Übersetzerbau

fldit-www.cs.uni-dortmund.de/∼peter/Cbau.pdf

Wintersemester 2008/2009

Thursday 3rd September, 2015

Peter Padawitz

Fachbereich Informatik
TU Dortmund

Inhaltsverzeichnis

Kapitel 1

Übersetzer und Haskell

Inhalt der Lehrveranstaltung sind Konzepte und Methoden der Übersetzung **imperativer** und **funktionaler Programmiersprachen**. Der Weg vom Quell- zum Zielprogramm durchläuft mehrere Phasen. Die **lexikalische Analyse** transformiert eine Zeichen- in eine Symbolfolge. **Syntaxanalyse** überführt die Symbolfolge in eine baumartige Termstruktur (Programm in **abstrakter Syntax**) und macht so übersetzungsrelevante Teile der Bedeutung des Programms sichtbar. **Semantische Analyse** attributiert die Termstruktur mit semantischen Informationen, die der Übersetzer benötigt. *Partielle Auswertung, Striktheitsanalyse*, usw. optimieren das Programm durch Veränderungen der Termstruktur. **Codeerzeugung** bildet die Termstruktur in das Zielprogramm ab, das i.a. eine lineare Befehlsfolge ist.

Wir wollen uns auf einige Konzepte und Algorithmen konzentrieren, die für jeweils eine der genannten Phasen grundlegend sind. Algorithmen werden hier oft in der **funktionalen Sprache Haskell** (haskell.org) formuliert, weil sie i.a. Verfahren zur **Symbol-** und **Termmanipulation** sind und deshalb in einer funktionalen, typorientierten Sprache ohne die sonst üblichen Implementierungsdetails am besten dargestellt werden können. Abschnitt 1.2 führt in die wichtigsten hier benötigten Konzepte von Haskell ein.

In Kapitel 5 wird mit diesen Mitteln die Erzeugung von Assemblercode für eine Pascal-ähnliche imperative Sprache beschrieben. Im Rahmen der Übersetzung funktionaler Sprachen (Kapitel 7) behandeln wir Algorithmen zur **Typinferenz**, zur Transformation funktionaler Programme in **λ-Ausdrücke**, in **Continuations** und dann in Assemblercode. Klassische Interpreter funktionaler Sprachen werden in Kapitel 8 vorgestellt. Allen hier behandelten Quellsprachen liegen **kontextfreie Grammatiken** zugrunde. Dementsprechend werden Assembler-Zielsprachen durch **Kellermaschinen** interpretiert. Das kommt auch daher, dass Assemblerprogramme Befehls*folgen* sind, während Quellprogramme (in abstrakter Syntax) eine *baumartige* Struktur haben. Eine Alternative zur Ausführung von Assemblercode ist die direkte Auswertung der abstrakten Programme. Wegen deren Baumstruktur sind dafür jedoch aufwendigere Datenstrukturen, Such- und Zugriffsalgorithmen (*heaps, pointer*) erforderlich. In Kapitel 7 und 8 werden wir auch solche Interpreter ansprechen.

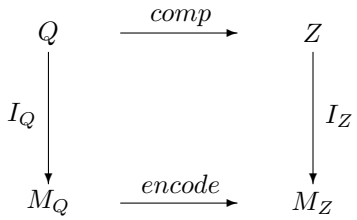
Die Implementierung **logischer Programmiersprachen** wie Prolog wird hier nicht behandelt. Deren Details findet man z.B. in [24] und [85]. Prinzipiell unterscheiden sich logische *Programme* kaum von funktionalen. Beide arbeiten i.w. mit *statischen* Datenstrukturen. Der entscheidende Unterschied liegt in der Ausführung. Funktionale und im Prinzip auch imperative Programme dienen der *Auswertung* funktionaler Ausdrücke, logische Programme hingegen der *Lösung* von Gleichungen oder anderen Formeln *in logischen Variablen* (vgl. [56]). Auch die Besonderheiten **objektorientierter Sprachen**—mit ihrem Schwergewicht auf *dynamischen, kommunizierenden* Datenstrukturen—werden hier nicht behandelt. Wegen ihres meist schwachen Typkonzeptes fügen sich ihre Compiler bisher (?) auch nur schwer in den mehrphasigen Ansatz ein, dessen Zwischensprache auf *statischen* Termstrukturen aufbaut (s.o.).

Optimierungen lassen sich sowohl auf den abstrakten Programmen der Zwischensprache durchführen (im sog. *front end* des Compilers) als auch auf dem Zielcode (im *back end*). Zu ersteren gehören die in Kapitel 6 angesprochenen **Programmtransformationen** sowie **abstrakte Interpretationen**, die gewisse Teile aus der Gesamtbedeutung eines Programms herausfiltern und genau diese als "Zielfunktionen" der Optimierung betrachten. Back-end-Optimierung setzt **Kontroll-** und **Datenflußanalyse** voraus, die auf zustandsorientierten dynamischen Modellen des Zielcodes (z.B. Flußgraphen) basiert.

1.1 Übersetzer: Funktion und Aufbau

Ein **Übersetzer** (**Compiler**) ist eine Folge von Programmen, die schrittweise ein Programm einer **Quellsprache** in ein *semantisch äquivalentes* Programm einer **Zielsprache** transformiert. I.a. ist die Quellsprache benutzerfreundlich, leicht modifizierbar, enthält komplexe Konstrukte, während die Zielsprache i.a. schlecht lesbar und kaum modifizierbar ist und nur einfache Konstrukte enthält. Die Zielsprache soll weniger von Menschen als von Maschinen verstanden werden.

Definition 1.1.1 Seien Q und Z die Mengen der Programme der Quell- bzw. Zielsprache, M_Q und M_Z mathematische Strukturen, die die Bedeutung (Semantik) von Q bzw. Z wiedergeben, $I_Q : Q \to M_Q$ und $I_Z : Z \to M_Z$ **Interpretationsfunktionen** (*evaluation functions*), die jedem Programm seine Bedeutung in dem jeweiligen semantischen Bereich zuordnet. Eine Funktion $comp : Q \to Z$ heißt **Compiler** von Q nach Z, wenn es eine Funktion $encode : M_Q \to M_Z$ gibt derart, dass folgendes Diagramm kommutiert



Die Mengen M_Q und M_Z werden oft als **abstrakte Maschinen** formuliert. Sie bestehen dann aus **Zustandstransformationen**, d.h. Funktionen auf einer Zustandsmenge S (*states*). Ein Zustand besteht in der Regel aus mehreren Komponenten. Dazu gehören *Eingabekomponenten* (E), interne Attribute und *Ausgabekomponenten* (A). Dann gibt es auch eine Einbettung (injektive Funktion) $input : E \to S$ und symmetrisch dazu eine Projektion (surjektive Funktion) $output : S \to A$.

Definition 1.1.2 Sei $M_Q = [S \to S]$ (Menge der Funktionen von S nach S). Eine Funktion $eval : Q \times E \to A$ heißt **Interpreter** von Q, wenn für alle $q \in Q$ und $e \in E$

$$eval(q, e) = output(I_Q(q)(input(e)))$$

gilt. ❑

Manchmal werden Quell- und/oder Zielprogramme selbst als Zustandskomponenten aufgefasst und die gesamten Zustände **Konfigurationen** genannt.

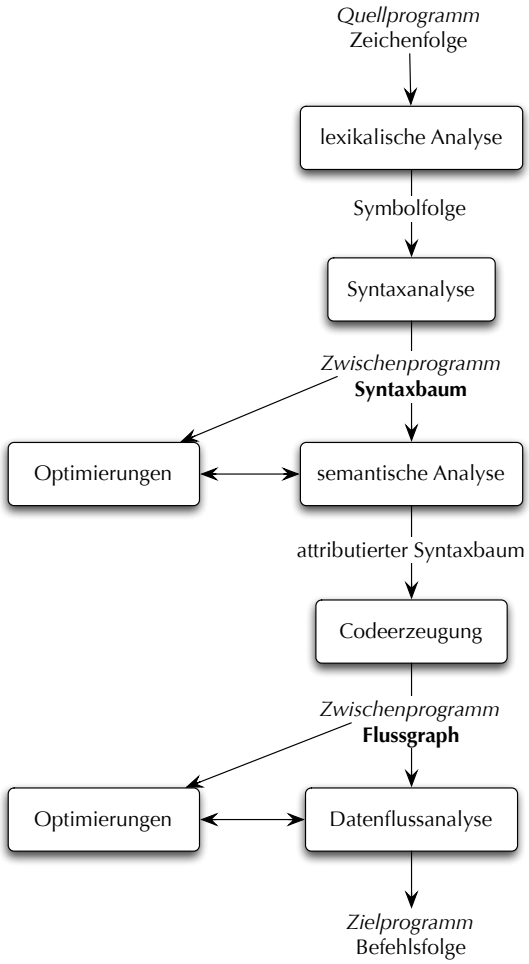


Figure 1.1. *Aufbau eines Übersetzers*

1.2 Einführung in Haskell und zwei *running examples*

Als Implementierungssprache für Algorithmen werden wir häufig die funktionale Programmiersprache Haskell benutzen. Wir beschränken uns hier auf die Konstrukte von Haskell, die in späteren Algorithmen benutzt werden. *Basistypen* sind **Bool, Int, Float, Double, Char, String** und **() (unit)**. Basisfunktionen für **Bool** sind `&&`, `||`, für **Int** die arithmetischen Operationen sowie Vergleichsoperationen. Strings sind in " eingeschlossene Zeichenfolgen. Der String "abcd" entspricht der *Liste* ['a','b','c','d'] von Zeichen, der leere String "" der leeren Liste []. ++ bezeichnet die Stringkonkatenation. Die Funktion *length* liefert die Länge eines Strings. Der unit-Typ besteht aus genau einem Element, das wie der Typ selbst mit **()** bezeichnet wird. Man verwendet den unit-Typ als Wertebereich von Funktionen, die eigentlich gar keine Werte liefern, sondern nur Zustandstransformationen bewirken (siehe §3.6).

An *zusammengesetzten Typen* verwenden wir Produkttypen, z.B. **(Int,Bool)**. $(5, true)$ ist ein Element dieses Typs. Benennt man die Komponenten eines Tupels $(a_1, ..., a_n)$, dann entsteht ein *Record*. Z.B. hat der Record `{x=5,y=True}` den Typ **{x::Int,y::Bool}**.[1] Die Namen der Komponenten eines Rekords heißen *Feldnamen* oder *Attribute*. Zum Zugriff auf Komponenten eines Records wenden wir das jeweilige Attribut auf den Record an. Z.B. liefert $x\{x = 5, y = True\}$ den Wert 5. Produkt- und Recordtypen können beliebig geschachtelt werden.

Weiterhin benutzen wir Listentypen, z.B. **[Int]**. [1,3,4,23] ist ein Element dieses Typs. [] bezeichnet die leere

[1] Diese Syntax entstammt der objektorientierten Haskell-Erweiterung O'Haskell [53].

Liste. Eine Basisfunktion auf Listen ist die *append*-Funktion

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

Z.B. ist der Ausdruck $12 : [1, 3, 4, 23]$ semantisch äquivalent zur Liste [12,1,3,4,23]. Hier bezeichnet a eine **Typvariable**, die durch einen beliebigen Basis- oder zusammengesetzten Typ substituiert werden kann. Ein Typ, der eine Typvariable enthält, heißt **polymorpher Typ**. Die Funktionen $head, tail, length$ liefern den Kopf, Rest bzw. die Länge einer Liste. Eine wichtige Rolle spielen **funktionale Typen** wie z.B. $Int \to Bool$. Damit lassen sich **Funktionen höherer Ordnung** definieren (s.u.). Der Funktionspfeil $\to$ wird als *rechtassoziative* binäre Operation auf Typen aufgefaßt, d.h. anstelle von $a \to (b \to c)$ schreiben wir $a \to b \to c$.

Ein Haskell-Programm besteht im wesentlichen aus einer Folge von Gleichungen der Form

```
f p1 ... pk | be = let q1 = e1          f p1 ... pk | be = e0
                       ...       oder                where q1 = e1
                       qn = en                             ...
                  in e0                                    qn = en
```

Die Gleichung definiert die Funktion f für alle Argumente, die zum Datenmuster $(p_1, \ldots, p_k)$ (s.u.) passen und die Bedingung (Boolescher Ausdruck) be erfüllen. Auch $q_1, \ldots, q_n$ sind Datenmuster. Jede Gleichung $q_i = e_i$ definiert lokale Variablen, deren Gültigkeitsbereich aus den Ausdrücken $e_0, e_1, \ldots, e_n$ (bei `let ... in`) bzw. $be, e_0, e_1, \ldots, e_n$ (bei `where ...`) besteht. Rekursive Aufrufe von f können in $e_0, e_1, \ldots, e_n$ auftreten. Beispiele folgen weiter unten.

Fallunterscheidungen können also mit Hilfe von Datenmustern (wie $p_1, \ldots, p_k$) und Booleschen Ausdrücken (wie be) formuliert werden, letztere auch – wie sonst üblich – im Kontext von Konditionalen: `if be then e1 else e2`. Hier ist *if-then-else* eine (implizite) Funktion vom Typ `(Bool,a,a) -> a`.

Datenmuster (**patterns**) sind aus Variablen und **Konstruktoren** bestehende Ausdrücke. Konstruktoren sind Konstanten und Funktionen, mit denen Daten aufgebaut werden. Beispiele sind Zahl-, String- und Boolesche Konstanten, die Listenkonstante [], die Listenfunktion : und die Tupelbildung mit Kommas und runden Klammern (s.o.). Z.B. stellt die Boolesche Funktion *null :: [a] -> Bool* fest, ob eine Liste leer ist oder nicht, indem sie auf dem Muster [] den Wert $True$ liefert, während sie auf allen nichtleeren Listen, d.h. auf allen Listen, die zum Muster x:s passen, den Wert $False$ zurückgibt. Gleichungen für verschiedene Argumentmuster derselben Funktion werden bei der Anwendung der Funktion von oben nach unten durchlaufen. Deshalb kann man *null* wie folgt definieren:

```
null (_:_) = False
null _     = True
```

Der Unterstrich ist eine Variable, die auf der rechten Seite von Definitionsgleichungen nicht verwendet werden kann. Seine Verwendung auf der linken Seite einer Gleichung macht deutlich, von welchen Argumenten der Wert der definierten Funktion in dem durch die Gleichung beschriebenen Fall unabhängig ist.

Die Listenfunktionen

```
head :: [a] -> a                    tail :: [a] -> [a]
head (a:_) = a                      tail (_:s) = s

init :: [a] -> [a]                  last :: [a] -> a
init [_]   = []                     last [a]   = a
```

```
init (a:s) = a:init s                  last (_:s) = last s

take :: Int -> [a] -> [a]              drop :: Int -> [a] -> [a]
take 0 _             = []              drop 0 s             = s
take n (x:s) | n > 0 = x:take (n-1) s  drop n (_:s) | n > 0 = drop (n-1) s
take _ []            = []              drop _ []            = []

(!!) :: [a] -> Int -> a
(a:_)!!0         = a
(_:s)!!n | n > 0 = s!!(n-1)

zip :: [a] -> [b] -> [(a,b)]
zip (a:s) (b:s') = (a,b):zip s s'
zip _ _          = []

sublist :: [a] -> Int -> Int -> [a]
sublist (a:_) 0 0                  = [a]
sublist (a:s) 0 j | j > 0          = a:sublist s 0 (j-1)
sublist (_:s) i j | i > 0 && j > 0 = sublist s (i-1) (j-1)
sublist _ _ _                      = []

repeat :: a -> [a]
repeat a = a:repeat a

replicate :: Int -> a -> [a]
replicate n a = take n (repeat a)
```

liefern einzelne Elemente bzw. Teillisten einer Liste.[2] Wie der Konstruktor (:) werden auch der Zugriff (!!) auf einzelne Listenelemente (s.o) und die *Listenkonkatenation*

```
(++) :: [a] -> [a] -> [a]
(a:s)++s' = a:(s++s')
_++s      = s
```

infix verwendet, d.h. zwischen ihre beiden Argumente geschrieben. Runde Klammern werden um ein Infixsymbol geschrieben, wenn es präfix verwendet wird. So wäre auch folgende Definition von (++) syntaktisch korrekt:

```
(++) (a:s) s' = a:(++) s s'
(++) _ s      = s
```

Hier können die runden Klammern um den rekursiven Aufruf von (++) entfallen, weil präfix-verwendete Funktionen eine höhere Priorität als infix-verwendete (hier der Konstruktor :).

Jede Funktion eines Typs $a \to b \to c$ kann man präfix oder infix verwenden. Allerdings schreibt man sie je nach Verwendung unterschiedlich: Aus Sonderzeichen zusammengesetzte Funktionssymbole wie ++ werden bei Präfixverwendung in runde Klammern gesetzt, während mit einem (Klein-)Buchstaben beginnende Funktionssymbole wie *mod* in Hochkommas ‘ eingeschlossen werden.

[2] Bis auf *sublist* sind dies alles Standardfunktionen.

Mit einem Großbuchstaben beginnende Zeichenfolgen interpretiert Haskell grundsätzlich als Typen, Typklassen bzw. Konstruktoren!

Mit einem Kleinbuchstaben beginnende Zeichenfolgen interpretiert Haskell grundsätzlich als Typ- bzw. Elementvariablen!

Hier noch einige grundlegende Listenfunktionen und ein paar ihrer Anwendungen:

Der Typkonstruktor `->` ist standardmäßig eine rechtsassoziative Funktion. Deshalb steht `a1 -> a2 -> ... a(n-1) -> an` für `a1 -> (a2 -> ... (a(n-1) -> an) ... )`. *map* wendet eine Funktion $f : a \to b$ auf jedes Element einer Liste an und liefert die Liste der Funktionswerte:

```
map :: (a -> b) -> [a] -> [b]
map f (a:s) = f a:map f s
map _ _     = []
```

zipWith wendet eine Funktion $f : a \to b \to c$ auf Paare von Elementen einer Liste vom Typ $[a]$ bzw. $[b]$ und liefert ebenfalls die Liste der Funktionswerte:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:s) (b:s') = f a b:zipWith f s s'
zipWith _ _ _          = []
```

foldl faltet eine Liste s zu einem Element durch wiederholte linksassoziative Anwendung einer Funktion $f : a \to b \to a$, beginnend mit einem festen Anfangselement a:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f a (b:s) = foldl f (f a b) s
foldl _ a _     = a
```

foldl f a s entspricht einer *for*-Schleife in imperativen Sprachen:

```
state = a;
for (i = 0; i < length s; i++) {state = f state (s!!i);}
return state
```

Einige Instanzen von *foldl*:

```
sum       :: [Int] -> Int
product   :: [Int] -> Int
concat    :: [[a]] -> [a]
concatMap :: (a -> [b]) -> [a] -> [b]

sum       = foldl (+) 0
product   = foldl (*) 1
and       = foldl (&&) True
or        = foldl (||) False
concat    = foldl (++) []
concatMap = concat . map
```

Umgekehrt bewirkt `foldr` die rechtsassoziative Anwendung einer Funktion $f : a \to b \to b$ auf eine Liste:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b (a:s) = f a (foldr f b s)
foldr _ b _     = b
```

foldr f a s entspricht einer *for*-Schleife mit Dekrementierung der Laufvariablen:

```
state = a;
for (i = length s-1; i >= 0; i--) {state = f (s!!i) state;}
return state
```

Soll das Anfangselement mit dem Kopf der Liste übereinstimmen, dann können nur nichtleere Liste verarbeitet werden:

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x:s) = foldl f x s

foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]   = x
foldr1 f (x:s) = f x (foldr1 f s)
```

any und *all* implementieren die Quantoren auf Listen, in dem sie prfen, ob die Boolesche Funktion $f : a \to Bool$ für ein bzw. alle Elemente einer Liste *True* liefert:

```
any :: (a -> Bool) -> [a] -> Bool
any f = or . map f

elem :: a -> [a] -> Bool
elem = any (a ==)

all :: (a -> Bool) -> [a] -> Bool
all f = and . map f

notElem :: a -> [a] -> Bool
notElem = all (a /=)
```

filter erzeugt die Teilliste aller Elemente einer Liste, für die die Boolesche Funktion $f : a \to Bool$ *True* liefert:

```
filter :: (a -> Bool) -> [a] -> [a]
filter f (x:s) = if f x then x:filter f s else filter f s
filter f _     = []
```

Der Aufruf *filter(f)(s)* lässt sich auch als **Listenkomprehension** schreiben:

```
filter f s = [x | x <- s, f x]
```

primes(n) filtert z.B. alle Primzahlen aus der Liste aller ganzen Zahlen zwischen 2 und n (*Sieb des Eratosthenes*):

```
primes :: Int -> [Int]
primes n = sieve [2..n]
```

```
        where sieve :: [Int] -> [Int]
              sieve (x:s) = x:sieve [y | y <- s, y `mod` x /= 0]
              sieve _     = []
```

Hier taucht die zweistellige Funktion *mod* auf, die im Gegensatz zu (:), (++) und (!!) aus einem Wortsymbol besteht. Auch eine solche Funktion kann sowohl infix als auch präfix verwendet werden. Im ersten Fall wird das Wortsymbol von dem Akzentsymbol ‘ begrenzt (‘mod‘), im zweiten entfallen die Akzentsymbole (mod).

Mehrere rekursive Aufrufe mit dem gleichen Parameter sollten mithilfe einer lokalen Definition immer zu einem zugefasst werden. So ist die zunächst naheliegende Definition einer Funktion *pascal* zur Berechnung der n-ten Zeile des *Pascalschen Dreiecks*:

```
pascal 0 = [1]
pascal n = 1:[pascal (n-1)!!(k-1)+pascal (n-1)!!k | k <- [1..n-1]]++[1]
```

sehr langsam, weil der doppelte Aufruf von $pascal(n-1)$ zu exponentiellem Aufwand führt. Mit einer lokalen Definition geht's gleich viel schneller und sieht's auch eleganter aus:[3]

```
pascal 0 = [1]
pascal n = 1:[s!!(k-1)+s!!k | k <- [1..n-1]]++[1]
           where s = pascal (n-1)
```

In dieser Version gibt es zwar keine überflüssigen rekursiven Aufrufe mehr, aber noch eine Verdopplung fast aller Listenzugriffe: Für alle $k \in \{1, \ldots, n-2\}$ wird $s!!k$ zweimal berechnet. Das lässt sich vermeiden, indem wir aus den Summen einzelner Listenelemente eine Summe zweier Listen machen:

```
pascal 0 = [1]
pascal n = zipWith (+) (s++[0]) (0:s)
           where s = pascal (n-1)
```

valid(n)(f) prüft die Gültigkeit einer Aussage, dargestellt als n-stellige Boolesche Funktion f:

```
valid :: Int -> ([Bool] -> Bool) -> Bool
valid n f = and [f vals | vals <- args n]
            where args :: Int -> [[Bool]]
                  args 0 = [[]]
                  args n = [True:vals | vals <- args']++[False:vals | vals <- args']
                           where args' = args (n-1)
```

Wie man schon an *init* und *last* (s.o.) sehen kann, lassen sich nicht alle Funktionen auf Listen mit Gleichungen für die beiden Muster [] (leere Liste) bzw. x:s (nichtleere Liste) definieren. So wird z.B. zur Definition der Booleschen Funktion $sorted : [a] \to Bool$, die feststellt, ob eine Liste aufsteigend sortiert ist, erst im Fall einer mindestens zweielementigen Liste ein rekursiver Aufruf benötigt:

```
sorted :: [Int] -> Bool
sorted (x:s@(y:_)) = x <= y && sorted s
sorted _           = True
```

Ebenso benötigen Sortieralgorithmen erst auf mindestens zweielementigen Listen einen rekursiver Aufruf:

[3] Besonders als *where*-Klausel. Die Verwendung von *let*-Klauseln ist oft umständlicher und manchmal auch ineffizienter.

```
quicksort :: [Int] -> [Int]
quicksort (x:s@(_:_)) = quicksort[y | y <- s, y <= x]++x:
quicksort[y | y <- s, y > x]
quicksort s           = s
```

Beispiel 1.2.1 Erkennung von Geraden Die Boolesche Funktion $straight : [(Float, Float)]] \to Bool$, die festellt, ob alle Elemente einer Punktliste auf einer Geraden liegen, setzt die Rekursion sogar erst auf mindestens dreielementigen Listen ein:

```
straight :: [(Float,Float)] -> Bool
straight (p:s@(q:r:_)) = straight3 p q r && straight s
straight _             = True

straight3 :: (Float,Float) -> (Float,Float) -> (Float,Float) -> Bool
straight3 p@(x1,_) q@(x2,_) r@(x3,_) | x1 == x2 = x2 == x3
                                     | x2 == x3 = x1 == x2
                                     | True     = coeffs p q == coeffs q r
                      where coeffs (x,y) (x',y') = (a,y-a*x) where a = (y'-y)/(x'-x)
```

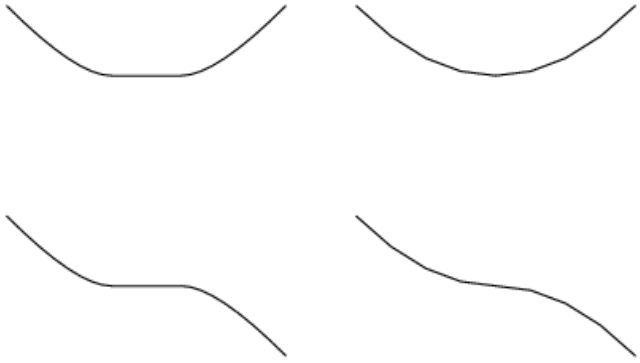


Figure 1.2. *Zwei geglättete Kantenzüge vor und nach der Reduzierung*

reduce reduziert eine Punktliste derart, dass niemals drei aufeinanderfolgende Punkte auf einer Geraden liegen:

```
reduce :: [(Float,Float)] -> [(Float,Float)]
reduce (p:s@(q:r:s')) = if straight3 p q r then reduce (p:r:s') else p:reduce s
reduce s              = s
```

❑

flip ordnet die Argumente einer Funktion höherer Ordnung um (siehe *evalCom* in Beispiel 1.2.3):

```
flip :: (a -> b -> c) -> b -> a -> c
flip f b a = f a b
```

curry und *uncurry* wechseln zwischen der mehrstelligen Version einer Funktion und ihrer **kaskadierten** oder **curryfizierten** einstelligen Version hin bzw. her:

```
curry :: ((a,b) -> c) -> a -> b -> c
```

```
curry f a b = f (a,b)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b) = f a b
```

Die obigen Beispiele für Funktionsdefinitionen folgen einem bestimmten Schema:

$$\begin{aligned} &f\ p_{11}\ ...\ p_{1n} = e_1 \\ &\vdots \\ &f\ p_{k1}\ ...\ p_{kn} = e_k \\ &f\ \_\ ...\ \_ \quad\ = e_{k+1} \end{aligned}$$

Der Wert einer Funktion f ergibt sich aus dem Ausdruck auf der rechten Seite der ersten Definitionsgleichung von f, auf deren Argumentmuster $p_{i1}\ ...\ p_{in}$ der aktuelle Funktionsparameter passt, wobei links vom Gleichheitszeichen eine Bedingung dazukommen kann, die das Argument erfüllen muss, und rechts lokale Definitionen (wie in dem am Anfang dieses Abschnitts angegebenen Schema einer einzelnen Gleichung). Eine semantisch äquivalente Definition von f erhalten wir mit dem case-Konstrukt:

$$\begin{aligned} f\ x_1, \ldots, x_n = \text{case } (x_1, \ldots, x_n) \text{ of } &(p_{11}, ..., p_{1n}) \to e_1 \\ &(p_{21}, ..., p_{2n}) \to e_2 \\ &\quad\vdots \\ &(p_{k1}, ..., p_{kn}) \to e_k \\ &\_ \to e_{k+1} \end{aligned}$$

Zwischen der Fallunterscheidung über mehrere Gleichungen und der Beschränkung auf eine Gleichung mit case-Konstrukt auf der rechten Seite gibt es in der Regel äquivalente Mischformen, die einer konkreten Funktion meist angemessener sind als die beiden Extremfälle. $x_1, \ldots, x_n$ können beliebige Muster sein. p_{ij} sollte dann eine Instanz von x_i sein. Schließlich gibt es noch die Möglichkeit, Funktionen durch λ-*Abstraktionen* zu definieren. Damit sähe das zweite Schema folgendermaßen aus:

$$\begin{aligned} f = \lambda\ x_1, \ldots, x_n \to \text{case } (x_1, \ldots, x_n) \text{ of } &(p_{11}, ..., p_{1n}) \to e_1 \\ &(p_{21}, ..., p_{2n}) \to e_2 \\ &\quad\vdots \\ &(p_{k1}, ..., p_{kn}) \to e_k \\ &\_ \to e_{k+1} \end{aligned}$$

Man benutzt λ-Abstraktionen eigentlich nur, wenn man die durch sie repräsentierten Funktionen nicht extra benennen will, weil man sie nur einmal in einem bestimmten Kontext benutzt. Bei der Definition funktionaler Attribute von Datentypobjekten (siehe §3.3) können λ-Abstraktionen allerdings sinnvoll sein, weil ein Attribut f stets **nicht-applikativ**, also durch eine Gleichung der Form $f = e$ definiert werden muss.

Konstruktorbasierte Datentypen

Funktionen, die keinen anderen Zweck erfüllen als Daten aufzubauen, heißen **Konstruktoren**. Im Übersetzerbau dienen sie der Implementierung einer abstrakten Syntax (siehe Abschnitt 3.2). Zur Definition von Konstruktoren und durch sie beschriebener Typen stellt Haskell das `data`-Konstrukt zur Verfügung. Das Schema einer solchen Typdefinition lautet wie folgt:

```
data New a1 ... am = Constructor_1 e11 ... e1n1 | ... |
                     Constructor_k ek1 ... eknk
```

`e11,...,eknk` sind beliebige Typausdrücke, die aus Typkonstanten (z.B. `Int`), **Typvariablen** (nur `a1,...,am`) und **Typkonstruktoren** (Produktbildung, Listenbildung, Datentypen) zusammengesetzt sind. Kommt `New` selbst in einem dieser Typausdrücke vor, spricht man von einem **rekursiven Datentyp**.

Die Elemente von `New a1 ... am` sind alle funktionalen Ausdrücke, die aus den Konstruktoren `Constructor_1,...,Constructor_k` und Elementen von Instanzen von `a1,...,am` zusammengesetzt sind. Als Funktion hat `Constructor_i` den Typ `ei1 -> ... -> eini -> New a1 ... am`.

Da Konstruktoren ihre Argumente nicht verändern, sondern nur kapseln, möchte man oft auf diese direkt zugreifen. Das erreicht man durch Einführung von (auch Felder genannten) **Attributen**, eins für jede Argumentposition des Konstruktors. In der Definition von `New` wird `Constructor_i ei1 ... eini` durch

```
Constructor_i {attribute_i1 :: ei1, ..., attribute_ini :: eini}
```

ersetzt. Z.B. lassen sich die aus anderen Programmiersprachen bekannten **Recordtypen** und **Objektklassen** als Datentypen mit genau einem Konstruktor, aber mehreren Attributen implementieren.

Wie ein Konstruktor, so ist auch ein Attribut eine Funktion. Als solche hat z.B. `attribute_ij` den Typ `attribute_ij :: New a1 ... am -> eij`.

Attribute sind also invers zu Konstruktoren. Man nennt sie deshalb heute auch **Destruktoren**. Z.B. hat `attribute_ij (Constructor_i a1 ... ani)` den Wert `aj`. `Constructor_i a1 ... ani` ist äquivalent zu:

```
Constructor_i {attribute_i1 = a1, ..., attribute_ini = ani}.
```

Einer der einfachsten und häufig verwendeten Datentypen

```
data Maybe a = Just a | Nothing,
```

dient u.a. der Totalisierung partieller Funktionen. `Nothing` steht dabei für "undefiniert", während `Just` "definierte" Werte kapselt. Die totalisierte Version einer Funktion `f::a->b` hat dann den Typ `a -> Maybe b` (siehe auch §3.6).

Die folgenden Beispiele zeigen Anwendungen im Bereich der Auswertung und Übersetzung symbolischer Ausdrücke und, allgemeiner, der Übersetzung von Programmen in *abstrakter Syntax* (siehe §3.2). Jeder Datentyp- bzw. Funktionsdefinition folgt die Angabe der Typen der Konstruktoren bzw. der Funktion, so wie sie ein Haskell-Compiler durch **Typinferenz** (siehe §6.1) aus der jeweiligen Definition errechnet.

Beispiel 1.2.2 1. running example: Ein Datentyp für arithmetische Ausdrücke

```
data Expr = Con Int | Var String | Sum [Expr] | Prod [Expr] | Expr :- Expr |
            Int :* Expr | Expr :^ Int

zero = Con 0
one  = Con 1
```

Z.B. lautet der Ausdruck 5*(x+2+3) als Objekt vom Typ *Expr* folgendermaßen:

```
Prod[Con 5,Sum [Var"x",Con 2,Con 3]]
```

Die folgende Funktion zweiter Ordnung (s.o.) implementiert **symbolische Differentiation** in Abhängigkeit vom Muster der Ausdrücke:

```
diff :: String -> Expr -> Expr
diff x (Con _)   = zero
diff x (Var y)   = if x == y then one else zero
diff x (Sum es)  = Sum (map (diff x) es)
diff x (Prod es) = Sum (map f [0..length es-1])
                   where f i = Prod (updList es i (diff x (es!!i)))
diff x (e :- e') = diff x e :- diff x e'
diff x (n :* e)  = n :* diff x e
diff x (e :^ n)  = n :* Prod [diff x e,e:^(n-1)]

updList :: [a] -> Int -> a -> [a]
updList s i a = take i s++a:drop (i+1) s
```

Beispiel 1.2.3 2. running example: Ein Datentyp für arithmetische oder Boolesche Ausdrücke und Kommandos

```
type Block   = [Command]
data Command = Skip | Assign String IntE | Cond BoolE Block Block | Loop BoolE Block
data IntE    = IntE Int | Var String | Sub IntE IntE | Sum [IntE] | Prod [IntE]
data BoolE   = BoolE Bool | Greater IntE IntE | Not BoolE
```

Das imperative Programm

```
{fact = 1; while (x > 0) {fact = fact*x; x = x-1;}}
```

wird z.B. dargestellt durch den Ausdruck

```
prog = [Assign "fact" (IntE 1),
        Loop (Greater (Var "x") (IntE 0))
             [Assign "fact" (Prod [Var "fact",Var "x"]),
              Assign "x" (Sub (Var "x") (IntE 1))]]
```

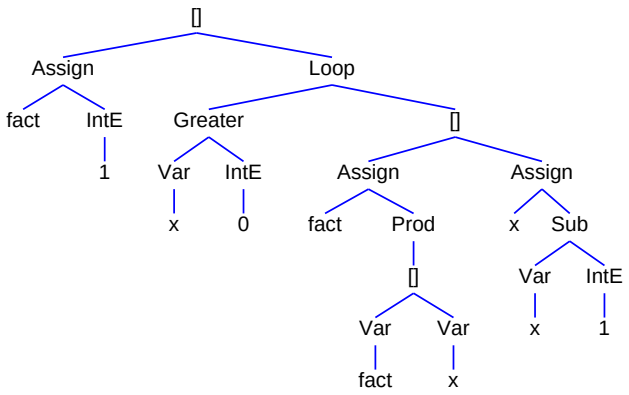


Figure 1.3. *Baumdarstellung von* `prog`

Ein **Interpreter** (siehe Def. 1.1.2) wertet die Ausdrücke aus, und zwar in Abhängigkeit von einer Belegung (*valuation*) *st* des Typs $State = (String \to Int)$ ihrer - als Strings repräsentierten - Variablen. Er besteht aus Funktionen zweiter Ordnung:

```
evalBlock :: Block -> State -> State
evalBlock cs st = foldl (flip evalCom) st cs

evalCom :: Command -> State -> State
evalCom Skip st             = st
evalCom (Assign x e) st     = update st x (evalInt e st)
evalCom (Cond be cs cs') st = if evalBool be st then evalBlock cs st
                                                else evalBlock cs' st
evalCom (Loop be cs) st     = if evalBool be st
                              then evalCom (Loop be cs) (evalBlock cs st)
                              else st

evalInt :: IntE -> State -> Int
evalInt (IntE n) st    = n
evalInt (Var x) st     = st x
evalInt (Sub e1 e2) st = evalInt e1 st - evalInt e2 st
evalInt (Sum es) st    = sum [evalInt e st | e <- es]
evalInt (Prod es) st   = product [evalInt e st | e <- es]

evalBool :: BoolE -> State -> Bool
evalBool (BoolE b) st       = b
evalBool (Greater e1 e2) st = evalInt e1 st > evalInt e2 st
evalBool (Not e) st         = not (evalBool e st)

update :: Eq a => (a -> b) -> a -> b -> a -> b
update f a b a' = if a' == a then b else f a'
```

Zum Beispiel liefert

```
evalBlock prog (update (const 0) "x" 4) "fact"
```

den Wert 24.[4] Mit Datentypen wie *IntE*, etc. werden in Haskell Syntaxbäume implementiert (siehe §3.2). Diese wiederum bilden den Wertebereich eines Parsers. Interpreter und Compiler hingegegen haben Syntaxbäume im Definitionsbereich und erlauben deshalb eine rekursive Definition entlang der Baumstruktur (s.o.). Aber auch die Definition eines Parsers kann nach einem Schema erfolgen, das die Baumstruktur ausnutzt (siehe Beispiel 3.3.2). ❑

Die Baumdarstellung von `prog` in Fig. 1.3 basiert auf einem Algorithmus, der Objekte vom Typ `Block` (und analog anderer Datentypen) mit Knotenpositionen in der Ebenen versieht, dort die jeweiligen Knoten zeichnet und mit Kanten verbindet. Um ihn zu verwenden, muss zunächst eine Funktion definiert werden, die Objekte des gegebenen Datentyps in Objekte der Instanz `Tree String` des polymorphen Typs

```
data Tree a = F a [Tree a]
```

von Bäumen mit beliebigem Knotenausgrad und Einträgen vom Typ `a` überführt. Dann kann z.B. `prog` mit gewünschten horizontalen bzw. vertikalen Knotenabständen gezeichnet werden. Die Knoteneinträge können beliebig lang, dürfen aber nicht mehrzeilig sein.[5] Mehr dazu in Abschnitt 1.3.

[4] `const :: a -> b -> a` ist die konstante Funktion, die immer ihren Parameter als Wert liefert: `const a _ = a`.
[5] Den Algorithmus dahingehend zu verallgemeinern wäre allerdings kein Problem.

Aufgabe Schreiben Sie eine Haskell-Funktion `mkTerm :: Block -> Tree String`, die jedes Objekt t vom Typ `Block` in ein Objekt vom Typ `Tree String` übersetzt und dabei die Konstruktoren von t durch Strings ersetzt! ❑

Natürlich kann man Syntaxbäume auch ohne Grafikerweiterung darstellen, z.B. als **hierarchische Listen**, einem Layout, das auch für hierarchische Menüs verwendet wird. `prog` sähe in einer solchen Darstellung folgendermaßen aus:

```
[Assign "fact" (IntE 1),
 Loop (Greater (Var "x")
               (IntE 0))
      [Assign "fact" (Prod[(Var "fact"),
                           (Var "x")]),
       Assign "x" (Sub (Var "x")
                       (IntE 1))]]
```

Beispiel 1.2.4 Kommando-Printer Die folgenden Haskell-Funktionen übersetzen jedes Objekt vom Typ `Block`, `Command`, `IntE` bzw. `BoolE` in die eben beschriebene Darstellung als hierarchische Liste. Beispielsweise liefert `showBlock prog 0 True` den obigen String. Der Boolesche Parameter der vier show-Funktionen gibt an, ob das Argument direkt hinter das jeweils umfassende Objekt oder linksbündig in eine neue Zeile geschrieben werden soll. Die Linksbündigkeit bezieht sich auf die Spalte, die durch den ganzzahligen Parameter der Funktionen gegeben ist.

```
showBlock :: Block -> Int -> Bool -> String
showBlock cs n = maybeBlanks (f cs) n
                 where f []     = "[]"
                       f [c]    = '[':showCom c (n+1) True ++"]"
                       f (c:cs) = "["++g True++concatMap h cs++"]"
                                  where g = showCom c (n+1); h c = ',':g False

showCom :: Command -> Int -> Bool -> String
showCom c n = maybeBlanks (f c) n
              where f Skip             = "Skip"
                    f (Assign x e)     = "Assign "++show x++' ':
                                         showInt e (n+10+length x) True
                    f (Cond be cs cs') = "Cond "++g showBool True be++
                                                 g showBlock False cs++
                                                 g showBlock False cs'
                                         where g h b e = h e (n+5) b
                    f (Loop be cs)     = "Loop "++g showBool True be++
                                                 g showBlock False cs
                                         where g h b e = h e (n+5) b

showInt :: IntE -> Int -> Bool -> String
showInt e n = maybeBlanks (f e) n
              where f (IntE i)      = "(IntE "++show i++")"
                    f (Var x)       = "(Var "++show x++")"
                    f (Sub e e')    = "(Sub "++g True e++g False e'++")"
                                      where g b e = showInt e (n+5) b
```

```
                  f (Sum (e:es))  = "(Sum["++g True++concatMap h es++"])"
                                    where g = showInt e (n+5); h e = ',':g False
                  f (Prod (e:es)) = "(Prod["++g True++concatMap h es++"])"
                                    where g = showInt e (n+6); h e = ',':g False

showBool :: BoolE -> Int -> Bool -> String
showBool be n = maybeBlanks (f be) n
                where f (BoolE b)      = "(BoolE "++show b++")"
                      f (Greater e e') = "(Greater "++g True e++g False e'++")"
                                         where g b e = showInt e (n+9) b++","
                      f (Not be)       = "(Not "++showBool be (n+5) True++")"

maybeBlanks :: String -> Int -> Bool -> String
maybeBlanks str _ True = str
maybeBlanks str n _    = '\n':replicate n ' '++str
```

Während einfache Argumente eines Konstruktors hintereinander in eine Zeile geschrieben werden, stehen Elemente von Listen immer linksbündig untereinander, das erste allerdings nicht in einer eigenen Zeile. Demzufolge folgen die Definitionen der vier show-Funktionen dem gleichen Schema. Insbesondere die Strukturen der drei Listenbildungen (Blöcke, Summen, Produkte) sind so ähnlich, dass man sie als Instanzen einer einzigen generischen Funktion wiedergeben könnte (Aufgabe!). Warum sind die Darstellungen von `IntE`- oder `BoolE`-Objekten geklammert, die von `Command`-Objekten aber nicht? ❑

Eine **Typklasse** stellt Bedingungen an die Instanzen einer Typvariablen. Die Bedingungen bestehen in der Existenz bestimmter Funktionen und bestimmten Beziehungen zwischen ihnen. Z.B. verlangt die Typklasse

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)          (*)
```

die Existenz einer Gleichheits- und einer Ungleichheitsfunktion auf `a`, wobei durch die Gleichung (*) mit der ersten auch die zweite festgelegt ist. Eine Instanz einer Typklasse besteht aus den Instanzen ihrer Typvariablen sowie Definitionen der von ihr geforderten Funktionen, z.B.:

```
instance Eq (Int,Bool) where
  (x,b) == (y,c) = x == y && b == c
```

Typklassen können wie Objektklassen in OO-Sprachen andere Typklassen erben. Die jeweiligen Oberklassen werden vor dem Erben vor dem Pfeil `=>` aufgelistet, z.B.:

```
class Eq a => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a
  max x y | x >= y = x
          | True   = y
  min x y | x <= y = x
          | True   = y

class Ord a => Enum a where
```

```
toEnum   :: Int -> a
fromEnum :: a -> Int
succ     :: a -> a
succ = toEnum . (+1) . fromEnum
```

Der Punkt bezeichnet die Funktionskomposition: `(f . g) x = f (g x)`. Das **Constraint** `Eq a =>` vor `Ord a` beschränkt die Instanzen von `a` bei der Instanziierung von `Ord a` auf Typen, für die `==` und `/=` definiert sind.

Gegeben sei folgender polymorpher Datentyp für binäre Bäume mit einem Konstruktor für den leeren Baum (`Empty`) und einem Konstruktor zur Bildung eines Baums mit einem Wurzelknoten vom Typ `a` und zwei Unterbäumen:

```
data Bintree a = Empty | Branch (Bintree a) a (Bintree a)
```

Der Typ einer Funktion auf `Bintree a`, die Funktionen von `Ord a` benutzt, muss mit dem Constraint `Ord a =>` versehen werden, z.B.:[6]

```
insert :: Ord a => a -> Bintree a -> Bintree a
insert a Empty                            = Branch Empty a Empty
insert a t@(Branch left b right) | a == b = t
                                 | a < b  = Branch (insert a left) b right
                                 | a > b  = Branch left b (insert a right)
```

Ist eine solche Funktion Teil der Instanz einer Typklasse, dann wird die Instanz mit dem Constraint `Ord a =>` versehen, z.B.:

```
instance Eq a => Ord (Bintree a) where
  Empty <= _                                    = True
  Branch left a right <= Empty                  = False
  Branch left a right <= Branch left' b right' = left <= left' && a == b &&
                                                  right <= right'
```

Beispiel 1.2.5 `Expr`-Instanz von `Show` Die folgende Instanz der Typklasse `Show` bewirkt, dass die Objekte des Datentyps Expr lesbar gedruckt werden. Z.B. werden die Objekte

`Prod[Con(3),Con(5),x,Con(11)]` und `Sum [11 :* (x :^ 3),5 :* (x :^ 2),16 :* x,Con 33]`

in die Strings

`(3*5*x*11)` bzw. `((11*(x^ 3))+(5*(x^ 2))+(16*x)+33)`

umgewandelt.

```
instance Show Expr where show (Con i)   = show i
                         show (Var x)   = x
                         show (Sum es)  = foldShow '+' es
                         show (Prod es) = foldShow '*' es
                         show (n :* e)  = '(':show n ++ '*':show e ++ ")"
                         show (e :^ n)  = '(':show e ++ '^':show n ++ ")"

foldShow :: Char -> [Expr] -> String
foldShow op (e:es) = '(':show e ++ foldl trans "" es ++ ")"
```

[6] `t@(Branch left b right)` ist ein *as-pattern*: `t` ist nur ein weiterer Bezeichner für das Muster `Branch left b right`.

```
                    where trans str e = str ++ op:show e
foldShow _ _       = ""
```

Alle bisherigen Typen waren solche *erster Ordnung*. `Bintree` ist ein Typ *zweiter Ordnung*. Typen beliebiger Ordnung werden **Kinds** genannt. `a, Int, Bintree a, Bintree Int` haben den kind $*$, während `Bintree` den Kind $* \to *$ hat. Die folgende Typklasse schränkt Typen mit diesem Kind ein:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Man kann sie demnach mit `Bintree` instanziieren:

```
instance Functor Bintree where
  fmap g Empty                 = Empty
  fmap g (Branch left a right) = Branch (fmap g left) (g a) (fmap g right)
```

Diese Instanz von `fmap` hat also den Typ `(a -> b) -> Bintree a -> Bintree b`. Sie wendet eine Funktion g auf jeden Knoten eines Baumes an und liefert den entsprechend veränderten Baum zurück. Weitere wichtige Typklassen für Typen mit Kind $* \to *$ bzw. $* \to * \to *$ sind `Monad` (siehe §3.6) und `Arrow`:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a

class Arrow a where
  (>>>) :: a b c -> a c d -> a b d
  pure  :: (b -> c) -> a b c
```

Die einfachsten Datentypen zur Instanziierung von `m` bzw. `a` lauten:

```
data Obj a = O a
data Fun a b = F (a -> b)
```

`Monad m` bzw. `Arrow a` lassen sich damit wie folgt instanziieren:

```
instance Monad Obj where
  O a >>= f = f a
  return = Obj

instance Arrow Fun where
  F f >>> F g = F (g . f)
  pure = Fun
```

Weitere Instanzen der Monadenklasse werden in Abschnitt 3.6 vorgestellt. Wichtige Instanzen der – noch recht "frischen" – Arrows findet man in [31, 61]. Grob gesagt dienen Monaden der Kapselung von Ausgabewerten, während Arrows Ein-Ausgabe-Paare kapseln. Arrows verallgemeinern Monaden insofern als für jede Monade `m` die Funktionen vom Typ `a -> m b` eine Instanz der Arrowklasse bilden:

```
data MFun m a b = M (a -> m b)
```

```
instance Arrow (MFun m) where
  M f >>= M g = M (\b -> f b >>= g)
  pure f = M (\b -> return (f b))
```

Weitere Haskell-Konstrukte werden on-the-fly eingeführt, wenn wir sie brauchen. Zur umfassenderen Einarbeitung in die Sprache empfehle ich [28] oder eines der Lehrbücher [12, 33, 66, 77, 29, 8].[7]

[7] Die Reihenfolge der Referenzen entspricht in etwa der Tiefe der jeweiligen Darstellung.

1.3 Testumgebung und Benutzung des Moduls Painter.hs

Eine einfache Umgebung zum Testen von Haskell-Funktionen sieht wie folgt aus: Es werden drei Dateien angelegt: eine Programmdatei (`prog.hs`), in der alle selbstdefinierte Typen und Funktionen einschließlich `test` (s.u) stehen sowie eine für Eingaben (`source`) und eine für Ausgaben (`target`). Dann wird ein Haskell-Interpreter (`ghci` oder `hugs`) aufgerufen und mit dem Kommando `:load prog` wird die Programmdatei geladen. Nun können Aufrufe von Funktionen aus `prog.hs` eingegeben werden, die nach Drücken der return-Taste ausgeführt und deren Ergebnisse im shell-Fenster angezeigt werden.

Typ und Code von `test` sind ganz einfach:

```
test :: (Read a,Show b) => (a -> b) -> IO ()
test f = readFile "source" >>= writeFile "target" . show . f . read
```

`readFile ''source''` liest den Inhalt der Datei `source` und gibt ihn als String zurück. `read` übersetzt einen String in ein Objekt vom Typ `a`. `f` verarbeitet dieses Objekt und gibt ein Objekt vom Typ `b` zurück. `show` übersetzt dieses Objekt in einen String. `writeFile ''target''` schreibt einen String in die Datei `target`.

`IO` ist eine (Standard-)Instanz der Typklasse `Monad` und `>>=` die entsprechende Monadenkomposition. Sie nimmt den vom ersten Argument berechneten Wert (hier: den eingelesenen String) und übergibt ihn an die Funktion im zweiten Argument, das ist hier die Komposition der vier oben beschriebenen Funktionen `read`, `f`, `show` und `writeFile ''target''`. Der Punkt ist die Haskell-Notation für die in der Mathematik üblicherweise als Kreis (∘) dargestellte Funktionskomposition.

Zur Wiedergabe von Texten, Bäumen und Wegen steht der Modul `Painter.hs`[8] zur Verfügung. Er baut auf der Graphikbibliothek HGL[9] auf, die mit `import Graphics.HGL` in das jeweilige Haskell-Programm eingebunden wird. Unter hugs heißt sie SOE (siehe http://haskell.org/soe/software1.htm). Kommandos vom Typ `IO ()`, die Zuweisungen an Variablen des HGL-Typs `Window` enthalten, dürfen nur als Argumente der Funktion `runGraphics :: IO () -> IO ()` vorkommen. Es können sonst Deadlocks auftreten.

`drawText ''source''` schreibt den Text in der Datei `source` in Spalten von jeweils 80 Zeichen und 37 Zeilen in ein Fenster.

`drawTree ''source''` startet eine Schleife, in der zunächst der horizontale bzw. vertikale Abstand zwischen benachbarten Knoten erfragt wird. Nach Eingabe der beiden Abstandsparameter und Drücken der return-Taste öffnet sich ein Fenster mit dem Titel `source`, in das der Inhalt der Datei `source` als Baum gezeichnet wird, sofern der Inhalt ein Objekt vom Typ `Tree String` ist. `Tree String` ist die String-Instanz des polymorphen Datentyps

```
data Tree a = F a [Tree a]
```

Um das Fenster zu schließen, müssen der Cursor hineinbewegt und die linke Maustaste gedrückt werden. Ist das Fenster begonnen, beginnt ein neuer Schleifendurchlauf. Verlassen wird die Schleife, wenn anstelle einer erneuten Parametereingabe die return-Taste gedrückt wird.

`drawTreeC ''source''` tut das Gleiche, jedoch werden jetzt die Knoten des Baumes abhängig von ihrem jeweiligen Abstand von der Wurzel unterschiedlich gefärbt.

[8] fldit-www.cs.uni-dortmund.de/∼peter/Haskellprogs/Painter.hs
[9] haskell.cs.yale.edu/ghc/docs/6.6/html/libraries/HGL/Graphics-HGL.html

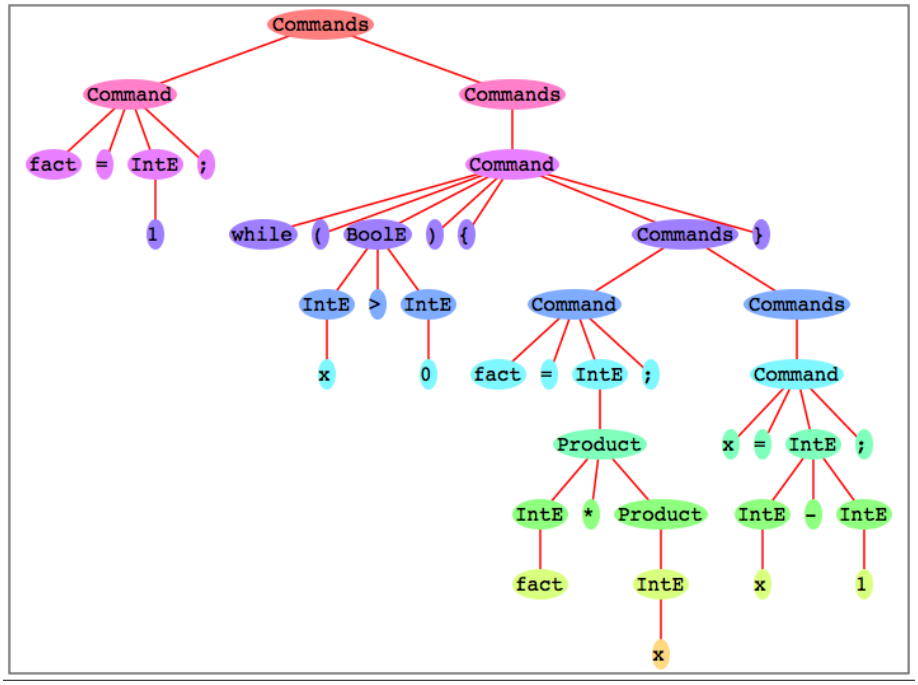


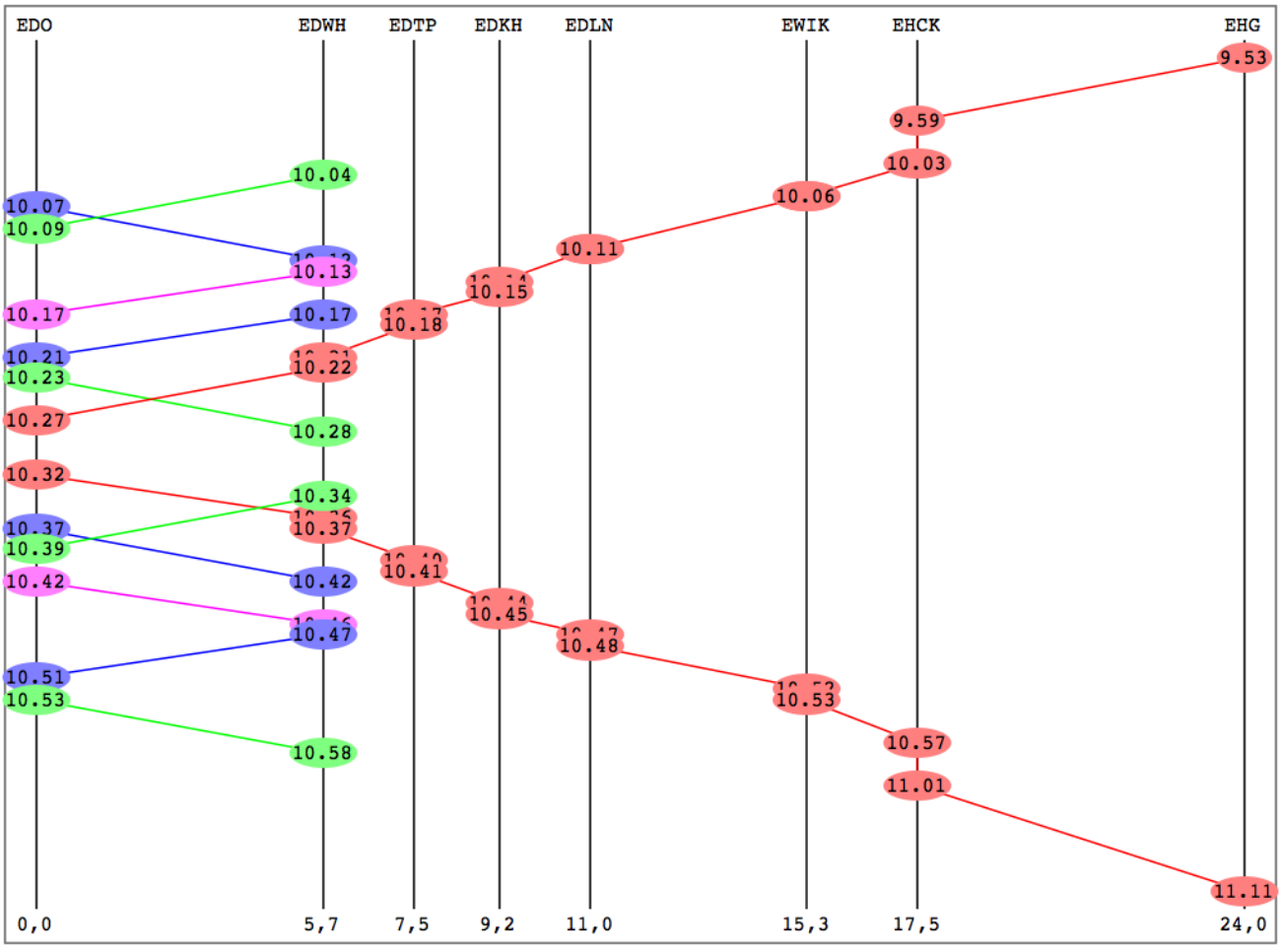
Figure 9.2.
Ergebnis von `drawTreeC ''Haskellprogs/factorial''`

Ist t ein Element eines beliebigen Typs, der eine Instanz der Typklasse `Show` ist, dann übersetzen `drawTerm t ''source''` und `drawTermC t ''source''` zunächst t in ein Objekt vom Typ `Tree String`, schreiben dieses in die Datei `source` und rufen zur Baumdarstellung von t `drawTree ''source''` bzw. `drawTreeC ''source''` auf.

`drawPaths ''source''` startet eine Schleife, in der zunächst ein horizontaler bzw. vertikaler Skalierungsfaktor erfragt wird. Nach Eingabe der beiden Faktoren und Drücken der return-Taste öffnet sich ein Fenster mit dem Titel `source`, in das der Inhalt der Datei `source` als Graph gezeichnet wird, sofern der Inhalt ein Objekt vom Typ `[Path]` ist. `Path` ist folgendermaßen definiert:

```
type Path = ([Node],RGB)
type Node = (String,Pos)
type Pos  = (Int,Int)
data RGB  = RGB Int Int Int
```

Ein Objekt vom Typ `Path` ist also eine Knotenliste zusammen mit einer RGB-Farbe, mit der alle Knoten der Liste und die sie verbindenden Kanten gefärbt werden. Ein Knoten ist ein Paar, bestehend aus einer Markierung vom Typ `String` und seiner Position in der Ebene. `RGB r g b` bezeichnet eine Farbe mit Rotwert `r`, Grünwert `g` und Blauwert `b`. Stimmen mindestens zwei der drei Werte mit 0 oder 255 überein, dann handelt es sich um eine sog. reine oder Hue-Farbe. Abweichungen vom Hue-Wert verändern die Helligkeit der Farbe und vermindern damit ihre Brillianz. Codiert werden RGB-Farben als sechsstellige Hexadezimalzahlen, z.B. ist `RGB 255 0 0` = FF0000 = rot. Die Kanten eines Weges vom Typ `Path` werden etwas heller als seine Knoten gefärbt. Ist der Weg schwarz (RGB 0 0 0), dann werden seine Knoten weiß gezeichnet. Knoteneinträge sind immer schwarz.



Ergebnis von `drawPaths ''http://fldit-www.cs.uni-dortmund.de/~peter/Haskellprogs/fahrplan''`

Kapitel 2

Lexikalische Analyse

Die Aufgabe der lexikalischen Analyse ist die Zusammenfassung von Zeichen des zunächst nur als Zeichenfolge eingelesenen Quellprogramms zu *Symbolen* sowie das Ausblenden bedeutungsloser Zeichen. Algorithmisch betrachtet, ist lexikalische Analyse der Teil der Syntaxanalyse, der *iterativ*, also ohne Keller, durchgeführt werden kann. Iterativ definierte Mengen, Prädikate oder Boolesche Funktionen (wie die **Scanner** oder **Lexer** genannten Algorithmen, die lexikalische Analyse durchführen) lassen sich mit verschiedenen formalen Modellen beschreiben, nämlich mit regulären Grammatiken, regulären Ausdrücken oder (erkennende) endliche Automaten. Iterativ definierte Funktionen mit komplexeren Wertebereichen als `Bool` sind ebenfalls als Automaten darstellbar (und umgekehrt!).

2.1 Reguläre Ausdrücke, Automaten und reguläre Grammatiken

Definition 2.1.1 Sei A ein Alphabet (Zeichenmenge). Die Menge $Reg(A)$ der **regulären Ausdrücke über** A ist induktiv definiert (d.h. $Reg(A)$ ist die kleinste Menge, die folgende Bedingungen erfüllt):

- $\varepsilon \in Reg(A)$
- $A \subseteq Reg(A)$
- $R, R' \in Reg(A) \quad \Longrightarrow \quad R|R',\ RR' \in Reg(A)$ (Summe und Produkt)
- $R \in Reg(A) \quad \Longrightarrow \quad R^+,\ R?,\ R^* \in Reg(A)$ (Abschlüsse)

R ist in **disjunktiver Normalform**, falls es $R_1, \ldots, R_n \subseteq A^*$ gibt mit $R = R_1|\ldots|R_n$.

Die Funktion $L : Reg(A) \to \wp(A^*)$ ordnet jedem regulären Ausdruck über A eine Menge von Wörtern über A zu. L ist induktiv über dem Aufbau von $Reg(A)$ definiert:

- $L(\varepsilon) = \{\varepsilon\}$ (leeres Wort)
- $L(a) = \{a\}$ für alle $a \in A$ (Symbol)
- $L(R|R') = L(R) \ \cup \ L(R')$ (Summe)
- $L(RR') = \{vw \mid v \in L(R), w \in L(R')\}$ (Produkt)
- $L(R^+) = \cup_{n>0} R^n$, wobei $R^1 =_{def} R$ und $R^{n+1} =_{def} RR^n$ (transitiver Abschluss)
- $L(R?) = L(R|\varepsilon\}$ (reflexiver Abschluss)

- $L(R^*) = L(R^+|\varepsilon\}$ (reflexiv-transitiver Abschluss)

Eine Menge $M \subseteq A^*$ heißt **reguläre Sprache über** A, wenn es einen regulären Ausdruck R über A mit $L(R) = M$ gibt. ❑

Ein Haskell-Datentyp für $Reg(A)$ könnte wie folgt lauten:

```
data RegExp a = Const a | Eps | Sum (RegExp a) (RegExp a) |
                Prod (RegExp a) (RegExp a) | Plus (RegExp a)
```

Da reflexiver wie reflexiv-transitiver Abschluss aus anderen Operatoren abgeleitet sind, bietet es sich an, diese Operatoren nicht als Konstruktoren, sondern als Funktionen vom Typ `RegExp a -> RegExp a` zu definieren:

```
refl e = Sum e Eps
star e = Sum (Plus e) Eps
```

Überflüssige Konstruktoren sollten grundsätzlich vermieden werden, da Funktionen auf einem Datentyp immer für jeden zugehörigen Konstruktor definiert werden müssen!

Nichtreguläre Sprachen sind z.B.

- geschachtelte Ausdrücke: $\{u^n w v^n \mid u, v \in A^*, n \geq 0\}$,
- String-Wiederholungen: $\{uwu \mid u \in A^*\}$,
- Vergleich von Präfix und Suffix: $\{uwv \mid u, v \in A^*, 3 * length(u) = 5 * length(v)\}$.

Hier ist immer ein Keller zur Speicherung von Teilwörtern erforderlich!

Die folgende *Graphgrammatik* beschreibt die Übersetzung eines regulären Ausdrucks R in einen nichtdeterministischen Automaten, der $L(R)$ erkennt:

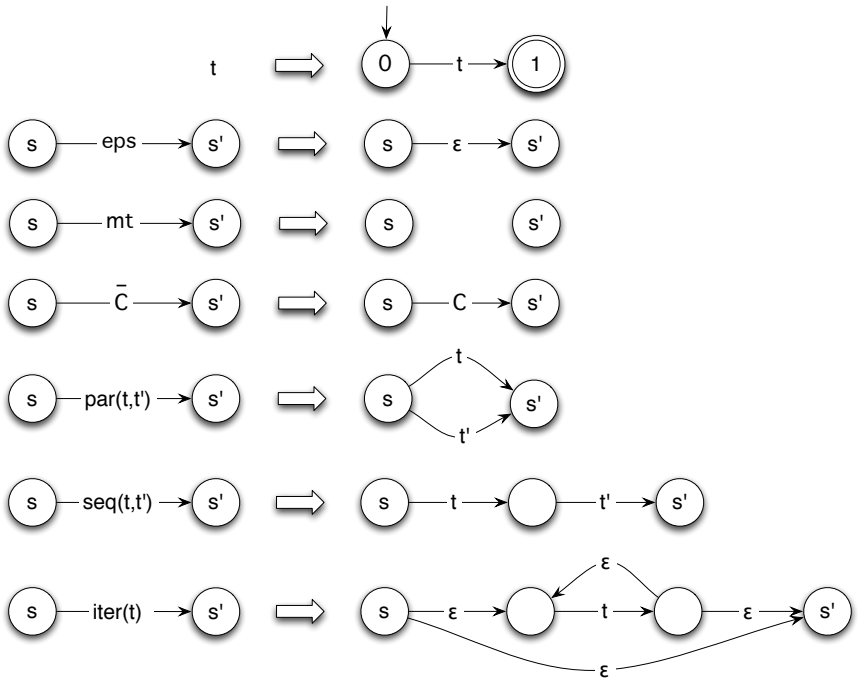


Figure 2.1. *Vom regulären Ausdruck zum erkennenden Automaten*[1]

Wende die erste Regel auf R an. Es entsteht ein Graph mit zwei Knoten und einer mit R markierten Kante. Dieser wird nun mit den anderen Regeln schrittweise vergrößert, bis an allen seinen Kanten nur noch Alphabetsymbole stehen. Dann stellt er den Transitionsgraphen eines Automaten dar, der R erkennt.

	a	b	c	eps
0				1 2
2				6 4
3				1 2
4	5			
5				3 4
6		8		
7				3 6
8			7	

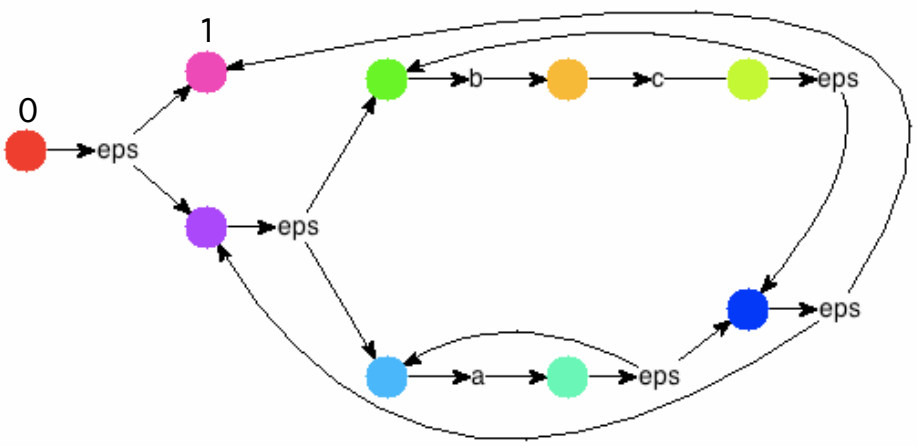


Figure 2.2. *Tabellen- bzw. Graphdarstellung des gemäß Fig. 2.1 aus dem regulären Ausdruck* $((a^+|(bc)^+)^*$ *konstruierten Automaten.*

Definition 2.1.2 Ein **endlicher Automat** $A = (Q, X, Y, \delta, \beta, q_0)$ besteht aus einer endlichen Zustandsmenge Q, einer endlichen Eingabemenge X, einer Ausgabemenge Y, einer Übergangsfunktion

$$\delta : Q \times X \to Q$$

(**deterministischer Automat**) bzw.

$$\delta : Q \times (X \cup \{\varepsilon\}) \ \to \wp(Q)$$

(**nichtdeterministischer Automat**), einer Ausgabefunktion $\beta : Q \to Y$ bzw. $\beta : Q \to \wp(Y)$ und einem Anfangszustand $q_0 \in Q$.

A heißt **erkennender Automat**, wenn die Ausgabemenge Y zweielementig ist, z.B. $Y = \{0, 1\}$. Man nennt dann jeden Zustand $q \in Q$ mit $\beta(q) = 1$ (im deterministischen Fall) bzw. $1 \in \beta(q)$ (im nichtdeterministischen Fall) einen **Endzustand** von A. Eine Menge von Endzuständen bezeichnen wir mit E.

Die **Fortsetzung von δ auf Wörter** ist eine induktiv definierte Funktion von $\delta^* : Q \times X^* \to Q$ (deterministischer Fall) bzw. $\delta^* : Q \times X^* \to \wp(Q)$[2] (nichtdeterministischer Fall):

$$\begin{aligned} \delta^*(q, \varepsilon) &= q \\ \delta^*(q, xw) &= \delta^*(\delta(q, x), w) \quad \text{für alle } x \in X \text{ und } w \in X^* \end{aligned}$$

bzw.

$$\begin{aligned} \delta^*(q, \varepsilon) &= \varepsilon hull(q) \\ \delta^*(q, xw) &= \delta^*(\varepsilon hull(\delta(q, x)), w) \quad \text{für alle } x \in X \text{ und } w \in X^* \\ \varepsilon hull(q) &= \textstyle\bigcup_{i \in \mathbb{N}} \varepsilon hull_i(q) \\ \varepsilon hull_0(q) &= \{q\} \\ \varepsilon hull_{i+1}(q) &= \varepsilon hull_i(q) \cup \delta(\varepsilon hull_i(q), \varepsilon) \end{aligned}$$

[1] Nach [85], Abb. 6.3. Entgegen entsprechenden Darstellungen in manchen anderen Büchern sind nur bei der Übersetzung von R^+ zusätzliche ε-Übergänge erforderlich. Ohne sie würde beispielsweise ein für $R^+|Q^+$ konstruierter Automat fälschlicherweise die größere Sprache $(R|Q)^+$ erkennen.

[2] $\wp(Q)$ bezeichnet die Potenzmenge von Q.

wobei eine Funktion $f : A \to B$ immer wie folgt auf Teilmengen von A fortgesetzt wird: $f(A) = \{f(a) \mid a \in A\}$. In Haskell lässt sich δ^* mit *foldl* aus der Übergangsfunktion δ bilden. Der *Iterationsoperator* $*$ entspricht der Faltung mit δ.[3] Im nichtdeterministischen Fall ist zu berücksichtigen, dass Q endlich ist, also ein $i \in \mathbb{N}$ mit $\varepsilon hull_{i+1}(q) = \varepsilon hull_i(q)$ existiert. Die Iteration in `epsHull` bricht also ab, wenn diese Bedingung erfüllt ist.

```
iter :: (state -> input -> state) -> state -> [input] -> state
iter = foldl
```

bzw.

```
iter :: Eq state => (state -> Maybe input -> [state]) -> state -> [input] -> [state]
iter delta q = foldl f (epsHull delta [q])
               where f qs x = epsHull delta $ joinMap (flip delta $ Just x) qs

epsHull :: Eq state => (state -> Maybe input -> [state]) -> [state] -> [state]
epsHull delta = f where f qs = if all (‘elem‘ qs) qs’ then qs else f $ qs ‘join‘ qs’
                              where qs’ = joinMap (flip delta Nothing) qs

joinMap :: Eq b => (a -> [b]) -> [a] -> [b]
joinMap f = foldl join [] . map f

join, minus :: Eq a => [a] -> [a] -> [a]
join (x:s) s’ = if x ‘elem‘ s’ then join s s’ else x:join s s’
join _ s      = s
minus (x:s) s’ = if x ‘elem‘ s’ then minus s s’ else x:minus s s’
minus _ _      = []
```

Die **Erreichbarkeitsfunktion** $r : X^* \to Q$ bzw. $r : X^* \to \wp(Q)$ eines deterministischen bzw. nichtdeterministischen Automaten ist definiert durch: $r(w) = \delta^*(q_0, w)$.

$L(A) =_{def} \{w \in X^* \mid r(w) \in E\}$ bzw. $L(A) =_{def} \{w \in X^* \mid r(w) \cap E \neq \emptyset\}$ ist die von einem deterministischen bzw. nichtdeterministischen erkennenden Automaten A **erkannte Sprache**.

Zwei erkennende Automaten sind **äquivalent**, wenn die von ihnen erkannten Sprachen übereinstimmen. ❑

Definition 2.1.3 Eine kontextfreie Grammatik (N, T, P, S) (s. Def. 3.1.1) heißt **regulär**, wenn für alle $A \longrightarrow w \in P$ $w \in T^*N$ oder $w \in T^*$ gilt. ❑

Satz 2.1.4 Sei $L \subseteq T^*$.

Es gibt einen regulären Ausdruck R, dessen Sprache mit L übereinstimmt.
$\iff$ Es gibt eine reguläre Grammatik, die L erzeugt (s. Def. 3.1.1).
$\iff$ Es gibt einen endlichen Automaten, der L erkennt. ❑

Der Übergang von regulären Ausdrücken zu nichtdeterministische Automaten ist durch die Ersetzungsregeln von Fig. 2.1 definiert. Noch einfacher ist die Übersetzung regulärer Grammatiken in nichtdeterministische Automaten. Dabei wird jede Produktion der Grammatik zu einer Folge von Zustandstransitionen (siehe Fig. 2.3).

[3] Endliche Mengen werden als Listen implementiert.

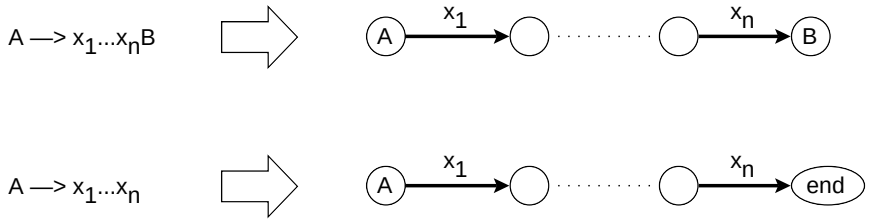


Figure 2.3. *Übersetzung regulärer Grammatiken in nichtdeterministische Automaten*

2.2 Scanner

Beispiel 2.2.1 Kommando-Scanner in Haskell. Wir wollen die in einer Zeichenfolge auftretenden Symbole der imperativen Sprache von Beispiel 1.2.3 erkennen. Dazu benötigen wir einen Automaten mit mindestens drei Zuständen, weil hier drei Typen von Symbolen zu unterscheiden sind: Schlüsselwörter, Variablen und Zahlen. Weitere Zustände sind notwendig, wenn der ein Symbol repräsentierende String Teilstrings hat, die andere Symbole repräsentieren können.

Der Scanner soll z.B. die Zeichenfolge

```
{fact = 1; while (x > 0) {fact = fact*x; x = x-1;}}
```

in die Symbolfolge

```
[Key "{",Ide "fact",Key "=",Num 1,Key ";",Key "while",Key "(",Ide "x",
 Key ">",Num 0,Key ")",Key "{",Ide "fact",Key "=",Ide "fact",Key "*",
 Ide "x",Key ";",Ide "x",Key "=",Ide "x",Key "-",Num 1,Key ";",Key "}",
 Key "}"]
```

überführen. Zunächst ein Datentyp für die Symbole:

```
data Symbol = Num Int | Ide String | Key String
```

Der Scanner wird als Menge iterativer (!) Funktionen auf dem Eingabewort realisiert, im Beispiel: `scan`, `scanNum` und `scanIde`. Neben dem Eingabewort hat jede Funktion ein Argument vom Typ `[Symbol]`, das die Symbolfolge schrittweise akkumuliert. Jede Funktion definiert die auf einen bestimmten Zustand angewendete Fortsetzung der Transitionsfunktion eines erkennenden Automaten auf Wörter.

```
scanAll :: String -> ([Symbol],String)
scanAll = scan []

scan :: [Symbol] -> String -> ([Symbol],String)
scan syms (x:str) | isSpecial x = scan (syms++[Key [x]]) str
                  | isDigit x   = scanNum syms [x] str
                  | isDelim x   = scan syms str
                  | True        = scanIde syms [x] str
scan syms str                   = (syms,str)

scanNum :: [Symbol] -> String -> String -> ([Symbol],String)
scanNum syms num (x:str) | isDigit x = scanNum syms (num++[x]) str
```

```
                        | True      = scan (syms++[Num (read num)])
                                         (x:str)
scanNum syms num str                = (syms++[Num (read num)],str)

scanIde :: [Symbol] -> String -> String ->([Symbol],String)
scanIde syms ide (x:str) | isSpecial x || isDelim x
                                    = scan (syms++[embed ide]) (x:str)
                         | True     = scanIde syms (ide++[x]) str
scanIde syms ide str                = (syms++[embed ide],str)

embed :: String -> Symbol
embed str | isKeyword str = Key str
          | True          = Ide str

isKeyword :: String -> Bool
isKeyword str = str `elem` words "true false if else while"

isSpecial, isDigit, isDelim :: Char -> Bool

isSpecial x = x `elem` "(){};=!>+-*"
isDigit x   = x `elem` ['0'..'9']
isDelim x   = x `elem` " \n\t"
```

Aufgabe Beschreiben Sie die reguläre Sprache, die `scanAll` erkennt, als regulären Ausdruck, reguläre Grammatik oder endlichen Automaten! ❑

Die folgende Erweiterung von `scanAll` übergibt zu jedem erzeugten Symbol dessen Anfangsposition (Zeile und Spalte) im Eingabewort. Parser werden später diese zusätzliche Information benutzen, um Fehlerpositionen auszugeben. In den Typen der scan-Funktionen wird daher `Symbol` durch `(Symbol,Int,Int)` ersetzt.

```
scanAll :: String -> ([(Symbol,Int,Int)],String)
scanAll = scan [] 1 1

scan :: [(Symbol,Int,Int)] -> String -> Int -> Int -> ([(Symbol,Int,Int)],String)
scan syms (x:str) i j | isSpecial x = scan (syms++[(Key [x],i,j)]) str i (j+1)
                      | isDigit x   = scanNum syms ([x],i,j) str i (j+1)
                      | isNewline x = scan syms str (i+1) 1
                      | isDelim x   = scan syms str i j
                      | True        = scanIde syms ([x],i,j) str i (j+1)
scan syms str _ _                   = (syms,str)

scanNum :: [(Symbol,Int,Int)] -> (String,Int,Int) -> String -> ([(Symbol,Int,Int)],String)
scanNum syms (num,k,l) (x:str) i j
                  | isDigit x = scanNum syms (num++[x],k,l) str i (j+1)
                  | True      = scan (syms++[(Num (read num),k,l)]) (x:str) i j
scanNum syms (num,k,l) str i j = (syms++[(Num (read num),k,l)],str)

scanIde :: [(Symbol,Int,Int)] -> (String,Int,Int) -> String -> ([(Symbol,Int,Int)],String)
scanIde syms (ide,k,l) (x:str) i j
```

```
    | isSpecial x || isDelim x = scan (syms++[(embed ide,k,l)]) (x:str) i j
    | True                     = scanIde syms (ide++[x],k,l) str i (j+1)
scanIde syms (ide,k,l) str i j = (syms++([(embed ide,k,l)],i,j),str)

isNewline :: Char -> Bool
isNewline x = x == '\n'
```

Allgemein gesprochen, führt ein Scanner das Quellprogramm Zeichen für Zeichen einem erkennenden Automaten zu, der jedes Zeichen liest, entsprechende Zustandsübergänge ausführt und dessen Endzustände mit jeweils einem Ausgabesymbol markiert sind (siehe Fig. 2.4; Endzustände sind hier die doppelt umrandeten ungesternten Zustände und die direkten Vorgänger gesternter Zustände.). Beim Erreichen eines Endzustandes werden sein Name als Wert einer Variablen *final* gespeichert und ein Zeiger *start_position* auf die Position des nächsten Zeichens des Quellprogramms gesetzt, und der Automat fährt mit dem Lesevorgang fort. Wird wieder ein Endzustand erreicht, dann werden *final* und *start_position* entsprechend umgesetzt. Beim Scanner von Beispiel 2.2.1 ist die Umsetzung durch rekursive Aufrufe von `scan` implementiert. Gibt es für das nächste Eingabezeichen keinen Zustandsübergang, dann wird das Symbol, mit dem der Endzustand *final* markiert ist, ausgegeben. Danach wird die Eingabe ab *start_position* (noch einmal) gelesen und wie oben fortgefahren.

Dieses Verfahren stellt sicher, dass der Scanner immer das *längste* Symbol zurückgibt, das Präfix der jeweiligen Zeichenfolge ist.

Aufgabe Zeigen Sie, dass `scanAll` aus Beispiel 2.2.1 ebenso vorgeht! ❑

Natürlich braucht nicht das gesamte Quellprogramm im Eingabepuffer gehalten zu werden. Wenn der Scanner die zweite Hälfte des in den Puffer geladenen Programmstückes analysiert, wird bereits das folgende Programmstück in die erste Hälfte geladen. Wenn dann dieses verarbeitet wird, wird die zweite Hälfte neu geladen, usw.

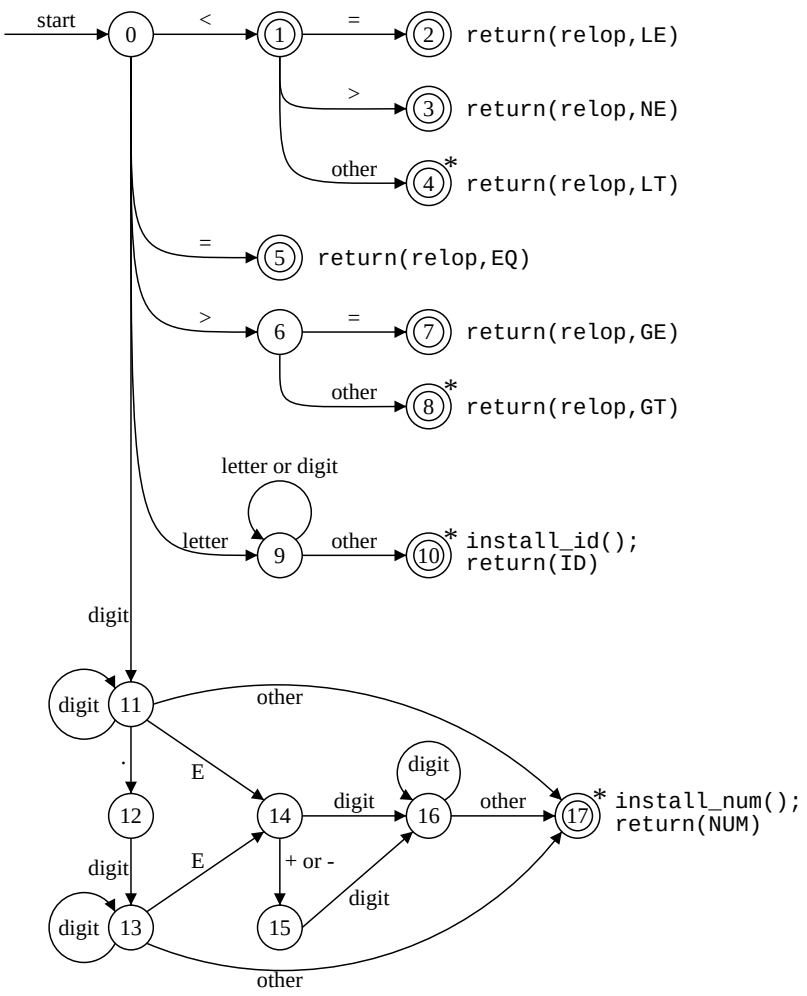


Figure 2.4. *Scanner-Automat*

Der Ausgangspunkt des obigen Scanners ist ein endlicher Automat, der von einem **Scanner-Generator** gemäß Fig. 2.1 aus regulären Ausdrücken konstruiert wird. Er ist (u.a. wegen der ε-Übergänge) i.a. nichtdeterministisch, lässt sich aber leicht in einen deterministischen überführen, indem man die Potenzmengen $\wp(Q)$ und $\wp(Y)$ seiner Zustands- bzw. Ausgabemenge als neue Zustandsmenge Q' bzw. Y' auffasst und seine Übergangs- und Ausgabefunktion an Q' anpasst (siehe z.B. [26], §2.3.5).

In Haskell lässt sich der Übergang vom nichtdeterministischen zum äquivalenten deterministischen Automaten recht einfach implementieren:

```
type NonDetAuto input output state = (state -> Maybe input -> [state], state -> [output],
                                      state)
type DetAuto input output state    = (state -> input -> state, state -> output, state)

makeDet :: (Eq state,Eq output) =>
                        NonDetAuto input output state -> DetAuto input [output] [state]
makeDet (delta, beta, q0) = (delta', joinMap beta, epsHull delta [q0])
                    where delta' qs x = epsHull delta (joinMap (flip delta (Just x)) qs)
```

Ein als Objekt vom Typ `DetAuto` oder `NonDetAuto` repräsentierter Automat lässt sich nicht direkt ausgeben, da es Funktionen enthält und diese keine standardmäßige Stringdarstellung haben. Glücklicherweise haben wir es hier mit *endlichen* Automaten zu tun. Übergangs- und Ausgabefunktion haben also einen endlichen

Definitionsbereich und können deshalb in Tabellen oder Adjanzenzlisten überführt werden, für die es dann wieder standardmäßige Stringdarstellungen gibt.

Da die Zustände eines endlichen Automaten unstrukturiert sind, ist es meistens sinnvoll, mit den einfachsten Namen für Zustände zu arbeiten, nämlich mit ganzen Zahlen. Berücksichtigt man noch die Einschränkung auf *erkennende* Automaten, dann bietet sich als Datenstruktur für (nichtdeterministische) Automaten der folgende Typ an:

```
type IntAuto a = ([Int],[a],NonDetAuto a Bool Int)
```

So wie `Int` die Obermenge der jeweiligen Zustandsmenge ist, so steht die Typvariable a für eine geeignete Obermenge des jeweiligen Eingabealphabets. Die Listen vom Typ [Int] bzw. [a] liefern dann die exakte Zustandsmenge bzw. das exakte Eingabealphabet. Die Ausgabe eines Automaten vom Typ `IntAuto` könnte beispielsweise mit folgender Funktion `writeAuto` erfolgen:

```
writeAuto :: Show a => IntAuto a -> IO ()
writeAuto (qs,as,(delta,beta,q0)) = writeFile "autos" (initial++trips++finals)
      where initial = "\ninitial state: "++show q0
            trips = "\ntransition function:"++
                    fun2ToTrips delta qs (Nothing:map Just as)
            finals = "\nfinal states: "++show [i | i <- qs, True `elem` beta i]
```

Die Hilfsfunktion `fun2ToTrips` übersetzt eine Funktion vom Typ `a -> b -> [c]` zunächst in die entsprechende Liste von (Argument,Werte)-Paaren und diese dann in Stringzeilen mit einem Paar pro Zeile:

```
fun2ToTrips :: (Show a,Show b,Show c) => (a -> b -> [c]) -> [a] -> [b] -> String
fun2ToTrips f as bs = concatMap h [trip | trip@(_,_,_:_) <- concatMap g as]
                      where g a = [(a,b,f a b) | b <- bs]
                            h (a,b,cs) = '\n':show (a,b)++" leads to "++show cs
```

Die Verwendung des Typs `IntAuto` ist übrigens auch sinnvoll bei der Implementierung von Funktionen, die Automaten erzeugen, wie der in Fig. 2.1 beschriebenen Übersetzung regulärer Ausdrücke in Automaten (siehe Beispiel 4.2.4). Bei dieser Übersetzung wird nicht nur die Übergangsfunktion, sondern auch die Zustandsmenge selbst schrittweise erweitert, ohne dass man zu Anfang weiß, wie viele Zustände notwendig sein werden. Deshalb braucht man hier einen Automatentyp mit einem unbegrenzten Vorrat an Zuständen und der eindeutigen Identifizierung des jeweils "nächsten freien" Zustands. Beides leistet der Typ `Int`.

Wie wir oben gesehen haben, gehören die Elemente der von einem Scanner erzeugten Symbolfolge in der Regel verschiedenen Kategorien an: Schlüsselwörter, Variablen, Zahlen, etc. Jede Kategorie ist durch ihren eigenen regulären Ausdruck definiert. Ein Scanner-Generator erzeugt zunächst pro Ausdruck einen Scanner. Die generierten Scanner werden dann zu einer globalen Scan-Funktion zusammengesetzt.

Der Scanner eines einzelnen regulären Ausdrucks e ordnet einem Eingabestring `str` zwei Werte zu: falls es dieses gibt, ein zu $L(e)$ gehöriges Präfix von `str`, sowie die jeweilige Resteingabe.

```
regScan :: RegExp Char -> String -> (Maybe String,String)
regScan (Const c) (d:str) = if c == d then (Just [d],str)
                                      else (Nothing,d:str)
regScan Eps str           = (Just [],str)
regScan (Sum e e') str    = case regScan e str of result@(Just _,_) -> result
```

```
                                                  _ -> regScan e' str
regScan (Prod e e') str   = case regScan e str of (Just sym,rest) -> regScan e' rest
                                                  _ -> (Nothing,str)
regScan (Plus e) str      = case regScan e str of
                                (Just sym,rest) -> restPlus e sym rest
                                _ -> (Nothing,str)

restPlus :: RegExp Char -> String -> String -> (Maybe String,String)
restPlus e sym str = case regScan e str of
                       (Just sym',rest) -> restScan (sym++sym') e rest
                                           _ -> (Just sym,str)
```

Nehmen wir an, es gibt n verschiedene Kategorien von Symbolen, jede mit einem eigenen regulären Ausdruck e und einem Compiler der Wörter von $L(e)$ in einen spezifischen Ergebnistyp:

```
data Symbol = C1 Result_1 | ... | Cn Result_n

compile_1 :: String -> Result_1
...
compile_n :: String -> Result_n
```

Seien $e_1, \ldots, e_n$ die den n Symbolkategorien zugeordneten regulären Ausdrücke und w der gesamte Eingabestring. Der vollständige Scanner liest die Zeichen von w, bis er ein Präfix von w gefunden hat, das zur Sprache einer der Ausdrücke $e_1, \ldots, e_n$ gehört und wiederholt diesen Vorgang auf der Resteingabe. Gibt es Wörter in $L(e_i)$, die Präfixe von Wörtern in $L(e_j)$ sind, dann sollte `regScan ej` vor `regScan ei` aufgerufen werden, um sicherzustellen, dass immer zuerst das längste Präfix der (Rest-)Eingabe zu einem Symbol gemacht wird. Dazu muss diese Reihenfolge auch beim Scannen von Teilausdrücken der Form `Sum e e'` gewählt werden.

```
scanAll :: String -> ([Symbol],String)
scanAll = scan []

scan :: [Symbol] -> String -> ([Symbol],String)
scan syms str = case regScan e1 str of
                  (Just str,rest) -> scan (syms++[C1 (parse_1 str)]) rest
                  _ -> case regScan e2 str of
                       (Just str,rest) -> scan (syms++[C2 (parse_2 str)]) rest
                       _ -> ...
                            _ -> case regScan en str of
                                 (Just str,rest) -> scan (syms++[Cn (parse_n str)]) rest
                                 _ -> (syms,str)
```

2.3 Minimierung deterministischer Automaten

Definition 2.3.1 Die Komposition $f_A = \beta_A \circ r_A : X^* \to Y$ der Ausgabe- und Erreichbarkeitsfunktionen eines deterministischen Automaten A heißt **Verhalten** von A. Zwei deterministische Automaten A und B sind **äquivalent**, wenn $f_A = f_B$ gilt. ❑

Dieser klassische Verhaltensbegriff ist insbesondere für erkennende Automaten geeignet, wo Verhalten auf die erkannte Sprache und die Äquivalenz von A und B auf die Gleichheit der von A bzw. B erkannten Sprachen hinausläuft. Für manche andere Anwendungen, vor allem jene im Bereich der Prozessmodellierung, ist ein feinerer Verhaltensbegriff adäquater, der auch die Entscheidungspunkte vor Verzweigungen im Übergangsgraphen berücksichtigt und nicht nur die Menge der möglichen Pfade. Für Robin Milner's berühmte *vending machine* ist die Pfadsemantik sicher inadäquat: seine Sprache ist durch den regulären Ausdruck $R_1 = coin(tea + (coin\ coffee))$ beschreiben, soll heißen: nach dem Einwurf einer Münze kann man sich einen Tee, nach dem Einwurf einer zweiten Münze einen Kaffee aus dem Automaten ziehen. Dieselbe Sprache wird auch durch den Ausdruck $R_2 = (coin\ tea) + (coin\ coin\ coffee)$ beschrieben. Intuitiv betrachtet, verhalten sich die gemäß Fig. 2.1 aus R_1 bzw. R_2 konstruierten Automaten A_1 und A_2 jedoch völlig verschieden voneinander: A_1 geht nach dem Einwurf einer Münze in einen Zustand, in dem es möglich ist, eine zweite Münze einzuwerfen oder einen Tee zu ziehen. Bei A_2 hingegen muss man diese Entscheidung bereits vor dem Einwurf der ersten Münze fällen! "If you bought a vending machine and found it behave like this, you would ask for your money back" ([46], Seite 15). Diese Beobachtung war der Ausgangspunkt für die Suche nach alternativen Äquivalenzbegriffen. Heraus kamen *Bisimulationen*, die im Gegensatz zur Pfadsemantik intuitive Unterschiede wie den zwischen R_1 und R_2 formal fassen können.

Definition 2.3.2 Ein deterministischer Automat A ist **erreichbar**, wenn seine Erreichbarkeitsfunktion r_A surjektiv ist, d.h. anschaulich, wenn alle Zustände von A durch eine Eingabe von $q_{0,A}$ aus erreichbar sind.

Beobachtungsfunktion

$$\sigma_A : Q \longrightarrow [X^* \to Y]$$
$$\sigma_A(q)(w) \ =_{def} \ \beta_A \circ \delta_A^*(q, w)$$

Ein deterministischer Automat A ist **beobachtbar**, wenn seine Beobachtungsfunktion σ_A injektiv ist, d.h. anschaulich, wenn je zwei Zustände q und q' voneinander unterscheidbar sind, weil es mindestens eine Eingabe w gibt, die, ausgehend von q bzw. q', zu verschiedenen Ausgaben führen. ❑

Satz 2.3.3 Ist A erreichbar und beobachtbar, dann ist die Anzahl der Zustände von A minimal innerhalb der Menge aller erreichbaren Automaten mit dem Verhalten von A.

Beweis. Sei B ein erreichbarer Automat mit $f_B = f_A$. Dann gilt für alle $v, w \in X^*$,

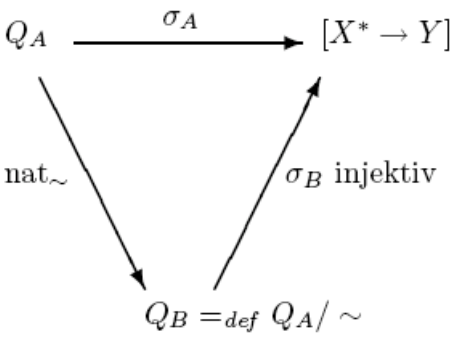
$$\begin{aligned}&\sigma_A \circ r_A(w)(v) = \sigma_A(\delta_A^*(q_{0,A}, w))(v) = \beta_A \circ \delta_A^*(\delta_A^*(q_{0,A}, w), v) = \beta_A \circ \delta_A^*(q_{0,A}, wv)\\ &= f_A(wv) = f_B(wv) = \cdots = \sigma_B \circ r_B(w)(v). \qquad (1)\end{aligned}$$

Die Funktion $h : Q_B \to Q_A$ mit $h(r_B(w)) =_{def} r_A(w)$ ist wohldefiniert: Da B erreichbar ist, gilt $Q_B = r_B(X^*)$. Seien $v, w \in X^*$ mit $r_B(v) = r_B(w)$. Dann folgt

$$\sigma_A(r_A(v)) = \sigma_A(h(r_B(v))) = \sigma_A(h(r_B(w))) = \sigma_A(r_A(w))$$

aus (1), was wiederum $r_A(v) = r_A(w)$ impliziert, weil A beobachtbar ist. h ist surjektiv, weil A erreichbar ist. Damit folgt aus der Definition von h sofort, dass A eine minimale Zustandsmenge unter allen erreichbaren äquivalenten Automaten B hat. ❑

Die **Konstruktion eines beobachtbaren Automaten** B aus einem erreichbaren Automaten A basiert auf dem **Homomorphiesatz** für endliche Automaten:



$$\begin{array}{lll} q \sim q' & \Longleftrightarrow_{def} & \sigma_A(q) = \sigma_A(q') \\ \sigma_B([q]_\sim) & =_{def} & \sigma_A(q) \end{array}$$

$$B =_{def} (Q_B, X, Y, \delta_B, \beta_B, q_{0,B})$$

$$\begin{array}{lll} \delta_B([q]_\sim, x) & =_{def} & [\delta_A(q,x)]_\sim \\ \beta_B([q]_\sim) & =_{def} & \beta_A(q) \\ q_{0,B} & =_{def} & [q_{0,A}]_\sim \end{array}$$

Iterative Konstruktion der **Zustandsäquivalenz** $\sim$:

$$\begin{array}{rcl} q \sim_0 q' & \Longleftrightarrow_{def} & \beta_A(q) = \beta_A(q') \\ q \sim_{k+1} q' & \Longleftrightarrow_{def} & q \sim_k q' \wedge \forall x \in X : \delta_A(q,x) \sim_k \delta_A(q',x) \\ q \sim q' & \Longleftrightarrow_{def} & q \sim_{|Q_A|} q' \Longleftrightarrow_{def} q \sim_k q', \text{ wobei } k \text{ bzgl. der Eigenschaft } \sim_k = \sim_{k+1} \text{ minimal ist} \end{array}$$

Aufwand: $O(|Q_A|^2|X|)$

Zusammen mit Satz 2.3.3 folgt aus dieser Konstruktion, dass die Überführung eines deterministischen Automaten in einen äquivalenten minimalen Automaten berechenbar ist.

Definition 2.3.4 Eine Abbildung $h : Q_A \to Q_B$ zwischen den Zustandsmengen zweier Automaten A und B mit denselben Ein- und Ausgabemengen X bzw. Y heißt **Homomorphismus**, wenn $h(q_{0,A}) = q_{0,B}$, $h(\delta_A(q,x)) = \delta_B(h(q),x)$ für alle $x \in X$ und $\beta_A = \beta_B \circ h$ gilt. A und B sind isomorph, wenn h darüberhinaus bijektiv ist. ❑

Wie man leicht nachrechnet, sind zwei äquivalente minimale Automaten isomorph. Dies und die Berechenbarkeit minimaler Automaten liefert sofort folgenden

Satz 2.3.5 Die Äquivalenz zweier endlicher Automaten ist entscheidbar. ❑

Das **Paull-Unger-Verfahren** zur Berechnung äquivalenter Zustände erstellt die Relation

$$R_0 = \{(q, q', \{(\delta(q,x), \delta(q',x)) \mid x \in X\}) \mid q \sim_0 q'\}$$

und wendet die Funktion

$$\begin{array}{rcl} \Phi : \wp(Q \times Q \times \wp(Q \times Q)) & \to & \wp(Q \times Q \times \wp(Q \times Q)) \\ R & \mapsto & \{(q, q', pairs) \in R \mid \forall\ (q_1, q_2) \in pairs\ \exists\ pairs' : (q_1, q_2, pairs') \in R\} \end{array}$$

auf R_0 an, dann auf $\Phi(R_0)$, usw., bis ein Fixpunkt von Φ erreicht ist. Die Projektion auf seine ersten beiden Komponenten stimmt mit $\sim$ überein.

In der folgenden in das Modellierungstool Expander2 [60] eingebauten Haskell-Implementierung des Paull-Unger-Verfahrens sind *transL* und *value* Listenimplementierungen von δ bzw. β. *valueL* wird nur aufgerufen, wenn der Automat ein **Mealy-Automat** ist, der sich von den oben behandelten **Moore-Automaten** dadurch unterscheidet, dass seine Ausgabefunktion nicht nur vom aktuellen Zustand, sondern auch von der aktuellen Eingabe abhängt. Zustände werden als ganze Zahlen dargestellt.

```
nerode :: Sig -> [Int] -> [Int] -> [(Int,Int)]
nerode sig sts labs = map (\(i,j,_) -> (i,j)) $ gfp f start
  where start = [(i,j,new i j) | i <- sts, j <- sts, c i j]
        c i j = i < j && eqVals sig labs i j
        new i j = mkSet $ (0,0):[(k,l) | (k,l) <- ps, k /= l]
                  where ps = if null labs then [(min k l,max k l)]
                                          else map h labs
                        ([k],[l]) = (sig.trans!!i,sig.trans!!j)
                        h a = (min k l,max k l)
                          where ([k],[l]) = (sig.transL!!i!!a,sig.transL!!j!!a)
        f old = foldl g old old
                where g rel trip@(_,_,ps) = if all h ps then rel
                                                        else rel`minus1`trip
                      h (k,l) = k == l && just (lookupL k l old)

gfp :: Eq a => ([a] -> [a]) -> [a] -> [a]                        -- greatest fixpoint
gfp f s = if s `subset` fs then s else gfp f fs where fs = f s

lookupL :: (Eq a,Eq b) => a -> b -> [(a,b,c)] -> Maybe c
lookupL a b ((x,y,z):s) = if a == x && b == y then Just z else lookupL a b s
lookupL _ _ _           = Nothing
```

Die lokal definierte Funktion `f` implementiert Φ (s.o.).

Beispiel 2.5

	D	dot	plus	minus	E
1	2	10	10	10	10
2	3	4	10	10	10
10	10	10	10	10	10
3	5	4	10	10	10
4	6	10	10	10	7
5	5	4	10	10	10
6	6	10	10	10	7
7	9	10	8	8	10
9	9	10	10	10	10
8	9	10	10	10	10

	final
2	●
3	●
4	●
5	●
6	●
9	●

Figure 2.5. *Übergangsfunktion (links) und Ausgabefunktion (rechts) eines (erkennenden) deterministischen Mooreautomaten*

$\sim$ ist hier der reflexiv-transitive Abschluss von $\{(2,3),(3,5),(4,6)\}$.

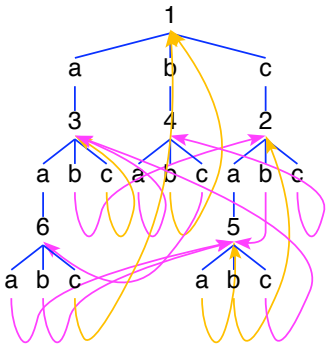
	1	2	10	3	4	5	6	7	9	8
1			1,1 2,10					1,1 2,9 10,8		1,1 2,9
2				1,1 3,5	1,1 3,6 10,4 10,7	1,1 3,5	1,1 3,6 10,4 10,7		1,1 3,9 10,4	
10								1,1 10,9 10,8		1,1 10,9
3					1,1 5,6 10,4 10,7	1,1	1,1 5,6 10,4 10,7		1,1 5,9 10,4	
4						1,1 5,6 10,4 10,7	1,1		1,1 6,9 10,7	
5							1,1 5,6 10,4 10,7		1,1 5,9 10,4	
6									1,1 6,9 10,7	
7										1,1 10,8
9										
8										

	1	2	10	3	4	5	6	7	9	8
1								1,1 2,9 10,8		1,1 2,9
2				1,1 3,5		1,1 3,5				
10										
3						1,1				
4							1,1		1,1 6,9 10,7	
5										
6									1,1 6,9 10,7	
7										1,1 10,8
9										
8										

	1	2	10	3	4	5	6	7	9	8
1										
2				1,1 3,5		1,1 3,5				
10										
3						1,1				
4							1,1			
5										
6										
7										
9										
8										

Figure 2.6. *Anfangs-, Zwischen- und Endzustand des auf den Automaten von Fig. 2.5 angewendeten Paull-Unger-Verfahrens*

Beispiel 2.7



	a	b	c
1	3	4	2
2	5	5	4
3	6	2	3
4	3	1	6
5	5	2	3
6	5	5	1

	a	b	c
1		final	final
2	final	final	
3	final	final	
4		final	final
5	final	final	
6	final	final	

Figure 2.7. *Übergangsfunktion (als Graph und Matrix) und Ausgabefunktion eines (erkennenden) deterministischen Mealyautomaten*

$\sim$ ist hier der reflexiv-transitive Abschluss von $\{(1,4),(2,6)\}$.

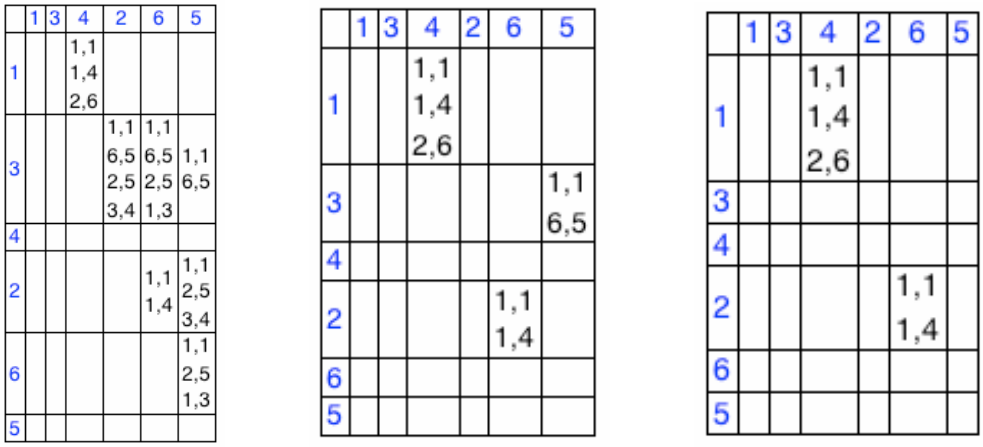


Figure 2.8. *Anfangs-, Zwischen- und Endzustand des auf den Automaten von Fig. 2.7 angewendeten Paull-Unger-Verfahrens*

2.4 Beispiel zur Compilerverifikation: Ein Formelübersetzer

Im folgenden soll das Compilerdiagramm von Def. 1.1.1 einmal ganz formal an einem kleinen Beispiel exemplifiziert werden. Es geht dabei weniger um das Beispiel selbst als um den grundsätzlichen Weg, die **Korrektheit** eines Übersetzers zu zeigen. Voraussetzung dafür ist nämlich, dass man nicht nur den Compiler $comp : Q \to Z$ selbst entwirft, sondern auch die Bedeutung sowohl der Quellsprache Q als auch der Zielsprache Z präzisiert (als Interpreterfunktionen $I_Q : Q \to M_Q$ bzw. $I_Z : Z \to M_Z$) und schließlich die Einbettung des semantischen Bereichs von Q in den semantischen Bereich von Z (als Funktion $encode : M_Q \to M_Z$). Dann erst kann $comp$ im Sinne der Kommutativität des Diagramms von Def. 1.1.1 überhaupt verfiziert werden.

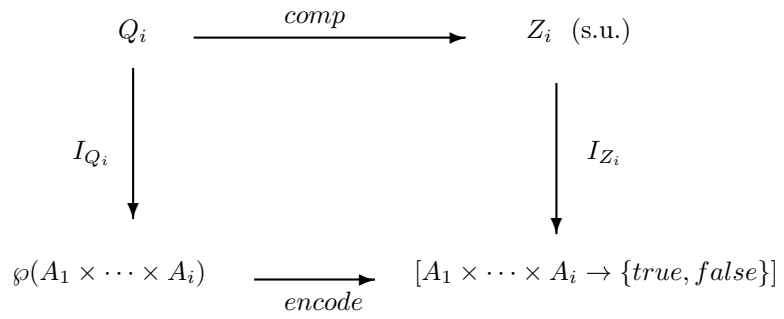
Höchstens n-stellige Relationen sollen durch prädikatenlogische Formeln dargestellt und in funktionale Haskell-Ausdrücke übersetzt werden. Seien $A_1, \ldots, A_n$ endliche Mengen (implementiert als Listen), $\mathcal{R}_i$, $1 \le i \le n$, eine Menge von Teilmengen von $A_1 \times \ldots \times A_i$ und $\overline{r}$ der Name von $r \in \mathcal{R}_i$. Die Formeln werden von folgender CF-Grammatik G erzeugt (s. Def. 3.1.1):

$$G \quad = \quad (\{R, R_1, \ldots, R_n\},\ \{\overline{r} \mid r \in \mathcal{R}_1 \cup \cdots \cup \mathcal{R}_n\} \cup \{x_1, \ldots, x_n, (,), ,, \wedge, \vee, \forall, \exists\},\ P,\ R),$$

wobei P aus folgenden Produktionen besteht: Sei $1 \le i \le n$.

$$\begin{array}{lcll} R & \longrightarrow & R_1 \mid \ldots \mid R_n & \\ R_i & \longrightarrow & \overline{r}(x_1, \ldots, x_i) & \text{für alle } r \in \mathcal{R}_i \\ R_i & \longrightarrow & R_i \wedge R_i & \\ R_i & \longrightarrow & R_i \vee R_i & \\ R_{i-1} & \longrightarrow & \forall x_i : R_i & \\ R_{i-1} & \longrightarrow & \exists x_i : R_i & \end{array}$$

Q_i bezeichne die Menge der aus R_i ableitbaren terminalen Wörter. Das Compilerdiagramm von Def. 1.1.1 gibt es für jedes $1 \le i \le n$:



Definition von I_{Q_i} über dem Aufbau von Q_i:

$$
\begin{array}{lcl}
I_{Q_i}(\overline{r}(x_1,\ldots,x_i)) & = & r \\
I_{Q_i}(F \wedge F') & = & I_{Q_i}(F) \cap I_{Q_i}(F') \\
I_{Q_i}(F \vee F') & = & I_{Q_i}(F) \cup I_{Q_i}(F') \\
I_{Q_{i-1}}(\forall x_i : F) & = & \{(a_1,\ldots,a_{i-1}) \in A_1 \times \cdots \times A_{i-1} \mid \text{für alle } a_i \in A_i \text{ gilt } (a_1,\ldots,a_i) \in I_{Q_i}(F)\} \\
I_{Q_{i-1}}(\exists x_i : F) & = & \{(a_1,\ldots,a_{i-1}) \in A_1 \times \cdots \times A_{i-1} \mid \text{es gibt } a_i \in A_i \text{ mit } (a_1,\ldots,a_i) \in I_{Q_i}(F)\}
\end{array}
$$

Definition von *encode*:

$$encode(r)(a_1,\ldots,a_i) = true \iff_{def} (a_1,\ldots,a_i) \in r \text{ für alle } r \in \mathcal{R}_i.$$

Sei $1 \leq i \leq n$ und $\vec{x}_i = (x_1,\ldots,x_i)$.

Induktive Definition von Z_i: Sei A_i als Liste dargestellt.[4]

$$
\begin{array}{rcl}
 & & \backslash\vec{x}_i \to \overline{r}(\vec{x}_i) \ \in \ Z_i \text{ für alle } r \in \mathcal{R}_i \\
\backslash\vec{x}_i \to e,\ \backslash\vec{x}_i \to e' \ \in \ Z_i & \Longrightarrow & \backslash\vec{x}_i \to (e \ \&\& \ e') \ \in \ Z_i \\
\backslash\vec{x}_i \to e,\ \backslash\vec{x}_i \to e' \ \in \ Z_i & \Longrightarrow & \backslash\vec{x}_i \to (e \ || \ e') \ \in \ Z_i \\
\backslash\vec{x}_i \to e \ \in \ Z_i & \Longrightarrow & \backslash\vec{x}_{i-1} \to all(\backslash x_i \to e)(A_i) \ \in \ Z_{i-1} \\
\backslash\vec{x}_i \to e \ \in \ Z_i & \Longrightarrow & \backslash\vec{x}_{i-1} \to any(\backslash x_i \to e)(A_i) \ \in \ Z_{i-1}
\end{array}
$$

Induktive Definition von *comp* über dem Aufbau von Q_i:

Sei $comp(F) = \backslash\vec{x}_i \to e$ und $comp(F') = \backslash\vec{x}_i \to e'$.

$$
\begin{array}{rcl}
comp(\overline{r}(\vec{x}_i)) & = & \backslash\vec{x}_i \to \overline{r}(\vec{x}_i) \\
comp(F \wedge F') & = & \backslash\vec{x}_i \to (e \ \&\& \ e') \\
comp(F \vee F') & = & \backslash\vec{x}_i \to (e \ || \ e') \\
comp(\forall x_i : F) & = & \backslash\vec{x}_{i-1} \to all(\backslash x_i \to e)(A_i) \\
comp(\exists x_i : F) & = & \backslash\vec{x}_{i-1} \to any(\backslash x_i \to e)(A_i)
\end{array}
$$

Definition von I_{Z_i} **über dem Aufbau von** Z_i:

- $I_{Z_i}(\backslash\vec{x}_i \to \overline{r}(\vec{x}_i))(a) = true \iff a \in r$
- $I_{Z_i}(\backslash\vec{x}_i \to (e \ \&\& \ e'))(a) = true \iff I_{Z_i}(\backslash\vec{x}_i \to e)(a) = true$ und $I_{Z_i}(\backslash\vec{x}_i \to e')(a) = true$

[4] Die Haskell-Funktionen *all* und *any* sind in Abschnitt 1.2 definiert.

- $I_{Z_i}(\backslash \vec{x}_i \to (e \mid\mid e'))(a) = true \iff I_{Z_i}(\backslash \vec{x}_i \to e)(a) = true$ oder $I_{Z_i}(\backslash \vec{x}_i \to e')(a) = true$
- $I_{Z_{i-1}}(\backslash \vec{x}_{i-1} \to all(\backslash x_i \to e)(A_i))(a_1, \ldots, a_{i-1}) = true$
  $\iff$ für alle $a_i \in A_i$ gilt $I_{Z_i}(\backslash \vec{x}_i \to e)(a_1, \ldots, a_i) = true$
- $I_{Z_{i-1}}(\backslash \vec{x}_{i-1} \to any(\backslash x_i \to e)(A_i))(a_1, \ldots, a_{i-1}) = true$
  $\iff$ es gibt $a_i \in A_i$ mit $I_{Z_i}(\backslash \vec{x}_i \to e)(a_1, \ldots, a_i) = true$

Satz 2.4.1 *comp* ist korrekt bzgl. der Quell- bzw. Zielspracheninterpreter, d.h. für alle $1 \leq i \leq n$ gilt:

$$encode \circ I_{Q_i} = I_{Z_i} \circ comp.$$

Beweis. Nach Definition von *encode* bleibt für alle $F \in Q_i$ und $a \in A_1 \times \ldots \times A_i$ zu zeigen:

$$a \in I_{Q_i}(F) \iff I_{Z_i}(comp(F))(a) = true. \tag{2.1}$$

Beweis von (2.1) durch Induktion über den Aufbau von Q_i:

$$\begin{array}{ll} & a \in I_{Q_i}(\overline{r}(\vec{x}_i)) \\ \overset{Def.\ I_{Q_i}}{\iff} & a \in r \\ \overset{Def.\ I_{Z_i}}{\iff} & I_{Z_i}(\backslash \vec{x}_i \to \overline{r}(\vec{x}_i))(a) = true \\ \overset{Def.\ comp}{\iff} & I_{Z_i}(comp(\overline{r}(\vec{x}_i)))(a) = true. \end{array}$$

$$\begin{array}{ll} & a \in I_{Q_i}(F \wedge F') \\ \overset{Def.\ I_{Q_i}}{\iff} & a \in I_{Q_i}(F) \cap I_{Q_i}(F') \\ \overset{Ind.vor.}{\iff} & I_{Z_i}(comp(F))(a) = true \text{ und } I_{Z_i}(comp(F'))(a) = true \\ \overset{Def.\ I_{Z_i}}{\iff} & I_{Z_i}(\backslash \vec{x}_i \to (e \;\&\&\; e'))(a) = true, \\ & \text{wobei } comp(F) = \backslash \vec{x}_i \to e \text{ und } comp(F') = \backslash \vec{x}_i \to e' \\ \overset{Def.\ comp}{\iff} & I_{Z_i}(comp(F \wedge F'))(a) = true. \end{array}$$

$F \vee F'$ analog.

$$\begin{array}{ll} & (a_1, \ldots, a_{i-1}) \in I_{Q_{i-1}}(\forall x_i : F) \\ \overset{Def.\ I_{Q_{i-1}}}{\iff} & \text{für alle } a_i \in A_i \text{ gilt } (a_1, \ldots, a_i) \in I_{Q_i}(F) \\ \overset{Ind.vor.}{\iff} & \text{für alle } a_i \in A_i \text{ gilt } I_{Z_i}(comp(F))(a_1, \ldots, a_i) = true \\ \overset{Def.\ I_{Z_{i-1}}}{\iff} & \text{für alle } a_i \in A_i \text{ gilt } I_{Z_{i-1}}(\backslash \vec{x}_{i-1} \to all(\backslash x_i \to e)(A_i))(a) = true, \\ & \text{wobei } comp(F) = \backslash \vec{x}_i \to e \\ \overset{Def.\ comp}{\iff} & I_{Z_{i-1}}(comp(\forall x_i : F))(a) = true. \end{array}$$

$\exists x_i : F$ analog. ❑

Beispiel 2.4.2 Noch ein Haskell-Programm zur Berechnung der Verhaltensäquivalenz.

Wir benutzen den Formelübersetzer, um die Definition der Verhaltensäquivalenz $\sim$ (siehe §2.3) in ein Haskell-Programm zu übertragen. Wir führen die folgenden vierstelligen Hilfsrelationen ein:

$$\begin{array}{lll} (k, q, q') \in r_0 & \iff_{def} & k = 0 \wedge \beta(q) = \beta(q') \\ (k, q, q') \in r_1 & \iff_{def} & k > 0 \wedge q \sim_{k-1} q' \\ (k, q, q', x) \in r_2 & \iff_{def} & k > 0 \wedge \delta(q, x) \sim_{k-1} \delta(q', x) \end{array}$$

$\sim_k$, $k > 0$, lässt sich demnach durch die Formel

$$\overline{r}_0(k, q, q') \vee (\overline{r}_1(k, q, q') \wedge \forall x : \overline{r}_2(k, q, q', x))$$

darstellen. Der Formelübersetzer transformiert sie in den funktionalen Haskell-Ausdruck

$$\backslash(k, q, q') \rightarrow (\overline{r}_0(k, q, q') \texttt{ || } \overline{r}_1(k, q, q') \texttt{ \&\& } all(x \rightarrow \overline{r}_2(k, q, q', x)) \ X).$$

Das Haskell-Programm für $\sim$ lautet damit wie folgt:

```
r0(k,q,q') = k = 0 && beta(q) = beta(q')
r1(k,q,q') = k > 0 && equiv_k(k-1,q,q')
r2(k,q,q',x) = k > 0 && equiv_k(k-1,delta(q,x),delta(q',x))
equiv_k(k,q,q') = r0(k,q,q') || r1(k,q,q') && all(\x -> r2(k,q,q',x)) X
equiv(q,q') = equiv_k(|Q|,q,q')
```

Aufgabe Überführen Sie auf die gleiche Weise die in Abschnitt 2.4 angegebene Definition der Inäquivalenz $\not\sim$ mit *comp* in ein Haskell-Programm!

Kapitel 3

Parser + Algebra = Compiler

3.1 Kontextfreie Sprachen

Die Aufgaben der **Syntaxanalyse** sind:

- Erkennung der Struktur des Quellprogramms gemäß einer kontextfreien Grammatik der Quellsprache,
- Erkennung von Syntaxfehlern,
- Transformation des als Symbolfolge gegebenen Quellprogramms in einen Syntaxbaum.

Wir werden hier noch einen großen Schritt weiter gehen und den Formalismus, der den Syntaxbäumen zugrundeliegt, nutzen, um die vorgestellten Parser in vollständige Compiler zu transformieren.

Definition 3.1.1 Eine **erweiterte CF-Grammatik (ECFG)** $G = (N, T, P, S)$ besteht aus

- einer endlichen Menge N von **Nichtterminalen**,
- einer endlichen Menge T von **Terminalen**,
- einer endlichen Menge P von **Produktionen** oder **Regeln** der Form $A \to e$ mit $A \in N$ und $e \in Reg(N \cup T)$,
- einem **Startsymbol** $S \in N$.

O.B.d.A. enthalte P für jedes Nichtterminal $A \in N$ höchstens eine Regel $A \to e$.

G heißt **CF-Grammatik (CFG)**, falls für alle $A \to e \in P$ e in disjunktiver Normalform ist. Hierfür heißt

$$L(G) = \{w \in T^* \mid S \overset{*}{\to}_G w\}$$

die **von G erzeugte Sprache**.[1] ❑

Für ECFGs muss der Sprachbegriff verallgemeinert werden. Wir verbinden diese Verallgemeinerung gleich mit einer weiteren: Da zur Bestimmung von $L(G)$ nach obiger Definition meistens auch die Mengen der aus anderen Nichtterminalen ableitbaren Wörter bekannt sein müssen, definieren wir $L(G)$ als Mengenfamilie:

Sei $N = \{A_1, \ldots, A_n\}$, $P = \{A_1 \longrightarrow e_1, \ldots, A_n \longrightarrow e_n\}$ und $S = A_1$. Die **von G erzeugte Sprache**

$$L(G) \;=\; \{L(G)_{A_1}, \ldots, L(G)_{A_n}\}$$

ist die **kleinste Lösung des Gleichungssystems**

$$A_1 \;=\; e_1, \;\; \ldots, \;\; A_n \;=\; e_n, \tag{3.1}$$

[1] Zur Definition von $\overset{*}{\to}_G$ vgl. Literatur über *Formale Sprachen*, z.B. [84].

d.i. das komponentenweise kleinste Tupel $(L_1, \ldots, L_n) \in \wp(T^*)^n$ derart, dass die Gleichungen

$$L_1 \;=\; L(e_1)[L_1/A_1, \ldots, L_n/A_n], \;\; \ldots, \;\; L_n \;=\; L(e_n)[L_1/A_1, \ldots, L_n/A_n]$$

gelten. Hierbei ist die Funktion L auf den regulären Ausdrücken $e_1, \ldots, e_n$ wie oben definiert. $[L_1/A_1, \ldots, L_n/A_n]$ bezeichnet die Substitution von A_i durch L_i ("L_i für A_i") für alle $1 \leq i \leq n$.

Für einige Nichtterminale A kann anstelle von Regeln für A die Sprache $L(G)_A$ vordefiniert sein, z.B. $L(G)_{Int} =_{def} \mathbb{Z}$. Zum Gleichungssystem (1) kommt dann

$$A \;=\; L(G)_A$$

hinzu, im Beispiel: $Int = \mathbb{Z}$. ❑

Jede ECFG $G = (N, T, P, S)$ lässt sich in eine äquivalente CFG überführen:

- Ersetze für alle $p \in P$ alle echten Teilausdrücke $e = e_1|\ldots|e_n$ der rechten Seite von p durch ein neues Nichtterminal A_e und füge die Regel $A_e \to e_1|\ldots|e_n$ zu P hinzu.
- Ersetze für alle $p \in P$ alle echten Teilausdrücke $e = e_1^*$ der rechten Seite von p durch ein neues Nichtterminal A_e und füge die Regel $A_e \to eA_e|\varepsilon$ zu P hinzu.
- Ersetze für alle $p \in P$ alle echten Teilausdrücke $e = e_1^+$ der rechten Seite von p durch ein neues Nichtterminal A_e und füge die Regel $A_e \to eA_e|e$ zu P hinzu.
- Ersetze für alle $p \in P$ alle echten Teilausdrücke $e = e_1?$ der rechten Seite von p durch ein neues Nichtterminal A_e und füge die Regel $A_e \to e|\varepsilon$ zu P hinzu.
- Wiederhole diese Schritte für die modifizierte Grammatik, bis die rechte Seite jeder Regel in disjunktiver Normalform ist.

CF(G) bezeichne die in dieser Weise aus G gebildete CFG. Für alle $A \in N$ gilt:

$$L(G)_A \quad = \quad \{w \in T^* \mid A \stackrel{*}{\to}_{CF(G)} w\}.$$

Die Eigenschaft, dass es für jedes Nichtterminal A genau eine Regel in G gibt, gilt auch für $CF(G)$. Daraus folgt insbesondere:

Enthält keine Regel von $CF(G)$ den Summenoperator $|$, dann ist $L(G)_A$ höchstens einelementig und damit regulär! Z.B. wird die nichtreguläre Sprache $\{a^n b^n \mid n \in \mathbb{N}\}$ von $S \to aSb|\varepsilon$ erzeugt. Ohne den ε-Fall wäre die erzeugte Sprache leer.

Eine Grammatik einer Programmiersprache nennt man auch deren **konkrete Syntax**.

Beispiel 3.1.2 (konkrete Syntax einer funktionalen Sprache) Eine funktionale Sprache mit verketteten (geschachtelten) Listen als grundlegender Datenstruktur (wie in Lisp) und nicht-applikativen Funktionsausdrücken (wie in FP; vgl. [5]), wird durch folgende ECFG definiert:[2] 1.3

★ $FPF = (N, T, P, S)$

★ $N = \{const, fun\}$

★ $T = \{0, \ldots, 9, [], <, >, id, head, tail, num, +, =, \equiv, \circ, [,], if, then, else, ,, \alpha, /\}$

[2] "functional programming fragment"; nach [43]

★ P besteht aus den Produktionen

$$\begin{array}{lcl} \mathit{pdigit} & \longrightarrow & \texttt{1} \mid \texttt{2} \mid ... \mid \texttt{9} \\ \mathit{digit} & \longrightarrow & \texttt{0} \mid \mathit{pdigit} \\ \mathit{const} & \longrightarrow & \mathit{pdigit}\ \mathit{digit}^* \mid <> \mid < \mathit{const}\ (,\mathit{const})^* > \\ \mathit{fun} & \longrightarrow & \texttt{id} \mid \texttt{head} \mid \texttt{tail} \mid \texttt{num} \mid + \mid = \mid \equiv \texttt{const} \mid \\ & & \mathit{fun} \circ \mathit{fun} \mid [\mathit{fun}\ (,\mathit{fun})^*] \mid \\ & & \texttt{if}\ \mathit{fun}\ \texttt{then}\ \mathit{fun}\ \texttt{else}\ \mathit{fun} \mid \alpha\mathit{fun} \mid /\mathit{fun} \end{array}$$

Wie auch in den folgenden Beispielen gehören die Symbole |, $^+$, *, (und) zu regulären Operatoren, währende alle anderen Symbole Terminale oder Nichtterminale sind. Runde Klammern werden unterstrichen, wenn sie Terminale darstellen.

★ $S = \mathit{fun}$

Also ist $L(FPF)$ die Menge der syntaktisch korrekten FPF-Programme. ❑

Beispiel 3.1.3 (konkrete Syntax einer imperativen Sprache) Eine imperative Sprache mit den üblichen nicht-rekursiven Kontrollstrukturen ist durch folgende ECFG gegeben:[3]

★ $IPF = (N, T, P, S)$

★ $N = \{\mathit{const}, \mathit{var}, \mathit{exp}, \mathit{boolexp}, \mathit{com}\}$

★ $T =$
$\{0, \ldots, 9, a, \ldots, z, A, \ldots, Z, +, =, \mathit{and}, :, ;, \texttt{if}, \texttt{then}, \texttt{else}, \texttt{while}, \texttt{do}, \texttt{repeat}, \texttt{until}, \texttt{skip}, \mathit{true}, \mathit{false}, >\}$

★ P besteht aus den Produktionen

$$\begin{array}{lcl} \mathit{pdigit} & \longrightarrow & \texttt{1} \mid \texttt{2} \mid ... \mid \texttt{9} \\ \mathit{digit} & \longrightarrow & \texttt{0} \mid \mathit{pdigit} \\ \mathit{letter} & \longrightarrow & \texttt{a} \mid \texttt{b} \mid ... \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid ... \mid \texttt{Z} \\ \mathit{char} & \longrightarrow & \mathit{letter} \mid \mathit{digit} \\ \mathit{const} & \longrightarrow & \mathit{pdigit}\ \mathit{digit}^* \\ \mathit{var} & \longrightarrow & \mathit{letter}\ \mathit{char}^* \\ \mathit{exp} & \longrightarrow & \mathit{const} \mid \mathit{var} \mid \mathit{exp} + \mathit{exp} \\ \mathit{boolexp} & \longrightarrow & \mathit{true} \mid \mathit{false} \mid \mathit{exp} = \mathit{exp} \mid \mathit{exp} > \mathit{exp} \mid \\ & & \mathit{boolexp}\ \mathit{and}\ \mathit{boolexp} \mid \neg\ \mathit{boolexp} \\ \mathit{com} & \longrightarrow & \texttt{skip} \mid \mathit{var} := \mathit{exp} \mid \mathit{com}; \mathit{com} \mid \\ & & \texttt{if}\ \mathit{boolexp}\ \texttt{then}\ \mathit{com}\ \texttt{else}\ \mathit{com} \mid \\ & & \texttt{while}\ \mathit{boolexp}\ \texttt{do}\ \mathit{com} \end{array}$$

★ $S = \mathit{com}$

Also ist $L(IPF)$ die Menge der syntaktisch korrekten IPF-Programme. ❑

Beispiel 3.1.4 (konkrete Syntax eines Javafragments) Der in Beispiel 1.2.3 behandelten imperativen Sprache könnte folgende ECFG zugrundeliegen:

★ $\mathit{JavaGra} = (N, T, P, S)$

[3] "imperative programming fragment"

★ $N = \{Block, Command, IntE, BoolE, Int, String, Bool\}$

★ $T = \{+, -, *, =, ;, \texttt{if}, \texttt{else}, \texttt{while}, >, !, (,), \{, \}\}$

★ P besteht aus den Produktionen

$$\begin{array}{lcl} Block & \longrightarrow & \{Command^*\} \\ Command & \longrightarrow & ; \mid String = IntE; \mid \texttt{if}\ (BoolE)\ Block \mid \texttt{if}\ (BoolE)\ Block\ \texttt{else}\ Block \mid \\ & & \texttt{while}\ (BoolE)\ Block \\ IntE & \longrightarrow & Int \mid String \mid (IntE) \mid IntE - IntE \mid IntE(+IntE)^+ \mid IntE(*IntE)^+ \\ BoolE & \longrightarrow & Bool \mid IntE > IntE \mid !\ BoolE \end{array}$$

★ $S = Block$

★ Für die Nichtterminale Int, $String$ und $Bool$ gibt es keine Regeln. Die Sprachen für diese Nichtterminale sollen per definitionem mit den gleichnamigen Haskell-Typen übereinstimmen.

Also ist $L(JavaGra)$ die Menge der syntaktisch korrekten JavaGra-Programme und

```
{fact = 1; while (x > 0) {fact = fact*x; x = x-1;}}
```

eins ihrer Elemente. ❑

Definition 3.1.5 Sei $G = (N, T, P, S)$ eine ECFG. Die Menge $Abl(G)$ der **Ableitungsbäume** (*parse trees*) von G besteht aus allen mit den Baumersetzungsregeln von Fig. 3.1 aus allen Bäumen, die aus einem mit einem Nichtterminal markierten Blatt ableitbar sind und deren Blättern mit terminalen Wörtern markiert sind.

Ein Ableitungsschritt $uAw \longrightarrow uvw$ mit $u, w \in (N \cup T)^*$ und $A \longrightarrow v \in P$ heißt **direkte Links-** bzw. **Rechtsableitung**, wenn $u \in T^*$ bzw. $w \in T^*$ gilt. Eine Folge $w_1 \to w_2, w_2 \to w_3, \ldots$ direkter Links- bzw. Rechtsableitungen heißt **Links-** bzw. **Rechtsableitung** (*left* bzw. *right* parse).

G ist **eindeutig**, wenn es zu jedem $w \in L(G)$ genau einen Ableitungsbaum mit Wurzel S und Blattfolge w gibt. Eine Sprache $L \subseteq T^*$ ist eindeutig, wenn eine eindeutige Grammatik G mit $L(G) = L$ existiert. ❑

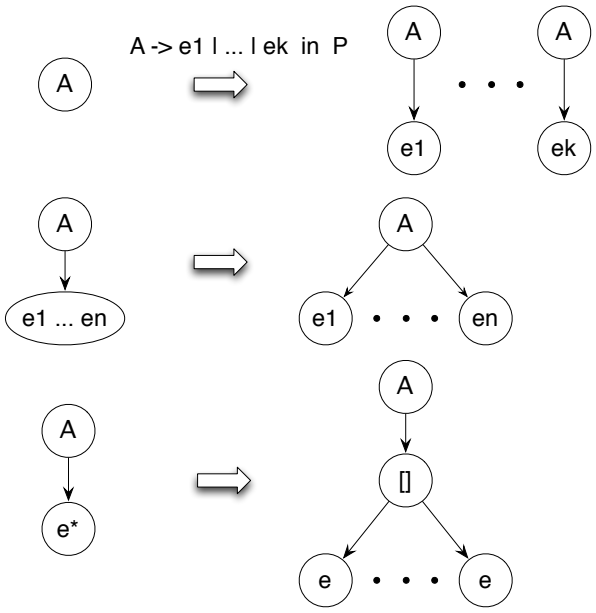


Figure 3.1. *Regeln zur Erzeugung von Ableitungsbäumen einer ECFG*

Jeder Links- und jeder Rechtsableitung $A \xrightarrow{*} w$ entspricht eineindeutig ein Ableitungsbaum mit Wurzel A und Blattfolge w.

Wichtige Ergebnisse aus der Theorie formaler Sprachen

1. Die Eindeutigkeit einer kontextfreien Grammatik ist nicht entscheidbar. (Wird auf die Unentscheidbarkeit des *Postschen Korrespondenzproblems* zurückgeführt.)
2. Die Äquivalenz zweier kontextfreier Grammatiken ist unentscheidbar, im Gegensatz zur Äquivalenz regulärer Grammatiken (vgl. Satz 2.3.5).
3. Deterministisch kontextfreie Sprachen, das sind Sprachen, die von *deterministischen Kellerautomaten*[4] erkannt werden, sind eindeutig.
4. - Es gibt mehrdeutige kontextfreie Sprachen, z.B. $L = \{a^i b^j c^k | i, j, k \geq 1, i = j \vee j = k\}$. (Die Wörter der Form $a^i b^i c^i$ haben immer zwei verschiedene Ableitungsbäume.)
   - $L = \{a^i b^i | i \geq 1\}$ ist eindeutig und kontextfrei, aber nicht regulär. (Ein Automat, der L erkennt, müßte unendlich viele Zustände haben. Warum?)
   - $L = \{a^i b^i c^i | i \geq 1\}$ ist nicht kontextfrei.
5. Eine kontextfreie Grammatik G ist **selbsteinbettend**, wenn es eine Ableitung $A \xrightarrow{+} \alpha A \beta$ mit $\alpha, \beta \in (N \cup T)^+$ gibt. Nicht-selbsteinbettende kontextfreie Sprachen sind regulär. Selbsteinbettung entspricht nicht-iterativer Rekursion (siehe Kapitel 6). Eine kontextfreie Sprache ist selbsteinbettend, wenn alle sie erzeugenden Grammatiken selbsteinbettend sind.

Beweis von $L(G) = L$

Gegeben seien eine ECFG (N, T, P, S) und eine Sprache $L_1 \subseteq T^*$, die mit $L(G)_S$ übereinstimmen soll. Die Definition von $L(G)$ (siehe Def. 3.1.1) erlaubt es uns, $L_1 = L(G)_S$ in drei Schritten zu zeigen:

Sei $N = \{A_1, \ldots, A_n\}$, $P = \{A_1 \longrightarrow e_1, \ldots, A_n \longrightarrow e_n\}$ und $S = A_1$. L_1 ist gegeben.

1. (*Verallgemeinerung*) Finde geeignete Sprachen $L_2, \ldots, L_n \subseteq T^*$, für die $L_2 = L(G)_{A_2}, \ldots, L_n = L(G)_{A_n}$ gelten soll.
2. (*Korrektheit*) Zeige, dass $(L_1, \ldots, L_n)$ eine Lösung des Gleichungssystems

   $$A_1 \;=\; e_1, \;\; \ldots, \;\; A_n \;=\; e_n$$

   ist. Da $L(G)$ als *kleinste* Lösung dieses Gleichungssystems definiert ist, folgt sofort:

   $$L(G)_{A_1} \subseteq L_1, \ldots, L(G)_{A_n} \subseteq L_n.$$

3. (*Vollständigkeit*) Zeige die Umkehrung $L_1 \subseteq L(G)_{A_1}, \ldots, L_n \subseteq L(G)_{A_n}$. Sind die Sprachen $L_1, \ldots, L_n$ unendlich, dann wird es hierzu erforderlich sein, auf ihren induktiven Aufbau zurückzugreifen.

3.2 Konkrete Syntax, abstrakte Syntax und deren Semantik

Die kontextfreie Grammatik G einer Programmierprache PS heißt auch **konkrete Syntax** von PS. Die abstrakte Syntax von PS erhält man aus G durch Entfernung der Terminalsymbole und Einführung eines Funktionssymbols für jede Produktion, das dem **Konstruktor** eines Haskell-Datentyps entsprichtt.[5] Abstrakte Programme sind aus Konstruktoren zusammengesetzte **Terme** (= funktionale Ausdrücke). Jedem Nichtterminal von G entspricht ein unstrukturierter Datentyp, der i.a. **Sorte** genannt wird.

[4] Diese werden hier nicht behandelt.
[5] Durch | getrennte Alternativen bilden verschiedene Funktionen.

Definition 3.2.1 (Sortierte Mengen und Funktionen) Sei S eine Menge. Eine Mengenfamilie $A = \{A_s \mid s \in S\}$ heißt **S-sortierte Menge**.

Seien A und B S-sortierte Mengen. Eine Funktionsfamilie $f = \{f_s : A_s \to B_s \mid s \in S\}$ heißt **S-sortierte Funktion**.

A und f werden wie folgt zur S^*-sortierten Menge bzw. Funktion erweitert:
Seien $s_1, \ldots, s_n \in S$.

$$\begin{array}{lcllcl} A_\varepsilon & = & \{1\}, & f_\varepsilon(1) & = & 1, \\ A_{s_1 \ldots s_n} & = & A_{s_1} \times \ldots \times A_{s_n}, & f_{s_1 \ldots s_n}(a_1, \ldots, a_n) & = & (f_{s_1}(a_1), \ldots, f_{s_n}(a_n)). \end{array}$$

Anstelle von $a \in A_s$ schreibt man auch $a : s \in A$. ❑

Z.B. ist die von einer ECFG (N, T, P, S) erzeugte Sprache ist eine N-sortierte Menge(siehe Def. 3.1.1).

Obwohl die Sprache $L(e)$eines regulären Ausdrucks e und die Menge A_e ähnlich aufgebaut sind, gibt es einen wichtigen Unterschied zwischen beiden und das ist die Bedeutung des Summenoperators $|$: L interpretiert ihn als Vereinigung, A hingegen als *disjunkte* Vereinigung. Warum das so sein muss, zeigt sich beim Übergang von konkreter zu abstrakter Syntax, die im wesentlichen darin besteht, die Terminalsymbole wegzulassen. Fallunterscheidungen, die in der konkreten Syntax auf unterschiedliche Terminalsymbole zurückgehen, muss die Abstraktion erhalten. Genau das gelingt durch die Interpretation von $|$ als disjunkte Vereinigung (siehe Def. 3.2.3 und die darauffolgende Bemerkung).

Die Sortierung von Mengen und Funktionen könnte man auch *Typisierung* nennen, wobei es sich hier allerdings um eine sehr einfache Typisierung handelt. Sorten sind nämlich nicht mehr als Namen. Sorten sind zunächst einmal unstrukturiert, d.h. sie bilden, im Gegensatz zu vielen sortierten Mengen, keine Terme. Eine Logik heißt mehrsortig, wenn die in ihr verwendeten Terme zu einer sortierten Menge gehören, die wie folgt aus einer *mehrsortigen Signatur* gebildet wird:

Definition 3.2.2 (Signaturen und Terme) Eine **Signatur** $\Sigma = (S_0, S, C)$ besteht aus zwei Mengen S_0 und S von **Sorten** mit $S_0 \subseteq S$ und einer S^+-sortierten Menge C von **Konstruktoren**. Die Elemente von S_0 heißen **Basissorten** von Σ.

Anstelle von $c \in C_{ws}$ schreiben wir $c : w \to s \in C$ oder auch $c : w \to s \in \Sigma$. w heißt **Domain** und s **Codomain** von $c : w \to s$. Ist $e = \varepsilon$, dann heißt c **Konstante** und wir schreiben $c : s$ anstelle von $c : \varepsilon \to s$.

Sei X eine S-sortierte Menge von Variablen. Die S-sortierte Menge $T_\Sigma(X)$ der **Σ-Terme über X** ist induktiv definiert:

- Für alle $t : s \in \Sigma \cup X$ ist $t \in T_{\Sigma,s}$.
- Für alle $c : w \to s \in \Sigma$ und $t \in T_{\Sigma,w}$ ist $c(t) \in T_{\Sigma,s}$.

Ist X leer, dann schreibt man T_Σ anstelle von $T_\Sigma(X)$.

Σ ist pure Syntax! Semantik erhält man durch eine **Interpretation** der Sorten und Funktionssymbole als Mengen bzw. Funktionen auf diesen Mengen. Jede Interpretation liefert eine **Σ-Algebra** (siehe unten Def. 3.2.7)
.

Implementierung von $\Sigma = (S_0, S, C)$ durch Datentypen

Für die Elemente von S_0 werden Standardimplementierungen vorausgesetzt, z.B. die Menge der ganzen Zahlen für die Sorte *int*.

Sei $s \in S \setminus S_0$. Gibt es genau einen Konstruktor $c : s_1 \ldots s_n \to s$, dann werden s und c durch

```
type S = (S1,...,Sn)
```

bzw. `(_,..,_)` (n-Tupelbildung) implementiert.

Gibt es genau zwei Konstruktoren $c_1 : s$ und $c_2 : s's \to s$, dann werden s, c_1 und c_2 durch

`type S = [S']`,

durch `[]` (leere Liste) bzw. `:` (append-Funktion) implementiert.

Gibt es genau zwei Konstruktoren $c_1 : s$ und $c_2 : s' \to s$, dann werden s, c_1 und c_2 durch

`type S = Maybe S' (= Just S' | Nothing)`,

`Nothing` bzw. `Just` implementiert.

Gibt es genau $k > 1$ andere Konstruktoren $c_{s1} : s_{11} \dots s_{1n_1} \to s,\ \dots,\ c_{sk} : s_{k1} \dots s_{kn_k} \to s$, dann werden s und c_{si}, $1 \leq i \leq k$, durch

`data S = CS1 S11 ... S1n1 | ... | CSk Sk1 ... Sknk`

bzw. `CSi` implementiert.

Alle neu eingeführten Konstruktoren müssen verschieden voneinander sein!

Die abstrakte Syntax einer kontextfreien Sprache ist eine Signatur, die man wie folgt aus einer ECFG der Sprache gewinnt.

Definition 3.2.3 Seien $G = (N, T, P, S)$ eine ECFG, N_0 und N_1 die Teilmengen von Nichtterminalen A ohne bzw. mit einer Regel $A \to e \in P$,

$$abs : (N \cup T)^* \to N^*$$

die Funktion, die alle Terminale aus einem Wort streicht, $(N', T, P', S) = CF(G)$ und

$$C \ = \ \{c_{A,i} : abs(w_i) \to A \mid (A \longrightarrow w_1 | \dots | w_k) \in P'\}.$$

Die Signatur $\Sigma(G) = (N, C)$ heißt **abstrakte Syntax** von G.

$\Sigma(G)$-Terme heißen **Syntaxbäume** von G. ❑

Beachte, dass z.B. für eine Regel $A \to v|w$ mit terminalen Worten v, w $abs(v|w) = \varepsilon|\varepsilon$ ist und daher zwei Konstruktoren mit Codomain A gebildert werden. Der Summenoperator | wird also hier als disjunkte Vereinigung interpretiert und nicht als Vereinigung wie bei der Definition der Sprache eines regulären Ausdrucks!

Das hängt mit dem Semantikwechsel zusammen, der den Übergang von konkreter zu abstrakter Syntax kennzeichnet: Im Rahmen konkreter Syntax dienen die regulären Operatoren der Bildung von Wortmengen, im Rahmen abstrakter Syntax bilden sie zwar auch Mengen, diese bestehen aber nicht aus Terminalen, sondern aus Elementen von Datentypen, so dass die regulären Operatoren hier Typkonstruktoren entsprechen.

Beispiel 3.2.4 Die abstrakte Syntax $\Sigma(FPF)$ von FPF (siehe Beispiel 3.1.2) lautet wie folgt:

$$
\begin{array}{lcll}
N & = & \{ & pdigit, digit, digits, const, consts, fun, funs \ \} \\
C & = & \{ & 1, 2 \ldots, 9 : & \varepsilon \to pdigit, \\
 & & & 0 : & \varepsilon \to digit, \\
 & & & pos : & pdigit \to digit, \\
 & & & int : & pdigit\ digits \to const, \\
 & & & emptyD : & \varepsilon \to digits, \\
 & & & appendD : & digit\ digits \to digits, \\
 & & & nil : & \varepsilon \to const, \\
 & & & list : & const\ consts \to const, \\
 & & & emptyC : & \varepsilon \to consts, \\
 & & & appendC : & const\ consts \to consts, \\
 & & & id, head, tail, num, add, eq : & \varepsilon \to fun, \\
 & & & cfun : & const \to fun, \\
 & & & comp : & fun\ fun \to fun, \\
 & & & prod : & fun\ funs \to fun, \\
 & & & emptyF : & \varepsilon \to funs, \\
 & & & appendF : & fun\ funs \to funs, \\
 & & & cond : & fun\ fun\ fun \to fun, \\
 & & & map, fold : & fun \to fun \ \}
\end{array}
$$

Beispiel 3.2.5 Die abstrakte Syntax $\Sigma(IPF)$ von IPF (siehe Beispiel 3.1.3) lautet wie folgt:

$$
\begin{array}{lcll}
N & = & \{ & pdigit, digit, digits, letter, char, chars, const, var, exp, boolexp, com \ \} \\
C & = & \{ & 1, 2 \ldots, 9 : & \varepsilon \to pdigit, \\
 & & & 0 : & \varepsilon \to digit, \\
 & & & pos : & pdigit \to digit, \\
 & & & a, b \ldots, z, A, B, \ldots, Z : & \varepsilon \to letter, \\
 & & & charL : & letter \to char, \\
 & & & charD : & digit \to char, \\
 & & & int : & pdigit\ digits \to const, \\
 & & & emptyD : & \varepsilon \to digits, \\
 & & & appendD : & digit\ digits \to digits, \\
 & & & ident : & letter\ chars \to var, \\
 & & & emptyC : & \varepsilon \to chars, \\
 & & & appendC : & char\ chars \to chars, \\
 & & & constE : & const \to exp \\
 & & & varE : & var \to exp \\
 & & & add : & exp\ exp \to exp, \\
 & & & true, false : & \varepsilon \to boolexp, \\
 & & & eq, greater : & exp\ exp \to boolexp, \\
 & & & and : & boolexp\ boolexp \to boolexp, \\
 & & & not : & boolexp \to boolexp, \\
 & & & skip : & \varepsilon \to com, \\
 & & & assign : & var\ exp \to com, \\
 & & & seq : & com\ com \to com, \\
 & & & cond : & boolexp\ com\ com \to com, \\
 & & & loop : & boolexp\ com \to com\ \}
\end{array}
$$

Beispiel 3.2.6 Die abstrakte Syntax $JavaSig = \Sigma(JavaGra)$ des Javafragments von Beispiel 3.1.4 lautet

wie folgt:

$$
\begin{array}{lcll}
N_0 & = & \{ & Int, Bool \ \} \\
N_1 & = & \{ & Block, Command, Commands, IntE, IntEs, BoolE \ \} \\
C & = & \{ & block: & Commands \to Block, \\
 & & & emptyC: & \varepsilon \to Commands, \\
 & & & appendC: & command\ Commands \to Commands, \\
 & & & skip: & \varepsilon \to Command, \\
 & & & assign: & String\ IntE \to Command, \\
 & & & cond: & BoolE\ Block\ Block \to Command, \\
 & & & cond(\_, \_, block[skip]): & BoolE\ Block \to Command, \\
 & & & loop: & BoolE\ Block \to Command, \\
 & & & intE: & Int \to IntE, \\
 & & & var: & String \to IntE, \\
 & & & embed: & IntE \to IntE, \\
 & & & sub: & IntE\ IntE \to IntE, \\
 & & & sum: & IntEs \to IntE, \\
 & & & prod: & IntEs \to IntE, \\
 & & & singleE: & IntE \to IntEs, \\
 & & & appendE: & IntE\ IntEs \to IntEs, \\
 & & & boolE: & Bool \to BoolE, \\
 & & & greater: & IntE\ IntE \to BoolE, \\
 & & & not: & BoolE \to BoolE \ \}
\end{array}
$$

Der Konstruktor $embed : IntE \to IntE$ ergibt sich aus dem Teilausdruck $\underline{(IntE)}$ der *JavaGra*-Regel für $IntE$. (siehe Defn. 3.1.4 und 3.2.2).

Eine Implementierung von *JavaSig* durch Datentypen, die dem o.g. Schema folgt, haben wir bereits in Beispiel 1.2.3 angegeben und verwendet:

```
type Block   = [Command]
data Command = Skip | Assign String IntE | Cond BoolE Block Block | Loop BoolE Block
data IntE    = IntE Int | Var String | Sub IntE IntE | Sum [IntE] | Prod [IntE]
data BoolE   = BoolE Bool | Greater IntE IntE | Not BoolE
```

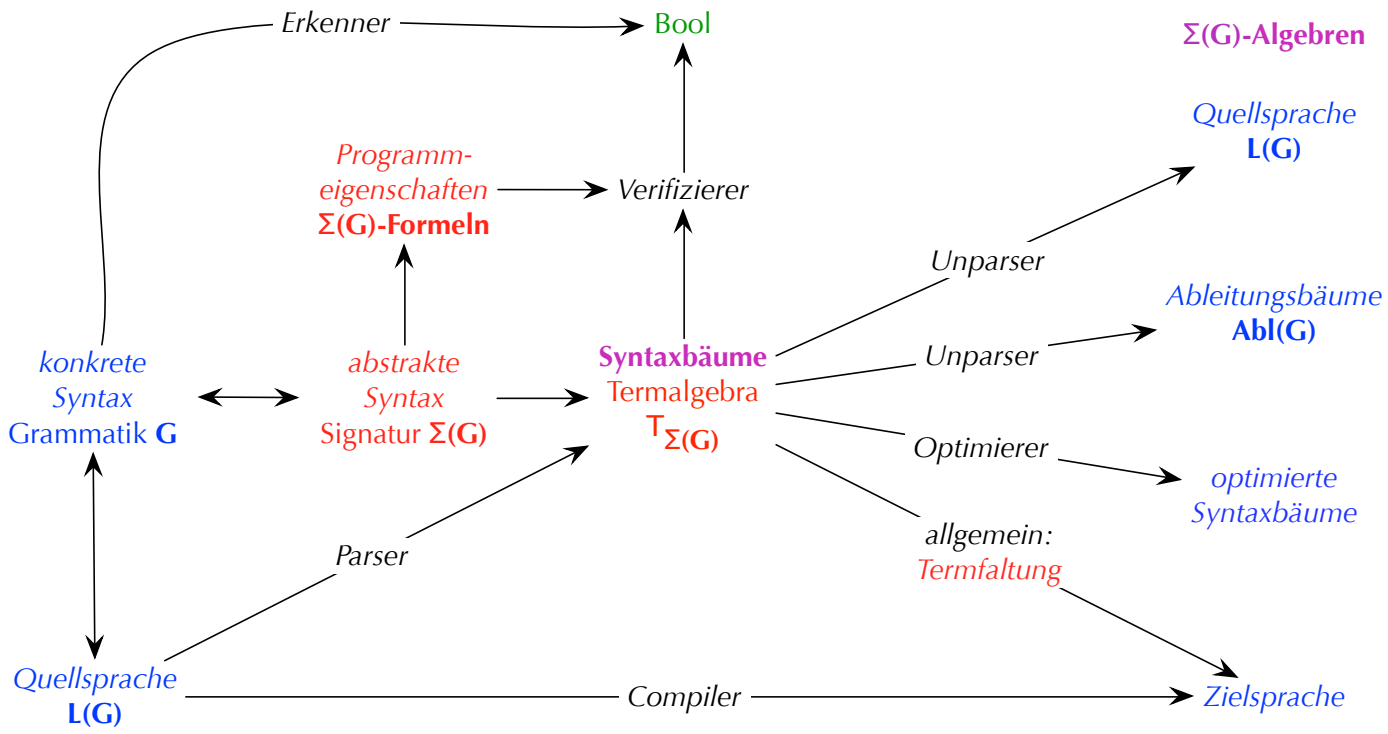


Figure 3.2. *Von Grammatiken zu Algebren*

Im Folgenden wird ein mathematischer Ansatz entwickelt, der es erlaubt, jeden Compiler oder Interpreter als Interpretation von Syntaxbäumen darzustellen und dementsprechend zu implementieren. Syntaxbäume bilden in diesem Zusammenhang nur eine (allerdings eine besondere; s.u.) unter vielen **Algebren**, die eine Signatur interpretieren:

Definition 3.2.7 Sei $\Sigma = (S_0, S, C)$ eine Signatur. Eine **Σ-Algebra** (A, OP), kurz: A, besteht aus einer S-sortierten Menge A und einer Menge OP von Funktionen (**Operationen**), den **Interpretationen** von F:

- Für alle $c : s \in C$ gibt es genau ein Element $c^A \in A$.
- Für alle $c : w \to s \in C$ gibt es genau eine Funktion $c^A : A_w \to A_s \in OP$.

Für alle $s \in S$ heißt A_s **Trägermenge** oder **Datenbereich von** s. ❑

Sei

```
data S1 = C11 e11 | ... | C1n1 e1n1
...
data Sk = Ck1 ek1 | ... | Cknk eknk
```

eine Implementierung von Σ durch Datentypen. Jede Instanz des folgenden Datentyps mit Attributen (siehe Abschnitt 1.2) repräsentiert eine Σ-Algebra:

```
data SigAlg s1...sk = SigAlg {c11 :: e11' -> s1, ..., c1n1 :: e1n1' -> s1,
                              ...
                              ck1 :: ek1' -> sk, ..., cknk :: eknk' -> sk}
```

e'_{ij} entsteht aus e_{ij}, indem dort für alle $1 \leq i \leq k$ der Typ S_i durch die Typvariable s_i ersetzt wird. Von ihrer Semantik her sind Attribute Namen für die einzelnen Komponentenmengen eines kartesischen Produktes (in

kaskadierter Form). Dementsprechend gibt es immer zwei Darstellungen eines Elements des Datentyps: eine mit Konstruktor und eine mit Attributen. So sind z.B.

```
termalg :: SigAlg S1...Sk
termalg = SigAlg {c11 = C11, ..., c1n1 = C1n1,
                  ...
                  ck1 = Ck1, ..., cknk = Cknk}
```

und

```
termalg :: SigAlg S1...Sk
termalg = SigAlg C11 ... C1n1 ... Ck1 ... Cknk
```

zwei äquivalente Implementierungen der Σ-Termalgebra (s.u.). Klassische programmiersprachliche Konstrukte für Typen mit Attributen sind `Recordtypen` und objektorientierte Klassen ohne Methoden. In Abschnitt 4.2 werden wir noch einmal Datentypen mit Attributen verwenden, und zwar um Algebren mit *funktionalen* Trägermengen zu implementieren.

Beispiel 3.2.8 *JavaSig*-**Algebren**. Der Datentyp, dessen Instanzen *JavaSig*-Algebren repräsentieren, lautet wie folgt:

```
data JavaAlg block command intE boolE =
             JavaAlg {block_ :: [command] -> block,
                      skip :: command,
                      assign :: String -> intE -> command,
                      cond :: boolE -> block -> block -> command,
                      loop :: boolE -> block -> command,
                      intE_ :: Int -> intE,
                      var :: String -> intE,
                      sub :: intE -> intE -> intE,
                      sum_, prod :: [intE] -> intE,
                      boolE_ :: Bool -> boolE,
                      greater :: intE -> intE -> boolE,
                      not_ :: boolE -> boolE}
```

Die Implementierung von *IntEs* durch `[IntE]` schliesst die durch die Konstruktoren *singleE* und *appendE* von *IntEs* (s.o) ausgeschlossene leere Liste wieder ein. Wir vermeiden damit die Einführung eines Datentyps für ausschließlich nichtleere Listen. Bei der Konstruktion von *JavaGra*-Parsern werden wir aber sicherstellen, dass nur nichtleere Listen erzeugt werden.

Eine Σ-Algebra kennen wir schon: die Menge T_Σ der Σ-Terme. Sie lässt sich sehr einfach zur Σ-Algebra, der **Σ-Termalgebra**, erweitern:

- Für alle $c : \varepsilon \to s \in \Sigma$ ist $c^{T_\Sigma} =_{def} c$.
- Für alle $c : w \to s \in \Sigma$ mit $e \neq \varepsilon$ und $t \in T_{\Sigma,w}$ ist $c^{T_\Sigma}(t) =_{def} c(t)$.

Beispiel 3.2.9 *JavaSig*-**Termalgebra**. Als Instanz von `JavaAlg` sieht $T_{JavaSig}$ folgendermaßen aus:

```
termAlg :: JavaAlg Block Command IntE BoolE
termAlg = JavaAlg id Skip Assign Cond Loop IntE Var Sub Sum Prod BoolE Greater Not
```

T_Σ ist **initiale** Σ-Algebra, d.h. zu jeder Σ-Algebra A gibt es einen eindeutigen **Σ-Homomorphismus**

$$eval^A : T_\Sigma \to A,$$

also eine S-sortierte Funktion, die mit den Interpretationen der Konstruktoren von Σ in T_Σ bzw. A vertauschbar ist, d.h. für alle $c : w \to s \in \Sigma$ gilt:

$$eval_s^A \circ c^{T_\Sigma} \;=\; c^A \circ eval_w^A.$$

Aufgabe Zeige, dass alle Σ-Homomorphismen von T_Σ nach A mit $eval^A$ übereinstimmen!

Definition 3.2.10 $eval^A$ ist induktiv über T_Σ definiert und wird die **Σ-Auswertungsfunktion in A** genannt:

- Für alle $c : s \in \Sigma$ ist $eval_s^A(c) = c^A$.
- Für alle $c : e \to s \in \Sigma$ und $t \in T_{\Sigma,w}$ ist $eval_s^A(c(t)) = c^A(eval_w^A(t))$. ❑

Mit den obigen Bezeichnungen lautet das Schema einer auf beliebige Algebren A anwendbaren Implementierung von $eval^A$ wie folgt: Sei $1 \leq i \leq k$.

```
eval_si :: SigAlg s1...sk -> Si -> si
eval_si alg (Ci1 ei1)   = ci1 (eval_ei1 alg e_i1)
...
eval_si alg (Cini eini) = c1n1 (eval_eini alg eini)
```

Hierbei ist `eval_s` für alle $s \notin \{s_1, \ldots, s_k\}$ die jeweilige Identitätsfunktion.

Beispiel 3.2.11 Auswertungsfunktion für $Expr$-Terme

```
eval :: ExprAlg a -> Expr -> a
eval alg (Con i)   = con alg i
eval alg (Var x)   = var alg x
eval alg (Sum es)  = sum_ alg (map (eval alg) es)
eval alg (Prod es) = prod alg (map (eval alg) es)
eval alg (e :- e') = sub alg (eval alg e) (eval alg e')
eval alg (n :* e)  = scal alg n (eval alg e)
eval alg (e :^ n)  = expo alg (eval alg e) n
```

Beispiel 3.2.12 Auswertungsfunktion für $JavaSig$-Terme

```
evBlock :: JavaAlg block command intE boolE -> Block -> block
evBlock alg = block_ alg . map (evCommand alg)

evCommand :: JavaAlg block command intE boolE -> Command -> command
evCommand alg Skip             = skip alg
evCommand alg (Assign x e)     = assign alg x (evIntE alg e)
evCommand alg (Cond be cs cs') = cond alg (evBoolE alg be)
                                          (evBlock alg cs)
                                          (evBlock alg cs')
evCommand alg (Loop be cs)     = loop alg (evBoolE alg be)
                                          (evBlock alg cs)
```

```
evIntE :: JavaAlg block command intE boolE -> IntE -> intE
evIntE alg (IntE i)   = intE_ alg i
evIntE alg (Var x)    = var alg x
evIntE alg (Sub e e') = sub alg (evIntE alg e) (evIntE alg e')
evIntE alg (Sum es)   = sum_ alg (map (evIntE alg) es)
evIntE alg (Prod es)  = prod alg (map (evIntE alg) es)

evBoolE :: JavaAlg block command intE boolE -> BoolE -> boolE
evBoolE alg (BoolE b)      = boolE_ alg b
evBoolE alg (Greater e e') = greater alg (evIntE alg e) (evIntE alg e')
evBoolE alg (Not be)       = not_ alg (evBoolE alg be)
```

Fazit:

Jeder Interpreter oder Compiler einer ECFG G ist eindeutig durch eine $\Sigma(G)$-Algebra bestimmt.

Die Idee, die Initialität der $\Sigma(G)$-Termalgebra zu nutzen, um die Definition eines Interpreters oder Compilers auf die Erweiterung der Zielsprache zur $\Sigma(G)$-Algebra zu reduzieren, wurde erstmalig in [19], §3.1 ausgeführt. Später wurde eine Reihe von Softwaretools geschaffen, die vorwiegend der Entwicklung von Compilern und explizit auf abstrakter Syntax aufbauen, z.B. ASF+SDF [10], Stratego/XT [79] und das *Rewriting Logic Semantics Project* [44]. In diesen Systemen wird die oben beschriebene Auswertung von Syntaxbäumen in einer Zielsprachen-Algebra im wesentlichen durch Programmtransformationen realisiert, die in der Anwendung von Termersetzungsregeln bestehen.

Nicht nur Compiler sind durch Σ-Algebren bestimmt: Jede rekursive Funktion auf T_Σ lässt sich als Auswertungsfunktion in eine passende Σ-Algebra darstellen! Darauf wird in dieser LV aber nicht näher eingegangen.

Die Elemente von T_Σ können auch als Objekte der Instanz `Tree String` des Datentyps

```
Tree a = F a [Tree a]
```

für Bäume mit Knotenmarkierungen vom Typ `a` repräsentiert werden. Diese Darstellung bildet ebenfalls eine Σ-Algebra:

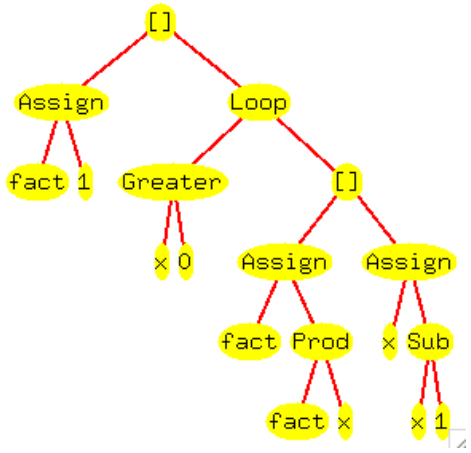


Figure 3.3. *Graphische Darstellung eines Elementes von* `treeAlg` *(mit Painter.hs; siehe §1.3)*

Beispiel 3.2.13 Darstellung von *JavaSig*-Termen durch `Tree String`-Objekte

```
treeAlg :: JavaAlg (Tree String) (Tree String) (Tree String) (Tree String)
```

```
treeAlg = JavaAlg (F "[]") (F "Skip" []) (\x e -> F "Assign" [F x [],e])
                  (\be b b' -> F "Cond" [be,b,b']) (\be b -> F "Loop" [be,b])
                  (leaf . show) leaf (\e e' -> F "Sub" [e,e'])
                  (F "Sum") (F "Prod") (leaf . show)
                  (\e e' -> F "Greater" [e,e']) (\be -> F "Not" [be])
```

Beispiel 3.2.14 Simplifikation von *Expr*-Termen Die Auswertung eines *Expr*-Terms e in der folgenden Algebra *simplExpr* bewirkt die Anwendung folgender Vereinfachungsregeln auf e:

$$
\begin{array}{lll}
0+e=e & 0*e=0 & 1*e=e \\
e+e=2*e & e*e=e^2 & \\
(m*e)+(n*e)=(m+n)*e & e^m*e^n=e^{m+n} & \\
m*(n*e)=(m*n)*e & (e^m)^n=e^{m*n} &
\end{array}
$$

```
simplExpr :: ExprAlg Expr
simplExpr = ExprAlg Con Var reduceSum reduceProd reduceSub
            reduceScal reduceExpo

reduceSum :: [Expr] -> Expr
reduceSum es = mkSum (map mkScal dom)
  where mkScal (Con 0) = Con (scal zero)
        mkScal e       = reduceScal (scal e) e
        (scal,dom) = foldl trans initState (mkSummands (Sum es))
        initState = (const 0,[zero])
        trans (scal,dom) (n:*e)  = (upd scal e (scal e+n),insert e dom)
        trans (scal,dom) (Con n) = (upd scal zero (scal zero+n),dom)
        trans sd e               = trans sd (1:*e)
        mkSum []  = zero
        mkSum [e] = e
        mkSum es  = case mkSummands (Sum es) of [] -> zero; [e] -> e
                                                es -> Sum es
        mkSummands (Sum es) = concatMap mkSummands (remove zero es)
        mkSummands e        = remove zero [e]

reduceProd :: [Expr] -> Expr
reduceProd es = mkProd (map mkExpo dom)
  where mkExpo (Con 1) = Con (expo one)
        mkExpo e       = reduceExpo e (expo e)
        (expo,dom) = foldl trans initState (mkFactors (Prod es))
        initState = (upd (const 0) one 1,[one])
        trans (expo,dom) (e:^n)  = (upd expo e (expo e+n),insert e dom)
        trans (expo,dom) (Con n) = (upd expo one (expo one*n),dom)
        trans ed e               = trans ed (e:^1)
        mkProd []  = one
        mkProd [e] = e
        mkProd es  = case mkFactors (Prod es) of [] -> one; [e] -> e
                                                 s -> Prod es
        mkFactors (Prod es) = concatMap mkFactors (remove one es)
        mkFactors e      = remove one [e]
```

```
reduceSub :: Expr -> Expr -> Expr
reduceSub e e' = reduceSum [e,(-1):*e']

reduceScal :: Int -> Expr -> Expr
reduceScal 0 e        = zero
reduceScal 1 e        = e
reduceScal m (Con i)  = Con (m*i)
reduceScal m (Sum es) = Sum (map (m :*) es)
reduceScal m (n :* e) = (m*n) :* e
reduceScal m e        = m :* e

reduceExpo :: Expr -> Int -> Expr
reduceExpo e 0         = one
reduceExpo e 1         = e
reduceExpo (Con i) n   = Con (i^n)
reduceExpo (Prod es) n = Prod (map (:^ n) es)
reduceExpo (e :^ m) n  = e :^ (m*n)
reduceExpo e n         = e :^ n

upd :: Eq a => (a -> b) -> a -> b -> a -> b
upd f a b x = if x == a then b else f x

remove :: Eq a => a -> [a] -> [a]
remove x (y:s) = if x == y then remove x s else y:remove x s
remove _ _     = []

insert :: Eq a => a -> [a] -> [a]
insert x (y:s) = if x == y then s else y:insert x s
insert x _     = [x]
```

Beispiel 3.2.15 Zustandsalgebra. Wie sieht die *JavaSig*-Algebra A aus, die man braucht, um den Interpreter aus Beispiel 1.2.3 als Instanz von `evBlock` (siehe Beispiel **??**) zu bekommen? Als Instanz von `JavaAlg block command intE boolE` ist A folgendermaßen definiert: Sei `State = String -> Int`.

```
stateAlg :: JavaAlg (State -> State) (State -> State) (State -> Int) (State -> Bool)
stateAlg = JavaAlg (foldl (flip (.)) id) id (\x e st -> update st x (e st))
                   (\be b b' st -> if be st then b st else b' st) loop
                   const (\x st -> st x) (\e e' st -> e st - e' st)
                   (\es st -> sum (map ($ st) es))
                   (\es st -> product (map ($ st) es))
                   const (\e e' st -> e st > e' st) (not .)
           where loop be b st = if be st then loop be b (b st) else st
```

Definition 3.2.16 Die Elemente von T_Σ lassen sich auch als **hierarchische String-Listen** repräsentieren. Diese Darstellung bildet ebenfalls eine Σ-Algebra. Im Fall von *JavaSig* sieht sie folgendermaßen aus:

```
listAlg :: JavaAlg (Int -> Bool -> String) (Int -> Bool -> String)
                   (Int -> Bool -> String) (Int -> Bool -> String)
```

```
listAlg = JavaAlg {block_ = \cs n -> let f []     = "[]"
                                         f [c]    = '[':c (n+1) True++"]"
                                         f (c:cs) = mkList c cs "[" "]" (n+1)
                                     in maybeBlanks (f cs) n,
                   skip = maybeBlanks "Skip",
                   assign = \x e n -> let str = "Assign "++show x++' ':e (n+10+length x) True
                                      in maybeBlanks str n,
                   cond = \be b b' n -> let str = "Cond "++g True be++g False b++g False b
                                               g b f = f (n+5) b
                                       in maybeBlanks str n,
                   loop = \be b n -> let str = "Loop "++g True be++g False b
                                                g b f = f (n+5) b
                                     in maybeBlanks str n,
                   intE_ = \i -> maybeBlanks ("(IntE "++show i++")"),
                   var = \x -> maybeBlanks ("(Var "++show x++")"),
                   sub = \e e' n -> let str = "(Sub "++ g True e++g False e'++")"
                                                g b e = e (n+5) b
                                    in maybeBlanks str n,
                   sum_ = \(e:es) n -> let str = mkList e es "(Sum[" "])" (n+5)
                                       in maybeBlanks str n,
                   prod = \(e:es) n -> let str = mkList e es "(Prod[" "])" (n+6)
                                       in maybeBlanks str n,
                   boolE_ = \b -> maybeBlanks ("(BoolE "++show b++")"),
                   greater = \e e' n -> let str = "(Greater "++ g True e++g False e'++")"
                                                g b e = e (n+9) b
                                        in maybeBlanks str n,
                   not_ = \be n -> maybeBlanks ("(Not "++be (n+5) True++")") n}
```

Die show-Funktionen von Beispiel 1.2.4 bilden die Auswertungsfunktion in `listAlg`. Dort sind auch die Hilfsfunktionen `maybeBlanks` und `mkList` definiert. Ein Element von `listAlg`:

```
[Assign "fact" (IntE 1),
 Loop (Greater (Var "x")
               (IntE 0))
      [Assign "fact" (Prod[(Var "fact"),
                           (Var "x")]),
       Assign "x" (Sub (Var "x")
                       (IntE 1))]]
```

In Fig. 3.2 sind neben der Termalgebra zwei weitere Mengen genannt, die sich als Algebren einer abstrakten Syntax darstellen lassen:

Definition 3.2.17 Sei $G = (N, T, P, S)$ eine ECFG. Die von G erzeugte Sprache $L(G)$ ist eine N-sortierte Menge, die sich durch folgende Interpretationen der Konstruktoren von $\Sigma(G)$ zur $\Sigma(G)$-Algebra, der **Quellcodealgebra von** G, erweitern lässt: Für alle $(A \longrightarrow e_1|\ldots|e_k) \in P$ und $1 \leq i \leq k$ ist

$$c_{A,i}^{L(G)} : L(G)_{abs(e_i)} \rightarrow L(G)_A$$

wie folgt definiert: Seien $A_1, \ldots, A_n$ die in e_i vorkommenden Nichtterminale und

$$w_1 \in L(G)_{A_1}, \ldots, w_n \in L(G)_{A_n}.$$

$$c_{A,i}^{L(G)}(abs(e_i)[w_1/A_1,\ldots,w_n/A_n]) \;=_{def}\; e_i[w_1/A_1,\ldots,w_n/A_n].$$

$[w_1/A_1,\ldots,w_n/A_n]$ bezeichnet die Substitution von A_i durch w_i ("w_i für A_i") für alle $1 \leq i \leq n$. Die substituierten Ausdrücke sind tatsächlich Elemente des Definitions- bzw. Wertebereichs von $c_{A,i}^{L(G)}$. ❑

Beispiel 3.2.18 Implementierung der Quellcodealgebra von *JavaGra*.

```
sourceAlg :: JavaAlg String String String String
sourceAlg = JavaAlg {block_ = \cs -> '{':concat cs++"}",
                     skip = "; ",
                     assign = \x e -> x++" = "++e++"; ",
                     cond = \be b b' -> "if ("++be++") "++b++
                                        (if b' == "{;}" then "" else " else "++b'),
                     loop = \be b -> "while ("++be++") "++b,
                     intE_ = show,
                     var = id,
                     sub = \e e' -> '(':e++'-':e'++")",
                     sum_ = ('(':) . f '+',
                     prod = ('(':) . f '*',
                     boolE_ = show,
                     greater = \e e' -> e++" > "++e',
                     not_ = \be -> "(!"++be++")"}
            where f _ [e]      = e++")"
                  f chr (e:es) = e++chr:f chr es
```

Definition 3.2.19 Sei $G = (N, T, P, S)$ eine ECFG. Die Menge $Abl(G)$ der Ableitungsbäume von G (siehe Def. 3.1.5) ist eine N-sortierte Menge ($B \in Abl(G)_A$ gdw B hat Wurzel A), die sich durch folgende Interpretationen der Konstruktoren von $\Sigma(G)$ zur $\Sigma(G)$-Algebra, der **Ableitungsbaumalgebra von** G, erweitern lässt: Für alle $p = (A \longrightarrow e_1|\ldots|e_k) \in P$ und $1 \leq i \leq k$ ist

$$c_{A,i}^{Abl(G)} : Abl(G)_{abs(e_i)} \;\rightarrow\; Abl(G)_A$$

wie folgt definiert: Seien $A_1,\ldots,A_n$ die in e_i vorkommenden Nichtterminale und

$$B_1 \in Abl(G)_{A_1},\ldots,B_n \in Abl(G)_{A_n}.$$

$$c_{A,i}^{Abl(G)}(abs(e_i)[B_1/A_1,\ldots,B_n/A_n]) \;=_{def}\; e_i[B_1/A_1,\ldots,B_n/A_n].$$

$[B_1/A_1,\ldots,B_n/A_n]$ bezeichnet die Substitution von A_i durch B_i ("B_i für A_i") für alle $1 \leq i \leq n$. Die substituierten Ausdrücke sind tatsächlich Elemente des Definitions- bzw. Wertebereichs von $c_{A,i}^{Abl(G)}$.

Beispiel 3.2.20 Implementierung der Ableitungsbaumalgebra von *JavaGra*.

```
deriAlg :: JavaAlg (Tree String) (Tree String) (Tree String) (Tree String)
deriAlg = JavaAlg {block_ = \cs -> F "Block" (leaf "{":cs++[leaf"}"]),
                   skip = F "Command" [leaf ";"],
                   assign = \x e -> F "Command" [leaf (x++" = "),e,leaf"; "],
                   cond = \be b b' -> F "Command" (leaf "if (":be:leaf ") ":b:
                                                  (if b' == F "{;}" [] then []
                                                   else [leaf " else ",b'])),
                   loop = \be b -> F "Command" [leaf "while (",be,leaf ") ",b],
                   intE_ = leaf . show, var = leaf,
```

```
            sub = \e e' -> F "IntE" [e,leaf "-",e'],
            sum_ = F "IntE" . f "+",
            prod = F "IntE" . f "*",
            boolE_ = leaf . show,
            greater = \e e' -> F "BoolE" [e,leaf ">",e'],
            not_ = \be -> F "BoolE" [leaf "!",be]}
where f _ [e]      = [e]
      f chr (e:es) = e:leaf chr:f chr es
```

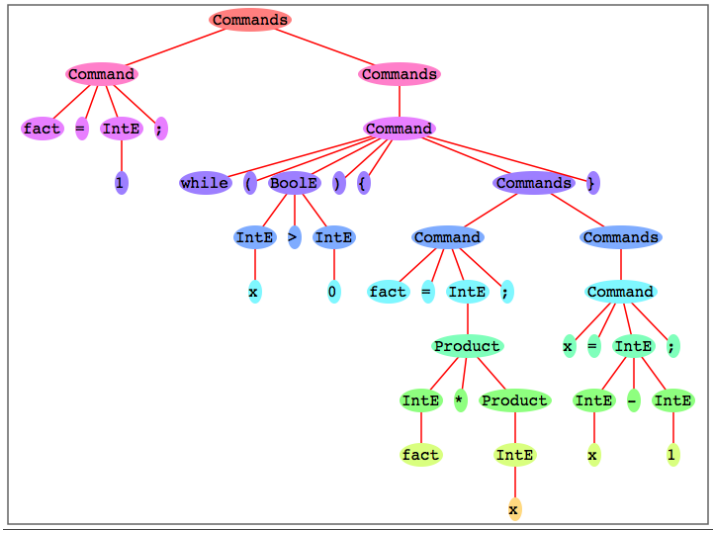


Figure 9.2. *Graphische Darstellung eines Elementes von* `deriAlg` *(mit Painter.hs; siehe §1.3)*

Da Auswertungsfunktionen zwar generisch für eine beliebige Algebra, aber dennoch für jeden einzelnen Konstruktor des zugehörigen Datentyps definiert werden müssen, spart diese Implementierung Schreibarbeit nur dann, wenn mehrere Σ-Algebren definiert und benutzt werden. Da verschiedene Konstruktoren verschiedenen Typen haben können, ist es nicht möglich, die Auswertung von Termen durch eine einzige Gleichung mit einer Funktionsvariablen, die durch jeden Konstruktor instanziiert werden könnte, zu definieren. Die oben beschriebene Implementierung ist geeignet für Anwendungen, in der einige fest vorgegebene Signaturen vorkommen, nicht für "Meta-Anwendungen" wie z.B. einen Parsergenerator, der eine Grammatik als Eingabe bekommt und daraus deren abstrakte Syntax konstruieren soll. Hierzu braucht man einen Datentyp, dessen Elemente der Darstellung *beliebiger* Terme dienen.

Dafür geeignet ist der bereits in Abschnitt 1.2 vorgestellte Typ `Tree String` von Bäumen mit beliebigem Knotenausgrad und Einträgen vom Typ `String`, die Konstruktoren repräsentieren. Die Auswertung eines Terms in einer Algebra besteht dann in seiner Faltung mit der – ähnlich der Listenfaltung – definierten Funktion `foldT`:

```
type Algebra a op = op -> [a] -> a
foldT :: Algebra a op -> Term op -> a
foldT alg (F op ts) = alg op (map (foldT alg) ts)
```

Algebren mit Trägermenge(n) `a` werden also als Funktionen `alg :: op -> [a] -> a` dargestellt: Jeder Operation `op` der Signatur ordnet `alg` eine Funktion `alg op` von `[a]` nach `a` zu, die in der Regel partiell ist, weil Algebren nur Funktionen fester Stelligkeit haben, `alg op` also nur für Listen einer festen Länge definiert sein wird, und weil alle Trägermengen der Algebra in `a` zusammengefasst werden, `alg op` aber nicht auf allen definiert ist.

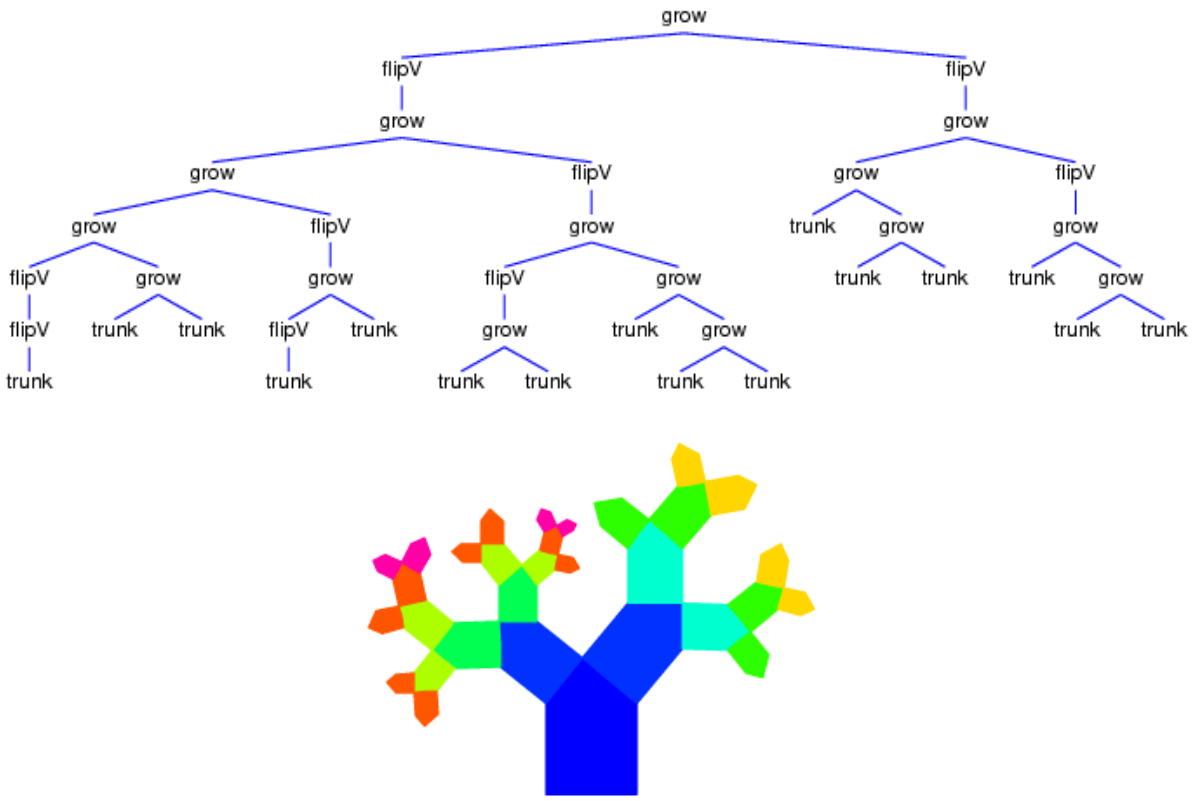


Figure 3.5. *Noch ein Syntaxbaum und das Ergebnis seiner Auswertung (die übrigens der Wuchsrichtung des Ergebnisbaumes entgegenläuft)*

Der Vorteil der algebraischen Sicht auf Compiler liegt nicht nur in der generischen Auswertung (= Compilation) von Syntaxbäumen. Im nächsten Abschnitt werden wir sehen, wie auf diese Weise auch Parser generisch gemacht werden können und damit die übliche Trennung von syntaktischer und semantischer Analyse aufgehoben werden kann, was natürlich zu einem erheblichen Effizienzgewinn führen kann. Auch der Nachweis der Korrektheit eines Compilers lässt sich auf das wirklich Notwendige reduzieren: Geht man von der Darstellung der Quellsprache als abstrakte Syntax $\Sigma(G)$ aus (siehe Def. 3.2.3), dann ist Q im Diagramm von Def. 1.1.1 durch die Termalgebra T_Σ gegeben. Formuliert man die Zielsprache Z und die beiden Semantiken M_Q und M_Z als Σ-Algebren, dann sind *comp* und I_Q in jenem Diagramm die Σ-Auswertungsfunktionen in Z bzw. M_Q. Zur Korrektheit von *comp* bleibt dann nur noch zu zeigen, dass auch *encode* und I_Z Σ-Homomorphismen sind. Dann sind nämlich auch die Kompositionen $I_Z \circ comp$ und $encode \circ I_Q$ homomorph und, da *beide* Homomorphismen von T_Σ nach M_Z führen, identisch, d.h. das Diagramm von Def. 1.1.1 ist kommutativ! Im Detail ausgeführt findet man einen solchen Korrektheitsbeweis für eine iterative Quellsprache in [74].

Schließlich subsumiert die algebraische Sicht Begriffe wie **Baumautomat**, **reguläre Baumsprache** und was noch alles an formalen Begriffen im Zusammengang mit Auszeichnungssprachen (XML u.ä.) – in zahlreichen Varianten – wiederbelebt oder neu eingeführt wurde. Ein (erkennender) Baumautomat scannt einen Σ-Term t und erreicht genau dann einen Endzustand, wenn t zu einer vorgebenen Teilmenge L von T_Σ gehört. Analog zu Wortsprachen nennt man L eine reguläre Baumsprache, wenn es einen Baumautomaten gibt, der L erkennt. (Baumautomaten entsprechen Σ-Algebren: L ist genau dann regulär, wenn es eine Teilmenge Fin einer Σ-Algebra A gibt, deren Urbild unter $eval^A$ mit L übereinstimmt [75, 68]:

$$(eval^A)^{-1}(Fin). \tag{3.2}$$

Die Auswertungsfunktion in A entspricht demnach der Übergangsfunktion des Baumautomaten. Seine Zustands-

menge entspricht der Trägermenge von A, Endzustände sind die Elemente von Fin. Schon reguläre Wortsprachen über einem Alphabet B lassen sich so charakterisieren: Mit

$$\Sigma_B =_{def} (\{s\}, \{\varepsilon : s\} \cup \{b : s \to s \mid b \in B\})$$

ist $B^* = T_{\Sigma_B}$ und damit ein Baumautomat ein erkennender Automat im Sinne von Def. 2.1.2. Also ist $L \subseteq B^*$ genau dann regulär im Sinne von Def. 2.1.1, wenn es eine Teilmenge Fin einer Σ_B-Algebra A gibt mit (3.2).

Die Korrespondenz zwischen Baumautomaten und Algebren geht über erkennende Automaten hinaus. Analog zu erkennenden Wortautomaten stellen auch erkennende Baumautomaten den Spezialfall der Ausgabemenge $\{0, 1\}$ dar (siehe §2.1). Ausgehend von einer beliebigen, aber festen Ausgabemenge Y beschreibt ein Baumautomat A über $\Sigma = (N, C)$ mit einer (N-sortierten) Ausgabefunktion $\beta : Q \to Y$ eine *Realisierung* einer vorgegebenen **Verhaltensfunktion** $f : T_\Sigma \to Y$: A **realisiert** f, wenn $f = \beta \circ eval^A$ gilt. So wie es für jede Verhaltensfunktion $f : X^* \to Y$ eines Wortautomaten eine minimale Realisierung gibt (siehe §2.3), so gibt es für jede Verhaltensfunktion $f : T_\Sigma \to Y$ einen Baumautomaten mit minimaler Zustandsmenge (= Algebra mit minimaler Trägermenge), der f realisiert. Wir fassen zusammen:

> Eine Algebra A mit Trägermenge Q entspricht der Übergangsfunktion δ eines Baumautomaten B mit Zustandsmenge Q.
> Die Ausgabefunktion $\beta : Q \to Y$ von B macht A zur Realisierung des Verhaltens $\beta \circ eval^A$.

Baumautomaten arbeiten also stets auf *abstrakter* Syntax und können folglich keine Parser sein (wie die Wortautomaten in §2.1), die per definitionem *konkrete* Syntax verarbeiten. Ein erkennender Baumautomat kann allerdings einen Parser insofern ergänzen, als er nur einige der vom Parser erzeugten Syntaxbäume akzeptiert und auf diesem Umweg die Eingabesprache weiter einschränkt.

3.3 Parser und Parserkombinatoren

Ein Parser für eine Grammatik G ist eine Umkehrfunktion der Auswertungsfunktion in die Quellcodealgebra von G (siehe Def. 3.2.17):

Definition 3.3.1 Sei $G = (N, T, P, S)$ eine ECFG. Ein Algorithmus, der zu jedem $w \in L(G)$ einen Syntaxbaum t mit $eval^{L(G)}(t) = w$ erzeugt und zu jedem $w \in T^* - L(G)$ eine Fehlermeldung liefert, heißt **Parser für** G. ❑

G ist genau dann eindeutig (siehe §3.1), wenn $eval^{L(G)}$ injektiv ist!

Man unterscheidet **top-down**- oder **shift-derive**-Parser von **bottom-up** oder **shift-reduce** Parsern. Erstere erzeugen Syntaxbäume von oben nach unten, letztere von unten nach oben. Die im letzten Abschnitt behandelte algebraische Sicht auf Syntaxbäume lässt nur bottom-up-Parser zu, da Aufbau und Auswertung von Elementen einer Termalgebra wie die der Syntaxbäume stets bottom-up vorgehen.

Eine weitere Klassifizierung von Parsern orientiert sich an der Klassennzugehörigkeit der verarbeitbaren Grammatiken. Z.B. erlauben *deterministisch* kontextfreie Sprachen deterministische Parser, die dadurch charakterisiert sind, dass sie jedes Eingabesymbol zwar mehrfach lesen, aber nur einmal verarbeiten können. Deterministische Parser werden dann weiter klassifiziert, jetzt nach Unterklassen deterministisch kontextfreier Sprachen. So sind die in den Abschnitten 3.4 und 3.5 behandelten LL- bzw. LR-Parser, die gerade sog. LL- bzw. LR-Sprachen erkennen können (die selbst in einer Unterklassenbeziehung stehen: LL ist echt enthalten in LR).

Da die Syntaxanalyse scheitert, wenn die eingelesene Symbolfolge nicht zur Sprache der zugrundeliegenden Grammatik gehört, und der Parser in diesem Fall eine Fehlermeldung liefern soll, benötigen wir zu seiner

Implementierung einen Datentyp, der sowohl Elemente eines Typs `a` (z.B. Syntaxbäume) zusammen mit der Resteingabe als auch Fehlermeldungen im Falle des Scheiterns umfasst:

```
type Parser a = [Symbol] -> Maybe (a,[Symbol])
```

Sei $G = (N, T, P, S)$ eine ECFG und alg eine $\Sigma(G)$-Algebra. Die Eingabe eines Parsers vom Typ `Parser a` ist eine (vom Scanner erzeugte) Folge von Symbolen (Elementen von T). Seine Ausgabe ist,

- falls die Eingabe ein Präfix in $L(G)$ hat: ein Element von alg, das das (interpretierte) Ergebnis der Syntaxanalyse darstellt, sowie die noch nicht parsierte Resteingabe (Wählen wir $alg = T_{\Sigma(G)}$, dann liefert der Parser Syntaxbäume!),
- sonst `Nothing`.

`Nothing` ist als Fehlermeldung natürlich etwas mager. Soll genauere Information über die Ursache des Scheiterns eines Parsevorganges geliefert werden, dann muss Maybe (a,[Symbol]) durch einen Datentyp der Form

```
data Result a = Correct (a,[Symbol]) | Error String
```

ersetzt werden, mit dem sich differenziertere Fehlermeldungen erzeugen lassen. Allerdings kann, da Parser wie Scanner mit Hilfe von **Kombinatoren** aus anderen Parsern zusammengesetzt werden, das Durchreichen von Ausnahmewerten durch die Aufrufstruktur der Teilparser eine nichttriviale Programmieraufgabe sein!

Kombinatoren für Scanner ergaben sich aus den Operatoren zur Bildung regulärer Ausdrücke. Der Aufbau des Parsers für G folgt den Regeln von $CF(G)$:

Sei $\Sigma(G) = (N_0, N_1, C)$, $A \in N_1$ und alg_A der Datenbereich von alg für A. Zur Erkennung und Interpretation der Sprache $L(G)_A$ (in $CF(G)$ aus A ableitbare terminale Wörter; s.o.) benötigen wir einen Parser pa_A des Typs `Parser` alg_A. Je nach der Menge der Nichtterminale auf der rechten Seite der Regel $A \to w \in P$ ruft pa_A die entsprechenden Parser für diese Nichtterminale auf.

Die Existenz von Parsern für die - Standardtypen repräsentierenden - Nichtterminale von N_0 wird vorausgesetzt. In Haskell sind sie meistens als Instanzen von *read* vordefiniert.

Wir betrachten exemplarisch drei Formen der rechten Seite von $A \to w$:

(1) $w = xByCz$ with $B, C \in N$ und $x, y, z \in T$,
(2) $w = B|CD|CE$ with $B, C, D, E \in N$ and $C \neq A$,
(3) $w = B|AD|AE$ mit $B, D, E \in N$,

Die Typvariable a in den folgenden Typen wird durch den Datenbereich für A der - als Parameter `alg` - an die Parser übergebenen Algebra instanziiert.

Parser für A im Fall 1 `x,y,z` sind Konstanten der Sorte `Symbol`, `f alg :: b -> c -> a` ist die Interpretation des aus $A \to w$ gebildeten Konstruktors in `alg`.

```
parseA :: SigAlg ... a ... -> Parser a
parseA alg (x:syms) = case parseB alg syms of
                          Just (b,y:syms)
                            -> case parseC alg syms of
                                    Just (c,z:syms) -> Just (f alg b c,syms)
                                    _ -> Nothing
                          _ -> Nothing
parseA _ _          = Nothing
```

Parser für A im Fall 2

`f alg :: b -> a`, `g alg :: c -> d -> a` und `h alg :: c -> e -> a` sind die Interpretationen der aus $s \to w$ gebildeten Konstruktoren in `alg`.

```
parseA :: SigAlg ... a ... -> Parser a
parseA alg syms = case parseB alg syms of
                        Just (b,syms) -> Just(f alg b,syms)
                        _ -> case parseC alg syms of
                                  Just (c,syms) -> parseArest alg c syms
                                  _ -> Nothing

parseArest :: SigAlg ... a ... c ... -> c -> Parser a
parseArest alg c syms = case parseD alg syms of
                              Just (d,syms) -> Just (g alg c d,syms)
                              _ -> case parseE alg syms of
                                        Just (e,syms) -> Just(h alg c e,syms)
                                        _ -> Nothing
```

Parser für A im Fall 3

`f alg :: b -> a`, `g alg :: a -> d -> a` and `h alg :: a -> e -> a` sind die Interpretationen der aus $s \to w$ gebildeten Konstruktoren in `alg`.

```
parseA :: SigAlg ... a ... -> Parser a
parseA alg syms = case parseB alg syms of
                        Just (b,syms) -> parse_s_rest alg (f alg b) syms
                        _ -> case parse_s alg syms of
                                  Just (a,syms) -> parseArest alg a syms
                                  _ -> Nothing

parseArest :: SigAlg ... a ... -> a -> Parser a
parseArest alg a syms = case parseD alg syms of
                              Just (d,syms) -> Just(g alg a d,syms)
                              _ -> case parseE alg syms of
                                        Just (e,syms) -> Just (h alg a e,syms)
                                        _ -> Just (a,syms)
```

Was passiert, wenn `parseA` auch bei $C = A$ wie im Fall 2 definiert wird? `parseA` würde die Erkennung von B stets der von BD oder BE vorziehen. Von mehreren Wörtern der Sprache $L(G)_A$ soll aber immer ein längstes erkannt werden. Deshalb darf die Erkennung eines Wortes von $L(G)_A$ nur dann mit der Erkennung eines Wortes von $L(G)_B$ enden, wenn das Weiterlesen der Eingabe zu keinem längeren längeren Wort von $L(G)_A$ führen würde. Genau das erreicht `parseArest`: Die letzte Alternative (`Just (a,syms)`) wird genau dann zurückgegeben, wenn `syms` das längste Eingabepräfix in $L(G)_A$ ist.

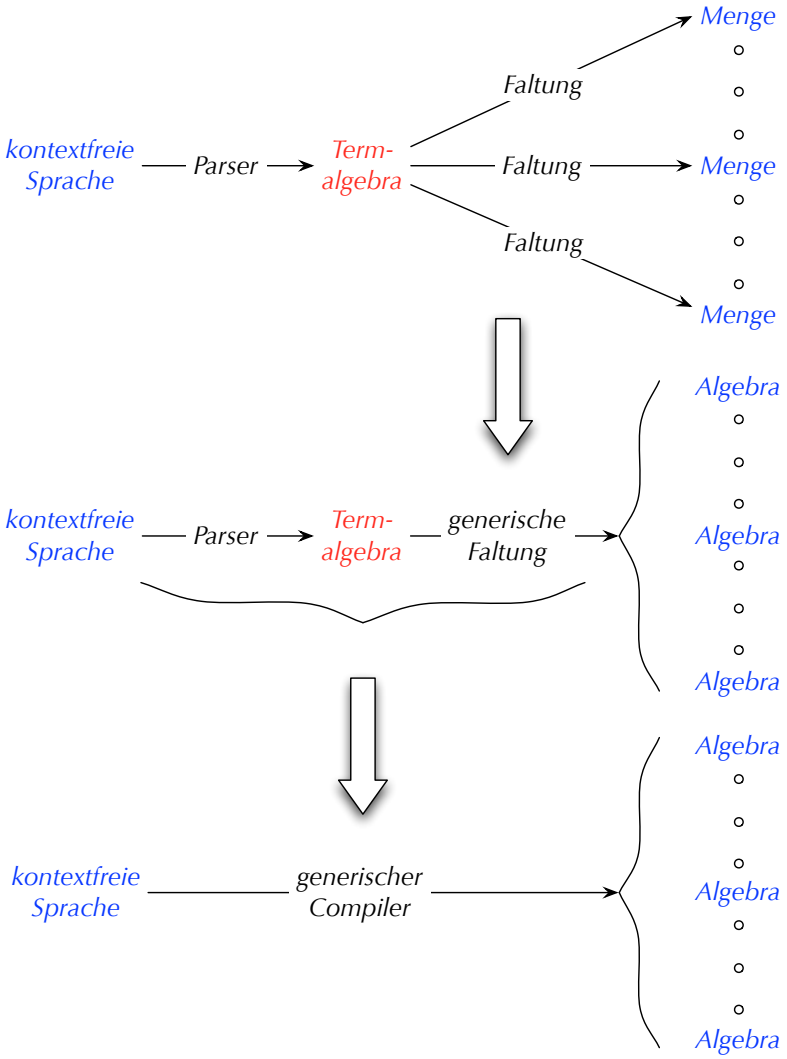


Figure 3.6. *Der Weg zum Compiler als interpretierendem Parser*

Beispiel 3.3.2 *JavaGra*-**Parser**. Der Typ `Symbol` entspricht hier dem Ausgabetyp des *JavaGra*-Scanners (siehe Beispiel 2.2.1). Wir definieren Parser für die vier Nichtterminale von *JavaGra* in die jeweiligen Datenbereiche einer beliebigen *JavaSig*-Algebra.

In *JavaSig* tauchen Listentypen der Form $[s]$ mit $s \in N_1$ auf. Für deren Parser verwenden wir folgenden Parserkombinator:

```
list :: Parser a -> Parser [a]
list p = iter p []
         where iter :: Parser a -> [a] -> Parser [a]
               iter p as syms = case p syms of
                                  Just (a,syms) -> iter p (as++[a]) syms
                                  _ -> Just (as,syms)
```

Unter der Voraussetzung, dass `Symbol` der vom *JavaGra*-Scanner verwendete Typ ist, lauten Parser für die vier Nichtterminale von *JavaGra* wie folgt:

```
paBlock :: JavaAlg block a b c -> Parser block
```

```
paBlock alg (Key "{":syms) = case list (paCommand alg) syms of
                                 Just (cs,Key "}":syms)
                                   -> Just (block_ alg cs,syms)
                                 _ -> Nothing
paBlock _ _                = Nothing

paCommand :: JavaAlg a command b c -> Parser command
paCommand alg (Key ";":syms)       = Just (skip alg,syms)
paCommand alg (Ide x:Key "=":syms) = case paIntE alg syms of
                                       Just (e,Key ";":syms)
                                         -> Just (assign alg x e,syms)
                                       _ -> Nothing
paCommand alg (Ide x:_)            = Nothing
paCommand alg (Key "if":Key "(":syms)
       = case paBoolE alg syms of
              Just (be,Key ")":syms)
                -> case paBlock alg syms of
                        Just (b,Key "else":syms)
                          -> case paBlock alg syms of
                                  Just (b',syms)
                                    -> Just (cond alg be b b',syms)
                                  _ -> Nothing
                        Just (b,syms)
                          -> Just (cond alg be b (block_ alg []), syms)
                        _ -> Nothing
              _ -> Nothing
paCommand alg (Key "if":_) = Nothing
paCommand alg (Key "while":Key "(":syms)
              = case paBoolE alg syms of
                     Just (be,Key ")":syms)
                       -> case paBlock alg syms of
                              Just (b,syms) -> Just (loop alg be b,syms)
                              _ -> Nothing
                     _ -> Nothing
paCommand _ _ = Nothing

paIntE :: JavaAlg a b intE c -> Parser intE
paIntE alg (Num i:syms)   = paIntErest alg (intE_ alg i) syms
paIntE alg (Ide x:syms)   = paIntErest alg (var alg x) syms
paIntE alg (Key "(":syms) = case paIntE alg syms of
                              Just (e,Key ")":syms) -> paIntErest alg e syms
                              _ -> Nothing
paIntE alg syms           = Nothing

paIntErest :: JavaAlg a b intE c -> intE -> Parser intE
paIntErest alg e (syms'@(Key "-":syms))
                    = case paIntE alg syms of
                           Just (e',syms) -> Just (sub alg e e',syms)
```

```
                              _ -> Just (e,syms')
paIntErest alg e syms = case list (appendE (Key "+")) syms of
                              Just (es,syms) -> Just (sum_ alg (e:es),syms)
                              _ -> case list (appendE (Key "*")) syms of
                                        Just (es,syms) -> Just (prod alg (e:es),syms)
                                        _ -> Nothing

appendE :: JavaAlg a b intE c -> Symbol -> Parser intE
appendE alg sym (sym':syms) | sym == sym' = paIntE alg syms
appendE _ _ _                             = Nothing

paBoolE :: JavaAlg a b c boolE -> Parser boolE
paBoolE alg (Key "true":syms)  = Just (boolE_ alg True,syms)
paBoolE alg (Key "false":syms) = Just (boolE_ alg False,syms)
paBoolE alg (Key "!":syms)     = case paBoolE alg syms of
                                       Just (be,syms) -> Just (not_ alg be,syms)
                                       _ -> Nothing
paBoolE alg syms               = case paIntE alg syms of
                                       Just (e,Key ">":syms)
                                         -> case paIntE alg syms of
                                                 Just (e',syms)
                                                   -> Just (greater alg e e',syms)
                                                 _ -> Nothing
                                       _ -> Nothing
```

Scanner lassen sich oft in Parser integrieren. Der Symboltyp wäre dann `Char`, Der resultierende Parser läuft im Fall von *JavaGra* auf die Zerlegung einiger Gleichungen, die den Parser von Beispiel 3.3.2 definieren, in Gleichungen, die an denjenigen des Scanners von Beispiel 2.2.1 orientiert sind.

In §3.6 werden wir die Implementierung interpretierender Parser auf der Basis von Monaden vereinfachen und mit differenzierten Fehlermeldungen versehen.

Iterative Parser für CFGs

Die in den beiden folgenden Abschnitten behandelten **LL-** bzw- **LR-Parser** sollen als iterative Funktionen formuliert werden. Deshalb benötigen sie als weiteres Argument eine Datenstruktur, die Zwischenergebnisse akkumuliert. Bei Parsern ist diese Datenstruktur im einfachsten Fall eine endliche (Zustands-)Menge und in allgemeineren Fällen eine Liste von Zuständen, die als Keller verwaltet wird.

Definition 3.3.3 Sei $G = (N, T, P, S)$ eine CFG und Q eine Zustandsmenge. Eine berechenbare Funktion

$$parse : Q^* \times T^* \longrightarrow \{true, false\}$$

heißt **iterative parse-Funktion** für G, falls für alle $w \in T^*$ gilt:

$$parse(q_0, w) = true \iff w \in L(G),$$

wobei $q_0 \in Q$ ein ausgezeichneter Anfangszustand ist. ❑

Ist G regulär, dann ist Q die Zustandsmenge eines (deterministischen) Automaten, der die von G erzeugte

Sprache erkennt. Die zugehörige parse-Funktion lautet wie folgt:

$$\begin{array}{lcll} parse(q, xw) & = & parse(\delta_A(q, x), w) & \text{“shift”} \\ parse(q, \varepsilon) & = & \begin{cases} true & \text{falls } q \in E_A \\ false & \text{sonst} \end{cases} & \begin{array}{r}\text{“accept”} \\ \text{“error”}\end{array} \end{array}$$

Hier genügt ein einzelner Zustand, um die jeweils nächste Aktion des Parsers zu bestimmen. Ist G nicht regulär, dann müssen i.a. mehrere Zustände *gekellert* werden. Deshalb enthalten die Argumente von $parse$ neben der jeweiligen Resteingabe ($\in T^*$) eine Zustands*folge* ($\in Q^*$), den **Zustandskeller**.

Definition 3.3.4 (first- und follow-Wortmengen) Sei $k \in \mathbb{N}$. Für terminale Wörter w bezeichnet $first_k(w)$ das k-elementige Präfix von w. Für beliebige Wörter α werden die k-elementigen Präfixe der aus α ableitbaren Wörter hinzugenommen. Sei $\alpha \in (N \cup T)^*$ und $A \in N$.

$$\begin{array}{rcl} first_k(\alpha) & = & \{w \in T^k \mid \exists v \in T^* : \alpha \xrightarrow{*} Gwv\} \ \cup \ \{w \in T^{<k} \mid \alpha \xrightarrow{*} Gw\} \\ follow_k(A) & = & \{w \in T^k \mid \exists u, v \in T^* : S \xrightarrow{*} GuAwv\} \\ & & \cup \{w \in T^{<k} \mid \exists u \in T^* : S \xrightarrow{*} GuAw\} \\ first(\alpha) & = & first_1(\alpha) \\ follow(A) & = & follow_1(A) \end{array}$$

Eine induktive Definition von $first(\alpha)$, $\alpha \in (N \cup T)^*$, lautet folgendermaßen:

$$\begin{array}{rcll} & & \varepsilon \in first(\varepsilon), & (1) \\ x \in T & \Rightarrow & x \in first(x\alpha), & (2) \\ (A \to \alpha) \in P \wedge x \in first(\alpha\beta) & \Rightarrow & x \in first(A\beta). & (3) \end{array}$$

Diese drei Implikationen bilden eine Definition von $first$, weil sie sich nur endlich oft so anwenden lassen, dass es in jedem Anwendungsschritt mindestens ein α gibt derart, dass $first(\alpha)$ nach der Anwendung mehr Elemente enthält als davor. Warum ist das so? Weil jedes $first(\alpha)$ eine Menge von Grammatiksymbolen (einschließlich ε) ist und es insgesamt nur endlich viele Grammatiksymbole gibt. Ein Haskell-Programm, das diese Definition wiedergibt, lautet wie folgt: *rules* sei die Menge aller Grammatikregeln, eine Regel $A \longrightarrow \alpha$ wird als Paar (A, α) dargestellt. Ein Wort ist als String mit Leerzeichen zwischen den Symbolen des Wortes impementiert. Das leere Wort ε ist die leere Liste [] bzw. – als Element von $first(\alpha)$ – das leere Wort “”.

```
first :: String -> [String]
first ""    = [""]                                                          (1)
first alpha = f (words alpha)
              where f (a:alpha) = g a `join` (if notNullable a then [] else f alpha)
                                  where (g,notNullable) = firstPlus

firstPlus :: (String -> [String],String -> Bool)
    berechnet first für einzelne Symbole und eine Boolesche Funktion notNullable, die einen
    String genau dann nach True abbildet, wenn aus ihm das leere Wort nicht ableitbar ist.
firstPlus = loop1 init (const True)
            where init = fold2 upd (const []) terminals (map single terminals)  (2)
                  single x = [x]

loop1 :: (String -> [String]) -> (String -> Bool)
         -> (String -> [String],String -> Bool)
    Schleife (bricht ab, wenn sich weder f noch notNullable verändern)
```

```
loop1 f notNullable = if b then loop1 f' notNullable' else (f,notNullable)
                      where (b,f',notNullable') = loop2 rules False f notNullable

loop2 :: [(String,String)] -> Bool -> (String -> [String]) -> (String -> Bool)
         -> (Bool,String -> [String],String -> Bool)
    Berechnung der i-ten Approximation von f und notNullable aus der (i-1)-ten.
    Die Boolesche Variable b ist genau dann True, wenn dabei f oder notNullable verändert werden.
loop2 ((a,rhs):rules) b f notNullable =
             case search notNullable rhs of
                  Just i -> loop2 rules (b || f a /= xs) (upd f a xs) notNullable
                            where xs = joinMap f (a:take (i+1) (words rhs))          (3)
                  _ -> loop2 rules (b || notNullable a) f (upd notNullable a False)
loop2 _ b f notNullable = (b,f,notNullable)

fold2 :: (a -> b -> c -> a) -> a -> [b] -> [c] -> a
fold2 f a (x:xs) (y:ys) = fold2 f (f a x y) xs ys
fold2 _ a _ _           = a

upd :: Eq a => (a -> b) -> a -> b -> a -> b
upd f x y z = if x == z then y else f z

search :: (a -> Bool) -> [a] -> Maybe Int
search f s = g s 0 where g (x:s) i = if f x then Just i else g s (i+1)
                         g _ _     = Nothing
```

3.4 LL-Parser

lesen die Eingabe von **l**inks nach rechts und erzeugen **deterministisch** und **top-down** eine **Linksableitung**.

Definition 3.4.1 Eine kontextfreie Grammatik G heißt **LL(k)-Grammatik**, wenn das Vorauslesen von k noch nicht verarbeiteten Eingabesymbolen genügt, um zu entscheiden, welche direkte Linksableitung als nächste durchzuführen ist, um schließlich das gesamte Eingabewort abzuleiten. Formal: Sind

$$S \xrightarrow{*} uA\alpha \longrightarrow u\beta\alpha \xrightarrow{*} uv$$
$$S \xrightarrow{*} uA\alpha \longrightarrow u\beta'\alpha \xrightarrow{*} uw$$

zwei Linksableitungen mit $u, v, w \in T^*, A \in N, \alpha, \beta, \beta' \in (N \cup T)^*$ und $first_k(v) = first_k(w)$, dann gilt $\beta = \beta'$. ❑

Beispiel 3.4.2 Sei G eine reduzierte[6] kontextfreie Grammatik ohne ε-Produktionen, aber mit einer linksrekursiven Produktion: $A \to A\beta$ derart, dass für alle anderen Produktionen der Form $A \to \beta' \quad |\beta'| \geq k$ gilt.

G ist nicht LL(k).

Beweis. Da G reduziert ist, muß es eine Linksableitung

$$S \xrightarrow{*} uA\alpha \longrightarrow u\beta'\alpha \underset{(1)}{\xrightarrow{*}} uv$$

[6] Jedes Nichtterminal kommt in mindestens einer Ableitung eines terminalen Wortes aus dem Startsymbol vor.

geben. Dann gilt auch

$$S \xrightarrow{*} uA\alpha \longrightarrow uA\beta\alpha \longrightarrow u\beta'\beta\alpha \xrightarrow[(2)]{*} uw$$

mit $first_k(v) = first_k(w)$ weil $|\beta'| \geq k$ und in (1) und (2) keine ε-Produktionen angewendet werden. ❑

Linksrekursionen einer Grammatik G ohne ε-Produktionen, d.h. Ableitungen der Form

$$A \xrightarrow{+} A\alpha,$$

können durch folgenden Algorithmus zur Transformation von G eliminiert werden: Sei $\{A_1, \ldots, A_n\}$ die Menge der Nichtterminale von G. Ersetze für alle $1 \leq i \leq n$

- jede Regel $A_i \to A_j\alpha$ mit $i > j$, für die eine Regel $A_j \to \beta$ existiert, durch die Regel $A_i \to \beta\alpha$ und
- jedes Regelpaar $(A_i \to A_i\alpha, A_i \to \beta)$ derart, dass A_i kein Präfix von β ist, durch die drei Regeln

$$A_i \to \beta A_i', \quad A_i' \to \alpha A_i', \quad A_i' \to \varepsilon,$$

wobei A_i' ein neues Nichtterminal ist.

Wegen der Berechenbarkeit von $first$ und $follow$ (siehe §3.3) lässt sich die LL(k)-Eigenschaft mit dem folgenden **LL(k)-Kriterium** entscheiden:

Eine reduzierte Grammatik G ist genau dann eine LL(k)-Grammatik, wenn für je zwei verschiedene Produktionen $A \to \beta, A \to \beta' \in P$ gilt:

$$first_k(\beta follow_k(A)) \cap first_k(\beta' follow_k(A)) = \emptyset.$$

Strenggenommen gilt diese Äquivalenz nur für **starke LL(k)-Grammatiken**, die dadurch definiert sind, dass die Bedingung in Def. 3.4.1 auch dann gilt, wenn die Kontexte u und α in den beiden Linksableitungen verschieden sind. Bei $k = 1$ macht das keinen Unterschied. Aber schon bei $k = 2$ gibt es ein trennendes Beispiel: Die Grammatik mit den Produktionen

$$\begin{array}{lcl} S & \longrightarrow & aAaa \mid bAba \\ A & \longrightarrow & b \mid \varepsilon \end{array}$$

ist LL(2), aber nicht stark LL(2) (siehe auch [21], Def. 3.9).

Als iterative parse-Funktion

$$parse : Q^* \times T^* \longrightarrow \{true, false\}$$

im Sinne von Def. 3.3.3 lautet der LL-Parser für eine LL(1)-Grammatik G ohne Linksrekursionen wie folgt: Die Zustandsmenge Q ist hier die Menge $N \cup T$ der Symbole von G und der Anfangszustand q_0 ist das Startsymbol S von G. Sei $x \in T$ und $A \in N$.

$$\begin{array}{lcll} parse_{LL}(x\alpha, xw) & = & parse_{LL}(\alpha, w) & \text{“shift”} \\ parse_{LL}(A\alpha, w) & = & parse_{LL}(\beta\alpha, w) & \text{“derive”} \\ & & \text{falls } A \to \beta \in P \text{ und } first(w) \in first(\beta follow(A)) & (*) \\ parse_{LL}(\varepsilon, \varepsilon) & = & true & \text{“accept”} \\ parse_{LL}(\alpha, w) & = & false \quad \text{sonst} & \text{“error”} \end{array}$$

Durch Induktion über die Definition von $parse_{LL}$ erhält man sofort:

$$parse_{LL}(w, \alpha) = true \iff \alpha \xrightarrow[G]{*} w,$$

also insbesondere:

$$parse_{LL}(w, S) = true \iff w \in L(G).$$

$parse_{LL}$ ist wohldefiniert: Jeder Leseschritt (*shift*) verkürzt das Eingabewort und jede Sequenz aufeinanderfolgender Ableitungsschritte (*derive*) ist endlich, weil N endlich ist und G keine Linksrekursionen erzeugt. Außerdem ist jeder Ableitungsschritt eindeutig, weil G eine LL(1)-Grammatik ist. Die Berechenbarkeit von $parse_{LL}$ folgt aus der Entscheidbarkeit von (*). Bei der Implementierung von $parse_{LL}$ werden zunächst für alle $(A \to \beta) \in P$ die Mengen $first(\beta follow(A))$ berechnet und in der **Aktionstabelle** act_{LL} festgehalten:

$$act_{LL} : N \times (T \cup \{\varepsilon\}) \longrightarrow P \cup \{error\}.$$

$$act_{LL}(A, x) = \begin{cases} A \longrightarrow \beta & \text{falls } A \to \beta \in P \text{ und } x \in first(\beta follow(A)) \\ error & \text{sonst} \end{cases}$$

Die Bedingungen in der Definition von $parse_{LL}$ werden zu Tabellen-Lookups reduziert:

$parse_{LL}(x\alpha, xw)$	=	$parse_{LL}(\alpha, w)$		"shift"
$parse_{LL}(y\alpha, xw)$	=	$false$	falls $x \neq y$	"error"
$parse_{LL}(\varepsilon, xw)$	=	$false$		"error"
$parse_{LL}(x\alpha.\varepsilon)$	=	$false$		"error"
$parse_{LL}(\varepsilon, \varepsilon)$	=	$true$		"accept"
$parse_{LL}(A\alpha, w)$	=	$parse_{LL}(\beta\alpha, w)$	falls $act_{LL}(A, first(w) = (A \to \beta)$	"derive"
$parse_{LL}(A\alpha, w)$	=	$false$	falls $act_{LL}(A, first(w)) = error$	"error"

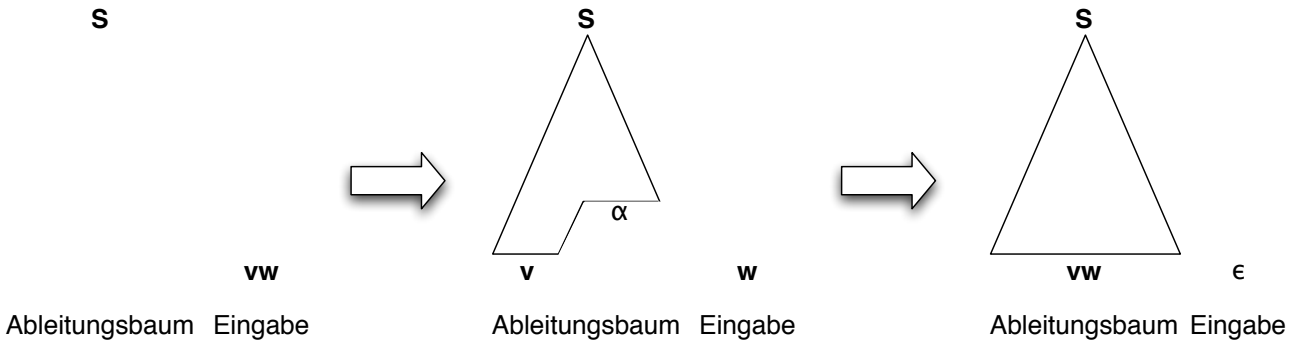


Figure 3.7. *Prinzip des LL-Parsers*

Beispiel 3.4.3 Zwei Grammatiken, ihre abstrakte Syntax und ihre Aktionstabellen

```
-- Konkrete Syntax (1. Grammatik)

    p1 = ("S","a S b")          S ⟶ aSb
    p2 = ("S","")               S ⟶ ε

-- Abstrakte Syntax

    data S = P1 S | P2

-- Aktionstabelle

    actLL ("S","a") = Just p1
```

```
    actLL ("S","b") = Just p2
    actLL ("S","")  = Just p2
    actLL _         = Nothing

-- Konkrete Syntax (2. Grammatik)

    p1 = ("S","a A S")              S ⟶ aAS
    p2 = ("S","b")                  S ⟶ b
    p3 = ("A","a")                  A ⟶ a
    p4 = ("A","b S A")              A ⟶ bSA

-- Abstrakte Syntax

    data S = P1 A S | P2
    data A = P3 | P4 S A

-- Aktionstabelle

    actLL ("S","a") = Just p1
    actLL ("S","b") = Just p2
    actLL ("A","a") = Just p3
    actLL ("A","b") = Just p4
    actLL _         = Error
```

Vom LL-Parser zum *rekursiven Abstieg*

So wie oben definiert ist $parse_{LL}$ iterativ, wird aber echt rekursiv, wenn parallel zur Worterkennung ein Syntaxbaum oder ein Wert in einer beliebigen $\Sigma(G)$-Algebra alg berechnet werden soll, wie es die Definition eines Parsers (3.3.1) erfordert. Wir erweitern deshalb $parse_{LL}$ zu

$$parse_{LL}^{alg} \ : \ (N \cup T)^* \times T^* \ \longrightarrow \ (alg^* \times T^*) \cup Fehlermeldungen.$$

Für alle $w \in T^*$ soll gelten:[7]

$$parse_{LL}^{alg}(S, w) = ([eval^A(t)], \varepsilon)$$
wobei t ein Syntaxbaum für die Ableitung $S \xrightarrow{*}_{CF(G)} w$ ist.

Dies ist ein Spezialfall der allgemeinen Anforderung an $parse_{LL}^{alg}$, die folgendermaßen lautet: Sei $w, w_0, w_1, \ldots, w_n \in T^*$ und $A_1, \ldots, A_n \in N$.

$$parse_{LL}^{alg}(w_0 A_1 w_1 A_2 w_2 \ldots A_n w_n, w_0 v_1 w_1 v_2 w_2 \ldots A_n v_n w) = ([(eval^A(t_1), \ldots, eval^A(t_n))], w)$$
wobei für alle $1 \leq i \leq n$ t_i ein Syntaxbaum für die Ableitung $S \xrightarrow{*}_{CF(G)} v_i$ ist.

Die Definition von $parse_{LL}^{alg}$ folgt derjenigen von $parse_{LL}$. Sei $x \in T$, $A \in N$ und f_p der zur Produktion p gehörige Konstruktor (siehe 3.2.3).

$$\begin{aligned}
parse_{LL}^{alg}(y\alpha, xw) &= \text{if } x = y \text{ then } parse_{LL}^{alg}(\alpha, w) \text{ else } \textit{“x ist aus y nicht ableitbar”} \\
parse_{LL}^{alg}(x\alpha, \varepsilon) &= \textit{“}\varepsilon \textit{ ist aus x nicht ableitbar”} \\
parse_{LL}^{alg}(A\alpha, w) &= (a : aL, v') \text{ where } (a, v) = \overline{A}(w)
\end{aligned}$$

[7] Die Elemente von alg^* notieren wir wie Haskell-Listen. $[a_1, \ldots, a_n]$ entspricht also dem Wort $a_1 \ldots a_n$. $a : aL$ ist die Liste mit Kopfelement a und Restliste aL (siehe §1.2).

$$(aL, v') = parse_{LL}^{alg}(\alpha, v)$$
$$parse_{LL}^{alg}(\varepsilon, w) \quad = ([], w)$$

Die Aktionstabelle kommt jetzt nur noch in Aufrufen der – wechselseitig rekursiv mit $parse_{LL}^{alg}$) definierten – Funktionen

$$\overline{A} \ : \ T^* \longrightarrow \ (alg^* \times T^*) \cup Fehlermeldungen,$$

$A \in N$, vor:

$$\begin{array}{lll} \overline{A}(w) = \text{case } act_{LL}(A, first(w)) \text{ of } (A \to \beta) & \to & (f_{A \longrightarrow \beta}^{alg}(aL), v) \text{ where } (aL, v) = parse_{LL}^{alg}(\beta, w) \\ error & \to & \textit{“first(w) ist aus A nicht ableitbar”} \end{array}$$

Durch den rekursiven Abstieg wird der Wert in alg ist aus dem ursprünglichen top-down-LL-Parser ist ein bottom-up-Parser geworden. Man sieht das auch daran, dass $parse_{LL}^{alg}$ im Gegensatz zu $parse_{LL}$ nicht iterativ, sondern echt rekursiv ist.

Aufgabe Erweitern Sie $parse_{LL}^{alg}$ um einen **Unparser**, der parallel zur Syntaxanalyse das Eingabewort rekonstruiert und dabei die Fehlermeldungen an den Stellen in den Eingabetext einstreut, an denen sie jeweils aufgetreten sind!

3.5 LR-Parser

lesen die Eingabe von **l**inks nach rechts und erzeugen **deterministisch** und **bottom-up** eine umgekehrte **Rechtsableitung**, die man wegen der Umkehrung auch **Reduktion** nennt.

Definition 3.5.1 Eine kontextfreie Grammatik G, in der das Startsymbol nicht auf der rechten Seite einer Produktion auftritt, heißt **LR(k)-Grammatik**, wenn das Vorauslesen von k noch nicht verarbeiteten Eingabesymbolen genügt, um zu entscheiden, ob ein weiteres Zeichen verarbeitet oder eine Reduktion und, wenn ja, welche durchgeführt werden soll. Formal: Sind

$$S \xrightarrow{*} \gamma A v \longrightarrow u\gamma\alpha v$$
$$S \xrightarrow{*} \gamma' A' w' \longrightarrow u\gamma'\alpha' w' = \gamma\alpha w$$

zwei Rechtsableitungen mit $\gamma, \gamma', \alpha, \alpha' \in (N \cup T)^*, A, A' \in N, v, w, w' \in T^*$ und $first_k(v) = first_k(w)$, dann gilt

$$\gamma A w = \gamma' A' w'. \ \square$$

Beispiel 3.5.2 Die Grammatik $G = (\{S, A\}, \{*, b, c\}, P, S)$ mit den Produktionen

$$\begin{array}{lll} S & \to & A \\ A & \to & A * A \\ A & \to & b \\ A & \to & c \end{array}$$

ist für kein k eine LR(k)-Grammatik.

Beweis. Es gibt zwei Rechtsableitungen

$$\begin{array}{lll} S \xrightarrow{*} A(*b)^k & \xrightarrow{(1)} & \overbrace{A * A}^{\gamma\alpha}\overbrace{(*b)^k}^{v} \\ S \xrightarrow{*} A * A * A(*b)^{k-1} & \xrightarrow{(2)} & \underbrace{A * A * b}_{\gamma'\alpha'}\underbrace{(*b)^{k-1}}_{w'} \end{array}$$

mit $w = *b(*b)^{k-1}, \gamma = \varepsilon$ und $\gamma' = A * A*$, also $\gamma A \neq \gamma' A$. Hier liegt ein **shift-reduce-Konflikt** vor. Soll man zuerst weiterlesen (das Präfix $*b$ der Resteingabe) und dann die Reduktion (2) durchführen, oder soll gleich gemäß (1) reduziert werden? Das Vorauslesen von k Zeichen der Resteingabe nimmt uns diese Entscheidung nicht ab. ❑

Das zum o.g. LL(k)-Kriterium analoge **LR(k)-Kriterium** lautet wie folgt:

Für je zwei verschiedene direkte Rechtsableitungen

$$\begin{array}{rcl} \gamma A & \longrightarrow & \gamma\alpha\beta \\ \gamma' A' & \longrightarrow & \gamma'\alpha'\beta' \end{array}$$

mit $S \xrightarrow{*} \gamma Aw, S \xrightarrow{*} \gamma' A'w'$ und $\gamma\alpha = \gamma'\alpha'$ gilt:

$$first_k(\beta follow_k(A)) \cap first_k(\beta' follow_k(A')) = \emptyset.$$

Wurde ein Präfix einer korrekten Eingabe zum Wort φ reduziert, dann muss es eine Produktion $A \to \alpha\beta$ geben derart, dass α Suffix von φ ist und die ersten k Zeichen der Resteingabe zu $first_k(\beta follow_k(A))$ gehören. Erfüllt G das LR(k)-Kriterium, dann ist diese Produktion eindeutig durch φ bestimmt. Ohne dass wir im Moment wissen, welche Produktion das jeweils ist, können wir demnach schließen, dass die folgende Abbildungsvorschrift eine Funktion ist, sofern G das LR(k)-Kriterium erfüllt und, wie in Def. 3.5.1 vorausgesetzt wurde, das Startsymbol S nicht auf der rechten Seite einer Produktion auftritt.

Da hier nicht je zwei Produktionen, sondern zwei Rechtsableitungen verglichen werden, lässt sich das LR(k)-Kriterium schwerer entscheiden als das LL(k)-Kriterium. Es ist genau dann erfüllt, wenn man mit der unten ausgeführten Entwicklung eines LR-Parsers zu einer deterministischen parse-Funktion gelangt.

Wir beschränken uns auf $k = 1$ und entwickeln den LR(1)-Parser in drei Schritten.

1. Schritt Wir beginnen mit folgender parse-Funktion:

$$parse^1_{LR} : (N \cup T)^* \times T^* \longrightarrow \{true, false\}$$

$$\begin{array}{lcll} parse^1_{LR}(\varphi, xw) & = & parse^1_{LR}(\varphi x, w) & \text{falls } S \xrightarrow{*} \gamma Av, \quad \text{“shift”} \\ & & & (A \to \alpha\beta) \in P,\ \beta \neq \varepsilon, \\ & & & \varphi = \gamma\alpha,\ x \in first(\beta follow(A)) \\ parse^1_{LR}(\varphi, w) & = & parse^1_{LR}(\gamma A, w) & \text{falls } S \xrightarrow{*} \gamma Av,\ A \neq S, \quad \text{“reduce”} \\ & & & (A \to \alpha) \in P,\ \varphi = \gamma\alpha \\ & & & first(w) \in follow(A) \\ parse^1_{LR}(\varphi, \varepsilon) & = & true & \text{falls } (S \to \varphi) \in P \quad \text{“accept”} \\ parse^1_{LR}(\varphi, w) & = & false & \text{sonst} \quad \text{“error”} \end{array}$$

Durch Induktion über die Definition von $parse^1_{LR}$ erhält man sofort:

$$parse^1_{LR}(\varphi, w) = true \iff S \xrightarrow{*} \varphi w,$$

also insbesondere:

$$parse^1_{LR}(\varepsilon, w) = true \iff w \in L(G). \tag{3.3}$$

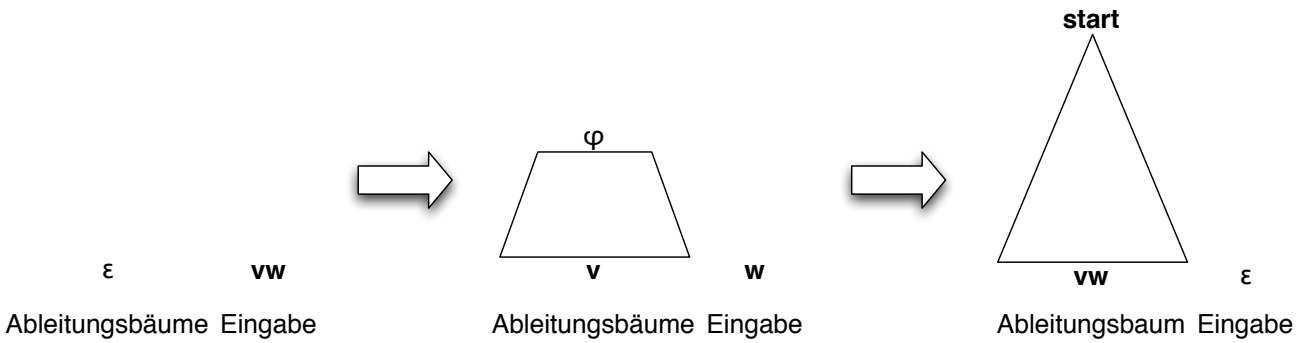


Figure 3.8. *Prinzip des LR-Parsers*

$parse^1_{LR}$ ist wohldefiniert: Jeder Leseschritt (*shift*) verkürzt das Eingabewort, jeder Reduktionsschritt (*reduce*) verringert den Abstand von φ zum Startsymbol S. Außerdem ist jeder Reduktionsschritt eindeutig, weil G eine LR(1)-Grammatik ist. Aber ist die "reduce"-Bedingung entscheidbar? Um das zu sehen, führen wir für jedes $\varphi \in (N \cup T)^*$ eine Relation

$$\mathbf{item}(\varphi) \subseteq N \times (N \cup T)^* \times (N \cup T)^* \times (T \cup \{\varepsilon\})$$

ein, die folgendermaßen definiert ist:

$$(A \to \alpha.\beta, first(v)) \in item(\gamma\alpha) \iff_{def} S \xrightarrow{*} \gamma A v \wedge (A \to \alpha\beta) \in P$$

Man schreibt $(A \to \alpha.\beta, x)$ anstelle von (A, α, β, x), weil das die Bedeutung von $item(\varphi)$ klarer macht. Können wir item(φ) berechnen, dann ist auch $parse^1_{LR}$ berechenbar:

$parse^1_{LR}(\varphi, xw)$	=	$parse^1_{LR}(\varphi x, w)$	falls $(A \to \alpha.\beta, y) \in item(\varphi),\ \beta \neq \varepsilon,\ x \in first(\beta y)$	"shift"
$parse^1_{LR}(\varphi, w)$	=	$parse^1_{LR}(\gamma A, w)$	falls $(A \to \alpha., first(w)) \in item(\varphi),\ \varphi = \gamma\alpha$	"reduce"
$parse^1_{LR}(\varphi, \varepsilon)$	=	$true$	falls $(S \to \alpha., \varepsilon) \in item(\varphi)$	"accept"
$parse^1_{LR}(\varphi, w)$	=	$false$	sonst	"error"

Das ist noch nicht sehr effizient, weil bei einem "reduce"-Schritt neben der Traversierung von $item(\varphi)$ γ ermittelt werden muss. γ lässt sich jedoch aus der Definition von $parse^1_{LR}$ vollständig eliminieren, wenn man diese Funktion in die folgende transformiert.

2. Schritt

$$parse^2_{LR} : ((N \cup T)^*)^* \times T^* \longrightarrow \{true, false\}$$

$parse^2_{LR}$ soll mit $parse^1_{LR}$ folgendermaßen zusammenhängen:[8]

$$\forall \varphi = x_1 \ldots x_n \in (N \cup T)^* : parse^1_{LR}(\varphi, w) = parse^2_{LR}([\varphi, (x_1 \ldots x_{n-1}), \ldots, (x_1 x_2), x_1, \varepsilon], w). \tag{3.4}$$

Dementsprechend wird die Korrektheitsbedingung (3.3) zu:

$$parse^2_{LR}([\varepsilon], w) = true \iff w \in L(G). \tag{3.5}$$

[8] Wir verwenden die Haskell-Notation für die äußere Listenstruktur von $((N \cup T)^*)^*$ (vgl. 1.2).

Wir wandeln die zuletzt angegebene Definition von $parse^1_{LR}$ in eine Definition von $parse^2_{LR}$ um so, dass (3.2) gilt: Sei $\varphi, \varphi_1, \ldots, \varphi_n \in (N \cup T)^*, a \in ((N \cup T)^*)^*$ und $|\alpha|$ die Länge von α.

$$
\begin{array}{lcll}
parse^2_{LR}(\varphi : a, xw) & = & parse^2_{LR}(\varphi x : \varphi : a, w) & \text{“shift”} \\
 & & \text{falls } (A \to \alpha.\beta, y) \in item(\varphi),\ \beta \neq \varepsilon,\ x \in first(\beta y) & \\
parse^2_{LR}(\varphi_1 : \cdots : \varphi_{|\alpha|} : \gamma : a, w) & = & parse^2_{LR}(\gamma A : \gamma : a, w) & \text{“reduce”} \\
 & & \text{falls } A \neq S,\ (A \to \alpha., first(w)) \in item(\varphi_1) & \\
parse^2_{LR}(\varphi : a, \varepsilon) & = & true \quad \text{falls } (S \to \alpha., \varepsilon) \in item(\varphi) & \text{“accept”} \\
parse^2_{LR}(a, w) & = & false \quad \text{sonst} & \text{“error”}
\end{array}
$$

Haben wir z.B. ein $(A \to \alpha., \varepsilon) \in item(\varphi)$ gefunden, dann ersetzen wir die letzten $|\alpha|$ Elemente des zweiten (Keller-)Argumentes von $parse^2_{LR}$ durch das Element φA. φ entspricht dem γ in der Definition von $parse^1_{LR}$, braucht aber jetzt nicht mehr berechnet werden.

3. Schritt $parse^2_{LR}$ arbeitet auf der Zustandsmenge des **initialen Automaten**

$$I = (Q, N \cup T, \{0,1\}, \delta, \beta, q_0)$$

mit

$$
\begin{array}{rcl}
Q & = & (N \cup T)^*, \\
\delta(\varphi, x) & = & \varphi x, \\
\beta(\varphi) = 1 & \Leftrightarrow & S \xrightarrow{*} \varphi, \\
q_0 & = & \varepsilon.
\end{array}
$$

I heißt initial, weil jeder Automat A, der L erkennt, Bild eines Homomorphismus $h : I \to A$ ist (vgl. Def. 2.3.5). Somit werden im Leseschritt von $parse^2_{LR}$ der Zustand φ im Keller durch seinen δ-Nachfolger φx und im Reduktionsschritt der Zustand γ durch dessen Nachfolger γA ersetzt.

Die endgültige LR(1)-parse-Funktion

$$parse_{LR} : Q \times T^* \longrightarrow \{true, false\}$$

erhalten wir aus $parse^2_{LR}$ durch Austausch des initialen Automaten mit seinem Bild unter $item$, dem **LR-Automaten**

$$LRA = (Q, N \cup T, \{0,1\}, \delta, \beta, q_0),$$

der folgendermaßen definiert ist:

$$
\begin{array}{rcl}
Q & = & \{item(\varphi) | \varphi \in (N \cup T)^*\}, \\
\delta(item(\varphi), z) & = & item(\varphi z), \\
\beta(item(\varphi)) = 1 & \Leftrightarrow & S \xrightarrow{*} \varphi, \\
q_0' & = & item(\varepsilon).
\end{array}
$$

Entsprechend ist der Zusammenhang zwischen den Ein/Ausgabe-Relationen von $parse^2_{LR}$ und $parse_{LR}$:

$$parse^2_{LR}([\varphi_1, \ldots, \varphi_n], w) = parse_{LR}([item(\varphi_1), \ldots, item(\varphi_n)], w). \tag{3.6}$$

Die Korrektheitsbedingung (3.3) wird zu:

$$parse_{LR}([q_0], w) = true \iff w \in L(G).$$

In der Definition von $parse^2_{LR}$ müssen lediglich die Komponenten von I durch die entsprechenden Komponenten von LRA ersetzt werden: Sei $q, q_1, \ldots, q_n \in Q$ und $a \in Q^*$.

$$\begin{array}{lcll}
parse_{LR}(q:a, xw) & = & parse_{LR}(\delta(q,x) : q : a, w) & \text{“shift”} \\
 & & \text{falls } (A \to \alpha.\beta, y) \in q,\ \beta \neq \varepsilon,\ x \in first(\beta y) & \\
parse_{LR}(q_1 : \cdots : q_{|\alpha|} : q : a, w) & = & parse_{LR}(\delta(q,A) : q : a, w) & \text{“reduce”} \\
 & & \text{falls } A \neq S,\ (A \to \alpha., first(w)) \in q_1 & \\
parse_{LR}(q:a, \varepsilon) & = & true \quad \text{falls } (S \to \alpha., \varepsilon) \in q^{9} & \text{“accept”} \\
parse_{LR}(a, w) & = & false \quad \text{sonst} & \text{“error”}
\end{array}$$

$parse_{LR}$ verwendet nur noch Namen für Zustände, von denen hier jeder einzelne eine Menge von Items ist. Zur Beantwortung von Anfragen der Form

$$\exists\ A, \alpha, \beta, x : (A \to \alpha.\beta, x) \in q\ ?$$

in Lese- bzw. Reduktionsschritten von $parse_{LR}$ müssen die Items, aus denen sich q zusammensetzt, allerdings bekannt sein. Aus der Definition einer Itemmenge:

$$(A \to \alpha.\beta, first(v)) \in item(\gamma\alpha) \iff S \xrightarrow{*} \gamma A v \wedge (A \to \alpha\beta) \in P,$$

ergeben sich die folgenden (verschränkt-)induktiven Definitionen von Q und δ:

$$\begin{array}{rcll}
(S \to \alpha) \in P & \Rightarrow & (S \to .\alpha, \varepsilon) \in q_0, & (1) \\
(A \to \alpha.B\beta, x) \in q \wedge (B \to \gamma) \in P \wedge y \in first(\beta x) & \Rightarrow & (B \to .\gamma, y) \in q, & (2) \\
(A \to \alpha.z\beta, x) \in q & \Rightarrow & (A \to \alpha z.\beta, x) \in \delta'(q, z). & (3)
\end{array}$$

Ähnlich der induktiven Definition von $first(\alpha)$ (siehe §3.3) lassen sich (1)-(3) als Schleife implementieren, die abbricht, wenn weder einzelne Zustände um neue Items erweitert werden noch die Zustandsmenge selbst vergrößert wird. *start* bezeichne das Startsymbol der Grammatik, *rules* die Menge der Regeln (dargestellt als Liste von Paaren der linken bzw. rechten Seite einer Regel). Außerdem führen wir zur Repräsentation der Zustände des LR-Automaten einen polymorphen Datentyp *Set* für Mengen ein. Das sind zwar auch nur Listen, als *Set*-Objekte können sie aber mit der Booleschen Funktion `==` so verglichen werden, als wären sie tatsächlich Mengen.

```
data Set a = Set {list::[a]}

instance Eq a => Eq (Set a)
                 where Set s == Set s' = all (`elem` s) s' && all (`elem` s') s

instance Show a => Show (Set a) where show (Set s) = show s

symbols = nonterminals++terminals

type LRState = Set (String,String,String,String)

q0 :: LRState                                          Anfangszustand (siehe (1))
q0 = [(a,[],alpha,"") | (a,alpha) <- rules, a == start]
```

[9] Da S auf keiner rechten Seite einer Produktion vorkommt, gilt $(S \to \alpha., \varepsilon) \in q = item(\varphi)$ genau dann, wenn $S \to \alpha \in P$ und $\alpha = \varphi$ gelten.

```
extend :: LRState -> LRState                              Erweiterung von q (siehe (2))
extend q = if q == q' then q else extend q'
           where qL = list q
                 q' = qL `join` [(a,"",beta,y) | (_,_,balpha,x) <- qL, (a,beta) <- rules,
                                                 not (null balpha), a == headw balpha,
                                                 y <- first (unwords (tailw balpha++[x]))]

trans :: LRState -> String -> Maybe (String,LRState)   erste Folgezustände von (q,x) (siehe (3))
trans q x = if null s then Nothing else Just (x,extend (Set s))
           where qL = list q
                 s = [(a,alpha++' ':x,unwords (tailw zbeta),y) | (a,alpha,zbeta,y) <- qL,
                                                                 not (null zbeta),
                                                                 headw zbeta == x]

headw = head . words
tailw = tail . words

type Transitions = [(LRState,String,LRState)]

mkLRauto :: ([Int],[(Int,String,Int)])  berechnet den LR-Automaten und codiert seine Zustände
                                        als ganze Zahlen
mkLRauto = (map encode qs,map f rel)
           where (qs,rel) = loop [extend q0] [] []
                 encode q = case search (== q) qs of Just i -> i
                                                     _ -> 0
                 f (q,x,q') = (encode q,x,encode q')

loop :: [LRState] -> [LRState] -> Transitions -> ([LRState],Transitions)
loop (q:qs) visited rel = loop (foldl add qs nonVisited) visited' (rel++map f allTrans)
                          where allTrans = map get (filter just (map (trans q) symbols))
                                visited' = add visited q
                                nonVisited = map snd allTrans `minus` visited'
                                f (x,q') = (q,x,q')
          q wird durch seine Folgezustände ersetzt und wandert selbst zu den besuchten Zuständen.
loop _ qs2 rel = (qs2,rel)

just :: Maybe a -> Bool
just (Just _) = True
just _ = False

get :: Maybe a -> a
get (Just x) = x

add :: Eq a => [Set a] -> [a] -> [Set a]      fügt eine Menge zu einer Menge von Mengen hinzu
add s@(x:s') y = if equal x y then s else x:add s' y
add _ x        = [x]
```

Nach der Konstruktion von Q (oben implementiert als Liste vom Typ `[LRState]`) und δ wird die **Aktionstabelle**

$$act_{LR} : (Q \times T \cup \{\varepsilon\}) \longrightarrow P \cup \{shift, error\}$$

angelegt, auf die $parse_{LR}$ zurückgreift, um die o.g. Anfragen zu beantworten. Sei $x \in T$.

$$act_{LR}(q,x) = \begin{cases} shift & \text{falls } (A \to \alpha.\beta, y) \in q,\ \beta \neq \varepsilon,\ x \in first(\beta y) \\ A \to \alpha & \text{falls } A \neq S,\ (A \to \alpha., x) \in q \\ error & \text{sonst} \end{cases}$$

$$act_{LR}(q,\varepsilon) = \begin{cases} S \to \alpha & \text{falls } (S \to \alpha., \varepsilon) \in q \\ error & \text{sonst} \end{cases}$$

Hier ist auch der Platz, um das zu Beginn des Abschnitts formulierte LR-Kriterium zu überprüfen. Es ist nämlich genau dann erfüllt, wenn act_{LR} wohldefiniert ist, also für jedes Argument (q, x) q

- weder $(A \to \alpha.\beta, y)$ mit $\beta \neq \varepsilon$ und $x \in first(\beta y)$ und gleichzeitig $(A' \to \alpha'., x)$ mit $A' \neq S$ (*shift-reduce-Konflikt*)
- noch zwei verschiedene Items $(A \to \alpha., x)$ und $(A' \to \alpha'., x)$ mit $A, A' \neq S$ (r*educe-reduce-Konflikt*)

enthält. Unter Verwendung von δ und act_{LR} erhält man schließlich eine kompakte Definition von $parse_{LR}$:

$$\begin{array}{lcll} parse_{LR}(q : a, xw) & = & parse_{LR}(\delta(q,x) : q : a, w) & \text{falls } act_{LR}(q,x) = shift \\ parse_{LR}(q_1 : \cdots : q_{|\alpha|} : q : a, w) & = & parse_{LR}(\delta(q,A) : q : a, w) & \text{falls } act_{LR}(q_1, first(w)) = A \to \alpha \\ parse_{LR}(q : a, \varepsilon) & = & true & \text{falls } act_{LR}(q,\varepsilon) = S \to \alpha \\ parse_{LR}(q : a, w) & = & false & \text{falls } act_{LR}(q, first(w)) = error \end{array}$$

Beispiel 3.5.3 Grammatik $(\{S, A, B\}, \{c, d, *\}, P, S)$ mit den Produktionen

$$S \to A \quad A \to A * B \quad A \to B \quad B \to c \quad B \to d$$

Implementierung in Haskell:

```
start = "S"
nonterminals = words "S A B"
terminals = words "c d *"
rules = [("S","A"),("A","A * B"),("A","B"),("B","c"),("B","d")]
```

Zustandsmenge Q und Übergangsfunktion $\delta : Q \times (N \cup T) \longrightarrow Q$:

$$\begin{array}{lcl} q_0 & = & \{(S \to .A, \varepsilon), (A \to .A * B, \varepsilon), (A \to .B, \varepsilon), (A \to .A * B, *), (A \to .B, *), (B \to .c, \varepsilon), \\ & & (B \to .d, \varepsilon), (B \to .c, *), (B \to .d, *)\} \\ q_1 & = & \delta(q_0, A) = \{(S \to A., \varepsilon), (A \to A. * B, \varepsilon), (A \to A. * B, *)\} \\ q_2 & = & \delta(q_0, B) = \{(A \to B., \varepsilon), (A \to B., *)\} \\ q_3 & = & \delta(q_0, c) = \{(B \to c., \varepsilon), (B \to c., *)\} = \delta(q_5, c) \\ q_4 & = & \delta(q_0, d) = \{(B \to d., \varepsilon), (B \to d., *)\} = \delta(q_5, d) \\ q_5 & = & \delta(q_1, *) = \{(A \to A * .B, \varepsilon), (A \to A * .B, *), (B \to .c, \varepsilon), (B \to .d, \varepsilon), (B \to .c, *), (B \to .d, *)\} \\ q_6 & = & \delta(q_5, B) = \{(A \to A * B., \varepsilon), (A \to A * B., *)\} \end{array}$$

Implementierung in Haskell: `fst (loop [extend q0] [] [])` liefert Q' für die o.g. Grammatik:

```
[[("S","","A",""), ("A","","A * B",""), ("A","","B",""),
  ("A","","A * B","*"), ("A","","B","*"), ("B","","c",""), ("B","","d",""),
  ("B","","c","*"), ("B","","d","*")],
 [("S","A","",""), ("A","A","* B",""), ("A","A","* B","*")],
 [("A","B","",""), ("A","B","","*")],
 [("B","c","",""), ("B","c","","*")],
 [("B","d","",""), ("B","d","","*")],
 [("A","A *","B",""), ("A","A *","B","*"), ("B","","c",""), ("B","","d",""),
  ("B","","c","*"), ("B","","d","*")],
 [("A","A * B","",""), ("A","A * B","","*")]]
```

Die Aktionstabelle $act_{LR} : Q \times (T \cup \{\varepsilon\}) \longrightarrow P \cup \{shift, error\}$ lautet:

	q_0	q_1	q_2	q_3	q_4	q_5	q_6
c	shift	error	error	error	error	shift	error
d	shift	error	error	error	error	shift	error
$*$	error	shift	$A \to B$	$B \to c$	$B \to d$	error	$A \to A * B$
ε	error	$S \to A$	$A \to B$	$B \to c$	$B \to d$	error	$A \to A * B$

Ein Parserlauf:

$$
\begin{aligned}
parse_{LR}(q_0, c * d) &= parse_{LR}(q_3 q_0, *d) && \text{wegen } act_{LR}(q_0, c) = shift \text{ und } \delta(q_0, c) = q_3 \\
&= parse_{LR}(q_2 q_0, *d) && \text{wegen } act_{LR}(q_3, *) = B \to c \text{ und } \delta(q_0, B) = q_2 \\
&= parse_{LR}(q_1 q_0, *d) && \text{wegen } act_{LR}(q_2, *) = A \to B \text{ und } \delta(q_0, A) = q_1 \\
&= parse_{LR}(q_5 q_1 q_0, d) && \text{wegen } act_{LR}(q_1, *) = shift \text{ und } \delta(q_1, *) = q_5 \\
&= parse_{LR}(q_4 q_5 q_1 q_0, \varepsilon) && \text{wegen } act_{LR}(q_5, d) = shift \text{ und } \delta(q_5, d) = q_4 \\
&= parse_{LR}(q_6 q_5 q_1 q_0, \varepsilon) && \text{wegen } act_{LR}(q_4, \varepsilon) = B \to d \text{ und } \delta(q_5, B) = q_6 \\
&= parse_{LR}(q_1 q_0, \varepsilon) && \text{wegen } act_{LR}(q_6, \varepsilon) = A \to A * B \text{ und } \delta(q_0, A) = q_1 \\
&= true && \text{wegen } act_{LR}(q_1, \varepsilon) = S \to A \quad \text{❑}
\end{aligned}
$$

Zusammengefasst erhalten wir folgende Übergangsfunktion bzw. Aktionstabelle:

	A	B	c	d	mul
0	1	2	3	4	
1					5
5		6	3	4	

	c	d	mul	end
0	shift	shift	error	error
1	error	error	shift	S A
2	error	error	A B	A B
3	error	error	B c	B c
4	error	error	B d	B d
5	shift	shift	error	error
6	error	error	A A mul B	A A mul B

Figure 3.9. *Übergangsfunktion und Aktionstabelle der obigen LR(1)-Grammatik*

Wie bei der LL-Analyse dient die Erstellung der Aktionstabelle auch der Prüfung, ob tatsächlich eine LR(1)-Grammatik vorliegt: Liefert act_{LR} eindeutige Werte, dann gilt das o.g. LR(1)-Kriterium. Andernfalls ist die Korrektheit der parse-Funktion nicht gewährleistet!

Eine Variante der LR(1)-Analyse besteht darin, zunächst die Menge Q_0 der erreichbaren **LR(0)-Zustände** zu bestimmen. Die Konstruktion von Q_0 folgt der induktiven Definition von Q (s.o.), wobei anstelle von Paaren $(A \to \alpha.\beta, x)$ nur $A \to \alpha.\beta$ gebildet wird. Nur wenn die entsprechende Aktionstabelle keine eindeutigen Werte liefert, ersetzt man das LR(0)-Item $A \to \alpha$ durch das LR(1)-Item $\{(A \to \alpha., x) \mid x \in follow(A)\}$ und bildet δ

und act_{LR} wie oben. Wird act_{LR} dabei konfliktfrei, dann nennt man G eine **SLR(1)-Grammatik**[10].

Q_0 ist i.a. gröber als Q. Eine schwächere Vergröberung von Q erreicht man durch die Identifizierung aller LR(1)-Zustände q, q', deren Items sich nur in der zweiten Komponente unterscheiden, für die also $\{p \mid \exists l : (p,l) \in q\} = \{p \mid \exists l : (p,l) \in q'\}$ gilt. Erhält man auf diese Weise eine konfliktfreie Aktionstabelle, dann heißt G **LALR(1)-Grammatik**[11]. Der YACC-Parser-Generator konstruiert LALR(1)-Parser.

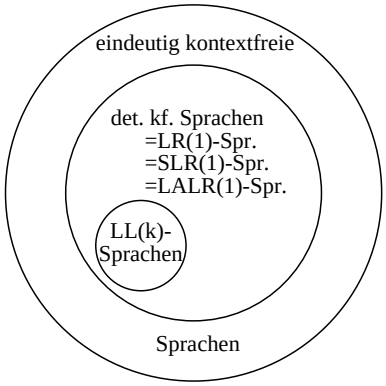


Figure 3.10. *Sprachklassen*

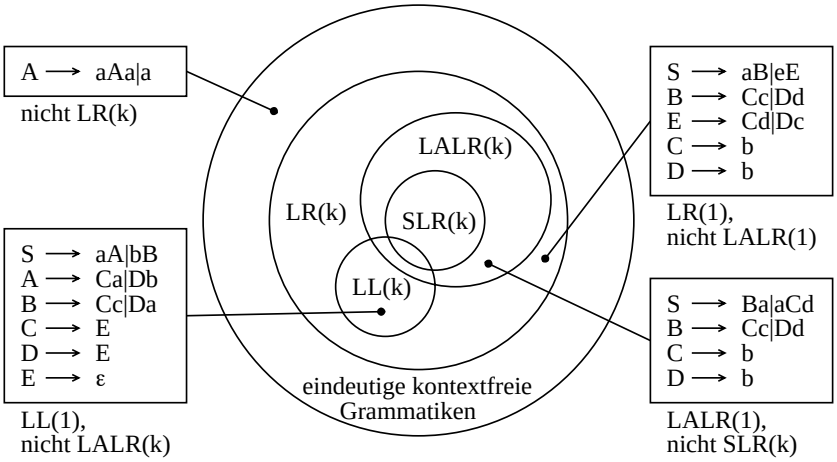


Figure 3.11. *Grammatikklassen*

Abschließend wollen wir—analog zu Abschnitt 3.4—die LR(1)-parse-Funktion zu einem interpretierenden Parser erweitern: Sei alg eine $\Sigma(G)$-Algebra.

$$parse_{LR}^{alg} \; : \; Q^* \times T^* \times alg^* \; \to \; alg \cup \{error\}.$$

Im Gegensatz zur entsprechenden LL-parse-Funktion (siehe §3.4) brauchen wir hier keinen rekursiven Abstieg. Parallel zur bottom-up-Konstruktion einer Ableitung von w aus dem Startsymbol soll $parse_{LR}^{alg}$ *iterativ* den Wert von w in alg liefern:

$$parse_{LR}^{alg}([q_0], w, []) = eval^{alg}(t),$$
wobei t ein Syntaxbaum für die Ableitung $S \xrightarrow{*}_{CF(G)} w$ ist.

[10] "simple" LR(1)-Grammatik
[11] "lookahead" LALR(1)-Grammatik

Dies ist ein Spezialfall der folgenden Invarianzbedingung an die Berechnungsschritte von $parse^{alg}_{LR}$:

$$parse^{alg}_{LR}(item(\varphi) : a, w, [t_1, \ldots, t_n]) = eval^{alg}(t[t_1/x_1, \ldots, t_n/x_n]),$$
wobei t ein Syntaxbaum für die Ableitung $S \xrightarrow{*}_{CF(G)} \varphi w_0 A_1 w_1 \ldots A_n w_n w$ ist,
der die Variablen $x_1 : A_1, \ldots, x_n : A_n$ enthält.[12]

Die Definition von $parse^{alg}_{LR}$ folgt derjenigen von $parse_{LR}$. $||\alpha||$ bezeichnet die Anzahl der Nichtterminale (!) von α.

$$\begin{aligned}
parse^{alg}_{LR}(q : a, xw, tL) &= parse^{alg}_{LR}(\delta(q, x) : q : a, w, tL) \\
&\qquad \text{falls } act_{LR}(q, x) = shift \\
parse^{alg}_{LR}(q_1 : \cdots : q_{|\alpha|} : q : a, w, tL +\!\!+ [t_1, \ldots, t_{||\alpha||}]) & \\
&= parse^{alg}_{LR}(\delta(q, A) : q : a, w, tL +\!\!+ [f^{alg}_{A \longrightarrow \alpha}(t_1, \ldots, t_{||\alpha||})]) \\
&\qquad \text{falls } act_{LR}(q_1, first(w)) = A \to \alpha \\
parse^{alg}_{LR}(q : a, \varepsilon, [t_1, \ldots, t_{||\alpha||}]) &= f^{alg}_{S \longrightarrow \alpha}(t_1, \ldots, t_{||\alpha||}) \quad \text{falls } act_{LR}(q, \varepsilon) = S \to \alpha \\
parse^{alg}_{LR}(q : a, w, tL) &= error \qquad \text{sonst}
\end{aligned}$$

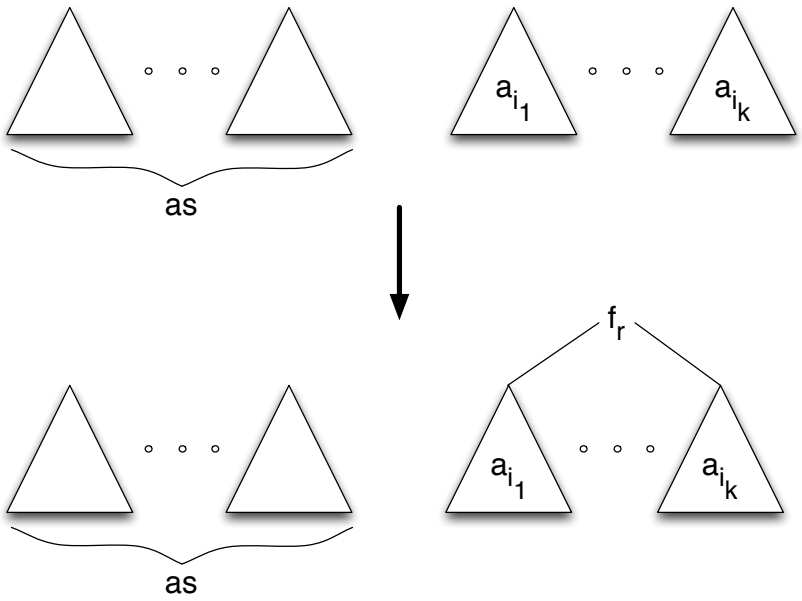


Figure 3.12. *Veränderung der Syntaxbaumliste bei einem Reduktionsschritt des LR-Parsers*

Es folgt ein mit Expander2 [60] erstellter Lauf von $parse^{B}_{LR}$ bezüglich der fünf Produktionen und der daraus ermittelten Zustandsmenge, Übergangsfunktion und Aktionstabelle von Beispiel 3.5.3. Die Knoten des schrittweise aufgebauten Syntaxbaums (letztes Argument von `parse`) sind hier direkt mit Produktionen markiert. So steht z.B. `A A mul B` für die Regel $A \to A\ mul\ B$.

```
parse(c mul d)
parse(3 0,mul d)
parse(2 0,mul d,B c)
parse(1 0,mul d,A B(B c))
parse(5 1 0,d,A B(B c))
```

[12] Hier tauchen mal Syntaxbäume mit Variablen auf (siehe Def. 3.2.2). Da die Sorten von $\Sigma(G)$ mit den Nichtterminalen von G übereinstimmen, ist die Verwendung von Nichtterminalen als Sorten von $x_1, \ldots, x_n$ korrekt.

```
parse(4 5 1 0,(),A B(B c))
parse(6 5 1 0,(),A B(B c),B d)
parse(1 0,(),A A mul B(A B(B c),B d))
S A(A A mul B(A B(B c),B d))
```

Wieder lässt sich analog zu den in §3.3 behandelten Parsern auch $parse^B_{LR}$ zu einem Compiler in eine beliebige Zielsprachenalgebra alg verallgemeinern, indem $f_{A\longrightarrow\beta}(t_1,\ldots,t_{||\alpha||})$ durch $f^{alg}_{A\longrightarrow\beta}(t_1,\ldots,t_{||\alpha||})$ ersetzt wird. tL und $t_1,\ldots,t_{||\alpha||}$ sind dann keine Baumlisten mehr, sondern Listen von (in Reduktionsschritten von $parse^B_{LR}$ berechneten) alg-Werten.

3.6 Monadische Parser

Die in §3.3 behandelten Parser unterscheiden sich in vielfacher Hinsicht von klassischen Parsern für deterministisch kontextfreie Sprachen:

- Sie werden mithilfe von **Kombinatoren** aus Teilparsern zusammengesetzt. In gewisser Weise verwendet auch die Erweiterung $parse^B_{LL}$ eines LL-Parsers zur Erzeugung von Syntaxbäumen dieses *Kompositionalitätsprinzip*, indem sie Parser $\overline{A}$ für einzelne Nichtterminale A kombiniert.
- Sie verarbeiten auch **erweiterte** kontextfreie Grammatiken und das nicht über den Umweg über deren Transformation in klassische CFGs. Die direkte Implementierung des in ECFGs auftretenden Summenoperators führt allerdings automatisch zu möglichem **backtracking** auf der Eingabe des Parsers. Durch geeignete Äquivalenztransformationen lassen sich die regulären Ausdrücke in einer ECFG jedoch optimieren in Richtung auf möglichst kleine Schnittmengen der Sprachen mit dem Summenoperator verknüpfter Teilausdrücke. So verringert man z.B. das backtracking, wenn man anstelle eines Parsers für $ab|ac$ einen für $a(b|c)$ benutzt. Wenn der erste beim Versuch, ab zu erkennen, scheitert, läuft er bis vor das a in der Eingabe zurück, während der zweite bei diesem Versuch nur bis vor b zurückgeht.[13]
  Generell ist es also möglich, nichtdeterministisch kontextfreie, ja sogar mehrdeutige Sprachen mit den Parsern von §3.3 zu verarbeiten.
- Sie lassen sich bezüglich sowohl der Quellsprache als auch der Zielsprache **generisch** formulieren und können damit nach geeigneter Instanziierung am *frontend* z.B. Scannerfunktionen und am *backend* Interpreter- oder Compilerfunktionen mit klassischer Syntaxanalyse kombinieren.

Schaut man sich die Struktur der konkreten Parser in §3.3 genauer an, dann lässt sich ein in jedem Teilparser wiederholtes Schema erkennen. Es betrifft das Lesen eines Präfixes der zu verarbeitenden Symbolfolge und die Übergabe der jeweiligen Resteingabe an andere oder auch denselben Parser. Repräsentation und Verarbeitung der jeweiligen (Rest-)Eingabe sind für den Entwurf eines konkreten Parsers weitgehend irrevelant, sollten also auch in dessen programmiersprachlicher Formulierung nicht explizit auftauchen müssen. Durch Erweiterung des Typkonstruktors `Parser sym` (siehe §3.3) zu einer Instanz der in §1.2 eingeführten Typklasse `Monad` (genauer gesagt: ihrer Typvariablen m) lässt sich die Eingabebehandlung generisch für alle Parser in diese Instanz einbauen, so dass sie nicht in jedem Teilparser wiederholt werden muss.

Wie schon in §1.2 bemerkt, steht die Variable m in der Klasse `Monad` für einen Typ zweiter Ordnung, also für eine Funktion, die Typen (erster Ordnung) in Typen abbildet. Charakteristisch für Monaden ist (und daher kommt auch ihr Name), dass m als Funktion *einstellig* ist, wie man an den Funktionen der Typklasse erkennt:

```
class Monad m where
```

[13] Man beachte, dass ein regulärer Ausdruck e in einer ECFG G i.a. Nichtterminale enthält, die ihre eigenen Parser haben, von denen der Parser für e dann abhängt. Wäre das nicht so, dann wäre $L(G)$ regulär und wir bräuchten uns keine Gedanken um Nichtdeterminismus und backtracking zu machen, denn als reguläre Sprache hätte $L(G)$ automatisch einen deterministischen Parser (siehe Kapitel 2).

```
(>>=)  :: m a -> (a -> m b) -> m b
return :: a -> m a
fail :: String -> m a
(>>)   :: m a -> m b -> m b
p >> q = p >>= const q
```

Die Instanzen von m sind – bis auf Standardmonaden wie IO (s.u.) – Typkonstruktoren von *Datentypen*. Wir wollen `Parser sym` zur Instanz von m machen, müssen dazu also `Parser sym a` als Datentyp formulieren, d.h. einen Konstruktor (und die dazu inverse Attributfunktion[14]) hinzufügen:

```
newtype MParser sym a = P {apply :: Parser sym a}
```

MParser wird nun zur Monade erweitert:

```
instance Monad (MParser sym) where
         p >>= f  = P (\syms -> case apply p syms of
                                  Result a syms -> apply (f a) syms
                                  Error str -> Error str)
         return = P . Result
         fail _ = P (const (Error "match error"))
```

Offenbar bewirkt die hier definierte Funktion

```
>>= :: MParser a -> (a -> MParser b) -> MParser b
```

die *Hintereinanderausführung* zweier Parser, wobei das vom ersten erzeugte Ergebnis a und die Resteingabe *syms* dem zweiten übergeben wird. Damit hat die Parsermonade hat den gleichen Charakter wie die IO-Monade, die als Standardmonade nicht auf einem benutzerdefinierten Datentyp aufsetzt, sondern auf einem internen Zustandstyp *state*:

```
type IO a = state -> (a,state)
instance Monad IO where
         (m >>= f) st = f a st' where (a,st') = m st
         return a st = (a,st)
         fail = error
```

Die Elemente der IO-Monade sind offenbar Zustandstransformationen, die neben dem jeweiligen Folgezustand einen Wert vom Typ a zurückgeben. >>= ist hier die Komposition solcher Zustandstransformationen: Der Aufruf $(m >>= f)st$ bewirkt zunächst die Ausführung von m im Zustand st. Zurückgegeben werden der Wert a und der Folgezustand st'. Dann wird f auf a und st' angewendet, was zur nächsten Ausgabe und zum Folgezustand von st' führt.

return bettet immer nur a in $m\ a$ ein. Daraus ergeben sich die jeweiligen Instanzen von *return* in MParser und IO fast automatisch. Das String-Argument von $fail$ ist eine Fehlermeldung, die ausgegeben wird, wenn das zweite Argument (f) von >>= eine partielle Funktion ist, die an der Stelle a nicht definiert ist (siehe die rechten Seiten der >>=-Gleichungen). >> ist bereits in der Typklasse definiert und zwar als der Spezialfall von >>=, bei dem die Ausgabe der ersten Zustandstransformation nicht an die zweite übergeben wird.

Um den imperativen *appeal* von >>= deutlicher zu machen, verwendet man meist die **do-Notation** und schreibt

[14] `SigAlg` war ein anderer Datentyp mit Attributfunktionen (siehe §3.2).

```
do x1 <- m0; x2 <- m1; x3 <- m2; ... xn <- m(n-1); mn
```

anstelle von

```
m0 >>= (\x1 -> m1 >>= (\x2 -> m2 >>= (\x3 -> ... m(n-1) >>= (\xn -> mn) ... )))
```

Bei der Auswertung dieses Ausdrucks passiert genau das, was die do-Notation suggeriert: Die Zustandstransformation m_i vom Typ $m\ a$ liefert eine Ausgabe vom Typ a, die der *Variablen* x_{i+1} als Anfangswert *zugewiesen* wird, auf den alle folgenden Zustandstransformationen $m_{i+1} \ldots m_n$ zugreifen können – sofern nicht ein x_j mit $j > i+1$ mit x_i übereinstimmt und damit der Anfangswert überschrieben wird! Die Monadenfunktion $fail$ wird hier genau dann aufgerufen, wenn x_{i+1} ein Muster ist, von dem der Ausgabewert von m_i keine Instanz ist. Die Parsermonade definiert $fail$ gerade so, dass hierbei die jeweils im Error-Konstruktor gekapselte Fehlermeldung zurückgegeben wird.

Als zweistellige Funktion hat die monadische Zuweisung den Typ `a -> m a -> m a`. Semantisch ist m_i für $0 \leq i < n$ äquivalent zu $x \leftarrow m_i$. Im ersten Fall hat man jedoch keinen Zugriff auf den von m berechneten Wert. Anstelle von Semikolons zwischen den Komponenten einer do-Sequenz können diese auch wie die Fälle einer case-Klausel linksbündig untereinander geschrieben werden, z.B.:

```
getLine :: IO String
getLine = do c <- getChar
             if c == '\n' then return "" else do cs <- getLine; return (c:cs)
```

Außerdem können Monaden auf unterschiedliche Weise verknüpft werden, genauso wie das auch mit Objekten nichtmonadischer Typen der Fall ist. Man muss nur die geeigneten Kombinatoren dafür bereitstellen:

```
done :: Monad m => m ()
done = return ()

sequence :: Monad m => [m a] -> m ()
sequence  = foldr (>>) done

mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f = sequence . map f

accumulate :: Monad m => [m a] -> m [a]
accumulate (m:ms) = do x <- m; as <- accumulate ms; return (a:as)
accumulate _      = return []

mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f = accumulate . map f

whileDo :: Monad m => m Bool -> m a -> m ()
whileDo e m = do b <- e; if b then do m; whileDo e m else done
```

Eine weitere häufig verwendete Monade ergibt sich aus dem Datentyp

```
data Maybe a = Just a | Nothing,
```

den man zur Totalisierung partieller Funktionen verwendet (siehe §1.2). Die totalisierte Version einer Funktion `f::a->b` hat dann den Typ `a -> Maybe b` und soll hier mit f' bezeichnet werden. `Maybe` wird standardmäßig wie folgt zur Monade erweitert:

```
instance Monad Maybe where
         Just a  >>= f = f a
         Nothing >>= _ = Nothing
         return = Just
         fail _ = Nothing
```

Hier gibt >>= die übliche Komposition zweier partieller Funktionen wieder: Liefert die zuerst angewendete einen undefinierten Wert, dann wird dieser von der zweiten einfach durchgereicht. Einmal `Nothing`, immer `Nothing`. Und wenn ein Funktionsaufruf zwar einen definierten Ausgabewert liefert, der aber keine Instanz des Musters ist, an das er zugewiesen wird, dann bewirkt die Definition von $fail$ in der Maybe-Monade, dass alle folgenden Funktionsaufrufe übersprungen werden und die gesamte Aufruffolge `Nothing` liefert. Damit lassen sich sehr einfach bedingte *exits* programmieren. Definiert man z.B. eine partielle Funktion $f : a \to c$ durch

```
f :: (a -> Maybe b) -> (b -> Bool) -> (b -> Maybe c) -> a -> Maybe c
f g p h a = do b <- g a; True <- Just (p a); h b
```

dann gilt: $f(a) = \begin{cases} h(g(a)) & \text{falls } g(a) \text{ definiert ist und } p \text{ für } g(a) \text{ gilt,} \\ undefined & \text{sonst.} \end{cases}$

Schließlich sei noch die (Standard-)Erweiterung des Listentyps zur Monade erwähnt:

```
instance Monad [ ] where
         as >>= f = concatMap f as
         return a = [a]
         fail _ = []
```

Aufgabe Zeige: `f as bs = [(a,b) | a <- as, b <- bs, a /= b]`, wobei f folgendermaßen definiert ist:

```
f :: Eq a => [a] -> [a] -> [(a,a)]
f as bs = do a <- as; b <- bs; True <- return (a /= b); return (a,b)
```

Die letzten beiden Monaden haben natürlich wenig mit Zustandstransformationen zu tun, illustrieren aber auch das allgemeine Ziel, mit Monaden gleichartige Berechnungen zu verstecken.

Ursprünglich stammt der Begriff aus der Kategorientheorie. Mehr zu Monaden findet man in [8, 28, 59, 80, 48], monadische Parser u.a. in [8], Kapitel 11, und [80], Kapitel 5, und eine ganze Bibliothek monadischer Parser unter www.cs.uu.nl/∼daan/parsec.html.

Zurück zur Parsermonade. Analog zur Parserstrukturierung in §3.3 werden wir zunächst monadische Parser für reguläre Ausdrücke und zwei monadische *JavaGra*-Parser in eine beliebige *JavaSig*-Algebra angeben. Der erste ist vom Typ `MParser Symbol a` und baut dementsprechend auf dem *JavaGra*-Scanner aus §2.2 auf. Der zweite ist vom Typ `MParser Char a` und realisiert die am Ende von §3.3 angesprochene Kombination von Scanner und Parser.

Zunächst zwei Hilfsparser. `item` liest ein einzelnes Symbol und gibt es zurück:

```
item :: MParser sym sym
item = P (\syms -> case syms of sym:syms -> Result sym syms
                                _ -> Error "no symbols")
```

`sat p f err` wendet den Parser p an, akzeptiert dessen Ergebnis aber nur dann, wenn es f erfüllt, sonst gibt's die Fehlermeldung *err*:

```
sat :: MParser sym a -> (a -> Bool) -> String -> MParser sym a
sat p f err = do a <- p; if f a then return a else fail err
```

Monadische Parser zur Erkennung regulärer Ausdrücke

Erkennung von sym

```
symbolM :: (Eq sym,Show sym) => sym -> Parser sym sym
symbolM sym = do sat item (== sym) ("no "++show sym)
```

Erkennung von RR' (p und q sind Parser für R bzw. R')

```
concM :: MParser sym a -> MParser sym b -> MParser sym b
concM p q = do p; q
```

Erkennung von R|R' (p und q sind Parser für R bzw. R' mit demselben Ausgabetyp)

```
parM :: MParser sym a -> MParser sym a -> MParser sym a
parM p q = P (\syms -> case apply p syms of res@(Result _ _) -> res
                                            _ -> apply q syms)
```

Erkennung von R1|...|Rn

```
parL :: [MParser sym a] -> MParser sym a
parL = foldr1 parM
```

Erkennung von R+ (p ist Parser für R)

```
plusM :: MParser sym a -> MParser sym [a]
plusM p = do a <- p; as <- starM p; return (a:as)
```

Erkennung von R (p ist Parser für R)*

```
starM :: MParser sym a -> MParser sym [a]
starM p = plusM p `parM` return []
```

Aufgabe Wie lauten die in §3.3 erwähnten allgemeineren Varianten von `conc` bzw. `par` in monadischer Form?

Monadische Versionen der in §3.3 beschriebenen Parser für drei beispielhafte Formen einer Regel $A \to e$ lauten wie folgt:

Schema 1: $A \to e$ hat die Form $A \to xByCz$ mit $B, C \in N$ und $x, y, z \in T$.

```
parseA :: SigAlg ... a b c ... -> MParser sym a
parseA alg = do x <- item; b <- parseB alg; y <- item; c <- parseC alg; z <- item
                return (f alg b c)
```

Schema 2: $A \to e$ hat die Form $A \to B|CD|CE$ mit $B, C, D, E \in N$ und $C \neq A$.

```
parseA :: SigAlg ... a b c d e ... -> MParser sym a
parseA alg = parL [do b <- parseB alg; return (f alg b),
                   do c <- parseC alg; parseArest alg c]
```

```
parseArest :: SigAlg ... a b c d e ... -> c -> MParser sym a
parseArest alg c = parL [do d <- parseD alg; return (g alg c d),
                         do e <- parseE alg; return (h alg c e)]
```

Schema 3: $A \to e$ hat die Form $A \to B|AD|AE$ mit $B, D, E \in N$.

```
parseA :: SigAlg ... a b d e ... -> Parser sym a
parseA alg = parL [do b <- parseB alg; parseArest alg (f alg b),
                   do a <- parseA alg; parseArest alg a]
parseArest :: SigAlg ... a b d e ... -> a -> Parser sym a
parseArest alg a = parL [do d <- parseD alg; return (g alg a d),
                         do e <- parseE alg; return (h alg a e),
                         return a]
```

Beispiel 3.6.1 Monadischer *JavaGra*-Symbol-Parser. Wir instanziieren die Typvariable `sym` wieder durch `Symbol`, den Ausgabetyp des *JavaGra*-Scanners (siehe Beispiel 2.2.1), und definieren monadische Parser für die vier Nichtterminale von *JavaGra* in die jeweiligen Datenbereiche einer beliebigen *JavaSig*-Algebra.

```
mBlock :: JavaAlg block a b c -> MParser Symbol block
mBlock alg = do Lcur <- item; cs <- starM (mCommand alg)
                Rcur <- item; return (block_ alg cs)

mCommand :: JavaAlg a command b c -> MParser Symbol command
mCommand alg = parL [do Semi <- item; return (skip alg),
                     do x <- ident; Upd <- item; e <- mIntE alg; Semi <- item
                        return (assign alg x e),
                     do If <- item; Lpar <- item; be <- p; Rpar <- item; b <- q
                        parL [do Else <- item; b' <- q; return (cond alg be b b'),
                              return (cond alg be b (block_ alg []))],
                     do While <- item; Lpar <- item; be <- p; Rpar <- item; b <- q
                        return (loop alg be b),
                     fail "no command"]
               where p = mBoolE alg; q = mBlock alg

mIntE :: JavaAlg a b intE c -> MParser Symbol intE
mIntE alg  = parL [do i <- number; p (intE_ alg i),
                   do x <- ident; p (var alg x),
                   do Lpar <- item; e <- mIntE alg; Rpar <- item; p e,
                   fail "no integer expression"]
             where p = mIntErest alg

mIntErest :: JavaAlg a b intE c -> intE -> MParser Symbol intE
mIntErest alg e = parL [do Minus <- item; e' <- p; return (sub alg e e'),
                        do es <- plusM (do Plus <- item; p); return (sum_ alg (e:es)),
                        do es <- plusM (do Times <- item; p); return (prod alg (e:es)),
                        return e]
                  where p = mIntE alg
```

`mIntErest` ist die monadische Version des in Beispiel 3.3.2 benutzten und erklärten Hilfsparsers `paIntErest`.

```
mBoolE :: JavaAlg a b c boolE -> MParser Symbol boolE
mBoolE alg = parL [do True_ <- item; return (boolE_ alg True),
                   do False_ <- item; return (boolE_ alg False),
                   do Neg <- item; be <- mBoolE alg; return (not_ alg be),
                   do e <- p; GR <- item; e' <- p; return (greater alg e e'),
                   fail "no Boolean expression"]
             where p = mIntE alg

number :: MParser Symbol Int
number = do sym <- sat item f "no number"; return (g sym)
         where f (Num _) = True
               f _       = False
               g (Num i) = i

ident :: MParser Symbol String
ident = do sym <- sat item f "no identifier"; return (g sym)
        where f (Ide _) = True
              f _       = False
              g (Ide x) = x
```

Die monadische Realisierung der Scanner-Parser-Kombination basiert auf Parsern vom Typ `Parser Char a`, da sie Zeichenfolgen verarbeitet. Spezielle Zeichenparser dienen der Erkennung von Zwischenräumen oder bestimmter Zeichen bzw. Strings inklusive einschließender Zwischenräume:

```
space :: MParser Char String
space = starM (sat item (`elem` " \t\n") "no blank")

token :: MParser Char a -> MParser Char a
token p = do space; a <- p; space; return a

char :: MParser Char Char
char = token item

string :: Int -> MParser Char String
string n = token (sequence (replicate n item))

isChar :: Char -> MParser Char Char
isChar = token . symbolM

isString :: String -> MParser Char String
isString = token . mapM symbolM
```

char und *string* lesen nur ein Zeichen bzw. einen String, *isChar* und *isString* prüfen zusätzlich, ob diese Eingabe mit einer vorgegebenen übereinstimmt. Wenn dabei erzeugte Fehlermeldungen übersprungen werden, weil noch alternative Parser aufgerufen werden, genügen die einfacheren Funktionen *char* und *string*. Unter Verwendung von Variablenmustern kann man nämlich auch mit *char* und *string* zu den Alternativen springen.

Beispiel 3.6.2 Monadischer *JavaGra*-Char-Parser.

```
chBlock :: JavaAlg block a b c -> MParser Char block
```

```
chBlock alg = do '{' <- char; cs <- starM (chCommand alg); '}' <- char
                 return (block_ alg cs)

chCommand :: JavaAlg _ command b c -> MParser Char command
chCommand alg = parL [do ';' <- char; return (skip alg),
                      do x <- identC; '=' <- char; e <- chIntE alg; ';' <- char
                         return (assign alg x e),
                      do "if" <- string 2; '(' <- char; be <- p; ')' <- char
                         b <- q; parL [do "else" <- string 4; b' <- q
                                          return (cond alg be b b'),
                                       return (cond alg be b (block_ alg []))],
                      do "while" <- string 5; '(' <- char; be <- p; ')' <- char
                         b <- q; return (loop alg be b),
                      fail "no command"]
                where p = chBoolE alg; q = chBlock alg

chIntE :: JavaAlg a b intE c -> MParser Char intE
chIntE alg = parL [do i <- numberC; p (intE_ alg i),
                   do x <- identC; p (var alg x),
                   do '(' <- char; e <- chIntE alg; ')' <- char; p e,
                   fail "no integer expression"]
             where p = chIntErest alg

chIntErest :: JavaAlg a b intE c -> intE -> MParser Char intE
chIntErest alg e = parL [do '-' <- char; e' <- p; return (sub alg e e'),
                         do es <- plusM (do '+' <- char; p); return (sum_ alg (e:es)),
                         do es <- plusM (do '*' <- char; p); return (prod alg (e:es)),
                         return e]
                   where p = chIntE alg

chBoolE :: JavaAlg a b c boolE -> MParser Char boolE
chBoolE alg = parL [do "true" <- string 4; return (boolE_ alg True),
                    do "false" <- string 5; return (boolE_ alg False),
                    do '!' <- char; be <- chBoolE alg; return (not_ alg be),
                    do e <- p; '>' <- char; e' <- p; return (greater alg e e'),
                    fail "no Boolean expression"]
              where p = chIntE alg

numberC :: MParser Char Int
numberC = do ds <- token (plusM (sat item isDigit "no digit"))
             return (read ds::Int)

identC :: MParser Char String
identC = token (sat (plusM (sat item (not . isSpecial) err)) f err)
         where f x = x `notElem` words "true false if else while"
               err = "no identifier"
```

Kapitel 4

Attributierte Übersetzung

Zusammengefasst erlauben uns die Erkenntnisse aus Kapitel 3 eine Präzisierung der Begriffe Quell- und Zielsprache einer Übersetzungsfunktion $comp : Q \to Z$ (siehe Def. 1.1.1): Ausgehend von einer ECFG G ist $Q = L(G)$, Z eine $\Sigma(G)$-Algebra und f darstellbar als die Komposition eines Parsers $p : L(G) \to T_{\Sigma(G)}$ und der $\Sigma(G)$-Auswertungsfunktion in A (siehe Def. 3.2.10).

In diesem Kapitel werden wir eine Klasse von Algebren genauer betrachten, die die meisten Zielsprachen umfasst, von klassischen Assemblersprachen über die Eingabesprachen graphischer Interpreter (Postscript, SVG, et al.) bis hin zu sog. Auszeichnungssprachen (HTML, XML), die von Webbrowsern verstanden werden (sollen). Jede Algebra dieser Klasse hat in der Regel funktionale Datenbereiche mit mehreren Argument- wie auch Wertkomponenten. Von letzteren bildet meistens nur eine das eigentliche Ergebnis der Übersetzung (Zielprogramm), während die anderen Wertkomponenten zusätzliche **Attribute** darstellen, die jenes Ergebnis zwar beeinflussen, aber am Ende der Übersetzung vergessen werden können. Während solche Attribute **abgeleitet** oder **synthetisiert** (*derived* oder *synthesized*) genannt werden, bezeichnet man die Argumentkomponenten der funktionalen Datenbereiche als **vererbte** Attribute (*inherited attributes*).

Ein vererbtes Attribut, das bei der Übersetzung blockstrukturierter Programme benötigt wird, ist die **Schachtelungstiefe** von Teilprogrammen im Gesamtprogramm. Diese ist aber nur ein Beispiel einer großen Klasse vererbter Attribute, die Positionen, Adressen, Ziffernstellen o.ä. angeben, an der das Zielobjekt der Übersetzung platziert wird. Vererbte Attribute repräsentieren immer eine *kontextabhängige* Information, die initialisiert und dann an die Komponenten des Quellprogramms – mit möglicherweise verändertem Wert – weitergereicht wird. Der Wert eines abgeleiteten Attributs hingegen errechnet sich aus der jeweiligen Programmkomponente selbst und den an sie übergebenen Werten vererbter Attribute. Abgeleitet sind deshalb z.B. der Typ eines Ausdrucks, der (Platzbedarf für den) Wert einer Variable und in jedem Fall das Zielprogramm selbst. Manche Attribute sind sowohl vererbt als auch abgeleitet und werden dann oft **transitional** genannt. Die Werte eines solchen Attributs *at* bezeichnet man in der Regel als **Zustände** und die funktionalen Bereiche, bei denen *at* als vererbtes wie auch als abgeleitetes Attribut auftritt, als **Zustandstransformationen**. Demnach sind z.B. die bei der Übersetzung imperativer Programme erstellte Symboltabelle und der Wert der nächsten freien für Zielkommandos verfügbaren Adresse transitionale Attribute (siehe Kapitel 5).

Zwei in diesem Sinne **attributierte Algebren** haben wir in bereits in den Beispielen 3.2.16 und 3.2.15 kennengelernt. Im ersten sind die Anfangsposition der jeweiligen Ausgabezeile (vom Typ `Int`) und das Prädikat "ist erste Komponente einer Liste" (vom Typ `Bool`) vererbte Attribute, während der ausgebene String natürlich ein abgeleitetes Attribut ist. Im zweiten Beispiel bilden die Variablenbelegungen vom Typ `State` ein für Kommandos transitionales und für Ausdrücke nur vererbtes Attribut.

Die Zuordnung eines Attributes zu "vererbt", "abgeleitet" oder "transitional" ist also nicht immer eindeutig.

Deshalb gehen wir bei einem allgemeinen Schema für attributierte Algebren zunächst von einer Menge $At = \{At_1, \ldots, At_n\}$ aller Attribute aus und ordnen dann – ausgehend von einer ECFG G – den (funktionalen) Datenbereichen einer $\Sigma(G)$-Algebra Elemente von At zu. Natürlich ist die Zuordnung von Attributen zu ihren jeweiligen Wertebereichen noch weniger eindeutig (z.B. sind viele Attribute vom Typ `Int`). Deshalb dürfen die Elemente von At nicht mit ihren Wertebereichen identifiziert werden und wir implementieren jedes Element von At als einen Datentyp mit genau einem Konstruktor:

```
newtype At_1 = At_1 typ_1; ...; newtype At_n = At_n typ_n
```

typ_i ist der Wertebereich von At_i. Das ist nun die dritte Verwendung von Haskell-Datentypen (`data` oder `newtype`) in Compilerdefinitionen: Zunächst wurden damit Terme, insbesondere Syntaxbäume implementiert. Dann haben wir Algebren als Objekte eines Datentyps realisiert. Und jetzt dienen sie dazu, den Namen eines Attributs von seinem Wertebereich zu unterscheiden.

Definition 4.1 Sei $\Sigma = (N, C)$ eine Signatur, $At = \{At_1, \ldots, At_n\}$ eine Menge von Attribut(typ)en und A eine Σ-Algebra A (siehe Def. 3.2.7) ist **At-attributiert**, wenn es für alle $s \in N$

$$Inh_{s,1}, \ldots, Inh_{s,m_s}, Der_{s,1}, \ldots, Der_{s,n_s} \in At$$

gibt mit

$$A_s \quad = \quad Inh_{s,1} \times \ldots \times Inh_{s,m_s} \;\rightarrow\; Der_{s,1} \times \ldots \times Der_{s,n_s} \tag{4.1}$$

und für alle $c : e \rightarrow s \in C$ c^A durch eine (Haskell-)Definition der folgenden Form gegeben ist: Seien $s_1, \ldots, s_n$ alle Vorkommen von Nichtterminalen in e. Für alle $1 \leq i \leq n$ sei $f_i \in A_{s_i}$.

$$\begin{aligned} c^A(f_1, \ldots, f_n)(Inh_{s,1}\ x_{s,1}, \ldots, Inh_{s,m_s}\ x_{s,m_s}) &= (Der_{s,1}\ e_{s,1}, \ldots, Der_{s,n_s}\ e_{s,n_s}) \\ \text{where } (Der_{s_1,1}\ x_{s_1,1}, \ldots, Der_{s_1,n_{s_1}}\ x_{s_1,n_{s_1}}) &= f_1(Inh_{s_1,1}\ e_{s_1,1}, \ldots, Inh_{s_1,m_{s_1}}\ e_{s_1,m_{s_1}}) \\ &\vdots \\ (Der_{s_n,1}\ x_{s_n,1}, \ldots, Der_{s_n,n_{s_1}}\ x_{s_n,n_{s_n}}) &= f_n(Inh_{s_n,1}\ e_{s_n,1}, \ldots, Inh_{s_n,m_{s_n}}\ e_{s_n,m_{s_n}}) \end{aligned} \tag{4.2}$$

Während für alle $s \in S$ und $i \in \mathbb{N}$ $x_{s,i}$ eine Variable ist, bezeichnet $e_{s,i}$ einen Ausdruck, der beliebige dieser Variablen enthalten kann. Folglich bilden die lokalen Definitionen von (4.2) ein Gleichungssystem in den (lokalen) Variablen $x_{s_i,i}, \ldots, x_{s_i,n_{s_i}}$, $1 \leq i \leq n$, das für jede Belegung der (globalen) Variablen $x_{s,1}, \ldots, x_{s,m_s}$ gelöst werden muss, um die Ausdrücke $e_{s,i}, \ldots, e_{s,n_s}$ und damit den Wert von c^A zu berechnen. Wird die in der i-ten Gleichung definierte lokale Variable $x_{s_i,k}$ nur in darauffolgenden Gleichungen benutzt, also in Ausdrücken der Form $e_{s_j,l}$ mit $i < j$, dann erhält man die gewünschte Lösung aller n Gleichungen durch sequentielle Auswertung ihrer rechten Seiten. Dies nennt man **einpässige Übersetzung**. Natürlich muss nicht nur (4.2) die genannte Bedingung erfüllen, sondern die entsprechenden Gleichungssysteme für andere Konstruktoren müssen ihr ebenfalls genügen.

Wir kommen damit in den oben zitierten Beispielen 3.2.16 und 3.2.15 aus und werden im folgenden Abschnitt drei weitere kleine Beispiele vorstellen, wo alle Berechnungen in einer attributierten $\Sigma(G)$-Algebra – seien es jene, die der von $\Sigma(G)$ induzierte generische Interpreter ausführt (siehe Def. 3.2.10), oder jene, die ein von G induzierter Parser (siehe §3.3), in einem Pass erfolgen können. Auch die in Kapitel 5 detailliert ausgeführte Übersetzung einer imperativen Sprache mit geschachtelten rekursiven Prozeduren ist einpässig. Dort wäre es vor allem die Aufgabe der Forderung, dass Programmvariablen stets vor ihrer Verwendung deklariert werden müssen, die eine mehrpässige Übersetzung verlangen würde. Worin die im allgemeinen besteht, ist das Thema von §4.2.

Attributierte Grammatiken wurden von Donald Knuth erfunden [36]. Ihre hier verwendete Darstellung als Systeme rekursiver Gleichungen geht zurück auf [14] und wurde z.B. in [45] weiterverfolgt.

4.2 Einpässige Übersetzung

Definition 4.2.1 Sei $\Sigma = (N, C)$ eine Signatur, $At = \{At_1, \ldots, At_n\}$ eine Menge von Attribut(typ)en und A eine At-attributierte Σ-Algebra. Die in Def. 3.2.10 angegebene Definition von $eval^A : T_\Sigma \to A$ ist ein **Ein-Pass-Compiler**, falls für alle Konstruktoren $c \in C$, $1 \leq i \leq n$ und $1 \leq k \leq n_{s_i}$ die in (4.2) definierte Variable $x_{s_i,k}$ nur in Ausdrücken $e_{s_j,l}$ mit $i < j$ vorkommt. ❑

Beispiel 4.2.2 Strings mit Hoch- und Tiefstellungen

konkrete Syntax G	**abstrakte Syntax** $\Sigma(G)$
$TEXT \to STRING$	$getText : STRING \to TEXT$
$STRING \to STRING\ BOX \mid$	$app : STRING\ BOX \to STRING$
$\quad STRING \uparrow BOX \mid$	$up : STRING\ BOX \to STRING$
$\quad STRING \downarrow BOX \mid$	$down : STRING\ BOX \to STRING$
$\quad \varepsilon$	$empty : STRING$
$BOX \to (STRING) \mid a \mid b \mid c$	$mkBox : STRING \to BOX$
	$a, b, c : BOX$

Attribute:

```
data Pos = Pos Int Int
data LHT = LHT Int Int Int
data Target = Target <Strings mit Hoch- und Tiefstellungen>
```

Pos liefert die Koordinaten der linken unteren Ecke des Rechtecks, in das der eingelesene String geschrieben werden soll. *LHT* liefert die Länge sowie—auf eine feste Grundlinie bezogen—die Höhe und die Tiefe des Rechtecks.

$\Sigma(G)$-Algebra A mit Attributen Pos, LHT und $Target$:

$$A_{TEXT} = Target$$
$$A_{STRING} = A_{BOX} = Pos \to LHT \times Target$$

$$getText^A : A_{STRING} \to A_{TEXT}$$
$$getText^A(f) = Target\ text \text{ where } (LHT\ l\ h\ t, Target\ text) = f(Pos\ 0\ 0)$$

$$app^A : A_{STRING} \times A_{BOX} \to A_{STRING}$$
$$app^A(f,g)(Pos\ x\ y) = (LHT\ (l+l')\ (max\ h\ h')\ (max\ t\ t'), Target\ (zz'))$$
$$\text{where } (LHT\ l\ h\ t, Target\ z) = f(Pos\ x\ y)$$
$$(LHT\ l'\ h'\ t', Target\ z') = f(Pos\ (x+l)\ y)$$

$$up^A : A_{STRING} \times A_{BOX} \to A_{STRING}$$
$$up^A(f,g)(Pos\ x\ y) = (LHT\ (l+l')\ (h+h'-1)\ (max\ t\ (t'-h+1)), Target\ (z^{z'}))$$
$$\text{where } (LHT\ l\ h\ t, Target\ z) = f(Pos\ x\ y)$$
$$(LHT\ l'\ h'\ t', Target\ z') = f(Pos\ (x+l)\ (y+h-1))$$

$$down^A : A_{STRING} \times A_{BOX} \to A_{STRING}$$
$$down^A(f,g)(Pos\ x\ y) = (LHT\ (l+l')\ (max\ h\ (h'-t-1))\ (t+t'+1), Target\ (z_{z'}))$$
$$\text{where } (LHT\ l\ h\ t, Target\ z) = f(Pos\ x\ y)$$
$$(LHT\ l'\ h'\ t', Target\ z') = f(Pos\ (x+l)\ (y-t-1))$$

$$empty^A : A_{STRING}$$
$$empty^A(x,y) = (LHT\ 0\ 0\ 0, \sqcup)$$

$$mkBox^A : A_{STRING} \to A_{BOX}$$
$$mkBox^A = id$$

$$a^A, b^A, c^A : A_{BOX}$$
$$a^A(x,y) = (LHT\ 1\ 2\ 0, Target\ a)$$
$$b^A(x,y) = (LHT\ 1\ 2\ 0, Target\ b)$$
$$c^A(x,y) = (LHT\ 1\ 2\ 0, Target\ c)$$

Beispiel 4.2.3 Binärdarstellung rationaler Zahlen (nach [36])

konkrete Syntax G	**abstrakte Syntax $\Sigma(G)$**
$RAT \to NAT. \mid RAT0 \mid RAT1$	$mkRat : NAT \to RAT$
	$app0 : RAT \to RAT$
	$app1 : RAT \to RAT$
$NAT \to 0 \mid 1 \mid NAT0 \mid NAT1$	$0, 1 :\to NAT$
	$app0 : NAT \to NAT$
	$app1 : NAT \to NAT$

Attribute:

```
data Inc = Inc Float
```

```
data Val = Val Float
```

Inc liefert das Inkrement, um das sich der Dezimalwert einer rationalen Zahl erhöht, wenn die Mantisse ihrer Binärdarstellung um eine 1 erweitert wird. Val liefert den Dezimalwert einer rationalen Zahl.
$\Sigma(G)$-Algebra A mit Attributen Inc und Val:

$$A_{RAT} = Inc \times Val$$
$$A_{NAT} = Val$$

$$mkRat^A : Val \to Inc \times Val$$
$$mkRat^A(val) = (Inc\ 1, val)$$

$$app0^A : Inc \times Val \to Inc \times Val$$
$$app0^A(Inc\ inc, val) = (Inc\ (inc/2), val)$$

$$app1^A : Inc \times Val \to Inc \times Val$$
$$app1^A(Inc\ inc, Val\ r)) = (Inc\ (inc/2), Val\ (r + inc/2))$$

$$0^A, 1^A : Val$$
$$0^A = Val\ 0$$
$$1^A = Val\ 1$$

$$app0^A : Val \to Val$$
$$app0^A(Val\ n) = Val\ (n * 2)$$

$$app1^A : Val \to Val$$
$$app1^A(Val\ n) = Val\ (n * 2 + 1)$$

Beispiel 4.2.4 Übersetzung regulärer Ausdrücke in endliche Automaten

Reguläre Ausdrücke liegen bereits in abstrakter Syntax vor (siehe §2.1):

```
data RegExp = Const String | Sum RegExp RegExp | Prod RegExp RegExp | Plus RegExp
```

Daher verzichten wir auf eine konkrete Syntax und benötigen nur geeignete Attribute, um daraus eine Algebra A zu bilden, die die Konstruktoren von *RegExp* so interpretiert, dass die gewünschte Übersetzungsfunktion der *RegExp*-Auswertungsfunktion in A entspricht (siehe §3.2). Die geeigneten Attribute ergeben sich aus Fig. 2.1, wo die Übersetzung mithilfe von Graphersetzungsregeln beschrieben wurde.
Attribute:

```
data IniFin = IniFin Int Int
data Next = Next Int
data Trans = Trans (Int -> String -> [Int])
```

$IniFin$ liefert den aktuellen Anfangs- bzw. Endzustand des erzeugten Automaten. $Next$ ist ein vererbtes Attribut. Es liefert die "nächste freie" ganze Zahl, die als Zustandsname vergeben werden kann. $Trans$ ist die aktuelle Übergangsfunktion des erzeugten Automaten.

RegExp-**Algebra** A **mit Attributen** $IniFin$, $Next$ **und** $Trans$:

$$A = IniFin \times Next \times Trans \to Next \times Trans$$

$$Const^A : String \to A$$
$$Const^A(str)(IniFin\ q\ q', next, trans) = (next, update2\ trans\ q\ str\ q')$$
$$Eps^A(IniFin\ q\ q', next, trans) = (next, update2\ trans\ q\ Nothing\ q')$$

$$Sum^A : A \times A \to A$$
$$Sum^A(f,g)(iniFin, next, trans) = g(iniFin, next', trans')$$
$$\text{where } (next', trans') = f(iniFin, next, trans)$$

$$Prod^A : A \times A \to A$$
$$Prod^A(f,g)(IniFin\ q\ q', Next\ q'', trans) = g(IniFin\ q''\ q', next', trans')$$
$$\text{where } (next', trans') = f(IniFin\ q\ q'', Next\ (q''+1), trans)$$

$$Plus^A : A \to A$$
$$Plus^A(f)(IniFin\ q\ q', Next\ q'', trans) = (next', update2\ trans3\ q1\ \ddot{e}ps - q')$$
$$\text{where } q_1 = q'' + 1$$
$$(next', trans1) = f(IniFin\ q''\ q_1, Next\ (q_1+1), trans)$$
$$trans2 = update2\ trans1\ q\ \ddot{e}ps - q''$$
$$trans3 = update2\ trans2\ q1\ \ddot{e}ps - q''$$

Die Definition von A ergibt sich aus den Graphersetzungsregeln von Fig. 2.1. `update2 trans q a q'` fügt den Übergang $q \xrightarrow{a} q'$ zur Übergangsfunktion hinzu:

```
update2 :: Trans -> Int -> String -> Int -> Trans
update2 (Trans delta) q a q' q1 a1 = Trans delta'
                        where qs = delta q1 a1
                              delta' q1 a1 = if q == q1 && a == a1 then q':qs else qs
```

Ein Aufruf `regToAuto e` der Funktion

```
regToAuto :: Eq a => RegExp a -> IntAuto a
```

(siehe §2.2) erweitert die von A berechnete Übergangsfunktion um die anderen Komponenten des e erkennenden Automaten: Zustandsmenge, Eingabemenge, Ausgabefunktion (die hier prüft, ob q = 1 gilt, weil 1 hier der einzige Endzustand ist) und Anfangszustand 0:

```
regToAuto e = ([0..q-1],symbols e,(delta,\q->[q==1],0))
      where (Next q,Trans delta) = eval^A e (IniFin 0 1) (Next 2) (Trans const (const []))
            symbols (Const a)   = [a]
            symbols (Sum e e')  = symbols e `join` symbols e'
            symbols (Prod e e') = symbols e `join` symbols e'
            symbols (Plus e)    = symbols e
```

Hierbei ist $eval^A$ die *RegExp*-Auswertungsfunktion in A.

4.3 Mehrpässige Übersetzung

Ist die in Def. 3.2.10 angegebene Definition der Auswertungsfunktion in A kein Ein-Pass-Compiler, dann genügt manchmal der Austausch einzelner Konstruktorargumente, um (4.2) in eine Form zu bringen, die der Bedingung von Def. 4.2.1 genügt. Hilft der "horizontale" Austausch nicht weiter, dann vielleicht ein vertikaler, der darin besteht, die gesamte Attributmenge At so in r Teilmengen $At^1, \ldots, At^r$ zu zerlegen, dass für alle Variablen $Der_{s_i,k}(x)$ und Ausdrücke $Inh_{s_j,l}(e)$ von (4.2) mit $x \in e$ gilt:

$$i < j \quad \textbf{oder} \quad \exists\, i', j' \in \mathbb{N} : i' < j' \wedge Der_{s_i,k} \in At^{i'} \wedge Inh_{s_j,l} \in At^{j'}. \tag{4.3}$$

Dann kann auch $eval^A$ zerlegt werden, und zwar in r Funktionen (*Pässe*) $eval^1_s, \ldots, eval^r_s$. Der Ausdruck e kann im j-ten Pass berechnet werden, weil der dazu erforderliche Wert von x bereits im früheren i-ten Pass ermittelt wurde. Diese Funktionen erzeugen bzw. transformieren *At-annotierte Syntaxbäume*:

Definition 4.3.1 Sei $\Sigma = (N, C)$ eine Signatur und $At = \{At_1, \ldots, At_n\}$ eine Menge von Attribut(typ)en. Die N-sortierte Menge T^{At}_Σ der **At-annotierten Σ-Terme** ist induktiv definiert: Für alle $s \in S$ sei $D_s = Der_{s,1} \times \ldots \times Der_{s,n_s}$ (siehe Def. 4.1).

- Für alle $c : \varepsilon \to s \in \Sigma$ und Teiltupel a von Elementen von D_s ist $[c, a] \in T^{At}_{\Sigma,s}$.
- Für alle $c : e \to s \in \Sigma$ mit $e \neq \varepsilon$, $t \in T^{At}_{\Sigma,e}$ und Teiltupel a von Elementen von D_s ist $[c, a](t) \in T_{\Sigma,s}$. ❑

Sei $1 \leq i \leq r$, $1 \leq i_1, \ldots, i_m \leq n$, $At' = At_{i_1} \times \ldots \times At_{i_m}$, $\{j_1, \ldots, j_n\} = \{k \in \{i_1, \ldots, i_m\} \mid At_k \in At^i\}$ und $a = (a_{i_1}, \ldots, a_{i_m}) \in At'$.

$$\begin{aligned} \pi^i(a) &=_{def} (a_{j_1}, \ldots, a_{j_n}), \\ \pi^i(At') &=_{def} \{\pi^i(a) \mid a \in At'\}. \end{aligned}$$

π^i projiziert also ein Tupel von At' auf das Teiltupel der Komponenten, die zu At^i gehören, also im i-ten Pass berechnet werden.

$eval^1_s, \ldots, eval^r_s$ annotieren einen Syntaxbaum schrittweise mit Werten abgeleiteter Attribute: Sei $1 \leq i \leq r$, $s \in N$ und $t \in T_{\Sigma,s}$.

$$\begin{aligned} &eval^A_s : T_{\Sigma,s} \to (Inh_{s,1} \times \ldots \times Inh_{s,m_s}) \to (Der_{s,1} \times \ldots \times Der_{s,n_s}) \\ &eval^A_s(t)(x) =_{def} attrs(root(t^r)) \text{ where } \begin{aligned} t^1 &= eval^1_s(t)(\pi^1(x)) \\ &\vdots \\ t^r &= eval^r_s(t^{r-1})(\pi^r(x)) \end{aligned} \end{aligned}$$

$attrs(root(t^r))$ liefert das Attributtupel in der Wurzel des mit allen abgeleiteten Attributen annotierten Terms t^r. Allgemein sei ein annotierter Term, dessen Wurzel mit dem Konstruktor c und dem Attributtupel a markiert ist, durch $[c, a](t_1, \ldots, t_n)$ dargestellt.

Wir fassen die Komponenten jedes Tupel von Variablen oder Ausdrücken in (4.2) zu jeweils einer Variablen bzw. einem Ausdruck zusammen und erhalten damit folgende Kurzversion von (4.2):

$$\begin{aligned} c^A(f_1, \ldots, f_n)(x) = e \text{ where } x_1 &= f_1(e_1) \\ &\vdots \\ x_n &= f_n(e_n) \end{aligned} \tag{4.4}$$

Angenommen, wir haben eine Zerlegung von At in r Teilmengen $At^1, \ldots, At^r$ gefunden, die (4.3) erfüllt, dann ergibt sich die Definition der r Pässe direkt aus (4.4): Sei $1 \leq i \leq r$, $s \in N$ und $[c,a](t_1,\ldots,t_n) \in T^{At}_{\Sigma,s}$.

$$
\begin{aligned}
&eval^i_s : T^{At}_{\Sigma,s} \to \pi^i(Inh_{s,1} \times \ldots \times Inh_{s,m_s}) \to T^{At}_{\Sigma,s}\\
&eval^i_s([c,a](t_1,\ldots,t_n))(\pi^i(x)) = \quad [c,a,\pi^i(e)](u_1,\ldots,u_n)\\
&\qquad\qquad\qquad\qquad\qquad\quad \text{where } u_1 = eval^i_{s_1}(t_1)(\pi^i(e_1))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\vdots\\
&\qquad\qquad\qquad\qquad\qquad\quad u_n = eval^i_{s_n}(t_n)(\pi^i(e_n))
\end{aligned}
$$

Die Zerlegung von At basiert auf dem **Abhängigkeitsgraphen** (*dependency graph*)

$$DG : C \to (At \times At) \to \wp(\mathbb{N} \times \mathbb{N}).$$

Er liefert für alle Konstruktoren $c \in C$ und Attributpaare[1] $(A,D) \in At \times At$ die Paare (i,j) von Indizes derart, dass, bezogen auf (4.2), $1 \leq k \leq n_{s_i}$ und $1 \leq l \leq m_{s_j}$ existieren mit $A = Der_{s_i,k}$, $D = Inh_{s_j,l}$ und $x_{s_i,k} \in e_{s_j,l}$, kurz:

$$(i,j) \in DG(c)(Inh_{s_i,k}, Der_{s_j,l}) \iff_{def} \quad x_{s_j,l} \text{ kommt in } e_{s_i,k} \text{ vor.}$$

Gilt $i < j$ für alle Elemente (i,j) von $\cup\{DG(c)(A,D) \mid c \in C,\ (D,A) \in At \times At\}$. dann ist keine Zerlegung von At erforderlich, weil (4.2) bereits in einem Durchgang lösbar ist. Andernfalls liefert der unten beschriebene LAG-Algorithmus die kleinste Zerlegung $\{At_1, \ldots, At_r\}$, die (4.3) erfüllt – falls überhaupt eine solche existiert.

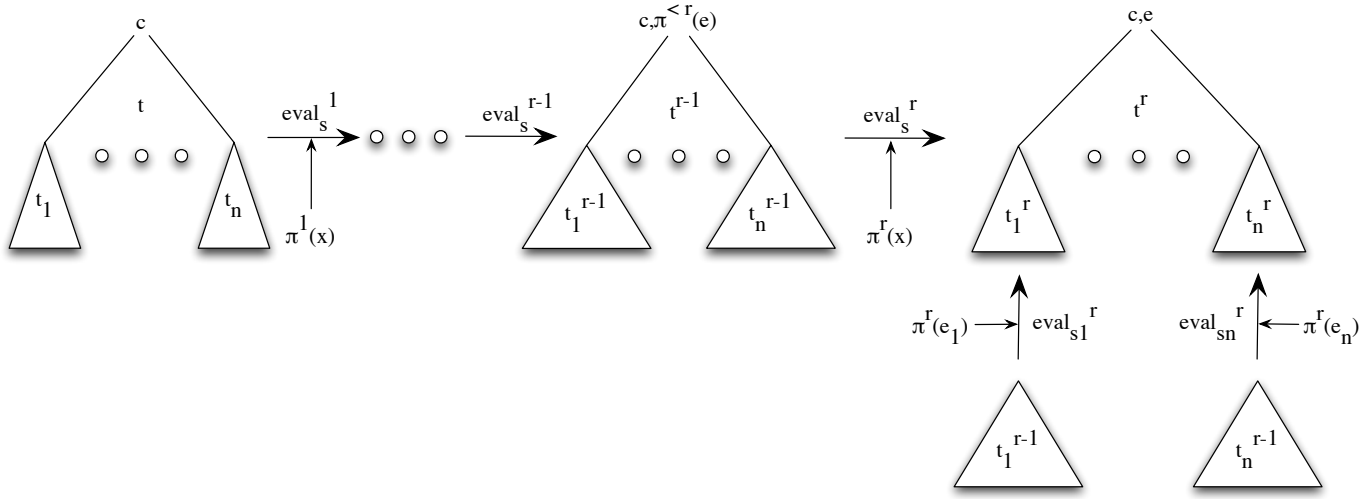


Figure 4.1. *Schrittweise Annotation eines Syntaxbaums*

Der LAG-Algorithmus ist ein Entscheidungsverfahren. Er terminiert also auch, wenn es keine Zerlegung gibt, die (4.3) erfüllt. Gibt es sie, dann nennt man die Ausgangsgrammatik eine **LAG(r)-Grammatik** (**r**-Pass-**L**inks-Rechts-**A**ttribut-**G**rammatik), weil jede Liste (annotierter) Unterbäume, die an derselben Wurzel hängen, von links nach rechts traversiert wird. Der Algorithmus liefert also die kleinste Zahl r derart, dass G eine LAG(r)-Grammatik ist.

Beispiel 4.3.2 einer LAG(2)-Grammatik.

```
-- Abstrakte Syntax
        data Z = C1 A B
```

[1] A für "appliziert", D für "definiert".

```
        data A = C2
        data B = C3

-- Attribute
        newtype A1 = A1 Int
        newtype A2 = A2 Int
        newtype B1 = B1 Int
        newtype B2 = B2 Int
        newtype Result = Result Int
```

Der Abhängigkeitsgraph laute wie folgt:

```
        DG C1 (B2,A1)     = [(2,1)]
        DG C1 (A2,Result) = [(1,2)]
        DG C2 (A1,A2)     = [(1,3)]
        DG C3 (B1,B2)     = [(1,2)]
        DG _ _            = []
```

Kritisch ist die Abhängigkeit von A1 von B2. B2 wird zur Definition von A1 benötigt. Wegen 2>1 kann A1 erst im zweiten Pass berechnet werden. Da aber A2 auf A1 zugreift und Result auf A2, können auch A2 und Result erst im zweiten Pass berechnet werden. Da es keine weiteren Abhängigkeiten gibt, genügt eine zweielementige Zerlegung mit $At^1 = \{B1, B2\}$ und $At^2 = \{A1, A2, Result\}$.

Beispiel 4.3.3 einer Grammatik, die für keine Zahl r eine LAG(r)-Grammatik ist.

```
-- Abstrakte Syntax
        data Z = C1 (D,E) | C2 (D,E)
        data D = C3
        data E = C4

-- Attribute
        newtype SZ = SZ Int
        newtype ID = ID Int
        newtype SD = SD Int
        newtype IE = IE Int
        newtype SE = SE Int
```

Der Abhängigkeitsgraph laute wie folgt:

```
        DG C1 (SE,ID) = [(1,2)]
        DG C1 (SD,SZ) = [(2,3)]
        DG C2 (SD,IE) = [(2,1)]
        DG C2 (SE,SZ) = [(1,2)]
        DG C3 (ID,SD) = [(1,2)]
        DG C4 (IE,SE) = [(1,3)]
        DG _ _        = []
```

Kritisch ist die Abhängigkeit von IE von SD. SD wird zur Definition von IE benötigt. Wegen $2 > 1$ kann IE erst im zweiten Pass berechnet werden. Nun greifen aber SZ auf SD, SD auf ID, ID auf SE und schließlich SE

auf IE zu. Also kann im ersten Pass überhaupt kein Attribut berechnet werden, d.h. es gibt keine mit dem Abhängigkeitsgraphen verträgliche Reihenfolge, in dem die Attribute berechnet werden könnten.

Implementierung des LAG-Algorithmus.[2]

Sei `constrs` die Menge aller Konstruktoren und `attrs` die Menge aller Attribute der gegebenen Grammatik. Ausgehend von der einelementigen Zerlegung $\{attrs\}$ verändert *check_partition* die ersten beiden Elemente der jeweils aktuellen Zerlegung (*next* und *curr*), bis entweder *next* leer ist und damit eine (4.3) erfüllende Zerlegung gefunden ist oder *curr* leer ist, was anzeigt, dass keine (4.3) erfüllende Zerlegung existiert.

```
least_partition ats = reverse . check_partition [ats] []

check_partition next (curr:partition) =
  if changed then check_partition next' (curr':partition)
  else case (next',curr') of
            ([],_) -> curr':partition                              Zerlegung gefunden
            (_,[]) -> []                                           keine Zerlegung möglich
            _ -> check_partition [] (next':curr':partition)        Zerlegung erweitern
where (next',curr',changed) = foldl check_constr (next,curr,False) constrs

check_constr state c = foldl (check_atp (DG c)) state
                             [(a,d) | a <- ats, d <- ats, not (null (DG c) (a,d))]

check_atp dg state atp = foldl (check_dep atp) state (dg atp)

check_dep (a,d) (next,curr,changed) (i,j) =
  if (a `elem` next || (a `elem` curr && i>=j)) && d `elem` curr
                                          Die aktuelle Zerlegung next:curr:... verletzt (4.3).
  then (d:next,curr`minus`[d],True) else (next,curr,changed)
       d wird vom vorletzten Zerlegungselement curr zum letzten Zerlegungselement next verschoben.
```

Mehrpässige Übersetzung ist zum Beispiel dann erforderlich, wenn Programmvariablen vor ihrer Deklaration verwendet werden dürfen. Da der Compiler zur Berechnung des Speicherbedarfs für den Wert einer Variablen deren Typ kennen muss, sind mindestens zwei Pässe erforderlich: Im ersten Pass werden alle Deklarationen gesucht und den Variablen Typen zugeordnet, die dann im zweiten Pass die Übersetzung von Variablenaufrufen in typabhängige Ladebefehle erlauben (siehe Abschnitte 5.1.2 und 5.1.4). Sind gar keine Deklarationen erforderlich, weil der Compiler Typen direkt aus den Applikationen der Variablen ableiten kann (wie in §7.1 ausgeführt), dann geht dies natürlich auch nur in einem der Übersetzung in die Zielsprache vorgeschalteten Pass.

Große Quellprogramme erzeugen mindestens ebenso große Syntaxbäume. Mehrpässige Compiler können deshalb zu langen Übersetzungszeiten führen. Einpässige Compiler haben demgegenüber einen hohen Speicherbedarf. Da dies aber heute nicht mehr so kritisch ist, geht die Tendenz in Richtung 1-Pass-Compiler, während früher bis zu vier (*Fortran*) oder gar sechs Pässe (*Algol68*) üblich waren. Anstatt den Syntaxbaum mehrmals zu traversieren, wird oft erst der Zielcode in mehreren Pässen schrittweise aufgebaut (*Backpatching*).

[2] Haskell-Programm für den in [34] auf Seite 113 angegebenen Algorithmus

Kapitel 5

Codeerzeugung für eine imperative Sprache

5.1 Übersetzung von Ausdrücken

Nachdem wir die wichtigsten Hilfsmittel zum Entwurf eines Compilers kennengelernt haben, soll jetzt ein Übersetzer für eine imperative Programmiersprache im einzelnen definiert werden. Ein großer Vorteil des konstruktorbasierten algebraischen Zugangs besteht darin, dass er für die einzelnen Konstrukte der Quellsprache getrennt entwickeln werden kann. Gemeinsame Attribute bilden dabei die Schnittstelle zwischen den Teilübersetzern.

Wir beginnen mit den Ausdrücken einer imperativen Sprache:

konkrete Syntax			*abstrakte Syntax*	
exp	→	Int	Ci:	Int → exp
		| Bool	Cb:	Bool → exp
		| String	Id:	String → exp
		| **fun** String	Fid:	String → exp
		| String(actList)	Apply:	String actList → exp
		| exp^	Deref	exp → exp
		| ¬exp	Not:	exp exp
		| exp ≤ exp	Leq:	exp exp → exp
		| exp + exp	Add:	exp exp → exp
		| exp - exp	Sub:	exp exp → exp
		| exp * exp	Mul:	exp exp → exp
		| exp[exp]	Af:	exp exp → exp
		| exp. String	Rf:	exp String → exp
actList	→	exp^*	Actuals:	exp^* → actList

Das wichtigste Attribut bei der Übersetzung einer imperativen Sprache ist die **Symboltabelle.** Wir stellen sie dar als Funktion

$$st: \ String \ \rightarrow \ TypeDesc \times Int \times Int,$$

die jedem Identifier einen **Typdeskriptor**, eine **relative Adresse** und die **Schachtelungstiefe** seiner Deklaration zuordnet. Der Typdeskriptor bestimmt u.a. die Anzahl der Kellerplätze, die für einen Wert des jeweiligen

Typs reserviert werden müssen. Typdeskriptoren werden durch folgenden Datentyp implementiert, dessen Konstruktoren übersetzungsrelevante Informationen über die Elemente des jeweiligen Typs tragen:

```
data TypeDesc = INT | BOOL | Pointer TypeDesc |
                Func Int TypeDesc | Formalfunc TypeDesc |
                Array Int Int TypeDesc | Record (String -> (TypeDesc,Int,Int)) |
                Name String
```

Das Argument von *Pointer* ist der Deskriptor des Typs der Elemente, auf die die Elemente des jeweiligen Zeigertyps verweisen. *Func* liefert Deskriptoren funktionaler Typen. Die Argumente von *Func* sind die **Codeadresse** der jeweiligen Funktion und der Deskriptor des Resultattyps der Funktion. *Formalfunc* liefert den Deskriptor eines formalen Funktionsparameters. Das Argument von *Formalfunc* entspricht dem zweiten Argument von *Func*. Da der Code eines formalen Funktionsparameters zur Übersetzungszeit nicht bekannt ist, entfällt hier das erste Argument von *Func*.

Name liefert den Deskriptor neu deklarierter Typen. Das Stringargument von *Name* ist der jeweilige Typidentifier x. Den Deskriptor des mit x benannten Typs findet man unter x in der Symboltabelle. Die Argumente des Feldtypkonstruktors *Array* sind die untere Schranke, die Länge und der Elementtypdeskriptor des jeweiligen Feldes. (Die Quellsprache erlaubt also nur konstante Feldgrenzen.) Das Argument von *Record* ist eine ganze Symboltabelle, die den Attributen des jeweiligen Records deren Typdeskriptoren und relative Adressen zuordnet. Die Funktion *offset* berechnet den Platzbedarf eines Elementes des jeweiligen Typs. Da Typidentifier in der Symboltabelle gehalten werden, hängt *offset* im Falle des *Name*-Konstruktors von jener ab.

```
offset (Func _ _ ) _                = 3
offset (Formalfunc _) _             = 3
offset (Array _ lg td) st           = lg * offset td st
offset (Record []) _                = 0
offset (Record ((_,td,_,_):st)) st' = offset (Record st) st' + offset td st'
offset (Name x) st                  = offset td st where (td,_,_) = st x
offset _ _                          = 1
```

Wir stellen noch eine Funktion *substType* zur Verfügung, die Typidentifier expandiert:

```
substType (Name x) st = td where (td,_,_) = st x
substType td _        = td
```

Die Übersetzung von Ausdrücken benötigt neben *TypeDesc* (s.o.) die folgenden Attribut(bereich)e:

```
type Symtab = Id -> (TypeDesc,Int,Int)
type Depth = Int
type Label = Int
type Code = [Command]
type Offset = Int
```

`Symtab` liefert die jeweils aktuelle Symboltabelle, `Depth` die Schachtelungstiefe des aktuellen Scopes, also des innersten Blocks, in dem der gerade übersetzte Ausdruck auftritt, `Label` die nächste freie Befehlsnummer, `Code` den Zielcode und `Offset` den Platzbedarf für eine Liste aktueller Parameter.

Wie in §3.2 wollen wir Zielsprache und Übersetzer als Algebra darstellen, genauer gesagt: als Objekt einer Instanz eines Datentyps, der beliebige Interpretationen der abstrakten Syntax unserer imperativen Quellsprache zulässt. Hier ist zunächst nur der anweisungsfreie Teil der Sprache:

```
data ImpAlg exp acts ... = ImpAlg {ci :: Int -> exp, cb :: Bool -> exp,
                                   id :: String -> exp, fid :: String -> exp,
                                   apply :: String -> actList -> exp,
                                   deref :: exp -> exp, not_ :: exp -> exp,
                                   leq, add, sub, mul :: exp -> exp -> exp,
                                   af :: exp -> exp -> exp, rf :: exp -> String -> exp,
                                   actuals :: [exp] -> acts, ...}
```

Da die Zielprogramme Listen von Assemblerbefehlen sein sollen, nennen wir die Zielalgebra `assemblyAlg`. Sie ist ein Objekt von `ImpAlg`:

```
assemblyAlg :: ImpAlg (Symtab -> Depth -> Label -> (Code,TypeDesc))    Datenbereich für exp
                      (Symtab -> Depth -> Label -> (Code,Offset))      Datenbereich für acts
                      ...
assemblyAlg = ImpAlg ci cb id fid apply deref af rf not_ leq add sub mul actuals ...
              where (siehe 5.1.1 bis 5.1.4)
```

Die Befehle der Zielsprache sind durch folgende abstrakte Syntax gegeben (siehe [1], Seite 519 ff.):

```
data  register = ACC                  | -- accumulator
                 BA                   | -- base address
                 TOP                  | -- stack top
                 STP                  | -- static predecessor
                 BIT                  | -- 0 or 1
                 HEAPTOP                -- heap top

data  Address  = REG Register         | -- c(Register) =  contents of Register
                 IND Register         | -- c(c(Register))
                 DEX Int Register       -- c(Int + c(Register))

data  Source   = ADR Address          | -- c(source) = c(Address)
                 CON Int                -- c(Source) = int

data  Command  = MOV Source Address   | -- c(Source) to  Address
                 ADD Address Source   | -- c(Address) +  c(Source)
                                           to Address
                 SUB Address Source   |
                 MUL Address Source   |
                 INC Address          | -- c(Address) + 1 to Address
                 DEC Address          | -- c(Address) - 1 to Address
                 LE Source Source     | -- if c(Source1) <= c(Source2)
                                         -- then 1 to BIT else 0 to BIT
                 INV Address          | -- not(c(Address)) to Address
                 GOTO Source          | -- goto Source
                 CJT Source           | -- if c(BIT) = 1 then goto Source
                 CJF Source           | -- if c(BIT) = 0 then goto Source
                 READ Address         | -- read into Address
                 WRITE Source         | -- write c(Source)
                 END
```

Die **Zielmaschine** hat 6 Register und einen Speicher, der in einen Keller und einen Heap aufgeteilt ist. Die Speicheradressen können direkt (REG), indirekt (IND) oder indiziert (DEX) angesprochen werden. Wir stellen folgende Makros zur Verfügung:

```
push = INC(REG(TOP))
pop = DEC(REG(TOP))

pushL n = replicate n push

popL n = replicate n pop

pushHeap n = replicate n (DEC (REG HEAPTOP))

loadL n r = foldl f [push,MOV (ADR (IND r)) (IND TOP)] [1..n]
            where f code i = code++[push,MOV (ADR (DEX i r)) (IND TOP)]

storeL n r = foldl f [push,MOV (ADR (IND TOP)) (IND r)] [1..n]
             where f code i = code++[push,MOV (ADR (IND TOP)) (DEX i r)]

readL n r = foldl f [READ (IND r)] [1..n]
            where f code i = code++[READ (DEX i r)]

writeL n = replicate n (WRITE (ADR (IND TOP)))

derefpointer(offset) = MOV (ADR (IND TOP)) (REG ACC):pop:loadL (offset-1) ACC

baseAddress declDepth depth = ADR (if declDepth == depth then REG BA
                                                         else DEX declDepth BA)
```

`baseAddress` berechnet die Anfangs- oder Basisadresse ba des Kellerbereichs des Scopes, in dem ein Identifier x gültig ist. Seine absolute Adresse ergibt sich dann als Summe von ba und der Relativadresse von x, die in der Symboltabelle gespeichert ist. Stimmen die Scopes der Deklaration und der Applikation von x überein (declDepth = depth), dann entspricht ba der aktuellen, im Register BA gehaltenen Basisadresse. Andernfalls umfaßt der Scope der Deklaration von id den der Applikation (declDepth < depth) und ba ergibt sich als Inhalt der auf die aktuelle Basisadresse folgenden declDepth-ten Speicherzelle). Der einem Scope zugeordnete Kellerbereich beginnt nämlich mit dem **Display** des Scopes, das sind die nach aufsteigender Tiefe geordneten Basisadressen aller diesen Scope umfassenden Gültigkeitsbereiche (siehe Fig. 5.1).

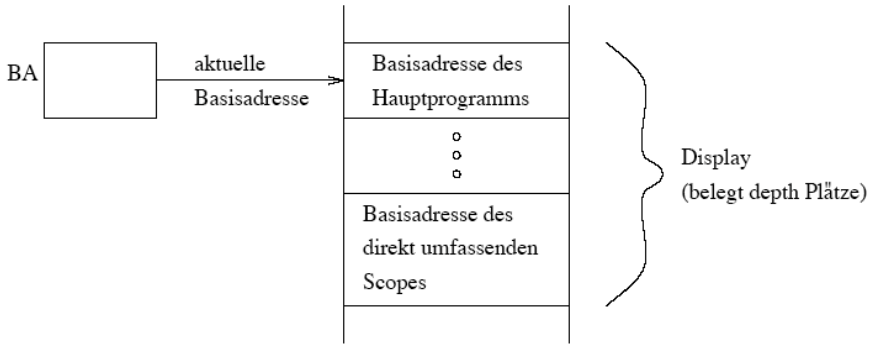


Figure 5.1. *Kellerauszug mit Display*

Ein mit dem Compiler $eval_{exp}^{assemblyAlg}$ verträglicher Interpreter $evalExp$ (vgl. 1.1) würde x abhängig vom jeweiligen Speicherzustand die absolute Adresse von x als Wert zuordnen. Insgesamt lauten die Komponenten des Diagramms aus Abschnitt 1.1 hier wie folgt:

Seien Val die Menge der speicherbaren Werte, $S = [Adr \to Val]$ die Menge der Speicherinhalte und $eval$ eine adäquate Fortsetzung des Interpreters der Zielsprache `[Command]` auf $assemblyAlg_{exp}$ (s.o.).

$$\begin{array}{ccc} T_{\Sigma(G),exp} & \xrightarrow{eval_{exp}^{assemblyAlg}} & assemblyAlg_{exp} \\ \Big\downarrow evalExp & & \Big\downarrow eval \\ [S \to Val] & \xrightarrow{encodeExp} & [S \to S] \end{array}$$

$assemblyAlg$ wird so definiert, dass für alle $e \in T_{\Sigma(G),exp}$, $f \in assemblyAlg_{exp}$, $s \in S$ und $x \in Adr$ gilt:

$$eval^{assemblyAlg}(e) = f \quad \overset{(*)}{\Rightarrow} \quad eval(f)(s)(x) = \begin{cases} evalExp(e)(s) & \text{falls } x = \text{IND(TOP)} \\ s(x) & \text{sonst.} \end{cases}$$

Definiert man $encodeExp$ für alle $f \in [S \to Val]$ durch

$$encodeExp(f)(s)(x) = \begin{cases} f(s) & \text{falls } x = \text{IND(TOP)} \\ s(x) & \text{sonst,} \end{cases}$$

dann entspricht die Gültigkeit von $(*)$ der Kommutativität des obigen Diagramms. $(*)$ ist natürlich keine vollständige Definition der Funktionen des Diagramms, sondern formalisiert lediglich Anforderungen an die Übersetzung von Ausdrücken:

- Jede Ausführung des Zielprogramms für den Ausdruck e endet mit dem Schreiben des Wertes von e auf den Keller (Adresse IND(TOP)).
- Außer an dieser einen Stelle bleibt der Speicherinhalt (Keller, Heap, etc.) von der Ausführung des Zielprogramms für e unberührt.[1]

5.1.1 Applikationen

Konstanten

```
ci i _ _ _ = ([push, MOV (CON i) (IND TOP)], INT)

cb b _ _ _ = ([push, MOV (CON (if b then 1 else 0)) (IND TOP)], BOOL)
```

`push` erhöht den Stacktop. `MOV` kellert die Konstanten.

[1] Wenn er sich während der Ausführung ändert, muss er am Ende wiederhergestellt sein.

Einfache Identifier

```
id x st depth _ = ([push, MOV ba (IND TOP), ADD (IND TOP) (CON adr)], Pointer td)
                  where (td,adr,declDepth) = st x
                        ba = baseAddress declDepth depth
```

In der Symboltabelle *st* steht unter x die Relativadresse *adr* und die Schachtelungstiefe declDepth des Scopes der Deklaration von x. Aus *declDepth* und der Tiefe k des aktuellen Scopes wird die Basisadresse *ba* des Gültigkeitsbereiches von x berechnet (s.o.). *ba* ist Parameter des Zielcodes, der durch das Kellern von *ba* und anschließende Addieren der Relativadresse *adr* die absolute Adresse von x berechnet und kellert. Deshalb liefert *id* den Deskriptor des Typs von Zeigern auf Objekte des Typs von x.

Funktionsidentifier

```
fid f st depth _ = (cs,Func codeadr td)
                   where ba = baseAddress declDepth depth
                         (Func codeadr td,resadr,declDepth) = st f
                         cs = [push, MOV ba (IND TOP),
                               push, MOV (CON codeadr) (IND TOP),
                               push, MOV ba (IND TOP), ADD (IND TOP) (CON resadr)]
```

Diese Applikation des Identifiers f einer Funktion tritt nur in Listen aktueller Parameter auf, z.B. in $g(\textbf{fun}\ f)$. *cs* speichert drei Adressen, die später als Wert von der Applikation aufgefasst werden.

- `ba` = Basisadresse des **statischen Vorgängers** von f, das ist der Kellerbereich des Scopes, in dem f *deklariert* wurde,[2]
- `codeadr` = Codeadresse von f,
- `ba+resadr` = absolute Adresse des Resultates eines Aufrufs von f.

Nach der Ausführung von *cs* ist $ba + resadr$ der Inhalt von IND(TOP).

5.1.2 Funktionsaufrufe

Im Programm deklarierte Funktionen

```
apply f es st depth lab
      = case ftd of Func codeadr td -> (cs++applyFunc codeadr offset resadr ba next,
                                        Pointer td)
                    Formalfunc td -> (cs++applyFormalfunc offset resadr ba next,
                                      Pointer td)
        where (cs,paroffset) = actuals es st depth lab
              (ftd,resadr,declDepth) = st f
              ba = baseAddress declDepth depth
              next = lab+length cs
```

[2]Der **dynamische Vorgänger** ist demgegenüber der Kellerbereich des Scopes, in dem ein Identifier *benutzt* wird. So wie die Deklarationstiefe niemals größer als die Aufruftiefe ist, so ist der statische Vorgänger niemals ein Nachfolger des dynamischen Vorgängers!

Zunächst wird die Parameterliste *es* übersetzt. Der Zielcode bewirkt das Kellern der Parameterwerte. Das abgeleitete Attribut *paroffset* liefert ihren Platzbedarf (s.o.). *ba* ist wieder die Basisadresse des statischen Vorgängers von f. Dann wird abhängig vom Typdeskriptor von f eine der Hilfsfunktionen *applyFunc* oder *applyFormalfunc* aufgerufen. Diese liefert den restlichen Zielcode. Beide möglichen Typdeskriptoren von f enthalten den Typdeskriptor *td* des Resultates von f. Ein entsprechender Zeigertyp liefert den Typdeskriptor *Pointer(td)* des Funktionsaufrufes.

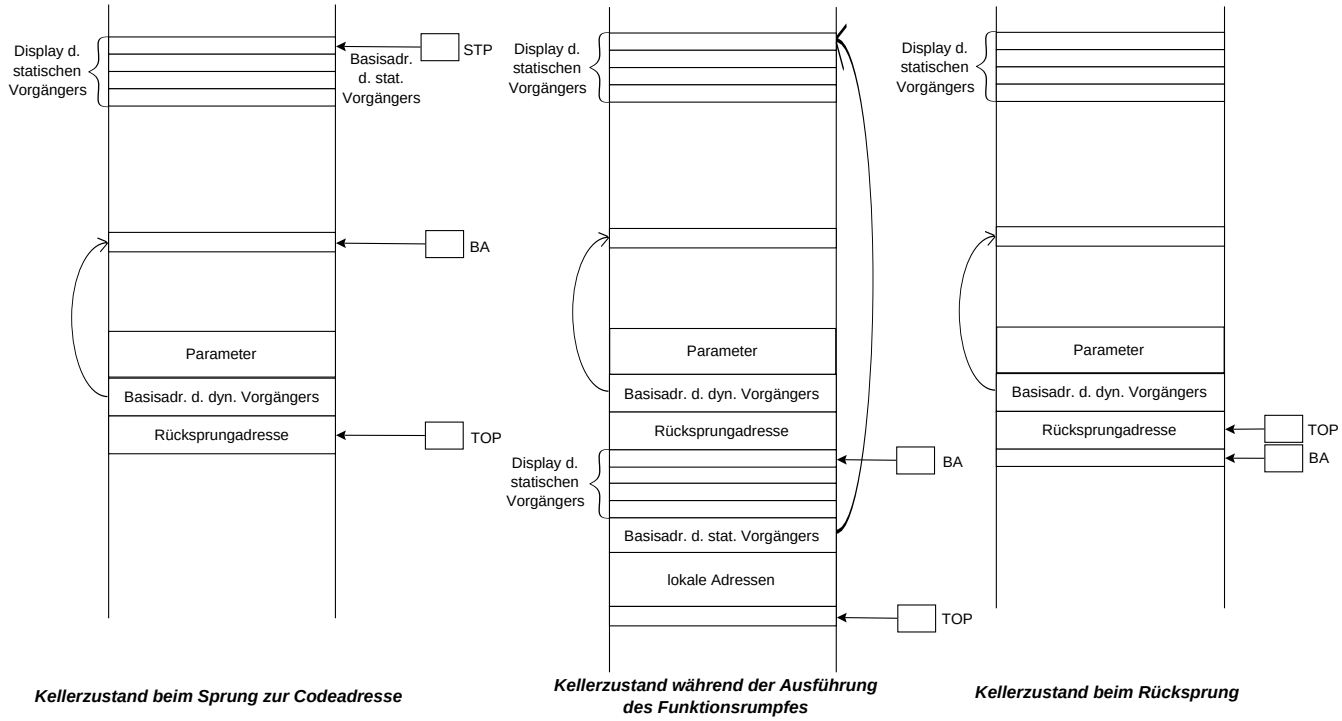


Figure 5.2. *Auf- und Abbau des Kellers bei der Ausführung eines Funktionsaufrufs*

Fall 1: f hat den Typdeskriptor `Func codeadr td`. Dann liefert `applyFunc` den restlichen Zielcode:

```
applyFunc codeadr paroffset resadr ba lab
      = push :
        MOV (ADR (REG BA)) (IND TOP) :   Kellern der aktuellen Basisadresse
        MOV ba (REG STP) :               Laden von STP mit der Basisadresse ba
        push :                           des statischen Vorgängers von f
        MOV (CON retadr) (IND TOP) :     Kellern der Rücksprungadresse retadr
        GOTO (CON codeadr) :             Sprung zur Codeadresse von f
retadr: pop :                            Entkellern der Rücksprungadresse[3]
        MOV (ADR (IND TOP)) (REG BA) :   Laden von BA mit der alten Basisadresse
        popL paroffset ++                Entkellern der aktuellen Parameter
        [MOV ba (IND TOP),               Berechnen und Kellern der absoluten Adresse
         ADD (IND TOP) (CON resadr)]     des Resultates von f(es)
        where retadr = lab+6
```

Die Codeadresse *codeadr* gehört zum Code der Deklaration von f. Bevor dieser mit einem Sprung zur Rücksprungadresse *retadr* endet, stellt er den Kellerzustand wieder her, der vor dem Sprung zur Codeadresse bestanden hat.

[3] Entkellern = Zurücksetzen des TOP-Zeigers

Fall 2: f hat den Typdeskriptor `Formalfunc td`, d.h. f ist ein formaler Funktionsparameter. Dann liefert `applyFormalfunc` den restlichen Zielcode:

```
applyFormalfunc paroffset resadr ba lab
       = push :
         MOV (ADR (REG BA)) (IND TOP) :        Kellern der aktuellen Basisadresse
         MOV ba (REG ACC) :                    Berechnen und Laden von STP mit der Basisadresse
         MOV (ADR (DEX resadr ACC)) (REG STP) :   des statischen Vorgängers von h (s.u.)
         push :
         MOV (CON retadr) (IND TOP) :          Kellern der Rücksprungadresse retadr
         GOTO (ADR (DEX (resadr+1) ACC)) :     Sprung zur Codeadresse von h
retadr:  pop :                                 Entkellern der Rücksprungadresse
         MOV (ADR (IND TOP)) (REG BA) :        Laden von BA mit der alten Basisadresse
         popL paroffset ++                     Entkellern der aktuellen Parameter
         [MOV ba (REG ACC),
          MOV (ADR (DEX (resadr+2) ACC)) (IND TOP)]   Berechnen und Kellern der absoluten
                                                      Adresse des Resultates des aktuellen Aufrufs von h
         where retadr = lab+7
```

f ist hier ein formaler Parameter, d.h. der Aufruf $f(es)$ steht im Rumpf einer Funktion g mit Parameter f:

$$\textbf{fun}\ g(\ldots, \textbf{fun}\ f, \ldots)\ \textbf{is} \ldots f(es) \ldots \textbf{end}$$

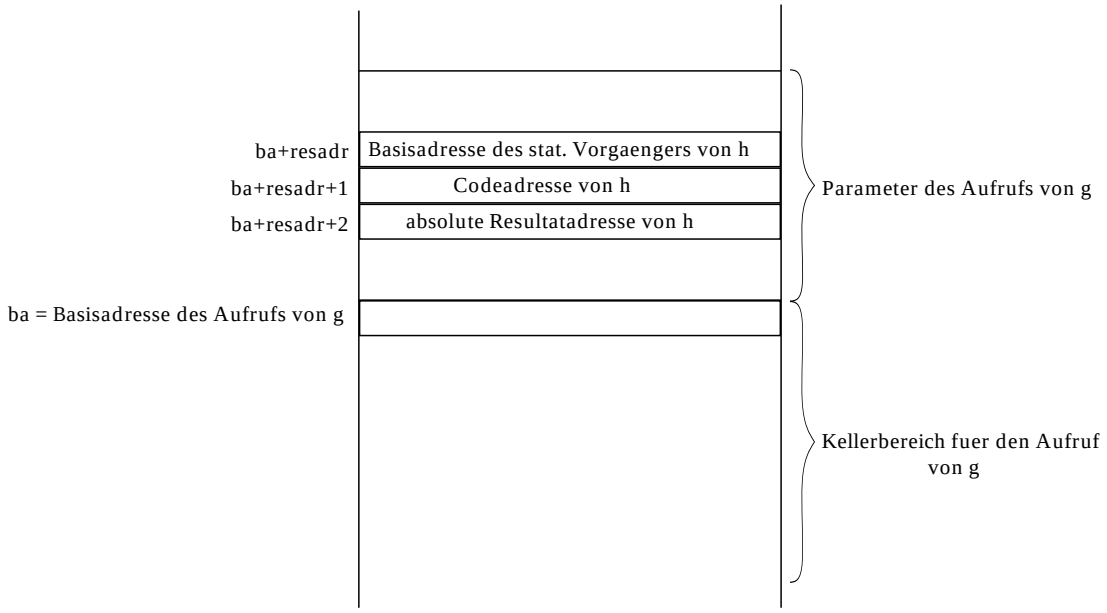


Figure 5.3. *Zur Adressrechnung bei der Übersetzung von f(es), falls f ein formaler Parameter ist, der durch h aktualisiert wurde*

Der statische Vorgänger von f ist also zur Übersetzungszeit gar nicht bekannt! Der Symboltabelleneintrag *(formalfunc(td),resadr,declDepth)* von f entspricht dem eines einfachen Identifiers (s.o.): *resadr* = Relativadresse, *declDepth* = Deklarationstiefe. Letztere entspricht der Deklarationstiefe von g, während sich die Relativadresse auf den bei Aufrufen von g für f reservierten Parameterplatz bezieht (siehe §5.3).

Der Aufruf $f(es)$ kann nur innerhalb eines Aufrufes von g erfolgen. Dabei wird f durch einen Funktionsidentifier h aktualisiert (siehe Fig. 5.3):

$$g(\ldots, \textbf{fun}\ h, \ldots)$$

Nach Abschnitt 5.1.3 besteht der Wert von **fun** h aus drei Komponenten: der Basisadresse des statischen Vorgängers, der Codeadresse und der absoluten Resultatadresse von h. ba ist hier die Basisadresse des Kellerbereichs für den Aufruf von g. Da ba im Akkumulator steht (s.o.), erhalten wir die Basisadresse des statischen Vorgängers von h als Summe des Inhalts des Akkumulators und der Relativadresse *resadr* von h (`DEX resadr ACC`). Die Codeadresse von h entspricht der zweiten Komponente (`DEX (resadr+1) ACC`) des Wertes von **fun** h. Nach dem Rücksprung geht's weiter wie im Fall 1. Die absolute Adresse für das Resultat des aktuellen Aufrufs von h entspricht der dritten Komponente (`DEX (resadr+2) ACC`) des Wertes von `fun h`.

Dereferenzierung

Zeigerargumente (z.B. Objektidentifier; s.o.) werden vor Anwendung einer Booleschen oder arithmetischen Funktion dereferenziert:

```
deref e st depth lab = case td of Pointer td -> (cs++derefpointer (offset td st),td)
                                  _ -> (cs,td)
                       where (cs,td) = e st depth lab
```

Boolesche Funktionen

```
not_ e st depth lab = (cs++[INV (IND TOP)], BOOL) where (cs, BOOL) = deref e st depth lab
```

```
leq e e' st depth lab = (cs1++cs2++cs3, BOOL)
                         where (cs1,_) = deref e st depth lab
                               (cs2,_) = deref e' st depth (lab+length cs1)
                               cs3 = [pop, LE (ADR (IND TOP)) (ADR (DEX 1 TOP)),
                                      MOV (ADR (REG BIT)) (IND TOP)]
```

Die Reihenfolge, in der e und e' übersetzt werden, bestimmt die Anordnung der entsprechenden Laufzeitwerte auf dem Stack: oben steht der Wert von e', darunter der von e. Also wird durch ein pop der TOP-Zeiger auf den Wert von e gesetzt, dann der Vergleich durchgeführt und schließlich das Ergebnis (BIT=0 oder BIT=1) gekellert.

Operationen auf ganzen Zahlen

Sei $op \in \{add, sub, mul\}$.

```
op e e' st depth lab = (cs1++cs2++cs3,INT))
                       where (cs1, INT) = deref e st depth lab
                             (cs2, INT) = deref e' st depth (Label (lab+length cs1))
                             cs3 = [pop,OP (IND TOP) (ADR (DEX 1 TOP))]
```

Assemblersprachen stellen heute Mehradreßbefehle und viele Register zur Verfügung, damit der Keller nicht zur Auswertung von Ausdrücken benutzt werden muß. Das spart Laufzeit, erhöht aber den Übersetzungsaufwand, denn dann stellt sich das Problem der Zuteilung von Registern an die Werte von (Teil-)Ausdrücken (siehe §9.4).

5.1.3 Zugriffe

e[e']: Feld e greift auf Index e' zu

```
af e e' st st' depth lab = (cs1++cs2++cs3,Pointer td')
                  where (cs1, Pointer td) = e st depth lab
                        Array lwb _ td' = substType td st
                        offset = offset td' st'
                        (cs2, INT) = deref e' st depth (lab+length cs1)
                        cs3 = [SUB (IND TOP) (CON lwb),          Wert(e')-lwb
                               MUL (IND TOP) (CON offset), pop,  (Wert(e')-lwb)*offset
                               ADD (IND TOP) (ADR (DEX 1 TOP))]  Adr(e)+(Wert(e')-lwb)*offset
```

$cs1$ kellert die Anfangsadresse a des Feldes e. Darüber kellert $cs2$ den Wert i des Feldindexes e'. Der restliche Zielcode berechnet zuerst die relative Adresse *adr = (i-lwb)*offset* und dann die absolute Adresse $a + adr$ von *e[e']*.

e.x: Record e greift auf Feld x zu

```
rf e x st depth lab = (cs++[ADD (IND TOP) (CON adr)], Pointer td)
                      where (cs, Pointer td) = e st depth lab
                                   Record st' = substType td st
                                   (td,adr,_) = st' x
```

$cs1$ kellert die Anfangsadresse a des Records e. Der restliche Zielcode addiert die relative Adresse adr des Attributes x zu a. $a + adr$ ist die absolute Adresse von $e.x$.

5.1.4 Aktuelle Parameter

```
actuals (e:es) st depth lab = (cs1++cs2,offsets+offset td st)
                              where (cs1,offsets) = actuals es st depth lab
                                    (cs2,td) = deref e st depth (lab+length cs1)
actuals _ _ _ _             = ([],0)
```

5.2 Übersetzung von Anweisungen

konkrete Syntax			*abstrakte Syntax*	
prog	→	stat	MkProg:	stat → prog
stat	→	ε	Skip:	stat
		| exp := exp	Assign:	exp exp → stat
		| **read** String	Read:	String → stat
		| **write** exp	Write:	exp → stat
		| stat; stat	Seq:	stat stat → stat
		| **if** exp **then** stat **else** stat **fi**	Cond:	exp stat stat → stat
		| **while** exp **do** stat **od**	Loop:	exp stat → stat
		| **type** String = type	Typevar:	String type → stat
		| **var** String: type	Var:	String type → stat
		| **new** String	New:	String → stat
		| **begin** stat **end**	Block:	stat → stat
		| **fun** String(parList):type **is** stat **end**	Fun:	String parList type stat → stat
par	→	String:type	For:	String type → par
		| **fun** String:type	Funfor:	String type → par
parList	→	par*	Formals:	par* → parList
type	→	**Int**	INT:	type
		| **Bool**	BOOL:	type
		| ^type	POINTER:	type → type
		| **array** [Int..Int] type	ARRAY:	Int Int type → type
		| **record** stat	RECORD:	stat → type
		| String	Name:	String → type

Auch Deklarationen werden hier als Anweisungen (*Statements*) betrachtet. Die Übersetzung von Anweisungen benötigt neben den Attributen aus §5.1 die nächste freie Relativadresse:

```
type Reladr = Int
```

`Depth` liefert hier die Schachtelungstiefe des innersten Blocks, in dem die gerade übersetzte Anweisung auftritt.

Aus der abstrakten Syntax für Anweisungen und Typen unserer imperativen Sprache ergeben sich weitere Typvariablen sowie Felder für Interpretationen von Konstruktoren des Datentyps *ImpAlg* (siehe §5.1):

```
data ImpAlg ... prog stat par parList type
          = ImpAlg {..., mkProg: stat -> prog,
                    skip: stat, assign: exp -> exp -> stat, read: String -> stat,
                    write: exp -> stat, seq: stat -> stat -> stat,
                    cond: exp -> stat -> stat -> stat, loop: exp -> stat -> stat,
                    typedef: String -> type -> stat, var: String -> type -> stat,
                    new: String -> stat, block: stat -> stat,
                    fun: String -> parList -> type -> stat -> stat,
                    for, funfor: String -> type -> par, formals: [par] -> parList,
                    int, bool: type, pointer: type -> type,
                    array: Int -> Int -> type, record: stat -> type,
```

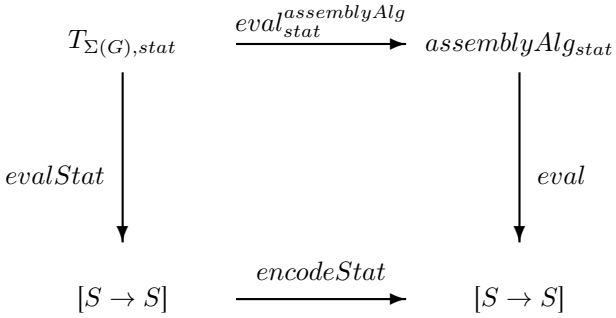
```
                    name: String -> type}
```

`assemblyAlg` wird um passende Interpretationen erweitert:

```
assemblyAlg :: ImpAlg
    Code                                                          Datenbereich für prog
    (Symtab -> Depth -> Reladr -> Label -> (Code,Symtab,Reladr))  Datenbereich für stat
    (Symtab -> Depth -> Reladr -> (Symtab,Reladr))                Datenbereich für par
    (Symtab -> Depth -> Reladr -> Symtab)                         Datenbereich für parList
    TypeDesc                                                      Datenbereich für type
 assemblyAlg = ImpAlg ... mkProg skip assign read write seq cond loop typedef var new
                      block fun for funfor int bool pointer array record name
               where (siehe 5.1.1 bis 5.1.4)
                     mkProg stat = code where (code,_,_) = stat (const (INT,0,0)) 0 0 0
                     (siehe 5.2.1 bis 5.2.3)
```

Während die Symboltabelle bei $assemblyAlg_{exp}$ nur vererbt wird, ist sie hier zugleich vererbtes und abgeleitetes Attribut: Sie wird gelesen und bei der Übersetzung von Deklararationen auch verändert.

Das dem Diagramm aus §5.1 entsprechende Diagramm für die Übersetzung von Anweisungen sieht folgendermaßen aus. Seien *Val* wieder die Menge der speicherbaren Werte, $S = [Adr \to Val]$ die Menge der Speicherinhalte und *eval* eine adäquate Fortsetzung des Interpreters der Zielsprache `[Command]` auf $assemblyAlg_{stat}$.



assemblyAlg wird so definiert, dass für alle $st \in T_{\Sigma(G),stat}$ und $f \in assemblyAlg_{stat}$ gilt:

$$eval_{stat}^{assemblyAlg}(st) = f \quad \overset{(*)}{\Rightarrow} \quad eval(f) = evalStat(st).$$

Definiert man *encodeStat* als Identität auf $[S \to S]$, dann entspricht die Gültigkeit von $(*)$ der Kommutativität des obigen Diagramms. Da im Gegensatz zur Semantik von Ausdrücken (siehe §5.1.2) die abstrakten Maschinen (siehe Def. 1.1.1) von Anweisungen in Quell- und Zielsprache gleich sind, nämlich Funktionen von S nach S, ist diese Wahl von *encodeStat* naheliegend.

5.2.1 Einfache Kommandos

```
skip st _ adr lab = ([],st,adr)
```

```
assign e e' st depth adr lab = (cs1++cs2++cs3),st,adr)
                               where (cs1,Pointer _) = e st depth lab
                                     (cs2,td) = deref e' st depth (lab+length cs1)
```

```
                                    offset = offset td st
                                    cs3 = popL offset ++ MOV (ADR (IND TOP)) (REG ACC) :
                                          storeL (offset-1) ACC ++ popL (offset+1)
```

Bei der Übersetzung einer Zuweisung $e := e'$ werden zunächst die Ausdrücke e und e' compiliert. Da e' auf eine niedrigere Referenzstufe als e gehört, wird e' zunächst dereferenziert. Der restliche Zielcode bewirkt das

- Entkellern des Wertes von *deref*(e'),
- Laden von ACC mit dem (Adress-) Wert von e,
- Speichern des Wertes von *deref*(e') in den mit der Adresse von e beginnenden Kellerbereich,
- Entkellern des Wertes von *deref*(e') und der Adresse von e.

```
read x st depth adr _ = (cs,st,adr)
                        where cs = MOV ba (REG ACC) : ADD (REG ACC) (CON idadr) :
                                   readL (offset td st-1) ACC
                              (td,idadr,declDepth) = st x
                              ba = baseAddress declDepth depth
```

cs bewirkt das

- Laden von ACC mit der Basisadresse des Scopes des Lesebefehls,
- Addieren der Relativadresse *idadr* von x zur Berechnung der absoluten Adresse von x,
- Einlesen des Wertes von x in den mit der absoluten Adresse von x beginnenden Kellerbereich.

```
write e st depth adr lab = (cs1++cs2,st,adr)
                           where (cs1,td) = deref e st depth lab
                                 offset = offset td st
                                 cs2 = popL offset ++ writeL offset ++ popL offset
```

*cs*1 dereferenziert und übersetzt den Ausdruck e. *cs*2 gibt den Wert von *deref*(e) aus und entkellert ihn dann.

5.2.2 Zusammengesetzte Kommandos

```
seq s s' st depth adr lab = (cs1++cs2,st2,adr2)
                            where (cs1,st1,adr1) = s st depth adr lab
                                  (cs2,st2,adr2) = s' st1 depth adr1 (lab+length cs1)

cond e s s' st depth adr lab = (cs1++cs4,st2,adr2)
                               where (cs1, BOOL) = deref e st depth lab
                                     thenLab = lab+length cs1+3
                                     (cs2,st1,adr1) = s st depth adr thenLab
                                     elseLab = thenLab+length cs2+1
                                     (cs3,st2,adr2) = s' st1 depth adr1 elseLab
                                     exit = elseLab+length cs3
                                     cs4 = MOV (ADR (IND TOP)) (REG BIT) : pop :
                                           CJF (CON elseLab) :
                               thenLab:  cs2 ++ GOTO (CON exit) :
                               elseLab:  cs3
                               exit:
```

$cs4$ kellert den Wert des Booleschen Ausdrucks e, transportiert ihn nach BIT, springt im false-Fall zur Marke $elseLab$, also hinter den Befehl `GOTO (CON exit)`, und führt den Code $cs3$ des else-Zweiges s' aus. Im true-Fall werden der Code $cs2$ des then-Zweiges ausgeführt und die Marke $exit$ hinter $cs3$ angesprungen.

```
loop e s st depth adr lab = (cs1++cs3,st',adr')
                            where (cs1, BOOL) = deref e st depth lab
                                  bodyLab = lab+length cs1+3
                                  (cs2,st',adr') = s st depth adr bodyLab
                                  exit = bodyLab+length cs2+1
                                  cs3  = MOV (ADR (IND TOP) (REG BIT) : pop :
                                         CJF (CON exit) :
                             bodyLab:  cs2 ++ [GOTO (CON lab)]
                             exit:
```

$cs3$ kellert den Wert des Booleschen Ausdrucks e, transportiert ihn nach BIT und springt im false-Fall zur Marke $exit$, also hinter den Befehl `GOTO (CON n)`. Im true-Fall werden der Code $cs2$ des Schleifenrumpfes von s ausgeführt und der erste Befehl des Codes $cs1$ zur Berechung von e angesprungen.

5.2.3 Deklarationen

```
typedef x t st _ adr _ = ([], update st x (type t,0,0), adr)
```

x wird mit dem Typdeskriptor von t in die Symboltabelle eingetragen. Der Zielcode ist leer. update ist in Beispiel 1.2.3 definiert.

```
var x t st depth adr _ = (pushL offset, update st x (td,adr,depth), adr+offset)
                         where td = type t
                               offset = offset td st
```

x wird mit dem Typdeskriptor td von t in die Symboltabelle eingetragen. Der Zielcode $pushL(offset)$ reserviert Platz für den Wert von x. $adr+offset$ liefert die nächste freie Relativadresse.

```
new x st depth adr lab = (cs,st,adr)
                         where (Pointer td,idadr,declDepth) = st x
                               offset = offset td st
                               ba = baseAddress declDepth depth
                               cs = pushHeap offset ++ [MOV ba (REG ACC),
                                                        ADD (REG ACC) (CON idadr),
                                                        MOV (ADR (REG HEAPTOP)) (IND ACC)]
```

Bei der Übersetzung einer Objektdeklaration **new** x muß der Typ von x ein Zeigertyp sein. Der entsprechende Deskriptor *Pointer(td)* hat als Argument den zugehörigen Objekttypdeskriptor. *ba* liefert die Basisadresse des Scopes der Deklaration von x.

pushHeap(offset) reserviert auf dem Heap Platz für das Objekt, auf das x zeigt. Der restliche Zielcode berechnet die absolute Adresse von x und legt den neuen Heaptop unter dieser Adresse ab. Das sieht zunächst so aus, als würde ein späterer Wert des deklarierten Objektes *hinter* dem dafür reservierten Bereich stehen. Das ist jedoch nicht der Fall, weil *pushHeap* den Heaptop *dekrementiert*, so dass der neue Heaptop tatsächlich den Anfang und nicht das Ende jenes Bereiches anzeigt. Stack und Heap sind nämlich i.a. Teile desselben

Speicherbereiches. Der Stacktop beginnt mit der kleinsten Adresse und wird dann hochgezählt, während der Heaptop mit der größten Adresse beginnt und dann heruntergezählt wird. Wir gehen an dieser Stelle nicht auf weitere Details der Implementierung verketteter Strukturen ein, weisen nur daraufhin, dass durch Anweisungen wie `x:=[]` oder `dispose(x)` das durch x referenzierte Objekt möglicherweise nicht mehr zugreifbar ist und dann der dafür reservierte Speicherplatz auf dem Heap freigegeben werden sollte. **Garbage-Collection** (d.h. Müllsammlungs-) Algorithmen erkennen die noch zugreifbaren Objekte auf dem Heap und kopieren sie von Zeit zu Zeit in einen freien zusammenhängenden Heapbereich.

Blockdeklaration

Man kann sie als Kombination der Deklaration und des Aufrufs einer Prozedur ohne Ein- und Ausgabeparameter ansehen. Dementsprechend enhält der Code einer Blockdeklaration Befehle des Codes einer Funktionsdeklaration wie auch eines Funktionsaufrufs (siehe §5.1.4). Da Deklaration und Aufruf beim Block zusammenfallen, gibt es hier auch keinen Unterschied zwischen statischem und dynamischen Vorgänger.

```
block stat st depth adr lab = (cs2,st,adr)
      where (cs1,_,locadr) = stat st (depth+1) (adr+1) (lab+2*depth+8)
            cs2 = push :
                  MOV (ADR (REG BA)) (IND TOP) :
                  MOV (ADR (REG BA)) (REG STP) : push :
                  MOV (ADR (REG TOP)) (REG BA) : pop ++
                  loadL (depth-1) STP ++          Display des umfassenden Blocks übernehmen
                  push:
                  MOV (ADR (REG STP)) (IND TOP) :  Display um statischen Vorgänger erweitern
                  cs1 ++ popL locadr ++ [MOV (ADR (IND TOP)) (REG BA), pop]
```

Der Zielcode $cs2$ der Deklaration eines Blockes mit Rumpf s bewirkt folgende Laufzeitaktionen:

- Kellern der aktuellen Basisadresse (= Basisadresse des umfassenden Scopes; an dieser Kellerposition steht beim Funktionsaufruf die Rücksprungadresse, vgl. Fig. 5.2),
- Laden von STP (Register für statischen Vorgänger) mit der aktuellen Basisadresse,
- Laden von BA (also Setzen der aktuellen Basisadresse) mit der nächsten freien Kelleradresse, die dann den Anfang des Kellerbereichs für den neuen Scope bildet,
- Kellern des Displays des umfassenden Scopes,
- Kellern der Basisadresse des umfassenden Scopes und damit Vervollständigen des Displays des neuen Scopes,
- Ausführen des Blockrumpfes `stat`,
- Entkellern der lokalen Adressen und des Displays des neuen Scopes (s.u.),
- Laden von BA mit der Basisadresse des umfassenden Scopes.

Die Attributwerte bei der Übersetzung des Blockrumpfes s ergeben sich wie folgt:

- Aus der Schachtelungstiefe `depth` wird die Schachtelungstiefe `depth+1`.
- Der Kellerbereich für den neuen Scope beginnt mit dem Anfang des Displays (s.o.). Das Display hat die Länge `depth+1`. Also ist `depth+1` die *erste freie Relativadresse* für lokale Variablen von `stat`.
- Die nächste freie Befehlsnummer `lab+2*depth+8` ergibt sich aus den acht Einzelbefehlen vor dem Code von `stat` und der Tatsache, dass der Ladebefehl `loadL (depth-1) STP` zum Kellern des alten Displays in jedem Iterationsschritt aus zwei Einzelbefehlen besteht.

- *locadr* liefert die nächste freie Relativadresse nach Übersetzung von `stat`. Da der Kellerbereich für den neuen Scope mit der Relativadresse 0 beginnt, wird mit *popL(locadr)* dieser gesamte Bereich (Display und lokale Adressen) entkellert.

Die nächste freie Relativadresse nach Ausführung des Blocks entspricht derjenigen davor: *adr* bleibt *adr*.

Funktionsdeklaration

```
fun f ps t stat st depth resadr lab = (cs2, Symtab st1, resadr+offset)
          where td = compType t
                offset = offset td st
                codeadr = lab+offset+1
                st1 = update st f (Func codeadr td,resadr,depth)          (*)
                st2 = formals ps st1 (depth+1) (-2)
                bodyLab = codeadr+2*k+5
                (cs1,_,locadr) = stat st2 (depth+1) (depth+1) bodyLab
                exit = bodyLab+length cs1+locadr+1
                cs2 = pushL offset ++ GOTO (CON exit) :
    codeadr:          push :
                      MOV (ADR (REG TOP)) (REG BA) : pop ++
                      loadL (depth-1) STP ++          Display des umfassenden Blocks übernehmen
                      push :
                      MOV (ADR (REG STP)) (IND TOP) : Display um statischen Vorgänger erweitern
    bodyLab:          cs1 ++ popL locadr ++
                      [GOTO (ADR (IND TOP))]          Rücksprung (siehe § 5.1.2)
    exit:
```

Der Zielcode *cs*2 der Deklaration **fun** `f(ps):t` **is** `stat` **end** besteht aus der Reservierung von Speicherplatz für das Ergebnis eines Aufrufs von f (`pushL(offset)`) und dem Code von `stat`, der vom Code eines Aufrufs von f angesprungen wird (zur Marke `codeadr`; siehe §5.1.4) und selbst mit einem Rücksprung zum Aufrufcode endet. Er entspricht im wesentlichen dem Code einer Blockdeklaration:

- Laden von BA mit der Anfangsadresse des Bereichs für den neuen Scope,
- Kellern des Displays des neuen Scopes,
- Ausführen des Funktionsrumpfes s,
- Entkellern der lokalen Adressen und des Displays des neuen Scopes.

Danach erfolgt der Rücksprung zum Aufrufcode. Wichtig ist, dass zu diesem Zeitpunkt der Kellerzustand vor Ansprung der Codeadresse wiederhergestellt ist (siehe Fig. 5.2). Nur dann ist sichergestellt, dass über den Stacktop tatsächlich die Rücksprungadresse `retadr` erreicht wird. Die Attributwerte ergeben sich wie bei der Übersetzung eines Blockrumpfes (s.o.). Die Marke `exit` entspricht der nächsten freien Befehlsnummer nach dem Deklarationscode.

Die nächste freie Relativadresse nach Ausführung der Deklaration von f entspricht der ersten freien Adresse hinter dem für die Parameter reservierten Platz: `resadr+offset`. Bei `resadr` steht zu Anfang der erste Parameters von f. Am Ende des Aufrufs von f geht `resadr` als Relativadresse in die Berechnung der absoluten Adresse des Funktionsergebnisses ein (siehe §5.1.4). Deshalb wird bei der Übersetzung der Deklaration von f `resadr` unter f in die Symboltabelle eingetragen (siehe (∗)).

Formale Parameter

$par : Symtab \to Depth \to Reladr \to (Symtab, Reladr)$ trägt einen formalen Parameter x einer Funktion f in die Symboltabelle ein. Die dabei berechnete Relativadresse bezieht sich auf die Basisadresse ba des Bereiches eines späteren Aufrufs von f. Da ba die Adresse des dritten Platzes hinter den Parametern ist (siehe mittlerer Kellerzustand in Fig. 5.2), ist -2 die Relativadresse des ersten Parameters und werden die Relativadressen weiterer Parameter durch Subtraktion ihres Offsets berechnet. Ist x ein *funktionaler* Parameter von f, dann werden drei Plätze für x reserviert: für die Basisadresse des statischen Vorgängers von x, die Codeadresse von x und die absolute Resultatadresse des aktuellen Aufrufs von x (siehe §5.1.1).

```
for x td st depth adr = (update st x (td,adr,depth),adr')
                        where adr' = adr -offset td st)

funfor x td st depth adr = (update st x (Formalfunc td,adr',depth),adr')
                           where adr' = adr-3

formals (par:pars) st depth adr = formals pars st' depth adr'
                                  where (st',adr') = par st depth adr
formals _ st _ _                = st
```

Typapplikationen

Die restlichen Funktionen von `assemblyAlg` erzeugen Typdeskriptoren (siehe §5.1):

```
int  = INT
bool = BOOL
pointer = Pointer
array lwb upb = Array lwb (upb-lwb+1)
record s = Record st where (_,st,_) = s (const (INT,0,0)) 0 0 0
name = Name
```

Kapitel 6

Transformation funktionaler Programme

Rekursion ist *das* Mittel zur Formulierung von Algorithmen, die eine verschachtelte Lösung von Teilaufgaben darstellen. Einerseits sind rekursive Algorithmen besonders leicht Korrektheits- und Aufwandsanalysen zugänglich. Andererseits arbeiten unsere Rechner i.a. nicht nach dem abstrakten Auswertungsmodell, in dem jede Berechnung eine Folge von Anwendungen der rekursive Funktionen definierenden Gleichungen ist. Stattdessen werden Maschinenprogramme ausgeführt, die jeden rekursiven Aufruf in eine Sequenz von Befehlen zerlegen. Tatsächlich ist es eine Hauptaufgabe jedes klassischen Übersetzers, Rekursion zu eliminieren. Üblicherweise werden dazu Keller verwendet. Das ist jedoch nicht notwendig, wenn es sich um einfache Formen der Rekursion handelt, insbesondere **repetitive Rekursion** oder **Iteration**, die nicht selbsteinbettend ist (siehe §3.1) und daher einer Programmschleife entspricht.

Zur Eliminierung von Rekursion ist die Methode der **Programmtransformation** geeignet. Allgemein wird damit eine Compile-Funktion $comp : Q \to Z$ in zwei Funktionen $transform : Q \to Q$ und $translate : Q \to Z$ zerlegt. Dabei $transform$ modifiziert die Struktur des Quellprogramms, bringt es z.B. von einer rekursiven in eine nichtrekursive Form, worauf $translate$ die eigentliche Übersetzung in die Zielsprache vornimmt, dabei die Struktur des Programms aber kaum noch ändert.

6.1 Akkumulatoren

Linear-rekursive Definitionen wie diejenige der Fakultätsfunktion:

```
fact 0 = 1
fact x = x*fact (x-1)
```

lassen sich häufig in eine iterative Form bringen:

```
factLoop x = factAcc x 1
factAcc 0 a = a
factAcc x a = factAcc (x-1) (a*x)
```

Man nennt $factAcc$ eine **Schleifenfunktion** und das im Vergleich zu $fact$ zusätzliche Argument a einen Akkumulator, weil in dieser Stelle die Werte der Fakultätsfunktion akkumuliert, d.h. angehäuft, also schrittweise berechnet werden. Charakteristisch für repetitive Rekursion ist die fehlende Einbettung rekursiver Aufrufe (siehe §3.1), d.h. es gibt keine Funktion, die auf das Ergebnis eines rekursiven Aufrufs angewendet wird. Tatsächlich lässt sich $factLoop$ sofort in eine imperative Version bringen:

```
a = 1;
while (x > 0) {a = a*x; x = x-1;};
fact1 = a
```

Die übliche Definition der Fibonacci-Folge ist *baumartig-rekursiv*:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Jeder Aufruf $fib(n)$ mit $n > 1$ erzeugt zwei rekursive Aufrufe, $fib(n-1)$ und $fib(n-2)$. Eine iterative Version von fib hat zwei Akkumulatoren x und y:

```
fibLoop n    = fibAcc n 0 1
fibAcc 0 x y = x
fibAcc n x y = fibAcc (n-1) y (x+y)
```

6.2 Keller

In Kapitel 5 haben wir beliebige rekursive Funktionsprozeduren einpässig in eine Assemblersprache übersetzen, die mit einem Keller arbeitet. Dass das prinzipiell geht, ergibt sich schon aus dem Zusammenhang zwischen kontextfreien Sprachen und Kellerautomaten (siehe GTI). Insbesondere bei der Auflösung nichtlinearer oder geschachtelter Rekursion wird der Keller dazu benutzt, um noch nicht ausgewertete rekursive Aufrufe zwischenzuspeichern. So entsteht eine iterative Version der ursprünglichen Funktion. Während ein Akkumulator nur Zwischen*werte* der Funktion aufnimmt, werden im Keller auch *Argumente* (Parameter) rekursiver Aufrufe der Funktion gespeichert. Wir demonstrieren das Prinzip an folgendem Schema einer baumartig-rekursiven Funktion $f : s_1 \to s_2$:

$$f\ x = \mathit{if}\ p\ x\ \mathit{then}\ g\ (x, f\ (in_1\ x), f\ (in_2\ x))\ \mathit{else}\ \mathit{out}\ x$$

Demnach haben die verwendeten Hilfsfunktionen folgende Typen: $g : s_3 \times s_2 \times s_2 \to s_2$, $in_1, in_2 : s_1 \to s_1$ und $out : s_1 \to s_2$. Ein iteratives Programm für f lautet wie folgt:

```
data StackElem s1 s2 = Arg s1 | Val s2

f :: s1 -> s2
f x = loop [Arg x]

loop :: [StackElem s1 s2] -> s2
loop (Arg x:s)             = if p x then loop (Arg (in1 x):Arg (in2 x):Arg x:s)
                                    else loop (Val (out x):s)
loop (Val y:Arg x:s)       = loop (Arg x:Val y:s)
loop (Val z:Val y:Arg x:s) = loop (Val (g (x,y,z)):s)
loop [Val y]               = y
```

Aufgabe Verallgemeinern Sie RECSTACK so, dass nicht nur *binär*baumartige, sondern auch andere baumartige Rekursionen eliminiert werden!

Um die Übersetzung imperativer Programme zu beschreiben, haben wir in Kapitel 5 die in Abschnitt 4.1 behandelte Methode der attributierten Übersetzung verwendet und ohne vorherige Transformation des abstrakten Quellprogramms sofort Zielcode erzeugt, der auf einem Keller arbeitet. Die zu Beginn dieses Abschnitts

erwähnte Zerlegung eines Compilers in $transform : Q \to Q$ und $translate : Q \to Z$ erleichtert—zumindest auf einer höheren Entwurfsebene—das Verständnis seiner Arbeitsweise oft erheblich. Programmtransformationen werden im Compilerbau in zunehmendem Umfang eingesetzt.

6.3 Continuations

Dies gilt insbesondere für funktionale Sprachen. Da bei ihnen die konkrete Syntax mit der abstrakten fast identisch ist und $transform$ immer auf der letzterer operiert, bieten sich solche Sprachen für die Zerlegung in $transform$ und $translate$ besonders an. Oft wird hier die zu übersetzende Funktion f nicht mit Akkumulatoren und Kellern, sondern mit **Continuations** versehen, die festlegen, wie die Werte von f weiterverarbeitet werden sollen. Formal: $f : A \longrightarrow B$ geht über in $f_{Cont} :: A \to (B \to C) \to C$ derart, dass für jede Funktion $c :: B \to C$, die Continuation, gilt: $f_{Cont}\ x\ c = c\ (f\ x)$, also insbesondere: $f_{Cont} x\ id = f\ x$.

Beispiel Fakultät (siehe §6.1)

```
fact1 x = factCont x id
factCont 0 c = c 1
factCont x c = factCont (x-1) (c . (x*))
```

Beispiel einer Berechnungsfolge von $fact1$:

```
fact1 3 = factCont 3 id
        = factCont 2 (3*)
        = factCont 1 ((3*) . (2*)) = factCont 1 (6*)
        = factCont 0 ((3*) . (2*) . (1*)) = factCont 0 (6*)
        = 6*1
        = 6
```

Beispiel Mergesort Die übliche nicht linear-rekursive Form des Mergesort-Algorithmus, der nach einer vorgebenen Ordnung r sortiert, lautet wie folgt:

```
sort r (x:y:s) = merge r (sort r (x:s1)) (sort r (y:s2))
                 where (s1,s2) = split s
sort _ s       = s

split (a:b:s) = (a:s1,b:s2) where (s1,s2) = split s
split s       = (s,[])

merge r (x:s1,y:s2) = if r (x,y) then x:merge r s1 (y:s2)
                                 else y:merge r (x:s1) s2
merge _ [] s        = s
merge _ s []        = s
```

Hier ist eine iterative Version mit Continuations:

```
sort1 r s = sortCont r s id

sortCont r (x:y:s) c = sortCont r (x:s1)
```

```
                                    (\a -> sortCont r (y:s2)
                                                      (c . merge r a))
                          where (s1,s2) = split s
sortCont _ s c       = c s
```

Beispiel einer Berechnungsfolge von $sort1$ mit $r(x, y) = (x \leq y)$:

```
sort1 r [5,1,3]
 = sortCont r [5,1,3] id
 = sortCont r [5,3] (\s -> sortCont r [1] (merge r s))
 = sortCont r [5] (\s -> sortCont r [3] (sortCont r [1] (merge r . merge r s)))
 = sortCont r [3] (\s -> sortCont r [1] (merge r . merge r [5] s))
 = sortCont r [1] (merge r . merge r [5] [3])
 = merge r (merge r [5] [3]) [1]
 = merge r (3:merge r [5] []) [1]
 = merge r (3:[5]) [1]
 = merge r [3,5] [1]
 = 1:merge r [3,5] []
 = 1:[3,5]
 = [1,3,5]
```

In Kapitel 7 werden wir einen Compiler angeben, der Haskell-Programme durch Programmtransformation in Continuation-Ausdrücke und diese in Assemblerprogramme überführt. Weitere Transformationsschemata einschließlich des Nachweises ihrer Korrektheit finden sich in [57], Kapitel 8.

Kapitel 7

Übersetzer funktionaler Sprachen

7.1 Typinferenz

Wir betrachten einige Haskell-Konstrukte zum Aufbau funktionaler Ausdrücke (siehe §1.2):

1. Variable oder Konstante x
2. Applikation $(e\ e')$[1]
3. Tupel $(e_1, ..., e_n)$
4. Konditional **if** e **then** e_1 **else** e_2
5. Scope mit lokaler Definition **let** $x = e'$ **in** e
6. Abstraktion $\backslash x \rightarrow e$ (oder: $\lambda x.e$)
7. Fixpunkt **fix** $x = e$[2]
8. Fixpunkt **fix** $x_1 = e_1 | \ldots | x_n = e_n$[3]

Beispiele:

e1 = (**fix** length = \s → **if** (null s) **then** 0 **else** length (tail s)+1) *Länge einer Liste*

e2 = (**fix** map = \f → \s → **if** (null s) **then** [] **else** f (head s) : map f (tail s)) *map-Funktion*

In diesem Kapitel werden wie in Haskell mehrstellige Funktionen als einstellige betrachtet, deren Argumente Tupel sind. Die Applikation einer Standardfunktion f auf ein Argument e hat demzufolge immer die Form $(f\ e)$.

Bei Funktionsdefinitionen haben wir stets auf die Angabe von Parameter- und Resultattypen verzichtet. Das ging gut, weil diese Typen aus entsprechenden aktuellen Parametern bzw. Aufrufkontexten hergeleitet (*inferiert*) werden konnten. Werden für einen Ausdruck mehrere Typen hergeleitet, dann liefert die semantische Analyse des Programms eine entsprechende Fehlermeldung. Im Gegensatz zu *Lisp* ist Haskell also durchaus eine *getypte Sprache,* auch wenn die Typen nicht angegeben werden müssen. In ungetypten Sprachen kann man Typisierung nur dadurch simulieren, dass man jede einem Typ entsprechende Datenmenge durch ein Prädikat definiert. Dessen Gültigkeit abzuprüfen ist Teil der Ausführung und nicht der Übersetzung des Programms.

[1] Applikationen, die Komponenten eines Tupels sind, schreiben wir ohne Klammern.
[2] Entspricht in Haskell der Definition $x = e$.
[3] Entspricht in Haskell der Definition $x_1 = e_1\ \ldots\ x_n = e_n$. Im Gegensatz zu **fix** $x_1 = e_1\ \ldots$ **fix** $x_n = e_n$ können hier $x_1, \ldots, x_n$ verschränkt-rekursiv definiert sein, d.h. e_i kann x_j mit $i \neq j$ enthalten.

Formal wird Typinferenz zunächst als System von **Inferenzregeln** beschrieben. Dabei hat jede Regel die Form

$$\frac{\text{state} \vdash \text{exp} : \text{value}}{\text{state}_1 \vdash \text{exp}_1 : \text{value}_1, \ldots, \text{state}_n \vdash \text{exp}_n : \text{value}_n} \Uparrow$$

Über dem waagerechten Strich stehen die **Prämissen**, darunter die **Konklusionen** der Regel. In unserem Fall ist *state* eine Funktion, die jeder Variablen oder Konstanten einen Typ zuordnet. $update(state)(x,t)$ verändert *state* an der Stelle x: x wird der Typ t zugeordnet. Ist x eine Variable, dann kann $state(x)$ freie, d.h. nicht durch $\forall$ gebundene (s.u.) Typvariablen enthalten. Ist x eine Konstante, dann sind alle Typvariablen von $state(x)$ gebunden. Der Ausdruck

$$\text{state} \vdash \text{exp} : \text{value}$$

bedeutet, dass im Zustand *state* dem Ausdruck *exp* ein Wert *value* (hier der Typ von *exp*) zugeordnet wird. Gilt das für die Konklusionen, dann auch für die Prämissen einer Regel. Diese Folgerungsrichtung wird durch den Pfeil $\Uparrow$ angedeutet. Häufig formuliert man auch Interpreter- oder Compilerdefinitionen zunächst als Regelsysteme (siehe Abschnitte 6.2 bis 6.4). Das Compilerschema (4.1) entspricht z.B. Regeln der Form:

$$\frac{a_0 \vdash p(x_1, \ldots, x_n) : e_n(a_0, \ldots, a_n)}{e_0(a_0) \vdash x_1 : a_1, \ldots, e_{n-1}(a_0, \ldots, a_{n-1}) \vdash x_n : a_n} \Uparrow$$

Die Regeln zur Herleitung der Typen der Ausdrücke 1-7 lauten wie folgt:[4] Sei x eine Variable oder Konstante.

1. $\dfrac{state \ \vdash \ x : t}{True} \Uparrow$ $\quad t$ ist eine *Instanz* (s.u.) des Typs $state(x)$

2. $\dfrac{state \ \vdash \ (e\ e') : t}{state \ \vdash \ e : t' \to t,\ state \ \vdash \ e' : t'} \Uparrow$

3. $\dfrac{state \ \vdash \ (e_1, \ldots, e_n) : (t_1, \ldots, t_n)}{state \ \vdash \ e_1 : t_1,\ \ldots,\ state \ \vdash \ e_n : t_n} \Uparrow$

4. $\dfrac{state \ \vdash \ \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : t}{state \ \vdash \ e : bool,\ state \ \vdash \ e_1 : t,\ state \ \vdash \ e_2 : t} \Uparrow$

5. $\dfrac{state \ \vdash \ \textbf{let}\ x = e'\ \textbf{in}\ e : t}{state \ \vdash \ e' : t',\ update(state)(x, \forall\alpha_1, \ldots, \alpha_n t') \ \vdash \ e : t} \Uparrow$

   $\alpha_1, \ldots, \alpha_n$ sind die Typvariablen von t', die in *state* nicht frei vorkommen (s.u.)

6. $\dfrac{state \ \vdash \ \lambda x.e : \alpha \to t}{update(state)(x,\alpha) \ \vdash \ e : t} \Uparrow$ $\quad \alpha$ ist eine neue Typvariable

7. $\dfrac{state \ \vdash \ \textbf{fix}\ x = e : t}{update(state)(x,\alpha) \ \vdash \ e : t} \Uparrow$ $\quad \alpha$ ist eine neue Typvariable

8. $\dfrac{state \ \vdash \ \textbf{fix}\ x_1 = e_1 | \ldots | x_n = e_n : (t_1, \ldots, t_n)}{state' \ \vdash \ e_1 : t_1,\ \ldots,\ state' \ \vdash \ e_n : t_n} \Uparrow$

   $\alpha_1, \ldots, \alpha_n$ sind neue Typvariablen, $state' = update(\ldots(update(state)(x_1, \alpha_1), \ldots))(x_n, \alpha_n)$

Typen sind hier aus Typvariablen wie α und β, den Typkonstruktoren $(\_,\_)$ (Produkt) und $\to$ sowie Datentypnamen zusammengesetzte Ausdrücke. Typvariablen können durch den Allquantor $\forall$ gebunden sein. Der Typ der *map*-Funktion (siehe §1.2) ist z.B. durch den Ausdruck

$$\forall\alpha, \beta\ (\alpha \to \beta) \to [\alpha] \to [\beta]$$

gegeben. α und β sind hier gebunden.

[4] vgl. Reade, Seite 395

Eine **Instanz** eines Typs t entsteht aus t durch Ersetzung einiger der durch $\forall$ gebundenen Variablen durch Typausdrücke ohne gebundene Variablen. Der o.g. Typ von *map* hat z.B. die Instanz

$$\forall\beta\ ([\gamma] \to \beta) \to [[\gamma]] \to [\beta].$$

Um aus den Regeln 1-7 einen Algorithmus zur Typherleitung zu machen, wird bei Regel 1 zunächst die *allgemeinste* Instanz gebildet, d.h. alle gebundenen Typvariablen von $state(x)$ werden durch freie Typvariablen ersetzt. Durch **Typunifikationen** wird diese Instanz schrittweise spezialisiert. Die Spezialisierung ergibt sich aus den Abhängigkeiten zwischen den Typen der Teilausdrücke. Diese Abhängigkeiten sind in den Konklusionen der Regeln 2-7 festgelegt. Regel 2 verlangt z.B. für e einen funktionalen Typ $t' \to t$.

Unifikation der Typen t und t' bedeutet, dass die Variablen von t und t' solange instanziiert werden, bis t und t' identisch sind.

$$t \ = \ (\alpha \to (\text{Int} \to \text{Bool})) \quad \text{und} \quad t' \ = \ ((\beta, \gamma) \to \delta)$$

haben z.B. den **Unifikator** $\sigma = \{(\beta, \gamma)/\alpha,\ (\text{Int} \to \text{Bool})/\delta\}$[5] Für die **Instanzen von** t **und** t' **durch** σ:

$$(\beta, \gamma) \to (\text{Int} \to \text{Bool})) \quad \text{bzw.} \quad (\beta, \gamma) \to (\text{Int} \to \text{Bool}))$$

schreibt man kurz $t\sigma$ bzw. $t'\sigma$. Die Typen

$$\alpha \to (\text{Int} \to \text{Bool}) \quad \text{und} \quad \alpha' \to (\beta, \gamma)$$

haben keinen Unifikator, weil $Int \to Bool$ mit (β, γ) *nicht* unifizierbar ist. Die folgende Funktion `unify` berechnet den *allgemeinsten* Unifikator zweier Typterme. Wir verwenden den polymorphen Datentyp `Tree a` aus Abschnitt 1.2, so dass man mit `unify` auch andere Terme behandeln könnte, z.B. diejenigen, die bei der Ausführung *logischer* Programme unifiziert werden müssen.

```
isin :: a -> Tree a -> Bool                    -- Vorkommen einer Variable in
                                               -- einem Term
x `isin` V y    = x == y
x `isin` F _ ts = any (isin x) ts

for :: Tree a -> a -> Tree a                   -- Substitution einer einzelnen
                                               -- Variable
(t `for` x) y = if x == y then t else V y

(>>>) :: Tree a -> (a -> Tree a) -> Tree a     -- Substitutionsoperator

V x >>> f    = f x
F x ts >>> f = F x (map (>>> f) ts)

andThen :: (a -> Tree a) -> (a -> Tree a) -> a -> Tree a
                                               -- Komposition zweier
(f `andThen` g) x = f x >>> g                  -- Substitutionen

unify    :: Tree a -> Tree a -> Maybe (a -> Tree a)
                                               -- zu Maybe siehe Abschnitt 3.6

unify (V x) (V y)       = if x == y then Just V else Just (V y `for` x)
```

[5] Lies: (β, γ) für Int; $Int \to Bool$ für δ.

```
unify (V x) t           = if x `isin` t then Nothing else Just (t `for` x)
unify t (V x)           = unify (V x) t
unify (F x ts) (F y us) = if x == y then unifyall ts us else Nothing

unifyall :: [Tree a] -> [Tree a] -> Maybe (a -> Tree a)

unifyall [] []          = Just V
unifyall (t:ts) (u:us)  = do f <- unify t u
                             let ts' = map (>>> f) ts
                                 us' = map (>>> f) us
                             g <- unifyall ts' us'
                             Just (f `andThen` g)
unifyall _ _            = Nothing
```

`a `isin` t` stellt fest, ob `a` in `t` vorkommt. Wenn ja, sind `a` und `t` nicht unifizierbar. Auf diesen sog. **occurs check** kann nicht verzichtet werden, auch wenn man die Aufrufe von `unify` auf Terme mit disjunkten Variablenmengen beschränkt. So liefern z.B. die variablendisjunkten Ausdrücke

$$(\alpha, \alpha) \quad \text{und} \quad (\beta, (Int \to \beta))$$

den Unifikator β/α der Teilausdrücke α und β. Bei rekursiver Anwendung des Algorithmus' wären dann β und $Int \to \beta$ zu unifizieren, was ohne den occurs check zum Konflikt führen würde.

Entsprechend dem Aufbau der funktionalen Ausdrücke unseres Mini-Haskell bilden wir aus den Typinferenzregeln die Funktion

`analyze` : Ausdruck → Zustand → Unifikator → (Typ,Unifikator)

mit folgender Eigenschaft:

`analyze e state` $id = (t, \sigma)$ ⇒ Bezüglich `state` ist t der allgemeinste Typ von e.

Der Unifikator σ wird schrittweise aufgebaut und zur Instanziierung der in *state* vorkommenden Typen benutzt.

id bezeichne den identischen Unifikator $\lambda\alpha.\alpha$. Unifikatoren lassen sich komponieren: $\sigma_0\sigma_1$ bezeichnet den Unifikator, der auf einen Typ zunächst σ_0 anwendet und dann auf die entstandenen Instanzen σ_1 anwendet. Das folgende Haskell-Programm für `analyze` entspricht der in [67] entwickelten ML-Implementierung der Typinferenz.

```
analyze x state σ = (t[β1/α1,...,βn/αn], id)
    where ∀α1,...,αn t = state(x)σ
          wobei β1,...,βn neue Typvariablen sind

analyze (e e') state σ = (ασ2, σ1σ2)
    where (t,σ0) = analyze e state σ
          (t',σ1) = analyze e' state σ0
          σ2 = unify t (t' → α)
          wobei α eine neue Typvariable ist

analyze (e1,...,en) state σ = ((t1σn,...,tnσn), σn)
    where (t1,σ1) = analyze e1 state σ
```

```
            (t_2, σ_2) = analyze e_2 state σ_1
            ⋮
            (t_n, σ_n) = analyze e_n state σ_{n-1})

    analyze (if e then e_1 else e_2) state σ = (t_2σ_4, σ_3σ_4)
        where (t, σ_0) = analyze e state σ
              σ_1 = unify t bool
              (t_1, σ_2) = analyze e_1 state σ_0σ_1
              (t_2, σ_3) = analyze e_2 state σ_2)
              σ_4 = unify t_1 t_2

    analyze (let x = e' in e) state σ = (t', σ_1)
        where (t, σ_0) = analyze e' state σ
              (t', σ_1) = analyze e (update state (x,∀α_1,...,α_n t)) σ_0
              wobei α_1,...,α_n die Typvariablen von t sind, die in state nicht frei vorkommen

    analyze (\x→ e) state σ = (ασ_0 → t, σ_0)
        where (t, σ_0) = analyze e (update state (x,α)) σ
              wobei α eine neue Typvariable ist

    analyze (fix x=e) state σ = (tσ_1, σ_0σ_1)
        where (t, σ_0) = analyze e (update state (x,α)) σ
              σ_1 = unify t (ασ_0)
              wobei α eine neue Typvariable ist

    analyze (fix x_1=e_1|...|x_n=e_n) state σ = ((t_1τ_n,...,t_nτ_n), σ_nτ_n)
        where state' = foldl update state [(x_1,α_1),...,(x_n,α_n)]
              (t_1, σ_1) = analyze e_1 state' σ
              τ_1 = unify t_1 (α_1σ_1)
              (t_2, σ_2) = analyze e_2 state' σ_1τ_1
              τ_2 = unify t_2 (α_2σ_2)
              ⋮
              (t_n, σ_n) = analyze e_n (state' σ_{n-1}τ_{n-1})
              τ_n = unify t_n (α_nσ_n)
              wobei α_1,...,α_n neue Typvariablen sind
```

Beispiel: `state` sei wie folgt definiert:

$$
\begin{array}{lcl}
\texttt{state}([]) & = & \forall\alpha\ [\alpha] \\
\texttt{state}(:) & = & \forall\alpha\ (\alpha, [\alpha]) \to [\alpha]) \\
\texttt{state}(\text{head}) & = & \forall\alpha\ [\alpha] \to \alpha \\
\texttt{state}(\text{tail}) & = & \forall\alpha\ [\alpha] \to [\alpha] \\
\texttt{state}(\text{null}) & = & \forall\alpha\ [\alpha] \to \mathit{Bool} \\
\texttt{state}(0) & = & \text{state}(1) \ = \ \text{Int} \\
\texttt{state}(+) & = & (\text{Int},\text{Int}) \to \text{Int} \\
\texttt{state}(\text{deref}) & = & \forall\alpha\ \text{pointer}(\alpha) \to \alpha \\
\texttt{state}(\text{p}) & = & \text{pointer}(\text{pointer}(\text{Int}))
\end{array}
$$

Zeige, dass `analyze` die folgenden Typen ableitet (siehe die Beispiele am Anfang von Abschnitt 6.1):

`analyze e1 state` *id*
$= \texttt{analyze}\ (\textbf{fix}\ \text{length} = \backslash s \rightarrow \textbf{if}\ (\text{null s})\ \textbf{then}\ 0\ \textbf{else}\ \text{length}\ (\text{tail s})+1)$
`state` *id*
$= \ldots = ([\alpha] \rightarrow \text{Int}, \ldots)$

`analyze e2 state` *id*
$= \texttt{analyze}\ (\textbf{fix}\ \text{map} = \backslash f \rightarrow \backslash s \rightarrow \textbf{if}\ (\text{null s})\ \textbf{then}\ []\ \textbf{else}\ f\ (\text{head s}) : \text{map f}\ (\text{tail s}))$
`state` *id*
$= \ldots = ((\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta],\ \ldots)$

`analyze` (deref (deref p)) `state` *id* $= \ldots = (\text{Int}, \ldots)$ ❑

7.2 Übersetzung in λ-Ausdrücke

Der in Abschnitt 1.2 erwähnte λ-Kalkül ist eine vereinfachte funktionale Sprache (siehe z.B. [67], Kap. 12). Im Vergleich zur Syntax des vorigen Abschnitts erlaubt er nur die Bildung der folgenden Ausdrücke:

1. Variable x
2. Konstante k
3. Applikation $(e\ e')$
4. Tupel $(e_1, ..., e_n)$
5. Abstraktion $\lambda x.e$

Das ist die Zielsprache des in diesem Abschnitt beschriebenen Übersetzers. Die Quellsprache ist eine Erweiterung der im vorigen Abschnitt analysierten Menge funktionaler Ausdrücke. Neben den dort angegebenen Konstrukten enthält die Quellsprache hier auch Abstraktionen der Form $\lambda p.e$ sowie Funktionsdefinitionen der Form

$$\text{f p}_1 = \text{e}_1\ \ldots\ \text{f p}_n = \text{e}_n$$

möglich, wobei $p, p_1, \ldots, p_n$ Muster (*patterns* im Sinne von Abschnitt 1.2) sind.

Die Übersetzung dieser Ausdrücke bzw. Definitionen in den λ-Kalkül ist eine reine Programmtransformation im Sinne von Kapitel 6. Es werden keine zusätzlichen Attribute benötigt. Wir formulieren sie wieder als Regelsystem, das allerdings von einfacherer Art als der Typinferenzkalkül des vorigen Abschnitts ist. Jeder Übersetzungsschritt besteht in der Ersetzung einer Instanz der Prämisse einer Regel durch die entsprechende Instanz der Konklusion der Regel.[6]

Bei der Transformation werden neue (unten kursiv gedruckte) Funktionen wie *fix, if, equal*, usw., eingeführt, deren Bedeutung in Abschnitt 6.2.1 definiert wird.

Außer den oben verwendeten Abkürzungen bezeichnet d eine Wert- oder Funktionsdefinition, $dList$ eine Liste von Definitionen, p ein Muster, f eine Funktionsvariable und $mList$ eine sog. *Matchliste* der Form $p_1.e_1\ |...|\ p_n.e_n$.

1. $\dfrac{\text{e}_1\ op\ \text{e}_2}{op\ (\text{e}_1,\text{e}_2)}$ für alle Infix-Funktionen op

[6] In [67], §12.2, ist die Übersetzung für ML-Ausdrücke beschrieben.

2. $\dfrac{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3}{if\ (e_1,e_2,e_3)}$

3. $\dfrac{\textbf{let } d\ dList \textbf{ in } e}{\textbf{let } d \textbf{ in let } dList \textbf{ in } e}$

4. $\dfrac{\textbf{let } p{=}e_1 \textbf{ in } e_2}{(\lambda\ p.e_2)\ e_1}$

5. $\dfrac{f\ p_1 = e_1\ ...\ f\ p_n = e_n}{\textbf{fix } f = \lambda\ p_1.e_1\ |...|\ p_n.e_n}$ für alle $n \geq 1$

6. $\dfrac{\textbf{fix } f = e}{f = (fix\ (\lambda\ f.e))}$

7. $\dfrac{\textbf{fix } f_1 = e_1\ |...|\ f_n = e_n}{(f_1,...,f_n) = (fix\ (\lambda(f_1,...,f_n).(e_1,...,e_n)))}$ für alle $n \geq 1$

8. $\dfrac{\lambda\ mList}{\lambda\ x.(match\ (mList,x))}$ falls x in mList nicht vorkommt

9. $\dfrac{match\ (p.e \mid mList,\ x)}{matchrule\ (p,x,e,(match\ (mList,x)))}$

10. $\dfrac{match\ (p.e,\ x)}{matchrule\ (p,x,e,matchfail)}$

11. $\dfrac{matchrule\ (x,e_1,e_2,e_3)}{\lambda x.e_2\ e_1}$

12. $\dfrac{matchrule\ (k,e_1,e_2,e_3)}{if\ (equal\ (k,e_1),e_2,e_3)}$

13. $\dfrac{matchrule\ (con\ p,e_1,e_2,e_3)}{if\ ((eq_{con}\ e_1),\ (matchrule\ (p,args\_of\ e_1,e_2,e_3)),e_3)}$ für alle Konstruktoren *con*

14. $\dfrac{matchrule\ ((p_1,...,p_n),e_1,e_2,e_3)}{matchrule\ (p_1,sel_{n,1}\ e_1,\ (matchrule\ ((p_2,...,p_n),(sel_{n,2}\ e_1,...,sel_{n,n}\ e_1),e_2,e_3)),e_3)}$ für alle $n > 1$

15. $\dfrac{\lambda x.e\ z}{e[z/x]}$ falls e keinen Teilausdruck der Form $\lambda z.e$ enthält.

In Regel 6 hat die dort eingeführte Funktion fix als Argument eine λ-Abstraktion $\lambda f.e$. Der Wert des Ausdrucks $(fix\ \lambda f.e)$ ist ein **Fixpunkt** von $\lambda f.e$, d.h. eine Lösung der Gleichung $f = e$ in der Funktionsvariablen f. Anstelle von $(fix\ \lambda f.e)$ schreibt man auch $\mu f.e$ und spricht von einer **μ-Abstraktion**.

Mit den Regeln 9-14 werden die Funktionen *match* und *matchrule* aus dem erzeugten λ-Ausdruck eliminiert.

Beispiel Fakultät

fact 0 = 1
fact x = x*(fact x-1)
$\xrightarrow{Regel\ 1}$
fact 0 = 1
fact x = *mul* (x,fact *sub* (x,1))
$\xrightarrow{Regel\ 5}$
fix fact = λ 0.1 | x.*mul* (x,fact *sub* (x,1))
$\xrightarrow{Regel\ 6}$
fact = fix (λfact.λ 0.1 | x.*mul* (x,fact *sub* (x,1)))

$\xrightarrow{Regel\ 8}$

fact = fix λfact.λz.($match$ (0.1 | x.mul (x,fact sub (x,1)),z))

$\xrightarrow{Regel\ 9}$

fact = fix λfact.λz.($matchrule$ (0,z,1,$match$ (x.mul (x,fact sub (x,1),z))))

$\xrightarrow{Regel\ 10}$

fact = fix λfact.λz.($matchrule$ (0,z,1,$matchrule$ (x,z,mul (x,(fact sub (x,1))), $matchfail$)))

$\xrightarrow{Regel\ 12}$

fact = fix λfact.λz.(if($equal$ (0,z),1,$matchrule$ (x,z,mul (x,(fact sub (x,1))), $matchfail$)))

$\xrightarrow{Regel\ 11}$

fact = fix λfact.λz.(if($equal$ (0,z),1,(λx.(mul (x,fact sub (x,1))) z)))

$\xrightarrow{Regel\ 15}$

fact = fix λfact.λz.(if($equal$ (0,z),1,mul (z,fact sub (z,1))))

Ein Aufruf (fact n) wird demnach wie folgt übersetzt:

let fact 0 = 1
    fact x = x*(fact x-1)
in fact n

$\xrightarrow{s.o.}$

let fact = fix λfact.λz.(if($equal$ (0,z),1,mul (z,fact sub (z,1))))
in fact n

$\xrightarrow{Regel\ 4}$

(λfact.fact n) (fix λfact.λz.(if($equal$ (0,z),1,mul (z,fact sub (z,1)))))

7.2.1 Auswertung der λ-Ausdrücke

Ein Interpreter für λ-Ausdrücke kann ebenfalls als Regelsystem formuliert werden:

1. $\dfrac{fix\ \mathrm{e}}{\mathrm{e}\ (fix\ \mathrm{e})}$

2. $\dfrac{sel_{n,i}\ (\mathrm{e}_1,\ldots,\mathrm{e}_n)}{\mathrm{e}_i}$ für alle $1 \leq i \leq n > 1$

3. $\dfrac{eq_{con}\ (con\ \mathrm{e})}{true}$

4. $\dfrac{eq_{con}\ (con'\ \mathrm{e})}{false}$ für alle Konstruktoren $con' \neq con$

5. $\dfrac{args\_of\ (con\ \mathrm{e})}{\mathrm{e}}$ für alle Konstruktoren con

6. $\dfrac{op\ \mathrm{k}}{\mathrm{Wert}(op(\mathrm{k}))}$ falls op eine Standardfunktion und k eine Konstante ist.

7. $\dfrac{\lambda\mathrm{x}.\mathrm{e}_1\ \mathrm{e}_2}{\mathrm{e}_1[\mathrm{e}_2/\mathrm{x}]}$ falls keine in e_1 auftretende, durch λ gebundene Variable in e_2 frei vorkommt.

Die letzte Regel enthält das entscheidende Prinzip dieses Interpreters: Auswertung basiert hier auf *textueller Substitution*, die im Rahmen des λ-Kalküls β-Reduktion genannt wird und in imperativen Sprachen der **call-by-name**-Übergabe aktueller Parameter an eine Funktionsprozedur entspricht. Da eine direkte Implementierung der

β-Reduktion sehr aufwändig ist, dient der λ-Kalkül i.a. nur als Zwischensprache bei der Übersetzung funktionaler Programme (und der hier angegebene Interpreter vor allem der semantischen Fundierung).

Regel 1 gibt die Fixpunkteigenschaft eines Ausdrucks der Form $(fix\ e)$ wieder. Sie wird immer dann angewendet, wenn eine Applikation $((fix\ e)\ e')$ ausgewertet werden soll (vgl. folgendes Beispiel). Auch diese Regel eignet sich kaum zur direkten Implementierung, es sei denn, das Kopieren des Ausdrucks e wird—wie bei den in Kap. 7 beschriebenen Interpretern—durch das Setzen von Zeigern vermieden. Regel 1 ist natürlich nur dann korrekt, wenn der Fixpunkt $(fix\ e)$ überhaupt existiert. Mit dieser Frage befassen sich die Modelltheorie des λ-Kalküls, insbesondere die **Fixpunktsemantik** (s. z.B. [40], [76], [43], [57]).

Beispiel Fakultät Wir werten nach der im vorangehenden Beispiel durchgeführten Übersetzung den λ-Ausdruck für (fact 3) aus:

(λfact.fact 3) (fix λfact.λz.(if($equal$ (0,z),1,mul (z,fact (sub (z,1))))))
$\xrightarrow{Regel\ 7}$
(fix λfact.λz.(if ($equal$ (0,z),1,mul (z,fact (sub (z,1))))) 3
$\xrightarrow{Regel\ 1}$
(λfact.λz.(if ($equal$ (0,z),1,mul (z,fact (sub (z,1)))) (fix ...))) 3
$\xrightarrow{Regel\ 7}$
λz.(if ($equal$ (0,z),1,mul (z, fix ...) (sub (z,1)))) 3
$\xrightarrow{Regel\ 7}$
if ($equal$ (0,3),1,mul (3, (fix ...) (sub (3,1))))
$\xrightarrow{Regel\ 6}$
mul (3, (fix ...) 2)
$\xrightarrow{Regel\ 1}$
mul (3, (λfact.λz.(if ($equal$ (0,z),1,mul (z,(fact (sub (z,1))))))) (fix ...) 2)
⋮
mul (3, mul (2, (fix ...) 1))
$\xrightarrow{Regel\ 1}$
mul (3, mul (2, (λfact.λz.(if ($equal$ (0,z),1,mul (z,fact (sub (z,1)))))) (fix ...) 1)))
⋮
mul (3, mul (2, mul (1, (fix ...) 0)))
$\xrightarrow{Regel\ 1}$
mul (3, mul (2, mul (1,(λfact.λz.(if ($equal$ (0,z),1,mul (z,fact (sub (z,1))))) (fix ...) 0))))
$\xrightarrow{Regel\ 7}$
mul (3, mul (2, mul (1, (λ.z(if ($equal$ (0,z),1,mul (z,((fix ...) (sub (z,1)))))) 0))))
$\xrightarrow{Regel\ 7}$
mul (3, mul (2, mul (1, (if ($equal$ (0,0),1,mul (0,((fix ...) (sub (0,1)))))))))
$\xrightarrow{Regel\ 6}$
mul (3, mul (2, mul (1, 1)))
$\xrightarrow{Regel\ 6}$
6

7.3 Übersetzung von λ-Ausdrücken in Continuations

Wie schon in Kapitel 6 erwähnt wurde, sind Übersetzungen durch Programmtransformation i.a. keine Übergänge zwischen verschiedenen Sprachen, sondern Modifizierungen der Quellprogramme mit dem Ziel, komplexe

Sprachkonstrukte zu eliminieren, um die Programme mit einem einfachen Auswertungsmechanismus schnell ausführen zu können. Die oben beschriebene Übersetzung von Haskell-Programmen in λ-Ausdrücke ist von dieser Art, weil der λ-Kalkül—von wenigen syntaktischen Unterschieden abgesehen—eine Teilsprache von Haskell ist. Zu den in Abschnitt 6.2.1 genannten Ineffizienzen des dort angegebenen λ-Kalkül-Interpreters kommt noch eine, die sich durch eine weitere Programmtransformation beheben lässt. Sie besteht darin, dass eine jeweils anwendbare Regel des Interpreters nicht immer auf den Gesamtausdruck angewendet werden kann, sondern nur auf einen—erst zu findenden—Teilausdruck. Damit hat dieser Interpreter vor jedem Ausführungsschritt ein nichttriviales Suchproblem zu lösen. Um genau das zu vermeiden, wird der erzeugte λ-Ausdruck zunächst in eine Continuation-Form gebracht.

Das Prinzip der Continuations haben wir in Abschnitt 6.3 zur Übersetzung rekursiver in iterative Programme benutzt. Die dort angegebenen Beispiel-Berechnungen von Ausdrücken in Continuation-Form zeigen die Vereinfachung bei der Auswertung: alle Ersetzungsschritte betreffen den Gesamtausdruck, womit das o.g. Suchproblem entfällt. Werden alle im gegebenen Ausdruck auftretenden Funktionen in Continuation-Form gebracht, dann können entsprechend geänderte Auswertungsregeln (siehe §6.3.1) immer auf den Gesamtausdruck angewendet werden.

Für einen beliebigen λ-Ausdruck e transformieren die u.a. Regeln den Ausdruck $(trans\ e)$ im Kontext einer Applikation in einen "optimierten" λ-Ausdruck, der in folgendem Sinne zu e äquivalent ist. Zunächst wird aus einer Funktion $f : A \to B$ durch folgende Definition die Funktion $f_{Cont} : (B \to C) \to (A \to C)$ gebildet: Für alle *Continuations* $g : B \to C$ und $a \in A$,

$$f_{Cont}(g)(a) \ =_{def} \ g(f(a))$$

(siehe §6.3). $trans$ überführt die durch e repräsentierte Funktion f in einen Ausdruck für f_{Cont}:

$$Semantik(trans\ e\ c)(a) \ = \ Semantik(e)_{Cont}(Semantik(c))(a) \ = \ Semantik(c)(Semantik(e)(a))$$

für alle typkonformen Continuations c. Die folgenden Übersetzungsregeln sind i.w. [6], S. 124, entnommen.

1. $\dfrac{trans\ \mathrm{x}\ \mathrm{c}}{\mathrm{c}\ \mathrm{x}}$ falls x eine Variable oder Konstante ist
2. $\dfrac{trans\ (\mathrm{x}\ \mathrm{e})\ \mathrm{c}}{trans\ \mathrm{e}\ (\mathrm{x}_C\ \mathrm{c})}$ falls x eine Variable oder Konstante ist
3. $\dfrac{trans\ (\mathrm{e}\ \mathrm{e'})\ \mathrm{c}}{trans\ \mathrm{e'}\ (trans\ \mathrm{e}\ \mathrm{id})\ \mathrm{c}}$
4. $\dfrac{trans\ (\mathrm{e}_1,\ldots,\mathrm{e}_n)\ \mathrm{c}}{trans\ \mathrm{e}_1\ (trans\ \mathrm{e}_2\ (\ldots(trans\ \mathrm{e}_n\ \mathrm{c})\ldots))}$
5. $\dfrac{trans\ (if\ (\mathrm{e}_1,\mathrm{e}_2,\mathrm{e}_3))\ \mathrm{c}}{trans\ \mathrm{e}_1\ (if_C\ ((trans\ \mathrm{e}_2\ \mathrm{c}),\ (trans\ \mathrm{e}_3\ \mathrm{c})))}$
6. $\dfrac{trans\ \lambda\mathrm{x.e}\ \mathrm{c}}{\mathrm{c}\ \lambda\mathrm{d}.\lambda\mathrm{x}.(trans\ \mathrm{e}\ \mathrm{d})}$
7. $\dfrac{trans\ (fix\ \lambda\mathrm{f}.\lambda\mathrm{x.e})\ \mathrm{c}}{\mathrm{c}\ (fix\ \lambda\mathrm{f}_C.\lambda\mathrm{d}.\lambda\mathrm{x}.(trans\ \mathrm{e}\ \mathrm{d}))}$

Die Übersetzung ist beendet, wenn der transformierte Ausdruck die Funktion $trans$ nicht mehr enthält.

Beispiel Fakultät Wir transformieren den in Abschnitt 6.2 erzeugten λ-Ausdruck für $fact$:

$trans$ fact id

$\xrightarrow{s.o.}$

$trans$ (fix λfact.λz.(if (($equal$ (0,z)),1,(mul (z,(fact (sub (z,1)))))))) id

$\xrightarrow{Regel\ 7}$

fix λfact$_C$.λc.λz.($trans$ (if (($equal$ (0,z)),1,(mul (z,(fact (sub (z,1)))))))) c)

$\xrightarrow{Regel\ 5}$

fix λfact$_C$.λc.λz.($trans$ ($equal$ (0,z)) (if_C ($trans$ 1 c) ($trans$ (mul (z,(fact (sub (z,1))))) c)))

$\xrightarrow{Regel\ 2}$

fix λfact$_C$.λc.λz.($trans$ (0,z) ($equal_C$ (if_C ($trans$ 1 c) ($trans$ (mul (z,(fact (sub (z,1)))))) c)))

$\xrightarrow{Regel\ 4}$

fix λfact$_C$.λc.λz.($trans$ 0 ($trans$ z ($equal_C$ (if_C ($trans$ 1 c) ($trans$ (mul (z,(fact (sub (z,1)))))) c))))

$\xrightarrow{Regel\ 1}$

fix λfact$_C$.λc.λz.($trans$ 0 ($trans$ z ($equal_C$ (if_C (c 1) ($trans$ (mul (z,(fact (sub (z,1)))))) c))))

$\xrightarrow{Regel\ 1}$

fix λfact$_C$.λc.λz.($trans$ 0 (($equal_C$ (if_C (c 1) ($trans$ (mul (z,(fact (sub (z,1)))) z))) c))))

$\xrightarrow{Regel\ 1}$

fix λfact$_C$.λc.λz.(($equal_C$ (if_C (c 1) ($trans$ (mul (z,(fact (sub (z,1)))) c)))) z) 0)

$\xrightarrow{Regel\ 2}$

fix λfact$_C$.λc.λz.(($equal_C$ (if_C (c 1) ($trans$ (z,fact (sub (z,1))) (mul_C c))) z) 0)

$\xrightarrow{Regel\ 4}$

fix λfact$_C$.λc.λz.(($equal_C$ (if_C (c 1) ($trans$ z ($trans$ (fact (sub (z,1))) (mul_C c)))) z) 0)

$\xrightarrow{Regel\ 1}$

fix λfact$_C$.λc.λz.((($equal_C$ (if_C (c 1) ($trans$ (fact (sub (z,1))) (mul_C c) z))) z) 0)

$\xrightarrow{Regel\ 2}$

fix λfact$_C$.λc.λz.(((($equal_C$ (if_C ((c 1) ($trans$ (sub (z,1)) (fact$_C$ (mul_C c))) z))) z) 0)

$\xrightarrow{Regel\ 2}$

fix λfact$_C$.λc.λz.((($equal_C$ (if_C (c 1) ($trans$ (z,1) (sub_C (fact$_C$ (mul_C c)))) z)) z) 0)

$\xrightarrow{Regel\ 4}$

fix λfact$_C$.λc.λz.(((($equal_C$ (if_C (c 1) ($trans$ z ($trans$ 1 (sub_C (fact$_C$ (mul_C c))) z)))) z) 0)

$\xrightarrow{Regel\ 1}$

fix λfact$_C$.λc.λz.((($equal_C$ (if_C (c 1) (((sub_C (fact$_C$ (mul_C c))) 1) z) z)) z) 0)

7.3.1 Auswertung der Continuation-Ausdrücke

Ein Interpreter für Continuation-Ausdrücke lautet wie folgt:

8. $\dfrac{fix\ \mathrm{e}}{\mathrm{e}\ (fix\ \mathrm{e})}$

9. $\dfrac{if_C\ \mathrm{c}\ \mathrm{c'}\ true}{\mathrm{c}}$

10. $\dfrac{if_C\ \mathrm{c}\ \mathrm{c'}\ false}{\mathrm{c'}}$

11. $\dfrac{op_C\ \mathrm{c}\ \mathrm{k}_1\ \ldots\ \mathrm{k}_n}{\mathrm{c}\ \mathrm{Wert}(op(\mathrm{k}_n,\ldots,\mathrm{k}_1))}$ falls op eine Standardfunktion ist und $k_1, \ldots, k_n$ Konstanten sind.

12. $\dfrac{\lambda \mathrm{x}.\mathrm{e}_1\ \mathrm{e}_2}{\mathrm{e}_1[\mathrm{e}_2/\mathrm{x}]}$ falls keine in e_1 auftretende, durch λ gebundene Variable in e_2 frei vorkommt.

Beispiel Fakultät Wir werten ($fact$ 3) aus:

trans (fact 3) id

$\xrightarrow{Regel\ 2}$

trans 3 (fact_C id)

$\xrightarrow{Regel\ 1}$

fact_C id 3

$\xrightarrow{s.o.}$

(fix $\lambda\text{fact}_C.\lambda\text{c}.\lambda\text{z}.(((equal_C\ (if_C\ (\text{c}\ 1)\ (((sub_C\ (\text{fact}_C\ (mul_C\ \text{c}))\ 1)\ \text{z})\ \text{z})))\ \text{z})\ 0))$ id 3

$\xrightarrow{Regel\ 8}$

$(\lambda\text{fact}_C.\lambda\text{c}.\lambda\text{z}.(((equal_C\ (if_C\ (\text{c}\ 1)\ (((sub_C\ (\text{fact}_C\ (mul_C\ \text{c}))\ 1)\ \text{z})\ \text{z})))\ \text{z})\ 0)\ (fix \ldots 0))$ id 3

$\xrightarrow{Regel\ 12}$

$(\lambda\text{c}.\lambda\text{z}.(((equal_C\ (if_C\ (\text{c}\ 1)\ (((sub_C\ ((fix \ldots 0)\ (mul_C\ \text{c}))\ 1)\ \text{z})\ \text{z})))\ \text{z})\ 0))$ id 3

$\xrightarrow{Regel\ 12}$

$(\lambda\text{z}.(((equal_C\ (if_C\ (id\ 1)\ (((sub_C\ ((fix \ldots 0)\ (mul_C\ id))\ 1)\ \text{z})\ \text{z})))\ \text{z})\ 0))$ 3

$\xrightarrow{Regel\ 12}$

$(equal_C\ (if_C\ (id\ 1)\ (((sub_C\ ((fix \ldots 0)\ (mul_C\ id))\ 1)\ 3)\ 3)))$ 3 0

$\xrightarrow{Regel\ 11}$

$(if_C\ (id\ 1)\ (((sub_C\ ((fix \ldots 0)\ (mul_C\ id))\ 1)\ 3)\ 3))\ false$

$\xrightarrow{Regel\ 10}$

$((sub_C\ ((fix \ldots 0)\ (mul_C\ id)))\ 1)$ 3 3

$\xrightarrow{Regel\ 11}$

$(((fix \ldots 0)\ (mul_C\ id)))$ 2 3

$\xrightarrow{Regel\ 8}$

$((((\lambda\text{fact}_C.\lambda\text{c}.\lambda\text{z}.(((equal_C\ (if_C\ (\text{c}\ 1)\ (((sub_C\ (\text{fact}_C\ (mul_C\ \text{c}))\ 1)\ \text{z})\ \text{z})))\ \text{z})\ 0))$
$(fix \ldots 0))\ (mul_C\ id)))$ 2 3

$\xrightarrow{dreimal\ Regel\ 12}$

$((equal_C\ (if_C\ ((mul_C\ id)\ 1)\ ((sub_C\ ((fix \ldots 0)\ (mul_C\ (mul_C\ id)))\ 1)\ 2)\ 2))\ 2)$ 0 3

$\xrightarrow{Regel\ 11}$

$((if_C\ ((mul_C\ id)\ 1)\ (((sub_C\ ((fix \ldots 0)\ (mul_C\ (mul_C\ id))))\ 1)\ 2)\ 2))\ false$ 3

$\xrightarrow{Regel\ 10}$

$(((sub_C\ ((fix \ldots 0)\ (mul_C\ (mul_C\ id))))\ 1)\ 2)$ 2 3

$\xrightarrow{Regel\ 11}$

$((((fix \ldots 0)\ (mul_C\ (mul_C\ id))))\ 1)$ 2 3

$\xrightarrow{Regel\ 8}$

$(((((\lambda\text{fact}_C.\lambda\text{c}.\lambda\text{z}.(((equal_C\ (if_C\ (\text{c}\ 1,((sub_C\ (\text{fact}_C\ (mul_C\ \text{c}))\ 1)\ \text{z})\ \text{z})))\ \text{z})\ 0))$
$(fix \ldots 0))\ (mul_C\ (mul_C\ id)))\ 1)$ 2 3

$\xrightarrow{dreimal\ Regel\ 12}$

$(((equal_C\ (if_C\ ((mul_C\ (mul_C\ id))\ 1)\ (((sub_C\ ((fix \ldots 0)\ (mul_C\ (mul_C\ (mul_C\ id))))\ 1)\ 1)\ 1)))\ 1)\ 0)$ 2 3

$\xrightarrow{Regel\ 11}$

$(((if_C\ ((mul_C\ (mul_C\ id))\ 1)\ (((sub_C\ ((fix \ldots 0)\ (mul_C\ (mul_C\ (mul_C\ id))))\ 1)\ 1)\ 1)))\ false)$ 2 3

$\xrightarrow{Regel\ 10}$

$((((sub_C\ ((fix \ldots 0)\ (mul_C\ (mul_C\ (mul_C\ id)))))\ 1)\ 1)\ 1)$ 2 3

$\xrightarrow{Regel\ 11}$

$(((((fix \ldots 0)\ (mul_C\ (mul_C\ (mul_C\ id)))))\ 0)\ 1)$ 2 3

$\xrightarrow{Regel\ 8}$

$((((((\lambda\text{fact}_C.\lambda\text{c}.\lambda\text{z}.(((equal_C\ (if_C\ (\text{c}\ 1,((sub_C\ (\text{fact}_C\ (mul_C\ \text{c}))\ 1)\ \text{z})\ \text{z})))\ \text{z})\ 0))$
$(fix \ldots 0))\ (mul_C\ (mul_C\ (mul_C\ id)))))\ 0)\ 1)$ 2 3

$$\overset{dreimal\ Regel\ 12}{\longrightarrow}$$

$((((equal_C\ (if_C\ ((mul_C\ (mul_C\ (mul_C\ id)))\ 1)$
$(((sub_C\ ((fix \ldots 0)\ (mul_C\ (mul_C\ (mul_C\ (mul_C\ id)))))\ 1)\ 0)\ 0)))\ 0)\ 0)\ 1)\ 2\ 3$

$$\overset{Regel\ 11}{\longrightarrow}$$

$((((if_C\ ((mul_C\ (mul_C\ (mul_C\ id)))\ 1)$
$(((sub_C\ ((fix \ldots 0)\ (mul_C\ (mul_C\ (mul_C\ (mul_C\ id)))))\ 1)\ 0)\ 0)))\ true)\ 1)\ 2\ 3$

$$\overset{Regel\ 9}{\longrightarrow}$$

$((mul_C\ (mul_C\ (mul_C\ id))\ 1)\ 1)\ 2\ 3$

$$\overset{Regel\ 11}{\longrightarrow}$$

$((mul_C\ (mul_C\ id))\ 1)\ 1\ 6$

$$\overset{Regel\ 11}{\longrightarrow}$$

$(mul_C\ id)\ 1\ 6$

$$\overset{Regel\ 11}{\longrightarrow}$$

$id\ 6 = 6$

Im Unterschied zur Auswertung des ursprünglichen λ-Ausdrucks für $(fact\ 3)$ (siehe §6.2.1) besteht jeder Schritt der Auswertung von $trans(fact\ 3)\ id)$ in der Anwendung der jeweils am weitesten links stehenden Funktion auf ihr Argument (*outermost*-Strategie). Deshalb entspricht die Interpretation eines Continuation-Ausdrucks der Ausführung einer linearen Befehlsfolge, also eines Assemblerprogramms! Dieses wird durch den folgenden letzten Übersetzungsschritt erzeugt.

7.4 Übersetzung von Continuations in Assemblerprogramme

Wir übersetzen die Continuation-Ausdrücke in **Kombinator**-Terme. Kombinatoren sind Funktionen, die ihre Argumente umordnen, kopieren o.ä., aber keine komplexen Berechnungen durchführen. Insbesondere wird durch ihre Verwendung die aufwendige β-Reduktion vermieden (siehe auch §7.2). Kombinatoraufrufe lassen sich direkt in Assemblerkommandos übertragen, d.h. Kombinatorterme entsprechen Assemblerprogrammen. Wir folgen dem Ansatz von [6], §6.3. Ähnlich arbeiten Compiler für funktionale Sprachen (vgl. [2]). Damit die erzeugten Programme linear ausgeführt werden können, erzeugen die Übersetzungsregeln—ggf. mehrere—markierte und nur aus unären Kombinatoren zusammengesetzte Terme:

1. $\dfrac{fix\ \lambda f_C.e}{f\ \rhd\ e}$ f wird zur Codeadresse von e.

2. $\dfrac{\lambda x_1 \ldots \lambda x_n.e}{[x_1, \ldots, x_n]_0\ e}$

3. $\dfrac{[x_1, \ldots, x_n]_p\ (e\ x_i)}{dupl_{p+i}\ :\ ([x_1, \ldots, x_n]_{p+1}\ e)}$ $1 \leq i \leq n$

4. $\dfrac{[x_1, \ldots, x_n]_p\ (e\ y)}{([x_1, \ldots, x_n]_{p+1}\ e)\ y}$ falls $y \notin \{x_1, \ldots, x_n\}$ eine Variable ist

5. $\dfrac{[x_1, \ldots, x_n]_p\ (e\ k)}{push\ k\ :\ ([x_1, \ldots, x_n]_{p+1}\ e)}$ falls k eine Konstante ist

6. $\dfrac{[x_1, \ldots, x_n]_p\ (op_C\ e)}{op\ :\ ([x_1, \ldots, x_n]_{p-k+1}\ e)}$

   falls op eine k-stellige Standardfunktion ist und e weder Konstante noch Variable ist

7. $\dfrac{[x_1,\ldots,x_n]_p\ (f_C\ e)}{\begin{array}{l}[push\ g,\ push\ f]\\ g\ \rhd\ ([x_1,\ldots,x_n]_p\ e)\end{array}}$ falls f keine Standardfunktion und g eine neue Adresse ist

8. $\dfrac{[x_1,\ldots,x_n]_p\ (if_C\ (e,e'))}{\begin{array}{l}if\ g\ :\ ([x_1,\ldots,x_n]_{p-1}\ e)\\ g\ \rhd\ ([x_1,\ldots,x_n]_{p-1}\ e')\end{array}}$ falls g eine neue Adresse ist

9. $\dfrac{[x_1,\ldots,x_n]_p\ x_i}{exed_{p,n,i}}$ $1 \le i \le n$

10. $\dfrac{[x_1,\ldots,x_n]_p\ y}{y}$ falls $y \notin \{x_1,\ldots,x_n\}$ eine Variable ist

Zu Beginn der Ausführung des aus $[x_1,\ldots,x_n]_p\ e$ gebildeten Assemblerprogramms (siehe §6.4.1) befindet sich das i-te Argument der dem Teilausdruck e entsprechenden Funktion an der (p+i)-ten Kellerposition.

Beispiel Fakultät Der in Abschnitt 6.3 gebildete Continuation-Ausdruck wird mit diesen Regeln in einen Kombinatorterm übersetzt:

$$fix\ \lambda fact_C.\lambda c.\lambda z.(((equal_C(if_C(c\ 1)(((sub_C(fact_C(mul_C c)))1)z)z))z)0)$$
$$\xrightarrow{Regel\ 1}$$
$$fact\ \rhd\ \lambda c.\lambda z.(((equal_C(if_C(c\ 1)(((sub_C(fact_C(mul_C c)))1)z)z))z)0)$$
$$\xrightarrow{Regel\ 2}$$
$$fact\ \rhd\ ([c,z]_0\ ((equal_C(if_C(c\ 1)(((sub_C(fact_C(mul_C c)))1)z)z))z)0)$$
$$\xrightarrow{Regel\ 5}$$
$$fact\ \rhd\ push\ 0\ :\ ([c,z]_1\ ((equal_C(if_C(c\ 1,((sub_C(fact_C(mul_C c)))1)z)z))z))$$
$$\xrightarrow{Regel\ 3}$$
$$fact\ \rhd\ push\ 0\ :\ dupl_3\ :\ ([c,z]_2\ (equal_C(if_C(c\ 1)(((sub_C(fact_C(mul_C c)))1)z)z)))$$
$$\xrightarrow{Regel\ 6}$$
$$fact\ \rhd\ push\ 0\ :\ dupl_3\ :\ equal:\ ([c,z]_1\ (if_C(c\ 1)(((sub_C(fact_C(mul_C c)))1)z)z))$$
$$\xrightarrow{Regel\ 8}$$
$$\begin{array}{l}fact\ \rhd\ push\ 0\ :\ dupl_3\ :\ equal\ :\ if\ g\ :\ ([c,z]_0\ (c\ 1))\\ g\ \rhd\ ([c,z]_0\ (((sub_C(fact_C(mul_C c)))1)z)z)\end{array}$$
$$\xrightarrow{Regeln\ 5\ und\ 3}$$
$$\begin{array}{l}fact\ \rhd\ push\ 0\ :\ dupl_3\ :\ equal\ :\ if\ g\ :\ push\ 1\ :\ ([c,z]_1\ c)\\ g\ \rhd\ dupl_2\ :\ ([c,z]_1\ (((sub_C(fact_C(mul_C c)))1)z))\end{array}$$
$$\xrightarrow{Regel\ 9}$$
$$\begin{array}{l}fact\ \rhd\ [push\ 0,\ dupl_3,\ equal,\ if\ g,\ push\ 1,\ exed_{1,2,1}]\\ g\ \rhd\ dupl_2\ :\ ([c,z]_1\ (((sub_C(fact_C(mul_C c)))1)z))\end{array}$$
$$\xrightarrow{Regel\ 3}$$
$$\begin{array}{l}fact\ \rhd\ \ldots\\ g\ \rhd\ dupl_2\ :\ dupl_3\ :\ ([c,z]_2\ ((sub_C(fact_C(mul_C c)))1))\end{array}$$
$$\xrightarrow{Regel\ 5}$$
$$\begin{array}{l}fact\ \rhd\ \ldots\\ g\ \rhd\ dupl_2\ :\ dupl_3\ :\ push\ 1\ :\ ([c,z]_3\ (sub_C(fact_C(mul_C c))))\end{array}$$
$$\xrightarrow{Regel\ 6}$$
$$\begin{array}{l}fact\ \rhd\ \ldots\\ g\ \rhd\ dupl_2\ :\ dupl_3\ :\ push\ 1\ :\ sub\ :\ ([c,z]_2\ (fact_C(mul_C c)))\end{array}$$

$\xrightarrow{Regel\ 7}$

$fact \rhd \ldots$
$g \rhd [dupl_2,\ dupl_3,\ push\ 1,\ sub,\ push\ h,\ push\ fact]$
$h \rhd ([c,z]_2\ (mul_C c))$

$\xrightarrow{Regel\ 6}$

$fact \rhd \ldots$
$g \rhd \ldots$
$h \rhd mul\ :\ ([c,z]_1\ c)$

$\xrightarrow{Regel\ 9}$

$fact \rhd \ldots$
$g \rhd \ldots$
$h \rhd [mul,\ exed_{1,2,1}]$

Nach zwei weiteren Übersetzungsschritten erhält man daraus den Kombinatorterm für den Aufruf ($fact$ 3) (siehe §6.3):

trans (fact 3) id

$\xrightarrow{siehe\ §6.3}$

$fact_C\ id\ 3$

$\xrightarrow{Regel\ 5}$

$push\ 3\ :\ (fact_C\ id)$

$\xrightarrow{Regel\ 7}$[7]

$[push\ 3,\ push\ id,\ push\ fact]$

7.4.1 Auswertung der Kombinatorterme

Die erzeugten Befehlsfolgen C werden auf einem Keller K (beide dargestellt als Haskell-Listen) ausgeführt. Die Auswertungsregeln lauten wie folgt:

1. $\dfrac{push\ x\ :\ C \quad K}{C \quad x:K}$
2. $\dfrac{dupl_n\ :\ C \quad x_1:\cdots:x_n:K}{C \quad x_n:x_1:\cdots:x_n:K}$
3. $\dfrac{exed_{p,n,i}\ :\ C \quad x_1:\cdots:x_p:y_1:\cdots:y_n:K}{C \quad y_i:x_1:\cdots:x_p:K}$
4. $\dfrac{op\ :\ C \quad x_1:\cdots:x_k:K}{C \quad Wert(op(x_k,\ldots,x_1)):K}$ falls op eine k-stellige Standardfunktion ist
5. $\dfrac{if\ g\ :\ C \quad true:K}{C \quad K}$
6. $\dfrac{if\ g\ :\ C \quad false:K}{[] \quad g:K}$
7. $\dfrac{[] \quad g:K}{C \quad K}$ falls $g \rhd C$ eine markierte Befehlsfolge ist (*Sprung zum Code von g*)

[7] Spezialfall für Konstanten e

Beispiel Fakultät Mithilfe dieser Regeln erhalten wir nacheinander folgende Kellerinhalte bei der Auswertung des Kombinatorterms für ($fact$ 3):

[push 3, push id, push fact]	*Regel 1*	[3]	
[push id, push fact]	*Regel 1*	[id,3]	
[push fact]	*Regel 1*	[fact,id,3]	
[]	*Regel 7*	[id,3]	*Sprung zum Code von $fact$*
[push 0, dupl_3, equal, if g, push 1, $\text{exed}_{1,2,1}$]	*Regel 1*	[0,id,3]	
[dupl_3, equal, if g, push 1, $\text{exed}_{1,2,1}$]	*Regel 2*	[3,0,id,3]	
[equal, if g, push 1, $\text{exed}_{1,2,1}$]	*Regel 4*	[false,id,3]	
[if g, push 1, $\text{exed}_{1,2,1}$]	*Regel 5*	[g,id,3]	
[]	*Regel 7*	[id,3]	*Sprung zum Code von g*
[dupl_2, dupl_3, push 1, sub, push h, push fact]	*Regel 2*	[3,id,3]	
[dupl_3, push 1, sub, push h, push fact]	*Regel 2*	[3,3,id,3]	
[push 1, sub, push h, push fact]	*Regel 1*	[1,3,3,id,3]	
[sub, push h, push fact]	*Regel 4*	[2,3,id,3]	
[push h, push fact]	*Regel 1*	[h,2,3,id,3]	
[push fact]	*Regel 1*	[fact,h,2,3,id,3]	
[]	*Regel 7*	[h,2,3,id,3]	*Sprung zum Code von $fact$*
[push 0, dupl_3, equal, if g, push 1, $\text{exed}_{1,2,1}$]	*Regel 1*	[0,h,2,3,id,3]	
[dupl_3, equal, if g, push 1, $\text{exed}_{1,2,1}$]	*Regel 2*	[2,0,h,2,3,id,3]	
[equal, if g, push 1, $\text{exed}_{1,2,1}$]	*Regel 4*	[false,h,2,3,id,3]	
[if g, push 1, $\text{exed}_{1,2,1}$]	*Regel 5*	[g,h,2,3,id,3]	
[]	*Regel 7*	[h,2,3,id,3]	*Sprung zum Code von g*
[dupl_2, dupl_3, push 1, sub, push h, push fact]	*Regel 2*	[2,h,2,3,id,3]	
[dupl_3, push 1, sub, push h, push fact]	*Regel 2*	[2,2,h,2,3,id,3]	
[push 1, sub, push h, push fact]	*Regel 1*	[1,2,2,h,2,3,id,3]	
[sub, push h, push fact]	*Regel 4*	[1,2,h,2,3,id,3]	
[push h, push fact]	*Regel 1*	[h,1,2,h,2,3,id,3]	
[push fact]	*Regel 1*	[fact,h,1,2,h,2,3,id,3]	
[]	*Regel 7*	[h,1,2,h,2,3,id,3]	*Sprung zum Code von $fact$*

[push 0, dupl$_3$, equal, if g, push 1, exed$_{1,2,1}$]	*Regel 1*	[0,h,1,2,h,2,3,id,3]	
[dupl$_3$, equal, if g, push 1, exed$_{1,2,1}$]	*Regel 2*	[1,0,h,1,2,h,2,3,id,3]	
[equal, if g, push 1, exed$_{1,2,1}$]	*Regel 4*	[false,h,1,2,h,2,3,id,3]	
[if g, push 1, exed$_{1,2,1}$]	*Regel 5*	[g,h,1,2,h,2,3,id,3]	
[]	*Regel 7*	[h,1,2,h,2,3,id,3]	*Sprung zum Code von g*
[dupl$_2$, dupl$_3$, push 1, sub, push h, push fact]	*Regel 2*	[1,h,1,2,h,2,3,id,3]	
[dupl$_3$, push 1, sub, push h, push fact]	*Regel 2*	[1,1,h,1,2,h,2,3,id,3]	
[push 1, sub, push h, push fact]	*Regel 1*	[1,1,1,h,1,2,h,2,3,id,3]	
[sub, push h, push fact]	*Regel 4*	[0,1,h,1,2,h,2,3,id,3]	
[push h, push fact]	*Regel 1*	[h,0,1,h,1,2,h,2,3,id,3]	
[push fact]	*Regel 1*	[fact,h,0,1,h,1,2,h,2,3,id,3]	
[]	*Regel 7*	[h,0,1,h,1,2,h,2,3,id,3]	*Sprung zum Code von $fact$*
[push 0, dupl$_3$, equal, if g, push 1, exed$_{1,2,1}$]	*Regel 1*	[0,h,0,1,h,1,2,h,2,3,id,3]	
[dupl$_3$, equal, if g, push 1, exed$_{1,2,1}$]	*Regel 2*	[0,0,h,0,1,h,1,2,h,2,3,id,3]	
[equal, if g, push 1, exed$_{1,2,1}$]	*Regel 4*	[true,h,0,1,h,1,2,h,2,3,id,3]	
[if g, push 1, exed$_{1,2,1}$]	*Regel 5*	[h,0,1,h,1,2,h,2,3,id,3]	
[push 1, exed$_{1,2,1}$]	*Regel 1*	[1,h,0,1,h,1,2,h,2,3,id,3]	
[exed$_{1,2,1}$]	*Regel 3*	[h,1,1,h,1,2,h,2,3,id,3]	
[]	*Regel 7*	[1,1,h,1,2,h,2,3,id,3]	*Sprung zum Code von h*
[mul, exed$_{1,2,1}$]	*Regel 4*	[1,h,1,2,h,2,3,id,3]	
[exed$_{1,2,1}$]	*Regel 3*	[h,1,2,h,2,3,id,3]	
[]	*Regel 7*	[1,2,h,2,3,id,3]	*Sprung zum Code von h*
[mul, exed$_{1,2,1}$]	*Regel 4*	[2,h,2,3,id,3]	
[exed$_{1,2,1}$]	*Regel 3*	[h,2,3,id,3]	
[]	*Regel 7*	[2,3,id,3]	*Sprung zum Code von h*
[mul, exed$_{1,2,1}$]	*Regel 4*	[6,id,3]	
[exed$_{1,2,1}$]	*Regel 3*	[id,6]	
[]	*Regel 7*	[6]	*Sprung zum leeren Code von id*

Insgesamt sind wir bei der Übersetzung funktionaler Programme wie in Kapitel 6 vorgegangen: Mit der Transformation von Haskell-Programmen in λ-Ausdrücke und anschließend in Continuations haben wir eine Funktion $transform : Q \to Q$ definiert, ein Quellprogramm so modifiziert, dass es einerseits im Rahmen der Quellsprache bleibt, andererseits eine dem gewünschten Zielprogramm ähnliche Struktur erhält. Eine weitere Funktion $translate : Q \to Z$ überführt den erzeugten Continuation-Ausdruck in die Zielsprache und ändert dabei kaum die Struktur, wohl aber die zugrundeliegende Sprache.

Kapitel 8

Interpreter funktionaler Sprachen

8.1 Die SECD-Maschine

Interpreter funktionalen Sprachen arbeiten üblicherweise auf der Zwischensprache des λ-Kalküls. Das klassische Verfahren heißt SECD-Maschine[1], wobei "Maschine" im Sinne einer *abstrakten* Maschine zu verstehen ist, d.h. einer Spezifikation, die die zur Darstellung, Übersetzung und Ausführung von Programmen notwendigen Datentypen umfaßt. Die SECD-Maschine wurde schon 1964 von [38] eingeführt. Neuere Darstellungen findet man in [18], Kap. 10, [67], Kap. 13, und [76], Kap. 15.

Zunächst wird ein Übersetzer definiert, der λ-Terme in abstrakter Syntax auf Listen von "Assemblerbefehlen", die zum Teil noch baumstrukturiert sind, abbildet. Die Köpfe von λ-Abstraktionen sind hier auf Variablen beschränkt. Übersetzer und Interpreter werden als Haskell-Programme formuliert. Wir definieren hier nur `Exp` (Ausdrücke der Quellsprache) und `Com` (Ausdrücke der Zielsprache).

```
data Exp = I Int | B Bool | F Fun | Var String | Abs Var Exp | Rec Var Exp |
           App Exp Exp | Tup [Exp] | Ite Exp Exp Exp
data Fun = Add | Sub | Mul | Equ
```

Abs, Rec, App, Tup und Ite sind die Konstruktoren für λ-Abstraktionen, μ-Abstraktionen, Applikationen, Tuplung bzw. Konditionale. Add, Sub, Mul und Equ sind Konstruktoren für Basisfunktionen.

```
data Com = LDI Int | LDB Bool | LDF Fun | LDV String | LDA String [Com] |
           LDR String [Com] | APP | TUP Int | SEL Com Com
```

"LD" steht für "lade", "SEL" für "selektiere" (einen der Zweige eines Konditionals).

```
compile :: Exp -> [Com]
compile (I n)        = [LDI n]
compile (B b)        = [LDB b]
compile (F f)        = [LDF f]
compile (Var x)      = [LDV x]
compile (Abs x e)    = [LDA x (compile e)]
compile (Rec x e)    = [LDR x (compile e)]
compile (App e e')   = compile e' ++ compile e ++ [APP]
```

[1] S = stack, E = environment, C = control, D = dump

```
compile (Tup eL)     = cL ++ [TUP(n)]
                         where compTup eL = (cL,n)
compile (Ite c e e') = compile c ++ [SEL (compile e) (compile e')]

compTup :: [Exp] -> ([Com],Int)
compTup []     = ([],0)
compTup (e:eL) = (compile(e):cL,n+1)
                 where compTup eL = (cL,n)
```

Z.B. lautet die Fakultätsfunktion als Exp-Term wie folgt:

```
Rec ("fact" (Abs "n" (Ite (App (F Equ) (Tup [n,I 0]))
                          (I 1)
                          (App (F Mul) (Tup [n,App fact (App (F Sub) (Tup [n,I 1]))])))))
where fact = Var "fact"
      n = Var "n"
```

compile transformiert diesen Term in die folgende Befehlsliste cL:

```
[LDR "fact" [LDA "n" [LDV "n",LDI 0,TUP 2,LDF Equ,APP,
                      SEL [LDI 1]
                          [LDV "n",LDI 1,TUP 2,LDF Sub,APP,LDV "fact",APP,TUP 2,
                           LDF Mul,APP]]]]
```

Ein Interpreter für solche Befehlslisten bildet den zweiten Teil der SECD-Maschine. Er ist definiert als Funktion

```
eval :: [Val] -> Env -> [Com] -> Dump -> Val
```

Das erste Argument von *eval* ist ein Keller zur Aufnahme von Zwischenwerten, die bei der Auswertung und Entrekursivierung der übersetzten λ-Terme entstehen. Das zweite Argument ist ein *Environment*, das den Programmvariablen ihre jeweils aktuellen Werte zuordnet. Das dritte Argument liefert das jeweilige Restprogramm. Das vierte Argument ist ein **Dump**, d.i. hier eine Liste von Zuständen, die jeweils aus einem Kellerinhalt, einem Environment und einem Restprogramm bestehen. Der Dump wird benötigt, damit nach Ausführung eines rekursiven Aufrufs der vor dessen Ausführung gültige Zustand wiederhergestellt werden kann (s.u. Gleichungen 7 und 8).

Z.B. entspricht $fact(3)$ dem Aufruf

```
eval [] [] (LDI 3:cL++[APP]) []
```

des Interpreters *eval*, wobei das o.a. Zielprogramm für die Fakultätsfunktion ist.

Wir müssen zunächst die Typen Val, Env und Dump definieren:

```
data Val  = VI Int | VB Bool | VF Fun | VT [Val] | Closure Env String [Com]
data Env  = [(String,Val)]
data Dump = [([Val],Env,[Com])]
```

eval ist dann wie folgt definiert:

```
eval s e (LDI n:c) d             = eval (VI n:s) e c d
eval s e (LDB b:c) d            = eval (VB b:s) e c d
eval s e (LDF f:c) d            = eval (VF f:s) e c d
eval s e (LDV x:c) d            = eval (get x e:s) e c d
eval s e (LDA x c:c') d         = eval (Closure e x c:s) e c' d              (1)
eval s e (LDR f [LDA x c]:c') d = eval (g:s) e c' d
                                  where g = Closure ((f,g):e) x c            (2)
eval (VF Add:VI m:VI n:s) e (APP:c) d  = eval (VI (n+m):s) e c d             (3)
eval (VF Sub:VI m:VI n:s) e (APP:c) d  = eval (VI (n-m):s) e c d             (4)
eval (VF Mul:VI m:VI n:s) e (APP:c) d  = eval (VI (n*m):s) e c d             (5)
eval (VF Equ:VI m:VI n:s) e (APP:c) d  = eval (VB (n==m):s) e c d            (6)
eval (Closure e x c:v:s) e' (APP:c') d = eval [] ((x,v):e) c ((s,e',c'):d)   (7)
eval s e (TUP n:c) d                   = eval (VT (take n s):drop n s) e c d
eval (VB true:s) e (SEL c1 c2:c) d     = eval s e (c1++c) d
eval (VB false:s) e (SEL c1 c2:c) d    = eval s e (c2++c) d
eval (v:_) _ [] []                     = v
eval (v:_) _ [] ((s,e,c):d)            = eval (v:s) e c d                    (8)

get :: String -> Env -> Val
get x ((y,v):e) = if x == y then v else get x e
```

Gleichung 1 definiert die Auswertung einer λ-Abstraktion $\lambda x.c$. Es wird ein *Abschluß (closure)* erzeugt und gekellert, in dem neben der gebundenen Variable x und dem Rumpf c das Environment e gespeichert ist. Letzteres macht die Werte globaler Variablen von $\lambda x.c$ verfügbar. Analog wird eine μ-Abstraktion $\mu f.\lambda x.c$ ausgewertet (Gleichung 2). Der erzeugte Abschluß muß hier auch sich selbst unter dem Namen g in sein Environment aufnehmen, damit die rekursiven Aufrufe von f den Rumpf $\lambda x.c$ finden. Die lokale Definition

```
g = Closure ((f,g):e) x c
```

liefert ein Objekt, dass man sich als verkettete Struktur vorstellen sollte: g ist ein Zeiger, der auf einen Baum mit Wurzel *Closure* und Nachfolgern $(f, g) : e$, x und c verweist.

Die Behandlung von Applikationen (Gleichungen 3 bis 7) macht deutlich, dass der SECD-Interpreter Funktionsaufrufe nach der **call-by-value-Strategie** auswertet, d.h. Funktionen nur auf bereits ausgewertete Argumente anwendet. Anders formuliert: *eval* kann nur *strikte* Funktionen auswerten. Eine rekursive Definition einer Funktion f, in der sonst nur strikte Funktionen auftreten, liefert aber immer die überall undefinierte Funktion! Eine nichtstrikte Funktion muss mindestens vorkommen. In der Regel ist das ein Konditional. Deshalb werden Konditionale schon von `compile` anders (eben als nichtstrikte Funktionen) behandelt als die übrigen Basisfunktionen (siehe oben).

Aufgabe Verallgemeinern Sie `compile` und `eval` so, dass λ-Abstraktionen $\lambda p.e$ mit beliebigen Datenmustern p (siehe §6.2) interpretiert werden können!

8.2 Kombinatoren zur Beseitigung von λ-Abstraktionen

Alle λ- und μ-Abstraktionen (siehe §6.2) lassen sich auf Terme reduzieren, die aus folgenden Kombinatoren bestehen:

$$\begin{aligned} I(x) &= x \\ K(x)(y) &= x \\ S(f)(g)(x) &= f(x)(g(x)) \\ B(f)(g)(x) &= f(g(x)) \\ C(f)(x)(y) &= f(y)(x) \\ Y(f) &= f(Y(f)) \end{aligned}$$

Der letzte heißt **Fixpunktkombinator**. Die Gleichung $Y(f) = f(Y(f))$ definiert Y nicht, sondern beschreibt nur die Fixpunkteigenschaft: Y bildet f auf einen Fixpunkt von f ab. Definieren kann man Y als Applikation einer λ-Abstraktion auf sich selbst:

$$Y(f) \;=\; (\lambda x.f(x(x)))(\lambda x.f(x(x))).$$

Wegen der Selbstanwendung von x ist sie allerdings nicht typisierbar! In Haskell lässt sich Y analog zum *closure*-Zyklus in Abschnitt 7.1 mit Hilfe einer lokalen Definition definieren:

```
y f = g where g = f g
```

Die Übersetzung beliebiger λ-Ausdrücke in Kombinatorterme lautet nun wie folgt:

$compile(x)$	$=$	$K(x)$	für alle Variablen und Konstanten x
$compile(\lambda x.t)$	$=$	$compileAbs(x, compile(t))$	
$compile(\mu f.t)$	$=$	$Y(compileAbs(f, t))$	
$compile((t\ u))$	$=$	$(compile(t)\ compile(u))$	
$compileAbs(x, t)$	$=$	$K(t)$	für alle variablenfreien Terme t
$compileAbs(x, x)$	$=$	I	
$compileAbs(x, y)$	$=$	$K(y)$	falls $x \neq y$
$compileAbs(x, (t\ u))$	$=$	$S(compileAbs(x, t))(compileAbs(x, u))$	

Die letzten beiden Gleichungen bewirken, dass die Anzahl der Symbole eines λ-Ausdrucks bei der Übersetzung in einen Kombinatorterm exponentiell wächst. Durch Anwendung der folgenden Gleichungen lässt sich der Platzaufwand verringern:

$$\begin{aligned} S(K(t))(K(u)) &= K(t(u)) \\ S(K(t))(I) &= t \\ S(K(t))(u) &= B(t)(u) \\ S(t)(K(u)) &= C(t)(u) \end{aligned}$$

Zeige, dass diese Gleichungen aus der o.a. Definition der Kombinatoren folgen!

Zeige, dass *compile* tatsächlich alle λ- und μ-Abstraktionen aus einem λ-Ausdruck entfernt!

Kapitel 9

Datenflussanalyse und Optimierung

9.1 Grundbegriffe

Neben Compilationsverfahren im engeren Sinne, die Programme von einer Sprache in eine andere übersetzen, spielen im Übersetzerbau Programmtransformationen eine wichtige Rolle, die statt auf einen Sprachwechsel auf eine Codeverbesserung abzielen. Die Optimierungen sollen den Zeit- und Platzbedarf bei der Ausführung des Programms reduzieren. Sie können sowohl auf der Zwischenrepräsentation (Syntaxbaum) als auch auf dem Zielprogramm vorgenommen werden. Beispiele für erstere haben wir bereits in Kapitel 6 kennengelernt. Man spricht hier auch von **algebraischen Optimierungen** und **strength reduction**. Bei letzteren werden Ausdrücke mit komplexen Funktionen durch einfachere ersetzt wie z.B. x^2 durch $x * x$, $2 * x$ durch $x + x$ oder $x * 2^n$ durch einen Linksshift auf der Binärdarstellung von x. Dazu zählen auch die in Kapitel 6 und in Kapitel 8 von [57] behandelten Programmtransformationen sowie **in-line-Expansion**, bei der Aufrufe von Standardfunktionen direkt durch äquivalente Maschinenbefehlsfolgen ersetzt werden.

Verbreiteter als Syntaxbaumoptimierungen sind jedoch die optimierenden Transformationen des Zielcodes, der, wie wir in Kapitel 5 gesehen haben, im wesentlichen nur noch aus Zuweisungen und (bedingten) Sprüngen besteht. Diese reduzierte Syntax macht es sinnvoll, von der *Baum*repräsentation zur Kontroll- oder Datenfluss*graphen*darstellung der Programme zu wechseln, wenn man die einzelnen Optimierungsmethoden beschreibt. Genaugenommen sind wir in Kapitel 5 schon ein Stück weitergegangen und haben mit expliziter Adressierung und Registerzuweisung bereits wesentliche Komponenten der Codeerzeugung durchgeführt, die in der Regel erst nach den Zielcodeoptimierungen erfolgt. Mit anderen Worten: Das Flussgraphenmodell basiert auf einer etwas abstrakteren Programmsyntax als der in Kapitel 5 als Zielsprache verwendeten Assemblersprache.

Optimierungsalgorithmen versuchen aufwändige Codestücke zu erkennen und zu vereinfachen, was oft selbst recht aufwändig ist und daher die Frage aufkommen lässt, ob man letztere nicht einsparen könnte, indem man Programmierer zu mehr Disziplin erzieht, so dass sie von vornherein weitgehend optimalen Code produzieren. Diese Überlegung mag auf Quellcodeoptimierung zutreffen, Zielcode hingegen wird vom Compiler erzeugt, und zwar nach allgemeinen Regeln, die es nicht erlauben, die Übersetzung von z.B. rekursiven Funktionen so zu gestalten, dass in jedem Einzelfall *unmittelbar* optimaler Code entsteht. Auf die eigentliche Übersetzung folgende Optimierungsschritte sind deshalb unvermeidlich.

Jede Optimierungsmethode reduziert die Anzahl der Vorkommen einer bestimmten Klasse von Programmkomponenten, wie Schleifenvariablen, *tote Variablen*, gleiche Teilausdrücke, überflüssige Zuweisungen u.ä. Sind die Komponenten prozedurübergreifend, spricht man von **interprozeduraler Optimierung**. Andernfalls ist sie **intraprozedural**.

Definition 9.1.1 Ein **Flussgraph** ist ein Quadrupel $G = (I, N, E, lab)$ mit $E \subseteq (I + N) \times N$ und $lab : E \to M$. G und entspricht damit einem kantenmarkierten gerichteten Graphen mit Knotenmenge $I + N$, Kantenmenge E und Markierungsfunktion lab. Die Knoten von I heißen **Eingangsknoten**. Sie haben offenbar keine einlaufenden Kanten.

Eine Analyse von G basiert auf einer Interpretation der Kantenmarkierungen von G als Endofunktionen auf einem vollständigen Verband:

Eine Menge A ist **halbgeordnet**, wenn es eine **Halbordnung**, also eine reflexive, transitive und antisymmetrische binäre Relation $\leq$ auf A gibt. Eine halbgeordnete Menge A ist ein **vollständiger Verband**, wenn für jede Teilmenge B von A ein Suprema $\bigsqcup B$ hat, das zu A gehört.

Damit enthält A auch ein Infimum $\bigsqcap B$ von B, nämlich $\bigsqcup\{a \in A \mid \forall\, b \in B : a \leq b\}$, sowie ein kleinstes Element $\bot$ und ein größtes Element $\top$, nämlich $\bigsqcup \emptyset = \bigsqcap A$ bzw. $\bigsqcup A = \bigsqcap \emptyset$.

Seien $A_1, \ldots, A_n$ halbgeordnete Menge. Die Halbordnungen auf $A_1, \ldots, A_n$ lassen sich wie folgt auf das Produkt $A_1 \times \cdots \times A_n$ fortsetzen: Für alle $(a_1, \ldots, a_n), (b_1, \ldots, b_n) \in A_1 \times \cdots \times A_n$,

$$(a_1, \ldots, a_n) \leq (b_1, \ldots, b_n) \quad \Longleftrightarrow_{def} \quad \forall\, 1 \leq i \leq n : a_i \leq b_i.$$

Seien $A_1, \ldots, A_n$ vollständige Verbände. Dann ist auch $A_1 \times \cdots \times A_n$ ein solcher: Für alle $B \subseteq A_1 \times \cdots \times A_n$,

$$\bigsqcup B \quad =_{def} \quad (\bigsqcup\{a_1 \mid \exists\, a_2, \ldots, a_n \in A : (a_1, \ldots, a_n) \in B\}, \ldots, \bigsqcup\{a_n \mid \exists\, a_1, \ldots, a_{n-1} \in A : (a_1, \ldots, a_n) \in B\}).$$

Sei A eine halbgeordnete Menge. Die Halbordnung auf A lässt sich wie folgt auf die Menge A^X der Funktionen von X nach A fortsetzen: Für alle $f, g : X \to A$,

$$f \leq g \quad \Longleftrightarrow_{def} \quad \forall\, x \in X : f(x) \leq g(x).$$

Sei X eine beliebige Menge und A ein vollständiger Verband. Dann ist auch A^X ein solcher: Für alle $B \subseteq A^X$ und $x \in X$,

$$(\bigsqcup B)(x) \quad =_{def} \quad \bigsqcup\{f(x) \mid f \in B\}.$$

Seien A, B halbgeordnete Mengen. Eine Funktion $F : A \to B$ ist **monoton**, wenn für alle $a, b \in A$ gilt:

$$a \leq b \;\Rightarrow\; F(a) \leq F(b).$$

Z.B. sind die Funktionen $\sqcup, \sqcap : A \times A \to A$, die jedem Paar $(a, b) \in A \times A$ das Supremum bzw. Infimum von $\{a, b\}$ zuordnet, monoton. Außerdem ist die Komposition $G \circ F : A \to C$ zweier monotoner Funktionen $F : A \to B$ und $G : B \to C$ monoton. Auch das Produkt

$$\begin{array}{rcl} F_1 \times \cdots \times F_n : A_1 \times \cdots \times A_n & \to & B_1 \times \cdots \times B_n \\ (a_1, \ldots, a_n) & \to & (F_1(a_1), \ldots, F_n(a_n)) \end{array}$$

von n monotonen Funktionen $F_1 : A_1 \to B_1, \ldots, F_n : A_n \to B_n$ und die Potenz

$$\begin{array}{rcl} F^X : A^X & \to & B^X \\ f & \to & F \circ f \end{array}$$

einer monotonen Funktion $F : A \to B$ sind monoton.

Eine Teilmenge $B = \{a_i \mid i < \omega\}$ von A heißt **Kette** bzw. **Cokette** von A, wenn für alle $i \in \mathbb{N}$ $a_i \leq a_{i+1}$ bzw. $a_{i+1} \leq a_i$ gilt. ω ist die kleinste unendliche Ordinalzahl. Sie entspricht der Kardinalität (Mächtigkeit) von $\mathbb{N}$. Man schreibt deshalb oft $i < \omega$ anstelle von $i \in \mathbb{N}$.

Seien A, B vollständige Verbände und $F : A \to B$ monoton. Dann bilden die Iterationen $F^i(\bot)$, $i < \omega$, eine Kette und die Iterationen $F^i(\top)$, $i < \omega$, eine Cokette. Ihr Supremum bzw. Infimum

$$F^\infty =_{def} \bigsqcup_{i<\omega} F^i(\bot) \quad \text{bzw.} \quad F_\infty =_{def} \bigsqcap_{i<\omega} F^i(\top)$$

heißt **oberer** bzw. **unterer Kleene-Abschluss**.

Sei A eine halbgeordnete Menge und $F : A \to A$. $a \in A$ heißt **F-abgeschlossen** oder **F-reduktiv**, wenn $f(a) \leq a$. $a \in A$ heißt **F-dicht** oder **F-extensiv**, wenn $a \leq f(a)$.

Lemma 9.1.2 Sei $F : A \to A$ monoton.

(1) Ist die Kette $\{F^i(\bot) \mid i < \omega\}$ endlich, gibt es also $n \in \mathbb{N}$ mit $F^\infty = F^n(\bot)$, dann ist F^∞ F-abgeschlossen.

(2) Ist die Cokette $\{F^i(\top) \mid i < \omega\}$ endlich, gibt es also $n \in \mathbb{N}$ mit $F_\infty = F^n(\top)$, dann ist F_∞ F-dicht.

Beweis. (1) $F(F^\infty) = F(F^n(\bot)) = F^{n+1}(\bot) \leq \bigsqcup_{i<\omega} F^i(\bot) = F^\infty$.

(2) $F_\infty = \bigsqcap_{i<\omega} F^i(\top) \leq F^{n+1}(\top) = F(F^n(\top)) = F(F_\infty)$. ❑

Satz 9.1.3 (Fixpunktsatz von Knaster und Tarski) Sei $F : A \to A$ monoton.

(1) $lfp(F) =_{def} \bigsqcap\{a \in A \mid a \text{ ist } F\text{-abgeschlossen}\}$ ist der kleinste Fixpunkt von F.

(2) $F^\infty \leq lfp(F)$.

(3) Ist F^∞ F-abgeschlossen, dann ist $lfp(F) \leq F^\infty$.

(4) $gfp(F) =_{def} \bigsqcup\{a \in A \mid a \text{ ist } F\text{-dicht}\}$ ist der größte Fixpunkt von F.

(5) $gfp(F) \leq F_\infty$.

(6) If F_∞ is F-dicht, dann ist $F_\infty \leq gfp(F)$.

Beweis. (1) Sei a F-abgeschlossen. Dann gilt $lfp(F) \leq a$, also $F(lfp(F)) \leq F(a) \leq a$, weil F monoton ist, d.h. $F(lfp(F))$ ist eine untere Schranke aller F-abgeschlossenen Elemente von A.

Demnach gilt (7) $F(lfp(F)) \leq \bigsqcap\{a \in A \mid a \text{ ist } F\text{-abgeschlossen}\} = lfp(F)$, d.h. $F(lfp(F))$ ist F-abgeschlossen. Außerdem gilt (8) $lfp(F) = \bigsqcap\{a \in A \mid a \text{ ist } F\text{-abgeschlossen}\} \leq F(lfp(F))$. Aus (7) und (8) folgt $F(lfp(F)) = lfp(F)$.

Sei a ein Fixpunkt von F. Dann ist a F-abgeschlossen und wir erhalten $lfp(F) \leq a$, d.h. $lfp(F)$ ist der *kleinste* Fixpunkt von F.

(2) Durch Induktion über n erhält man $F^n(\bot) \leq lfp(F)$: $F^0(\bot) = \bot \leq lfp(F)$ und

$$F^{n+1}(\bot) = F(F^n(\bot)) \overset{ind.\ hyp.}{\leq} F(lfp(F)) \overset{(1)}{=} lfp(F),$$

weil F monoton ist. Daraus folgt $F^\infty = \bigsqcup_{n<\omega} F^n(\bot) \leq lfp(F)$.

(3) Let F^∞ F-abgeschlossen. Dann gilt $lfp(F) = \bigsqcap\{a \in A \mid a \text{ ist } F\text{-abgeschlossen}\} \leq F^\infty$.

(4)-(6) kann analog gezeigt werden. ❑

Korollar 9.1.4 Sei $F : A \to A$ monoton, $B = \{F^i(\bot) \mid i < \omega\}$ und $C = \{F^i(\top) \mid i < \omega\}$.

(1) Ist B endlich, dann ist das (bzgl. $\leq$) größte Element $F^n(\bot)$ von B der kleinste Fixpunkt von F.

(2) Gibt es $n \in \mathbb{N}$ derart, dass $F^n(\bot)$ ein Fixpunkt von F ist, dann ist B endlich und $F^n(\bot)$ das größte Element von B.

(3) Ist C endlich, dann ist das (bzgl. $\leq$) kleinste Element $F^n(\top)$ von C der größte Fixpunkt von F.

(4) Gibt es $n \in \mathbb{N}$ derart, dass $F^n(\top)$ ein Fixpunkt von F ist, dann ist C endlich und $F^n(\top)$ das kleinste Element von C.

Beweis. (1) folgt direkt aus Lemma 9.1.2 und Satz 9.1.3 (1)-(3).

(2) Sei $F^n(\bot)$ ein Fixpunkt von F. Durch Induktion über i erhält man $F^{n+i}(\bot) = F^n(\bot)$ für alle $i \in \mathbb{N}$: Für $i = 0$ gilt die Behauptung trivialerweise. Für $i > 0$ erhält man

$$F^{n+i}(\bot) = F(F^{n+i-1}(\bot)) \overset{ind.\ hyp.}{=} F(F^n(\bot)) = F^n(\bot).$$

Also ist B endlich und $F^n(\bot)$ das größte Element von B.

(3) folgt direkt aus Lemma 9.1.2 und Satz 9.1.3 (4)-(6).

(4) kann analog zu (2) gezeigt werden. ❑

Sei $G = (I, N, E, lab)$ ein Flussgraph, A ein vollständiger Verband, mit dessen Elementen die Knoten von G bewertet werden sollen, und $\delta : M \to A^A$ eine Funktion, die jede Kantenmarkierung als Funktionen auf A interpretiert. Da *delta* eingesetzt wird um den jeweils aktuellen Wert des Quellknotens n einer Kante (n, n') in einen neuen Wert am Zielknoten n' zu transformieren, werden die Bilder von δ auch **Transferfunktionen** genannt.

Die folgenden **Schrittfunktionen** $join_{G,\delta}, meet_{G,\delta} : A^I \to (A^N \to A^N)$ modifizieren in Abhängigkeit einer Bewertung g der Eingangsknoten von G jede Bewertung val der restlichen Knoten: Für alle $g \in A^I$, $h \in A^N$ und $n \in N$,

$$\begin{aligned} join_{G,\delta}(g)(h)(n) &=_{def} h(n) \sqcup \bigsqcup_{(n',n)\in E} \delta(lab(n',n))([g,h](n')), \\ meet_{G,\delta}(g)(h)(n) &=_{def} h(n) \sqcap \bigsqcap_{(n',n)\in E} \delta(lab(n',n))([g,h](n')). \end{aligned}$$

Ein Fixpunkt von $join_{G,\delta}(g)$ bzw. $meet_{G,\delta}(g)$ heißt **join-** bzw. **meet-Lösung von** (G, g) **bzgl.** δ. Nach Satz 9.1.3 ist

- $lfp(join_{G,\delta}(g))$ eine join-Lösung von (G, g) bzgl. δ, falls $join_{G,\delta}(g)$ monoton ist; (9.1)
- $gfp(meet_{G,\delta}(g))$ eine meet-Lösung von (G, g) bzgl. δ, falls $meet_{G,\delta}(g)$ monoton ist; (9.2)

Aus der Monotonie von $\sqcup$ und $\sqcap$ und der Abgeschlossenheit von Monotonie unter Produkt- und Potenzbildung folgt, dass $join_{G,\delta}(g)$ und $meet_{G,\delta}(g)$ monoton sind, wenn alle Bilder von δ monoton sind. Demnach können (9.1) und (9.2) wie folgt verschärft werden:

Korollar 9.1.5 Sei $G = (I, N, E, lab)$ ein Flussgraph, A ein vollständiger Verband, $g \in A^I$ und $\delta : M \to A^A$ derart, dass für alle $m \in M$, $I(m) : A \to A$ monoton ist. Dann ist $lfp(join_{G,\delta}(g))$ eine join-Lösung und $gfp(meet_{G,\delta}(g))$ eine meet-Lösung von (G, g) bzgl. δ. ❑

Bleibt die Frage, wie der kleinste Fixpunkt von $join_{G,\delta}(g)$ und der größte Fixpunkt von $meet_{G,\delta}(g)$ berechnet werden.

Nach Korollar 9.1.4 (1) ergibt sich $lfp(join_{G,\delta}(g))$ aus den auf $\bot$ angewendeten Iterationen von $join_{G,\delta}(g)$, während man $gfp(meet_{G,\delta}(g))$ nach Korollar 9.1.4 (3) aus den auf $\top$ angewendeten Iterationen von $meet_{G,\delta}(g)$ erhält. Gefunden ist der jeweilige Fixpunkt nach Korollar 9.1.4 (2) bzw. (4), sobald eine Iteration erreicht ist, deren Ausführung die in der vorherigen Iteration berechneten Knotenwerte nicht mehr ändert.

Um das Erreichen eines solchen Endzustands zu erkennen, wird eine vor jeder Iteration mit *False* initialisierte Boolesche Variable innerhalb der Schleife auf *True* gesetzt, sobald dort ein Knoten n besucht wird, dessen vorheriger Wert vom neu berechneten verschieden ist. Das ist nach folgendem Lemma genau dann der Fall, wenn eine in n einlaufende Kante (n', n) Bedingung (9.5) bzw. (9.6) verletzt:

Lemma 9.1.6 Sei $g \in A^I$. $h \in A^N$ ist genau dann eine join-Lösung von (G,g) bzgl. δ, wenn für alle $(n',n) \in E$

$$\delta(lab(n',n))([g,h](n')) \leq h(n) \tag{9.5}$$

gilt. $h \in A^N$ ist genau dann eine meet-Lösung von (G,g) bzgl. δ, wenn für alle $(n',n) \in E$

$$h(n) \leq \delta(lab(n',n))([g,h](n')) \tag{9.6}$$

gilt.

Beweis. Wir zeigen, dass g genau dann ein Fixpunkt von $join_{G,\delta}(g)$ ist, wenn (9.5) für alle $(n',n) \in E$ gilt. Die andere Behauptung lässt sich analog beweisen.

"$\Rightarrow$": Sei h ein Fixpunkt von $join_{G,\delta}(g)$ und $(n',n) \in E$. Dann gilt (9.5):

$$\begin{aligned}&\delta(lab(n',n))([g,h](n')) \leq \textstyle\bigsqcup_{(k,n)\in E} \delta(lab(k,n))([g,h](k)) \leq h(n) \sqcup \bigsqcup_{(k,n)\in E} \delta(lab(k,n))([g,h](k))\\ &= join_{G,\delta}(g)(h)(n) = h(n).\end{aligned}$$

"$\Leftarrow$": (9.5) gelte für alle $(n',n) \in E$. Daraus folgt

$$join_{G,\delta}(g)(h)(n) = h(n) \sqcup \bigsqcup_{(n',n)\in E} \delta(lab(n',n))([g,h](n')) \overset{(9.5)}{\leq} h(n) \sqcup h(n) = h(n) \leq join_{G,\delta}(g)(h)(n)$$

für alle $n \in N$, also $join_{G,\delta}(g)(h) = h$. ❏

Zusammenfassend erhalten wir den folgenden iterativen Algorithmus (als funktionales Programm; größtenteils in Haskell-Notation) zur Ermittlung des kleinsten bzw. größten Fixpunkts von $\Phi = join_{G,\delta}(g)$ bzw. $\Psi = meet_{G,\delta}(g)$ – sofern $\{\Phi^i(\bot)\}_{i<\omega}$ bzw. $\{\Psi^i(\top)\}_{i<\omega}$ endlich ist.

Algorithmus zur Fixpunktbestimmung von Interpretationen kantenmarkierter Flussgraphen

$I = \{in_1,\ldots,in_k\}$ und $N = \{n_1,\ldots,n_m\}$ werden als natürliche Zahlen dargestellt: $i \in \mathbb{N}$ mit $1 \leq i \leq k$ repräsentiert den Eingangsknoten in_1, $j \in \mathbb{N}$ mit $k < j \leq k+m$ den Knoten n_{j-k}. Eine Knotenbewertung $val \in A^{I+N}$ wird als Liste (oder Feld) mit $(k+m)$ Elementen aus A dargestellt: Für alle $1 \leq i \leq k+m$ steht $val(i)$ an der i-ten Stelle der Liste.

$$\begin{aligned}
lfp, gfp &:: A^k \to A^{k+m}\\
lfp(s) &= s +\!\!+ loop_1[\bot,\ldots,\bot]\\
gfp(s) &= s +\!\!+ loop_2[\top,\ldots,\top]\\
loop_i &:: A^m \to A^m\\
loop_i(val) &= \mathit{if}\ or(bs)\ \mathit{then}\ loop_i(val')\ \mathit{else}\ val\ \textbf{where}\\
&\quad (val', bs) = unzip(map(h)[1,\ldots,m])\\
&\quad h :: [k+1,\ldots,k+m] \to (A \times Bool)\\
&\quad h(n) = fold1(f_i)(val!!(n-1), False)[n' \mid n' \leftarrow [1,\ldots,k+m],\ (n',n) \in E]\ \textbf{where}\\
&\qquad f_i :: (A \times Bool) \to (N \to (A \times Bool))\\
&\qquad f_1(a,b)(n') = \mathit{if}\ a' \leq a\ \mathit{then}\ (a,b)\ \mathit{else}\ (a \sqcup a', True)\ \textbf{where}\\
&\qquad\qquad a' = \delta(lab(n',n))(val!!(n'-1))\\
&\qquad f_2(a,b)(n') = \mathit{if}\ a \leq a'\ \mathit{then}\ (a,b)\ \mathit{else}\ (a \sqcap a', True)\ \textbf{where}\\
&\qquad\qquad a' = \delta(lab(n',n))(val!!(n'-1))
\end{aligned}$$

Die hier berechneten Knotenbewertungen werden in der Datenflussanalyse-Literatur als **MFP-Lösungen** (MFP = maximal fixpoint) bezeichnet (siehe z.B. [51], §8.2). Wie der obige Algorithmus zeigt, werden sie *lokal*

berechnet, insofern als sich der Wert eines einzelnen Knotens – über die Interpretation ein- bzw. auslaufender Kanten – nur aus den Werten seiner direkten Vorgänger bzw. Nachfolger ergibt.

Demgegenüber werden bei einer *globalen* Berechnung des Wertes eines Knotens n nicht nur dessen ein- bzw. auslaufende Kanten, sondern alle Wege vom Eingangsknoten nach n bzw. von n in den Ausgangsknoten berücksichtigt.

Formal ist ein **Weg von** G eine endliche Folge $w = ((n_1, n_2), (n_2, n_3), \ldots, (n_{k-1}, n_k)) \in E^+$. Die Knoten n_1 und n_k bezeichnen wir mit $\mathbf{src}(w)$ (*source of* w) , bzw. $\mathbf{trg}(w)$ (*target of* w). $\mathbf{Path}(G)$ bezeichnet die Menge der Wege von G. $Path(G)$ ist genau dann endlich, wenn G keinen Zyklus, d.h. keinen Weg w mit $src(w) = trg(w)$, enthält.

Die Transferfunktionen $\delta(m) : A \to A$, $m \in M$, werden von einzelnen Markierungen auf Markierungsfolgen fortgesetzt:

$$\begin{array}{rcl} \delta^+ : M^+ & \to & A^A \\ (m_1, \ldots, m_k) & \mapsto & \delta(m_k) \circ \cdots \circ \delta(m_1). \end{array}$$

Angewendet wird δ^* auf Markierungsfolgen, die aus den Kantenmarkierungen eines Weges entstehen. Dazu setzen wir die Markierungsfunktion $lab : E \to N$ von G auf Wege fort:

$$\begin{array}{rcl} lab^+ : Path(G) & \to & M^+ \\ ((n_1, n_2), (n_2, n_3), \ldots, (n_{k-1}, n_k)) & \mapsto & (lab(n_1, n_2), lab(n_2, n_3), \ldots, lab(n_{k-1}, n_k)) \end{array}$$

Alternativ zu lokalen MFP-Lösungen lassen sich mit folgenden Funktionen globale **MOP-Lösungen** berechnen:

$$\begin{array}{rcl} joinMOP : A^I & \to & A^N \\ g & \to & \lambda n. \bigsqcup \{\delta^+(lab^+(w))(g(src(w))) \mid w \in Path(G),\ src(w) \in I,\ trg(w) = n\} \\ meetMOP : A^I & \to & A^N \\ g & \to & \lambda n. \bigsqcap \{\delta^+(lab^+(w))(g(src(w))) \mid w \in Path(G),\ src(w) \in I,\ trg(w) = n\} \end{array}$$

Seien A, B vollständige Verbände. $F : A \to B$ heißt **stetig** oder **distributiv**, wenn für alle nichtleeren[1] Teilmengen C von A $F(\bigsqcup C) = \bigsqcup F(C)$ und $F(\bigsqcap C) = \bigsqcap F(C)$ gilt.

Stetige Funktionen sind monoton: Sei $a \leq b$. Dann ist $F(b) = F(a \sqcup b) = F(a) \sqcup F(b)$, also $F(a) \leq F(b)$.

Satz 9.1.7 (Fixpunktsatz von Kleene) Sei A ein vollständiger Verband und $F : A \to A$ stetig. Dann ist F^∞ der kleinste und F_∞ der größte Fixpunkt von F.

Beweis. Da F stetig ist, gilt $F(F^\infty) = F(\bigsqcup_{i<\omega} F^i(\bot)) = \bigsqcup_{i<\omega} F(F^i(\bot)) = F^\infty$. Also ist F^∞ nach Satz 9.1.3 (1)-(3) der kleinste Fixpunkt von F. Da F stetig ist, gilt auch $F(F_\infty) = F(\bigsqcap_{i<\omega} F^i(\top)) = \bigsqcap_{i<\omega} F(F^i(\top)) = F_\infty$. Also ist F^∞ nach Satz 9.1.3 (4)-(6) der größte Fixpunkt von F. ❑

Satz 9.1.8 Sei $G = (I, N, E, lab : E \to M)$ ein Flussgraph, A ein vollständiger Verband und $\delta : M \to A^A$.

(i) Sei $\delta(m)$ für alle $m \in M$ monoton. Dann gilt $joinMOP \leq lfp \circ join_{G,\delta}$ und $gfp \circ meet_{G,\delta} \leq meetMOP$.

(ii) Sei $\delta(m)$ für alle $m \in M$ stetig. Dann ist $joinMOP$ ein Fixpunkt von $join_{G,\delta}$ und $meetMOP$ ein Fixpunkt von $meet_{G,\delta}$.

(iii) Sei $\delta(m)$ für alle $m \in M$ stetig. Dann gilt $joinMOP = lfp \circ join_{G,\delta}$ und $gfp \circ meet_{G,\delta} = meetMOP$, d.h. die globalen stimmen mit den lokalen Lösungen überein.

Beweisskizze.

(i) Durch Induktion über i lässt sich zeigen, dass für alle $i \in \mathbb{N}$, $w \in Path(G) \cap E^i$ und $g \in A^I$ mit $src(w) \in I$

[1] Warum die Einschränkung auf *nichtleere* Teilmengen?

die folgenden Ungleichungen gelten:

$$\delta^+(lab^+(w))(g(src(w))) \leq join_{G,\delta}(g)^{i+1}(\bot), \tag{9.7}$$
$$meet_{G,\delta}(g)^{i+1}(\top) \leq \delta^+(lab^+(w))(g(src(w))). \tag{9.8}$$

Daraus folgt für alle $w \in Path(G)$ und $g \in A^I$ mit $src(w) \in I$:

$$\delta^+(lab^+(w))(g(src(w))) \overset{(9.7)}{\leq} \bigsqcup_{i<\omega} join_{G,\delta}(g)^i(\bot) = join_{G,\delta}(g)^\infty \overset{\text{Satz 9.1.7}}{=} lfp(join_{G,\delta}(g)), \tag{9.9}$$

$$gfp(meet_{G,\delta}(g)) \overset{\text{Satz 9.1.7}}{=} meet_{G,\delta}(g)_\infty = \bigsqcap_{i<\omega} meet_{G,\delta}(g)^i(\top) \overset{(9.8)}{\leq} \delta^+(lab^+(w))(g(src(w))), \tag{9.10}$$

also für alle $n \in N$,

$$\begin{aligned} joinMOP(g)(n) &= \bigsqcup\{\delta^+(lab^+(w))(g(src(w))) \mid w \in Path(G),\ src(w) \in I,\ trg(w) = n\} \\ &\overset{(9.9)}{\leq} lfp(join_{G,\delta}(g))(n), \\ gfp(meet_{G,\delta}(g))(n) &\overset{(9.10)}{\leq} \bigsqcap\{\delta^+(lab^+(w))(g(src(w))) \mid w \in Path(G),\ src(w) = in,\ trg(w) = n\} \\ &= meetMOP(g)(n). \end{aligned}$$

(ii) ???

(iii) folgt direkt aus (i), (ii) und der Monotonie stetiger Funktionen. ❏

9.2 Datenflussaufgaben

9.2.1 Dominanzrelation

Sei $G = (I, N, E \subseteq (I + N) \times N, lab : E \to M)$ ein Flussgraph. $k \in N$ **dominiert** $n \in N$, wenn k auf jedem Weg von einem Eingangsknoten nach n liegt. Mit dieser Information kann man die Reduzierbarkeit eines Flussgraphen überprüfen:

Definition 9.2.1.1 Ein Flussgraph ist **reduzierbar**, wenn jeder Zyklus eine Kante enthält, deren Zielknoten den Quellknoten dominiert.

Ein Flussgraph ist genau dann reduzierbar, wenn er keine Schleife enthält, in die von ausserhalb hineingesprungen werden kann. Programme, deren Sprunganweisungen der Compiler ausschließlich aus for-, while- oder repeat-Schleifen erzeugt hat, liefern reduzierbare Flussgraphen. Beispielsweise macht die Kante e den Graphen in Fig. 9.1 zu einem nicht reduzierbaren Flussgraphen: das Ziel z der Kante e' dominiert die Quelle q nicht!

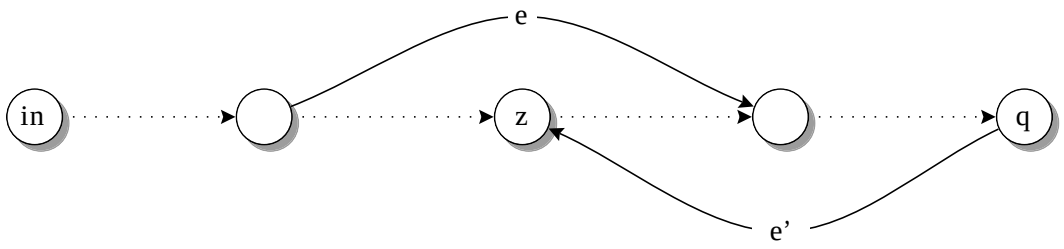


Figure 9.1. *Ein nicht reduzierbarer Flussgraph.*

Die Dominanzrelation erhält man aus einer größten **meet-Lösung** von $(G, \lambda n.\{n\})$. Die Kantenmenge E von G stimmt hier mit der Markierungsmenge M überein. Dementsprechend ist $lab : E \to M$ die Identität auf E. Der

Wertetebereich A wird als Potenzmenge von $I+N$ definiert, $\leq$ als Mengeninklusion, $\sqcap$ als Mengendurchschnitt und $\delta : M \to A^A$ wie folgt: Für alle $(n', n) \in E$ und $N' \subseteq I+N$, $\delta(n', n)(N') = \{n\} \cup N'$. Wie man sofort sieht, ist $\delta(n', n)$ monoton.

Sei $g = \lambda n.\{n\} : I \to A$. $F = meet_{G,\delta}(g) : A^N \to A^N$ ist wie folgt definiert: Für alle $h : N \to A$ und $n \in N$,

$$F(h)(n) \quad = \quad h(n) \sqcap \bigsqcap_{(n',n)\in E}(\{n\} \cup [g, h](n')).$$

Da $I + N$ endlich ist, sind auch A, A^N und die Cokette $\{F^i(\top)\}_{i<\omega}$ endlich. Folglich ist der größte Fixpunkt von F mit dem obigen Algorithmus berechenbar.

Nach Lemma 9.1.6 stimmt $gfp(F)$ mit der größten Funktion $doms : N \to A$ überein, die folgende Bedingung erfüllt:

$$\text{Für alle } (n', n) \in E,\ doms(n) \subseteq \{n\} \cup doms(n'). \tag{9.11}$$

(9.11) ist äquivalent zu:

$$doms(n) \subseteq \{n\} \cup \bigcap_{(n',n)\in E} doms(n'). \tag{9.12}$$

Also ist $doms$ ist die funktionale Version der größten Relation $R \subseteq N \times (I+N)$, die folgende Implikation erfüllt:

$$(n, k) \in R \quad \Rightarrow \quad n = k \ \vee \ \forall\, (n', n) \in E : (n', k) \in R. \tag{9.13}$$

(9.13) charakterisiert die Dominanzrelation: $(n, k) \in R \Leftrightarrow k$ dominiert n. Also ist $doms(n)$ die Menge der n dominierenden Knoten. ❑

Sei $\Sigma = (S, F, P)$ eine Signatur, V eine endliche S-sortige Variablenmenge (siehe [58]) und M_Σ die Menge aller Zuweisungen $x := t$ von Σ-Termen über V an Elemente von V sowie aller aussagenlogischen Σ-Formeln über V (siehe [58]). Für alle Σ-Terme und aussagenlogischen Σ-Formeln φ über V bezeichnet $free(\varphi)$ die Menge der freien Variablen von φ vorkommenden Variablen. Ist $M = M_\Sigma$, dann nennen wir

$$G = (I, N, E \subseteq (I + N) \times N, lab : E \to M)$$

einen (lokalen) **Programmflussgraphen über** Σ.

Einige der folgenden Datenflussanalysen durchlaufen Programmflussgraphen in umgekehrter Richtung, d.h. Eingangsknoten sind nicht mit Anfangswerten, sondern mit gewünschten Ergebnissen der Programmausführung belegt, während jede Transferfunktion aus dem Wert am *Ziel* einer Kante einen Wert an deren *Quelle* berechnet. Um die Bestimmung kleinster bzw. größter Fixpunkte an diese Richtungsänderung anzupassen, muss am obigen Algorithmus nichts geändert werden. Man braucht nur die Pfeile im Graphen umzukehren, so dass auch im Falle sog. **Rückwärtstraversierungen** jede Transferfunktion einer Kante auf den jeweiligen Wert an deren *Quelle* angewendet wird.

9.2.2 Tote und lebendige Variablen

Definition 9.2.2.1 Eine Programmvariable x ist **tot** am Knoten n, wenn sie vor der nächsten Zuweisung an x nicht mehr benutzt wird. Andernfalls ist sie bei n **lebendig**.

Ist x bei n tot, dann kann ein x zur Verfügung gestelltes Register zwischen n und späteren Zuweisungen an x anderweitig verwendet werden. Außerdem lassen sich mit dieser Information manche redundanten Zuweisungen erkennen (siehe §9.3).

Nachdem die Pfeile von G zum Zwecke der Rückwärtstraversierung umgedreht wurden (s.o.), erhält man die Mengen toter Variablen an Knoten von G aus einer größten **meet-Lösung** von $(G, \lambda n \in I.V)$ und die Mengen lebendiger Variablen aus einer kleinsten **join-Lösung** von $(G, \lambda n \in I.\emptyset)$. Der Wertebereich A wird

als Potenzmenge von V definiert, $\leq$ als Mengeninklusion, $\sqcup$ als Mengenvereinigung, $\sqcap$ als Mengendurchschnitt und $\delta, \gamma : M \to A^A$ wie folgt: Für alle $x \in V$, Σ-Terme t über V, aussagenlogischen Σ-Formeln φ über V und $V' \subseteq V$,

$$\begin{aligned} \delta(x := t)(V') &= (V' \cup \{x\}) \setminus \mathit{free}(t), \\ \delta(\varphi)(V') &= V' \setminus \mathit{free}(\varphi), \\ \gamma(x := t)(V') &= (V' \setminus \{x\}) \cup \mathit{free}(t), \\ \gamma(\varphi)(V') &= V' \cup \mathit{free}(\varphi). \end{aligned}$$

Wie man sofort sieht, sind für alle $m \in M$ $\delta_{dead}(m)$ und $\delta_{live}(m)$ monoton.

Sei $f = \lambda n \in I.V$ und $g : \lambda n \in I.\emptyset$. $F = meet_{G,\delta}(f)$ und $G = join_{G,\gamma}(g) : A^N \to A^N$ sind wie folgt definiert: Für alle $h : N \to A$ und $n \in N$,

$$\begin{aligned} F(h)(n) &= h(n) \cap \textstyle\bigcap_{(n',n)\in E} \delta(lab(n',n))([g,h](n')), \\ G(h)(n) &= h(n) \cup \textstyle\bigcup_{(n',n)\in E} \gamma(lab(n',n))([g,h](n')). \end{aligned}$$

Da $I + N$ endlich ist, sind auch die Cokette $\{F^i(\top)\}_{i<\omega}$ und die Kette $\{G^i(\bot)\}_{i<\omega}$ endlich. Folglich sind der größte Fixpunkt von F und der kleinste Fixpunkt von G mit dem obigen Algorithmus berechenbar.

Nach Lemma 9.1.6 stimmt $\mathit{gfp}(F)$ mit der größten Funktion $\mathit{dead} : N \to A$ überein, die folgende Bedingung erfüllt:

$$\text{Für alle } (n',n) \in E,\ \mathit{dead}(n) \subseteq \begin{cases} (\mathit{dead}(n') \cup \{x\}) \setminus \mathit{free}(t) & \text{falls } lab(n',n) = (x := t), \\ \mathit{dead}(n') \setminus \mathit{free}(\varphi) & \text{falls } lab(n',n) = \varphi \in Fo_\Sigma(V). \end{cases} \tag{9.14}$$

Ebenfalls nach Lemma 9.1.6 stimmt $\mathit{lfp}(G)$ mit der kleinsten Funktion $\mathit{live} : N \to A$ überein, die folgende Bedingung erfüllt:

$$\text{Für alle } (n',n) \in E,\ \left\{ \begin{array}{ll} (\mathit{live}(n') \setminus \{x\}) \cup \mathit{free}(t) & \text{falls } lab(n',n) = (x := t) \\ \mathit{live}(n') \cup \mathit{free}(\varphi) & \text{falls } lab(n',n) = \varphi \in Fo_\Sigma(V) \end{array} \right\} \subseteq \mathit{alive}(n). \tag{9.15}$$

Also sind dead und alive die funktionalen Versionen der größten Relation $R \subseteq N \times V$ bzw. kleinsten Relation $R' \subseteq N \times V$ die folgende Implikationen erfüllen:

$$(n,z) \in R \ \Rightarrow\ \forall\ (n',n) \in E : \left\{ \begin{array}{lcl} lab(n',n) = (x := t) & \Rightarrow & ((n',z) \in R \vee z = x) \wedge z \notin \mathit{free}(t) \\ lab(n',n) = \varphi \in Fo_\Sigma(V) & \Rightarrow & (n',z) \in R \wedge z \notin \mathit{free}(\varphi) \end{array} \right. \tag{9.16}$$

$$\exists\ (n',n) \in E : \left\{ \begin{array}{lcl} lab(n',n) = (x := t) & \Rightarrow & ((n',z) \in R' \wedge z \neq x) \vee z \in \mathit{free}(t) \\ lab(n',n) = \varphi \in Fo_\Sigma(V) & \Rightarrow & (n',z) \in R' \vee z \in \mathit{free}(\varphi) \end{array} \right\} \ \Rightarrow\ (n,z) \in R' \tag{9.17}$$

(9.16) und (9.17) charakterisieren die toten bzw. lebendigen Variablen am Knoten n:

$$\begin{aligned} (n,z) \in R &\Leftrightarrow z \text{ ist tot am Knoten } n, \\ (n,z) \in R' &\Leftrightarrow z \text{ ist lebendig am Knoten } n. \end{aligned}$$

Also ist $\mathit{dead}(n)$ die Menge der am Knoten n toten Variablen und $\mathit{live}(n)$ die Menge der am Knoten n lebendigen Variablen.

Tote oder lebendige Variablen lassen sich auch für **globale Programmflussgraphen** bestimmen, deren Kanten nicht mit einzelnen Zuweisungen oder aussagenlogischen Formeln markiert sind, sondern mit **Basisblöcken**, d.h. Folgen von Zuweisungen und Konditionalen. An die Stelle der Menge $\{x\}$ bzw. $\mathit{free}(t)$ in den obigen Definitionen von $\delta(x := t)$ und $\gamma(x := t)$ tritt die Menge $\mathit{kill}(B)$ der im Basisblock B definierten

Variablen bzw. die Menge $gen(B)$ derjenigen Variablen, deren erstes Vorkommen in B eine Anwendung ist. Dementsprechend lauten die Definitionen von $\delta(B), \gamma(B) : A \to A$ wie folgt: Für alle $V' \subseteq V$,

$$\begin{aligned} \delta(B)(V') &= (V' \cup kill(B)) \setminus gen(B), \\ \gamma(B)(V') &= (V' \setminus kill(B)) \cup gen(B). \end{aligned}$$

9.2.3 Erreichende Zuweisungen

Definition 9.2.3.1 Eine Zuweisung $x := t$ **erreicht** einen Knoten n, wenn es einen Weg von $x := t$ nach n gibt, auf dem keine weitere Zuweisung an x oder an eine in t vorkommende Variable erfolgt.

Erreicht $x := t$ den Knoten n, dann können alle Vorkommen von x im Label einer von n ausgehenden Kante durch t ersetzt und $x := t$ ggf. gestrichen werden (siehe *Konstanten-* und *Variablenfortpflanzung* in Abschnitt 9.3).

Die Mengen der erreichenden Definitionen an Knoten von G erhält man aus einer kleinsten **join-Lösung** von $(G, \lambda n \in I.\emptyset)$. A wird als Potenzmenge der Menge $Assign$ aller im Flussgraph auftretenden Zuweisungen definiert, $\leq$ und $\sqcup$ wie in §9.2.2 und $\delta : M \to A^A$ wie folgt: Für alle $x \in V$, Σ-Terme t über V, aussagenlogischen Σ-Formeln φ über V und $Z \subseteq Assign$,

$$\begin{aligned} \delta(x := t)(Z) &= (Z \setminus \{z \in Assign \mid \exists\, u : z = (x := u)\}) \cup \{x := t\}, \\ \delta(\varphi)(Z) &= Z. \end{aligned}$$

Zur Analyse globaler Flussgraphen (siehe §9.2.2) definieren wir für jeden Basisblock B $kill(B)$ als die Menge aller Zuweisungen $x := t$ außerhalb von B, deren Variable x in B redefiniert wird, und $gen(B)$ als die Menge der Zuweisungen $x := t$ von B derart, dass hinter $x := t$ in B keine weitere Zuweisung an x erfolgt. Dementsprechend lautet die Definition von $\delta(B) : A \to A$ wie folgt: Für alle $Z \subseteq Assign$,

$$\delta(B)(Z) = (Z \setminus kill(Z)) \cup gen(Z).$$

Die Menge der den Knoten n von G erreichenden Definitionen ist also durch den Wert des kleinsten Fixpunktes von $join_{G,\delta}(\lambda n \in I.\emptyset) : A^N \to A^N$ an der Stelle n gegeben.

9.2.4 Verfügbare Ausdrücke

Definition 9.2.4.1 Ein Σ-Term t ist **verfügbar** am Knoten n, wenn jeder Weg vom letzten Vorkommen von t nach n keine Zuweisung an eine Variable von t enthält.

Ist t bei n verfügbar, dann können erneute Berechnungen von t an von n ausgehenden Kanten vermieden werden (siehe *Entfernung redundanter Ausdrücke* in Abschnitt 9.3).

Die Mengen der verfügbaren Ausdrücke an Knoten von G erhält man aus einer größten **meet-Lösung** von $(G, \lambda n \in I.\emptyset)$. A wird als Potenzmenge der Menge Exp aller im Flussgraph auftretenden Σ-Terme oder -Formeln definiert, $\leq$ und $\sqcap$ wie in §9.2.2 und $\delta : M \to A^A$ wie folgt: Für alle $x \in V$, Σ-Terme t über V, aussagenlogischen Σ-Formeln φ über V und $E \subseteq Exp$,

$$\begin{aligned} \delta(x := t)(E) &= E \setminus \{x\} \cup \{t\}, \\ \delta(\varphi)(E) &= E \cup \{\varphi\}. \end{aligned}$$

Zur Analyse globaler Flussgraphen (siehe §9.2.2) definieren wir für jeden Basisblock B $kill(B)$ als die Menge aller Ausdrücke, deren Variablen in B (zum Teil) neu belegt werden, und $gen(B)$ als die Menge der in B

benutzten und an seinem Ausgang verfügbaren Ausdrücke. Dementsprechend lautet die Definition von $\delta(B) : A \to A$ wie folgt: Für alle $E \subseteq Exp$,

$$\delta(B)(E) \quad = \quad (E \setminus kill(E)) \cup gen(E).$$

Die Menge der am Knoten n von G verfügbaren Ausdrücke ist also durch den Wert des größten Fixpunktes von $meet_{G,\delta}(\emptyset) : A^N \to A^N$ an der Stelle n gegeben.

9.2.5 Wichtige Ausdrücke

Definition 9.2.5.1 Ein Σ-Term t ist **wichtig** (*very busy expression*) am Knoten n, wenn er auf allen von n ausgehenden Wegen benutzt wird, bevor eine seiner Variablen neu belegt wird.

Nachdem die Pfeile eines globalen Programmflussgraphen G zum Zwecke der Rückwärtstraversierung umgedreht wurden, erhält man die Mengen wichtiger Ausdrücke an Knoten von G aus einer größten **meet-Lösung** von von $(G, \lambda n. \in I.\emptyset)$. A, $\leq$, $\sqcap$ und δ werden wie in §9.2.4 definiert.

Die Menge der am Knoten n von G wichtigen Ausdrücke ist also durch den Wert des größten Fixpunktes von $meet_{G,\delta}(\emptyset) : A^N \to A^N$ an der Stelle n gegeben.

9.2.6 Veränderte Variablen

Dies ist ein Beispiel einer interprozeduralen Analyse (s.o.). Der Flussgraph gibt hier die Aufrufstruktur zwischen den Prozeduren des Programms wieder: Seine Knoten bezeichnen Prozeduren. Eine Kante von p nach q zeigt an, dass q einen Aufruf von p enthält. Berechnet werden soll für jede Prozedur p die Menge der von p veränderten Variablen, die keine lokalen Variablen von p sind.

Sei $glob(p)$ die Menge der globalen Variablen von p, $form(p)$ die Menge der formalen Parameter von p und $def(p)$ die Menge aller Variablen von $glob(p) \cup form(p)$, an die im Rumpf von p eine Zuweisung erfolgt. Wir setzen wie in \$ 9.2.1 $M = E$ und definieren A, $\leq$ und $\sqcup$ wie in §9.2.2 und $\delta : M \to A^A$ wie folgt: Für alle $(p, q) \in E$ und $V' \subseteq V$,

$$\begin{aligned} \delta(p,q)(V') \quad = \quad & (V' \cap glob(q)) \cup def(q) \cup \\ & \{a \in glob(q) \cup form(q) \mid \exists i \in \mathbb{N} : a \text{ ist der i-te aktueller Parameter eines Aufrufs von } p \\ & \qquad \text{und der i-te formale Parameter von } p \text{ gehört zu } V'\} \end{aligned}$$

Die Menge der von p veränderten nichtlokalen Variablen ist durch den Wert des größten Fixpunktes von $join_{G,\delta}(\emptyset) : A^N \to A^N$ an der Stelle p gegeben.

9.2.7 Variablenbelegungen

Ähnlich wie man Programme ausführen kann, indem man ihre Syntaxbäume auswertet (siehe Beispiel 1.2.3), lässt sich ein Programmflussgraph G so interpretieren, dass nach der Eingabe einer Menge $g(k) \subseteq Dom^V$ von Variablenbelegungen ("Zuständen") an jedem Eingangsknoten k und dem Einschwingen aller Knotenbewertungen an einem Knoten $n \in N \setminus I$ all die Belegungen stehen, die sich aus der Anwendung des durch G repräsentierten Programms P auf $g(k)$, $k \in I$, ergeben. Gibt es z.B. $x \in V$ derart, dass alle Werte von x am Knoten n mit $a \in Dom$ übereinstimmen, obwohl jeder Eingangsknoten mit der Menge Dom^V aller Belegungen bewertet wurde, dann ist der von P errechnete Wert von x an der n entsprechenden Programmposition offenbar von der Eingabe unabhängig. Folglich kann P optimiert werden, indem die Vorkommen von x auf allen Wegen, die keine Zuweisung an x enthalten und zu n führen, durch a ersetzt werden (*constant propagation*).

Die Mengen möglicher Variablenbelegungen an Knoten von G erhält man aus einer kleinsten **join-Lösung** von $(G, g : I \to A)$, wobei $A = \mathcal{P}(Dom^V)$ ist. Um die Σ-Terme und -Formeln von G auswerten zu können, setzen wir voraus, dass Dom eine – durch die jeweilige Anwendung bestimmte – Σ-Algebra ist. $\leq$ und $\sqcup$ sind wie in §9.2.2 definiert und $\delta : M \to A^A$ wie folgt: Für alle $x \in V$, Σ-Terme t über V, aussagenlogischen Σ-Formeln φ über V und $Z \subseteq Dom^V$,

$$\begin{aligned} \delta(x := t)(Z) &= \{f[f^*(t)/x] \mid f \in Z\}, \\ \delta(\varphi)(Z) &= Z \cap \varphi^{Dom}, \end{aligned}$$

wobei die **Update-Funktion** $\_[\_/\_] : Dom^V \times V \times Dom \to Dom^V$ wie folgt definiert ist: Für alle $f \in Dom^V$, $x \in V$ und $a \in Dom$, $f[a/x](x) = a$ und $f[a/x](y) = f(y)$ für alle $y \in V \setminus \{x\}$. Zur Definition der Fortsetzung g^* von g auf Terme und zum Wert φ^{Dom} von φ in der Σ-Algebra Dom siehe [55] bzw. [58].

I.d.R. wird bei einer Implementierung dieser Analyse (wie auch der in §9.2.9 beschriebenen) ein Knotenwert $S \subseteq Dom^V$ durch eine Σ-Formel φ repräsentiert, deren Wert φ^{Dom} mit S übereinstimmt. Dann ist A der Verband $Fo_\Sigma(V)$ aller Σ-Formeln über V, dessen Halbordnung, Supremum, Infimum, kleinstes und größtes Element als Implikation, Disjunktion, Konjunktion, *False* bzw. *True* definiert sind.[2] Dementsprechend transformieren die Transferfunktionen keine Belegungsmengen, sondern Formeln: Für alle $x \in V$, Σ-Terme t über V und Σ-Formeln φ, ψ über V,

$$\begin{aligned} \delta(x := t)(\psi) &= \psi[t/x], \\ \delta(\varphi)(\psi) &= \psi \wedge \varphi. \end{aligned}$$

Im Gegensatz zu §9.2.1-9.2.6 terminiert der obige Algorithmus zur Fixpunktberechnung hier nicht immer, und zwar genau dann nicht, wenn auch jeder andere Interpreter von P auf der durch $g : I \to A$ gegebenen Eingabe nicht terminieren würde. M.a.W.: die zu berechnende Kette $\{join_{G,\delta}(g)^i(\bot)\}_{i<\omega}$ ist möglicherweise unendlich. Warum? Weil Dom – i.d.R. – unendlich ist und damit auch Dom^V, A sowie der Definitions- und Wertebereich A^N von $join_{G,\delta}(g)$. Das soll uns aber nicht stören, denn der Algorithmus ist auf jeden Fall *partiell* korrekt, d.h., wenn er terminiert, dann liefert er an jedem Knoten von G das jeweils erwartete Ergebnis.

Beispiel Fakultätsprogramm. Sei $V = \{x, y\}$, $Dom = \mathbb{N}$ und $G = (\{in\}, N = \{n_1, n_2, n_3, out\}, E, lab)$ folgender Programmflussgraph zur Berechnung von Fakultäten:

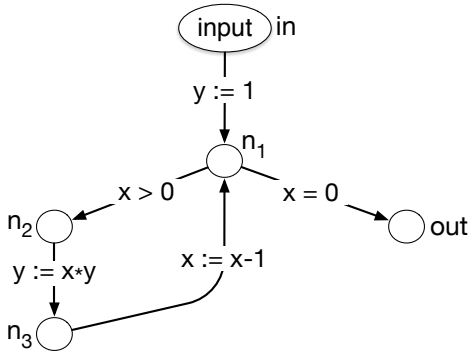


Figure 9.2.

Alle Knoten von N sind am Anfang der Berechnung der kleinsten Lösung von $(G, input)$ mit $\bot$ bewertet, also mit der leeren Belegungsmenge. Sei *input* die Menge aller Belegungen $f \in Dom^V$ mit $f(x) = 3$. Hier stehen die einzelnen Glieder der Kette $\{join_{G,\delta}(input)^i(\bot)\}_{i<\omega}$ in einer selbsterklärenden, zu unseren *kanten*markierten Flussgraphen äquivalenten Darstellung als *knoten*markierte Graphen. Der – von links nach rechts gelesen – letzte Nachfolger eines Knotens gibt seine jeweiligen Wert (Menge von Variablenbelegungen) wieder. subsflow/002.png

[2] Genaugenommen müssen A als Quotient von $Fo_\Sigma(V)$ nach der logischen Äquivalenz $\Leftrightarrow$ und $\leq$ als relationale Komposition $\Leftrightarrow \circ \Rightarrow \circ \Leftrightarrow$ definiert werden, damit $\leq$ antisymmetrisch wird.

entspricht Fig. 9.2. Im eingeschwungenen Endzustand subsflow/014.png findet man am Ausgangsknoten, d.i. der dritte Nachfolger von `ite` (if-then-else) die Belegung $\{x = 0, y = 6\}$, also das gewünschte Ergebnis: $y = f(x)! = 3! = 6$. ❑

Anstatt mit Belegungsmengen können die Knoten auch nur mit einer einzigen Belegung bewertet werden, die mehrere Belegungen repräsentiert. Dazu wird Dom zum **flachen** Verband

$$Dom' \;=_{def}\; Dom + \{\bot, \top\}$$

erweitert und A als Dom'^V – anstelle von $\mathcal{P}(Dom^V)$ – definiert. In Dom' gilt $a \leq b$ genau dann, wenn $a = \bot$, $a = b$ oder $b = \top$ ist. Daraus folgt $a \sqcup b = \top$ für alle $a, b \in Dom$ mit $a \neq b$. Damit Σ-Terme und -Formeln auch in Dom' ausgewertet werden können, muss die Σ-Algebra Dom zur Σ'-Algebra erweitert werden, d.h. die Operationen und Prädikate von Σ müssen auch auf $\bot$ und $\top$ definiert werden!

Dom'^V und $\mathcal{P}(Dom^V)$ hängen folgendermaßen zusammen: Jede Belegung $f' : V \to Dom'$ mit $\bot \in f'(V)$ entspricht der leeren Belegungsmenge $\emptyset \in \mathcal{P}(Dom^V)$; $f' : V \to Dom'$ mit $\bot \notin f'(V)$ repräsentiert die Menge aller $f \in Dom^V$, die für alle $x \in V$ die Bedingung $f'(x) \in \{f(x), \top\}$ erfüllt. M.a.W.: Durch einzelne Funktionen von V nach Dom' lassen sich nur solche Belegungsmengen $Z \subseteq Dom^V$ darstellen, für die es Variablen $x_1, \ldots, x_n$ gibt, auf denen alle Elemente von Z übereinstimmen, während Z für alle $x \in V \setminus \{x_1, \ldots, x_n\}$ und $a \in Dom$ eine Belegung f mit $f(x) = a$ enthält.

Variablenbelegungen vom Typ Dom'^V an Knoten von G erhält man aus einer kleinsten **join-Lösung** von $(G, g : I \to A)$, wobei $A = Dom'^V$ ist. $\leq$ und $\sqcup$ werden wie oben vom Verband Dom' auf den Verband $A = Dom'^V$ fortgesetzt. Die δ entsprechende Kanteninterpretation $\delta' : M \to A^A$ ist wie folgt definiert: Für alle $x \in V$, Σ-Terme t über V, aussagenlogischen Σ-Formeln φ über V und $f' : V \to Dom'$,

$$\begin{aligned} \delta'(x := t)(f') &= f'[f'^*(t)/x], \\ \delta'(\varphi)(f') &= \begin{cases} f' & \text{falls } f' \in \varphi^{Dom'}, \\ \lambda x.\bot & \text{sonst.} \end{cases} \end{aligned}$$

Ist Dom unendlich, dann ist auch Dom' unendlich, so dass der obige Algorithmus zur Fixpunktberechnung wie oben möglicherweise nicht terminiert. Mehr noch: Nicht alle Transferfunktionen $\delta'(x := t) : A^N \to A^N$ sind stetig und deshalb kann an manchen Knoten n die MOP-Lösung $joinMOP(g : I \to A) \in A^N$ verschieden von $lfp(join_{G,\delta'}(g))$ sein (siehe Satz 9.1.8). Auf einen solchen Fall stößt man z.B. bei der Berechnung des kleinsten Fixpunkts von $join_{G,\delta'}(\lambda n.\top)$, wobei $G = (\{n_1\}, \{n_2, \ldots, n_5\}, E, lab)$ durch folgenden Programmflussgraphen gegeben ist:

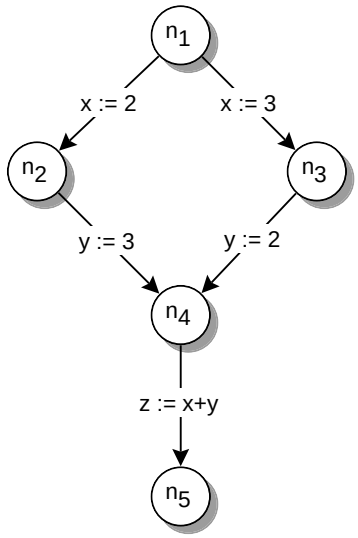


Figure 9.3.

Seien $V = \{x, y, z\}$, $act = (z := x + y)$ und $f, f' : V \to Dom'$ wie folgt definiert: $f(x) = f'(y) = 2$, $f(y) = f'(x) = 3$ und $f(z) = f'(z) = \top$. Dann gilt $\delta'(act)(f)(z) = 5 = \delta'(act)(f')(z)$. Daraus folgt

$$\delta'(act)(f \sqcup f')(z) = (f \sqcup f')^*(x + y) = (f \sqcup f')(x) + (f \sqcup f')(y) = (f(x) \sqcup f'(x)) + (f(y) \sqcup f'(y))$$
$$= (2 \sqcup 3) + (3 \sqcup 2) = \top \sqcup \top = \top \neq 5 = 5 \sqcup 5 = \delta'(act)(f)(z) \sqcup \delta'(act)(f')(z) = (\delta'(act)(f) \sqcup \delta'(act)(f'))(z),$$

also $\delta'(act)(f \sqcup f') \neq \delta'(act)(f) \sqcup \delta'(act)(f')$, d.h. $\delta'(act)$ ist nicht stetig! Demgegenüber ist die oben definierte Interpretation von $x := t$ als Funktion von $\mathcal{P}(Dom^V)$ nach $\mathcal{P}(Dom^V)$ immer stetig: Für alle $x \in V$, Σ-Terme t über V und $Z, Z' \subseteq Dom^V$,

$$\delta'(x := t)(Z \cup Z') = \{f[f^*(t)/x] \mid f \in Z \cup Z'\} = \{f[f^*(t)/x] \mid f \in Z\} \cup \{f[f^*(t)/x] \mid f \in Z'\}$$
$$= \delta'(x := t)(Z) \cup \delta'(x := t)(Z').$$

9.2.8 Abstrakte Interpretationen

Hierunter versteht man von Dom oder Dom' (siehe §9.2.7) ausgehende Σ-Homomorphismen mit endlichem Wertebereich Abs. Der Informationsgehalt einer Knotenbewertung des Typs Abs^V oder $\mathcal{P}(Abs^V)$ ist zwar geringer als der einer Knotenbewertung des Typs Dom'^V bzw. $\mathcal{P}(Dom^V)$, aber der obige Algorithmus zur Fixpunktbestimmung terminiert jetzt, weil V *und* Abs endlich sind.

Ist Dom total geordnet, dann besteht eine typische Abstraktion von Dom in einem Σ-Homomorphismus h, der jedem Element a von Dom den Namen eines von endlich vielen Intervallen zuordnet, in dem a liegt. Sei z.B. Σ eine Signatur mit den Operationen und Prädikaten von Fig. 9.2, $Dom = \mathbb{R}$, $Abs = \{+1, 0, -1, ?\}$ und $h : Dom \to Abs$ die Vorzeichenfunktion. +1, 0, -1 und ? stehen für die Intervalle $(0, \infty)$, $[0, 0]$, $(-\infty, 0)$ bzw. $(-\infty, \infty)$.

Abs lässt sich so zu einer Σ-Algebra erweitern, dass h Σ-homomorph wird, dass also für alle Operationen $op : e \to e'$ und Prädikate $p : e$ von Σ $h_e \circ op^{Dom} = op^{Abs} \circ h_e$ bzw. $h_e(p^{Dom}) \subseteq p^{Abs}$ gilt (siehe [58]). Dann erfüllt Abs z.B. die atomaren Formeln $0* ? = 0$, $(+1) + (-1) = ?$, $(-1) + (-1) = -1$ und $+1 < +1$.

Die Vorzeichen möglicher Variablenwerte an den Knoten von G erhält man aus einer kleinsten **join-Lösung** von $(G, g : I \to A)$, wobei $A = Abs^V$ ist. Auch in der Definition der Transferfunktionen $\delta(m) : \mathcal{P}(Dom^V) \to \mathcal{P}(Dom^V)$ von §9.2.7 braucht man nur Dom durch Abs ersetzen. Wir definieren also $\leq$ und $\sqcup$ wieder wie in §9.2.2 und $\delta : M \to A^A$ wie folgt: Für alle $x \in V$, Σ-Terme t über V, aussagenlogischen Σ-Formeln φ über V und $Z \subseteq Abs^V$,

$$\begin{aligned} \delta(x := t)(Z) &= \{f[f^*(t)/x] \mid f \in Z\}, \\ \delta(\varphi)(Z) &= Z \cap \varphi^{Abs}. \end{aligned}$$

Knotenwerte vom Typ $\mathcal{P}(Abs^V)$ liefern zwar weniger Information über das durch G repräsentierte Programm als die in §9.2.7 berechneten Variablenbelegungen. Da jedoch Abs und V endlich sind und damit auch Abs^V, A, A^N und die Kette $\{join_{G,\delta}(g)(\bot)^i : A^N \to A^N\}_{i<\omega}$, können wir im Gegensatz zu §9.2.7 sicher sein, dass der obige Algorithmus zur Fixpunktberechnung terminiert.

9.2.9 Schwächste Vorbedingungen

Dreht man die Kanten von G um, dann wird aus dem in §9.2.7 realisierten Interpreter $lfp(join_{G,\delta}(g))$ des durch G repräsentierten Programms ein Verfahren zur Berechnung schwächster Vorbedingungen, das sind notwendige Anforderungen an die Eingaben des Programms. Wir betrachten wieder das

Beispiel Fakultätsprogramm. Sei $V = \{x, y\}$, $Dom = \mathbb{N}$, $k \in \mathbb{N}$, $output = (x = 0 \wedge y = k!)$ und $G = (\{out\}, \{in, n_0, \ldots, n_3\}, E, lab)$ folgender Programmflussgraph (mit an die Rückwärtstraversierung angepasster Pfeilrichtung):

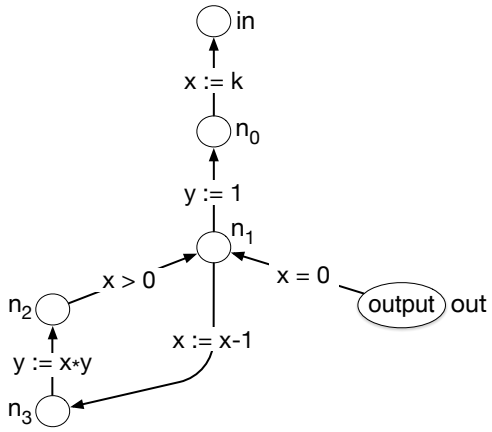


Figure 9.4.

Die schwächsten Vorbedingungen (in Gestalt der sie erfüllenden Variablenbelegungen) an Knoten von G erhält man aus einer größten **meet-Lösung** von $(G, g : I \to A)$, wobei wie in §9.2.7 $A = \mathcal{P}(Dom^V)$ ist. $\leq$ und $\sqcap$ sind wieder wie in §9.2.2 definiert und $\delta : M \to A^A$ wie folgt: Für alle $x \in V$, Σ-Terme t über V, aussagenlogischen Σ-Formeln φ über V und $Z \subseteq Dom^V$,

$$\begin{aligned} \delta(x := t)(Z) &= \{f \in Z \mid f[f^*(t)/x] \in Z\}, \\ \delta(\varphi)(Z) &= Z \cup (Dom^V \setminus \varphi^{Dom}). \end{aligned}$$

Anstelle von erfüllenden Belegungen können wie in 9.2.7 Formeln die Knoten von G markieren: Mit $A = Fo_\Sigma(V)$ ist $\delta : M \to A^A$ wie folgt definiert: Für alle $x \in V$, Σ-Terme t über V und $\varphi, \psi \in Fo_\Sigma(V)$,

$$\begin{aligned} \delta(x := t)(\psi) &= \psi' \text{ mit } \psi'[t/x] = \psi, \\ \delta(\varphi)(\psi) &= \psi \vee \neg\varphi. \end{aligned}$$

Ist Dom unendlich, dann kann es analog in §9.2.7 passieren, dass der obige Algorithmus zur Berechnung einer größten meet-Lösung von (G, g) nicht terminiert. Das lässt sich oft dadurch vermeiden, dass jede atomare Formel, die in eine Schleife hineinführt, mit einer Invariante der Schleife verknüpft wird.

Beispiel Fakultätsprogramm. Beschreibt $output \in Fo_\Sigma(V)$ die Ein/Ausgabe-Relation ("Nachbedingung") des durch den Programmflussgraphen $G = (\{out\}, \{in, n_0, \ldots, n_3\}, E, lab)$ von Fig. 9.4 repräsentierten Fakultätsprogramms P, dann liefert der Wert des größten Fixpunktes von $\Phi = meet_{G,\delta}(output) : A^N \to A^N$ am Knoten in die schwächste Anforderung an die Werte von x und y, die vor der Ausführung vor P erfüllt sein muss, damit nach der Ausführung $output$ gilt. Die zu P passende Nachbedingung lautet $output = (x = 0 \wedge y = k!)$.

Nach Korollar 9.1.4 (3) ist der größte Fixpunkt von Φ das kleinste Element h der Cokette $\{\Phi^i(\top) \mid i < \omega\}$, falls diese endlich ist. Nach Lemma 9.1.6 gilt dann

$$h(n) \leq \delta(lab(n', n))([g, h](n')) \tag{9.18}$$

für alle $(n', n) \in E$. Wegen

$$\delta(\varphi)(\psi) = (\psi \vee \neg\varphi) \leq (\psi \vee \neg\varphi \vee \neg\vartheta) = \delta(\varphi \wedge \vartheta)(\psi)$$

für alle $\varphi, \psi, \vartheta \in Fo_\Sigma(V)$ steigt die Chance, dass h mit (9.18) nach endlich vielen Iterationen von Φ erreicht wird, wenn Bedingungen, die in Schleifen hineinführen ($x > 0$ in Fig. 9.4), durch konjunktive Verknüpfung mit einer Formel **Invariante** ϑ der jeweiligen Schleife verstärken, d.h. ϑ muss am Eintrittsknoten (n_1 in Fig. 9.4) in jeder Iteration von Φ erfüllt sein. Das gilt in Fig. 9.4 für $\vartheta = (x! * y = k!)$. Fig. 9.5 zeigt den entsprechend modifizierten Flussgraphen G':

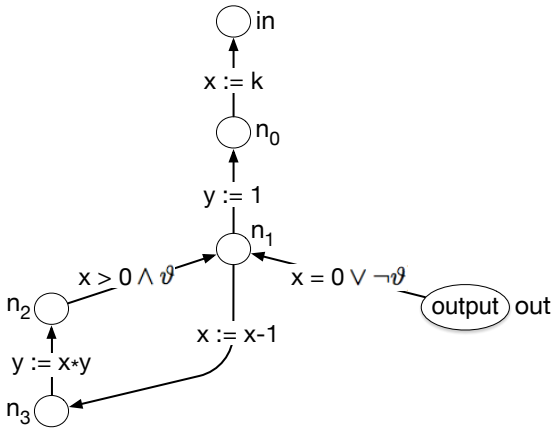


Figure 9.5.

Nach der ersten Iteration von $\Psi = meet_{G',\delta}(output)$ erhalten wir an den Knoten $n_0, \ldots, n_3$ folgende Werte:

$$\begin{aligned}
&\Psi(\top)(n_1) = \top(n_1) \sqcap \delta(lab(n_1,n_2))(\top(n_2)) \sqcap \delta(lab(n_1,out))(output)\\
&\quad = True \wedge \delta(x>0 \wedge \vartheta)(True) \wedge \delta(x=0 \vee \neg\vartheta)(output) = (True \vee x=0 \vee \neg\vartheta) \wedge (output \vee (x>0 \wedge \vartheta))\\
&\quad = output \vee (x>0 \wedge \vartheta),\\
&\Psi(\top)(n_0) = \top(n_0) \sqcap \delta(lab(n_0,n_1))(\top(n_1)) = True \wedge \delta(y:=1)(True) = True \geq \top(n_0),\\
&\Psi(\top)(in) = \top(in) \sqcap \delta(lab(in,n_0))(\top(n_0)) = True \wedge \delta(x:=k)(True) = True \geq \top(in),\\
&\Psi(\top)(n_2) = \top(n_2) \sqcap \delta(lab(n_2,n_3))(\top(n_3)) = True \wedge \delta(y:=x*y)(True) = True \geq \top(n_2),\\
&\Psi(\top)(n_3) = \top(n_3) \sqcap \delta(lab(n_3,n_1))(\top(n_1)) = True \wedge \delta(x:=x-1)(True) = True \geq \top(n_3).
\end{aligned}$$

Wegen $\top(n_1) \not\leq \Psi(\top)(n_1)$ müssen wir Ψ ein zweites Mal anwenden und erhalten folgende Werte:

$$\begin{aligned}
&\Psi^2(\top)(n_1) = \Psi(\top)(n_1) \sqcap \delta(lab(n_1,n_2))(\Psi(\top)(n_2)) \sqcap \delta(lab(n_1,out))(\Psi(\top)(out))\\
&\quad = (output \vee (x>0 \wedge \vartheta)) \wedge \delta(x>0 \wedge \vartheta)(True) \wedge \delta(x=0 \vee \neg\vartheta)(output) = output \vee (x>0 \wedge \vartheta) = \Psi(\top)(n_1)\\
&\Psi^2(\top)(n_0) = \Psi(\top)(n_0) \sqcap \delta(lab(n_0,n_1))(\Psi(\top)(n_1)) = True \wedge \delta(y:=1)(output \vee (x>0 \wedge \vartheta))\\
&\quad = (x=0 \wedge 1=k!) \vee (x>0 \wedge x!=k!) = (x!=k!),\\
&\Psi^2(\top)(in) = \Psi(\top)(in) \sqcap \delta(lab(in,n_0))(\Psi(\top)(n_0)) = \top(in) \sqcap \delta(lab(in,n_0))(\top(n_0)) = \Psi(\top)(in),\\
&\Psi^2(\top)(n_2) = \Psi(\top)(n_2) \sqcap \delta(lab(n_2,n_3))(\Psi(\top)(n_3)) = \top(n_2) \sqcap \delta(lab(n_2,n_3))(\top(n_3)) = \Psi(\top)(n_2),\\
&\Psi^2(\top)(n_3) = \Psi(\top)(n_3) \sqcap \delta(lab(n_3,n_1))(\Psi(\top)(n_1)) = True \wedge \delta(x:=x-1)(output \vee (x>0 \wedge \vartheta))\\
&\quad = (x=1 \wedge y=k!) \vee (x>1 \wedge (x-1)! * y = k!) = (x>0 \wedge (x-1)! * y = k!).
\end{aligned}$$

Wegen $\Psi(\top)(n_0) \not\leq \Psi^2(\top)(n_0)$ und $\Psi(\top)(n_3) \not\leq \Psi^2(\top)(n_3)$ müssen wir Ψ ein drittes Mal anwenden und erhalten folgende Werte:

$$\begin{aligned}
&\Psi^3(\top)(n_1) = \Psi^2(\top)(n_1) \sqcap \delta(lab(n_1,n_2))(\Psi^2(\top)(n_2)) \sqcap \delta(lab(n_1,out))(\Psi^2(\top)(out))\\
&\quad = \Psi(\top)(n_1) \sqcap \delta(lab(n_1,n_2))(\Psi(\top)(n_2)) \sqcap \delta(lab(n_1,out))(\Psi(\top)(out)) = \Psi^2(\top)(n_1),\\
&\Psi^3(\top)(n_0) = \Psi^2(\top(n_0)) \sqcap \delta(lab(n_0,n_1))(\Psi^2(\top)(n_1))\\
&\quad = (x!=k! \wedge \delta(y:=1)(output \vee (x>0 \wedge \vartheta))) = (x!=k! \wedge (x=0 \wedge 1=k!) \vee (x>0 \wedge x!=k!))\\
&\quad = (x!=k!) = \Psi^2(\top)(n_0),\\
&\Psi^3(\top)(in) = \Psi^2(\top)(in) \sqcap \delta(lab(in,n_0))(\Psi^2(\top)(n_0)) = True \wedge \delta(x:=k)(x!=k!) = (k!=k!) = True\\
&\quad \geq \Psi^2(\top)(in),\\
&\Psi^3(\top)(n_2) = \Psi^2(\top)(n_2) \sqcap \delta(lab(n_2,n_3))(\Psi^2(\top)(n_3))\\
&\quad = True \wedge \delta(y:=x*y)(x>0 \wedge (x-1)! * y = k!) = (x>0 \wedge (x-1)! * x * y = k!) = (x>0 \wedge x! * y = k!),
\end{aligned}$$

$$\begin{aligned}
\Psi^3(\top)(n_3) &= \Psi^2(\top)(n_3) \sqcap \delta(lab(n_3,n_1))(\Psi^2(\top)(n_1)) \\
&= (x>0 \wedge (x-1)! * y = k! \wedge \delta(x := x-1)(output \vee (x>0 \wedge \vartheta))) \\
&= (x>0 \wedge (x-1)! * y = k! \wedge ((x=1 \wedge y=k!) \vee (x>1 \wedge (x-1)! * y = k!)) = (x>0 \wedge (x-1)! * y = k!) \\
&= \Psi^2(\top)(n_3).
\end{aligned}$$

Wegen $\Psi^2(\top)(n_2) \not\leq \Psi^3(\top)(n_2)$ müssen wir Ψ ein viertes Mal anwenden und erhalten folgende Werte:

$$\begin{aligned}
\Psi^4(\top)(n_1) &= \Psi^3(\top)(n_1) \sqcap \delta(lab(n_1,n_2))(\Psi^3(\top)(n_2)) \sqcap \delta(lab(n_1,out))(\Psi^3(\top)(out)) \\
&= (output \vee (x>0 \wedge \vartheta)) \wedge \delta(x>0 \wedge \vartheta)(x>0 \wedge x! * y = k!) \wedge \delta(x=0 \vee \neg\vartheta)(output) \\
&= (output \vee (x>0 \wedge \vartheta)) \wedge ((x>0 \wedge x! * y = k!) \vee x=0 \vee \neg\vartheta) \wedge (output \vee (x>0 \wedge \vartheta)) \\
&= ((x=0 \wedge y=k!) \vee (x>0 \wedge x! * y = k!)) \wedge ((x>0 \wedge \vartheta) \vee x=0 \vee \neg\vartheta) = (x! * y = k! \wedge True) \\
&= (x! * y = k!) \geq ((x=0 \wedge y=k!) \vee (x>0 \wedge x! * y = k!)) = output \vee (x>0 \wedge \vartheta) = \Psi^3(\top)(n_1), \\
\Psi^4(\top)(n_0) &= \Psi^3(\top(n_0)) \sqcap \delta(lab(n_0,n_1))(\Psi^3(\top)(n_1)) = \Psi^2(\top(n_0)) \sqcap \delta(lab(n_0,n_1))(\Psi^2(\top)(n_1)) = \Psi^3(\top(n_0)), \\
\Psi^4(\top)(in) &= \Psi^3(\top)(in) \sqcap \delta(lab(in,n_0))(\Psi^3(\top)(n_0)) \geq \Psi^2(\top)(in) \sqcap \delta(lab(in,n_0))(\Psi^2(\top)(n_0)) = \Psi^3(\top(in)), \\
\Psi^4(\top)(n_2) &= \Psi^3(\top)(n_2) \sqcap \delta(lab(n_2,n_3))(\Psi^3(\top)(n_3)) \\
&= (x>0 \wedge x! * y = k! \wedge \delta(y := x * y)(x>0 \wedge (x-1)! * y = k!)) \\
&= (x>0 \wedge x! * y = k! \wedge x>0 \wedge (x-1)! * x * y = k!) = (x>0 \wedge x! * y = k!) = \Psi^3(\top)(n_2), \\
\Psi^4(\top)(n_3) &= \Psi^3(\top)(n_3) \sqcap \delta(lab(n_3,n_1))(\Psi^3(\top)(n_1)) = \Psi^2(\top)(n_3) \sqcap \delta(lab(n_3,n_1))(\Psi^2(\top)(n_1)) = \Psi^3(\top)(n_3).
\end{aligned}$$

Also gilt $\Psi^3(\top) = \Psi^4(\top)$. Daraus folgt nach Korollar 9.1.4 (3), dass $\Psi^3(\top)$ der größte Fixpunkt von Ψ und damit die größte meet-Lösung von $(G', output)$ bzgl. δ ist. Wegen der Invarianzeigenschaft von ϑ ist $\Phi = \Psi$, also $gfp(\Phi)(in) = gfp(\Psi)(in) = \Psi^3(\top)(in) = True$. Daraus schließen wir, dass $True$ die schwächste Vorbedingung des durch G repräsentierten Programms P ist, m.a.W.: P berechnet die Fakultät *jeder* natürlichen Zahl.

Fig. 9.6 und Fig. 9.7 zeigen $\Psi(\top)$ und $\Psi^3(\top)$ als knotenmarkierte Bäume, ähnlich den Kettengliedern des Beispiels in §9.2.7. Der – von links nach rechts gelesen – letzte Nachfolger eines Knotens n liefert wieder seinen jeweiligen Wert, der hier die jeweilige Bedingung vor Ausführung der anderen direkten Nachfolger von n darstellt.[3] Im (eingeschwungenen) Endzustand (Fig. 9.7) der Fixpunktberechnung zeigt den Wert von in die Bedingung $n > 0 \vee n = 0$, was den oben abgeleiteten Wert $True$ von $\Psi^3(\top)(in)$ wiedergibt. Um die Baumdarstellung nicht zu überfrachten, wurden in Fig. 9.7 die Werte innerer Knoten ausgeblendet und durch das Symbol @ ersetzt.

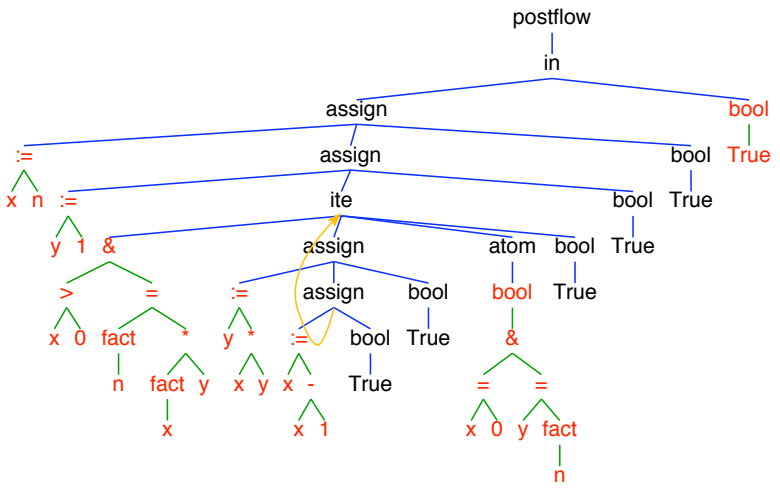


Figure 9.6.

[3] Der Operator *bool* konvertiert prädikatenlogische Formeln in funktionale Terme.

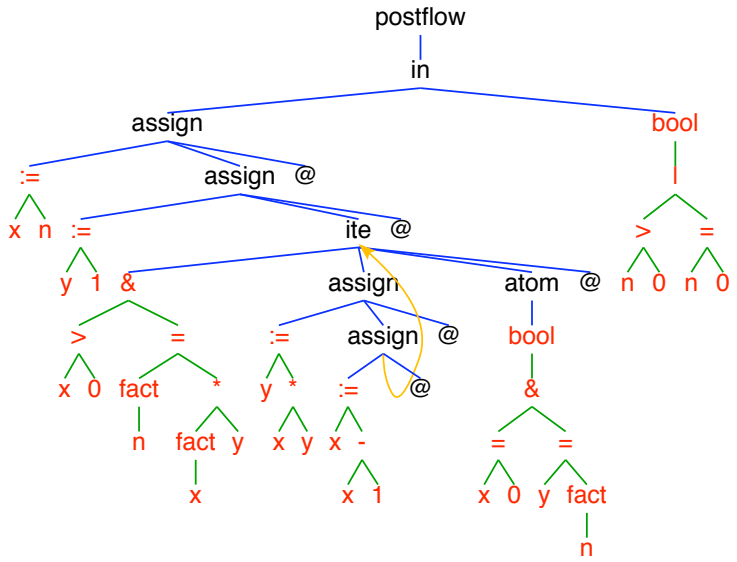


Figure 9.7.

9.2.10 Die Lösung von Datenflussaufgaben in algebraischen Theorien

Dieser Abschnitt ist veraltet!!

Sei $\Sigma = (S, BS, BF, F)$ eine konstruktive Signatur und I, V endliche S-sortige Variablenmengen (siehe [55]). Eine S-sortige Funktion

$$E : V \to T_\Sigma(I + V)$$

heißt **iteratives Σ-Gleichungssystem**, falls das Bild von E keine Variablen enthält.

Sei A eine Σ-Algebra und A^{I+V} die Menge der S-sortigen Funktionen von $I + V$ nach A.

$g : I + V \to A$ **löst** E **in** A, wenn für alle $x \in V$, $g^*(E(x)) = g(x)$ gilt.

E entspricht einem knotenmarkierten Graphen G mit folgenden Eigenschaften. Sei N die Menge aller Knoten und M die Menge aller Knotenmarkierungen von G.

- $V \subseteq N$.
- Jeder Knoten von G ist mit einem Element von $\Sigma \cup I$ markiert.
- Fasst man Terme als Bäume auf, dann setzt sich G aus den Bildern von E zusammen.
- Für alle $x \in V$ repräsentiert $E(x)$ denjenigen Teilgraph von G, dessen Wurzel mit x übereinstimmt.

Beispiel Model checking

Die Formeln des *modalen μ-Kalküls* beschreiben Eigenschaften eines markierten Transitionssystems (siehe [52], §3.2, oder [57], §10.1). Die globale Interpretation einer solchen Formel ist eine Menge von Zuständen des Transitionssystems (intuitiv: die die Formel erfüllenden Zustände). Das entspricht schon mal der Knoteninterpretation in den beiden vorangegangenen Beispielen. Bleibt die Frage, inwieweit eine Formel des μ-Kalküls als Flussgraph darstellbar ist. Schauen wir uns dazu zunächst die Syntax und die wie üblich induktiv definierte Semantik der Formeln genauer an:

Sei $(State, Act, \to \subseteq State \times Act \times State)$ ein markiertes Transitionssystem, Var eine Menge von Zustandsmengenvariablen, $X \in Var$, $S \subseteq State$, $a \in Act$ und für alle $st \in State$,

$$succ(st, a) = \{st' \in State \mid st \xrightarrow{a} st'\}.$$

Die Formeln des modalen μ-Kalküls werden durch folgende Grammatik erzeugt:

Syntax modallogischer Formeln

$$\varphi \longrightarrow true \mid false \mid S \mid X \mid \wedge_{i=1}^{n} \varphi_i \mid \vee_{i=1}^{n} \varphi_i \mid [a]\varphi \mid \langle a\rangle\varphi \mid \mu X.\varphi \mid \nu X.\varphi \quad \text{für alle } X \in Var$$

Die semantische Funktion Sem, die, abhängig von einer Belegung $\rho : Var \to \wp(State)$ der Zustandsmengenvariablen, jeder Formel eine Zustandsmenge zuordnet, wird wie üblich induktiv über der Formelmenge definiert:

Semantik modallogischer Formeln Sei $X \in Var$ und $a \in Act$.

$$\begin{array}{lcll}
Sem(\rho)(true) & = & State & \\
Sem(\rho)(false) & = & \emptyset & \\
Sem(\rho)(S) & = & S & \\
Sem(\rho)(X) & = & \rho(X) & \\
Sem(\rho)(\wedge_{i=1}^{n}\varphi_i) & = & \cap_{i=1}^{n} Sem(\rho)(\varphi_i) & \\
Sem(\rho)(\vee_{i=1}^{n}\varphi_i) & = & \cup_{i=1}^{n} Sem(\rho)(\varphi_i) & \\
Sem(\rho)([a]\varphi) & = & \{st \in State \mid \forall st' : st \xrightarrow{a} st' \Rightarrow st' \in Sem(\rho)(\varphi)\} & \\
 & = & \{st \in State \mid succ(st,a) \subseteq Sem(\rho)(\varphi)\} & \\
Sem(\rho)(\langle a\rangle\varphi) & = & \{st \in State \mid \exists st' : st \xrightarrow{a} st' \wedge st' \in Sem(\rho)(\varphi)\} & \\
 & = & \{st \in State \mid succ(st,a) \cap Sem(\rho)(\varphi) \neq \emptyset\} & \\
Sem(\rho)(\mu X.\varphi) & = & \cup_{i=1}^{n} F^n(\emptyset) & \mu\textbf{-Abstraktion} \\
Sem(\rho)(\nu X.\varphi) & = & \cap_{i=1}^{n} F^n(State) & \nu\textbf{-Abstraktion}
\end{array}$$

Hierbei ist die Funktion $F : \wp(State) \to \wp(State)$ definiert durch $F(S) = Sem(\rho[S/X])(\varphi)$. Nach Kleene's Fixpunktsatz ist $Sem(\rho)(\mu X.\varphi)$ bzw. $Sem(\rho)(\nu X.\varphi)$ der kleinste bzw. größte Fixpunkt von F, m.a.W. die kleinste bzw. größte Lösung der Gleichung $X = \varphi$ in X.

A ist hier die Potenzmenge von $State$, $\bot$ die leere Menge, $\top = State$, $\sqcup$ die Mengenvereinigung und $\sqcap$ der Mengendurchschnitt.

Z.B. lässt sich die modallogische Formel

$$\varphi \quad = \quad \nu X.(\mu Y.(\langle a\rangle true \vee \langle b\rangle Y) \wedge [b]X) \tag{9.2}$$

(siehe 9.2.10) durch das Gleichungssystem E mit

$$\begin{array}{lcl} E(X) & = & \nu(and(Y, AX(b,X))) \\ E(Y) & = & \mu(or(EX(a,true), EX(b,Y))) \end{array} \tag{9.3}$$

darstellen.[4]

Sei A eine Σ-Algebra, die die Bedingungen von Def. 9.1.1 erfüllt. Werden die Operationen von Σ als auf- bzw. abwärtsstetige Funktionen interpretiert, dann hat E nach dem Fixpunktsatz von Kleene eine kleinste Lösung $lfp(E) \in A^n$ bzw. größte Lösung $gfp(E) \in A^n$ in $X_1, \ldots, X_n$, die wie folgt definiert sind:

$$\begin{array}{lcl} lfp(E) & = & \sqcup_{i\in\mathbb{N}}(f_1,\ldots,f_n)^i(\bot,\ldots,\bot), \\ gfp(E) & = & \sqcap_{i\in\mathbb{N}}(f_1,\ldots,f_n)^i(\top,\ldots,\top), \end{array}$$

wobei $(f_1,\ldots,f_n)(a_1,\ldots,a_n) =_{def} (f_1(a_1,\ldots,a_n),\ldots,f_n(a_1,\ldots,a_n))$.

[4] $AX(a,\varphi))$ stands for $[a]\varphi$ ("All neXt"). $EX(a,\varphi))$ stands for $\langle a\rangle\varphi$ ("Exists neXt").

(9.1) entspricht einem Graphen, enthält einem Flussgraphen mit Knotenmenge $X_1, \ldots, X_n$ entsteht, hängt davon ab, ob er vorwärts oder rückwärts durchlaufen werden soll: Sei $1 \leq i \leq n$. Bei einer Vorwärtsanalyse ist f_i eine Funktion direkter Vorgänger von X_i, bei einer Rückwärtsanalyse ist f_i eine Funktion direkter Nachfolger von X_i.

Ausserdem sind in (9.3) die logischen Operatoren von (9.2) durch Funktionen ersetzt worden, die (zusammen mit weiteren, modallogischen Operatoren entsprechenden Funktionen) wie folgt definiert sind: Sei $val, val_1, val_2 \subseteq State$.

$$
\begin{array}{lcll}
true & = & State & \\
false & = & \emptyset & \\
and(val_1, val_2) & = & val_1 \cap val_2 & \\
or(val_1, val_2) & = & val_1 \cup val_2 & \\
AX(a, val) & = & \{st \in State \mid succ(st, a) \subseteq val\} & \text{für alle } a \in Act \\
EX(a, val) & = & \{st \in State \mid succ(st, a) \cap val \neq \emptyset\} & \text{für alle } a \in Act \\
\mu(val) & = & val & \\
\nu(val) & = & val &
\end{array}
$$

Es sieht so aus, als würden die μ- und ν-Knoten ihre Nachfolger gar nicht verändern. Das ist aber nicht richtig, weil die Rückwärtsanalyse von φ aus sowohl any- als auch all-Anteile enthält. An einem μ-Knoten X wird any-Analyse gemacht, d.h. in jedem Durchlauf des Formelgraphen der alte Wert von X mit dem neuen vereinigt, an einem ν-Knoten Y hingegen wird all-Analyse gemacht, d.h. in jedem Durchlauf des Formelgraphen der alte Wert von Y mit dem neuen geschnitten.

Um im Flussgraphen die Veränderung der Knotenwerte bei einer Analyse sichtbar zu machen, versehen wir jeden Knoten mit einem weiteren Nachfolger, in dem sein jeweils aktueller Wert gespeichert wird. Fig. 9.3 (1) zeigt ein markiertes Transitionssystem (1) sowie den Anfangszustand (2), einen Zwischenzustand (3) und den Endzustand (4) der Rückwärtsanalyse der Formel (9.2). Sobald sich der Wert an einem μ- oder ν-Knoten (oder einem Blatt) nicht mehr ändert ("eingeschwungen" ist), wird dieser durch einen *atom*-Knoten mit dem jeweiligen Wert ersetzt. Der Wert des *atom*-Knotens im Endzustand liefert das Ergebnis: $\{3, 4\}$ ist die Menge der Zustände, die (9.2) erfüllen. Die Analyse wurde mit Expander2 [60] durchgeführt.

Das Verfahren ist auf *alternierungsfreie* Formeln beschränkt (siehe [52]). Eine modallogische Formel ist alternierungsfrei, wenn sich die Gültigkeitsbereiche zweier (Zustandsmengen-)Variablen nur dann überlappen, wenn entweder beide μ- oder beide ν-Abstraktionen sind. Nicht alternierungsfreie Formeln kommen zum Glück in der Praxis kaum vor.

Will man klassische Datenflussaufgaben analog zum globalem model checking lösen, dann müssen die jenen zugrundeliegenden kantenmarkierten Flussgraphen in knotenmarkierte und die Kanteninterpretation (siehe Def. 9.1.1) in eine Knoteninterpretation überführt werden. Ausgehend vom Eingangsknoten lässt sich ein lokaler Programmflussgraph wie folgt in einen knotenmarkierten transformieren:

- Jede Kante $n \xrightarrow{x:=t} n'$ wird durch eine Kante $n \to n'$ ersetzt. Der Quellknoten n wird mit *assign* markiert.
- Je zwei Kanten $n \xrightarrow{p} n_1$ und $n \xrightarrow{\neg p} n_2$ werden durch zwei Kanten $n \to n_1$ und $n \to n_2$ ersetzt. Der Quellknoten n wird mit *ite* markiert.

Z.B. liefert der Flussgraph des Fakultäts-Programms

```
x := n; y := 1; lab: if x > 0 then begin y := x*y; x := x-1 goto lab end
                                else skip
```

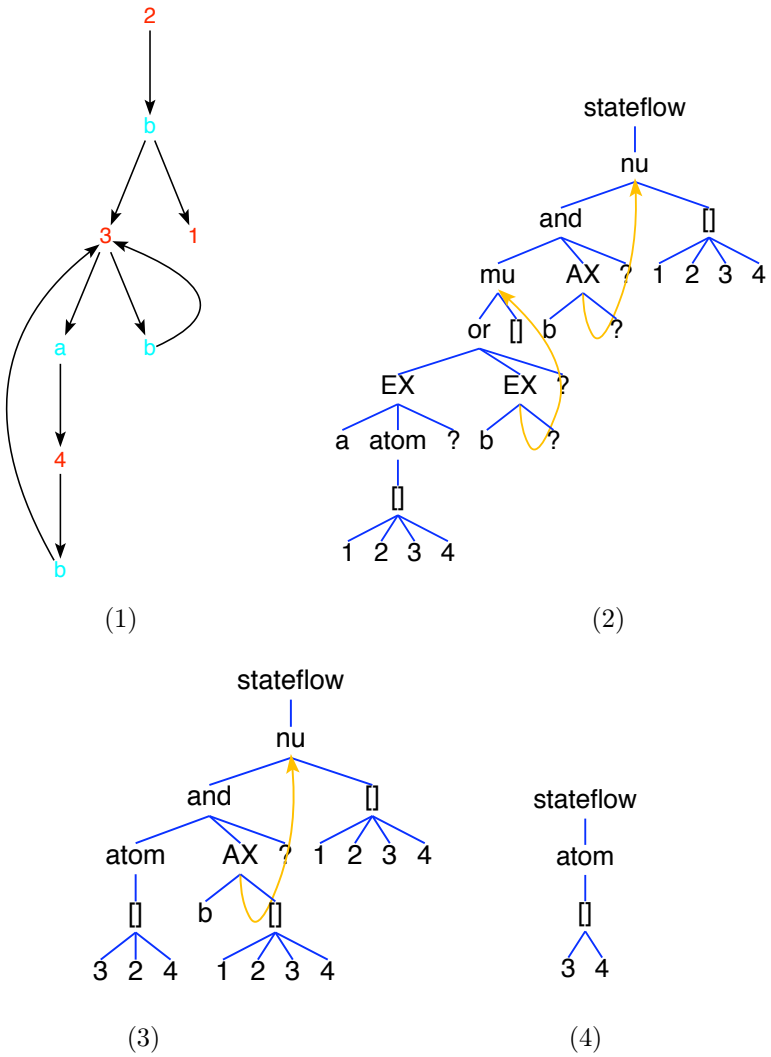


Figure 9.8.

das Gleichungssystem E über $\{IN\}$ und $\{X, Y\}$ mit:

$$\begin{aligned} E(X) &= assign(y,1)(assign(x,n)(IN)) \\ E(Y) &= join(\lambda(\iota_1(z).assign(x,x-1)(assign(y,x*y)(z))|\iota_2(z).\iota_2(z))(fork(x>0)(Y))) \end{aligned} \tag{9.4}$$

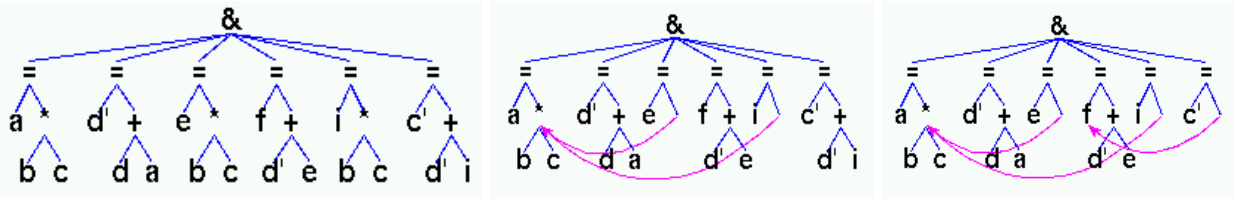
9.3 Optimierungen

Die folgenden Optimierungen können zum Teil sowohl global (basisblockübergreifend) als auch lokal nur für einzelne Basisblöcke durchgeführt werden. Manchmal genügt es auch, die Optimierung auf noch kleinere Codestücke zu beschränken (*peephole optimization*). Das tut man vor allem mit kurzen Folgen von *Maschinenbefehlen*, um bei der Übersetzung von Basisblöcke in Maschinencode entstandene Redundanzen wie z.B. überflüssige Ladebefehle zu beseitigen oder Sprungketten zu verkürzen.

Konstantenfaltung bezeichnet die Auswertung von Teilausdrücken und die Einsetzung der Werte zur Übersetzungszeit. Z.B. lässt sich eine Zuweisungsfolge x:=3;...;y:=4*x durch x:=3;...;y:=12 ersetzen, sofern die erste Zuweisung die letzte *erreicht* (s. Def. 9.2.3.1).

Variablenfortpflanzung (copy propagation) bezeichnet die Substitution benutzter Variablen durch vorher an diese zugewiesene Ausdrücke: x:=t;...;y:=4*x wird durch x:=t;...;y:=4*t ersetzt, sofern wieder die erste Zuweisung die letzte erreicht.

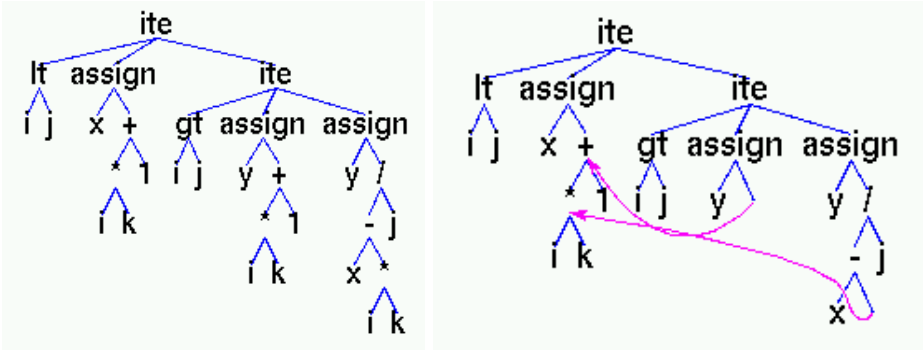
Entfernung redundanter Ausdrücke. Hier werden gleiche Teilausdrücke erkannt und ggf. an neue Variablen zugewiesen, die spätere Vorkommen der Teilausdrücke ersetzen. Die Erkennung gleicher Teilausdrücke kann man mit einem **Kollabierungsalgorithmus** durchführen, der z.B. zur Reduzierung von OBDDs verwendet wird (siehe [32], §6.2.1) und in Expander2 [60] implementiert ist. Die Variante dieses Algorithmus', die keine Blätter kollabiert, transformiert z.B. die Konjunktion von Gleichungen in Fig. 9.9 (1) in den Graphen (2). Liest man (1) als Zuweisungsfolge, die von links nach rechts ausgeführt wird, dann erhält man aus (2) die optimierte Zuweisungsfolge (3).



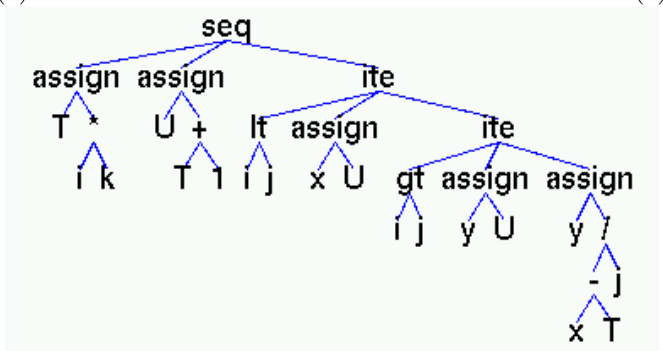
(1) (2) (3)

Figure 9.9.

Code hoisting erkennt gleiche Teilausdrücke basisblockübergreifend und stellt ggf. neue Zuweisungen den beteiligten Basisblöcken voran. Z.B. besteht der Syntaxbaum (1) in Fig. 9.10 aus drei Basisblöcken, weil jeder Zweig eines Konditionals (ite) einem eigenen Basisblock entspricht. Der Kollabierungsalgorithmus überführt (1) in den Graphen (2). In diesem erkennt man die gemeinsamen Teilausdrücke $i * k$ und $i * k + 1$, für die Variablen T und U eingeführt werden. Man erhält das optimierte Programmstück (3).



(1) (2)



(3)

Figure 9.10.

Entfernung redundanter Zuweisungen (dead code elimination). Eine Anweisung ist redundant, wenn sie niemals ausgeführt wird oder es keinen Unterschied macht, ob sie ausgeführt wird oder nicht. Wenn z.B. in einer Zuweisungsfolge x:=3;...;x:=4 die Variable x hinter der ersten Zuweisung tot ist (siehe §9.2.2), dann ist x:=3 toter Code und kann entfernt werden.

Algorithmus zur Entfernung redundanter Zuweisungen eines Basisblocks B. Sei M die Menge aller Variablen, an die in B zugewiesen wird. Setze Aux auf M. Durchlaufe B rückwärts und führe an jeder Zuweisung $x := t$ die folgende Anweisung aus: Ist $x \in Aux$, dann setze Aux auf $Aux \setminus \{x\} \cup free(t)$. Andernfalls entferne $x := t$ aus B. Beispiel: Der Basisblock $a := b*2;\ c := a*d;\ a := 5;\ c := b-d$ wird auf diese Weise zu $a := 5;\ c := b-d$ reduziert.

Basisblockübergreifend kann toter Code entstehen, wenn ein Zweig eines Konditionals oder der Rumpf einer Schleife nicht erreicht werden, weil die Fortpflanzung von Konstanten den zugehörigen Booleschen Ausdruck immer wahr oder immer falsch macht.

Schleifenentfaltung bezeichnet die Ersetzung einer Schleife durch eine äquivalente Sequenz von Schleifendurchläufen. Diese Optimierung lässt sich natürlich nur anwenden, wenn die Anzahl der Schleifendurchläufe konstant ist. Dies kann sich jedoch durch Konstantenfaltung (s.o.) ergeben haben.

Definition 9.3.1 Eine Variable, die in einer Schleife verwendet, dort aber nicht verändert wird, ist **schleifeninvariant**. Ein Ausdruck ist schleifeninvariant, wenn er nur schleifeninvariante Variablen enthält. Eine Variable, die am Anfang oder am Ende einer Schleife und nur dort verändert wird, heisst **Induktionsvariable**.

Export schleifeninvarianter Ausdrücke. Um die wiederholte Berechnung eines schleifeninvarianten Ausdrucks zu vermeiden, führt man wie beim code hoisting eine zusätzliche Variable ein und weist ihr den Ausdruck vor Eintritt in die Schleife zu.

Transformation von Induktionsvariablen. Taucht eine Induktionsvariable i in der zugehörigen Schleife ausser bei ihrer Veränderung $i := f(i,c)$ immer im selben Kontext $g(i)$ auf, dann kann man unter bestimmten Bedingungen eine neue Induktionsvariable x einführen, mit $g(i)$ initialisieren und i entfernen. Wir betrachten zwei Fälle:

- *Veränderung von i am Anfang der Schleife*

  L: i:=f(i,c); ...(∗)...; x:=g(i,a,b); if i R t then goto L

  Gilt $g(f(i,c),a,b) = h(g(i,a,c),a,b,c)$, dann ist die folgende Befehlsfolge zur obigen äquivalent:

  j:=g(i,a,b); L: i:=f(i,c); ...(∗)...; x:=j; j:=h(j,a,b,c); if i R t then goto L
- *Veränderung von i am Ende der Schleife*

  L: x:=g(i,a,b); ...(∗)...; i:=f(i,c); if i R t then goto L

  Gilt $g(f(i,c),a,b) = h(g(i,a,c),a,b,c)$, dann ist die folgende Befehlsfolge zur obigen äquivalent:

  j:=g(i,a,b); L: x:=j; ...(∗)...; j:=h(j,a,b,c); i:=f(i,c); if i R t then goto L

Die transformierten Schleifen sind zumindest dann effizienter, wenn h eine weniger aufwendige Operation als g ist. Beispiele: $g(i,a,b) = (a*i)+b,\ h(j,a,b,c) = j+(a*c)$ und $g(i,a,b) = (a**i)*b,\ h(j,a,b,c) = j*(a**c)$. Darüberhinaus kann in beiden Fällen die Zuweisung i:=f(i,c) entfernt und die Bedingung i R t durch j R k(t,a,b) ersetzt werden, wenn $i\ R\ t \Leftrightarrow g(i,a,b)\ R\ k(t,a,b)$ gilt und i im Programmstück $(*)$ nicht vorkommt. Beispiel: $R = >,\ t = 0,\ g(i,a,b) = (a*i)+b,\ k(t,a,b) = b.$

9.4 Codeerzeugung mit Registerzuteilung

Als erstes muss jede Zuweisung x:=t eines Basisblocks ggf. durch Einführung temporärer Variablen in eine Sequenz von 3-Adress-Befehlen zerlegt werden. Tut man das auf beliebige Weise und ordnet dann jede Variable einem Register zu, werden in der Regel mehr Register als nötig verbraucht. Deshalb werden zunächst mit dem folgenden Algorithmus[5] alle Knoten eines (Ausdrucks-)Baums t mit der Anzahl der Register markiert, die man zur Auswertung des jeweiligen Unterbaums benötigt (Funktion `label`, s.u.). Danach wird 3-Adress-Code erzeugt und dem jeweiligen Ergebnis der Auswertung ein Register zugeteilt, wobei Knoten mit großer Registerzahl priorisiert werden (Funktion `allocate`, s.u.).

Sei *maxReg* die Anzahl aller verfügbaren Register.

```
data Register = [0..maxreg]

data Tree a = Two a Tree Tree | One a Tree | Const a | Reg Register

label :: Tree Operation -> Tree (Operation,Int)
label (Two op t u) = if m == n then Two (op,1+m) t' u'
                               else Two (op,max m n) t' u'
                     where t' = label t
                           u' = label u
                           m = need t'
                           n = need u'
label (One op t)   = One (op,max 1 (need t')) t'
                     where t' = label t
label (Const op)   = Const (op,1)
label t            = t

need (Two (_,n) t u  = n
need (One (_,n) _) t = n
need (Const (_,n))   = n
need (Reg _)         = 0

data Instruction = Mk3 Operation Register Register Register |
                   Mk2 Operation Register Register |
                   -- Der Wert von Operation wird im ersten Register-Argument
                   -- von Mk2 bzw. Mk3 abgelegt.
                   Mk1 Operation Register |
                   Store Register | Load Register

allocate :: Tree (Operation,Int) -> Register -> ([Instruction],Register)
allocate (Two op t u) reg
  | m >= maxreg && n >= maxreg                              -- spilling case
                       = let (code1,reg1) = allocate t 0
                             (code2,reg2) = allocate u 0
                         in (code1++Store reg1:code2++
                             [Load 1,Mk3 op 0 1 reg2], 0)
```

[5] Haskell-Versionen der Algorithmen 11.10 und 11.11 aus [3], letzterer in der verbesserten Version [22]. Siehe auch [72], §2.

```
  | m >= n              = let (code1,reg1) = allocate t reg
                              (code2,reg2) = allocate u (reg+1)
                          in (code1++code2++[Mk3 op reg reg1 reg2], reg)
  | True                = let (code1,reg1) = allocate u reg
                              (code2,reg2) = allocate t (reg+1)
                          in (code1++code2++[Mk3 op reg reg2 reg1], reg)
                          where m = need t
                                n = need u
allocate (One op t) reg = (code++[Mk2 op reg reg1], reg)
                          where (code,reg1) = allocate t reg
allocate (Const op) reg = ([Mk1 op reg], reg)
allocate (Reg reg) reg' = ([], reg)
```

Der Aufruf `fst (allocate (label t) 0)` liefert 3-Adress-Code für t, der optimal ist bzgl. der Anzahl verwendeter Register. Die Registerzuteilung mit `allocate` ist auf Funktionsterme wie Tests oder zugewiesene Ausdrücke beschränkt, zerlegt diese aber in direkt ausführbaren 3-Adress-Code. Komplette Basisblöcke werden zuvor mit der im Folgenden beschriebenen **Registerzuteilung durch Graphfärbung** bezüglich der Anzahl benötigter Register optimiert.

Man bestimmt zunächst den **Interferenzgraph** IG des Basisblocks, das ist die Menge aller Paare von Variablen, deren Werte nicht im selben Register gehalten werden können, weil zum Beispiel an mindestens einem Knoten des zugehörigen Flussgraphen beide Variablen gleichzeitig lebendig sind. IG lässt sich induktiv berechnen:

- Ist L die Menge der am Eingangsknoten des Flussgraphen lebendigen Variablen, dann gehört die Menge der Paare (x, y) mit $x, y \in L$ zu IG.
- Für jede Kante $e = (n', n)$, die mit einem *move-Befehl* x:=y markiert ist, gehört die Menge der Paare (x, z) mit $z \in live(n) \setminus \{y\}$ zu IG, wobei $live(n)$ die Menge der bei n lebendigen Variablen ist (siehe §9.2.2).
- Für jede Kante $e = (n', n)$, die mit einer anderen Zuweisung an x markiert ist, gehört die Menge der Paare (x, z) mit $z \in live(n)$ zu IG.

Man entfernt nun schrittweise alle Knoten zusammen mit den jeweils adjazenten Kanten aus IG und legt sie in einem Keller ab. Die Knoten mit mindestens $maxreg$ adjazenten Kanten werden dabei als *Spillknoten* markiert. Dabei behält ein mit $maxreg$ vielen Farben färbbarer Graph diese Eigenschaft, da jeder entfernte und nicht als Spillknoten markierte Knoten höchstens $maxreg - 1$ viele Nachbarn hat, also zu seiner Färbung immer noch eine von seinen Nachbarn verschiedene Farbe zur Verfügung steht. Anschließend wird IG Knoten für Knoten in der umgekehrten Reihenfolge von deren Ablage im Keller wieder aufgebaut und dabei jedem eingefügten und nicht als Spillknoten markierten Knoten ein Register zugeordnet, das von den seinen Nachbarn zugeordneten Registern verschieden ist. Das ist möglich, weil ein solcher Knoten höchstens $maxreg - 1$ Nachbarn hat.

Literaturverzeichnis

[1] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers*, Addison-Wesley 1986

[2] Appel, *Compiling with Continuations*, Cambridge University Press 1992

[3] Appel, *Modern Compiler Implementation in ML*, Cambridge University Press 1998

[4] Arbib, Kfoury, Moll, *A Basis for Theoretical Computer Science*, Springer 1981

[5] J. Backus, *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, Comm. ACM 21 (1978) 613-641

[6] Banâtre, Jones, Le Métayer, *Prospects for Functional Programming in Software Engineering*, Springer 1991

[7] Bauer, Höllerer, *Übersetzung objektorientierter Sprachen*, Springer 1998

[8] R. Bird, *Introduction to Functional Programming using Haskell*, 2nd Edition, Prentice Hall 1998

[9] R. Bird, P. Wadler, *Introduction to Functional Programming*, Prentice-Hall 1988; deutsch: *Einführung in die funktionale Programmierung*, Hanser 1992

[10] M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier, *Compiling Rewrite Systems: The ASF+SDF Compiler*, ACM Transactions on Programming Languages and Systems 24 (2002) 334-368.

[11] M. Broy, *Informatik 1: Programmierung und Rechnerstrukturen*, Springer 1997 (Taschenbuch)

[12] M.M.T. Chakravarty, G.C. Keller, *Einführung in die Programmierung mit Haskell*, Pearson Studium 2004

[13] Davie, *An Introduction to Functional Programming Systems using Haskell*, Cambridge University Press 1992

[14] B. Courcelle, P. Franchi-Zannettacci, *Attribute Grammars and Recursive Program Schemes*, Theoretical Computer Science 17 (1982) 163-191 and 235-257

[15] K. Doets, J. van Eijck, *The Haskell Road to Logic, Maths and Programming*, King's College 2004

[16] M. Erwig, *Grundlagen funktionaler Programmierung*, Oldenbourg 1999

[17] C.N. Fischer, R.J. LeBlanc, *Crafting a Compiler*, Benjamin/Cummings 1988

[18] A.J. Field, P.G. Harrison, *Functional Programming*, Addison-Wesley 1988

[19] J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright, *Initial Algebra Semantics and Continuous Algebras*, J. ACM 24 (1977) 68-95

[20] G. Goos, W. Zimmermann, *Vorlesungen über Informatik 1: Grundlagen und funktionales Programmieren*, Springer 2005 (Taschenbuch)

[21] R.H. Güting, M. Erwig, *Übersetzerbau: Techniken, Werkzeuge, Anwendungen*, Springer 1999

[22] M. Hailperin, *The Sethi-Ullman Algorithm*, www.gustavus.edu/ max/courses/S1999/MC97/SethiUllman.ps

[23] C. Hall, J. O'Donnell, *Discrete Mathematics Using a Computer*, Springer 2000

[24] M. Hanus, *Implementierung logischer Programmiersprachen*, Vorlesungsskript, FB Informatik, TU Dortmund 1989

[25] H. Hofbauer, R. Kutsche, *Grundlagen des maschinellen Beweisens*, 2. Aufl., Vieweg 1991

[26] J.E. Hopcroft, R. Motwani, J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley 2001; deutsch: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*, Pearson Studium 2002

[27] J.E. Hopcroft, J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley 1979

[28] P. Hudak, J. Peterson, J.H. Fasel, *A Gentle Introduction to Haskell 98*, Report, 1999, siehe www.haskell.org

[29] P. Hudak, *The Haskell School of Expression: Learning Functional Programming through Multimedia*, Cambridge University Press 2000

[30] G. Huet, ed., *Logical Foundations of Functional Programming*, Addison-Wesley 1990

[31] J. Hughes, *Generalising Monads to Arrows*, Science of Computer Programming 37 (2000) 67-111

[32] M.R.A. Huth, M.D. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*, 2nd Edition, Cambridge University Press 2004

[33] G. Hutton, *Programming in Haskell*, Cambridge University Press 2007 *Meine Wahl als Lehrbuch für den Haskell-Kurs in Bachelor-Studiengang Informatik.*

[34] U. Kastens, *Übersetzerbau*, Oldenbourg, 1990

[35] H. Klaeren, M. Sperber, *Die Macht der Abstraktion: Einführung in die Programmierung*, Teubner 2007; *Funktional-algebraischer Zugang wie in diesem Skript. Allerdings wird hier die Erweiterung Scheme der m.E. etwas veralteten ungetypten Sprache Lisp verwendet.*

[36] D. Knuth, *Semantics of Context-Free Languages*, Mathematical Systems Theory 2 (1968) 127-145; Correction: Math. Systems Theory 5 (1971) 95-96; *Das erste Paper über attributierte Grammatiken!*

[37] H.-J. Kreowski, *Logische Grundlagen der Informatik*, Oldenbourg 1991 (in der Lehrbuchsammlung erhältlich)

[38] P.J. Landin, *The Mechanical Evaluation of Expressions*, BCS Computing Journal 6 (1964) 872-923

[39] J.W. Lloyd, *Foundations of Logic Programming*, 2. Auflage, Springer 1987

[40] J. Loeckx, K. Sieber, *The Foundations of Program Verification*, Teubner 1987

[41] J. Loeckx, K. Mehlhorn, R. Wilhelm, *Grundlagen der Programmiersprachen*, Teubner 1986

[42] Rita Loogen, *Integration funktionaler und logischer Programmiersprachen*, Oldenbourg 1995

[43] E.G. Manes, M.A. Arbib, *Algebraic Approaches to Program Semantics*, Springer 1986

[44] J. Meseguer, G. Rosu, *The Rewriting Logic Semantics Project*, Theoretical Computer Science 373 (2007) 213-237

[45] E.A. van der Meulen, *Deriving incremental implementations from algebraic specifications*, Proc. of the 2nd Int. Conf. on Algebraic Methodology and Software Technology, Workshops in Computing, Springer (1992) 277-286

[46] R. Milner, *A Communicating and Mobile Systems: the π-Calculus*, Cambridge University Press 1999

[47] B. Möller, H. Partsch, S. Schuman, eds., *Formal Program Development*, IFIP WG 2.1 State-of-the-Art Report, Springer LNCS 755 (1993)

[48] E. Moggi, *Notions of Computation and Monads*, Information and Computation 93 (1991) 55-92

[49] R.N. Moll, M.A. Arbib, A.J. Kfoury, *An Introduction to Formal Language Theory*, Springer 1988

[50] F.L. Morris, *Advice on Structuring Compilers and Proving Them Correct*, Proc. ACM POPL (1973) 144-152

[51] S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann 1997

[52] M. Müller-Olm, D. Schmidt, B. Steffen, *Model Checking: A Tutorial Introduction*, Proc. SAS '99, Springer LNCS 1694 (1999) 330-394

[53] M.P. Jones, J. Nordlander, B. v. Sydow, M. Carlsson, *A Survey of O'Haskell*, www.cs.chalmers.se/∼nordland/ohaskell/survey.html, 2001

[54] F. Nielson, H.R. Nielson, C. Hankin, *Principles of Program Analysis*, Springer 1999

[55] P. Padawitz, *Übersetzerbau*, Vorlesungsfolien, FK Informatik, TU Dortmund, fldit-www.cs.uni-dortmund.de/∼peter/CbauFolien.html

[56] P. Padawitz, *Grundlagen und Methoden funktionaler Programmierung*, Vorlesungsskript, FB Informatik, TU Dortmund 1999, fldit-www.cs.uni-dortmund.de/∼peter/ProgNeu.pdf

[57] P. Padawitz, *Formale Methoden des Systementwurfs*, Vorlesungsskript, FB Informatik, TU Dortmund, fldit-www.cs.uni-dortmund.de/∼peter/FMS.html

[58] P. Padawitz, *Fixpoints, Categories, and (Co)Algebraic Modeling*, TU Dortmund 2015

[59] P. Padawitz, *Swinging Types At Work*, Report, TU Dortmund 2000, fldit-www.cs.uni-dortmund.de/∼peter/BehExa.pdf

[60] P. Padawitz, *Expander2: A Formal Methods Presenter and Animator*, fldit-www.cs.uni-dortmund.de/∼peter/Expander2.html

[61] Ross Paterson, *Arrows and Computation*, in: J. Gibbons, O. deMoor, *The Fun of Programming*, Palgrave 2003

[62] L.C. Paulson, *ML for the Working Programmer*, 2nd Edition, Cambridge University Press 1996

[63] P. Pepper, P. Hofstedt, *Funktionale Programmierung*, Springer 2006

[64] S.L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall 1987

[65] S.L. Peyton Jones, P. Wadler, *Imperative Functional Programming*, Proc. POPL '93, ACM Press (1993) 71-84, research.microsoft.com/Users/simonpj/Papers/imperative.ps.Z

[66] F. Rabhi, G. Lapalme, *Algorithms: A Functional Programming Approach, Addison-Wesley 1999*; in der Lehrbuchsammlung unter L Sr 482/2

[67] C. Reade, *Elements of Functional Programming*, Addison-Wesley 1989

[68] H. Reichel, *An Algebraic Approach to Regular Sets*, in: K. Futatsugi et al., Goguen Festschrift, Springer LNCS 4060 (2006) 449-458

[69] U. Schöning, *Logik für Informatiker*, BI 1989

[70] Th. Schwentick, *Grundbegriffe der Theoretischen Informatik*, Folien zur gleichnamigen Vorlesung, TU Dortmund 2007, fldit-www.cs.uni-dortmund.de/∼peter/GTI07.pdf

[71] R. Sethi, *Programming Languages - Concepts and Constructs*, 2nd Edition, Addison-Wesley 1996

[72] R. Sethi, J.D. Ullman, *The Generation of Optimal Code for Arithmetic Expressions*, Journal of the ACM 17 (1970) 715-728

[73] G. Smolka, *Programmierung: Eine Einführung in die Informatik mit Standard ML*, Oldenbourg 2008 *Funktional-algebraischer Zugang wie in diesem Skript. Hier wird zwar nicht Haskell, aber die sehr ähnliche, etwas ältere funktionale Sprache Standard ML verwendet. Enthält viele Themen aus dem Übersetzerbau.*

[74] J.W. Thatcher, E.G. Wagner, J.B. Wright, *More on Advice on Structuring Compilers and Proving Them Correct*, Theoretical Computer Science 15 (1981) 223-249

[75] J.W. Thatcher, J.B. Wright, *Generalized Finite Automata Theory with an Application to a Decision Problem of Second-Order Logic*, Theory of Computing Systems 2 (1968) 57-81

[76] P. Thiemann, *Grundlagen der funktionalen Programmierung*, Teubner 1994

[77] S. Thompson, *Haskell: The Craft of Functional Programming*, Addison-Wesley 1999

[78] R. Turner, *Constructive Foundations for Functional Languages*, McGraw-Hill 1991

[79] E. Visser, *Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems*, in: C. Lengauer et al., eds., Domain-Specific Program Generation, Springer LNCS 3016 (2004) 216-238

[80] P. Wadler, *Monads for Functional Programming*, in: M. Broy, ed., Program Design Calculi, Springer 1993

[81] H. Wagner, *Logische Systeme der Informatik*, Vorlesungsskript, FB Informatik, TU Dortmund 2000

[82] W.M. Waite, G. Goos, *Compiler Construction*, Springer 1983

[83] D. Watt, *Programming Language Concepts and Paradigms*, Prentice Hall 1990; deutsch: *Programmiersprachen - Konzepte und Paradigmen*, Hanser 1996

[84] I. Wegener, *Theoretische Informatik*, Teubner 1992

[85] R. Wilhelm, D. Maurer, *Übersetzerbau*, 2. Auflage, Springer 1997