ein *shift* vor *reduce* gelöst wird:

```
%right '^'
```

Desweiteren kann man Operatoren als nicht-assoziativ deklarieren, wodurch ein Fehler ausgegeben wird, falls der Fall auftritt (d.h. shift/reduce-Einträge werden durch error-Aktionen ersetzt):

```
%nonassoc '='
```

Hierdurch wäre `id=id=id` ein unzulässiger Satz.

Weitere Konflikte (Mehrdeutigkeiten) können durch Prioritäten in Ausdrücken entstehen. Wir betrachen beispielsweise folgende Grammatik:

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

Dann ist der Ausdruck `id + id * id` mehrdeutig. In der üblichen mathematischen Notation soll * stärker als + binden. Dies kann man durch eine Deklarationssequenz ausdrücken, wobei die späteren Deklarationen stärker binden:

```
%left '+'
%left '*'
```

Mittels expliziter Parserdeklarationen hätte also die folgende Spezifikation keine Konflikte und entspricht den üblichen Bindungsregeln:

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

5. Parsing-Aktionen und Abstrakte Syntax

Bisher haben wir Methoden kennengelernt, um die Syntax von Programmen zu erkennen. Das Ziel ist allerdings die Codeerzeugung entsprechend der erkannten Syntax. Der erste Schritt hin zu diesem Ziel ist das Einfügen semantischer Aktionen in die Syntaxanalyse.

5.1. Semantische Aktionen

Idee ist, dass jedes Grammatiksymbol (Terminal- und Nichtterminalsymbol) eine *Bedeutung* trägt. Bei den Terminalsymbolen ist dies der konkrete Wert der Symbolklasse (`Id`: String, `Num`: Zahlenwert) und bei Nichtterminalsymbolen ist dies die Bedeutung des abgeleiteten Sprachkonstrukts – z.B. bei Ausdrücken der Wert, bei Anweisungen 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 Nichtterminalzeichens berechnet.

Die konkrete Realisierung semantischer Aktionen ist abhängig vom Syntaxanalyseprinzip. Wir werden daher verschiedene Realisierungen betrachten, bevor wir dann später auf eine eher unabhängige Methode eingehen.

5.1.1. Recursive Descent Parser

In einen rekursiven Abstiegsparser kann man semantische Aktionen wie folgt integrieren:

- Erweitere die Terminalzeichen um eine Komponente, die die Bedeutung dieses Symbols repräsentiert, z.B. bei Ausdrücken:

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

- Nichtterminale werden ja durch Funktionen repräsentiert, so dass der Bedeutungswert dem Ergebnis der Funktion entspricht. Somit werden Nichtterminalfunktionen implementiert durch Aufruf der Symbole der rechten Seite und eine Berechnung der Bedeutung dieses Nichterminalsymbols.

Beispiel: Berechnung des Werts von Binärzahlen:

Wir benutzen die folgende Grammatik:

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

Diese ist linksrekursiv, daher wenden wir die übliche Transformation an:

$$digits \rightarrow 0\ digits \mid 1\ digits \mid 0 \mid 1$$

Der semantische Wert soll nun der zugehörige Integer-Wert sein. Die Berechnung dieses Wertes können wir in den RD-Parser integrieren (der Einfachheit halber lassen wir hier auch leere Bitfolgen zu). Ein Problem ist dabei, dass wir die Stellung der Bits zur Wertberechnung benötigen. Zum Beispiel ergibt sich der semantische Wert von „1$digits$“ aus der Summe des semantischen Wert von $digits$ und 2^l, wobei l die Länge von $digits$ ist. Aus diesem Grund berechnen wird zwei semantische Werte:

1. Die Länge der Bitfolge
2. Den Wert der Bitfolge

Am Ende wird aber nur der Wert zurückgegeben:

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

-- Der semantische Wert ist ein Paar:
-- (Anzahl digits, Wert der schon gelesenen digits)

digits :: [Token] → ([Token],(Int,Int))
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,(2^l)+v))
  EOF  → (ts,(0,0))

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

Zu beachten ist, dass die Längenberechnung nur notwendig ist wegen der Umstellung der Regeln in eine rechtsrekursive Form. Bei einer linksrekursiven Regel ist die Berechnung einfacher, wie wir nun sehen werden.

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, welche in der Implementierung einfach auf dem Stack gespeichert werden kann, d.h. der Stack enthält nicht nur die Symbole (bzw. Automatenzustände), sondern auch die semantische Werte dieser Symbole. Die semantischen Aktionen werden am Ende jeder Regel angegeben. Beispielsweise kann unsere Binärzahlberechnung in Happy wie folgt definiert werden:

```
%name parser
%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 einer **Reduktion** ausgeführt; dann sind alle Symbole der rechten Seite auf dem Stack, so dass deren semantische Werte zur Verfügung stehen und man mit der Notation `$`i auf den semantischen Wert des i-ten Symbols der rechten Seite zugreifen kann. Somit wird der Wert des reduzierten Nichtterminals in den Klammern `{...}` definiert.

Die *reduce*-Aktion einer Regel mit semantischen Aktionen wird nun wie folgt durchgeführt:

1. Ersetze im Ausdruck der semantischen Aktion alle Symbole `$`i durch die auf dem Stack entsprechend gespeicherten Werte.
2. Reduziere den Stack (wie früher).
3. Werte die semantische Aktion aus und speichere das Ergebnis beim obersten Stackelement.

Beispiel: Berechnung der Werte von Ausdrücken DeskCalculator:
Die Ausdrücke entsprechen dabei der schon früher angegebenen Grammatik; als semantischer Wert wird nun der Integer-Wert des Ausdrucks benutzt.[1]

```
{
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 }
```

[1] `$$` bedeutet, dass der semantische Wert von `id` nicht das Token selbst ist, sondern das Argument, auf das der `ID`-Konstruktor bei der Konstruktion durch den Scanner angewendet wurde.

```
term   : term '*' factor { $1 * $3 }
       | factor          { $1 }

factor : num             { $1 }
       | '(' exp ')'     { $2 }
```

Beachte: Die Umsetzung mit einem RD-Parser wäre schwieriger wegen der Transformation der Regeln z.B. zu $T \to FT'$ und $T' \to *FT'$ und $T' \to$. Weil innerhalb der T'-Regeln das linke Argument fehlt, was aber zur Wertberechnung notwendig ist, müssen wir dessen semantischen Wert als erstes Argument in den T'-Parser hineinreichen:

```
parse_T (t:ts) = let (ts1, fRes) = f (t:ts) in
                 case t of
                  NUM _   → parse_T' fRes ts1
                  LPAREN  → parse_T' fRes ts1
                  _       → error ...

parse_T' lval (t:ts) = case t of
                   PLUS    → (t:ts, lval)
                   TIMES   → let (ts1, fRes) = f ts
                              in parse_T' (lval * fRes) ts1
                   RPAREN  → (t:ts, lval)
                   EOF     → (t:ts, lval)
                   _       → error ...
```

Wie man sieht, ist die konkrete Syntax häufig (insbesondere LL(1)) nicht direkt geeignet für semantische Berechnungen. Aus diesem Grund benutzt man eine **abstrakte Syntax** zur weiteren semantischen Berechnung (siehe unten).

Die semantischen Aktionen reichen im Prinzip schon für alle Übersetzungsaufgaben aus, daher bezeichnet man Yacc, Happy etc. auch als Compilergeneratoren. In der Praxis sind jedoch bei größeren Projekten die Übersetzungsaufgaben zu komplex, um alles in der syntaktischen Analyse zu integrieren, so dass man eine bessere Struktur erhält, wenn man die syntaktische und semantische Analyse mittels einer abtrakten Syntax trennt.

Zunächst aber demonstrieren wir die Mächtigkeit der semantischen Aktionen am Beispiel eines einfachen Interpreters, beispielsweise für Simple-Programme.

Die Hauptdatenstruktur eines Interpreters ist eine **Umgebung**, d.h. eine Abbildung vom Typ

```
type Env = String → Int
```

die Bezeichnern aktuelle Werte zuordnet. Notwendig ist eine Möglichkeit zur Änderung eines Eintrags, die wir wie folgt definieren können:

```
update :: Env → String → Int → Env
```

```
update e n v = \m → if m == n then v
                                 else e m
```

Die „leere“ Umgebung liefert immer einen Fehler bei einem Zugriff:

```
emptyEnv :: Env
emptyEnv n = error ("Access to undefined variable " ++ n)
```

Der semantische Wert eines Ausdrucks ist dann eine Abbildung `Env → Int`, der semantische Wert einer Zuweisung ist dagegen eine Abbildung `Env → Env`. Insgesamt erhalten wir mit diesen Überlegungen den folgenden Interpreter:

```
{
type Env = String → Int

update :: Env → String → Int → Env
update e n v = \m → if m == n then v
                                 else e m

emptyEnv :: Env
emptyEnv n = error ("Access to undefined variable " ++ n)
}

%name interpreter
%tokentype { Token     }
%token ';' { SEMICOLON }
       '=' { ASSIGN    }
       '+' { PLUS      }
       '*' { TIMES     }
       '(' { LPAREN    }
       ')' { RPAREN    }
       id  { ID  $$ }
       num { Num $$ }
%%
progrm : stmts '(' exp ')'    { $3 ($1 emptyEnv) }

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). Dies ist eine Standardtechnik der funktionalen Programmierung.
- Für die Beschreibung der Semantik von Programmiersprachen gibt es auch die sog. denotationelle Semantik, die auf der Idee beruht, jedem Syntaxkonstrukt eine entsprechende semantische Funktion zuzuordnen. Dieses Beispiel zeigt, dass wir also die denotationelle Semantik mittels semantischer Aktionen in Haskell direkt implementieren können.
- Was noch fehlt, sind Fallunterscheidungen und Schleifen ($\rightsquigarrow$ Übung). Zudem sind noch Ausgaben nötig. Hierzu kann man z.B. bei der Semantik einen Ausgabestring mitführen, so dass der Wert einer Anweisung eine Transformation

  (alte Umgebung, bisherige Ausgabe) $\rightarrow$ (neue Umgebung, neue Ausgabe)

  ist. Eine Zuweisung verändert nun die erste Komponente, während eine `print`-Anweisung die zweite Komponente modifiziert.

Der neue Interpreter mit der Berechnung der Ausgabe sieht dann wie folgt aus:

```
...
%%
program : stmts              { snd ($1 (emptyEnv,"")) }

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) ++ "\n") }

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 }
```

5.2. Attributierte Grammatiken

Semantische Aktionen ordnen jedem Nichtterminalsymbol einen Wert (auch **Attribut** genannt) zu, welcher aus den Attributen der Symbole der rechten Regelseite berechnet wird. Diese Berechnung entsprecht einem **Datenfluss von unten nach oben** im Ableitungsbaum. Diese Berechnungsrichtung ist jedoch für manche Aufgaben nicht ausreichend, z.B. bei:

- Bestimmung der Schachtelungstiefe bei Blöcken: Datenfluss von oben nach unten
- Aufbau einer Symboltabelle aller Deklarationen: Datenfluss: oben → unten → links → rechts → oben

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

DEFINITION: Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik. Eine **attributierte Grammatik** ordnet jedem Symbol $X \in N \cup T$ eine Menge von **synthetisierten Attributen** $\mathrm{Syn}(X)$ und eine Menge von **vererbten Attributen** $\mathrm{Inh}(X)$ zu (wobei $\mathrm{Inh}(a) = \varnothing$ für alle $a \in T$ gilt) und enthält für jede Regel $A_0 \to A_1 \dots A_n \in P$ eine Menge von **semantischen Regeln** (**Attributgleichungen**) der Form

$$i.a = f(j_1.a_1, \dots, j_k.a_k)$$

mit

- $a \in \mathrm{Syn}(A_i)$ falls $i = 0$
- $a \in \mathrm{Inh}(A_i)$ falls $i > 0$
- für alle $1 \leq l \leq k$ gilt:
  - $a_l \in \mathrm{Syn}(A_{j_l})$ falls $j_l > 0$
  - $a_l \in \mathrm{Inh}(A_{j_l})$ falls $j_l = 0$ gilt

Intuition:

- $j.a$ bezeichnet das Attribut a des Symbols A_j
- Die linke Gleichungsseite darf nur ein synthetisiertes Attribut der linken Seite der Produktion oder ein vererbtes Attribut der rechten Seite sein.

- In der rechten Gleichungsseite dürfen umgekehrt nur vererbte Attribute der linken Seite oder synthetisierte Attribute der rechten Seite benutzt werden.

Durch diese Einschränkungen vermeidet man zyklische Abhängigkeiten innerhalb einer Regel, z.B. ist die Attributgleichung

$$0.a = 0.a + 0.a$$

unzulässig (man beachte, dass hierdurch Zyklen in der gesamten Attributierung nicht ausgeschlossen werden).

Eine **Attributierung** eines Ableitungsbaums ist nun eine Zuordnung von Knoten zu Attributwerten (entsprechend den Anforderungen der attributierten Grammatik), so dass bei jeder Produktion im Ableitungsbaum alle Attributgleichungen erfüllt sind.

Beispiel: Berechnung der größten Schachtelungstiefe in einer blockstrukturierten Sprache, die durch folgende vereinfachte (mehrdeutige!) Grammatik definiert ist:

$$\begin{array}{llll} S' \to S & S \to S\,S & P \to \texttt{assign} \\ & S \to P & P \to \texttt{begin}\ S\ \texttt{end} \end{array}$$

Die Attribute sind hierbei zum einen die Tiefe eines Blockes, d.h.

$$\mathrm{Inh}(S) = \mathrm{Inh}(P) = \{d\} \qquad \text{(depth of block)}$$

zum anderen die maximale Tiefe aller enthaltenen Blöcke, d.h.

$$\mathrm{Syn}(S) = \mathrm{Syn}(P) = \mathrm{Syn}(S') = \{m\} \qquad \text{(maximum depth)}$$

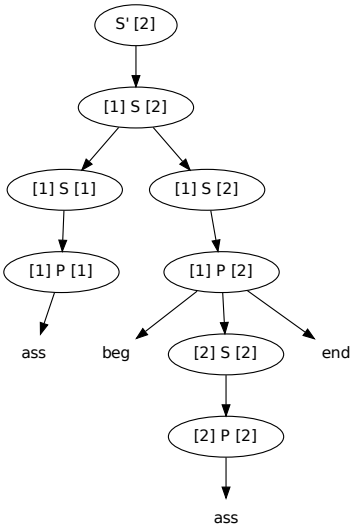
Die Attributgleichungen für die einzelnen Regeln sind:

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

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

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

Eine Attributierung notieren wir direkt an die Knoten im Ableitungsbaum, wobei links der Wert des Attributs d und rechts der Wert des Attributs m steht:



Mittels attributierter Grammatiken lassen sich fast alle Übersetzungsprobleme als Berechnung von Informationen im Syntaxbaum formal definieren, z.B.

- den Aufbau und das Durchreichen einer Symboltabelle (der Aufbau entspricht einem vererbten Symboltabellenattribut und das Durchreichen entspricht einem synthetisierten Attribut),
- Codegenerierung: Zusammenbau von Codestücken,
- Aufruftiefe: s.o.

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

Problem: Wie berechnet man Attributierungen? Hierzu sind zunächst weitere Einschränkungen wie die Zyklenfreiheit notwendig, andererseits muss man eine Berechnungsstrategie finden. Ideal wäre dabei eine Attributierung schon während der Syntaxanalyse, was für bestimmte Klassen attributierter Grammatiken möglich ist.

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

Dies lässt sich so auffassen, als ob jedes Symbol ein Attribut (bzw. ein Tupel von Attributen) hat, das bei Anwendung der Syntaxregel berechnet werden kann. Somit entspricht eine S-Attributierung der Berechnung semantischer Aktionen, so dass dieses leicht implementierbar sowohl in Recursive-Descent-Parsern als auch in *shift-reduce*-Parsern ist. 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 $A_1, \ldots, A_{j-1}$ 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: $i < j$.

Somit bestehen Abhängigkeiten im Baum nur von links nach rechts, so dass die Berechnung in einem top-down-links-rechts-Durchlauf möglich ist. Damit ist die Implementierung in RD-Parsern sehr 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, so dass wir die Attributierung wie folgt berechnen können:

```
data Token = Assign | Begin | End | Seq
 deriving (Eq,Show)

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 Attributberechnung geschrieben werden müssten und damit zu stark mit der Syntaxanalyse verwoben sind. Eine Lösung wäre die Verwendung mehrere Attributgrammatiken oder Funktionen auf Syntaxbäumen. Dazu kann die abstrakte Syntax sehr hilfreich sein.

5.3. Abstrakte Syntax

Ein **Nachteil** einer konkreten Syntax ist, dass diese häufig sehr komplex und ungeeignet für Übersetzungsaufgaben ist. Wir haben z.B. schon Ableitungsbäume für Ausdrücke bzgl. einer LR(1)- oder LL(1)-Grammatik gesehen, die selbst für einfache Ausdrücke komplex werden können. Eine einfache und klare Grammatik für Ausdrücke wäre:

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

Ein Nachteil dieser Grammatik ist, dass sie **mehrdeutig** ist. Dies ist allerdings nur ein Problem für den Parser! Nach dem Parsing können wir diese Grammatik verwenden, wenn der Parser immer Terme/Ableitungsbäume dieser Struktur liefert. Eine solche einfache Grammatik vereinfacht alle weiteren Übersetzungsaufgaben.

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. Die abstrakte Syntax selbst kann aber mehrdeutig sein und sie verzichtet auf überflüssige syntaktische Elemente, die nur für eine eindeutige konkrete Syntax eingeführt werden.

Die **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“ zum Weiterreichen von semantischen Werten, z.B. $exp \rightarrow term$, $term \rightarrow factor$, ...

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

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

Beispiel: Eine abstrakte Syntax für Simple könnten wir durch die folgende kontextfreie (mehrdeutige!) Grammatik definieren:

$$\begin{aligned} Stm &\rightarrow Stm\texttt{;}Stm \mid \texttt{Id:=}Exp \mid \texttt{print(}Exps\texttt{)} \\ Exp &\rightarrow \texttt{Num} \mid \texttt{Id} \mid Exp\ BinOp\ Exp \mid \texttt{(}Stm\texttt{,}Exp\texttt{)} \\ Exps &\rightarrow Exp\texttt{,}Exps \mid Exp \\ BinOp &\rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \end{aligned}$$

Wir können diese abstrakte Syntax wie folgt als Datentyp in Haskell umsetzen (vgl. Abschnitt 1.3), wobei hier die Terminalsymbole wie `print` überflüssig sind; stattdessen

werden entsprechende Konstruktoren zur Kennzeichnung der jeweiligen Syntaxregelalternativen benutzt:

```
type Id = String

data BinOp = Plus | Minus | Times | Div
  deriving (Eq, Show)

data Stm = CompoundStm Stm Stm
         | AssignStm   Id  Exp
         | PrintStm    [Exp]
  deriving (Eq, Show)

data Exp = NumExp  Int
         | IdExp   Id
         | OpExp   Exp BinOp Exp
         | EseqExp Stm Exp
  deriving (Eq, Show)
```

Zur Erzeugung des abstrakten Syntaxbaums fügen wir einfach in den Happy-Parser entsprechende semantische Aktionen ein:

```
...
%%
program : stmts               { $1 }

stmts   : stm ';' stmts       { CompoundStm $1 $3 }
        | stm                 { $1                }

stm     : id ':=' exp         { AssignStm $1 $3 }
        | print '(' exp ')'   { PrintStm  $3    }

exps    : exp ',' exps        { $1 : $3 }
        | exp                 { [$1]    }

exp     : exp '+' term        { OpExp $1 Plus  $3 }
        | exp '-' term        { OpExp $1 Minus $3 }
        | term                { $1                }

term    : term '*' factor     { OpExp $1 Times $3 }
        | term '/' factor     { OpExp $1 Div   $3 }
        | factor              { $1 }

factor  : num                 { NumExp  $1    }
```

```
        | id                    { IdExp   $1    }
        | '(' Stm ',' Exp ')' { EseqExp $2 $4 }
```

Beispiel: Die Eingabe "`x:=2; print(3+x*5)`" führt zum Berechnungsergebnis

```
(CompoundStm
  (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 der bei der Fehlerausgabe (z.B. Typfehler) fehlende Bezug zur konkreten Syntax, insbesondere fehlen die Positionsangaben des Quelltextes!

  Lösung: Diese Informationen werden in den abstrakten Syntaxbaum eingefügt:

  1. Füge einen Typ der Positionsangaben hinzu, z.B.

     ```
     type Pos = (Int, Int)  -- Zeile, Spalte
     ```

  2. Erweitere die abstrakte Syntax, wo es eventuell notwendig ist:

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

  3. Initialisiere die 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. Typische Aufgaben der semantischen Analyse sind:

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

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

- Blöcke (Algol, Pascal, Modula, ...)
- Prozedur- und Funktionsrümpfe
- Methoden, Records, Klassen
- `let`, `letrec`, `where`, ...
- `structure`, `signature` (Standard ML)
- Module, Schnittstellen

Die Symboltabelle muss daher 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 für *id* mit *info*-Informationen (z.B. Typangabe) im aktuellen Scope (kann z.B. auch einen Fehler erzeugen, falls *id* schon vorhanden ist)

- *lookup(id)* sucht den *info*-Eintrag für `id` von innen nach außen (und erzeugt eventuell eine Fehlermeldung, falls der Eintrag nicht vorhanden ist)

Das **Problem** ist, dass die Suche schnell sein muss, damit Anwendungen von Bezeichnern schnell den Deklarationen zugeordnet werden können!

Implementierungsmöglichkeiten:

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

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

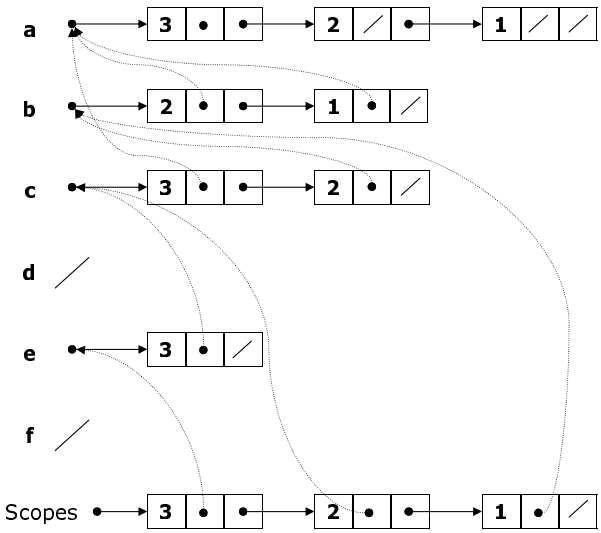
2. Kellerartig mit schnellerer Suche: wie oben, aber jeder Scope wird als Suchbaum verwaltet, d.h. wir erhalten eine logarithmische Suche in jedem Scope.

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; ist zu einem Bezeichner kein Eintrag vorhanden, so ist der Bezeichner nicht sichtbar.
   - Zusätzlich werden aller Bezeichner eines Scopes verkettet, um das effiziente Löschen eines Scopes zu ermöglichen.
   - 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 an der Stelle `(*)` sieht nun folgendermaßen aus:



Die Operationen darauf lassen sich dann wie folgt implementieren:

- *insert*: neuen Listenkopf als erstes Element in die Eintragsliste einfügen und verketten, mit aktuellem Scope-Ende verketten
- *lookup*: zugehörigen Listenkopf suchen (konstante Zeit!)
- *enter_scope*: neuen Scope-Kopf einfügen
- *exit_scope*: Lösche alle Listenköpfe in diesem Scope ausgehend vom Scope-Kopf

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. Codeerzeugung

Die Codeerzeugung ist in der Regel eine komplexe Aufgabe, insbesondere, wenn Optimierungen durchgeführt werden. Daher ist es sinnvoll, die Codeerzeugung auch zu modularisieren, z.B. in die folgenden Phasen zu zerlegen:

- Laufzeitspeicherorganisation planen
- Zwischencodeerzeugung
- Zielcodeauswahl
- Optimierungen
- Registerallokation

Diese Phasen werden wir im Folgenden genauer erläutern.

7.1. Laufzeitspeicherorganisation

Zur Laufzeit soll der Zugriff auf Variablen über ihre **Adresse im Zielprogramm** erfolgen. Zu diesem Zweck muss vor der Codeerzeugung geplant werden, wo die Objekte abgelegt werden und wie man auf die Objekte zugreift. Betrachten wir z.B. das folgende Beispielprogramm mit geschachtelten Prozeduren:

```
proc p
begin
   int x;

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

Bei `(*)` erfolgt ein Zugriff auf die Variablen `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 also den Speicherplatz von `x` zur Laufzeit?

Allgemeine Fragestellung:

Welche Werte müssen wo und wie lange gespeichert werden?

Eine generelle Antwort darauf gibt es nicht, denn dies hängt von der verwendeten Programmiersprache ab!

7.1.1. Speicherplatz für Variablen

Programmglobale Variablen haben dieselbe Lebensdauer wie das Programm, *prozedurlokale Variablen* die Lebensdauer der Prozedurabarbeitung. Ausnahmen von dieser allgemeinen Regel treten z.B. bei *Funktionen höherer Ordnung* auf. Betrachten wir hierzu folgendes Haskell-Programm

```
f x = let g y = x + y
      in  g

h = f 0

j = f 3
```

und die Abarbeitung von

```
> (h 2) + (j 5)
10
```

Die Funktion `f` wird hierbei zweimal aufgerufen und beendet, allerdings muss der Wert von `x` nach der jeweiligen Beendigung der Abarbeitung noch gespeichert werden!

Im Folgenden betrachten wir jedoch zunächst *keine Funktionen höherer Ordnung*, d.h. alle lokalen Variablen leben nur bis zum Prozedurende.

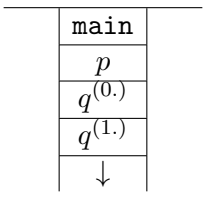
Damit können die Prozedur-Variablen kellerartig verwaltet werden:

- Beim Prozedureintritt wird auf dem Prozedurkeller Speicherplatz für lokale Variablen reserviert.
- Bei Prozedurende wird dieser Speicher wieder freigegeben.

Einfaches Beispiel als Skizze:

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

Prozedurkeller beim 1. rekursiven Aufruf von `q`:



Manche Programmiersprachen erlauben lokale Deklarationen in **Blöcken**. Die Lebensdauer lokaler Variablen in Blöcken ist dann die Dauer der Blockabarbeitung.

Die **Implementierung von Blöcken** könnte im ersten Ansatz wie bei Prozeduren erfolgen (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. Zu beachten ist dabei, dass der Speicherplatz bei geschachtelten Blöcken addiert wird, bei disjunkten Blöcken ist dagegen der Maximalwert der einzelnen Blöcke ausreichend:

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

Der 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), so dass deren gesamter Speicherplatz zur Übersetzungszeit geplant werden könnte.

Dynamische Datenstrukturen sind dadurch gekennzeichnet, dass die Größe erst zur Laufzeit bekannt wird. Beispielsweise werden bei *dynamischen Arrays* die genauen Indexgrenzen erst zur Laufzeit bekannt.

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

Zur **Realisierung** bieten sich zwei Alternativen an:

- entweder durch Festlegung des Speicherplatzes für `a` nach Eintritt in `p` auf dem Stack;

- oder durch Anforderung von Speicherplatz für `a` zur Laufzeit in einem separaten dynamischen Speicherbereich (**Heap**, siehe unten).

Bei **dynamischen Records** (Objekten) wird Speicherplatz durch explizite Operationen (`new`, `malloc`) bereitgestellt, die Lebensdauer ist dann häufig über das Prozedurende hinaus, d.h. eine Speicherung im Stack ist nicht möglich!

Daher muss ein neuer Speicherbereich für dynamische (nicht kellerartige) Speicherverwaltung verwendet werden: ein **Heap**. Eine Speicheranforderung geschieht explizit durch `new`/`malloc`. Zur Freigabe gibt es zwei Möglichkeiten:

Explizite Freigabe des Speicherplatzes (`dispose`/`free`). Dies birgt die Gefahr von **fehlerhaften Freigaben** („dangling pointers“).

Implizite Freigabe des Speicherplatzes durch **garbage collection** (sicherer, aber algorithmisch schwieriger und weniger effizient).

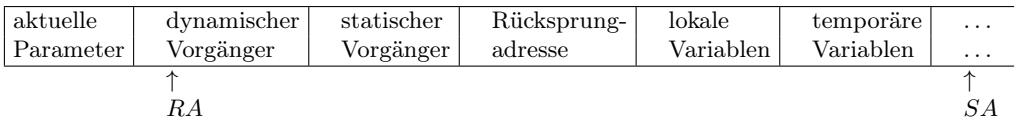
Typische Laufzeitspeicheraufteilung:

Der Code ist das übersetzte Programm, im statischen Speicher werden globale Variablen, Konstanten (Strings) u.ä. gespeichert (manchmal wird dies auch ersetzt durch den Speicher der 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.

7.1.2. Aufbau des Stack

Für jeden Prozeduraufruf liegt auf dem Stack ein Eintrag (**stack frame**, **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 den Aufbau eines Prozedurrahmens:



Hierbei gibt es zwei spezielle Register:

- `SA` ist die Adresse des nächsten freien Stackelements („top of stack“).
- `RA` ist die Rahmenadresse, d.h. die Basisadresse für den aktuellen Rahmen. Hierüber werden alle lokalen und temporären Variablen (positiver Offset) sowie die aktuellen Parameter (negativer Offset) adressiert.

Die Bedeutung der einzelnen Rahmenelemente ist:

- Die **aktuellen Parameter** werden bei Prozeduraufruf vom Aufrufer auf den Stack gelegt (vor einem Sprung zum Prozedurcode), hier können eventuell auch Ergebnisparameter (bzw. die Adressen der Ergebnisse) abgelegt werden. Bei einem Aufruf `p(0,1)` würden also die folgenden Aktionen durchgeführt werden:

  ```
  -- M bezeichnet den Speicher
  M[SA]     := 0;      -- 1. Parameter
  M[SA + 1] := 1;      -- 2. Parameter
  SA        := SA + 2; -- Naechste freie Stackadresse
  ```

- Der **dynamische Vorgänger** ist der Wert von `RA` der aufrufenden Prozedur. Dieser dient zur Wiederherstellung des Zustandes nach der Prozedurabarbeitung. Daher müssen die folgenden Aktionen bei Prozedureintritt durchgeführt werden:

  ```
  M[SA] := RA;     -- RA des dynamischen Vorgaengers speichern
  RA    := SA;     -- Neue Rahmenadresse festlegen
  SA    := SA + 1; -- Naechste freie Stackadresse
  ```

  Bei dem Austritt aus der Prozedur muss schließlich der Zustand vor der Prozedurabarbeitung wiederhergestellt werden:

  ```
  SA := RA - 2; -- Top of Stack, nach Prozeduraufuf mit 2 Parametern
  RA := M[RA];  -- RA des dynamischen Vorgaengers setzen
  ```

- Im **statischen Vorgänger** wird die `RA` des letzten Aufrufs der textuell umfassenden Prozedur gespeichert. Dieser ist notwendig zur Adressierung nicht-lokaler Variablen. Er ist aber nicht notwendig bei Programmiersprachen ohne textuell geschachtelte Prozeduren, z.B. in C. Weil dies etwas komplexer ist, erläutern wir die Verwaltung des statischen Vorgängers weiter unten.

- Die **Rücksprungadresse** ist die Codeadresse, bei der nach Prozedurende weitergemacht wird. Viele klassische Maschinenarchitekturen haben eine *call*-Instruktion, die automatisch die Rücksprungadresse auf den Stack speichert. Diese Technik kann jedoch wieder den Nachteil haben, dass bei jedem Prozeduraufruf immer ein Zugriff auf den Hauptspeicher nötig ist.

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

  – Speichere die Rückkehradresse in ein spezielles Register (`RET`).

  ```
  RET := <continue>; -- Ruecksprungadresse setzen
  ```

  – Springe zum Code der Prozedur.

```
JUMP <proc>; -- Sprung zu Code von proc
```

– Springe am Prozedurende einer Prozedur ohne weiteren Prozeduraufruf zu `RET`.

```
JUMP RET; -- Sprung zu Ruecksprungadresse
```

– Innerhalb einer Prozedur mit weiterem Prozeduraufruf: Speichere den Inhalt von `RET` im Prozedurrahmen und schreibe ihn nach Aufruf oder bei Prozedurende zurück.

```
M[RA + 2] := RET;       -- Speichern
...                     -- Weitere Prozeduraufrufe
RET       := M[RA + 2]; -- Zurueckschreiben
```

Damit läßt sich eine Zeitersparnis bei einfachen Prozeduren erreichen, bei komplexeren Prozeduren haben wir das gleiche Verhalten wir bei kellerbasierten Aufrufen.

- Anschließend werden **lokale Variablen** auf dem Stack gespeichert, deren Adresse relativ zur Rahmenadresse `RA` ist. Dies gilt ebenfalls für Variablen, die innerhalb eines Blocks deklariert wurden.
- Für komplizierte Berechnungen können **temporäre Variablen** notwendig sein, in denen Zwischenergebnisse gespeichert werden. Da diese Variablen nicht im Programmtext vorkommen benötigen diese auch keine feste Adresse und können dynamisch während der Prozedurabarbeitung angelegt werden.

Ein vollständiges Beispiel für einen Prozeduraufruf wird in der Übung besprochen. Zur Effizienzsteigerung wird zudem das folgende allgemeine Prinzip verfolgt:

> 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 teilweise 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.

7.1.3. Dynamische und statische Vorgänger

Wir diskutieren noch einmal genauer die Verwaltung dynamischer und statischer 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 Rahmenadresse des letzten Aufrufs von `p`, der statische Vorgänger von `p` ist die Rahmenadresse 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 sein muss! Der dynamische Vorgänger ist die Rahmenadresse der dynamisch vorherigen Prozedur, der statische Vorgänger ist die Rahmenadresse des dynamisch letzten Aufrufs der statisch (textuell) umfassenden Prozedur.

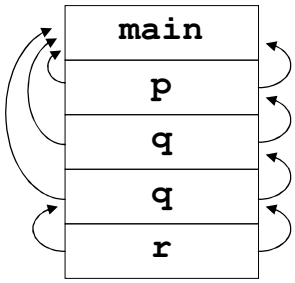
Beispiel: Betrachten wir die folgenden Definitionen geschachtelter Prozeduren:

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

Die Aufrufreihenfolge sei

`main` $\rightarrow$ `p` $\rightarrow$ `q` $\rightarrow$ `q` $\rightarrow$ `r`

Dann sieht die Stackstruktur wie folgt aus (links stehen die statischen Vorgänger und rechts die dynamischen Vorgänger):



Zur Verwaltung des statischen Vorgängers werden beim Aufruf einer Prozedur folgende Aktionen ausgeführt:

- Die Übergabe des statischen Vorgängers erfolgt in einem Register `SV`. Die Berechnung von `SV` kann durch die Bildung der Differenz der Schachtelungstiefe zwischen der aufrufenden und der aufgerufenen Prozedur erfolgen (die Schachtelungstiefe kann in der semantischen Analyse ermittelt werden):
  - Ist die Differenz gleich -1, so liegt ein Aufruf einer lokal definierten Prozedur vor, z. B. der Aufruf von `r` aus `q`. Setze in einem solchen Fall

    ```
    SV := RA;
    ```

    Differenzen kleiner als -1 können nicht vorkommen, da dies einem Aufruf einer nicht sichtbaren Prozedur entspräche (z. B. ein Aufruf von `r` aus `main`).
  - Ist die Differenz gleich 0, so liegt ein Aufruf einer Prozedur auf der gleichen Ebene vor (z. B. `q` aus `p` oder `q`), d.h. wir berechnen

    ```
    SV := M[RA + 1];
    ```

  - Ist die Differenz größer als 0, so liegt ein Aufrut einer Prozedur vor die in einem daraüberliegenden Sichtbarkeitsbereich definiert ist (z. B. ein Aufruf von `p` aus `q`). Wir müssen daher entsprechend lange der statischen Vorgängerkette folgen, d.h. wir müssen entsprechenden Code hierfür erzeugen.

    ```
    SV := M[RA + 1]; -- RA des 0. Vorgaengers
    SV := M[SV + 1]; -- RA des 1. Vorgaengers
    ...
    SV := M[SV + 1]; -- RA des n. Vorgaengers
    ```

- Bei Prozedureintritt wird dann der folgende Code ausgeführt:

  ```
  M[RA + 1] := SV;
  SA        := SA + 1;
  ```

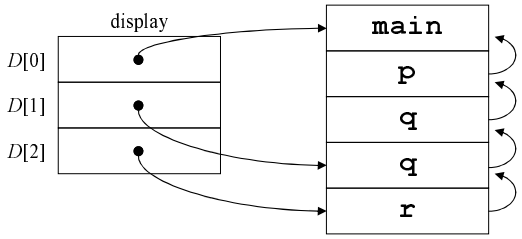
Auch hier wird wie beim dynamischen Vorgänger `SA` erhöht. Diese Inkrementierungen müssen nicht einzeln erfolgen, sondern die Berechnung der gesamten Prozedurrahmengröße kann auch einmal durch den Übersetzer erfolgen.

7.1.4. Alternative Berechnung des statischen Vorgängers

Ein Nachteil des eingeführten Schemas ist, dass bei tiefen Schachtelungen ein Zugriff auf globalere Variablen dem Durchlauf durch die statische Vorgängerkette entspricht und daher dieser Zugriff nicht mit einer einzigen Zugriffsoperation erfolgen kann. Hierzu gibt es zwei Alternativen.

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 + Relativadresse]
```

Auf globale Variablen in Tiefe j erfolgt der Zugriff durch

```
M[D[j] + Relativadresse]
```

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]`.

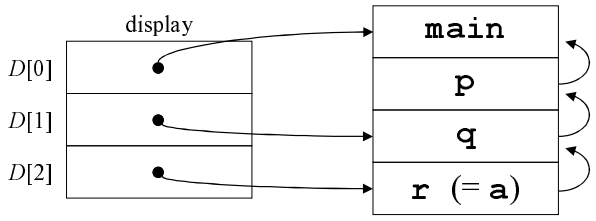
Implementierung:

- Im Prozedurrahmen ersetzen wir den statischen Vorgänger durch ein Display, dies bedeutet aber, dass beim Aufruf einer Prozedur das Display des statischen Vorgängers komplett kopiert werden muss.
- Oder 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      // Schachtelungstiefe 2
         x = 1;
      end

      call q(a);
   end

   proc q(proc r) // Schachtelungstiefe 1
      call r;
   end

   call p;
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 Prozeduren als Parametern außer der Codeadresse auch den statischen Vorgänger. Zusätzlich muss der Prozedurrahmen das Display enthalten, um dieses im Aufruf von `r (=a)` zu kopieren.

Lambda-Lifting

Diese Technik stammt aus der Implementierung funktionaler Sprachen. Die Idee ist, dass jede nicht-lokale Variable, auf die ein Zugriff in einer Prozedur erfolgt, als Referenzparameter hinzugefügt wird:

```
proc main
    int x;
```

```
    proc p
      int y;

      proc q
          int z;
          x := y + z;
      end

      call q;
    end

    call p;
end
```

Die Umformung mittels Lambda-Lifting ergibt jetzt folgendes Programm. Da Prozeduren nicht mehr auf nicht-lokale Variablen zugreifen, können alle Prozeduren nun zudem auf Top-level verschoben werden:

```
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
```

Ein **potenzieller Nachteil** ist, dass statische Vorgängerketten nun durch Verweisketten ersetzt werden (vgl. Variable `x`), zudem werden eventuell viele Prozedurparameter benötigt. Demgegenüber stehen als **Vorteile**, dass kein Speicher für Displays und statische Vorgänger benötigt wird und somit auch keine Verwaltung zur Laufzeit (Zeitersparnis!) notwendig ist. Zudem ist nun auch eine problemlose Verwendung von Prozeduren als Parameter möglich.

7.1.5. Speicherorganisation für spezielle Datentypen

Im Prozedurrahmen ist ein Speicherbereich für lokale Variablen reserviert – doch wie ist dieser genau aufgebaut? Die Grundidee basiert auf einer sequentiellen 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
...
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, was durch den Compiler berechenbar sein sollte, 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. Daher könnten wir auch 4 Bytes bei 32-Bit-Architekturen dafür reservieren.
Wir betrachten im Folgenden die Speicherung komplexerer Datentypen.

Records

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

Dann würden c_1 bis c_k sequentiell gespeichert mit

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

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

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

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 $d_i = o_i - u_i + 1$. Dann ist

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

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

AA
u_1
o_1
$\ldots$
u_k
o_k

wobei in AA die Anfangsadresse des späteren Feldes und in u_i, o_i die Feldgrenzen gespeichert werden. Damit ergibt sich

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

Damit wird also nur fester Speicherplatz für die dynamisch relevanten Parameter reserviert. Für die eigentlichen Daten wird zur Laufzeit Speicherplatz reserviert, entweder am Ende des Stacks oder auch auf dem Heap. Bei Bekanntwerden der Grenzen wird dann Folgendes zur Laufzeit ausgeführt, wenn die Feldelemente am Ende des Stacks abgelegt werden sollen:

```
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 kann dies auch für spaltenweises Abspeichern definiert werden.

Vereinigungstypen

$$\tau = \tau_1 + \tau_2$$

Verwende hier überlappendes Speichern:

$$\text{size}(\tau) = \max(\text{size}(\tau_1), \text{size}(\tau_2))$$

Variante Records

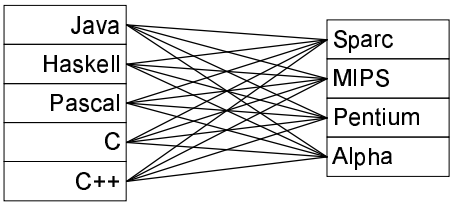
Kombination von Records und Vereinigungstypen.

Funktionen/Prozeduren

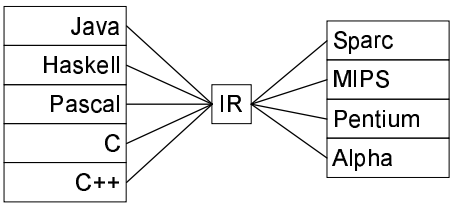
Speicherung der Codeadresse und des statischen Vorgängers (siehe oben).

7.2. Zwischencodeerzeugung

Nach der Speicherorganisation können wir uns nun der Erzeugung des Zielcodes zuwenden. Eine direkte Übersetzung in eine konkrete Zielarchitektur hat den Nachteil, dass wir bei mehreren Zielarchitekturen viele Übersetzer implementieren müssen, z.B.



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 ist es besser, eine Übersetzung in eine gemeinsame **Zwischensprache** (IR, **intermediate representation**) zu realisieren:



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 IR:

- Der IR-Code sollte durch die semantische Analyse einfach generierbar sein (daher sollte sich die Zwischensprache auch etwas an den Konstrukten der Quellsprache orientieren).
- Der IR-Code sollte in reale Zielmaschinen einfach übersetzbar sein.
- IR-Konstrukte haben eine einfache Semantik (dies ist wichtig für die Optimierung und Transformation von Zwischencodeprogrammen)

Es existieren grundsätzliche mehrere Ansätze für die Definition einer Zwischensprache IR:

1. **Stackmaschinencode**: Einfache Stackmaschine, alle Berechnungen finden auf dem Stack statt. So wird beispielsweise "`x := a + 3 * 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 *   // verknuepfe die beiden obersten Elemente
          // mit * und lege Ergebnis auf den Stack
apply +   // analog
assign    // weise oberstes Element an zweitoberste
          // Adresse zu
```

2. **3-Adresscode**: einfache Assemblersprache, allerdings können die Befehle bis zu drei Operanden verknüpfen. Es folgen einige Beispiele für Befehle im 3-Adresscode:

```
x := y op z         // mit op ∈ {+, -, *, /,...}
x := op y           // mit unaerem Operator, z.B. not
x := y              // mit y Konstante oder Variable
goto L              // Sprung
if x cop y goto L   // mit cop ∈ {=, <, >,...}
x := y[i]           // Feldindizierung
x[i] := y           // Feldindizierung
```

   Beispielübersetzung von "`x := a + 3 * 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 verwendet werden. Diese können später eventuell bestimmten Register zugeordnet werden.

3. **abstrakte Ausdrucksbäume**: Dies ist ein etwas abstrakterer Code, der später leicht in 3-Adresscode übersetzbar ist. Einen Vorschlag für so einen Zwischencode wollen wir im Folgenden genauer beschreiben.

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 haben nur Seiteneffekte.

Definiert werden die Bäume als abstrakte Assemblersprache durch folgende 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
         | LABEL Label  -- Definition einer Sprungmarke

data BinOp = PLUS | MINUS | MULT | DIV
           | AND | OR | LSHIFT | RSHIFT

data RelOp = EQ | NE | LT | GE
```

Wir betrachen nun die **Bedeutung** der einzelnen Konstrukte:

- `CONST` i – Konstante i
- `NAME` l – Symbolischer Name (l entspricht einer Assemblermarke)
- `TEMP` t – temporärer Wert mit Namen t (Maschinenregister). Beachte: hier gibt es beliebig viele Register und Marken. Dazu definiert ein Modul `Temp` die Datentypen `Label` und `Temp` und entsprechende Operationen zur Erzeugung neuer Objekte.
- `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, dass alle Speicherinhalte die gleiche Länge haben; sonst müssten noch spezialisierte Operationen angegeben werden wie `MEM` e n mit n als Anzahl an Bytes, die gelesen werden.
- `CALL` f es – Auswertung von f zu einem Namen und Auswertung der Argumentliste es von links nach rechts, dann Aufruf der Prozedur mit den ausgewerteten Argumenten. Da f ein Ausdruck und nicht nur ein symbolischer Name ist kann das Aufrufziel dynamisch berechnet werden, was z. B. für Funktionen höherer Ordnung sinnvoll ist.
- `ESEQ` sts e – Auswertung von sts (von links nach rechts), Auswertung von e und Rückgabe des Ergebnisses von e
- `MOVE` (`MEM` e_1) e_2 – werte e_1 aus und erhalte Adresse a, werte danach e_2 aus und schreibe das Ergebnis in Speicheradresse a

- `MOVE` (`TEMP` t) e – Ergebnis der Auswertung von e speichern in Register t
- `EXP` e – werte e aus und ignoriere das Ergebnis
- `JUMP` e – werte e aus und springe zum Ergebnis (das eine Marke sein sollte)
- `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
- `LABEL` l – Einsprungmarke

Die genaue **Semantik** ist nun leicht definierbar, z.B. durch einen Interpreter, der den 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 könnte durch

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

geschehen. Dabei ist `[Stm]` der Rumpf und das letzte `Temp` der Funktion das Register, in dem das Ergebnis abgelegt wird. Allerdings ist der Anfangs- und Endcode von Prozeduren oft sehr maschinenabhängig und kann daher erst später hinzugefügt werden.

7.2.2. Temporäre Namen

`LABEL` und `TEMP` haben beliebig viele Werte, und der Compiler benötigt eine Operation zur Erzeugung neuer Namen, da dieser bei der Übersetzung ständig eindeutige Namen benötigt. Zu diesem Zweck definieren wir in einem Modul `Temp` die folgenden Datentypen:

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

Die konkreten Namen sind beliebig (später werden diese durch Adressen bzw. Registernummern ersetzt). Die **Namensgenerierung** soll *on the fly* geschehen; möglich wäre eine globale „Funktion“ `newName`, dies ist jedoch nur mit Seiteneffekten möglich. Funktional kann man dies durch die Erweiterung von Basisnamen realisieren. 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.

7.2.3. 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.

```
ra = TEMP "RA"
```

Die Übersetzung spezifizieren wir durch eine Funktion `trans`, die als Eingabe ein annotiertes Quellsprachenkonstrukt in abstrakter Syntax erwartet und den zugehörigen Zwischencode zurück gibt.

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! Daher benötigen wir unterschiedliche Trans-Funktionen:

- `transL` liefert Code für *l*-Werte von Ausdrücken
- `transR` liefert Code für *r*-Werte von Ausdrücken
- `transS` liefert Code für Anweisungen

Nun ist beispielsweise

```
transR (NumExp i) = CONST i
```

aber es erfolgt keine Definition von `transL (NumExp i)`!

Die Darstellung von Variablen erfolgt nun durch

$$\texttt{(VarExp}\quad \underbrace{\textit{offset}}_{\text{relative Adresse im Prozedurrahmen}}\quad \underbrace{\textit{level}}_{\text{relative Schachtelungstiefe}}\quad \texttt{)}$$

Somit können wir die Adressierung lokaler Variablen wie folgt übersetzen:

```
transL (VarExp offset 0) =
          BINOP PLUS (TEMP ra) (CONST offset)
```

Bei nicht-lokalen Variablen müssen wir der statischen Vorgängerkette folgen:

```
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 1)))
                          (CONST 1)))
             (CONST 15)
```

Wir definieren `transR` für Variablen durch Benutzung von `transL`:

```
transR (VarExp offset l) = MEM (transL (VarExp offset l))
```

Operatoren und Funktionen

Die Übersetzung von Operatoranwendungen und Funktionsaufrufen ist recht einfach:

```
transR (OpExp e1 op e2) = BINOP op (transR e1) (transR e2)
transR (FunExp  f args) = CALL f (map transR args)
```

Zeiger und Referenzparameter

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

Der 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 notwendig für einen sicheren Programmablauf und sollte in keinem vernünftigen Compiler fehlen. Eine Alternative dazu ist, dass man mittels einer Programmanalyse zeigt, dass die Indexgrenzen immer eingehalten werden, dies ist aber nicht in allen Fällen möglich (z. B. wenn der Index von einer Benutzereingabe abhängt).

Anweisungen

`transS` hat die Parameter `bl` *(base label)* und `bt` *(base temp)* zur Generierung neuer Namen, d.h. `transS` hat den Typ

```
transS :: Label → Temp → Statement → [Stm]
```

Dann können wir die einzelnen Übersetzungen der verschiedenen Arten von Anweisungen wie folgt festlegen:

```
transS bl bt (e1 := e2) = [MOVE (MEM (transL e1)) (transR e2)]
```

Kontrollstrukturen können auch intuitiv entsprechend ihrer Bedeutung übersetzt werden. Hierzu nehmen wir an, dass wir `false` durch 0 und `true` durch 1 repräsentieren. Dann definieren wir folgende Übersetzungen für bedingte Anweisungen und Schleifen:

```
transS bl bt (if b then s1 else s2) =
  let t   = newLabel bl 1
      f   = newLabel bl 2
      e   = newLabel bl 3
      bl1 = extendLabel bl 1
      bl2 = extendLabel bl 2
   in    [CJUMP NE (transR b) (CONST 0) t f] -- if b
      ++ LABEL t : transS bl1 bt s1          -- then s1
      ++ [JUMP (NAME e)]
      ++ LABEL f : transS bl2 bt s2          -- else s2
      ++ [LABEL e]
```

```
transS bl bt (while b do s) =
  let start = newLabel bl 1
      body  = newLabel bl 2
      stop  = newLabel bl 3
      bl1   = extendLabel bl 1
   in    [LABEL start, CJUMP EQ (transR b) (CONST 0) stop body]
      ++ LABEL body : transS bl1 bt s
      ++ [JUMP (NAME start), LABEL stop]
```

Prozeduraufrufe können wie folgt übersetzt werden, wenn die Parameterübergabe *call-by-value* ist:

```
transS bl bt (p(args)) = [EXP (CALL p (map transR args))]
```

Falls einige Parameter als Referenz übergeben werden sollen (*var*-Parameter), dann können diese Parameter beim Aufruf mit `transL` statt `transR` übersetzt werden.

Prozeduren

Die notwendigen Aktionen am Anfang und Ende einer Prozedur wurden oben schon beschrieben (dynamischer/statischer Vorgänger setzen, ...) und sind auch einfach in Zwischencode übersetzbar. Der Prozedurrumpf wird wie oben angegeben übersetzt. Somit wäre der gesamte Zwischencode dann eine Folge von Prozedurrümpfen.

Konstrollstrukturen für boolesche Ausdrücke

Boolesche Ausdrücke könnten im Prinzip analog zu anderen Ausdrücken übersetzt werden. Allerdings legen viele Programmiersprachen fest, dass boolesche Ausdrücke nur partiell ausgewertet werden, falls dies ausreicht, um den Gesamtwert zu bestimmen. Beispielsweise wird in der Programmiersprache C beim Ausdruck e_1 `&&` 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. Z.B. wird der Ausdruck e_1 `&&` e_2 in den folgenden Code übersetzt:

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

Dies können wir im Zwischencode wie folgt realisieren:

```
let r = newTemp bt 1
    t = newLabel bl 1
    f = newLabel bl 2
    e = newLabel bl 3
 in ESEQ [CJUMP EQ (transR e1) (CONST 0) t f,
          LABEL t, MOVE (TEMP r) (CONST 0), JUMP (NAME e),
          LABEL f, MOVE (TEMP r) (transR e2),
```

```
 LABEL e]
(TEMP r)
```

7.3. Basisblöcke

Abstrakte Ausdrucksbäume sind gut geeignet zur Codeerzeugung nach der semantischen Analyse, aber bei der Implementierung auf realen Maschinenarchitekturen können noch Effizienzprobleme auftreten:

- Für Optimierungen (z.B. bei einer guten Registerallokation) sollte man den Code umordnen können, um Teilausdrücke in beliebiger Reihenfolge zu berechnen. Dies ist schwierig, weil Audrücke auch Seiteneffekte haben können (`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 eine kanonische Form, d.h. in eine *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 ein `CJUMP`-Sprungziel direkt dahinter folgt.

7.3.1. Phase 1: Linearisierung

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

`ESEQ`-Ausdrücke werden wie folgt transformiert:

1. Zusammenfassung von `ESEQ`s:

   `ESEQ` s_1 `(ESEQ` s_2 e`)` $\rightarrow$ `ESEQ (`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`)]` $\rightarrow$ s `++ [JUMP` e`]`
   `[CJUMP` op `(ESEQ` s e_1`)` e_2 l_1 l_2`]` $\rightarrow$ s `++ [CJUMP` op e_1 e_2 l_1 l_2`]`

3. `ESEQ` in rechten Argumenten: beachte Seiteneffekt von s auf e_1, daher wird der alte Wert von e_1 in einem neuen temporären Wert t gespeichert:

   `BINOP` *op* e_1 `(ESEQ` s e_2`)` $\rightarrow$ `ESEQ ([MOVE (TEMP` t`)` e_1`] ++` s`)`
   `(BINOP` *op* `(TEMP` t`)` e_2`)`

   `[CJUMP` *op* e_1 `(ESEQ` s e_2`)` l_1 l_2`]` $\rightarrow$ `[MOVE (TEMP` t`)` e_1`] ++`
   s `++ [CJUMP` *op* `(TEMP` t`)` e_2 l_1 l_2`]`

   Falls die Auswertung von s den Ausdruck e_1 nicht beeinflusst (d.h. falls $\text{indep}(s, e_1)$ gilt, s.u.), dann können wir die folgende vereinfachte Transformation verwenden:

   `BINOP` *op* e_1 `(ESEQ` s e_2`)` $\rightarrow$ `ESEQ` s `(BINOP` *op* e_1 e_2`)`
   `[CJUMP` *op* e_1 `(ESEQ` s e_2`)` l_1 l_2`]` $\rightarrow$ s `++ [CJUMP` *op* e_1 e_2 l_1 l_2`]`

   Wichtig ist nun eine korrekte und möglichst gute Approximation von indep. Eine sehr einfache (Verfeinerung ist möglich!) ist die folgende:

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

   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. Dies sind noch nicht alle Transformationsmöglichkeiten. Weitere Regeln wären z.B.

   `MOVE (TEMP` t`) (ESEQ` s e`)` $\rightarrow$ s `++ [MOVE (TEMP` t`)` e`]`
   `EXP (ESEQ` s e`)` $\rightarrow$ 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 mit der umgebenden Anweisung verschmolzen werden.

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 $args$ $\rightarrow$ `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**, d.h. der erzeugte Zwischencode ist eine Liste von Anweisungen mit seiteneffektfreien Ausdrücken.

7.3.2. Phase 2: Gruppierung zu Basisblöcken

Bisher erzeugt unser Übersetzungsvorgang eine Liste von Anweisungen. Um daraus später möglichst guten Code zu generieren, müssen Anweisungen in dieser Liste eventuell umgeordnet werden:

1. Umordnung von Instruktionen, so dass Sprünge möglichst eliminiert werden (z.B. wenn das Sprungziel die nächste Instruktion ist).

2. Optimierung von Folgen von `MOVE`-Instruktionen durch Umordnung und Wiederbenutzung nicht mehr benötigter Register.

Bei der ersten Aufgabe interessieren nur `JUMP`/`LABEL`-Instruktionen, während bei der zweiten Aufgabe nur `MOVE`-Instruktionen relevant sind. Aus diesem Grund unterteilen wir die Instruktionsfolge in Teillisten, innerhalb derer keine Sprünge stattfinden:

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

Die Umsetzung einer Instruktionsliste in eine Liste von Basisblöcken 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) und wird daher hier noch nicht betrachtet.

7.3.3. Phase 3: Umordnung von Basisblöcken

Beobachtung: Die Anordnung der Basisblöcke ist beliebig! Wähle daher eine Ordnung mit **möglichst wenigen Sprüngen**; ordne dazu die Basisblöcke zu (möglichst wenigen) größeren Traces um.

Definition: Ein **Trace** (**Spur**) ist eine Instruktionsfolge, die fortlaufend ausgeführt werden kann, d.h. innerhalb eines Traces kommen keine `JUMP`s vor, jedoch `CJUMP`s mit einer Zielmarke, die als nächste Instruktion nach dem `CJUMP` kommt.

Eine einfache Methode zur Anordnung der Basisblöcke zu Traces ist: Wähle Basisblock b und füge dahinter den Basisblock, zu dem verzweigt wird. Diese werden auch **Nachfolger eines Basisblocks** b genannt, 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. Somit können wir einen Algorithmus zur Berechnung von Traces realisieren, in dem wir die Basisblöcke entsprechend markieren.

Algorithmus zur Berechnung von Traces:

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

7.3.4. Sprungcodeanpassung

Auf vielen realen Maschinen sind bedingte Sprünge nur mit einem `true`-Label möglich (d.h. ein `false`-Label ist nicht erlaubt). Passe daher den Sprungcode an und modifiziere `CJUMP`, so dass dahinter immer das `false`-Label folgt:

- Falls hinter `CJUMP` das `true`-Label folgt: negiere die `CJUMP`-Bedingung und vertausche die Label.
- Falls hinter `CJUMP` weder `true`- noch `false`-Label folgt: 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, ad-hoc möglichst lange Traces zu erzeugen. Dies muss nicht unbedingt die beste Lösung sein, denn für Optimierungen (z.B. Registerallokation) kann es wichtig sein, dass häufig ausgeführte Instruktionssequenzen (z.B. Schleifenrümpfe) zusammen in einem Trace sind. Das folgende Beispiel zeigt, dass unser einfacher Algorithmus nicht unbedingt den besten Trace erzeugt.

Beispiel: Bei der Übersetzung einer While-Schleife könnte unser Algorithmus aus den Basisblöcken die folgenden Traces erzeugen:

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

Hier sind also zwei Traces vorhanden. Bei einer Sprungcodeanpassung könnte die erste Sprunginstruktion gelöscht werden.

Möglich wäre allerdings auch die folgende Anordnung:

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

Nach einer Sprungcodeanpassung ergibt sich die folgende Anordnung:

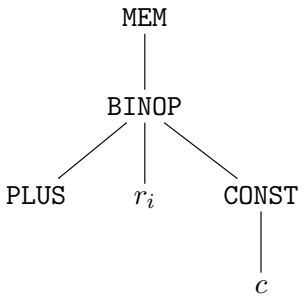
```
JUMP (NAME test)
--------------------
LABEL body
<loop body>
LABEL test
CJUMP <= i N body done
LABEL done
...
```

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

7.4. Zielcodeauswahl

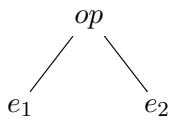
Jeder Knoten eines abstrakten Ausdrucksbaums enstpricht einer einfachen Operation (Speicherzugriff, Addition). Reale Maschinen bieten jedoch komplexe Instruktionen, die mehrere Operationen beinhalten und damit mehreren Knoten im Baum entsprechen. Betrachten wir z.B. einen Speicherzugriff mit relativer Adressierung ($M[r_i + c]$). Dieser

hat als Ausdrucksbaum die Darstellung

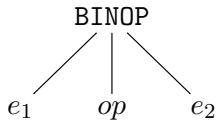


In einer realen Maschine könnte dies durch eine Maschineninstruktion realisiert werden, die als Operand $M[r_i + c]$ enthält. Übliche reale Maschinen enthalten aber auch primitive Instruktionen, so dass man den obigen Baum auch in mehrere Maschineninstruktionen übersetzen kann.
Die **Aufgabe der Zielcodeauswahl** ist nun, eine Maschineninstruktionsfolge zu finden, die einen abstrakten Ausdrucksbaum implementiert. Wie wir an dem Beispiel gesehen haben, ist die Lösung nicht eindeutig – wir suchen eine **optimale** Folge.
Hierzu stellen wir die Maschineninstruktionen als Teilbäume abstrakter Ausdrucksbäume, sogenannte **Baummuster**, dar. Die **Zielcodeauswahl** entspricht dann der (optimalen) vollständigen Überdeckung des Ausdrucksbaums mit Baummustern, d.h. die Zielcodeauswahl entspricht dann der Erkennung von Baummustern in einem abstrakten Ausdrucksbaum.

Da konkrete Zielmaschinen viele Instruktionen haben, betrachten wir hier eine vereinfachte Maschine, wobei wir die verschiedene Varianten von Instruktionen dadurch reduzieren, indem wir annehmen, dass das Register r_0 immer den Wert 0 enthält. Um das Aufschreiben der Baummuster zu vereinfachen, schreiben wir einfach



statt



Jede Maschineninstruktion kann typischerweise mehrere Baummuster abdecken (siehe Abbildung 7.1). 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 immer in Registern abgelegt (zumindest bei den üblichen Maschinen). Aus diesem Grund benötigen wird bei der Übersetzung komplexer Bäume neue Register, um die Zwischenergebnisse der Teilbäume zu speichern.

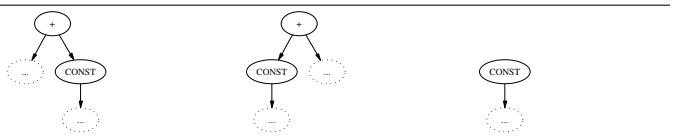
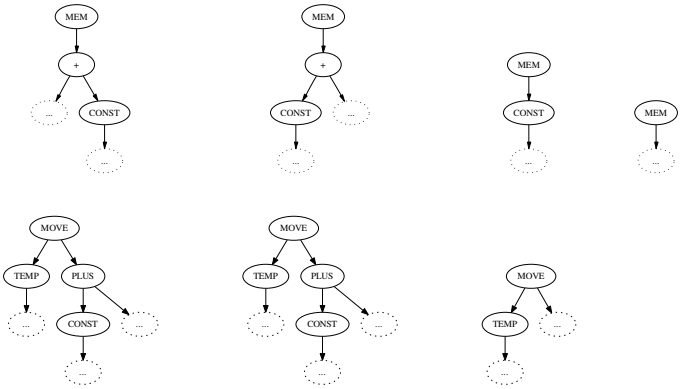
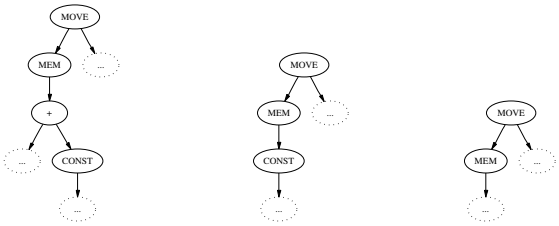
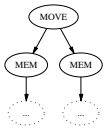
Name	Wirkung	Baummuster
`ADD`	$r_i \leftarrow r_j + r_k$	
`MUL`	$r_i \leftarrow r_j * r_k$	
`ADDI`	$r_i \leftarrow r_j + c$	
`LOAD`	$r_i \leftarrow M[r_j + c]$	
`STORE`	$M[r_j + c] \leftarrow r_i$	
`MOVEM`	$M[r_i] \leftarrow M[r_j]$	

Abbildung 7.1.: Maschineninstruktionen und zugehörige Baummuster

Die Anzahl der notwendigen Register kann man durch Wiederverwendung vorhandener Register reduzieren (dies ist die Aufgabe der Registerallokation, die später beschrieben wird).

Die Baummuster der angegebenen Beispielinstruktionen können nicht alle abstrakten Ausdrucksbäume überdecken. Um dies zu ermöglichen, führen wir noch ein Baummuster für den Zugriff auf ein Register ein:

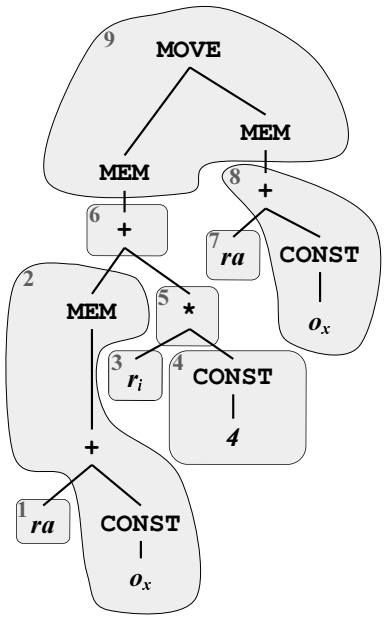
Dieses Baummuster entspricht einfach dem Register r_i (und stellt damit keine echte Maschineninstruktion dar).

In diesem Kapitel zeigen wir nur die Übersetzung von Basisblöcken ohne Sprünge. Vollständige Programme mit Sprüngen können allerdings ganz analog übersetzt werden (s. Übung).

Beispiel: Wir betrachten die Quellprogrammanweisung

```
a[i] := x;
```

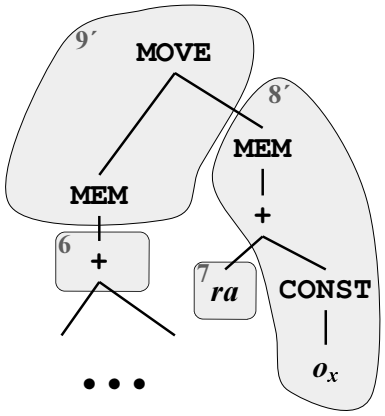
Wir nehmen an, dass die Elemente von `a` eine Größe von vier Bytes haben. $ra+o_a$ enthält die Anfangsadresse von `a` und $ra + o_x$ sei die Adresse von `x`. Zudem steht der Wert der Variablen `i` in dem Register r_i. Unter diesen Annahmen wird für diese Instruktion ein Zwischencode erzeugt, für den die folgende Überdeckung möglich ist:



Die Zwischencodeerzeugung ergibt also folgenden Zielcode, wenn wir das Ergebnis jeden Teilausdrucks in einem neuen Register speichern:

2	LOAD	$r_2 \leftarrow M[ra + o_a]$
4	ADDI	$r_4 \leftarrow r_0 + 4$
5	MUL	$r_5 \leftarrow r_i * r_4$
6	ADD	$r_6 \leftarrow r_2 + r_4$
8	ADDI	$r_8 \leftarrow r_a + o_x$
9	MOVEM	$M[r_6] \leftarrow M[r_8]$

Es ist auch noch eine andere Überdeckung möglich:



2	LOAD	$r_2 \leftarrow M[ra + o_a]$
4	ADDI	$r_4 \leftarrow r_0 + 4$
5	MUL	$r_5 \leftarrow r_i * r_4$
6	ADD	$r_6 \leftarrow r_2 + r_4$
8'	LOAD	$r_8 \leftarrow M[ra + o_x]$
9'	STORE	$M[r_6 + 0] \leftarrow r_8$

Es ist auch ein trivialer Code möglich, bei dem jeder einzelne Knoten in eine Maschinenanweisung übersetzt wird:

ADDI	$r_1 \leftarrow r_0 + o_a$
ADD	$r_2 \leftarrow ra + r_1$
LOAD	$r_3 \leftarrow M[r_2 + 0]$
ADDI	$r_4 \leftarrow r_0 + 4$
MUL	$r_5 \leftarrow r_i * r_4$
ADD	$r_6 \leftarrow r_3 + r_5$
ADDI	$r_7 \leftarrow r_0 + o_x$
ADD	$r_8 \leftarrow ra + r_7$
LOAD	$r_9 \leftarrow M[r_8 + 0]$
STORE	$M[r_6 + 0] \leftarrow r_8$

Dieser Code ist jedoch länger und in der Regel auch langsamer! Es ist also wichtig, **guten Code** zu erzeugen – doch was ist guter Code und wie erzeugt man diesen?

7.4.1. Kosten von Code

Guter Code ist ein Code, der minimale **Kosten** hat, wozu man jedoch ein Kostenmaß benötigt. Mögliche Kostenmaße sind beispielsweise:

- die Länge aller Instruktionen (z.B. für kleinen Code von Applets, die über ein (langsames) Netz geladen werden, oder in eingebetteten Systemen)
- die Ausführungszeit aller Instruktionen (Taktzyklen)
- der Energieverbrauch aller Instruktionen (z.B. bei mobilen Geräten wie Smartphones)

Bezüglich eines gegebenen Kostenmaßes können wir dann definieren:

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 die Instruktion `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 Zielcodes lokal optimal.

7.4.2. Algorithmen für (lokal) optimale Zielcodes

Intuitiv ist lokal optimaler Zielcode einfacher zu finden als optimaler Zielcode. Ob optimaler Zielcode unbedingt lohnenswert ist, hängt auch von der Zielarchitektur ab. In der Praxis kann man zwei wesentliche Zielarchitekturen unterscheiden:

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

Wir stellen daher im Folgenden Algorithmen für beide Aufgaben (lokal optimaler und optimaler Zielcode) vor.

Berechnung eines lokal optimalen Zielcodes

Der Algorithmus **Maximal Munch** (maximales Konsumieren/Mampfen) erzeugt lokal optimalen Zielcode. Es handelt sich um einen Top-down-Algorithmus, für den die Baummuster wie folgt vorbereitet werden:

> Falls ein Baummuster existiert, das in zwei kleinere Baummuster mit geringeren kombinierten Kosten aufteilbar ist, dann lösche dieses Muster.

Die Idee des Algorithmus ist nun, die Wurzel mit dem größten passenden Muster zu überdecken (d.h. ein passendes Muster mit den meisten Knoten zu wählen) und danach den Algorithmus rekursiv auf alle nichtüberdeckten Teilbäume anzuwenden.

Wähle also im obigen Beispiel `MOVEM` als größtes Muster für die Wurzel und berechne danach eine Überdeckung für die beiden Teilbäume (d.h. für die Söhne der beiden `MEM`-Knoten).

Eine **Implementierung** dieser Idee ist in Haskell weitgehend trivial. Die Überdeckungen entsprechen Pattern Matching in Regeln. Da Haskell die Regelmuster von oben nach unten austestet, bis eine passende Regel gefunden wurde, muss man nur die größten Muster zuerst aufschreiben.
In dem folgenden Programm geben wir lediglich die Liste der generierten Instruktionen aus:

```
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 das Pattern Matching gezeigt worden. Notwendig ist noch die Registerauswahl für die Teilergebnisse und das Weiterreichen der gewählten Register ($\rightsquigarrow$ Erweiterung von `codegen` in `munchExp`).
- Die Instruktionen werden in umgekehrter Reihenfolge generiert, denn die Wurzelinstruktion ist die letzte, da zuvor die Instruktionen zur Berechnung der Wurzelargumente benötigt werden.
- Der Algorithmus *MaximalMunch* terminiert immer erfolgreich, falls es für jeden Einzelknotentypen ein passendes Baummuster gibt.
- Der Algorithmus *MaximalMunch* findet immer eine lokal optimale Überdeckung, aber nicht unbedingt eine optimale.

Berechnung des optimalen Zielcodes

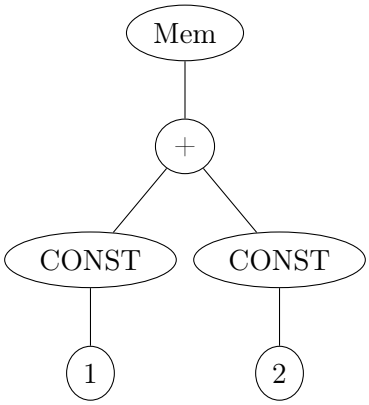
Hierzu nutzen wir das Technik der **dynamischen Programmierung**, die auf folgendem Prinzip beruht: Finde optimale Lösungen durch Finden optimaler Lösungen für Teilprobleme.

Idee: Wir berechnen für jeden Knoten die Kosten für die beste Instruktionsfolge, die diesen Teilbaum überdeckt.
Hierbei handelt es sich um einen Bottum-Up-Algorithmus: Berechne zunächst die Kosten für die Blätter, dann die Kosten für die darüber liegende Schicht usw.

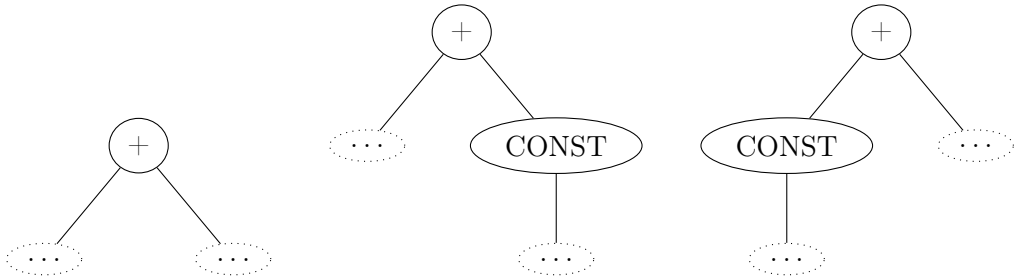
Die Kostenberechnung für einen Knoten läuft folgendermaßen ab:

- Betrachte ein passendes Baummuster für diesen Knoten, dieses hat die Kosten c.
- Die n Blätter dieses Muster entsprechen Knoten im Baum, deren Kosten c_1, c_2, ..., 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. Dies ergibt dann die Kosten für den betrachteten Knoten.

Beispiel: Wir betrachten den folgenden Teilbaum:

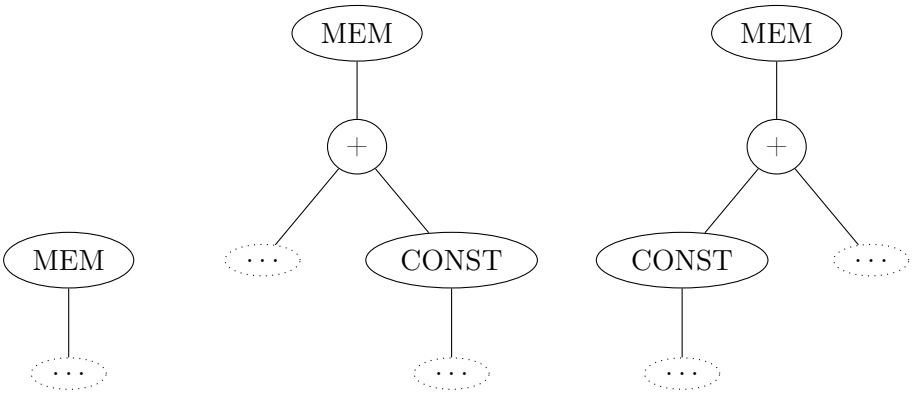


Wir nehmen an, dass jede Instruktion zwei Einheiten kostet. Die `CONST`-Muster entsprechen `ADDI`-Instruktionen und kosten daher jeweils zwei Einheiten. Es gibt die folgenden Möglichkeiten zur Überdeckung des "`+`"-Teilbaums:



`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:

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

Somit muss man bei einem optimalen Zielcode keine Instruktion für den `+`-Knoten generieren.

7.5. Programmanalyse

In dem bisherigen Verfahren zur Codeerzeugung haben wir angenommen, dass beliebig viele temporäre Werte (d.h. Register) zur Verfügung stehen. In der Praxis stehen aber nur wenige Register zur Verfügung – ein Register sollte also für mehrere temporäre Werte benutzt werden, falls die temporären Werte nicht gleichzeitig benötigt werden. Zu diesem Zweck muss der Compiler die **Lebendigkeit** von temporären Werten analysieren. Darüber hinaus sind auch andere Analysen, wie z.B. die Konstantenpropagation sinnvoll, worauf wir allerdings später noch eingehen.

Eine Variable heißt **lebendig** (an einer Programmstelle), wenn deren Wert später noch verwendet wird. Daher führen wir eine **Lebendigkeitsanalyse** (LA, **liveness analysis**) durch, die die Form einer **Rückwärtsanalyse** hat (andere Analyseformen werden später noch besprochen).

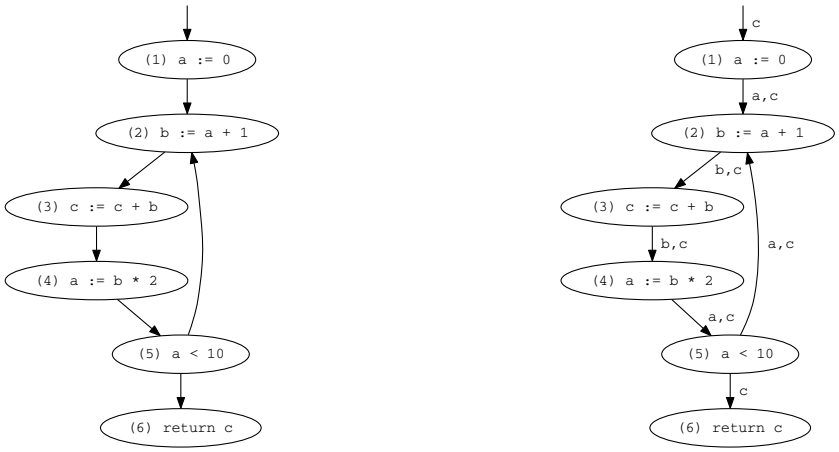
- Bei Programmende ist keine Variable lebendig, die LA arbeitet daher vom Ende zum Anfang eines Programms.
- Die für die Analyse geeignete Datenstruktur ist ein **Kontrollflussgraph**. Die Knoten entsprechen den Anweisungen im Code, und Kanten von x nach y besagen, dass die Anweisung y direkt nach der Anweisung x ausgeführt werden kann.

Beispiel: Wir betrachten den folgenden Code (dies könnte z.B. ein Zwischencode sein):

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

Der zugehörige Kontrollflussgraph hat die in Abbildung 7.2a dargestellte Struktur. Betrachten wir nun die einzelnen Variablen ausgehend vom Programmende:

- Variable `b`:
  - benutzt in 4, daher lebendig in $3 \to 4$
  - nicht überschrieben in 3, daher lebendig in $2 \to 3$
  - überschrieben in 2 und nicht benutzt in 2, daher tot in $1 \to 2$
- Variable `a`:
  - lebendig in $4 \to 5$, $1 \to 2$, $5 \to 2$
  - tot in $3 \to 4$, $2 \to 3$ (obwohl sie einen Wert hat!)
- Variable `c`: lebendig im gesamten Programm, daher auch lebendig am Anfang: es handelt sich somit um eine nicht initialisierte Variable. Hierfür könnte zum Beispiel der Compiler eine Warnung ausgeben oder eine Initialisierungsanweisung einfügen, falls `c` eine lokale Variable ist und kein Prozedurparameter.



(a) Kontrollflussgraph (b) Lebendigkeitsinformation

Abbildung 7.2.: Kontrollflussgraph und Lebendigkeitsanalyse

Wir notieren diese Lebendigkeitsinformation an den Kanten des Kontrollflussgraphen (Abbildung 7.2b). Damit sind `a` und `b` nicht gleichzeitig lebendig, sodass das gleiche Register für `a` und `b` benutzt werden kann. Somit reichen also zwei Register für die Realisierung dieser Codesequenz aus.

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

- **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?

Die hierfür verwendeten Techniken sind im Prinzip ähnlich, daher werden wir im Folgenden nur die Lebendigkeitsanalyse behandeln.

Lebendigkeitsanalyse

Wir definieren einige Begriffe, um die Lebendigkeitsanalyse exakter zu beschreiben und ein Verfahren anzugeben, wie man die Lebendigkeit effektiv berechnen kann.

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

  Beispiel: Knoten 5 hat die *out*-Kanten $5 \to 6$ und $5 \to 2$ und die *in*-Kante $4 \to 5$.

- Eine Zuweisung im Knoten n an eine Variable v *definiert* v, damit sei $\text{def}(n) = \{v\}$.

  BEISPIEL: $\text{def}(3) = \{\texttt{c}\}$, $\text{def}(5) = \varnothing$

- Die Verwendung des Wertes einer Variablen v im Knoten n *benutzt* v, damit sei $v \in \text{use}(n)$.

  BEISPIEL: $\text{use}(3) = \{\texttt{b}, \texttt{c}\}$, $\text{use}(5) = \{\texttt{a}\}$

- Eine Variable v ist **lebendig** in der Kante e, wenn ein Pfad von e zu einem 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: $\text{in}(1) = \{\texttt{c}\}$

- $\text{out}(n)$ ist die Menge der lebendigen Variablen in *out*-Kanten von n

  BEISPIEL: $\text{out}(1) = \{\texttt{a}, \texttt{c}\}$

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

- $v \in \text{use}(n) \Rightarrow v \in \text{in}(n)$
- Für alle $m \in \text{pred}(n)$: $v \in \text{in}(n) \Rightarrow v \in \text{out}(m)$
- $v \in \text{out}(n) \wedge v \notin \text{def}(n) \Rightarrow v \in \text{in}(n)$

Die letzte Regel zeigt, dass die Information von Ende zum Anfang im Programm fließt (Rückwärtsanalyse).

Damit können wir für jeden Kontrollflussgraphen ein **Gleichungssystem** aufstellen. Die Analyse muss dann dieses Gleichungssystem, bei dem die Werte von $\text{in}(n)$ und $\text{out}(n)$ gesucht sind, lösen. Für alle Knoten n gelten folgende Gleichungen:

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

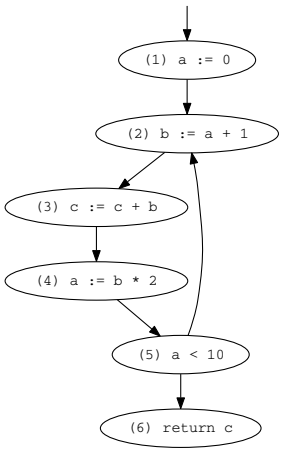
Die Menge der Variablen in diesem Gleichungssystem sind also

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

Da die Werte der zu bestimmenden Variablen Mengen sind, kann die Berechnung einer Lösung durch eine **Fixpunktiteration** erfolgen:

```
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 ∀n : in(n) == in'(n) ∧ out(n) == out'(n)
```

Betrachten wir hierzu noch einmal das Beispiel von oben:



Dann ergibt sich die in Tabelle 7.1 dargestellte Iteration. Die siebte Runde ergibt die glei-

			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	

Tabelle 7.1.: Iterationen der Lebendigkeitsanalyse

chen Ergebnisse wie Runde 6, der Algorithmus terminiert daher. Bezüglich der **Berechnungsstrategie** stellen wir fest, dass die Ordnung in der inneren „`for each`“-Schleife für

Knoten beliebig ist. Wir wählen daher die Ordnung mit einem schnellen Datenfluss. Da die Abhängigkeit

$$\text{out}(n) = \bigcup_{m \in \text{succ}(n)} \text{in}(m)$$

besteht, bestimmen die Werte der Nachfolger die Werte der Vorgänger. Wir berechnen daher die Werten von den Nachfolgern zu den Vorgängern (d. h. für dieses Beispiel in der Knotenordnung 6, 5, 4, 3, 2, 1). In jeder Iteration wird zudem zunächst out(n) und dann in(n) berechnet, da in(n) von out(n) abhängt. Damit ergibt sich die in Tabelle 7.2 dargestellte verkürzte Ausführung, bei der das Ergebnis der dritten Iteration schon dem der zweiten Iteration 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

Tabelle 7.2.: Iterationen der verbesserten Lebendigkeitsanalyse

7.6. Registerallokation

Die bisherige Annahme bei der Codeerzeugung war, dass beliebig viele Register zur Verfügung stehen. In realen Maschinen stehen jedoch nur endlich viele Register zur Verfügung. Daher ist die Aufgabe der **Registerallokation** die Zuordnung von temporären Werten zu konkreten Registern (falls möglich) und damit die Einsparung von `MOVE`/`LOAD`-Instruktionen durch die Wiederverwendung von Registern.

Das generelle Vorgehen besteht aus einer **sequentiellen Codeinspektion** nach der Codeauswahl:

- Entscheide bei jedem Vorkommen eines temporären Wertes: Liegt dieser in einem Register? Falls nein, in welches Register kann man diesen legen? Die generelle Annahme ist hier, dass alle Maschinenoperationen auf Registern arbeiten (wie z.B. bei RISC-Architekturen) und somit vorkommende temporäre Werte sich in einem Register befinden müssen.
- Falls ein temporärer Wert nicht in einem 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 für diesen Wert zuweisen.

Nachfolgend wird die Registerallokation für **Basisblöcke** vorgestellt. Im Allgemeinen werden kompliziertere Techniken angewendet wie z.B. die globale Datenflussanalyse, um zu analysieren, wo welche temporären Werte benötigt werden. 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* (mehr Details hierzu findet man z.B. im Buch von Andrew Appel).

Die **Menge der freien Register** ist eine Teilmenge der Maschinenregister, über die wirklich frei verfügt werden kann, da einzelne Maschinenregister üblicherweise für die Rahmenadresse, den Stack- und Heap-Pointer etc. reserviert sind.

Zu jeder Programmvariablen und jedem temporären Wert v bezeichnet M_v den **Speicherplatz** im Stack, der zur Speicherung für diese Programmvariable reserviert ist. Dieser Platz wird für temporäre Werte eventuell gar nicht benötigt, wenn die Registerallokation dies erreicht. Im Folgenden sprechen wir daher einheitlich von **Variablen** v, auch wenn dies keine Programmvariable, sondern nur ein temporärer Wert ist, der bei der Codeauswahl eingeführt wurde.

7.6.1. Benötigte Informationen

Der Algorithmus zur Registerallokation verwaltet folgende Informationen:

- **Registerinhalte** (*content*)

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

  Hierbei ist $\mathrm{Con}(R)$ die Menge der Variablen (d.h. Programmvariablen oder temporäre Werte), die momentan im Register R gespeichert sind. Zu Beginn eines Basisblocks ist $\mathrm{Con}(R) = \varnothing$ für alle Register R. $\mathrm{Con}(R)$ kann eventuell auf Grund von Kopieroperationen mehrere Elemente enthalten.

- **Variablenposition**

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

  Hierbei gibt $\mathrm{Pos}(v)$ die Stelle an, wo sich der aktuelle Wert von v befindet (d.h. ein Register R oder eine Speicherposition M_v). Zu Beginn eines Basisblocks ist $\mathrm{Pos}(v) = M_v$ für alle Programmvariablen. $\mathrm{Pos}(v)$ kann undefiniert sein, falls v im aktuellen Block nicht mehr benötigt wird. Im Prinzip könnte $\mathrm{Pos}(v)$ auch eine Menge sein, aber dann wird der Algorithmus zur Registerallokation komplexer.

- **Lebendigkeit von Variablen**:

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

An einer Programmstelle ist Live(v) = `true`, falls v später noch benötigt wird. Zum Ende eines Basisblocks ist Live(v) = `false` für alle temporären Werte v. Die Berechnung dieser Information kann durch Rückwärtspropagation in einem Basisblock erfolgen: Falls in der Instruktion I der Wert x überschrieben und der Wert y benutzt wird, so ist Live(x) = `false` und Live(y) = `true`; alle andere Informationen bleiben unverändert für Live von I (dies ist also eine Spezialisierung des Verfahrens aus Kapitel 7.5; diese Information kann aber durch eine globale Analyse verbessert werden).

Die zentrale **Hilfsoperation** für die Registerallokation ist eine Prozedur `getreg()`, die ein Register zur Speicherung einer Variablen bestimmt. Dabei benutzt diese Funktion die Parameter Con / Pos / Live, verändert diese gegebenenfalls und erzeugt auch Code, d.h. es ist eine Funktion $(\text{Con}, \text{Pos}, \text{Live}) \to (\text{Con}, \text{Pos}) + \text{Code}$.

Wir verwenden folgenden **Algorithmus** für `getreg()`:[1]

- Falls ein Register R mit $\text{Con}(R) = \{v\}$ und Live(v) = `false` existiert, dann setze $\text{Con}(R) := \varnothing$ und gib das Register R zurück (dies entspricht der Wiederbenutzung eines nicht mehr benötigten Registers).
- Sonst: wähle ein freies Register R (d.h. ein Register R mit $\text{Con}(R) = \varnothing$) und gib R zurück.
- Sonst (d.h. falls kein Register frei ist): Wähle ein belegtes Register R: Erzeuge für alle $v \in \text{Con}(R)$ mit $\text{Pos}(v) = R$ einen Befehl

  `STORE` $M[M_v] \leftarrow R$

  und setze $\text{Pos}(v) := M_v$. Setze dann $\text{Con}(R) := \varnothing$ und gib R zurück.

  Die Auswahl von R ist in diesem Schritt beliebig, da im Allgemeinen kein bestes Verfahren zur Registerauswahl bekannt ist.

Mit `getreg` können wir nun realen Maschinencode generieren: Wann immer der bisherige Code ein Register benötigt, so bestimme ein reales Register mittels `getreg()`.

7.6.2. Algorithmus für die Registerallokation

1. **Instruktionen im Basisblock:** Betrachte nacheinander alle Instruktionen eines Basisblocks (in der Ausführungsreihenfolge). Sei hierbei *instr* die aktuelle Instruktion:

   (a) **Operanden:** Für jeden temporären Wert (virtuelles Register) r_i, der als Operand in *instr* vorkommt:[2]

[1] Dies ist nur eine Möglichkeit zur Registerallokation, es sind auch viele andere Varianten vorstellbar.

[2] Als Operand (d.h. nicht als Ziel einer Zuweisung) 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.

1. Wähle R_i mit $r_i \in \text{Con}(R_i)$, sofern dieses existiert.
2. Ansonsten wähle $R_i :=$ `getreg()` und füge zunächst die eventuell hierdurch erzeugten `STORE`-Instruktionen dem Zielprogramm hinzu. Füge nun die Instruktion `LOAD` $R_i \leftarrow M[M_{r_i}]$ hinzu und setze $\text{Con}(R_i) = \{r_i\}$.

(b) **Ergebnis:** Falls die Instruktion *instr* das Ergebnis in einem temporären Wert r_j speichert (z.B. bei Instruktionen wie `ADD`, `MUL`, `ADDI`, `LOAD`):

1. Falls einer der Operanden gleich dem temporärem Ergebniswert r_j ist, so benutze das Register dieses Operanden für r_j.
2. Falls unter (a) mindestens ein neues Operandenregister R für *instr* mittels `getreg()` gewählt wurde, so benutze das erste dieser Register für das Ergebnis r_j.[3]
3. Ansonsten setze $R :=$ `getreg()`, füge die eventuell generierte `STORE`-Instruktion ein und benutze R für das Ergebnis.

(c) **Instruktion:** Erzeuge die Maschineninstruktion für *instr* mit den in (a) und (b) gewählten Maschinenregistern.

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

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

(e) **Weitere Aktualisierung:**

- Aktualisiere Con bezüglich der Live-Information nach der Instruktion *instr*: Für alle Register S mit $x \in \text{Con}(S)$ und $\text{Live}(x) =$ `false` setze $\text{Con}(S) = \text{Con}(S) \setminus \{x\}$. Eigentlich ist diese Aktualisierung nicht notwendig, da sie auch bei `getreg()` gemacht wird, aber zur Verdeutlichungen notieren wir diese hier.

[3] Man könnte natürlich auch das zweite nutzen.

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

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

Dieses Verfahren kann auch in die Codeauswahl integriert werden, indem bei der Übersetzung von Ausdrucksbäumen `getreg()` direkt zur Auswahl der Register für Zwischenergebnisse benutzt wird.

Beispiel: Wir nehmen an, dass die Maschinenregister R, S und T zur Verfügung stehen. Im Programm wird die Variable w verwendet, alle anderen Variablen seien temporär. Wir betrachten die folgende Instruktionsfolge (aus Vereinfachungsgründen lassen wir die Befehlsnamen weg):

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

Dann erfolgt die Codeerzeugung mit Registerallokation und Berechnung der Funktionen Con und Pos wie folgt:

Instruktion	Maschinencode	Con	Pos	Kommentar
$y \leftarrow 1$	$R \leftarrow 1$	$R \mapsto \{y\}$	$y \mapsto R$	`getreg()` $= R$ (für y)
$z \leftarrow w$	$S \leftarrow M[M_w]$	$R \mapsto \{y\}$ $S \mapsto \{z\}$	$y \mapsto R$ $z \mapsto S$	`getreg()` $= S$ (für z)
$x \leftarrow y + z$	$T \leftarrow R + S$	$R \mapsto \{y\}$ $S \mapsto \{z\}$ $T \mapsto \{x\}$	$y \mapsto R$ $z \mapsto S$ $x \mapsto T$	`getreg()` $= T$ (für x)
$v \leftarrow x + y$	$M[M_z] \leftarrow S$ $S \leftarrow T + R$	$R \mapsto \{y\}$ $S \mapsto \{v\}$ $T \mapsto \{x\}$	$y \mapsto R$ $z \mapsto M_z$ $x \mapsto T$ $v \mapsto S$	`getreg()` $= S$ (für v) $\Longrightarrow M[M_z] \leftarrow S$
$t \leftarrow z * v$	$M[M_x] \leftarrow T$ $T \leftarrow M[M_z]$ $T \leftarrow T * S$	$R \mapsto \{y\}$ $S \mapsto \varnothing$ $T \mapsto \{t\}$	$y \mapsto R$ $z \mapsto M_z$ $x \mapsto M_x$ $v \mapsto S$ [4] $t \mapsto T$	`getreg()` $= T$ (für z) $\Longrightarrow M[M_x] \leftarrow T$
$u \leftarrow y * x$	$S \leftarrow M[M_x]$ $S \leftarrow R * S$	$R \mapsto \varnothing$ $S \mapsto \{u\}$ $T \mapsto \{t\}$	$y \mapsto R$ $z \mapsto M_z$ $x \mapsto M_x$ $v \mapsto S$ $t \mapsto T$ $u \mapsto S$	`getreg()` $= S$ (für x)
$w \leftarrow u + t$	$R \leftarrow S + T$	$R \mapsto \{w\}$ $S \mapsto \varnothing$ $T \mapsto \varnothing$	$w \mapsto R$ $\vdots$	`getreg()` $= R$ (für w)
	$M[M_w] \leftarrow R$			

Da nur drei Register zur Verfügung stehen, muss somit Speicherplatz für die temporären Werte z und x reserviert werden, aber nicht für die temporären Werte y, v, t und u, die alle in Registern gehalten werden können.

[4] An dieser Stelle wude das Register S durch den Schritt (e) bereits freigegeben. Da auf die Variable v aber nicht mehr zugegriffen wird (Live(v) = `false`) ist diese inkonsistente Information hier unschädlich. Alternativ könnte der Algorithmus auch um eine Bereinigung der Pos-Information erweitert werden.

8. Techniken zur Code-Optimierung

Bisher haben wir uns nur darum gekümmert, Code zu erzeugen, der semantisch äquivalent zum Quellprogramm und ausführbar auf einer konkreten Zielmaschine ist. In diesem Kapitel wollen wir uns mit der Erzeugung „möglichst guten“ Zielcodes (eigentlich „optimaler Zielcode“, aber dieser Begriff ist hier schwer zu fassen) beschäftigen. „Möglichst gut“ kann sich dabei auf verschiedene Aspekte beziehen, wie z.B. möglichst schnelle Ausführung, möglichst wenig Laufzeitspeicher, aber auch möglichst geringe Codegröße (dies ist z.B. bei Applets oder eingebetteten Systemen wichtig).

Grundsätzliche Methoden hierzu sind:

- **Algebraische Optimierung**: Gesetze der Quellsprache werden ausgenutzt, um Transformationen auf Quellsprachebene vor der eigentlichen Übersetzung vorzunehmen.
- **Partielle Auswertung**: Auswertung bekannter Teile im Quellprogramm (siehe Kapitel 2.4)
- **Maschinenunabhängige Optimierung**: Transformationen auf der Ebene der Zwischensprache, wie z.B.
  - **lokale Optimierung**: betrachte kleinere Programmausschnitte (z.B. Basisblöcke) für die Optimierung
  - **globale Optimierung**: betrachte größere Programmeinheiten (z.B. Prozeduren, aber beispielsweise auch das gesamtes Programm) für die Optimierung. Hierzu gibt es wieder zwei wesentliche Techniken:
    - **Datenflussanalyse**: liefert Informationen zum Laufzeitverhalten durch Analyse des Kontrollflussgraphen (z.B. durch Fixpunktberechnung)
    - **abstrakte Interpretation**: liefert Informationen zum Laufzeitverhalten durch „abstrakte“ Programmausführung, d.h. Berechnung mit abstrakten Werten statt mit konkreten Werten (z.B. Rechnen mit Vorzeichen statt konkreten Zahlwerten)
- **maschinenabhängige Optimierung**: Ausnutzung spezieller Maschineneigenschaften, z.B. Adressierungsmodi oder Registerauwahl

Im Folgenden werden wir die Ideen dieser Methoden an Hand einiger Beispiele erläutern.

8.1. Algebraische Optimierung

Diese Technik nutzt mathematische Gesetze der Quellsprache aus, beispielsweise:

```
0 * x ⇝ 0
x ^ 2 ⇝ x * x
```

Dies wird in imperativen Sprachen wegen der Seiteneffekte seltener eingesetzt. Relevant ist diese Technik hingegen bei deklarativen Sprachen. Betrachten wir hierzu das folgende Haskell-Programm:

```
foldr :: (a → b → b) → b → [a] → b
foldr _ 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 :: [Int] → [Int]
sum = foldr (+) 0

qu :: Int → Int
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 sogenannte **Zwischendatenstrukturen**, $[1, 4, 9]$ und $[1, 16, 81]$.

Zwischendatenstrukturen sind kein Bestandteil des Endergebnisses, sondern werden nur in Zwischenschritten verwendet. Deren Aufbau kostet jedoch Zeit und Speicher. Die Benutzung von Zwischendatenstrukturen ist typisch für einen funktionalen/kompositionellen Programmierstil. Der explizite Aufbau von Zwischendatenstrukturen kann durch bestimmte Quellsprachentransformationen vermieden werden. Eine Möglichkeit hierzu ist die Ausnutzung mathematischer Gesetzmäßigkeiten.

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

BEWEIS: durch strukturelle Induktion über xs:

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

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

- Induktionsschritt: Wir müssen zeigen, dass wenn die Behauptung für eine Liste xs richtig ist, dann gilt sie auch für die Liste $x : xs$.

Wir nehmen also an (Induktionsannahme): Das Lemma gelte für xs. Zu zeigen ist nun die Behauptung für $x : xs$. Es gilt:

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

Ein weiteres hilfreiches Lemma ist das folgende:

LEMMA: Für alle f, g, e, xs gilt:

$$\texttt{foldr } f \texttt{ } e \texttt{ (map } g \texttt{ } xs) \texttt{ = foldr (\ } x \texttt{ } y \texttt{ -> } f \texttt{ (} g \texttt{ } x) \texttt{ } y) \texttt{ } e \texttt{ } xs$$

Der Beweis hierzu kann analog wie oben geführt werden.

Mit diesen Aussagen lässt sich obiger Ausdruck e reduzieren zu:

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

Als Effekt dieser Transformation kann man sehen, dass die oben gezeigten Zwischendatenstrukturen nicht mehr aufgebaut werden.

Hier wurde die **Seiteneffektfreiheit** wesentlich ausgenutzt, d.h. solche Transformationstechniken können gut bei funktionalen Sprachen angewendet werden. In imperativen Sprachen können zumindest seiteneffektfreie arithmetische Ausdrücke transformiert werden. Wichtig ist die algebraische Optimierung auch in relationalen Anfragesprachen (vgl. Datenbanken).

8.2. Partielle Auswertung

Die **partielle Auswertung** hat eine ähnliche Motivation wie die algebraische Optimierung, jedoch soll dies möglichst automatisch geschehen. Dies bedeutet, dass die Transformationsgesetze nicht von Hand gefunden und bewiesen werden, sondern korrekte Transformationen automatisch gefunden werden sollen. Dazu wird das Quellprogramm **partiell** ausgewertet, d.h. dort, wo immer das möglich ist, und dadurch simplifiziert (siehe Kapitel 2.4). Das Problem hierbei ist allerdings, dass die partielle Auswertung immer terminieren soll, selbst bei Endlosschleifen im Quellprogramm. Dazu werden Techniken zur Schleifenentdeckung (*loop checking*), angwendet, was hier aber nicht genauer erläutert wird.

BEISPIEL: Als einfaches Beispiel betrachten wir die Auswertung eines Haskell-Ausdrucks:

```
  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
```

Die Mächtigkeit der partiellen Auswertung kommt allerdings erst dann wirklich zum Tragen, wenn wir Ausdrücke mit Variablen (z.B. Parameter innerhalb von Funktionen) partiell auswerten, was auch möglich ist. Dies kann man beispielsweise dadurch tun, dass man mögliche Werte für bestimmte Variablen einsetzt und dann austestet, ob man damit weiterkommt. Betrachten wir dazu den Ausdruck $e =$ `sum (map qu ys)` und werten ihn etwas aus:

```
sum (map qu ys)
  = foldr (+) 0 (map qu ys)
```

Nun kommt man nicht weiter, da der Aufruf von `map` nur ausgewertet werden kann, wenn das Argument `ys` genauer bekannt ist (da `map` durch Pattern Matching über dieses Argument definiert ist). Aus diesem Grund „raten“ wir mögliche Werte für `ys`, d.h. wir setzen für `ys` zunächst einmal die beiden möglichen Werte der leeren oder nicht-leeren Liste ein:

1. `ys = []`: Dann können wir wie folgt weiter auswerten:

   ```
   foldr (+) 0 (map qu [])
     = foldr (+) 0 []
     = 0
   ```

2. `ys = x:xs`: Dann können wir wie folgt weiter auswerten:

   ```
   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)
   ```

Wenn man nun mit der partiellen Auswertung weitermachen würde, d.h. indem wir wieder `map` auswerten, dann würde der partielle Auswerter in eine Endlosschleife laufen. Stattdessen definieren wir den übrig gebliebenen komplexen Ausdruck als eine neue Funktion `f`, d.h. der Aufruf “`f zs`” steht für ein Vorkommen des Ausdrucks

```
foldr (+) 0 (map qu zs)
```

Wenn wir jetzt in die obigen partiellen Auswertungen den komplexen Ausdruck durch einen Aufruf von `f` ersetzen, erhalten wir folgende Gleichungen oder Regeln für `f`:

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

Damit haben wir eine neue Funktion `f` definiert, so dass wir den ursprünglichen Ausdruck e durch "`f ys`" ersetzen können.

Diese Transformation führt zu einem neuen Code, der keine Zwischendatenstrukturen aufbaut! Dieser Code ist auch schneller (da `sum`, `qu` schon ausgerechnet worden sind). Außerdem kann der Code auch kleiner werden (`sum`, `qu`, `foldr`, `map` sind eventuell überflüssig, siehe *dead code elimination*), aber dies muss nicht unbedingt der Fall sein. Darüber hinaus funktioniert dieses Verfahren im Prinzip automatisch. Allerdings sind hierzu viele technische Probleme zu lösen, die wir hier nicht behandeln können.

8.3. Maschinenunabhängige lokale Optimierung

Diese Optimierungen werden üblicherweise auf der Zwischensprachebene angewendet und optimieren in Bezug auf lokale Codestücke (z.B. Basisblöcke). Einige dieser Methoden sind auch zur globalen Optimierung von Programmen (s.u.) einsetzbar.

8.3.1. Konstantenpropagation (constant propagation)

Falls der Wert einer Variablen an einer Stelle bekannt ist, dann ersetze die Variable durch den Wert der Variablen.
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
```

Diese Optimierung wird oft mit der folgenden Optimierung kombiniert.

8.3.2. Konstantenfaltung (constant folding)

Werte konstante Ausdrücke zur Compilezeit aus. Dies wird oft erst ermöglicht durch die Konstantenpropagation. Daher werden üblicherweise beide Methoden iterativ angewendet.
Beispiel: Durch Konstantenfaltung des obigen Programms ergibt sich:

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

Wiederum durch Konstantenpropagation:

```
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
```

8.3.3. Kopierpropagation (copy propagation)

Variablen werden hier durch andere identische Variablen ersetzt, zum Beispiel:

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

Ersetze "`z := a * x`" dann durch "`z := a * y`". Als Effekt wird eventuell `x` tot (weil woanders nicht mehr benötigt), so dass dann auch die Zuweisung

```
x := y
```

eliminiert werden kann.

8.3.4. Reduktion der Ausdrucksstärke von Operationen

Ersetze komplexe Operatoren durch einfachere (auch auf Quell- und Zielsprachebene). Beispiele:

ersetze	`x ^ 2`	durch	`x * x`	
ersetze	`x * 2`	durch	`x + x`	
ersetze	`x * 2`	durch	`lshift x`	(Zielsprachebene)

8.3.5. In-line Expansion

Ersetze „einfache“ Funktionsaufrufe (z.B. ohne Kontrollstrukturen oder Rekursion) durch den Funktionsrumpf. Dies kann auch schon auf Quellsprachebene geschehen.

Beispiel: Betrachte die Definition

```
sq x = x * x
```

Ersetze dann im Codestück "`...(sq y)...`" den Funktionsaufruf durch den Rumpf (wobei die aktuellen Parameter substituiert werden) und erhalte "`...(y * y)...`".

Achtung: dies ist nur sinnvoll bei einfachen Argumenten, sonst besteht die Gefahr von „Codeexplosion“, falls die formalen Parameter mehrfach im Rumpf vorkommen.

Man kann auch bei einfachen Kontrollstrukturen, insbesondere bei Fallunterscheidungen im Rumpf, In-line Expansion anwenden. Betrachten wir dazu das folgende Programm

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

und den Codeausschnitt

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

wobei der Wert `True` z.B. durch Konstantenpropagation dort auftauchen könnte. Diesen Teilausdruck kann man mittels In-Line Expansion wie folgt vereinfachen:

```
(f True e1 e2)
  = if True then e1 else e2
  = e1
```

Hierbei ist der letzte Schritt eine Konstantenfaltung.

Die Schwierigkeit der In-line Expansion sind gute Kriterien, bei denen diese Transformation Vorteile (und keine Nachteile) hat. Mögliche Kriterien sind beispielsweise:

- es handelt sich nicht um eine rekursive Funktion (ansonsten wäre eine partielle Auswertung besser)
- die aktuellen Parameter sind entweder einfach oder werden im Rumpf nicht dupliziert (ansonsten besteht die Gefahr der Codeexplosion)

Diese Transformation ist eventuell auch auf Zielsprachebene möglich: Beispielsweise könnte man den Funktionsaufruf `(abs x)` auf der Zielsprachebene zum Beispiel durch eine Instruktion „setze das Vorzeichen-Bit auf positiv“ ersetzen.

8.3.6. Elimination redundanter Berechnungen

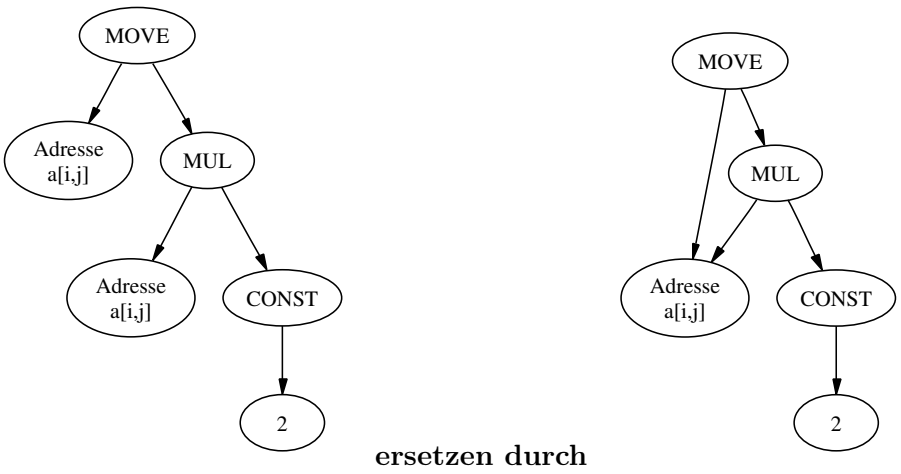
Falls Teilberechnungen mehrfach durchgeführt werden, so eliminiere die Mehrfachberechnungen.

Beispiel: Im Codestück

```
a[i,j] := a[i,j] * 2
```

kommt im Zielcode der Teilausdruck zur Adressberechnung von `a[i,j]` doppelt vor. Hierbei könnte die zweite Berechnung eliminiert werden, indem man das Ergebnis der ersten Berechnung in einem Register zwischenspeichert.

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



8.3.7. Schleifenoptimierungen

Die Optimierung von Schleifen gehört eher zu den globalen Optimierungen, da sie keine Optimierung im Basisblock, sondern im Kontrollflussgraphen ist. Diese Optimierungen sind aber sehr lohnenswert, da die Ausführung von Schleifencode oft einen großen Anteil an der Gesamtlaufzeit hat (typischerweise ca. 90%). Hierbei sind z.B. die folgenden Optimierungen möglich.

Verschiebung von Schleifeninvarianten

Falls Operanden in einer Schleife nicht verändert werden und keine Seiteneffekte auftreten, so ziehe diese 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;
```

Hier ist `(i + n)` invariant bezüglich der inneren Schleife und `sqrt(n)` ist invariant bezüglich beider Schleifen. Daher können wir diese vor die entsprechenden Schleifen verschieben:

```
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 hierdurch: 19.999 Berechnungen von `sqrt(n)` und 19.800 Berechnungen von `r * (i + n)`.

Schleifenoptimierungen sind auch lohnenswert auf Zwischensprachenebene, z.B. bei der Adressberechnung von Feldzugriffen, wo auch Teilausdrücke der Adressberechnung invariant sein können.

Schleifenentfaltung

Kleinere Schleifen können durch Initialisierung, Inkrementierung und Prüfung der Abbruchbedingung einen relativ großen Overhead haben. Eine Verbesserung kann man daher durch die „Entfaltung“ kleinerer Schleifen erreichen.

Beispiel:

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

Die innere Schleife wird entfaltet und ersetzt:

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

8.4. Maschinenunabhängige globale Optimierung

Hierbei wird mehr als ein Basisblock betrachtet, zum Teil werden bekannte lokale Optimierungen wie Konstantenpropagation und -faltung auf das gesamte Programm ausgeweitet.

8.4.1. Elimination toten Codes (dead code elimination)

Toter Code sind Anweisungen, die nie bei einer Programmausführung erreichbar sind; dies wird häufig durch Konstantenpropagation durch das gesamte Programm erkennbar.

Beispiel: Häufig fügt man Debugging-Code zu seinem Programm hinzu und steuert dessen Ausführung durch eine globale Konstante:

```
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

8.4.2. Codeverschiebung über Basisblöcke

Dies ist eine ähnliche Optimierung wie die Verschiebung von Schleifeninvarianten, aber hier wird die Verschiebung auch zwischen anderen Kontrollstrukturen, z.B. bei Fallunterscheidungen, betrachtet.

Beispiel: Betrachten wir die folgende Fallunterscheidung:

```
if i < j then a := i + k - 1 else
if i > j then a := i + k + 1 else a := i + k
```

Hier hat `i + k` in allen Zweigen den gleichen Wert, so dass wir diesen vor die Fallunterscheidung schieben:

```
s := i + k;
if i < j then a := s - 1 else
if i > j then a := s + 1 else a := s
```

Hier ändert sich zwar die Laufzeit nicht, aber dafür wird der erzeugte Zielcode kleiner.

8.5. Maschinenabhängige Optimierungen

Diese Optimierungen sind abhängig von der Zielmaschinenstruktur. Die Zielcodeauswahl und die Registerauswahl sind bereits diskutierte Beispiele. Eine der wichtigsten weiteren Techniken ist die sogenannte **Peephole Optimization**. Hierunter versteht man die Optimierung kleiner Befehlssequenzen am Ende der Zielcodeerzeugung („Guckloch"-Optimierung), die durch schematische Übersetzung Redundanzen aufweisen könnten.

Beispiel: Der Programmcode

```
x := y + 2;
z := x * 3;
```

ergibt bei einer naiven Übersetzung[1] den folgenden Zielcode:

```
        LOAD R ← M_y
        ADDI R ← R + 2
(1)     STORE M_x ← R
(2)     LOAD R ← M_x
        MUL R ← R * 3
        STORE M_z ← R
```

Zur Peephole-Optimierung wird eine „Schablone“ über benachbarte Instruktionen des Zielcodes geschoben. In unserem Beispiel lässt sich bei Betrachtung von (1) und (2) die Instruktion (2) einsparen.

Weiteres Beispiel:

```
(3)     JUMP l_1
        ...
        l_1 : JUMP l_2
```

Ersetze im Beispiel die Instruktion (3) durch `JUMP` l_2.

8.6. Datenflussanalyse

Die **Datenflussanalyse** basiert auf Flussgraphen und liefert Informationen an Punkten im Flussgraph (z.B. am Anfang/Ende von Basisblöcken). Diese Informationen sollen für **alle** möglichen Programmläufe gelten. Sie sind häufig Voraussetzung für weitere mächtige Optimierungen.

Klassifikation:

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

Bestandteile einer Datenflussanalyse ist eine Flussgraph mit Basisblöcken B als Knoten. Der Flussgraph enthält eine Kante, falls man im Kontrollfluss von einem Block zum nächsten verzweigen kann (definiere Mengen der Vorgänger- und Nachfolgerblöcke pred(B) und succ(B)). Folgende Informationen spielen bei der Berechnung eine Rolle (der genaue Inhalt ist abhängig von der konkreten Analyse):

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

Die Analyse geschieht dann in zwei Schritten:

- Aufstellen von Gleichungen über in(B_i), out(B_j), gen(B_k), kill(B_l)
- Lösen der Gleichungen durch Fixpunktiteration

[1] Bei der vorher vorgestellten Methode zur Registerallokation würde dies nicht auftauchen.

Beispiele:

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

  ```
  x := 5;
  ...
  if x < 10 then ...
  ```

  Hierbei ist z.B. die Frage interessant, ob die Zuweisung “`x := 5`” in der Bedingung der `if`-Abfrage gültig ist.

  Die Information bei der UD-Verkettung ist hier die Menge der gültigen Definitionen (Zuweisungen). Die Gleichungen ergeben sich folgendermaßen:

  - Für den Startblock B_1 gilt: $\text{in}(B_1) = \varnothing$ (keine gültige Definition)
  - Zu Beginn eines Block sind alle Definitionen irgendeines Vorgängers gültig:

    $$\text{in}(B_i) = \bigcup_{B_j \in \text{pred}(B_i)} \text{out}(B_j)$$

  - 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 von Definitionen in einem Block gilt, falls diese in *irgendeinem* Vorgängerblock gültig sind.

- **Lebendigkeitsanalyse** (vgl. Kapitel 7.5): Ziel dieser Analyse ist die Feststellung, welche Variablen an einer Stelle leben, d.h. welche Werte später noch benötigt werden. Die Information dieser Analyse ist also eine Menge von (lebendigen) Variablen.

  Sei B_n der letzte Block. Dann gelten die folgenden Gleichungen:

  $$\begin{aligned} \text{out}(B_n) &= \varnothing \\ \text{out}(B_i) &= \bigcup_{B_j \in \text{succ}(B_i)} \text{in}(B_j) \\ \text{in}(B_i) &= \text{gen}(B_i) \cup (\text{out}(B_i) \setminus \text{kill}(B_i)) \end{aligned}$$

  Hierbei enthält $\text{gen}(B)$ die Variablen des Basisblocks B, deren erstes Vorkommen in B als Operand ist (und nicht auf der linken Seite einer Zuweisung). $\text{kill}(B)$ sind die Variablen, die in B durch Zuweisungen definiert werden.

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

Das Berechnen von Lösungen zu den so aufgestellten Gleichungssystemen erfolgt durch eine Fixpunktiteration (vgl. Kapitel 7.5).

8.7. Abstrakte Interpretation

Die abstrakte Interpretation [Cousot/Cousot 77] ist eine Programmanalysetechnik, die sich an der operationalen Semantik (Interpreter) orientiert. Man rechnet hier jedoch mit abstrakten statt konkreten Werten. Notwendig ist hierzu eine abstrakte operationale Semantik. Dabei ist zu beachten, 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 bei der abstrakten Interpretation sind:

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

Hierdurch kann man eine semantisch korrekte Programmanalyse erhalten, wie dies in [Cousot/Cousot 77] gezeigt wurde.

Um einen kleinen Einblick in dieses große Gebiet zu geben, betrachten wir als einfaches Beispiel die **Vorzeichenanalyse**. Hierbei wollen wir analysieren, welche Vorzeichen Variablen haben.

Abstrakte Werte sind hierfür p (für Werte > 0), n (< 0) und z ($= 0$). Abstrakte Operationen sind hier markiert mit $\bar{}$. Es gilt dann z.B. für die abstrakte Addition:

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

Doch die Festlegung des Ergebnisses von $p \bar{+} n$ bereitet Probleme: welcher abstrakte Wert soll dies sein?

Als Lösung führen wir einen weiteren abstrakten Wert “?” ein, der für beliebige Werte steht. Nun können wir die abstrakten Operationen vollständig durch folgende Tabellen definieren:

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

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

Wir ersetzen nun noch die Konstanten, die im Programm vorkommen, durch abstrakte Werte und rechne dann mit abstrakten Werten bzw. abstrakten Operationen, bis ein

Fixpunkt erreicht ist (daher der Name „abstrakte Interpretation“).

Für jede Analyse ist zu zeigen, dass die abstrakten Operationen die konkreten Operationen korrekt approximieren. Falls dies der Fall ist, dann sind die Ergebnisse der abstrakten Interpretation beweisbar korrekt. Zum Beispiel könnten wir dann mit der Vorzeichenanalyse beweisen, dass die Definition einer Fakultätsfunktion nur positive Werte liefert.

Ausblick

Wir haben in dieser Vorlesung einen Überblick zu den wichtigsten Techniken der Übersetzung von Programmiersprachen gegeben. Vieles konnte aber nicht betrachtet werden, wie z.B. die Übersetzung spezieller Programmiersprachen: funktionale, logische, objektorientierte Sprachen.

Generell können zu diesem Zweck viele der hier vorgestellten Techniken übertragen werden. Darüber hinaus benötigt man zusätzlich spezielle Implementierungstechniken für diese Sprachen, wie z.B.

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

A. 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
- kompakter Quellcode, kürzere Entwicklungszeiten, lesbarere Programme
- modularer Programmaufbau, Polymorphie, Funktionen höherer Ordnung → Wiederverwendbarkeit

Nachteil ist die Effizienz der Implementierung, aber hier wurden in den letzten Jahren große Verbesserungen erreicht.

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[1])

A.1. Funktionsdefinitionen

Funktionen werden definiert durch ein Anweisung wie:

```
f x1 ... xn = e
```

Im Rumpf können vorkommen:

- Zahlen: `3`, `3.1415`

[1] Church (1941)

- Basisoperationen: `3+4`, `5*7`
- Funktionsanwendungen: `(f` e_1 `...` e_n`)`
- bedingte Ausdrücke: `if b then` e_1 `else` e_2

Beispiele:

- Eine Quadratfunktion definiert man durch

  ```
  square x = x * x
  ```

  Ablauf in Hugs/GHCi:

  ```
  > :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
```

Eine 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
```

A.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 es mit `1:2:3:[]`, durch `[1,2,3]` oder bei Listen von Zahlen: `[a..b]` sowie `[a..]`. Viele Operationen auf Listen sind vordefiniert in den Standardmodulen `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.

`[1,2]++[3,4]` $\equiv$ `(++) [1,2] [3,4]` $\equiv$ `[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 einer 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[2], 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

`add 3 4` $\equiv$ `((add 3) 4)` $\implies$ `(add 3)::Int->Int`

A.3. Selbstdefinierte bzw. algebraische Datenstrukturen

Konstrukturen zum Aufbau algebraischer Datentypen sind frei interpretierte Funktionen (nicht reduzierbar, z.B. `[]` und `(:)`). Datendefinition:

$\texttt{data}\ t\ \texttt{=}\ \texttt{c}_1\ \texttt{t}_{11}\ \ldots\ \texttt{t}_{1n_1}\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{c}_k\ \texttt{t}_{k1}\ \ldots\ \texttt{t}_{kn_k}$

Dies führt einen neuen Typ t ein mit k Konstruktoren mit $\texttt{c}_j\texttt{::t}_{j1}\texttt{->}\ldots\texttt{->t}_{jn_j}\ \texttt{->t}$.

Sowohl Namen von Typen als auch Konstruktoren müssen mit Großbuchstaben beginnen. Beispiel:

```
data ListInt = Nil | Cons Int ListInt
```

Damit ist `Nil::ListInt` und `Cons::Int->ListInt->ListInt`.

Allgemeine Listen: `[a]` sind polymorph bezüglich der Elemente:

```
data [a] = [] | a:[a]
```

[2] nach Schönfinkel und Curry

Dabei ist `a` eine Typ-Variable. Vorteil dieser Polymorphie: Der Code wird wiederverwendbar, wir benötigen keine einzelnen Funktionen auf speziellen Typen wie `++Char`, `++Int`, `++[Int]` ...

Beispielsweise funktionieren die folgenden Funktionen auf allen Listen:

```
length::[a]→Int
(++)::[a]→[a]→[a]
```

Z.B. liefert die folgende Funktion das letzte Element einer beliebigen Liste:

```
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)

-- Typ von 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 der allgemeinen Datentypdefinition sind:

- Aufzählungstypen:

  ```
  data Color = Red | Green | Blue
  ```

- Verbundstypen:

  ```
  data Complex = Complex Float Float
  ```

  Dabei hat der Name `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 gibt es folgende wichtige Klassen:

- In `Eq` sind Vergleiche 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 kann man Datentypen einer dieser Typklassen zuordnen und für die Funktionen von `Show` und `Eq` automatisch Code erzeugen zu lassen.

A.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 Muster `p` passt und `x` wird an entsprechenden 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. ist

```
eq x x = True
```

unzulässig, da nicht unbedingt ein Vergleich für beide Vorkommen der formalen Parameter 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 Haskell 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.

A.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

```
derive :: (Float→Float) → (Float→Float)
derive f = f'
  where f' x = (f(x+dx)-f x)/dx
        dx   = 0.0001

> derive sin 0.0
1.0
> derive (\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`. Daher ist es sinnvoll, diesen Unterschied als Parameter zu 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 einer Liste 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
```

A.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.

A.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 *leftmost outermost*:

- Alles, was durch irgendeine Reduktion berechnet werden kann, wird berechnet.
- Vermeidung überflüssiger Berechnungen (z.B. `h` $\rightarrow$ `h` $\rightarrow$ `h` $\rightarrow$ `h` $\rightarrow$ ...)
- Rechnen mit unendlichen Datenstrukturen möglich
- zur Effizienz: betrachte

  ```
  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 für das 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)
```

Berechnung der ersten zehn Primzahlen: `take 10 primes`

A.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 ist es wegen der Lazy-Auswertung noch schwerer nachzuvollziehen, wann die Ausgabe erfolgt!

Lösung: Monade (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'`. Weitere Definition:

```
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 zum Rechnen mit I/O-Aktionen:

```
head [putStr "Hallo", putStr "Ihr"]
```

In der IO-Monade gelten folgende Gesetze:

```
return () >> m = m
m >> return () = m
m >> (n >> o) = (m >> n) >> o
```

Damit ist "`return ()`" 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 Monad 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 es mit einer Monade. Idee hierbei: 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)
   []     >>= 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 Haskell98 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 Comprehensions sind eine kompakte Notation für Listen, deren Elemente aus anderen Liste über eine Bedingung ausgewählt werden. Damit kann man z.B. die mathematische Notation

$$\{(i,j) \mid i \in \{1,\ldots,3\}, j \in \{2,\ldots,4\}, i \neq j\}$$

fast direkt umsetzen in die Haskell-Notation

```
[(i,j) | i <- [1..3], j <- [2..4], i != j]
```

Die Auswertung dieses Ausdrucks ergibt in Haskell die Liste

```
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,2),(3,4)]
```

Literaturverzeichnis

[Cousot/Cousot 77]
P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pp. 238–252, 1977.

[Güting/Erwig 99]
R.H. Güting and M. Erwig. *Übersetzerbau: Techniken, Werkzeuge, Anwendungen.* Springer-Verlag, Berlin, Heidelberg, New York, 1999.

[Kahn/Carlsson 84]
K.M. Kahn and M. Carlsson. The Compilation of Prolog Programs without the Use of a Prolog Compiler. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pp. 348–355. ICOT, 1984.

Index