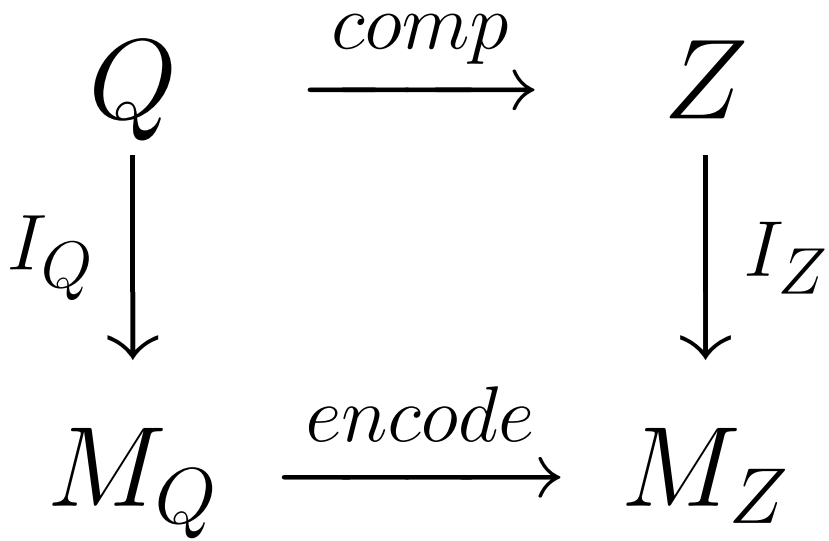
Übersetzerbau – Wintersemester 2007/2008



Peter Padawitz

Überblick auf die Inhalte der Vorlesung

Inhalt der Lehrveranstaltung sind Konzepte und Methoden der Übersetzung imperativer und funktionaler Programmiersprachen.

Der Weg vom Quell- zum Zielprogramm durchläuft mehrere Phasen:

- **Lexikalische Analyse:** Die Transformation von *Zeichen-* in *Symbolfolgen* (Kapitel 2)
- **Syntaxanalyse:** Überführung der Symbolfolge in eine *baumartige Termstrukur* bzw. ein Programm in *abstrakter Syntax* (Kapitel 3)
- **Semantische Analyse:** *Attributierung* der Termstruktur mit semantischen Informationen, die vom Compiler benötigt werden (Kapitel 4)

Überblick auf die Inhalte der Vorlesung

- **Codeerzeugung:** Abbildung der Termstruktur in das Zielprogramm, i.a. eine lineare Befehlsfolge (Kapitel 5)
- **Codeoptimierung:** Datenflussanalyse, Codeerzeugung mit Registerzuteilung (Kapitel 8)

Die Algorithmen, die für die jeweiligen Phasen grundlegend sind, werden in der funktionalen Sprache **Haskell** formuliert.

☞ Als Verfahren zur **Symbol- und Termmanipulation** ist Haskell besonders geeignet – ohne die sonst üblichen Implementierungsdetails – die Aufmerksamkeit auf die Algorithmen und Konzepte zu konzentrieren.

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 transformieren.

Eigenschaften von Progammiersprachen

- **Quellsprache**: benutzerfreundlich, leicht modifizierbar, enthält komplexe Konstrukte
- **Zielsprache**: schlecht lesbar, kaum modifizierbar, enthält nur einfache Konstrukte, maschinenorientiert

Im Folgenden bezeichnen Q und Z die Mengen der Programme der Quell- bzw. Zielsprache.

Definition der Übersetzer-Funktion $comp$

Definition 1.1.1 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:

$$\begin{array}{ccc} Q & \xrightarrow{comp} & Z \\ {\scriptstyle I_Q}\downarrow & & \downarrow{\scriptstyle I_Z} \\ M_Q & \xrightarrow{encode} & M_Z \end{array}$$

- M_Q und M_Z sind mathematische Strukturen, die die **Bedeutung (Semantik)** von Programmen aus Q und Z widergeben.
- $I_Q : Q \to M_Q$ und $I_Z : Z \to M_Z$ sind **Interpretationsfunktionen** (*evaluation functions*), die jedem Programm seine Bedeutung zuordnen.

Abstrakte Maschinen und Zustandstransformationen

Die Mengen M_Q und M_Z werden oft als **abstrakte Maschinen** formuliert; abstrakte Maschinen bestehen aus:

- **Zustandstransformationen**, d.h. Funktionen $f_i : S \to S$ auf einer *Zustandsmenge* S (*states*).

Ein Zustand besteht in der Regel aus vielen **Komponenten**:

- *Eingabekomponenten* (E),
- *internen Attributen* und
- *Ausgabekomponenten* (A).

☞ Manchmal werden Quell- und oder Zielprogramm selbst als Zustandskomponenten aufgefasst und die so angereicherten Zustände **Konfigurationen** genannt.

Definition der Interpreter-Funktion $eval$

Definition 1.1.2 Sei

$$M_Q = [S \to S] \text{ (=}\textit{Menge der Funktionen von } S \textit{ nach } S\text{)}.$$

Eine partielle Funktion:

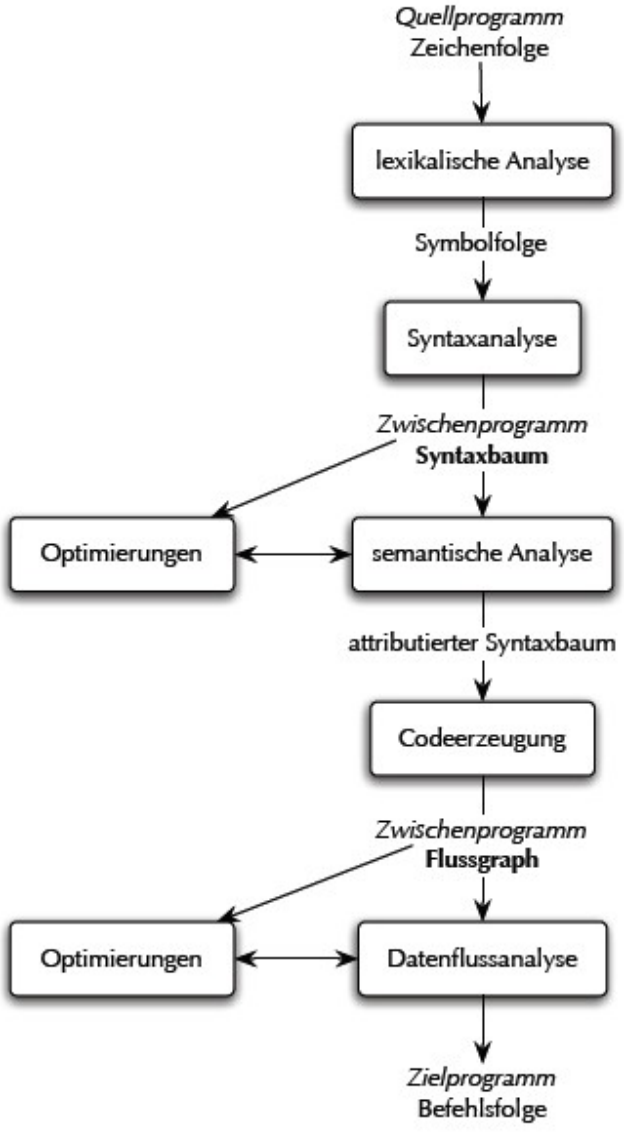
$$eval : Q \times E \to A$$

heißt **Interpreter** von Q, wenn für alle $q \in Q$ und $e \in E$ gilt:

$$eval(q, e) = output(\underbrace{I_Q(q)}_{\in(S \to S)} (\underbrace{input(e)}_{\in S})).$$

Dabei ist $input : E \to S$ eine injektive **Einbettungsfunktion** in die Zustandsmenge sowie $output : S \to A$ symmetrisch dazu die surjektive **Projektion** in die Menge der Ausgabekomponenten (A).

Aufbau eines Übersetzers



Kurze Einführung in Haskell

Wir beschränken uns hier auf die Konstrukte von Haskell, die in späteren Algorithmen benutzt werden.
Für weitergehende Informationen zur funktionalen Programmierung mit Haskell (Tutorials und frei verfügbarer Software) sei auf folgende Informationsquellen hingewiesen:

- Die Haskell-Homepage mit vielen Informationen zum Sprachstandard, verfügbare Software u.v.m. (`www.haskell.org`)
- Der online-Haskell-Kurs von Ralf Hinze (`www.informatik.uni-bonn.de/~ralf/`)
- O'HUGS, eine objektorientierte Erweiterung von Haskell – empfohlen für die Vorlesung (`www.cs.chalmers.se/~nordland/ohaskell/`):

Basistypen in Haskell (Auswahl)

Zu den vordefinierten Basistypen in Haskell gehören:

- **`Bool`** die Menge {`True, False`}
- **`Int`** ganze Zahlen mit beschränkter Genauigkeit
- **`Integer`** ganze Zahlen mit *unbeschränkter* Genauigkeit
- **`Float`** Fließkommazahlen (einfache Genauigkeit)
- **`Double`** Fließkommazahlen (doppelte Genauigkeit)
- **`Ratio`** Rationale Zahlen (Notation: `1%2,1%3...`)
- **`Char`** ASCII-Zeichen (Notation: `'A','B',...`)
- **`String`** Zeichenketten, Listen von Zeichen (Chars)
  ➥ (Notation: `"Hallo"` oder `['H','a','l','l','o']`)
- **`()`** (*unit-Typ*) spezieller Wertebereich für Funktionen

Basistypen in Haskell – Typisierung

Um den Typ eines Ausdrucks explizit festzulegen, kann der `::`-Operator eingesetzt werden. Zum Beispiel wird mit

```
2::Double
```

der Typ der Konstanten 2 als `Double` bestimmt. Insbesondere in der **Typ-Signatur-Deklaration** (*type signature declaration*) von Funktionen wird der Typ-Operator verwendet:

```
substring::String -> Int -> Int -> String
```

ist ein Beispiel für die Signatur-Deklaration einer Funktion `substring` vom Typ:

```
String -> Int -> Int -> String
```

- ☞ Haskell sorgt dafür, dass nur entsprechend getypte Ausdrücke akzeptiert werden.

Funktionen auf den Basistypen

Die wichtigsten Funktionen auf dem Basistyp `Bool` sind

- **Und**-Verknüpfung:

  `(&&) :: Bool -> Bool -> Bool.`

  Beispiel: `True && False` $\longrightarrow$ `False`
- **Oder**-Verknüpfung:

  `(||) :: Bool -> Bool -> Bool.`

  Beispiel: `True || False` $\longrightarrow$ `True`
- **Negation**:

  `not :: Bool -> Bool.`

  Beispiel: `not (1<0)` $\longrightarrow$ `True`

Funktionen auf `String`: Weil eine Zeichenkette in Haskell ein Spezialfall des *Listentyps* ist, sind alle später vorgestellten allgemeinen Listenfunktionen auch auf Zeichenketten anwendbar.

Funktionen auf den Basistypen

Zu den vordefinierten Funktionen für die numerischen Basistypen (`Int`, `Integer`, `Double`, etc.) gehören

- **arithmetische Funktionen**:
  - Addition, Subtraktion: `+`, `-`
  - Multiplikation, Division: `*`,`/`
  - Wurzel-, Potenzfunktion: `sqrt`, `^`
  - Exponential-, Logarithmusfunktion: `exp`, `log`

  sowie
- **zahlentheoretische Funktionen**:
  - Modulo und ganzzahlige Division: `mod`,`div`
    (Beachte: InfixNotation – Beispiel: `23 ´mod´ 5`)
  - Gerade-/Ungerade-Test: `even`, `odd`

und viele andere Funktionen mehr, z.B. die Betragsfunktion `abs`, die trigonometrischen Funktionen `sin`, `cos`, `tan`, ... und natürlich die Vergleichsoperatoren: `<`,`>`,`>=`,`<=`,`==`

Zusammengesetzte Typen – Produkttypen

Als zusammengesetzte Typen verwenden wir **Produkttypen**: Zum Beispiel kann das Tupel `(5,True,'A')` als Element des Typs `Integer` $\times$ `Bool` $\times$ `Char` auftreten.

In Haskell sind **Tupel beliebiger Stelligkeit*** vordefiniert:

- `()` das leere Tupel
- `(A,B)` Menge der Paare aus `A` $\times$ `B`
- `(A,B,C)` Menge der Tripel aus `A` $\times$ `B` $\times$ `C`
- `(A1,A2,...,An)` Menge der n-stelligen Tupel aus `A1` $\times$ `A2` $\times \ldots \times$ `An`

* ('Einstellige Tupel' sind aus naheliegenden Gründen nicht erlaubt!)

☞ In Tupeln werden eine *feste Zahl von Werten* in der Regel *unterschiedlicher Typen* zusammengefasst.

Funktionen auf Produkttypen

Als Funktionen auf zweistelligen Tupeln sind die beiden Funktionen

```
fst :: (a,b) -> a        snd :: (a,b) -> b
fst (x,y) = x            snd (x,y) = y
```

standardmäßig vordefiniert. Die Defintion der beiden Funktionen erfolgt hier mit Hilfe von **Datenmustern** (*patterns*):

> Der **Konstruktor**, d.i. eine Funktion, mit der zweistellige Tupel syntaktisch aufgebaut werden:
>
> `(_,_)`
>
> bildet zusammen mit Variablen (hier: `x,y`) ein Muster, anhand dessen Haskell über die Anwendung der entsprechenden Funktionsgleichung entscheidet.

Listentypen als Datenstruktur

Die wichtigste Datenstruktur in Haskell sind **Listen**. Als Datentyp treten Listen daher in verschiedenen Zusammenhängen in Erscheinung, z.B. als

- `[]` leere Liste,
- `['A','B','C']` eine Liste von Elementen aus `Char` – äquivalent zum Ausdruck `"ABC"`,
- `[1..]` die Liste aller ganzen Zahlen ≥ 1,
- `[1,3..99]` die Liste aller ungeraden Zahlen bis einschließlich 99,
- `[(64,'A'),(65,'B'),(66,'C'),(67,'D')]` eine Liste von Tupeln des Typs $(\texttt{Int} \times \texttt{Char})$.

☞ In Listen wird eine *beliebige Anzahl* von Werten *desselben Typs* zusammengefasst.

Listentypen als Datenstruktur

Für Listen gibt es in Haskell ein Unzahl von Funktionen, von denen hier einige wichtige vorgestellt werden sollen:

- Die (Listenkonstruktor-)* Funktion

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

  fügt ein Element *vorne* in eine gegebene Liste ein. Beispiel:

  ```
  1: [2,3,4] ⟶ [1,2,3,4]
  ```

- Mit der Funktion

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

  werden zwei gegebene Listen konkateniert:

  ```
  [1,2] ++ [3,4] ⟶ [1,2,3,4]
  ```

* Besondere Bedeutung hat die Konstruktoreigenschaft dieser Funktion für die musterbasierte Fallunterscheidung.

Datenmuster und Definition von Funktionen

Ein Haskell-Programm besteht im Wesentlichen aus einer Folge von **Funktionsgleichungen** der Form:

$$\begin{aligned} \texttt{f}\ p_1\ \ldots\ p_k \mid be = \textbf{let}\ & q = e_1 \\ & \ldots \\ & q_n = e_n \\ \textbf{in}\ & e_0 \end{aligned}$$

wobei `f` für Argumente definiert wird, die zum Datenmuster `p1,...,pk` passen und die Bedingung be erfüllen.

Der boolescher Ausdruck, der mit | eingeleitet wird, heißt auch **Wächter** (guard). Ist die Bedingung erfüllt, wird `f` zu e_0 ausgewertet.

Mit dem **`let`** ... **`in`** -Konstrukt können innerhalb der Gleichung **lokale Definitionen** von Variablen eingeführt werden.

Datenmuster und Definition von Funktionen

Äquivalent zu der oben angegebenen Funktionsdefinition ist folgende Gleichung, wobei **`where`** statt **`let`** verwendet wird:

$$\begin{aligned} \mathbf{f}\ p_1\ \ldots\ p_k \mid be &= e_0 \\ \mathbf{where}\ q_1 &= e_1 \\ \ldots & \\ q_n &= e_n \end{aligned}$$

Beispiel. Die Funktion `f1` ersetzt alle Elemente $x < 0$

```
f1 [] = []
f1 (x:xs)  | x>0        = x : f1 xs
           | x<0        = (-x) : f1 xs
           | otherwise = f1 xs
```

einer Liste durch ihren Betrag und löscht alle Nullen.

Datenmuster und Definition von Funktionen

Fallunterscheidungen mit booleschen Ausdrücke können auch – wie sonst üblich – im Kontext von **Konditionalen** verwendet werden:

$$\texttt{if}\ be\ \texttt{then}\ e1\ \texttt{else}\ e2.$$

Hier ist *if-then-else* eine (implizite) Funktion vom Typ

```
(Bool,a,a) -> a.
```

Beispiel:

```
signum x = if (x > 0)
           then 1
           else if (x == 0)
                then 0
                else (-1)
```

Verwendung von Datenmustern

Datenmuster (**patterns**) sind aus Variablen und Konstruktoren bestehende Ausdrücke.
Der *Unterstrich* steht für Teile eines Musters, auf die der Ausdruck auf der rechten Seite der jeweiligen Gleichung keinen Bezug nimmt:

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

definiert die Funktion `null`. Man kann stattdessen auch Variablen bzw. Konstruktoren für das Muster benutzen:

```
null (a:ax)     =  False
null []         =  True
```

☞ Bei Verwendung des Unterstrichs wird aber sofort klar, von welchen Argumenten eine Funktion unabhängig ist!

Verwendung von Datenmustern

Mit Hilfe von Datenmustern können **Fallunterscheidungen** formuliert werden: Für die Funktion

```
null :: [a]      ->  Bool
null (a:ax)      =   False
null []          =   True
```

die feststellt, ob eine Liste leer ist oder nicht, gilt:

`null []` $\rightarrow$ `True` *und* `null [1,3,4,23]` $\rightarrow$ `False`

Die Funktion liefert also

- auf dem Muster `[]` den Wert $True$,
- während sie auf allen Listen, die zum Muster `x:s` – das ist im Beispiel das Muster `1:[3,4,23]` – passen, den Wert $False$ zurückgibt.

Funktionen auf Listentypen

- Die Länge einer Liste liefert

  ```
  length :: [a] -> Int
  ```

  Beispiel:

  ```
  length [1,3..99]  ⟶ 50
  ```

- Ob ein Element x in einer Liste enthalten ist, prüft man mit

  ```
  elem :: Eq a => a -> [a] -> Bool
  ```

  Beispiel:

  ```
  elem 'A' ['A','B','D']  ⟶ True
  ```

Funktionen auf Listentypen

- Das erste Element einer Liste liefert die Funktionen

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

  Beispiel:

$$\texttt{head [1,2,3,4]} \longrightarrow \texttt{1}\ ,$$

- Den Rest einer Liste liefert die Funktionen

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

  Beispiel:

$$\texttt{tail [1,2,3,4]} \longrightarrow \texttt{[2,3,4]}$$

Funktionen auf Listentypen

- Für eine gegebene Liste erhält man mit

```
init :: [a] -> [a]
init [_]   = []
init (x:s) = x:init s
```

die Liste *ohne* das letzte Element.
Beispiel:

`init ['1','2','3']` $\longrightarrow$ `"12"`

Funktionen auf Listentypen

- Für eine gegebene Liste erhält man mit

```
last :: [a] -> a
last [x]    = x
last (_:s) = last s
```

das letzte Element der Liste.
Beispiel:

`last` ['A','B','C'] $\longrightarrow$ 'C'

Funktionen auf Listentypen

- Für eine gegebene Liste erhält man mit

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

die ersten n Elemente einer Liste.
Beispiel:

`take 3 ['A','B'..'Z']` $\longrightarrow$ `"ABC"`

Funktionen auf Listentypen

- Für eine gegebene Liste erhält man mit

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

  die Liste *ohne* die ersten n Elemente.
  Beispiel:

$$\texttt{drop 23 ['A','B'..'Z']} \longrightarrow \texttt{"XYZ"}$$

Die Funktionen `map`, `foldl` und `flip` werden später im Zusammenhang mit Funktionen höherer Ordnung vorgestellt.

Funktionen auf Listentypen

- Für eine gegebene Liste erhält man mit

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

das n-te Element der Liste.
Beispiel:

```
['A','B','C','D'] !! 0 ⟶ 'A'
['A','B','C','D'] !! 2 ⟶ 'C'
```

Funktionen auf Listentypen

- Zwei Listen werden mit

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

zu einer Liste von Paaren der Elemente der beiden Listen in gegebener Reihenfolge zusammengefasst. Beispiel:

```
zip [1..3]['A'..'Z'] →
[(1,'A'),(2,'B'),(3,'C')]
```

(☞ Die zwei Listen können unterschiedlich lang sein!)

Funktionen auf Listentypen

- Für eine gegebene Liste erhält man mit

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

eine Teilliste, beginnend ab Position i, bis hin zur Position j.
Beispiel:

`sublist` ['A','B','C','D'] 2 3 $\longrightarrow$ "CD"

Funktionen auf Listentypen

- Mit

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

  wird eine unendliche Liste erzeugt.

- Mit

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

  wird eine endliche Liste von n Elementen erzeugt.

Infix und Präfix

Wie der Konstruktor `(:)` werden auch der Zugriff `(!!)` auf einzelne Listenelemente 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
```

Infix und Präfix

Jede Funktion eines Typs `a -> b -> 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.
- Mit einem Kleinbuchstaben beginnende Funktionssymbole wie `mod` werden in Hochkommas ' eingeschlossen.

  (Mit einem Großbuchstaben beginnende Zeichenfolgen interpretiert Haskell grundsätzlich als Typen, Typenklassen bzw. Konstruktoren! Mit einem Kleinbuchstaben beginnende Zeichenfolgen interpretiert Haskell grundsätzlich als Typ- bzw. Elementvariablen!)

Funktionen auf speziellen Listen

- Für eine Liste von Zeichenketten (Strings) liefert

  ```
  unwords :: [String] -> String
  ```

  die Konkatenation der Zeichenketten, wobei zwischen den Zeichenketten jeweils Leerzeichen eingefügt werden werden. Beispiel:

  `unwords ["eins", "zwei"]` $\longrightarrow$ `"eins zwei"`

  Die Umkehrfunktion zu `unwords` ist

  ```
  words :: String -> [String]
  ```

  und trennt eine Zeichenkette entsprechend der Leerzeichen in eine Liste von Strings auf. Beispiel:

  `words "eins zwei"` $\longrightarrow$ `["eins","zwei"]`

Listenbeschreibungen – Listenkomprehension

Häufig lassen sich Funktionen einfacher und verständlicher mit Hilfe von **Listenbeschreibungen** (*list comprehensions*) definieren. Als Beispiel für eine Listenbeschreibung, kann der folgende Ausdruck:

```
[f x | x <- xs]
```

gelesen werden als: „Liste aller `f x`, so dass `x` in der Liste `xs` ist“.

- Das Konstrukt `x <- xs` wird als **Generator** bezeichnet, und es muss gelten: Wenn die Variable `x` vom Typ typ_i ist, dann muss `xs` vom (Listen-)Typ $[typ_i]$ sein.
- Der Querstrich | trennt den Ausdruck auf seiner linken Seite vom sogenannten **Qualifier** auf der rechten Seite. Ein Qualifier ist eine Folge von Generatoren und/oder **Wächtern** (*guards*). Wächter sind boolesche Ausdrücke.

Listenbeschreibungen – Listenkomprehension

Falls im Qualifier der Listensbeschreibung mehr als ein Generator steht, dann werden die Elemente für den Ausdruck entsprechend der Reihenfolge der zugehörigen Generatoren verschachtelt erzeugt. Zum Beispiel erhält man mit der folgenden Listenbeschreibung:

```
[(x,y) | x <- ['A'..'C'], y <- [1..2]]
```

ein Liste von Paaren:

```
[('A',1),('A',2),('B',1),('B',2),('C',1),('C',2)]
```

Ein weiteres Beispiel für eine Listenbeschreibung, in dem zusätzlich ein boolescher Ausdruck als **Wächter** (*guard*) verwendet wird:

```
[(x,y) | x <- [1..10], y <- [1..10], x+y == 7 ]
```

erzeugt die Liste aller Paare, deren Summe gleich 7 ist.

Definition von Funktionen höherer Ordnung

Analog zu den obigen Beispielen für Funktionsdefinitionen können **Funktionen höherer Ordnung** definiert werden.

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

`map` wendet eine Funktion $f : a \rightarrow b$ auf alle Elemente einer Liste vom Typ `[a]` an und liefert eine Liste vom Typ `[b]`. **Beispiel:**

`map (+1) [1..5]` $\rightarrow$ `[2,3,4,5,6]`

Definition von Funktionen höherer Ordnung

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

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

Definition von Funktionen höherer Ordnung

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

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

`foldl (+) 0 [1..10]` $\to$ `55`

`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
```

Definition von Funktionen höherer Ordnung

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

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

Definition von Funktionen höherer Ordnung

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
```

Definition von Funktionen höherer Ordnung

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

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

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

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

Definition von Funktionen höherer Ordnung

Soll das Anfangselement mit dem Kopf der Liste übereinstimmen, dann können nur nichtleere Listen 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)
```

Definition von Funktionen höherer Ordnung

`any` und **`all`** implementieren die Quantoren auf Listen, in dem sie prüfen, ob die Boolesche Funktion $f : a \rightarrow Bool$ für ein bzw. alle Ellemente 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 /=)
```

Definition von Funktionen höherer Ordnung

`filter` erzeugt die Teilliste aller Elemente einer Liste, für die die Boolesche Funktion $f : a \rightarrow Bool$ *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]
```

Definition von Funktionen höherer Ordnung

Der Aufruf **`primes (n)`** filtert z.B. alle Primzahlen aus der Liste der 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 _     = []
```

Definition von Funktionen höherer Ordnung

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.

Definition von Funktionen höherer Ordnung

Mit einer lokalen Definition geht's gleich viel schneller und sieht's auch eleganter aus:

```
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)
```

Definition von Funktionen höherer Ordnung

`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)
```

Definition von Funktionen höherer Ordnung

`sorted : [a] -> Bool` stellt fest, ob eine Liste aufsteigend sortiert ist:

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

Erst im Fall einer mindestens zweielementigen Liste ist ein rekursiver Aufruf nötig. Dies ist der gleiche Fall bei Sortieralgorithmen:

```
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)] -> Bool`,

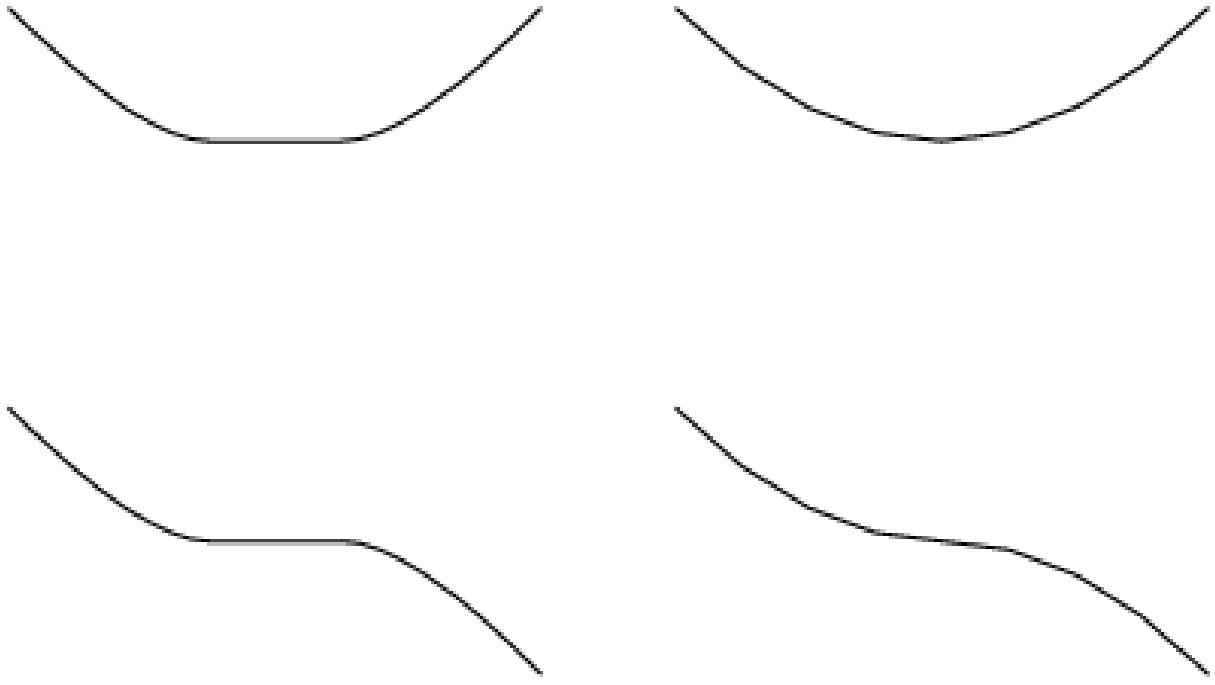
die feststellt, ob alle Elemente einer Punktliste auf einer Geraden liegen, setzt die Rekursion sogar erst auf mindestens dreielementige Listen ein:

```
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)
```

Zwei geglättete Kantenzüge vor und nach der Reduzierung



Reduce und Flip

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
```

Der Umordnung der Argumente einer Funktion höherer Ordnung dient die Funktion

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

Curryfizierung von Funktionen

Häufig wird in Funktionsdefinitionen von der Möglichkeit Gebrauch gemacht, durch Klammern strukturierte Argumente, wie z.B. in

```
fun :: (x,y) -> z
```

durch eine Sequenz einfacher Argumente zu ersetzen:

```
fun :: x -> y -> z
```

Dieses Verfahren heißt **Curryfizierung** (*currying*), nach dem Amerikanischen Logiker Haskell B. Curry.

Vorteile der Curryfizierung von Funktionen:

- Die Anzahl von Klammern in den Ausdrücken wird reduziert.
- Die curryfizierte Funktion kann *partiell* mit Argumenten instanziiert werden, so dass man wieder eine Funktion erhält.

Curryfizierung von Funktionen

Um eine nichtcurryfizierte Funktion in eine curryfizierte zu konvertieren gibt es die Funktion

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f(x,y)

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

Beispielsweise kann die folgende Funktion:

```
smaller :: (Int, Int) -> Int
smaller(x,y) = if x <= y then x else y
```

ersetzt werden durch eine curryfizierte:

```
smallerc = curry smaller
```

`smallerc` ist damit eine Funktion, die auf das Argument `x` angewendet, eine Funktion vom Typ `Int -> Int` zurückgibt.

Auswertung einer Funktion

Der Wert einer definierten Funktion ergibt sich aus dem Ausdruck auf der rechten Seite der $ersten$ Gleichung ihrer Definition derart, dass der aktuelle Funktionsparameter auf das Muster der linken Seite passt.

Das allgemeine Schema einer Funktionsdefinition lautet:

$$
\begin{array}{l}
\texttt{f}\ p_{11}\ \ldots\ p_{1n}\ \texttt{=}\ e_1 \\
\vdots \\
\texttt{f}\ p_{k1}\ \ldots\ p_{kn}\ \texttt{=}\ e_k \\
\texttt{f}\ \texttt{\_}\ \ldots\ \texttt{\_}\ \texttt{=}\ e_{k+1}
\end{array}
$$

- Hierbei ist $(p_{i1}, ..., p_{in})$, $1 \leq i \leq k$, das zum Wert e_i gehörige Argumentmuster.
- $f(a_1, ..., a_n)$ hat den Wert e_{k+1}, wenn $(a_1, ..., a_n)$ auf keines der vorherigen Muster passt.

Auswertung einer Funktion

Eine semantisch äquivalente Definition von f erhalten wir mit dem case-Konstrukt:

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

Konstruktoren und Datentypdefinition

Konstruktoren sind Konstanten und Funktionen, mit denen Daten aufgebaut werden.
Beispiele für Konstruktoren sind

- *Zahl-*, *String-* und *Boolesche Konstanten*,
- die *Listenkonstante* `[]`, *die Listenfunktion* `:`,
- der *Tupelkonstruktor* `(...)` und der Funktionspfeil `->`.

Man verwendet Konstruktoren

- in **Datenmustern** (*patterns*) zur Fallunterscheidung, z.B. bei der Funktionsdefinition für verschieden aufgebaute Argumente
- und für die **Definition neuer Datentypen**, indem man mit Hilfe der entsprechenden Konstruktoren (*data constructors*) angibt, wie Daten dieses neuen Typs aufgebaut sein sollen.

Konstruktoren und Datentypdefinition

Zur **Definition neuer Typen** stellt Haskell das **`data`**-Konstrukt zur Verfügung. Zum Beispiel wird mit:

```
data Farbe = Rot | Gelb | Blau
```

ein neuer Datentyp `Farbe` definiert, wobei `Rot, Gelb, Blau` die zugehörigen Konstruktoren sind.

☞ Achtung: Nur Bezeichner von Typen und Konstruktoren werden in Haskell großgeschrieben!

Das **allgemeine Schema einer Typdefinition** lautet:

$$\texttt{data dt = } Con_1\ typ_1 \mid ... \mid Con_n\ typ_n$$

Die Daten des Typs `dt` sind gerade die aus den Konstruktoren $Con_1, ..., Con_n$ aufgebauten Ausdrücke.

Konstruktoren und Datentypdefinition

Ein Konstruktor Con_i ist eine Funktion des Typs

$$typ_i \rightarrow \texttt{dt} \ .$$

Demzufolge ist ein Ausdruck der Form boldmath $Con_i(e_1, ..., e_k)$ dann und nur dann ein wohldefiniertes Objekt vom Typ `dt`,

- wenn $(e_1, ..., e_k)$ den Typ typ_i hat.

Im obigen Beispiel des neu definierten Datentyps `Farbe` sind die Konstruktoren **nullstellig**, d.h. Konstanten. Also entspricht die Menge:

$$\{\texttt{Rot, Gelb, Blau}\}$$

gerade der Menge der wohldefinierten Elemente des Datentyps `Farbe`.

Konstruktoren und Datentypdefinition

Die Konstruktoren `Rot`,`Gelb` und `Blau` im obigen Beispiel waren nullstellig. Durch

```
data Point = P Int Int
```

wird der Datentyp `Point` mit einem zweistelligen Konstruktor `P` vom Typ `Int` $\times$ `Int` definiert. In Typkonstruktoren und Funktionsdefinitionen können auch **polymorphe Typen** auftreten; in diesem Fall werden statt der Typnamen sogenannte **Typvariablen** notiert. Zum Beispiel ist:

```
data Point a b = P a b
```

eine Verallgemeinerung des Datentyps `Point`.

Haskell-Beispiele

Beispiel 1.2.2 Symbolische Differentiation

Wir definieren einen neuen Datentyp `Expr` mit

```
data Expr =  Con Int | Var String | Sum [Expr] | Prod [Expr]
```

Mit dem Datentyp `Expr` können wir z.B. den Ausdruck

$$5 * (x + 2 + 3)$$

in abstrakter Syntax darstellen als:

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

Haskell-Beispiele

Wie definieren nun `d` als eine *Funktion zweiter Ordnung* für die symbolische Differentiation in Abhängigkeit vom Muster der abstrakten Ausdrücke:

```
d :: String -> Expr -> Expr
d x (Con _)       = Con 0
d x (Var y)       = if x == y then Con 1 else Con 0
d x (Sum s)       = Sum (map (d x) s)
d x (Prod [])     = Con 0
d x (Prod [e])    = d x e
d x (Prod (e:s)) = Sum [Prod [d x e,p], Prod [e,d x p]]
                     where p = Prod s
```

Haskell-Beispiele

```
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
```

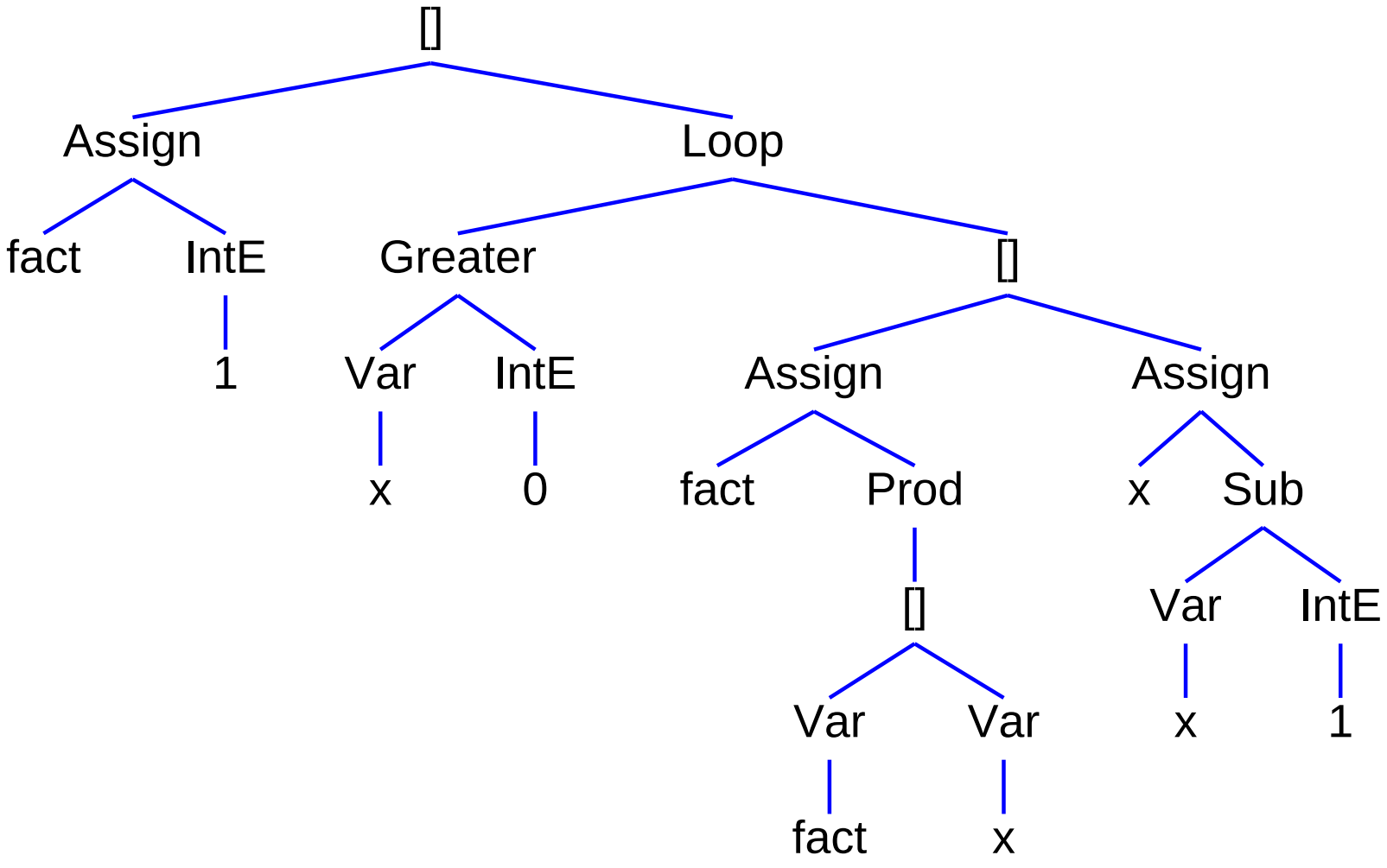
Das imperative Programm

```
{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))]]
```

Baumdarstellung von `prog`



Interpreter

Ein Interpreter wertet die abstrakten Ausdrücke aus, in Abhängigkeit von einer Belegung $(valuation)$ st des Typs $State = (String \to Int)$ ihrer 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     = st' where st' y | x == y = evalInt e st
                                             | True   = st y
evalCom (Cond e cs cs') st = if evalBool e st
                             then evalBlock cs st
                             else evalBlock cs' st
```

Interpreter

```
evalCom (Loop e cs) st     = evalCom (Cond e
                                      (cs++[Loop e c]) []) 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 (map (flip evalInt st) es)
evalInt (Prod es) st   = product (map (flip evalInt st) 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)
```

Interpreter

Zum Beispiel liefert

```
evalBlock prog st "fact" where st "x" = 4
```

den Wert 24.

- Mit Datentypen wie $IntE$, etc. werden in Haskell Syntaxbäume implementiert. Diese wiederum bilden den Wertebereich eines Parsers.
- Interpreter und Compiler hingegegen haben Syntaxbäume im Definitionsbereich und erlauben deshalb eine rekursive Definition entlang der Baumstruktur.
- Aber auch die Definition eines Parsers kann nach einem Schema erfolgen, das die Baumstruktur ausnutzt.

Baumdarstellung

- Die Baumdarstellung von `prog` basiert auf einem Algorithmus, der
  - Objekte vom Typ `Block` (und analog anderer Datentypen) mit Knotenpositionen in der Ebenen versieht,
  - dort die jeweiligen Knoten zeichnet
  - die Knoten mit Kanten verbindet.

☞ fldit-www.cs.uni-dortmund.de/∼peter/Termpainter.hs

- Zunächst muss eine Funktion definiert werden, die Objekte des gegebenen Datentyps in Objekte der Instanz `Term String` des polymorphen Typs

```
data Term a = F a [Term a]
```

  von Bäumen mit beliebigem Knotenausgrad und Einträgen vom Typ `a` überführt.

Baumdarstellung

- Dann kann z.B. `prog` mit der Funktion `drawTerm` aus Termpainter.hs und gewünschten horizontalen bzw. vertikalen Knotenabständen gezeichnet werden.

Natürlich kann man Syntaxbäume auch ohne Grafikerweiterung darstellen. `prog` könnte in einer solchen Darstellung folgendermaßen ausehen:

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

Haskell-Beispiele

Die folgenden Haskell-Funktionen übersetzen jedes Objekt vom Typ `Block`, `Command`, `IntE` bzw. `BoolE` in die eben beschriebene String-Darstellung.

- 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.

Haskell-Beispiele

```
showBlock :: Bool -> Int -> Block -> String
showBlock firstLine n b = if firstLine then f b else blanks n++f b
  where f []     = "[]"
        f [c]    = '[':showCom True (n+1) c++"]"
        f (c:cs) = '[':g True c++',':str++g False (last cs)++"]"
                   where str = concat (map ((++",") . g False) (init cs))
                         g b = showCom b (n+1)

showCom :: Bool -> Int -> Command -> String
showCom firstLine n c = if firstLine then f c else blanks n++f c
  where f Skip             = "Skip"
        f (Assign x e)     = "Assign "++show x++
                             ' ':showIntE True (n+10+length x) e
        f (Cond be cs cs') = "Cond "++showBoolE True (n+5) be++g cs++g cs'
                             where g = showBlock False (n+5)
        f (Loop be cs)    = "Loop "++showBoolE True (n+5) be++
                                     showBlock False (n+5) cs
```

Haskell-Beispiele

```
showIntE :: Bool -> Int -> IntE -> String
showIntE firstLine n e = if firstLine then f e else blanks n++f e
  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 = showIntE b (n+5)
        f (Sum (e:es))  = "(Sum["++g True e++',':str++g False (last es)++"])"
                          where str = concat (map ((++",") . g False)
                                                  (init es))
                                g b = showIntE b (n+5)
        f (Prod (e:es)) = "(Prod["++g True e++',':str++g False (last es)++"])"
                           where str = concat (map ((++",") . g False)
                                                   (init es))
                                 g b = showIntE b (n+6)
```

Haskell-Beispiele

```
showBoolE :: Bool -> Int -> BoolE -> String
showBoolE firstLine n be = if firstLine then f be else blanks n++f be
  where f (BoolE b)      = "(BoolE "++show b++")"
        f (Greater e e') = "(Greater "++g True e++g False e'++")"
                           where g b = showIntE b (n+9)
        f (Not be)       = "(Not "++showBoolE True (n+5) be++")"

blanks n = '\n':replicate n ' '
```

- 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.

Typklassen in Haskell

Eine **Typklasse** stellt Bedingungen an die Instanzen einer Typvariablen.

Die Bedingungen bestehen in der Existenz bestimmter Funktionen und bestimmten Beziehungen zwischen ihnen.

Zum Beispiel 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.

Typklassen in Haskell

Eine Instanz einer Typklasse besteht aus den Instanzen ihrer Typvariablen sowie Definitionen der von ihr geforderten Funktionen (**Constraints**).

Beispiel für eine Instanz einer Typklasse:

```
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.
Im folgenden Beispiel beschränkt das Constraint `Eq a =>` die Instanziierung auf Typen, für die `==` und `/=` definiert sind... ➠

Typklassen in Haskell

Das Constraint `Eq a =>` vor `Ord a` beschränkt Instanzen von a:

```
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 Komposition: `(f . g) x = f (g x)`)

Typklassen in Haskell

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. für die Funktion:

```
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)
```

Typklassen in Haskell

Ist eine solche Funktion Teil der Instanz einer Tyklassse, 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'
```

Typklassen in Haskell

Alle bisherigen Typen waren solcher *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 $* \longrightarrow *$ hat. Die folgende Typklasse schränkt Typen mit diesem Kind ein:

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

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 h right)
```

Typklassen in Haskell

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 $* \longrightarrow *$ bzw. $* \longrightarrow * \longrightarrow *$ sind `Monad` und `Arrow`:

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

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

Typklassen in Haskell

`const x` liefert die konstante Funktion, die jede Eingabe auf `x` abbildet. Die einfachsten Datentypen zur Instanziierung von `m` bzw. `a` lauten:

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

Damit lassen sich `Monad` und `Arrow` 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
```

Typklassen in Haskell

Grob gesagt, dienen Monaden der Kapselung von Ausgaben, während Arrows Übergänge zwischen Ein- und Ausgaben kapseln. Arrows verallgemeinern Monaden. Zumindest bilden für jede Monade `m` die Funktionen `a -> m b` eine Instanz der Arrowklasse:

```
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))
```

Literatur

Weitere Haskell-Konstrukte werden on-the-fly eingeführt, wenn sie benötigt werden. Die folgenden Lehrbücher eignen sich zur umfassenden Einarbeitung:

- P. Hudak, J. Peterson, J.H. Fasel, *A Gentle Introduction to Haskell 98*, Report, 1999, siehe www.haskell.org
- M.M.T. Chakravarty, G.C. Keller, *Einführung in die Programmierung mit Haskell*, Pearson Studium 2004
- G. Hutton, *Programming in Haskell*, Cambridge University Press 2007 *Meine Wahl als Lehrbuch für den Haskell-Kurs in Bachelor-Studiengang Informatik.*
- F. Rabhi, G. Lapalme, *Algorithms: A Functional Programming Approach, Addison-Wesley 1999*; in der Lehrbuchsammlung unter L Sr 482/2
- S. Thompson, *Haskell: The Craft of Functional Programming*, Addison-Wesley 1999
- P. Hudak, *The Haskell School of Expression: Learning Functional Programming through Multimedia*, Cambridge University Press 2000
- R. Bird, *Introduction to Functional Programming using Haskell*, 2nd Edition, Prentice Hall 1998

Testumgebung

Legt drei Dateien für ein Programm und seine Ein/Ausgabe an:

- eine für die selbstdefinierten Typen und Funktionen (`prog.hs`),
- eine für Eingaben
- und eine für Ausgaben.

Ladet die erste beim Aufruf des Haskell-Interpreters: `ghci prog` oder `hugs prog`.

Testumgebung

Angenommen, `prog.hs` enthält die Fuktion `f :: a -> b`, die getestet werden soll. Dann fügt die Funktion

```
test_f :: String -> String -> IO()
test_f infile outfile = do readFile infile;
                           writeFile outfile . unparse . f . parse
```

sowie zwei Funktionen

- `parse :: String -> a`
- `unparse :: b -> String`

zu `prog.hs` hinzu.

Testumgebung

`test_f`

- liest den Inhalt von `infile`,
- übersetzt ihn mit `parse` in ein Objekt des Argumenttyps `a` von `f`,
- wendet `f` darauf an,
- übersetzt das Ergebnis mit `unparse` in einen String
- und schreibt diesen in die Datei `outfile`.

`IO` ist eine (Standard-)Instanz der Typklasse `Monad` (s.o.).
Sie nimmt den vom ersten Argument berechneten Wert und übergibt ihn an die Funktion im zweiten Argument.
Hier ist dies eine Komposition der vier Funktionen `writeFile outfile`, `unparse`, `f`, `parse`.

Testumgebung

Der Punkt ist die Haskell-Notation für die in der Mathematik üblicherweise als Kreis ($\circ$) dargestellte Funktionskomposition.

Gibt es für den Eingabetyp `a` von `f` einen Standardparser, dann entfällt die Definition von `parse` und man schreibt `read` anstelle von `parse`.
Gibt es für den Ausgabetyp `b` von `f` einen Standardunparser, dann entfällt die Definition von `unparse` und man schreibt `show` anstelle von `unparse`.

Baumzeichner

Zum Zeichnen von Syntaxbäumen mit textuellen einzeiligen Knotenmarkierungen könnt ihr meinen Modul `Termpainter.hs` (siehe fldit-www.cs.uni-dortmund.de/ peter/Termpainter.hs) verwenden: `import Termpainter`.

Er enthält eine Funktion

```
drawTerm :: String -> Int -> Int -> Term String -> IO ()
```

mit der ein Syntaxbaum vom Typ `Term String` graphisch dargestellt wird.

Der Stringparameter von `drawTerm` erscheint als Titel des Fensters, in das der Baum gezeichnet wird. Die beiden ganzzahligen Parameter sind der horizontale bzw. vertikale Abstand zweier benachbarter Konoten.

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.

Gängige Begriffe

Lexem Zeichenfolge, die ein Symbol repräsentiert,

Token Symbol, Terminalsymbol der Grammatik der Quellsprache (s. Def. 3.1.1)

Pattern Muster für Symbole, regulärer Ausdruck (s. Def. 2.1.1)

Regulären Ausdrücke

Definition 2.1.1 Sei A ein endliches Alphabet (Zeichenmenge). Die Menge $\boldsymbol{Reg(A)}$ der **regulären Ausdrücke über A** ist dann wie folgt induktiv definiert:

- $\varepsilon \in \boldsymbol{Reg(A)}$ (das „leere Wort“)
- $\emptyset \in \boldsymbol{Reg(A)}$
- $\boldsymbol{A \subseteq Reg(A)}$
- $\boldsymbol{R, R' \in Reg(A) \Rightarrow RR' \in Reg(A)}$ „Konkatenation“
- $\boldsymbol{R, R' \in Reg(A) \Rightarrow R + R' \in Reg(A)}$ „Vereinigung“
- $\boldsymbol{R \in Reg(A) \Rightarrow R^+, R^*, R? \in Reg(A)}$ „Abschlüsse“

Beispiel: „$\boldsymbol{01^* + 10^*}$“ ist ein regulärer Ausdruck auf der Zeichenmenge $A = \{0, 1\}$.

Regulären Ausdrücke

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

```
data RegExp a = Const a | Eps | Empty | Seq (RegExp a) (RegExp a) |
                Par (RegExp a) (RegExp a) | Plus (RegExp a)
```

Da sich reflexiver wie reflexiv-transitiver Abschluss aus anderen Operatoren ableiten lassen, bietet es sich an, $*$ und ? nicht als Konstruktoren, sondern als Funktionen vom Typ `RegExp a -> RegExp a` zu definieren:

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

Regulären Ausdrücke

Die Funktion $L : Reg(A) \to \mathfrak{P}(A^*)$ ordnet jedem regulären Ausdruck über A *eine Menge von Wörtern* über A zu.

L ist induktiv über dem Aufbau von $Reg(A)$ definiert :

- $L(\varepsilon) = \{\varepsilon\}$
- $L(\emptyset) = \emptyset$
- $L(a) = \{a\}$ für alle $a \in A$
- $L(RR') = \{vw \mid v \in L(R), w \in L(R')\}$
- $L(R + R') = L(R) \cup L(R')$
- $L(R^+) = \bigcup_{n \geq 1} \{w_1 ... w_n \mid w_i \in L(R), 1 \leq i \leq n\}$
- $L(R^*) = L(R^+) \cup \{\varepsilon\}$
- $L(R?) = L(R) \cup \{\varepsilon\}$

Reguläre Sprache über A

Jede Wortmenge M mit

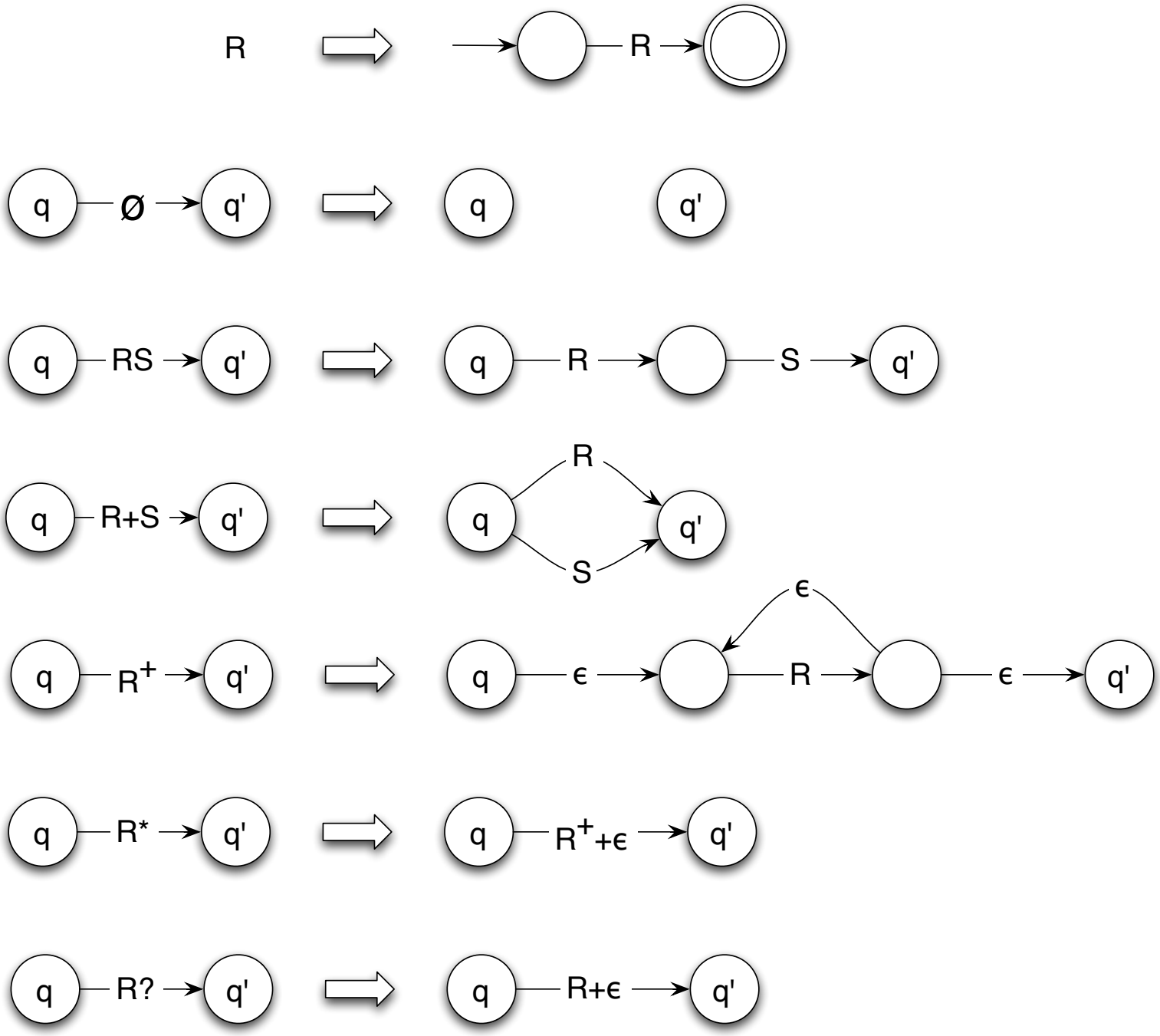
$$L(R) = M \text{ für ein } R \in Reg(A)$$

heißt **reguläre Sprache über A.**

Beispiele für *nichtreguläre* **Sprachen** sind

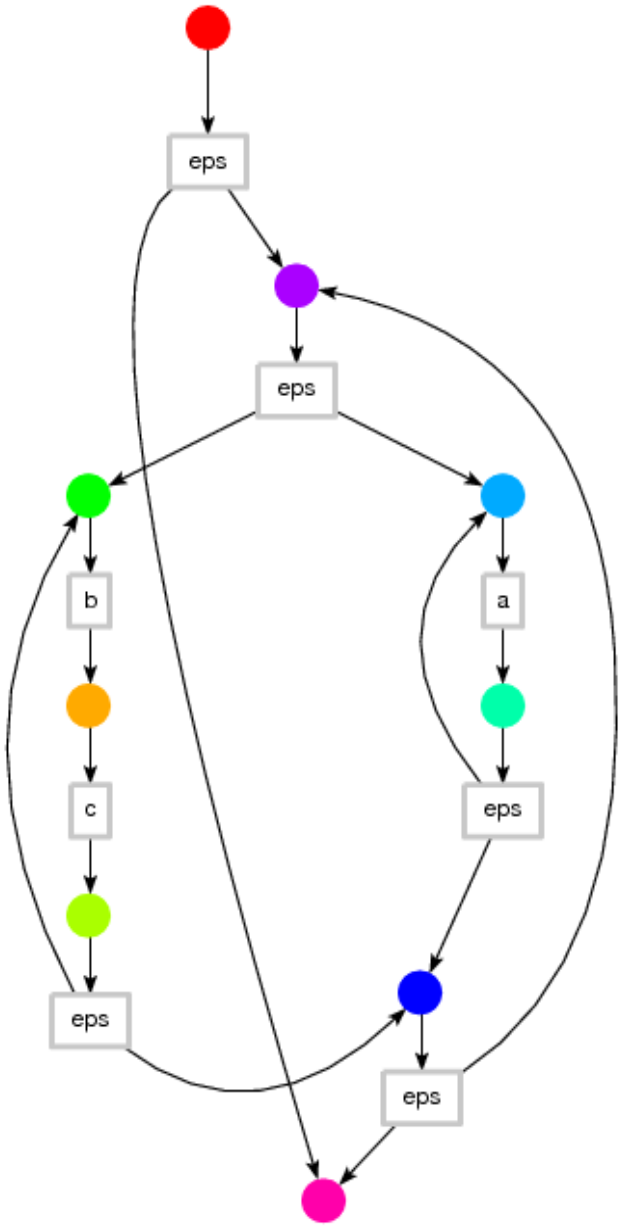
- *geschachtelte Ausdrücke:*
$$\{u^n w v^n \mid u, v \in A^*, n \geq 0\},$$
- *String-Wiederholungen:*
$$\{uwu \mid u \in A^*\}$$
- *Vergleich von Präfix und Suffix:*
$$\{uwv \mid u, v \in A^*, 3 * length(u) = 5 * length(v)\}$$

☞ Hier ist immer ein Keller zur Speicherung von Teilwörtern erforderlich!


R
R
q
Ø
q'
q
q'
q
RS
q'
q
R
S
q'
q
R+S
q'
q
R
S
q'
q
R+
q'
q
ϵ
R
ϵ
q'
ϵ
q
R*
q'
q
R+ϵ
q'
q
R?
q'
q
R+ϵ
q'

Tabellen und Graphdarstellung eines Automaten

	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	



Endliche 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)

  oder

  einer *Übergangsrelation* $\delta : Q \times (X \cup \{\epsilon\}) \to \mathfrak{P}(Q)$
  (nichtdeterministischer Automat)
- einer *Ausgabefunktion* $\beta : Q \to Y$, bzw. $\beta : Q \to \mathfrak{P}(Y)$
- einem *Anfangszustand* $q_0 \in Q$.

Endliche Automaten

Ein endlicher Automat heißt **erkennender Automat**, wenn die Ausgabemenge Y zweielementig ist, also z.B.:

$$Y = \{0, 1\}.$$

Wenn A ein erkennender Automat mit Ausgabefunktion β und $Y = \{0, 1\}$ ist, dann nennt man jeden Zustand

$$\boldsymbol{q \in Q} \text{ mit } \boldsymbol{\beta(q) = 1}$$

einen **Endzustand** von A.

☞ Mit $\boldsymbol{E} \subseteq Q$ bezeichnen wir im Weiteren die *Menge der Endzustände* eines erkennenden Automaten.

Beschreibung deterministischer Automaten

Um die Eigenschaften endlicher Automaten einfacher beschreiben zu können, definieren wir:

- Die **Fortsetzung** δ^* der Übergangsfunktion auf Wörter $w \in X^*$ für *deterministische* Automaten ist eine induktiv definierte Funktion von $Q \times X^*$ in die Zustandsmenge Q:

$$\begin{aligned} \delta^*(q, \varepsilon) &= q \\ \delta^*(q, xw) &= \delta^*(\delta(q, x), w) \end{aligned}$$

- Die **Erreichbarkeitsfunktion** $r : X^* \to Q$ eines *deterministischen* Automaten wird dann definiert durch:

$$r(w) = \delta^*(q_0, w)$$

wobei $w \in X^*$ und $q_0 \in Q$ der Anfangszustand von A ist.

Beschreibung nichtdeterministischer Automaten

- Die **Fortsetzung** δ^* der Übergangsrelation auf Wörter $w \in X^*$ für *nichtdeterministische* Automaten ist eine induktiv definierte Relation $\delta^* : Q \times X \to \mathfrak{P}(Q)$:

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

- Die **Erreichbarkeitsrelation** $r : X^* \to \mathfrak{P}(Q)$ eines *nichtdeterministischen* Automaten wird definiert durch: $r(w) = \delta^*(q_0, w)$

Darstellung in Haskell

In Haskell lässt sich δ^* mit *foldl* aus der Übergangsfunktion δ bilden:

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

bzw.

```
iter :: Eq state => (state -> Ext input -> [state])
                                  -> state -> [input] -> [state]
iter delta q = foldl f (epsHull delta [q])
               where f qs x = epsHull delta rs
                              where rs = joinMap (flip delta (Def x)) qs
```

```
epsHull :: Eq state => (state -> Ext input -> [state])
                                          -> [state] -> [state]
epsHull delta qs = f qs qs
                   where f qs visited = if null new then visited
                                        else f new (visited++new)
                                 where rs  = joinMap (flip delta Epsilon) qs
                                       new = rs 'minus' visited

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

minus,join :: Eq a => [a] -> [a] -> [a]
xs 'minus' ys = [x | x <- xs, x 'notElem' ys]
xs 'join'  ys = xs++(ys 'minus' xs)
```

Der Datentyp `Ext a = Epsilon | Def a` dient der Repräsentation der um das leere Wort erweiterten Eingabemenge $X \cup \{\varepsilon\}$: Der Konstruktor *Epsilon* implementiert ε, den Elementen von X wird der Konstruktor *Def* vorangestellt.

Endliche Automaten und die erkannte Sprachen

- Die von einem *deterministischen* erkennenden Automaten A **erkannte Sprache** $L(A)$ ist definiert als

$$L(A) =_{def} \{w \in X^* | r(w) \in E\}$$

- Die von einem *nichtdeterministischen* erkennenden Automaten A **erkannte Sprache** $L(A)$ ist definiert als

$$L(A) =_{def} \{w \in X^* | \exists q \in E : (w, q) \in r\}$$

☞ Äquivalenz endlicher Automaten: Zwei erkennende Automaten sind äquivalent, wenn die von ihnen erkannten Sprachen übereinstimmen!

Reguläre Grammatiken

Definition 2.1.3 Eine kontextfreie Grammatik (N, T, P, S) heißt **regulär**, wenn für alle Produktionen

$$(A \to w) \in P$$

gilt, dass

$$w \in T^*N \quad \textit{oder} \quad w \in T^*.$$

Satz 2.1.4 Sei $L \subseteq T^*$. Dann sind folgende Aussagen äquivalent:

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

Übersetzung regulärer Ausdrücke in NEA

Der *Übergang von regulären Ausdrücken zu nichtdeterministischen Automaten* ist durch die o.g. Ersetzungsregeln definiert.
Noch einfacher ist die *Übersetzung regulärer Grammatiken in nichtdeterministische Automaten*. Dabei wird jede Produktion der regulären Grammatik zu einer Folge von *Zustandstransitionen*:

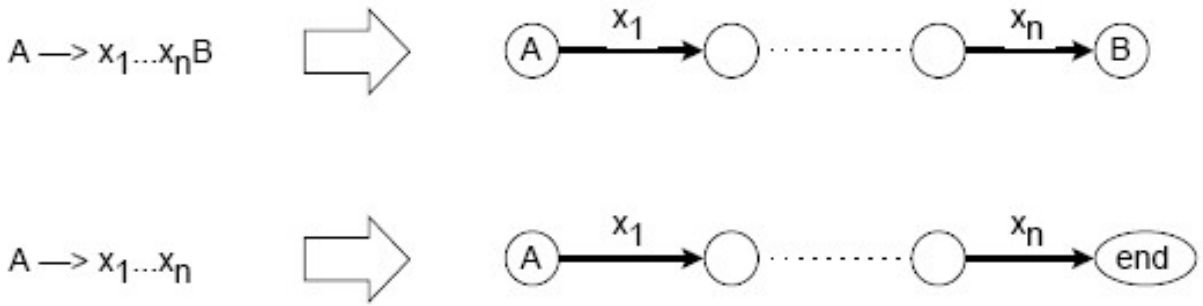


Abbildung: Übersetzung regulärer Grammatiken in nichtdeterministische Automaten

Kommando-Scanner

- Es sollen die in einer Zeichenfolge auftretenden Symbole der imperativen Sprache von Beispiel 1.2.2 erkannt werden.
- ☞ Ein Automat mit drei Zuständen: Schlüsselwörter, Variablen und Zahlen
- Weitere Zustände sind notwendig, wenn der ein Symbol repräsentierende String Teilstrings hat, die andere Symbole repräsentieren können.

Hier zunächst der Datentyp für die Symbole:

```
data Symbol = Lpar | Rpar | Lcur | Rcur | Semi | Upd | GR | Neg |
              Plus | Minus | Times | Num Int | Ide String |
              True_ | False_ | If | Else | While
```

Kommando-Scanner

Ein Scanner bildet `String` auf `[Symbol]` ab. Jeder Zustand entspricht einer Scanfunktion:

```
scan :: String -> [Symbol]
scan ('(':str)            = Lpar:scan str
scan (')':str)            = Rpar:scan str
scan ('{':str)            = Lcur:scan str
scan ('}':str)            = Rcur:scan str
scan (';':str)            = Semi:scan str
scan ('=':str)            = Upd:scan str
scan ('!':str)            = Neg:scan str
scan ('>':str)            = GR:scan str
scan ('+':str)            = Plus:scan str
scan ('-':str)            = Minus:scan str
scan ('*':str)            = Times:scan str
scan (x:str) | isDigit x = scanNum [x] str
             | isDelim x = scan str
             | True      = scanIde [x] str
scan _                    = []
```

Kommando-Scanner

```
scanNum :: String -> String -> [Symbol]
scanNum num str@(x:rest) | isDigit x = scanNum (num++[x]) rest
                         | True      = Num (read num):scan str
scanNum num _                        = [Num (read num)]

scanIde :: String -> String -> [Symbol]
scanIde ide str@(x:rest) | isSpecial x = checkWord ide:scan str
                         | True        = scanIde (ide++[x]) rest
scanIde ide _                          = [checkWord ide]

checkWord :: String -> Symbol
checkWord "true"  = True_
checkWord "false" = False_
checkWord "if"    = If
checkWord "else"  = Else
checkWord "while" = While
checkWord ide     = Ide ide
```

Kommando-Scanner

```
isDelim :: Char -> Bool
isDelim = (‘elem‘ " \n\t")

isSpecial :: Char -> Bool
isSpecial = (‘elem‘ "(){},;=+-* \n\t")

isDigit :: Char -> Bool
isDigit = (‘elem‘ [’0’..’9’])
```

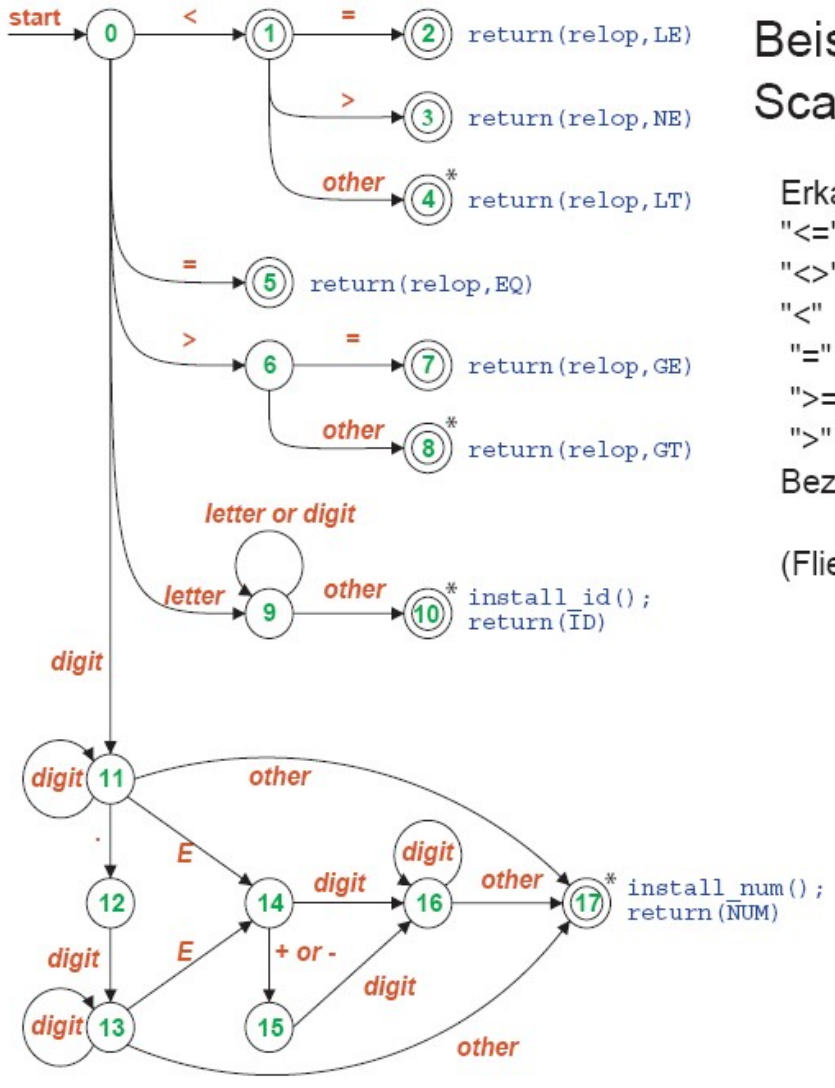
- Allgemein gesprochen, führt ein Scanner das Quellprogramm Zeichen für Zeichen einem erkennenden Automaten zu.
- Dieser Automat liest jedes Zeichen, führt entsprechende Zustandsübergänge aus und markiert dessen Endzustände mit jeweils einem Ausgabesymbol.

Kommando-Scanner

- 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. Der Automat fährt mit dem Lesevorgang fort.
- Wird wieder ein Endzustand erreicht, dann werden *final* und *start_position* entsprechend umgesetzt.
- 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* gelesen und wie oben fortgefahren.

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

Scanner-Automat



Beispiel für einen Scanner-Automat

Erkannt werden:
"<=" (LE)
"<>" (NE),
"<" (LT)
"=" (EQ),
">=" (GE),
">" (GT)
Bezeichner aus Buchstaben und Ziffern (ID)
(Fließkomma-)Zahlen (NUM)

Implementierung in Haskell

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

```
type DetAuto input output state    = (state -> input -> state,
                                      state -> output, state)
type NonDetAuto input output state = (state -> Ext 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 rs
                                  where rs = joinMap (flip delta (Def x)) qs
```

Implementierung in Haskell

- 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.
- Da wir es hier mit *endlichen* Automaten zu tun haben, haben Übergangs- und Ausgabefunktion einen endlichen Definitionsbereich und können deshalb in Tabellen oder Adjanzenzlisten überführt werden.
- Für diese Tabellen gibt es dann wieder standardmäßige Stringdarstellungen.

- Berücksichtigt man noch die Einschränkung auf *erkennende* Automaten, dann bietet sich als Datenstruktur für (nichtdeterministische) Automaten der folgende Typ an:

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

Implementierung in Haskell

Die Ausgabe eines Automaten vom Typ `IntAuto` könnte beispielsweise mit folgender Funktion `writeAuto` erfolgen:

```
writeAuto :: Show a => IntAuto a -> IO ()
writeAuto ((delta,beta,q0),qs,as) = writeFile "autos" (initial++trips++finals)
      where initial = "\ninitial state: "++show q0
            trips = "\ntransition function:"++
                    fun2ToTrips delta qs (Epsilon:map Def as)
            finals = "\nfinal states: "++
                     show [i | i <- qs, True 'elem' beta i]

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

instance Show a => Show (Ext a) where show Epsilon = "eps"
                                      show (Def a) = show a
```

Syntaxanalyse

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*

Kontextfreie Grammatiken

Definition 3.1.1 Eine **kontextfreie** oder **CF-Grammatik** ist ein Quadrupel

$$\boldsymbol{G = (N, T, P, S)}$$

bestehend aus:

- einer endlichen Menge $\boldsymbol{N}$ von *Nichtterminalen*,
- einer zu N disjunkten endlichen Menge $\boldsymbol{T}$ von *Terminalen*
- einer endlichen Menge $\boldsymbol{P}$ von *Produktionen* oder *(Ableitungs-) Regeln* der Form

  $$\boldsymbol{A \longrightarrow w}$$

  mit $A \in N$ und $w \in (N \cup T)^*$,
- einem *Startsymbol* $\boldsymbol{S} \in N$.

Kontextfreie Grammatiken

Anstelle von n **Produktionen mit derselben linken Seite**:

$$A \to w_1,\ A \to w_2,\ \ldots\ ,\ A \to w_n$$

schreiben wir stattdessen:

$$A \to w_1 \mid w_2 \mid \ldots \mid w_n$$

Für einen **Ableitungsschritt** schreiben wir:

$$v \to_G w \quad \text{oder} \quad v \to w$$

wenn das Wort w in *einem* Schritt aus v abzuleiten ist.

Für eine **Folge von Ableitungsschritten** schreiben wir:

$$v \overset{*}{\to}_G w \quad \text{bzw.} \quad v \overset{*}{\to} w$$

wenn sich w aus v in *endlich vielen Schritten* ableiten läßt.

Die erzeugte Sprache $L(G)$

Für eine gegebene Grammatik $G = (N, T, P, S)$ ist die Menge

$$\boldsymbol{L(G)} \;=\; \left\{ \boldsymbol{w} \in \boldsymbol{T^*} \mid \boldsymbol{S} \stackrel{*}{\rightarrow}_{\boldsymbol{G}} \boldsymbol{w} \right\}$$

die **von G erzeugte Sprache**.

Die Grammatik einer Programmiersprache PS nennt man auch **konkrete Syntax von PS**.

Beispiel 3.1.2

Eine *funktionale Sprache* mit *verketteten (geschachtelten) Listen* als grundlegender Datenstruktur – wie in Lisp – und *nicht-applikativen Funktionsausdrücken* wird mit der folgenden Grammatik $\boldsymbol{FPF} = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S})$ definiert:

$$
\begin{aligned}
\boldsymbol{N} &= \{const, fun\} \\
\boldsymbol{T} &= \mathbb{Z} \cup \{[], <, >, \texttt{id}, \texttt{head}, \texttt{tail}, \texttt{num}, +, =, \equiv, \circ, [,], \\
&\qquad \texttt{if}, \texttt{then}, \texttt{else}, \texttt{,}, \alpha, /\}
\end{aligned}
$$

Beispiel 3.1.2

$$
\begin{aligned}
\mathbf{P} \quad = \quad & \{const \to i \text{ für alle } i \in \mathbb{Z}, \\
& const \to [], \\
& const \to <const, \ldots, const>, \\
& fun \to \texttt{id} \mid \texttt{head} \mid \texttt{tail} \mid \texttt{num} \mid \texttt{+} \mid \texttt{=} \mid \equiv const \\
& fun \to fun \circ fun \mid \texttt{[}fun, \ldots, fun\texttt{]} \\
& fun \to \texttt{if } fun \texttt{ then } fun \texttt{ else } fun \mid \alpha\ fun \mid / fun \ \} \\
\mathbf{S} \quad = \quad & fun
\end{aligned}
$$

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

Beispiel 3.1.3

Eine *imperative Sprache* mit den üblichen nicht-rekursiven Kontrollstrukturen ist durch die folgende Grammatik $\boldsymbol{IPF} = (\boldsymbol{N}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S})$ gegeben: Sei V die Menge der Denotationen von Zustandsvariablen:

$$
\begin{aligned}
\textbf{N} &= \{const, var, exp, boolexp, com\} \\
\textbf{T} &= \mathbb{Z} \cup V \cup \{+, =, \texttt{and}, :, ;, \texttt{if}, \texttt{then}, \texttt{else}, \texttt{while}, \texttt{do}, \\
&\qquad \texttt{repeat}, \texttt{until}, \texttt{skip}, \texttt{true}, \texttt{false}, > \} \\
\textbf{P} &= \{ \\
&\quad const \to i \text{ für alle } i \in \mathbb{Z} \\
&\quad var \to x \text{ für alle } x \in V
\end{aligned}
$$

Beispiel 3.1.3

$$
\begin{aligned}
& \ldots \\
& exp \rightarrow\ const \mid var \mid exp + exp \\
& boolexp \rightarrow\ \texttt{true} \mid \texttt{false} \\
& boolexp \rightarrow\ exp\ \texttt{=}\ exp \mid exp\ \texttt{<}\ exp \\
& boolexp \rightarrow\ boolexp\ \texttt{and}\ boolexp \mid \neg\ boolexp \\
& com \rightarrow\ var\ \texttt{:=}\ exp \mid com\texttt{;}com \\
& com \rightarrow\ \texttt{if}\ boolexp\ \texttt{then}\ com\ \texttt{else}\ com \\
& com \rightarrow\ \texttt{while}\ boolexp\ \texttt{do}\ com \\
& com \rightarrow\ \texttt{repeat}\ com\ \texttt{until}\ boolexp \\
& com \rightarrow\ \texttt{skip}\ \} \\
\mathbf{S}\ =\ & com
\end{aligned}
$$

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

Ableitungsbäume

Definition 3.1.4 Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik. Die Menge der **Ableitungsbäume** (*parse trees*) von G ist induktiv definiert:

- $\boldsymbol{x \in \{\varepsilon\} \cup T}$ ist ein Ableitungsbaum.
- Ist $\boldsymbol{A \to x_1 \dots x_n \in P}$ und ist für alle $1 \leq i \leq n$ B_i ein Ableitungsbaum von G mit Wurzel x_i, dann ist auch $\boldsymbol{A(B_1, \dots, B_n)}$ ein Ableitungsbaum.

Links- und Rechtsableitungen

Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik.
Ein Ableitungsschritt $uAw \to uvw$ mit $u, w \in (N \cup T)^*$ und $A \to v \in P$ heißt **direkte Links-** bzw. **direkte Rechtsableitung**, wenn $u \in T^*$ bzw. $w \in T^*$ gilt.

Eine Folge direkter Links- bzw. Rechtsableitungen:

$$\boldsymbol{w_1 \longrightarrow w_2,\ w_2 \longrightarrow w_3,\ \ldots}$$

heißt **Links-** bzw. **Rechtsableitung** (*left parse* bzw. *right parse*).

Eindeutigkeit von CF-Grammatiken und Sprachen

Die Existenz eindeutiger Ableitungsbäume charakterisiert eine Grammatik G und die erzeugte Sprache:

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

Außerdem gilt:

Jeder Links- und jeder Rechtsableitung $A \overset{*}{\rightarrow} w$ entspricht *eineindeutig* ein Ableitungsbaum mit Wurzel A und der Battfolge w.

Wichtige Ergebnisse

Bekannte Resultate aus den Grundlagen der theoretischen Informatik:

1. Die **Eindeutigkeit** einer **kontextfreien Grammatik** ist *nicht entscheidbar* $\rightsquigarrow$ Wird mittels der Unentscheidbarkeit des *Postschen Korrespondenzproblems* gezeigt.

2. Die **Äquivalenz zweier kontextfreier Grammatiken ist unentscheidbar**, im Gegensatz zur Äquivalenz regulärer Grammatiken.

3. **Deterministisch kontextfreie Sprachen**, das sind Sprachen, die von *deterministischen Kellerautomaten* erkannt werden, sind *eindeutig*.

Wichtige Ergebnisse

4. Sprachbeispiele
   - Eine **mehrdeutige kontextfreie Sprache**:
     $$L = \{a^i b^j c^k \mid i, j, k \geq 1, i = j \vee j = k\}$$
     Denn: Zu jeder Grammatik G mit $L = L(G)$ existieren Wörter in L mit zwei verschiedenen Ableitungsbäumen.
   - Eine **eindeutige kontextfreie Sprache**, die aber *nicht regulär* ist:
     $$L = \{a^i b^i \mid i \geq 1\}$$
     Denn: Ein Automat, der L erkennt, müßte unendlich viele Zustände haben
   - Eine **nicht kontextfreie Sprache**:
     $$L = \{a^i b^i c^i \mid i \geq 1\}$$

Wichtige Ergebnisse

5. Eine kontextfreie Gramatik G ist **selbsteinbettend**, wenn es eine Ableitung

$$\boldsymbol{A \xrightarrow{+} \alpha A \beta}$$

mit $\alpha, \beta \in (N \cup T)^+$ gibt.

Charakterisierung von CF-Grammatiken:

- Nicht-selbsteinbettende kontextfreie Grammatiken sind regulär.
- Selbsteinbettung entspricht nicht-iterativer Rekursion.
- Eine **kontextfreie Sprache ist selbsteinbettend**, wenn *alle* sie erzeugenden Grammatiken selbsteinbettend sind.

Konkrete und abstrakte Syntax

Eine kontextfreie Grammatik G einer Programmiersprache PS heißt auch **konkrete Syntax** von PS. Die **abstrakte Syntax** von PS erhält man aus G durch

- Entfernung der Terminalsymbole *und*
- Ersetzung der Produktionen durch Funktionssymbole.

☞ Den Funktionssymbolen entsprechen die *Konstruktoren* eines Haskell-Datentyps.

Resultat der Transformation in abstrakte Syntax:

- **Abstrakte Programme** sind aus Konstruktoren zusammengesetzte **Terme** (=*funktionale Ausdrücke*).
- Jedem Nichtterminal von G entspricht ein *unstrukturierter Datentyp*, der im Allgemeinen **Sorte** genannt wird.

Sortierte Mengen und Funktionen

Definition 3.2.1 Sei $\mathcal{S}$ eine Menge.

- Eine **Menge A** heißt **$\mathcal{S}$-sortiert**, wenn es für jedes $s \in \mathcal{S}$ eine Menge A_s gibt mit

$$\biguplus \{A_s \mid s \in \mathcal{S}\} = A.$$

- Eine **Funktion $f : A \longrightarrow B$** heißt **$\mathcal{S}$-sortiert**, wenn gilt:
  - A und B sind $\mathcal{S}$-sortierte Mengen
  - zu jedem $s \in \mathcal{S}$ gibt es eine Funktion

$$f_s : A_s \to B_s \text{ mit } f_s(a) = f(a)$$

  für alle $a \in A_s$.

Notation: Für $s_1, \ldots, s_n \in \mathcal{S}$ und $w = s_1 \ldots s_n$ schreiben wir A_w anstelle von $A_{s_1} \times \ldots \times A_{s_n}$.

Sortierte Mengen und Funktionen

Ist $w = s_1 \dots s_n$ mit $s_i \in \mathcal{S}$, dann wird die Funktion $f_w : A_w \to B_w$ definiert durch:

$$\boldsymbol{f_w(a) = (f_{s_1}(a_1), \dots, f_{s_n}(a_n))}$$

für alle $a = (a_1, \dots, a_n) \in A_w$.

- Die Sortierung von Menge könnte man auch *Typisierung* nennen.
- Eine Logik heißt *mehrsortig*, wenn die in ihr verwendeten Terme zu einer sortierten Menge gehören, die aus einer *mehrsortigen Signatur* gebildet werden:

Sortierte Mengen und Funktionen

Definition 3.2.2 Eine **Signatur** $\Sigma = (\mathcal{S}, \mathcal{F})$ besteht aus

- einer Menge $\mathcal{S}$ von *Sorten*
- einer Menge $\mathcal{F}$ von $\mathcal{S}^+$-sortierten *Funktionssymbolen*

Die Elemente von $\mathcal{F}_s$ mit $s \in \mathcal{S}$ heißen *Konstanten*.

Notation: Anstelle von $f \in \mathcal{F}_s$ bzw. $g \in \mathcal{F}_{s_1 \ldots s_n s}$
schreiben wir $f :\longrightarrow s \in \Sigma$ bzw. $g : s_1 \ldots s_n \longrightarrow s \in \Sigma$

Σ ist pure Syntax! Semantik erhält man durch **Interpretation** der Sorten und Funktionssymbole als Mengen bzw. Funktionen auf diesen Mengen.

Abstrakte Syntax einer CF-Grammatik

Definition 3.2.3 Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik.

Die folgende Signatur heißt **abstrakte Syntax** von G:

- N ist die Sortenmenge.
- Für jede Produktion $p \in P$:

  $$A \longrightarrow w_0 A_1 w_q \ldots A_n w_n \quad \text{mit } w_i \in T^* \text{ und } A_i \in N$$

  enthält Σ ein Funktionssymbol

  $$f_p : A_1 \times \ldots \times A_n \longrightarrow A,$$

  das **Konstruktor** genannt wird.
- Σ enthält keine weiteren Symbole.

Produktionen $A \to w$ von G mit $w \in T^*$ werden zu Konstanten (=nullstelligen Funktionssymbolen) von $\Sigma(G)$.

Beispiel 3.2.4: Abstrakte Syntax von FPF

Die abstrakte Syntax $\Sigma_{FPF} = (S, F)$ von FPF (Beispiel 3.1.2) lautet:

Sortenmenge $= \{const, fun\}$

Funktionssymbole $= \{$

$\mathbf{i} : \rightarrow const$ für alle $i \in \mathbb{Z}$,

$[\![\,]\!] : \rightarrow const$,

$\mathbf{list} : const \times \ldots \times const \rightarrow const$,

$\mathbf{id}, \mathbf{head}, \mathbf{tail}, \mathbf{num}, \mathbf{add}, \mathbf{eq} : \rightarrow fun$,

$\mathbf{C} : const \rightarrow fun$,

$\mathbf{comp} : fun \times fun \rightarrow fun$,

$\mathbf{mul} : fun \times \ldots \times fun \rightarrow fun$,

$\mathbf{cond} : fun \times fun \times fun \rightarrow fun$,

$\mathbf{map}, \mathbf{fold} : fun \rightarrow fun$ $\}$

Beispiel 3.2.5: Abstrakte Syntax von IPF

Die abstrakte Syntax $\Sigma_{IPF} = (S, F)$ von IPF (Beispiel 3.1.3) lautet:

Sortenmenge $= \{const, var, exp, boolexp, com\}$

Funktionssymbole $= \{$

$\mathbf{i} : \rightarrow const$ für alle $i \in \mathbb{Z}$,

$\mathbf{x} : \rightarrow var$ für alle $x \in V$,

$\mathbf{C} : const \rightarrow exp$,

$\mathbf{V} : var \rightarrow exp$,

$\mathbf{add} : exp \times exp \rightarrow exp$,

$\mathbf{True}, \mathbf{False} : \rightarrow boolexp$,

...

Beispiel 3.2.5: Abstrakte Syntax von IPF

Fortsetzung von Beispiel 3.2.5:

$$
\begin{aligned}
&\ldots \\
&\mathbf{eq}, \mathbf{greater} : exp \times exp \to boolexp, \\
&\mathbf{and} : boolexp \times boolexp \to boolexp, \\
&\mathbf{not} : boolexp \to boolexp, \\
&\mathbf{assign} : var \times exp \to com, \\
&\mathbf{seq} : com \times com \to com, \\
&\mathbf{cond} : boolexp \times com \times com \to com, \\
&\mathbf{loop} : boolexp \times com \to com, \\
&\mathbf{repeat} : com \times boolexp \to com, \\
&\mathbf{skip} :\to com\}
\end{aligned}
$$

Grundterme und Syntaxbäume

Definition 3.2.6 Sei $\Sigma = (S, F)$ eine Signatur. Die S-sortierte Menge T_Σ der Σ-**Grundterme** ist *induktiv* definiert:

- Für alle $w \in S^*,\ s \in S,\ f : w \to s \in \Sigma$ und $t \in T_{\Sigma,w}$

Ist Σ die abstrakte Syntax einer CF-Grammatik $G = (N, T, P, S)$, dann nennen wir Σ-Grundterme auch **Syntaxbäume von** G und schreiben **B(G)** für $T_{\Sigma(G)}$.

Implementierung in Haskell

Die Haskell-Implementierung der abstrakten Syntax lautete wie folgt:

```
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
```

Der Ausgangspunkt des Parsers sei folgende kontextfreie Grammatik, die auch dem Kommando-Scanner zugrundelag:

```
Block   --> {Seq}
Seq     --> empty | Command Seq
Command --> ; | String = IntE ; | Block | if (BoolE) Block |
            if (BoolE) Block else Block | while (BoolE) Block
IntE    --> Int | String | (IntE) | IntE - IntE | Sum | Prod
Sum     --> IntE | IntE + Sum
Prod    --> IntE | IntE * Prod
BoolE   --> true | false | IntE > IntE | not BoolE
```

Implementierung in Haskell

- Jeder der vier Typen `IntE`, `BoolE`, `Block` und `Command` entspricht einer Sorte der abstrakten Syntax, die wiederum aus einem Nichtterminal der konkreten Syntax entstand,
- Wir verwenden für Typ, Sorte und Nichtterminal jeweils denselben Namen.
- Semantisch stehen alle drei für eine Datenmenge. Die Nichtterminale `Sum`, `Prod` und `Seq` entsprechen Listentypen.
- Die Syntaxanalyse scheitert, wenn die eingelesene Symbolfolge nicht zur Sprache der zugrundeliegenden Grammatik gehört. Der Parser soll dann eine möglichst präzise Information über die Ursache des Scheiterns liefern.

Implementierung in Haskell

Wir benötigen zunächst einen Datentyp, der (korrekte) Syntaxbäume *und* Fehlermeldungen umfasst:

```
data Result a  = Result a [Symbol] | Error String
```

- Der Kommando-Parser wird die Typvariable `a` durch `IntE`, `BoolE` und `Command` instanziieren.
- `Result a [Symbol]` beschreibt die Menge der Paare, bestehend aus einem Syntaxbaum der Sorte `a` und der **Resteingabe**
- Die gesamte Symbolfolge wurde vom Scanner aus einer Zeichenfolge erzeugt Scheitert der Parser, dann gibt er die jeweilige Fehlermeldung `msg` als Element `Error msg` aus.

Implementierung in Haskell

- Jedes Nichtterminal N benötigt eine eigene Parsefunktion `parseN` vom Typ `[Symbol] -> Result N`.
- Das reicht aber meistens nicht aus, wie man bei N = `IntE` sieht. `IntE` umfasst viele Fälle, von denen sich einige bei der Parsierung überlappen.
- Um immer nur die jeweils *längsten* Symbolfolgen zu erkennen und in Syntaxbäume umzuwandeln, benötigt `parseIntE` mehrere Hilfsparser. Hinzu kommen eigene Parser für die verwendeten Listentypen (hier: `Sum`, `Prod` und `Seq`).

Implementierung in Haskell

```
parseBlock :: [Symbol] -> Result [Command]
parseBlock (Lcur:syms) = case parseSeq syms of
                           Result cs (Rcur:syms) -> Result cs syms
                           _ -> Error "missing }"
parseBlock _           = Error "missing {"

parseSeq :: [Symbol] -> Result [Command]
parseSeq syms = case parseCom syms of
                  Result c syms -> case parseSeq syms of
                                     Result cs syms -> Result (c:cs) syms
                  _ -> Result [] syms
```

Implementierung in Haskell

```
parseCom :: [Symbol] -> Result Command
parseCom (Semi:syms)        = Result Skip syms
parseCom (Ide x:Upd:syms)   = case parseIntE syms of
                                Result e (Semi:syms) -> Result (Assign x e) syms
                                Error str -> Error str
                                _ -> Error "missing ;"
parseCom (Ide x:_)          = Error "missing ="
```

Implementierung in Haskell

```
parseCom (If:Lpar:syms)    = case parseBoolE syms of
                               Result be (Rpar:syms)
                                 -> case parseBlock syms of
                                        Result cs (Else:syms)
                                          -> case parseBlock syms of
                                                 Result cs' syms
                                                  -> Result (Cond be cs cs')
                                                            syms
                                                 Error str -> Error str
                                        Result cs syms
                                         -> Result (Cond be cs []) syms
                                        Error str -> Error str
                               Error str -> Error str
                               _ -> Error "missing )"
parseCom (If:_)            = Error "missing ("
```

Implementierung in Haskell

```
parseCom (While:Lpar:syms) = case parseBoolE syms of
                               Result be (Rpar:syms)
                                -> case parseBlock syms of
                                   Result cs syms -> Result (Loop be cs) syms
                                   Error str -> Error str
                             Error str -> Error str
                             _ -> Error "missing )"
parseCom (While:_)         = Error "missing ("
parseCom _                 = Error "no command"

parseInt :: (IntE -> [Symbol] -> Result Symbol IntE)
            -> [Symbol] -> Result Symbol IntE
parseInt f (Num i:syms) = f (IntE i) syms
parseInt f (Ide x:syms) = f (Var x) syms
parseInt f (Lpar:syms)  = case parseInt parseRest syms of
                               Result e (Rpar:syms) -> f e syms
                               Error str -> Error str
                               _ -> Error "missing )"
parseInt _ _            = Error "no integer expression"
```

Implementierung in Haskell

```
parseRest :: IntE -> [Symbol] -> Result IntE
parseRest e (Minus:syms) = parseSub e syms
parseRest e (Plus:syms)  = parseSum [e] syms
parseRest e (Times:syms) = parseProd [e] syms
parseRest e syms         = Result e syms

parseSub :: IntE -> [Symbol] -> Result IntE
parseSub e syms = case parseInt syms of
                    Result e' syms -> Result (Sub e e') syms
                    Error str -> Error str
```

Implementierung in Haskell

```
parseSum :: [IntE] -> [Symbol] -> Result IntE
parseSum es syms = case parseInt syms of
                       Result e (Plus:syms) -> parseSum (es++[e]) syms
                       Result e syms -> Result (Sum (es++[e])) syms
                       Error str -> Error str

parseProd :: [IntE] -> [Symbol] -> Result IntE
parseProd es syms = case parseInt syms of
                        Result e (Times:syms) -> parseProd (es++[e]) syms
                        Result e syms -> Result (Prod (es++[e])) syms
                        Error str -> Error str
```

Implementierung in Haskell

```
parseBoolE :: [Symbol] -> Result BoolE
parseBoolE []             = Error "no Boolean expression"
parseBoolE (True_:syms)   = Result (BoolE True) syms
parseBoolE (False_:syms)  = Result (BoolE False) syms
parseBoolE (Neg:syms)     = case parseBoolE syms of
                                 Result be syms -> Result (Not be) syms
                                 Error str -> Error str
parseBoolE syms           = case parseInt parseRest syms of
                            Result e (GR:syms)
                             -> case parseInt parseRest syms of
                                Result e' syms -> Result (Greater e e') syms
                                Error str -> Error str
                            Error str -> Error str
                            _ -> Error "missing >"
```

Σ-Algebra

Definition 3.2.8 Sei $\Sigma = (S, F)$ eine Signatur. Eine Σ-**Algebra** ist ein Paar (A, OP), kurz: A, bestehend aus einer

- S-sortierten Menge A
- einer Menge OP von Funktionen (oftmals **Operationen** genannt), wobei
  - OP aus **Interpretationen** von F besteht,
  - ☞ für alle $f : w \to s \in F$ gibt es genau eine Funktion $f^A : A_w \to A_s \in OP$.
  - Die Menge A_s, $s \in S$, heißt **Trägermenge** oder **Datenbereich von** s.

Eine Σ-Algebra kennen wir schon: die Σ-(Grund-)Termalgebra T_Σ. Ihre Trägermengen bzw. Funktionen sind wie folgt definiert:

- Für alle $s \in S$ ist $T_{\Sigma,s}$ die Trägermenge von s.
- Für alle $f : w \to s \in \Sigma$ und $t = (t_1, ..., t_n) \in T_{\Sigma,w}$ ist der Wert von f^{T_Σ} an der Stelle t der Σ-Term $f(t)$, kurz: $f^{T_\Sigma} =_{def} f(t)$.

Σ-Algebra

Sei Σ die abstrakte Syntax einer CF-Grammatik $G = (N, T, P, S)$. Dann bildet die von G erzeugte Sprache $L(G)$ eine weitere Σ-Algebra. Ihre Trägermengen bzw. Funktionen sind wie folgt definiert:

- Für alle $A \in N$ ist $\{w \in T^* \mid A \xrightarrow{*}_G w\}$ die Trägermenge von A.
- Für jede Produktion $p = (A \to w_0 A_1 w_1 \dots A_n w_n)$ von G und $v = (v_1 \dots v_n) \in L(G)_{A_1 \dots A_n}$ ist der Wert von $f_p^{L(G)}$ an der Stelle v das terminale Wort $w_0 v_1 w_1 \dots v_n w_n$, kurz:

$$f_p^{L(G)}(v_1 \dots v_n) \quad =_{def} \quad w_0 v_1 w_1 \dots v_n w_n.$$

In der ursprünglichen Definition der von G erzeugten Sprache war $L(G)$ auf die Trägermenge des Startsymbols S von G beschränkt. Demgegenüber haben wir $L(G)$ hier zu einer N-sortierten Menge erweitert.

Σ-Algebra

Auch die Menge $Abl(G)$ der Ableitungsbäume von G bildet eine Σ-Algebra, die wir mit $Abl(G)$ bezeichnen. Ihre Trägermengen bzw. Funktionen sind wie folgt definiert:

- Für alle $A \in N$ ist $\{B \in Abl(G) \mid A \text{ ist die Wurzel von } B\}$ die Trägermenge von A.
- Für jede Produktion $p = (A \to w_0 A_1 w_1 \dots A_n w_n)$ von G und $B = (B_1 \dots B_n) \in Abl(G)_{A_1 \dots A_n}$ ist der Wert von $f_p^{Abl(G)}$ an der Stelle B der Ableitungsbaum $A(w_0 B_1 w_1 \dots B_n w_n)$, kurz:

$$f_p^{Abl(G)}(B_1 \dots B_n) \quad =_{def} \quad A(w_0 B_1 w_1 \dots B_n w_n).$$

Homomorphismus

Zurück zu einer beliebigen Signatur Σ.
Das Besondere an der Σ-Termalgebra T_Σ ist, dass es von T_Σ zu jeder Σ-Algebra A einen eindeutigen (!) Σ-**Homomorphismus** $eval^A$ gibt, d.i. eine S-sortierte Funktion, die mit den Interpretationen der Funktionssymbole von Σ in T_Σ bzw. A vertauschbar ist: Für alle $s \in S$ und $f : w \to s \in \Sigma$ gilt

$$eval^A_s \circ f^{T_\Sigma} \quad = \quad f^A \circ eval^A_w.$$

☞ Jeder Interpreter oder Compiler einer CF-Grammatik $G = (N, T, P, S)$ entspricht dem eindeutigen $\Sigma(G)$-Homomorphismus von $B(G) = T_{\Sigma(G)}$ in eine Σ-Algebra.

Definition 3.2.9 Sei $\Sigma = (S, F)$ eine Signatur und A eine Σ-Algebra. Der eindeutige Σ-Homomorphismus $eval^A : T_\Sigma \to A$ heißt **Auswertungsfunktion** in A.

Kommando-Signatur

Den vier Datentypen `Block`, `Command`, `IntE` und `BoolE` aus dem obigen Beispiel entspricht folgende Signatur $\Sigma = (S, F)$:

$$
\begin{array}{lcll}
S & = & \{ & block, command, intE, boolE\} \\
F & = & \{ & mkBlock : [command] \to block, \\
 & & & [] :\to [command], \quad \_ : \_ : command \times [command] \to [command], \\
 & & & skip :\to command, \quad assign : String \times intE \to command, \\
 & & & cond : boolE \times block \times block \to command, \\
 & & & loop : boolE \times block \to command, \\
 & & & mkIntE : Int \to intE, \quad var : String \to intE, \\
 & & & sub : intE \to intE \to intE, \\
 & & & sum : [intE] \to intE, \quad prod : [intE] \to intE, \\
 & & & [] :\to [intE], \quad \_ : \_ : intE \times [intE] \to [intE], \\
 & & & mkBoolE : Bool \to boolE, \quad greater : boolE \times boolE \to boolE, \\
 & & & not : boolE \to boolE\}.
\end{array}
$$

Kommando-Signatur

Die entsprechende Σ_{com}-Algebra A entsteht durch:

- Interpretation der Sorten von Σ_{com}:
  Sei *State* die Menge der Funktionen von *String* nach $\mathbb{Z}$, also *State* $=$ *(String* $\to \mathbb{Z}$*)*. Dann ist
  - $\mathit{block}^A = \mathit{command}^A = (\mathit{State} \to \mathit{State})$.
  - $\mathit{intE}^A = (\mathit{State} \to \mathbb{Z})$
  - $\mathit{boolE}^A = (\mathit{State} \to \{\mathit{True}, \mathit{False}\})$

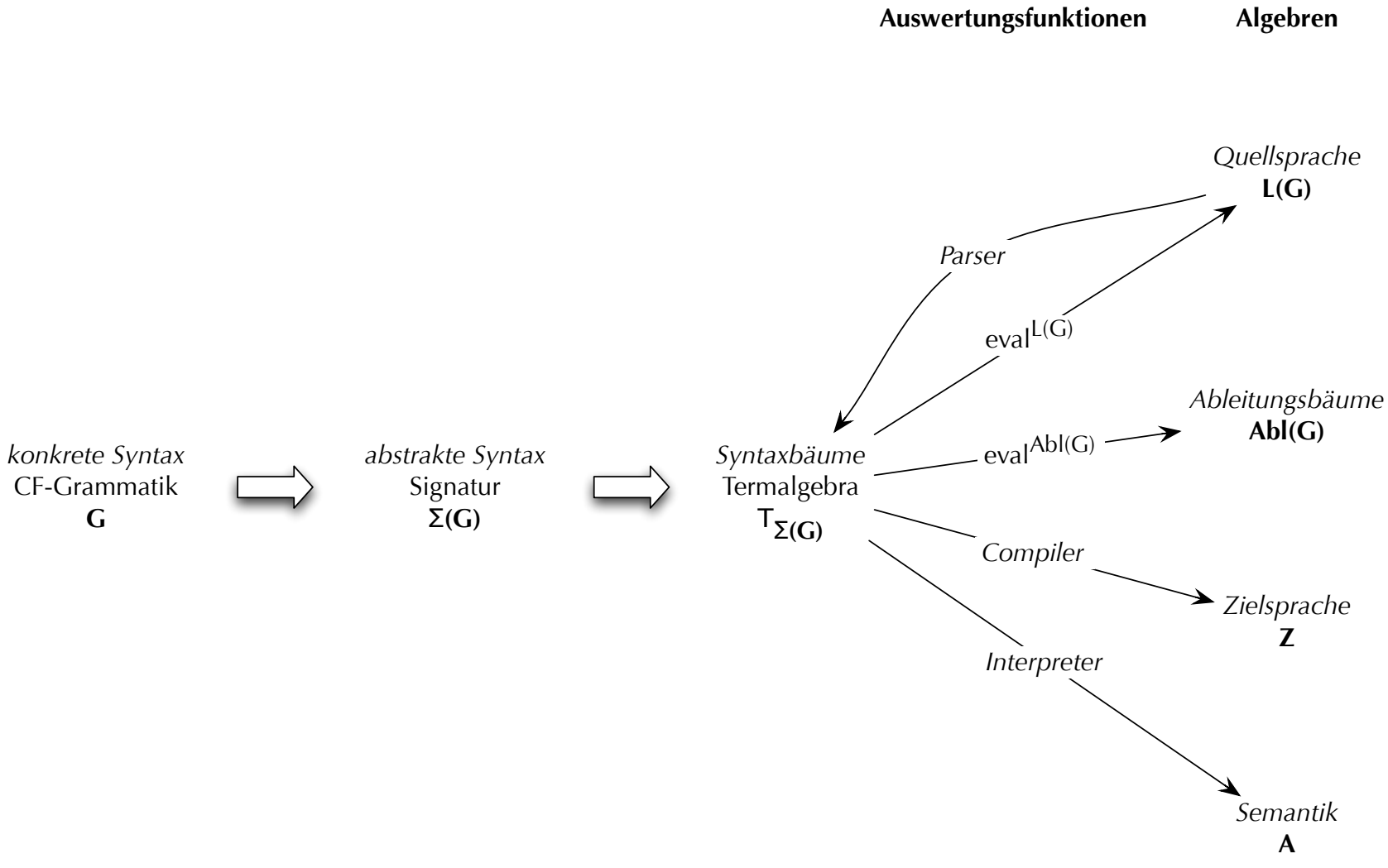
Kommando-Signatur

- Interpretation jedes Funktionssymbols $f : w \to s$ von Σ_{com} als Funktion $f^A : A_w \to A_s$.
  Dann ist z.B. *assign : String* $\times$ *intE* $\to$ *Command* als Funktion
  - assign^A *: (String* $\times$ *IntE*^{A}*)* $\to$ *Command*^{A}
    $= (\mathit{String} \times (\mathit{State} \to \mathbb{Z})) \to (\mathit{State} \to \mathit{State})$.

Aus der Rückübersetzung von `evalCom (Assign x e)` ergibt sich die folgende Definition von $assign^A$: Für alle $x, y \in String$, $f : State \to \mathbb{Z}$ und $st \in State$ ist

$$assign^A(x, f)(st)(y) = \begin{cases} f(st) & \text{falls } x = y \\ st(y) & \text{sonst} \end{cases}$$

Von Grammatiken zu Algebren



Kommando-Signatur

Umgekehrt lässt sich jede Signatur $\Sigma = (S, F)$ mit Sortenmenge $S = \{s_1, \ldots, s_n\}$ durch n Datentypen $S_1, \ldots, S_n$ und eine zusammenfassende Typklasse mit generischen Auswertungsfunktionen implementieren:

```
data S1 = F Sf1 ... Sfk | .                    für alle f:sf1...sfk→ s1 ∈ F
...
data Sn = G Sg1 ... Sgm | ...                  für alle g:sg1...sgm→ sn ∈ F
class Sigma s1 ... sn where
      f :: sf1 -> ... -> sfk -> s1             für alle f:sf1...sfk→ s1 ∈ F
      ...
      g :: sg1 -> ... -> sgm -> sn             für alle g:sg1...sgm→ sn ∈ F
      eval_s1 :: S1 -> s1
      eval_s1 (F a1 ... ak) = f (eval_sf1 a1) ... (eval_sfk ak)
                                               für alle f:sf1...sfk→ s1 ∈ F
      ...
      eval_sn :: Sn -> sn
      eval_sn (G a1 ... am) = g (eval_sg1 a1) ... (eval_sgm am)
                                               für alle g:sg1...sgm→ sn ∈ F
```

wobei für alle Typen $s \notin \{s_1, \ldots, s_n\}$, $eval_s$ die jeweilige Identitätsfunktion ist.

Kommando-Algebra

```
class Sigma block command intE boolE where
      mkBlock :: [command] -> block
      skip    :: command
      assign  :: (String,intE) -> command
      cond    :: (boolE,block,block) -> command
      loop    :: (boolE,block) -> command
      mkIntE  :: Int -> intE
      var     :: String -> intE
      sum_    :: [intE] -> intE
      prod    :: [intE] -> intE
      sub     :: (intE,intE) -> intE
      mkBoolE :: Bool -> boolE
      greater :: (boolE,boolE) -> boolE
      not_     :: boolE -> boolE

      eval_Block :: Block -> block
      eval_Block cs = mkBlock cs
```

Kommando-Algebra

```
eval_Command :: Command -> command
eval_Command Skip              = skip
eval_Command (Assign x e)      = assign (x,eval_IntE e)
eval_Command (Cond be cs cs') = cond (eval_BoolE be,
                                      eval_Command cs,
                                      eval_Command cs')
eval_Command (Loop be cs)      = loop (eval_BoolE be,
                                      eval_Command cs)
eval_IntE :: IntE -> intE
eval_IntE (IntE i)   = mkIntE i
eval_IntE (Var x)    = var x
eval_IntE (Sub e e') = sub (eval_IntE e,eval_IntE e')
eval_IntE (Sum es)   = sum_ (map eval_IntE es)
eval_IntE (Prod es)  = prod (map eval_IntE es)
```

Kommando-Algebra

```
eval_BoolE :: BoolE -> boolE
eval_BoolE (BoolE b) = mkBoolE b
eval_BoolE (Greater e e') = greater (eval_IntE e,eval_IntE e')
eval_BoolE (Not be) = not_ (eval_BoolE be)
```

Interpreter der Sprache als Instanz von Sigma

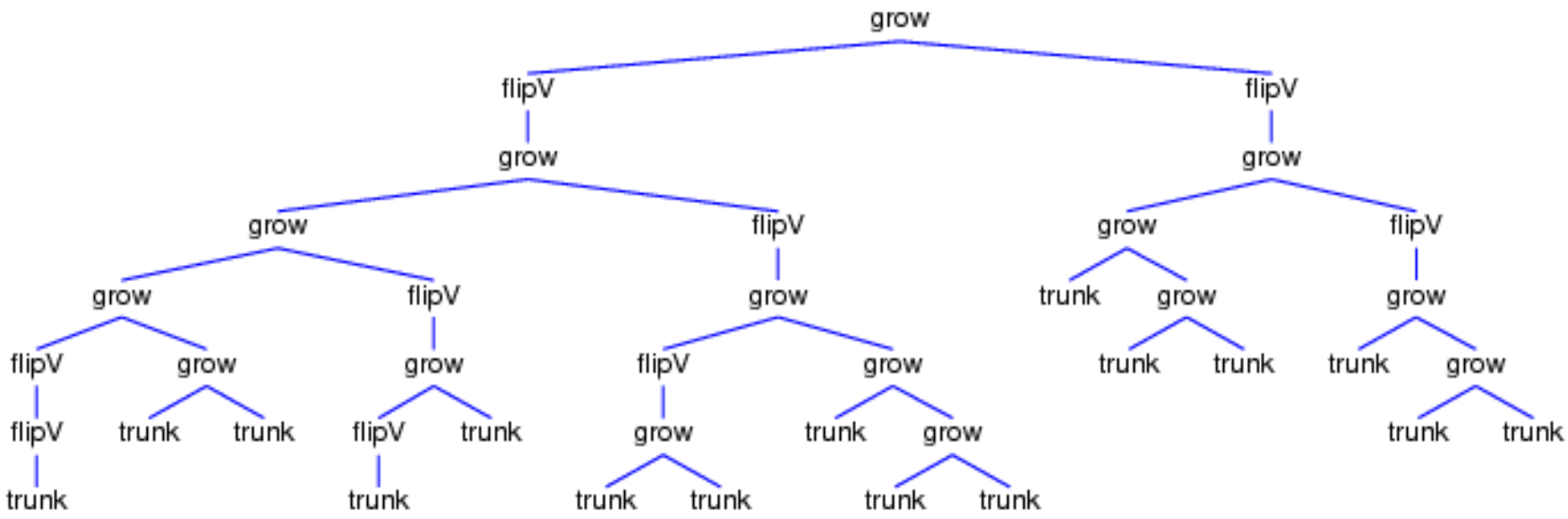
```
instance Sigma (State -> State) (State -> State)
               (State -> Int) (State -> Bool) where
         mkBlock = foldl (flip (.)) id
         skip = id
         assign (x,f) st y = if x == y then f st else st y
         cond (f,g,h) st = if f st then g st else st
         loop (f,g) = cond (f,loop (f,g) . g,id)
         intE i _ = i
         var x st = st x
         sub (f,g) st = f st - g st
         sum_ fs st = foldl (+) 0 [f st | f <- fs]
         prod fs st = foldl (*) 1 [f st | f <- fs]
         boolE b _ = b
         greater (f,g) st = f st > g st
         not_ f st = not (f st)
```

Allgemeine Termdarstellung und -auswertung in einer Algebra

```
data Term op = F op [Term op]

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)
```

Ein Syntaxbaum und seine Interpretation



Ein Syntaxbaum und seine Interpretation

Parse-Funktion

- **Top-Down** (shift-derive) Parser erzeugen die Syntaxbäume top-down.
- **Buttom-Up** (shift-reduce) Parser erzeugen die Syntaxbäume buttom-up.

Parser sind von ihrer Implementierung her **iterative Funktionen**:
Definition 3.2.12 Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik und Q eine Zustandsmenge. Eine berechenbare Funktion

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

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

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

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

Parse-Funktion

Ist G regulär, dann ist Q die Zustandsmenge eines Automaten, der die von G erzeugte Sprache erkennt. Die zugehörige parse-Funktion lautet wie folgt:

$$parse(xw, q) = parse(w, \delta_A(q, x)) \quad \textbf{"shift"}$$

$$parse(\epsilon, q) = \begin{cases} true, \text{ falls } q \in E_A & \textbf{"accept"} \\ false, \text{ sonst} & \textbf{"error"} \end{cases}$$

Definition der 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 $A \in N$.

$$
\begin{aligned}
first_k(\alpha) &= \{w \in T^k \mid \exists v \in T^* : \alpha \xrightarrow[G]{*} wv\} \\
&\quad \cup \{w \in T^{<k} \mid \alpha \xrightarrow[G]{*} w\} \\
follow_k(A) &= \{w \in T^k \mid \exists u, v \in T^* : S \xrightarrow[G]{*} uAwv\} \\
&\quad \cup \{w \in T^{<k} \mid \exists u \in T^* : S \xrightarrow[G]{*} uAw\} \\
first(\alpha) &= first_1(\alpha) \\
follow(A) &= follow_1(A)
\end{aligned}
$$

Definition der First- und Follow-Wortmengen

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

- $\varepsilon \in \ first(\varepsilon)$
- $x \in T \Rightarrow x \in \ first(x\alpha)$
- $(A \to \alpha) \in P \wedge x \in \ first(\alpha\beta) \Rightarrow x \in first(A\beta)$.

Implementierung in Haskell

```
first :: String -> [String]
first ""    = [""]
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)
firstPlus = loop1 init (const True)
            where init = fold2 upd (const []) terminals (map single terminals)
                  single x = [x]

loop1 :: (String -> [String]) -> (String -> Bool)
         -> (String -> [String],String -> Bool)
loop1 f notNullable = if b then loop1 f' notNullable' else (f,notNullable)
                      where (b,f',notNullable') = loop2 rules False f
                                                  notNullable
```

Implementierung in Haskell

```
loop2 :: [(String,String)] -> Bool -> (String -> [String]) -> (String -> Bool)
           -> (Bool,String -> [String],String -> Bool)
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))
                  _ -> loop2 rules (b || notNullable a) f
                                   (upd notNullable a False)
loop2 _ b f notNullable = (b,f,notNullable)
```

Implementierung in Haskell

```
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
```

Definition der LR(k)-Grammatik

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 durchgeführt werden soll.

Definition der LR(k)-Grammatik

Formal:

Sind

$$\begin{array}{lllll} S & \stackrel{*}{\rightarrow} & \gamma A v & \rightarrow & \gamma \alpha v \\ S & \stackrel{*}{\rightarrow} & \gamma' A' w' & \rightarrow & \gamma' \alpha' w' = \gamma \alpha w \end{array}$$

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

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

Beispiel 3.5.2

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

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

ist für kein k eine LR(k)-Grammatik, da es einen **Shift-Reduce-Konflikt** gibt.

LR(k)-Kriterium

Die LR(k)-Eigenschaft lässt sich mithilfe des folgenden **LR(k)-Kriteriums** entscheiden:

G ist eine LR(k)-Grammatik, wenn für je zwei verschiedene direkte Rechtsableitungen

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

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

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

☞ Das LR(k)-Kriterium ist schwierig zu entscheiden, da je zwei Ableitungen miteinander verglichen werden.

1.Schritt

Wir entwickeln den LR(1)-Parser in drei Schritten.

1.Schritt: Wir beginnen mit folgender parse-Funktion:

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

"shift"

$$parse^1_{LR}(xw, \varphi) = parse^1_{LR}(w, \varphi x) \quad \text{falls } S \overset{*}{\longrightarrow} \gamma A v,\ (A \to \alpha\beta) \in P,\ \beta \neq \varepsilon,\ \varphi = \gamma\alpha,\ x \in first(\beta follow(A))$$

"reduce"

$$parse^1_{LR}(w, \varphi) = parse^1_{LR}(w, \gamma A) \quad \text{falls } S \overset{*}{\longrightarrow} \gamma A v,\ A \neq S,\ (A \to \alpha) \in P,\ \varphi = \gamma\alpha,\ first(w) \in follow(A)$$

1.Schritt

"accept"
$parse^1_{LR}(\varepsilon, \varphi) \quad = \quad true \quad$ falls $(S \to \varphi) \in P$

"error"
$parse^1_{LR}(w, \varphi) \quad = \quad false \quad$ sonst

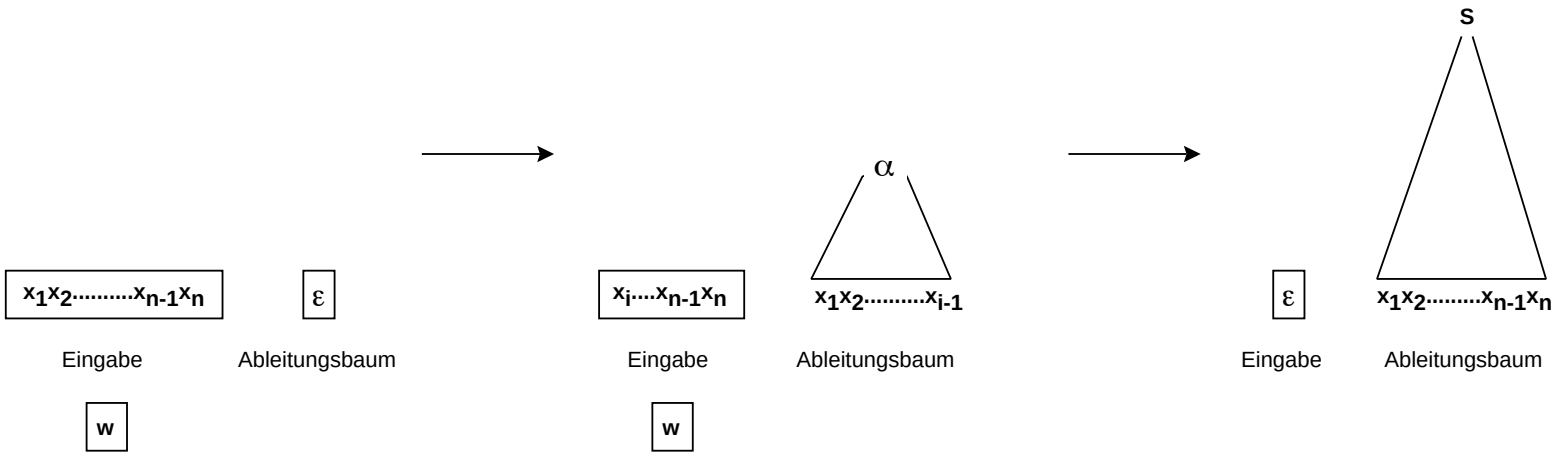
Durch Induktion über die Definition von $parse^1_{LR}$ erhält man sofort:

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

also insbesondere:

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

1.Schritt



$parse^1_{LR}$ ist wohldefiniert:

- jeder Leseschritt (*shift*) verkürzt das Eingabewort
- jeder Reduktionsschritt (*reduce*) verringert den Abstand von φ zum Startsymbol S. .
- jeder Reduktionsschritt ist eindeutig, weil G eine LR(1)-Grammatik ist
- für die Entscheidbarkeit der "reduce"-Bedingung wird die Relation **item(φ)** eingeführt

item(φ)

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

ist folgendermaßen definiert:

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

- 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:

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

2.Schritt

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

Umgekehrt soll sich $parse^1_{LR}$ folgendermaßen aus $parse^2_{LR}$ ergeben:

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

Dementsprechend wird die Korrektheitsbedingung zu:

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

2.Schritt

Die zuletzt angegebenen Definitionen von $parse^1_{LR}$ werden in eine Definition von $parse^2_{LR}$ umgewandelt:

$$parse^2_{LR}(xw, \varphi : a) = parse^2_{LR}(w, \varphi x : \varphi : a)$$
falls $(A \to \alpha.\beta, y) \in item(\varphi), \beta \neq \varepsilon, x \in first(\beta y)$ "shift"

$$parse^2_{LR}(w, \varphi_1 : \cdots : \varphi_{|\alpha|} : \gamma : a) = parse^2_{LR}(w, \gamma A : \gamma : a)$$
falls $A \neq S,\ (A \to \alpha., first(w)) \in item(\varphi_1)$ "reduce"

$$parse^2_{LR}(\varepsilon, \varphi : a) = true$$
falls $(S \to \alpha., \varepsilon) \in item(\varphi)$ "accept"

$$parse^2_{LR}(w, a) = false$$
sonst "error"

3.Schritt

$parse^2_{LR}$ arbeitet auf der Zustandsmenge $Q =_{def} (N \cup T)^*$ mit dem Anfangszustand $q_0 =_{def} \varepsilon$.

Für eine Eingabemenge X und eine Sprache $L \subseteq X^*$ heißt der (**unendliche!**) erkennende Automat

$$I = (X^*, X, \{0,1\}, \delta, \beta, \varepsilon)$$

initial bzgl. L, wenn seine Übergangsfunktion $\delta_I : X^* \times X \to X^*$ und seine Ausgabefunktion wie folgt definiert sind:

$$\delta_I(w,x) = wx, \qquad \beta(w) = 1 \Longleftrightarrow_{def} w \in L.$$

I erkennt L heißt initial, weil jeder Automat A, der L erkennt, Bild eines Homomorphismus $h : I \to A$ ist.

3.Schritt

Wir ersetzen den in $parse^2_{LR}$ implizit benutzten initialen Automaten I für $\{\varphi \mid S \to \varphi \in P\}$ durch das folgende homomorphe Bild B von I:

$$\begin{aligned} B &= (Q, N \cup T, \{0,1\}, \delta, \beta, q_0) \\ Q &= \{item(\varphi) | \varphi \in (N \cup T)^*\} \\ \delta(item(\varphi), z) &= item(\varphi z) \\ \beta(item(\varphi)) = 1 &\Leftrightarrow S \to \varphi \in P \\ q_0 &= item(\varepsilon) \end{aligned}$$

3.Schritt

Die Beziehung zwischen $parse^2_{LR}$ und der endgültigen LR(1)-parse-Funktion

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

ist durch die Gleichung

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

gegeben. Die Korrektheitsbedingung wird zu:

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

3.Schritt

Aus der Definition von $parse^2_{LR}$ und

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

ergibt sich sofort folgende Definition von $parse_{LR}$:

Sei $q, q_1, \ldots, q_n \in Q$ und $a \in Q^*$.

$parse_{LR}(xw, q : a) = parse_{LR}(w, \delta(q, x) : q : a)$
falls $(A \to \alpha.\beta, y) \in q,\ \beta \neq \varepsilon,\ x \in first(\beta y)$ "shift"

$parse_{LR}(w, q_1 : \cdots : q_{|\alpha|} : q : a) = parse_{LR}(w, \delta(q, A) : q : a)$
falls $A \neq S,\ (A \to \alpha., first(w)) \in q_1$ "reduce"

$parse_{LR}(\varepsilon, q : a) = true$
falls $(S \to \alpha., \varepsilon) \in q$ "accept"

$parse_{LR}(w, a) = false$
sonst "error"

3.Schritt

Zur Beantwortung der Frage

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

müssen die Items, aus denen sich q zusammensetzt, angesehen werden.

Hier noch einmal die Definition des Zustandes

$$q = item(\varphi)\text{: } (A \to \alpha.\beta, first(v)) \in item(\gamma\alpha) \Leftrightarrow S \xrightarrow{*} \gamma A v \wedge (A \to \alpha\beta) \in P.$$

Die gesamte Zustandsmenge wird mithilfe der folgenden induktiven Definition von Q und δ konstruiert:

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

3.Schritt - Implementierung

Ähnlich der induktiven Definition von $first(\alpha)$ lassen sich (1)-(3) als Schleife implementieren:

```
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

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

symbols = nonterminals++terminals

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

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

3.Schritt - Implementierung

```
extend :: LRState -> LRState
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)
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
```

3.Schritt - Implementierung

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

mkLRauto :: ([Int],[(Int,String,Int)])
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')
loop _ qs2 rel = (qs2,rel)
```

3.Schritt - Implementierung

```
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]
add s@(x:s') y = if equal x y then s else x:add s' y
add _ x        = [x]
```

Aktionstabelle

Nach der Konstruktion von Q und δ wird die **Aktionstabelle**

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

angelegt, auf die dann $parse_{LR}$ zurückgreift.

Sei $x \in T$.

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

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

Aktionstabelle

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

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

Beispiel 3.5.3

Gegeben sei die 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")]
```

Beispiel 3.5.3

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

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

Beispiel 3.5.3

Implementierung in Haskell:

`fst automaton` liefert Q in folgender Form:

```
[[("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","","*")]]
```

Beispiel 3.5.3

`snd automaton` liefert δ als Relation auf den Codierungen von Zuständen und Symbolen durch ihre jeweiligen Positionen in der Liste `states` bzw. `symbols`:

```
[(0,1,1), (0,2,2), (0,3,3), (0,4,4), (1,5,5), (5,2,6), (5,3,3), (5,4,4)]
```

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

	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	$A \to A * B$

Beispiel 3.5.3

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

SLR(1)-Grammatiken

- Eine Variante der LR(1)-Analyse besteht darin, zunächst die Menge Q_0 der erreichbaren **LR(0)-Zustände** zu bestimmen.
  - Konstruktion von Q_0 folgt der induktiven Definition von Q
  - anstelle von Paaren $(A \to \alpha.\beta, x)$ wird nur $A \to \alpha.\beta$ gebildet
  - Nur wenn die 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**

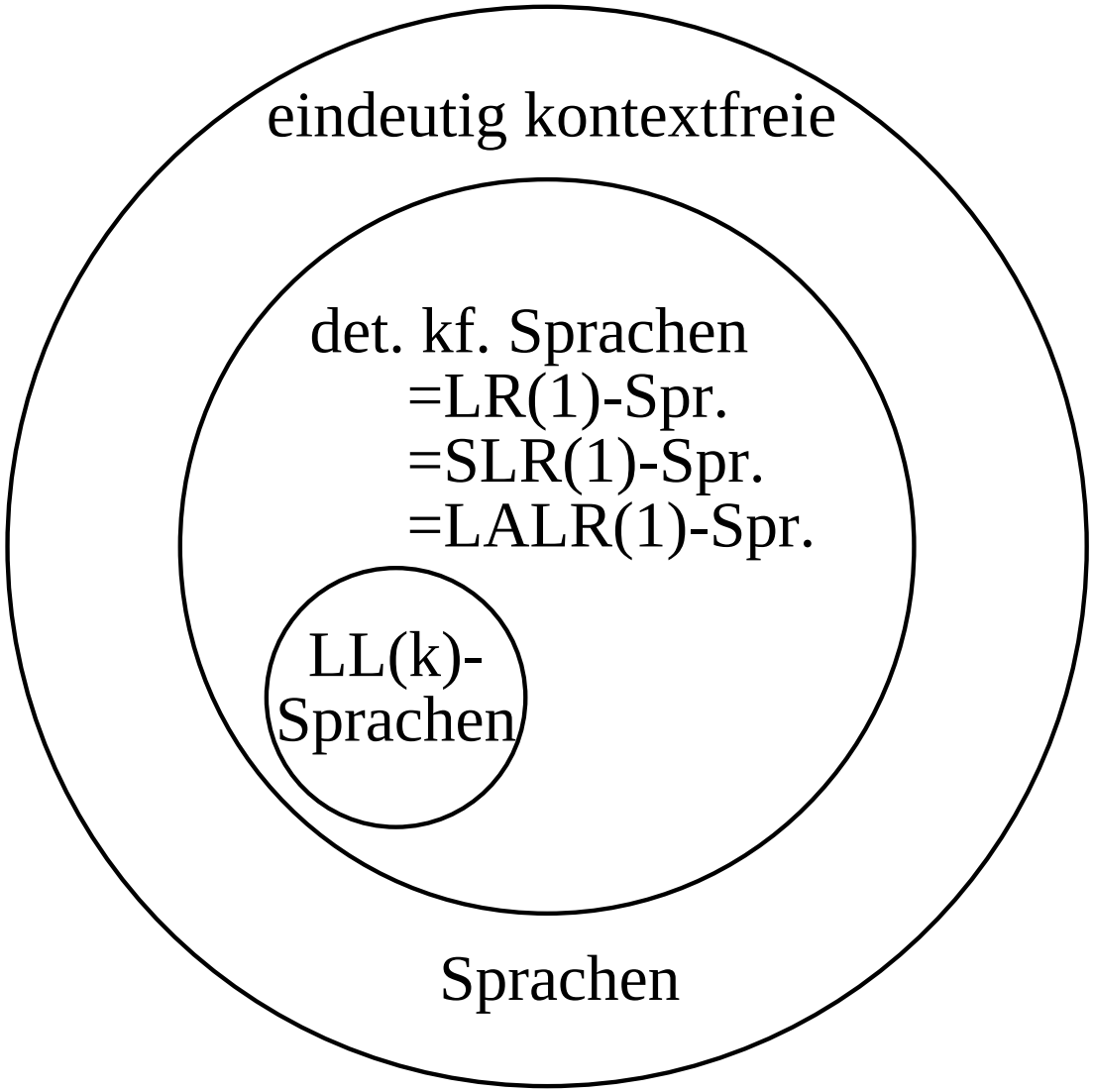
LALR(1)-Grammatiken

- Q_0 ist i.a. gröber als Q.
  Eine schwächere Vergröberung von Q erreicht man durch die Identifizierung aller LR(1)-Zustände q, q', für die gilt

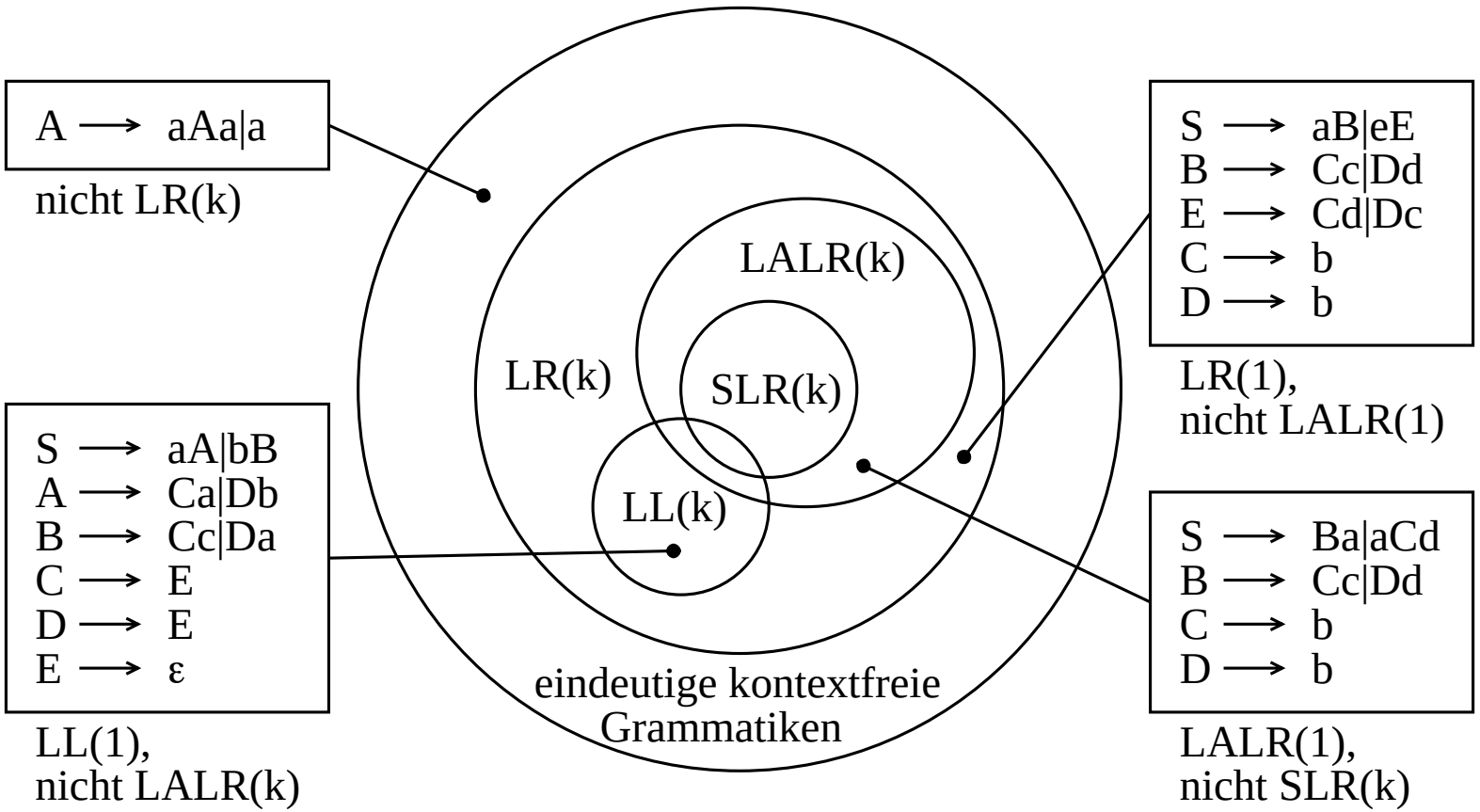
$$\{p \mid \exists l : (p, l) \in q\} = \{p \mid \exists l : (p, l) \in q'\}$$

  - ☞ Erhält man auf diese Weise eine konfliktfreie Aktionstabelle, dann heißt G **LALR(1)-Grammatik**

Sprachklassen



Grammatikklassen



Parse^B_{LR}

- Die LR(1)-parse-Funktion soll zu einem vollständigen Parser erweitert werden, indem $parse_{LR}$ schrittweise Syntaxbäume erzeugt.
  $parse_{LR}$ hat also jetzt folgenden Typ:
  $$parse^B_{LR} \ : \ T^* \times Q^* \times B(G)^* \ \rightarrow \ B(G) \cup \{error\}.$$
- Parallel zur bottom-up-Konstruktion einer Ableitung aus dem Startsymbol liefert $parse^B_{LR}$ iterativ den entsprechenden Syntaxbaum. Es soll gelten:
  $parse^B_{LR}(w, [q_0], []) = t \ \Leftrightarrow \ t$ ist ein Syntaxbaum für $S \stackrel{*}{\rightarrow} w$.

Parse^B_{LR}

- Dies ist ein Spezialfall der allgemeinen Anforderung an $parse^B_{LR}$, die folgendermaßen lautet:

$$parse^B_{LR}(w, item(\varphi) : a, [t_1, \ldots, t_n]) = t[t_1/x_1, \ldots, t_n/x_n]$$

$$\Leftrightarrow \begin{cases} t \text{ ist ein Syntaxbaum für die Ableitung } S \xrightarrow{*} \varphi w, \\ var(t) = \{x_1 : s_1, \ldots, x_n : s_n\}, \\ \exists\ w_0, \ldots, w_n \in T^* : \varphi = w_0 s_1 w_1 \ldots s_n w_n. \end{cases}$$

Parse^B_{LR}

Die Definition von $parse^B_{LR}$ folgt derjenigen von $parse_{LR}$.
$||\alpha||$ bezeichnet die Anzahl der Nichtterminale von α.

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

Parse_{LR}^{B}

Hier ist ein mit **Expander2** erstellter Lauf von $parse_{LR}^{B}$ auf der Eingabe $c * d$.

- Hinter `TransL` und `Actions` verbirgt sich die δ bzw. act_{LR}.
- In der Argumentliste von `parse` wird hinter `Actions` die Liste der Syntaxbäume aufgebaut.

```
parse(c mul d,[0],TransL,Actions)
parse(mul d,[3,0],TransL,Actions)
parse(mul d,[2,0],TransL,Actions,r4)
parse(mul d,[1,0],TransL,Actions,r3(r4))
parse(d,[5,1,0],TransL,Actions,r3(r4))
parse(,[4,5,1,0],TransL,Actions,r3(r4))
parse(,[6,5,1,0],TransL,Actions,r3(r4),r5)
parse(,[1,0],TransL,Actions,r2(r3(r4),r5))
r1(r2(r3(r4),r5))
```

Monadische Parser

In diesem Abschnitt werden Parser entwickelt, die sich von LL- und LR-Parsern in dreierlei Hinsicht unterscheiden:

- Auf der Eingabe ist **Backtracking** möglich. Es handelt sich also in der Regel um nichtdeterministische Parser. Sie lassen sich demzufolge auch für nicht deterministisch kontextfreie, ja sogar für mehrdeutige Sprachen schreiben.
- Monadische Parser werden hierarchisch mithilfe von **Parserkombinatoren** aus Teilparsern zusammengesetzt. Ein vergleichbares *Kompositionalitätsprinzip* wird in Abschnitt 3.4 verwendet, wo die parse-Funktion $parse^{B}_{LL}$ zurückgreift auf Parser der Form $\overline{A}$ für aus einem einzelnen Nichtterminal A ableitbare terminale Wörter. Das Gleiche gilt für den Parser von Beispiel.

Monadische Parser

In diesem Abschnitt werden Parser entwickelt, die sich von LL- und LR-Parsern in dreierlei Hinsicht unterscheiden (Fortsetzung):

- Monadische Parser lassen sich einfach **mit Scannern kombinieren**, d.h. an die Stelle zweier Funktionen, von denen die erste Zeichen- in Symbolfolgen überführt und die zweite Symbolfolgen in Syntaxbäume, tritt eine einzige, die Zeichenfolgen direkt in Syntaxbäume umwandelt.

Eine frei verfügbare Haskell-Bibliothek monadischer Parser steht unter www.cs.uu.nl/∼daan/parsec.html.

Monaden in Haskell

Eine **Monade** m ist ein generischer Datentyp mit einem Parametertyp a. In Haskell werden Monaden als Instanzen der folgenden Typklasse definiert:

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

Monaden in Haskell

>>= simuliert die imperativen Sprachkonstrukte Zuweisung und Sequentialisierung. Das wird deutlich, wenn man die do-Notation verwendet und anstelle von

```
p0 >>= (\x1 -> p1 >>= (\x2 -> p2 >>= (\x3 -> ...
                                  p(n-1) >>= (\xn -> pn) ... )))
```

die semantisch äquivalente Form benutzt:

```
do x1 <- p0; x2 <- p1; x3 <- p2; ... xn <- p(n-1); pn
```

Monaden in Haskell

Die Monade `pi` vom Typ `m a` liefert eine "Ausgabe" vom Typ `a`, die der Variablen `xi` zugewiesen wird. Was die Monade sonst noch tut, wird durch eine Instanziierung der Klasse `Monad m`, d.h. der Operationen >>= und `return` festgelegt.
Dazu muss man zunächst `m` durch einen Datentyp instanziieren. So ist z.B.

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

ein Standarddatentyp von Haskell, den man zur Totalisierung partieller Funktionen verwendet.
`Nothing` steht dabei für "undefiniert". Bei der Hintereinanderausführung zweier partieller Funktionen f und g muss `Nothing` durchgereicht werden. Das könnte man folgendermaßen implementieren:

```
compose :: (a -> Maybe b) -> (b -> Maybe c) -> a -> Maybe c
compose f g a = case f a of Just b -> g b
                            Nothing -> Nothing
```

Monaden in Haskell

Diese Besonderheit der Komposition partieller Funktionen lässt sich "verstecken", wenn man `Maybe` wie folgt zur Monade macht:

```
instance Monad Maybe where
         Just a  >>= f  =  f a
         Nothing >>= f  =  Nothing
         return         =  Just
```

Nun kann man `compose` "imperativ" implementieren und das Durchreichen von `Nothing` wird zum unsichtbaren Seiteneffekt:

```
compose f g a = do b <- f a; g b
```

Die IO-Monade

Die wichtigste in Haskell eingebaute Monade ist die IO-Monade, die es erlaubt, alle möglichen Schreib- und Leseprozeduren als Funktionen auf Instanzen des Typs `IO a` zu formulieren.

Im Gegensatz zu `Maybe a` ist der hinter `IO a` verborgene Datentyp selbst unsichtbar. Es handelt sich nämlich um einen Typ von *Zustandstransformationen*, wobei der Typ der Zustände maschinenabhängig ist:

```
type IO a = state -> (a,state)
instance Monad IO where
         (m >>= f) st = f a st' where m st = (a,st')
         return a st = (a,st)
```

Parser-Monade

Wie die IO-Monade, so bestehen auch Parser-Monaden aus Funktionen. Wir benutzen den in Abschnitt 1.2 eingeführten Typ

```
data Result a  = Result a [Symbol] | Error String
```

für die Ausgaben des Parser:

Die Typvariable a wird durch (Mengen von) Syntaxbäumen instanziiert. Fehlerhafte Resteingaben werden in `[Symbol]` abgelegt, Fehlermeldungen mit dem Konstruktor `Error` gekapselt.

Es handelt sich hier um *deterministische* Parser. Nichtdeterministische können mehrere Syntaxbäume liefern, hätten also den Ergebnistyp

```
data Result a  = [Result a [Symbol]] | Error String
```

Parser-Monade

Wie jede selbstdefinierte Monade muss auch eine Parser-Monade auf einem Datentyp aufgebaut werden. Da der Parser Symbolfolgen verarbeiten soll, lautet sie wie folgt:

```
newtype Parser a = P ([Symbol] -> Result a)
```

Die folgende Funktion braucht man, um die in `P` gekapselte Funktion anzuwenden:

```
apply :: Parser a -> [Symbol] -> Result a
apply (P f) = f
```

Parser-Monade

Nun wird der Datentyp `Parser` zur Monade erweitert:

```
instance Monad Parser where
         P f >>= g  = P h
             where h syms = case f syms of
                                Result a rest -> apply (g a) rest
                                Error str -> Error str
             return = P . Result
```

Die Funktion

```
>>= :: Parser a -> (a -> Parser b) -> Parser b
```

bewirkt die *Hintereinanderausführung* zweier Parser, wobei der zweite den vom ersten erzeugten Syntaxbaum als Parameter benutzen kann.

Parser-Monade

Backtracking hingegen wird durch die *Parallelausführung* mehrerer Parser erreicht: Scheitert der erste, dann wird der zweite auf dieselbe Eingabe angesetzt. Dazu definieren wir die Parserkombinatoren `orelse`, `zero` (der stets scheiternde Parser) und `tryseq` (die Parallelausführung einer ganzen Liste von Parsern):

```
orelse :: Parser a -> Parser a -> Parser a
P f `orelse` P g = P h where h syms = case f syms of p@(Result _ _) -> p
                                                     _ -> g syms

zero :: String -> Parser a
zero = P . const . Error

tryseq :: String -> [Parser a] -> Parser a
tryseq = foldr orelse . zero
```

Parser-Monade

Der einfachste Parser liest ein einzelnes Symbol:

```
item :: Parser Symbol
item = P f where f (sym:syms) = Result sym syms
       f _           = Error "no symbols"
```

Die Parser `some p` und `many p` wenden den Parser `p` wiederholt an. Bei `some p` muss mindestens eine Anwendung erfolgreich sein:

```
some :: Parser a -> Parser [a]
some p  = do x <- p; xs <- many p; return (x:xs)

many :: Parser a -> Parser [a]
many p  = some p `orelse` return []
```

Parser-Monade

`sat p f err` wendet den Parser `p` an, akzeptiert dessen Ergebnis aber nur dann, wenn es `f` erfüllt, sonst gibt's eine Fehlermeldung:

```
sat :: Parser a -> (a -> Bool) -> String -> Parser a
sat p f err = do x <- p; if f x then return x else zero err
```

`symbol sym` ist genau dann erfolgreich, wenn das aktuelle Eingabesymbol mit `sym` übereinstimmt. Die Ausgabe diese Parsers ist unwichtig. Deshalb ist sie vom (einelementigen) Typ `()`.

```
symbol :: Symbol -> Parser ()
symbol sym = do sat item (== sym) ("missing "++show sym); return ()
```

Diese Teilparser bzw. Parserkombinatoren genügen, um darauf einen zum Kommando-Parser von Beispiel **??** äquivalenten monadischen Parser aufzubauen.

Beispiel 3.6.1 Monadischer Kommando-Parser

```
monBlock :: Parser [Command]
monBlock = do symbol Lcur; cs <- monSeq; symbol Rcur; return cs

monCom :: Parser Command
monCom = tryseq "no command"
                [do symbol Semi; return Skip,
                 do x <- ident; symbol Upd; e <- monInt; symbol Semi
                    return (Assign x e),
                 do symbol If; symbol Lpar; be <- monBool; symbol Rpar
                    cs <- monBlock
                    tryseq "" [do symbol Else; cs' <- monBlock
                                  return (Cond be cs cs'),
                               return (Cond be cs [])],
                 do symbol While; symbol Lpar; be <- monBool; symbol Rpar
                    cs <- monBlock; return (Loop be cs)]
```

Beispiel 3.6.1 Monadischer Kommando-Parser

```
monInt :: (IntE -> Parser Symbol IntE) -> Parser Symbol IntE
monInt f = tryseq "no integer expression"
                  [do i <- number; f (IntE i),
                   do x <- ident; f (Var x),
                   do symbol Lpar; e <- monInt monRest; symbol Rpar; f e]

monRest :: IntE -> Parser Symbol IntE
monRest e = tryseq "" [do symbol Minus; monSub e,
                       do symbol Plus; monSum [e],
                       do symbol Times; monProd [e],
                       return e]

monSub :: IntE -> Parser Symbol IntE
monSub e = do e' <- monInt monRest; return (Sub e e')
```

Beispiel 3.6.1 Monadischer Kommando-Parser

```
monSum :: [IntE] -> Parser Symbol IntE
monSum es = do e <- monInt return
               tryseq "" [do symbol Plus; monSum (es++[e]),
                          return (Sum (es++[e]))]

monProd :: [IntE] -> Parser Symbol IntE
monProd es = do e <- monInt return
                tryseq "" [do symbol Times; monProd (es++[e]),
                           return (Prod (es++[e]))]
```

Beispiel 3.6.1 Monadischer Kommando-Parser

```
monBool :: Parser Symbol BoolE
monBool = tryseq "no Boolean expression"
                 [do symbol True_; return (BoolE True),
                  do symbol False_; return (BoolE False),
                  do symbol Neg; be <- monBool; return (Not be),
                  do e <- monInt monRest; symbol GR; e' <- monInt monRest
                     return (Greater e e')]
```

Beispiel 3.6.1 Monadischer Kommando-Parser

```
number :: Parser Symbol Int
number = do sym <- sat item f "no number"; return (g sym)
         where f (Num _) = True
               f _ = False
               g (Num i) = i

ident :: Parser Symbol String
ident = do sym <- sat item f "no identifier"; return (g sym)
        where f (Ide _) = True
              f _ = False
              g (Ide x) = x
```

Beispiel 3.6.1 Monadischer Kommando-Parser

Zur Realisierung des oben erwähnten kombinierten Scanner+Parser wird im Typ `Result sym a` die Typvariable `sym` durch `Char` instanziiert, da er ja direkt Zeichenfolgen verarbeitet. Spezielle Zeichenparser dienen der Erkennung bestimmter Strings, von Zwischenräumen bzw. bestimmter Strings inklusive einschließender Zwischenräume:

```
string :: String -> Parser Char ()
string = foldr (>>) (return ()) . map char

space :: Parser Char ()
space = do many (sat item (‘elem‘ " \t\n") "no blank"); return ()

token :: Parser Char a -> Parser Char a
token p = do space; a <- p; space; return a

symbolC :: String -> Parser Char ()
symbolC = token . string
```

Beispiel 3.6.2 Monadischer Kommando-Scanner+Parser

An die Stelle der Objekte vom Typ `Symbol` (siehe Beispiel **??**) treten nun die Zeichenketten, aus denen sie jeweils hervorgehen.

```
monBlockC :: Parser Char [Command]
monBlockC = do symbolC "{"; cs <- many monComC; symbolC "}"; return cs

monComC :: Parser Char Command
monComC = tryseq "no command"
                 [do symbolC ";"; return Skip,
                  do x <- identC; symbolC "="; e <- monIntC monRestC
                     symbolC ";"; return (Assign x e),
                  do symbolC "if"; symbolC "("; be <- monBoolC; symbolC ")"
                     cs <- monBlockC
                     tryseq "" [do symbolC "else"; cs' <- monBlockC
                                   return (Cond be cs cs'),
                                return (Cond be cs [])],
                  do symbolC "while"; symbolC "("; be <- monBoolC; symbolC ")"
                     cs <- monBlockC; return (Loop be cs)]
```

Beispiel 3.6.2 Monadischer Kommando-Scanner+Parser

```
monIntC :: (IntE -> Parser Char IntE) -> Parser Char IntE
monIntC f = tryseq "no integer expression"
                   [do i <- numberC; f (IntE i),
                    do x <- identC; f (Var x),
                    do symbolC "("; e <- monIntC monRestC; symbolC ")"; f e]

monRestC :: IntE -> Parser Char IntE
monRestC e = tryseq "" [do symbolC "-"; monSubC e,
                        do symbolC "+"; monSumC [e],
                        do symbolC "*"; monProdC [e],
                        return e]

monSubC :: IntE -> Parser Char IntE
monSubC e = do e' <- monIntC monRestC; return (Sub e e')

monSumC :: [IntE] -> Parser Char IntE
monSumC es = do e <- monIntC return
                tryseq "" [do symbolC "+"; monSumC (es++[e]),
                           return (Sum (es++[e]))]
```

Beispiel 3.6.2 Monadischer Kommando-Scanner+Parser

```
monProdC :: [IntE] -> Parser Char IntE
monProdC es = do e <- monIntC return
                 tryseq "" [do symbolC "*"; monProdC (es++[e]),
                               return (Prod (es++[e]))]

monBoolC :: Parser Char BoolE
monBoolC = tryseq "no Boolean expression"
                  [do symbolC "true"; return (BoolE True),
                   do symbolC "false"; return (BoolE False),
                   do symbolC "!"; be <- monBoolC; return (Not be),
                   do e <- monIntC monRestC; symbolC ">"
                      e' <- monIntC monRestC; return (Greater e e')]

numberC :: Parser Char Int
numberC = do ds <- token (some (sat item isDigit "no digit"))
             return (read ds::Int)

identC :: Parser Char String
identC = token (sat (some (sat item (not . isSpecial) "no identifier")) f
                    "no identifier")
         where f x = x `notElem` words "true false if else while"
```

Übersetzungsfunktion

- Entwicklung eines allgemeinen Schemas zum Entwurf einer Übersetzungsfunktion $comp : Q \to Z$
  - Voraussetzung: die Elemente von Q liegen als Syntaxbäume vor, also $Q = B(G)$

- $B(G)$ entspricht der Termalgebra, die sich aus der abstrakten Syntax $\Sigma(G)$ ergibt,
  $\Longrightarrow$ $comp$ entspricht der Auswertung(sfunktion) in einer geeigneten $\Sigma(G)$-Algebra mit Trägermenge Z

Übersetzungsfunktion

- Die Trägermenge einer $\Sigma(G)$-Algebra ist N-sortiert
  $\Longrightarrow$ $comp$ ist ebenfalls N-sortiert:
  $$comp = \{comp_s : B(G)_s \to Z_s \mid s \in N\}.$$

- $B(G)_s$ besteht aus den Syntaxbäumen, die aus s in G ableitbaren Wörtern (Quellprogrammen) entsprechen:
  $$B(G)_s = \{t \in B(G) \mid \exists\, w \in T^* : s \xrightarrow{*}_G w\}.$$
  $\Longrightarrow$ Z_s ist die Menge der den aus s in G ableitbaren Quellprogrammen entsprechenden Zielprogramme

Attribuierte Übersetzung

- $comp_s(t)$ hängt nicht nur von t ab, sondern auch von semantischen Informationen, die aus Teilen von t **abgeleitet** und dann an andere Teile von t **vererbt** werden
- Anstelle von Z_s wird also eine Struktur der Form $V_s \to (A_s \times Z_s)$ benötigt, womit der Typ von $comp_s$ die folgende Form erhöht:

$$comp_s : B(G)_s \to (V_s \to (A_s \times Z_s)).$$

  - V_s Und A_s bezeichnen die Mengen von Daten, die zur Berechnung der Zielprogramme von Z_s vererbt bzw. abgeleitet werden mssen.
  - V_s und A_s sind also in der Regel kartesische Produkte. Attribute sind dann nichts anderes als die Indizes dieser Produkte.

Definition von $comp_s$

Sei $\Sigma(G) = (N, F)$ die abstrakte Syntax von G.
$comp_s$ lässt sich nun in der Form eines Systems $\{e_p\}_{p \in F}$ rekursiver Gleichungen definieren, wobei e_p folgende Form hat:

$$
\begin{aligned}
comp_s(p(x_1, \ldots, x_n))(a_0) = e_n(a_0, \ldots, a_n) \\
\text{where } a_1 &= comp_{s_1}(x_1)(e_0(a_0)) \\
a_2 &= comp_{s_2}(x_2)(e_1(a_0, a_1)) \\
&\vdots \\
a_n &= comp_{s_n}(x_n)(e_{n-1}(a_0, \ldots, a_{n-1}))
\end{aligned}
$$

Definition von $comp_s$

Alternativ kann die N-sortierte Menge B mit $B_s =_{def} V_s \to A_s$, $s \in N$ zur $\Sigma(G)$-Algebra erweitert werden, deren Auswertungsfunktion $eval^B$ mit $comp$ bereinstimmt.

Die Definition der Interpretation

$$p^B : B_{s_1} \times \ldots \times B_{s_n} \to B_s$$

folgt dann einem analogen Schema:

$$\begin{aligned} p^B(x_1, \ldots, x_n)(a_0) = e_n(a_0, \ldots, a_n) \\ \text{where } a_1 &= x_1(e_0(a_0)) \\ a_2 &= x_2(e_1(a_0, a_1)) \\ &\vdots \\ a_n &= x_n(e_{n-1}(a_0, \ldots, a_{n-1})) \end{aligned}$$

Beispiel 4.1.1

Grammatik G für linearisierte Textdarstellungen

konkrete Syntax G

TEXT	$\longrightarrow$	STRING
STRING	$\longrightarrow$	STRING BOX
STRING	$\longrightarrow$	STRING$\uparrow$BOX
STRING	$\longrightarrow$	STRING$\downarrow$BOX
STRING	$\longrightarrow$	ε
BOX	$\longrightarrow$	(STRING)
BOX	$\longrightarrow$	$a \mid b \mid c$

abstrakte Syntax $\Sigma(G)$

start:	STRING	$\longrightarrow$	TEXT
app:	STRING$\times$BOX	$\longrightarrow$	STRING
up:	STRING$\times$BOX	$\longrightarrow$	STRING
down:	STRING$\times$ BOX	$\longrightarrow$	STRING
empty:		$\longrightarrow$	STRING
make_box:	STRING	$\longrightarrow$	BOX
a,b,c:		$\longrightarrow$	BOX

Beispiel 4.1.1

Übersetzungsfunktionen

$comp_{TEXT} : B(G)_{TEXT} \to Z =$ Texte mit Hoch- und Tiefstellungen
$comp_{STRING} : B(G)_{STRING} \longrightarrow (\mathbb{N}^2 \to (\mathbb{N}^3 \times Z))$
$comp_{BOX} : B(G)_{BOX} \longrightarrow (\mathbb{N}^2 \to (\mathbb{N}^3 \times Z))$

Das vererbte zweistellige Attribut liefert die Koordinaten der linken unteren Ecke des Rechtecks, in das der eingelesene String geschrieben werden soll. Das abgeleitete dreistellige Attribut liefert die Länge sowie die Höhe und die Tiefe des Rechtecks.

Beispiel 4.1.1

Übersetzungsfunktionen

$$comp_{TEXT}(\mathsf{start}(s)) = z \quad \text{where } (l,h,t,z) = comp_{STRING}(s,0,0)$$

$$comp_{STRING}(\mathsf{app}(s,b))(x,y) = (l+l', \mathsf{max}(h,h'), \mathsf{max}(t,t'), zz') \quad \text{where } (l,h,t,z) = comp_{STRING}(s)(x,y),\ (l',h',t',z') = comp_{BOX}(b)(x+l,y)$$

$$comp_{STRING}(\mathsf{up}(s,b))(x,y) = (l+l', h+h'-1, \mathsf{max}(t,t'-h+1), z^{z'}) \quad \text{where } (l,h,t,z) = comp_{STRING}(s)(x,y),\ (l',h',t',z') = comp_{BOX}(b)(x+l,y+h-1)$$

Beispiel 4.1.1

Übersetzungsfunktionen

$comp_{TEXT}(\mathsf{start}(s)) = z$
$comp_{STRING}(\mathsf{down}(s,b))(x,y) = (l+l', \mathsf{max}(h, h'-t-1), t+t'+1, z_{z'})$
where $(l,h,t,z) = comp_{STRING}(s)(x,y)$
$(l',h',t',z') = comp_{BOX}(b)(x+l, y-t-1)$

$comp_{STRING}(\mathsf{empty})(x,y) = (0,0,0,\sqcup)$
$comp_{BOX}(\mathsf{make\_box}(s))(x,y) = comp_{STRING}(s)(x,y)$
$comp_{BOX}(a)(x,y) = (1,2,0,a)$

Positionsangaben, Ziffernstellen, Adressen, usw. sind meistens vererbte Attribute. Typangaben, Platzbedarf, Gewichte, Werte und Zielprogramme hingegen werden abgeleitet.

Beispiel 4.1.2

Grammatik G für binäre rationale Zahlen

konkrete Syntax G	**abstrakte Syntax $\Sigma(G)$**
$RAT \to NAT.$	$make\_rat : NAT \to RAT$
$RAT \to RAT0$	$app\_0 : RAT \to RAT$
$RAT \to RAT1$	$app\_1 : RAT \to RAT$
$NAT \to 0$	$0 :\to NAT$
$NAT \to 1$	$1 :\to NAT$
$NAT \to NAT0$	$app\_0 : NAT \to NAT$
$NAT \to NAT1$	$app\_1 : NAT \to NAT$

Beispiel 4.1.2

Übersetzungsfunktionen

$$comp_{RAT} : B(G)_{RAT} \to \mathbb{Q} \times \mathbb{Q}$$
$$comp_{NAT} : B(G)_{NAT} \to \mathbb{N}$$

Die abgeleiteten RAT-Attribut haben folgende Bedeutung: Sei $x.y$ eine binäre rationale Zahl in konkreter Syntax.

$$comp_{RAT}(x.y) = (r, s) \Leftrightarrow \begin{cases} \text{Dezimalwert von } x.y \\ s = 2^{-|y|} \end{cases}$$

Beispiel 4.1.2

Übersetzungsfunktionen

$comp_{RAT}(make\_rat(x)) = (n, 1)$ where $n = comp_{NAT}(x)$
$comp_{RAT}(app\_0(x)) = (r, s/2)$ where $(r, s) = comp_{RAT}(x)$
$comp_{RAT}(app\_1(x)) = (r + s/2, s/2)$ where $(r, s) = comp_{RAT}(x)$

$comp_{NAT}(0) = 0$
$comp_{NAT}(1) = 1$
$comp_{NAT}(app\_0(x)) = 2 * n$ where $n = comp_{NAT}(x)$
$comp_{NAT}(app\_1(x)) = 2 * n + 1$ where $n = comp_{NAT}(x)$

Beispiel 4.1.3

Übersetzung regulärer Ausdrücke in endliche Automaten

Die Übersetzungsfunktion hat also folgenden Typ:

```
regToAuto :: Eq a => RegExp a -> IntAuto a
```

Sie verwendet die Hilfsfunktion

```
autoRec :: Eq a => RegExp a -> Int -> Int
                -> (Int -> Ext a -> [Int]) -> Int
                -> (Int -> Ext a -> [Int], Int)
```

Beispiel 4.1.3

```
regToAuto e = ((delta,beta,0),[0..nextq-1],symbols e)
              where (delta,nextq) = autoRec e 0 1 (const (const [])) 2
                    beta q = if q == 1 then [True] else [False]
                    symbols (Const a)  = [a]
                    symbols (Seq e e') = symbols e `join` symbols e'
                    symbols (Par e e') = symbols e `join` symbols e'
                    symbols (Plus e)   = symbols e
                    symbols _          = []
```

Beispiel 4.1.3

```
autoRec (Const a) q q' delta nextq  = (upd2 delta q (Def a) q',nextq)
autoRec Eps q q' delta nextq        = (upd2 delta q Epsilon q',nextq)
autoRec Empty _ _ delta nextq       = (delta,nextq)
autoRec (Seq e e') q q' delta nextq = autoRec e' nextq q' delta' nextq'
                        where (delta',nextq') = autoRec e q
                                                nextq delta (nextq+1)
autoRec (Par e e') q q' delta nextq = autoRec e' q q' delta' nextq'
                        where (delta',nextq') = autoRec e q q' delta nextq
autoRec (Plus e) q q' delta nextq   = (upd2 delta3 q1 Epsilon q',nextq1)
                        where q1 = nextq+1
                              (delta1,nextq1) = autoRec e nextq q1 delta (q1+1)
                              delta2 = upd2 delta1 q Epsilon nextq
                              delta3 = upd2 delta2 q1 Epsilon nextq

upd2 :: (Eq a,Eq b) => (a -> b -> [c]) -> a -> b -> c -> (a -> b -> [c])
upd2 f a b c a' b' = if a == a' && b == b' then c:cs else cs where cs = f a'
```

Mehrpässige Übersetzung

Da v_i und a_k i.a. aus mehreren Komponenten bestehen, kann es zu wechselseitigen Abhängigkeiten kommen. So können z.B. der Wert von v_{i_3} für die Berechnung von a_{k_1} und der Wert von a_{k_2} für die Berechnung von v_{i_4} notwendig sein.

Dann ist es nicht möglich, alle Komponenten von v_i und a_k gleichzeitig zu berechnen und die Übersetzung muss in mehreren Pässen (= Traversierungen des Syntaxbaumes) durchgeführt werden.

- ☞ $comp_s$ muß entsprechend einer Zerlegung der Attributmenge in eine Folge von Funktionen $comp_s^1, \ldots, comp_s^r$ aufgeteilt werden, die sequentiell auf demselben Syntaxbaum, aber verschiedenen Attributmengen ausgeführt werden.

Mehrpässige Übersetzung

Sei At die Menge aller Attribute. Jedes Attribut ist nichts weiter als ein *Index* der Produkte V_s und A_s vererbter bzw. abgeleiteter Attributwerte.

V_s und A_s sind von der Form $\prod_{at \in At'} Val_{at}$, wobei At' eine Teilmenge von At ist (nicht jedes Attribut wird vererbt bzw. abgeleitet) und Val_{at} die Menge der möglichen Werte von at ist.

Ein r-pässiger Compiler zerlegt At in Teilmengen $At^1, \ldots, At^r$, wobei At^k die Menge der im k-ten Pass berechneten Attribute bezeichnet. Daraus ergeben sich r Projektionen $\pi^1, \ldots, \pi^r$ auf jeder Produktmenge, deren Komponenten mit Attributen indiziert sind (wie z.B. V_s und A_s).

Mehrpässige Übersetzung

Für alle $1 \leq k \leq r$ ist

$$\pi^k : \prod_{at \in At} Val_{at} \to \prod_{at \in At^k} Val_{at}$$

definiert durch:

$$\pi^k((x_{at})_{at \in At}) \;=\; (x_{at})_{at \in At^k}.$$

Für die Bildmenge $\pi^k(\prod_{at \in At} Val_{at})$ schreiben wir kurz: $(\prod_{at \in At} Val_{at})^k$. Die eine Übersetzungsfunktion

$$comp_s : B(G)_s \quad \longrightarrow \quad (V_s \to A_s)$$

aus Abschnitt 4.1 wird nun in r Funktionen

$$comp_s^k : B(G)_s \quad \longrightarrow \quad (V_s^k \to (B(A)_s^{k-1} \to B(A)_s^k)),$$

$1 \leq k \leq r$, zerlegt. Hierbei ist $B(A)^k$ eine (N-sortierte) Menge von **Attributbäumen**.

Mehrpässige Übersetzung

$comp_s$ verknüpft $comp_s^1, \ldots, comp_s^r$ wie folgt:

$$\begin{aligned} comp_s(t)(a_0) = root(b_r) \\ \text{where } b^1 &= comp_s^1(t)(\pi^1(a_0)) \\ b^2 &= comp_s^2(t)(\pi^2(a_0))(b^1) \\ &\vdots \\ b^r &= comp_s^r(t)(\pi^r(a_0))(b^{r-1}) \end{aligned}$$

Hierbei ist a^i ein Attributbaum von $B(A)_s^k$.

Da $comp_s^1, \ldots, comp_s^r$ induktiv über $B(G)$ definiert werden, führt ein Aufruf von $comp_s$ zu r Traversierungen des Syntaxbaums. Die Übersetzungszeit hängt also wesentlich von der Anzahl r der Pässe ab.
Zur Bestimmung der minimalen Anzahl gehen wir vom allgemeinen Definitionsschema aus, wo aber jetzt die Variablen $a_0, \ldots, a_n$ beliebig in $e_0, \ldots, e_n$ vorkommen können.

Mehrpässige Übersetzung

Produkte werden mit vorgegebener Indexmenge (= Attributmenge) üblicherweise durch **Recordtypen** implementiert. Die einzelnen Attribute entsprechen dort verschiedenen Feldnamen.

In Haskell hat ein Record mit den Feldnamen $at_1, \ldots, at_n$ die Form $\{at_1 = val_1, \ldots, at_n = val_n\}$, wobei val_i der Wert von at_i ist. Die entsprechende Verfeinerung von (4.1) mit Records lautet wie folgt:

$$
\begin{aligned}
&comp_s(p(t_1, \ldots, t_n))\{A_{01} = a_{01}, \ldots, A_{0k_0} = a_{0k_0}\} = e_n \\
&\qquad \text{where } e_0 = \{D_{01} = e_{01}, \ldots, D_{0l_0} = e_{0l_0}\} \\
&\qquad\qquad \{A_{11} = a_{11}, \ldots, A_{1k_1} = a_{1k_1}\} = comp_{s_1}(t_1)(e_0) \\
&\qquad\qquad \vdots \\
&\qquad\qquad e_{n-1} = \{D_{(n-1)1} = e_{(n-1)1}, \ldots, D_{(n-1)l_{n-1}} = e_{(n-1)l_{n-1}}\} \\
&\qquad\qquad \{A_{n1} = a_{n1}, \ldots, A_{nk_n} = a_{nk_n}\} = comp_{s_n}(t_n)(e_{n-1}) \\
&\qquad\qquad e_n = \{D_{n1} = e_{n1}, \ldots, D_{nl_n} = e_{nl_n}\}
\end{aligned}
$$

Mehrpässige Übersetzung

Daraus ergeben sich die Mengen At_{s_i}, $0 \leq i \leq n$, der **für** $s_0 = s$ **bzw.** $s_1, \ldots, s_n$ **relevanten Attribute**:

$$At_{s_i} =_{def} \{A_{ij} \mid 0 \leq j \leq k_i\} \cup \{D_{ij} \mid 0 \leq j \leq l_i\}.$$

Es impliziert auch, dass $A_{01}, \ldots, A_{0k_0}$ sowie $D_{i1}, \ldots, D_{il_i}$,

$0 \leq i < n$, vererbte Attribute und $A_{i1}, \ldots, A_{ik_i}$, $1 \leq i \leq n$, sowie $D_{n1}, \ldots, D_{nl_n}$ abgeleitete Attribute sind.

Mehrpässige Übersetzung

In klassischen Darstellungen attributierter Grammatiken werden Variablen für Attributwerte benutzt und durch Feldnamen der Form $s.at$ repräsentiert. Die Definitionen der D-Attribute werden als Zuweisungen an jene Variablen notiert und in die *konkrete* Syntax von G eingegefügt:

$$
\begin{aligned}
p \ : \ s \ \rightarrow \ & ...s_1...s_2... \ ...s_n \\
& \quad s_1.D_{01} := e'_{01}, \ \ldots, \ s_1.D_{0l_0} := e'_{0l_0} \\
& \quad \vdots \\
& \quad s_n.D_{(n-1)1} := e'_{(n-1)1}, \ \ldots, \ s_n.D_{(n-1)l_{n-1}} := e'_{(n-1)l_{n-1}} \\
& \quad s.D_{n1} := e'_{n1}, \ \ldots, \ s.D_{nl_n} := e'_{nl_n}
\end{aligned}
$$

Hierbei entsteht e'_{ij} aus e_{ij}, indem für alle $0 \leq k \leq n$ und $1 \leq m \leq k_i$ A_{km} durch $s_k.A_{km}$ ersetzt wird.

Abhängigkeitsgraph

Die o.g. Analyse von basiert auf dem **Abhängigkeitsgraphen (*dependency graph*) für** G,

$$DG : P \to (At \times At \to \wp(\mathbb{N} \times \mathbb{N})).$$

Er liefert für alle Regeln p von G und alle Attributpaare (Aik, D_{jl}) die Paare (i, j) mit der Eigenschaft, dass der Wert a_{ik} von Aik im Ausdruck e_{jl} vorkommt:

Sei $0 \leq i, j \leq n$, $1 \leq k \leq k_i$ und $1 \leq l \leq l_j$.

$$(i, j) \in DG(p)(A_{ik}, D_{jl}) \quad \Longleftrightarrow_{def} \quad a_{ik} \text{ kommt in } e_{jl} \text{ vor.}$$

Abhängigkeitsgraph

Die o.g. Formel ist genau dann ausführbar, wenn für alle $A, D \in At$ und $(i, j) \in DG(p)(A, D)$ $i \leq j$ gilt. Ist diese Bedingung nicht erfüllt, dann wird eine Zerlegung $\{At^1, \ldots, At^r\}$ von At gesucht derart, daß für alle $1 \leq k \leq r$, $A, D \in At$ und $(i, j) \in DG(p)(A, D)$ gilt:

$$D \in At^k \quad \Rightarrow \quad (\exists l < k : A \in At^l) \ \vee \ (A \in At^k \wedge i \leq j).$$

Erfüllen alle $1 \leq k \leq r$ diese Bedingung, dann lassen sich aus dem allgemeinen Definitionen und den Definitionen von $comp_s$ für andere Produktionen p mit linker Seite s Definitionen von $comp_s^1, \ldots, comp_s^r$ gewinnen.

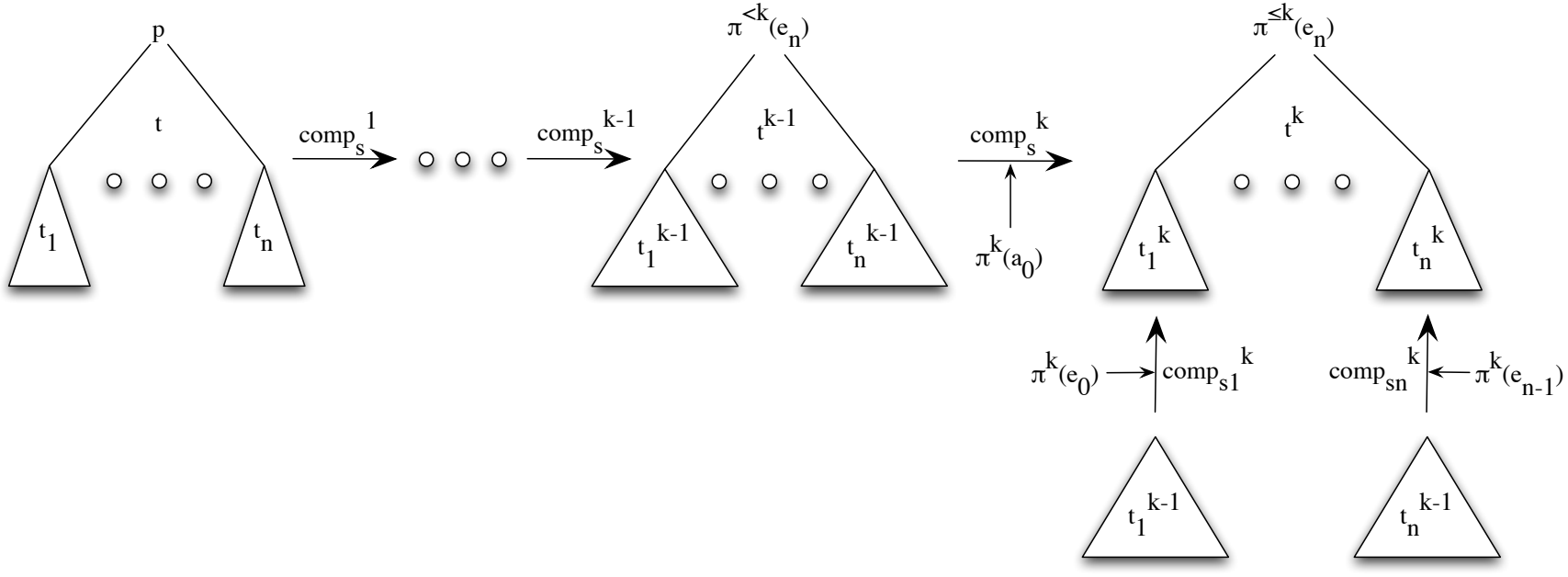
Abhängigkeitsgraph

Seien $1 \leq k \leq r$, $p : s_1 \times \ldots \times s_n \longrightarrow s$ ein Konstruktor von $\Sigma(G)$ und für alle $1 \leq i < k$ und $1 \leq j \leq n$, $b^i_j \in B(A)^i_{s_j}$ der j-te Unterbaum von b^i.

$$comp^k_s(p(t_1, \ldots, t_n))(\pi^k(a_0))(b^{k-1}) = \pi^{\leq k}(e_n)[b^k_1, \ldots, b^k_n]$$
$$\text{where } b^k_1 = comp^k_{s_1}(t_1)(\pi^k(e_0))(b^{k-1}_1)$$
$$\vdots$$
$$b^k_n = comp^k_{s_n}(t_n)(\pi^k(e_{n-1}))(b^{k-1}_n)$$

Der Ergebnisbaum $\pi^{\leq k}(e_n)[b^k_1, \ldots, b^k_n]$ des k-ten Passes hat den Wurzeleintrag $\pi^{\leq k}(e_n)$ (Werte aller bis zum k-ten Pass abgeleiteten Attribute) und die Unterbäume $b^k_1, \ldots, b^k_n$.

Schrittweise Attributierung eines Syntaxbaumes



Mehrpässige Übersetzung

Sei $s \in N$ und $1 \leq k \leq r$.

$$B_s^k =_{def} V_s^k \rightarrow (B(A)_s^{k-1} \rightarrow B(A)_s^k)).$$

Die Definition der Interpretation

$$p^{B^k} : B_{s_1}^k \times \ldots \times B_{s_n}^k \rightarrow B_s^k$$

von $p : s_1 \times \ldots \times s_n \rightarrow s \in F$ in B^k folgt dem o.g. Schema:

Mehrpässige Übersetzung

Für alle $1 \leq i \leq n$ sei $f_i : V_{s_i}^k \to A_{s_i}^k$.

$$p^{B^k}(g_1, \ldots, g_n)(\pi^k(a_0))(b^{k-1}) = \pi^{\leq k}(e_n)[b_1^k, \ldots, b_n^k]$$
$$\text{where } b_1^k = g_1(\pi^k(e_0))(b_1^{k-1})$$
$$\vdots$$
$$b_n^k = g_n(\pi^k(e_{n-1}))(b_n^{k-1})$$

$$p^B(g_1, \ldots, g_n)(a_0) = root(b^r)$$
$$\text{where } b^1 = p^{B^1}(g_1, \ldots, g_n)(\pi^1(a_0))$$
$$b^2 = p^{B^2}(g_1, \ldots, g_n)(\pi^2(a_0))(b^1))$$
$$\vdots$$
$$b^r = p^{B^r}(g_1, \ldots, g_n)(\pi^r(a_0))(b^{r-1})$$

Eine LAG(2)-Grammatik

```
data Z = P1 (A,B)
data A = P2
data B = P3

type     V_Z = ()
type     A_Z = {result::Int}
type     V_A = {a1::Int}
type     A_A = {a2::Int}
type     V_B = {b1::Int}
type     A_B = {b2::Int}
```

Also ist $At = \{result, a1, a2, b1, b2\}$. Der Abhängigkeitsgraph laute wie folgt:

```
DG(P1)("b2","a1")          = [(2,1)]
DG(P1)("a2","result")      = [(1,2)]
DG(P2)("a1","a2")          = [(0,0)]
DG(P3)("b1","b2")          = [(0,0)]
DG _ _                     = []
```

Eine LAG(2)-Grammatik

Kritisch ist die Abhängigkeit $(b2, a1)$.

- Das abgeleitete Attribut $b2$ wird zur Definition des vererbten Attributs $a1$ 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, genügen also zwei Pässe:

$$At^1 = \{b1, b2\} \text{und} At^2 = \{result, a2, a1\}.$$

Grammatik, die für kein r eine LAG(r)-Grammatik ist

```
data Z = P1 (D,E) | P2 (D,E)
data D = P3
data E = P4

type     V_Z = ()
type     A_Z = {sz::Int}
type     V_D = {id::Int}
type     A_D = {sd::Int}
type     V_E = {ie::Int}
type     A_E = {se::Int}
```

Also ist $At = \{sz, sd, id, se, ie\}$. Der Abhängigkeitsgraph laute wie folgt:

```
DG(P1)("se","id") = [(1,2)]
DG(P1)("sd","sz") = [(2,2)]
DG(P2)("sd","ie") = [(2,1)]
DG(P2)("se","sz") = [(1,2)]
DG(P3)("id","sd") = [(0,0)]
DG(P4)("ie","se") = [(0,0)]
DG _ _            = []
```

Grammatik, die für kein r eine LAG(r)-Grammatik ist

Kritisch ist hier die Abhängigkeit (sd, ie).

- Das abgeleitete Attribut sd wird dort zur Definition des vererbten Attributes ie benötigt.
- Wegen $2 > 1$ kann ie wieder erst im zweiten Pass berechnet werden. Nun greifen aber sz auf sd, sd auf id, id auf se und schließlich se auf das Attribut ie zu, das erst im zweiten Pass berechnet werden kann.
- Also kann im ersten Pass überhaupt kein Attribut berechnet werden, d.h. es gibt keine mit DG verträgliche Reihenfolge, in der die Attribute berechnet werden könnten.

Algorithmus zur Berechnung der kleinsten LAG-Zerlegung

```
least_partition ats = reverse . check_partition [ats] []

check_partition (curr:partition) next =
 if changed then check_partition (curr':partition) next'
 else if null next' then curr':partition
      else if null curr' then []
           else check_partition (next':curr':partition) []
 where ((curr',next'),changed) = check_prods (curr,next) prods

check_prods cn (p:ps) = (cn2,changed1 || changed2)
 where f = depGraph p
       atps = [(x,y) | x <- ats, y <- ats, not (null f (x,y))]
       (cn1,changed1) = check_atps cn f atps
       (cn2,changed2) = check_prods cn1 ps
check_prods cn _      = (cn,False)
```

Algorithmus zur Berechnung der kleinsten LAG-Zerlegung

```
check_atps cn f (atp:atps) = (cn2,changed1 || changed2)
  where (cn1,changed1) = check_deps cn atp (f atp)
        (cn2,changed2) = check_atps cn1 f atps
check_atps cn _ _        = (cn,False)

check_deps cn@(curr,next) (a,d) ((i,j):deps) = (cn2,changed1 || changed2)
  where (cn1,changed1) = if d `elem` curr &&
                            (a `elem` next || (a `elem` curr && j<i))
                         then ((filter (/= d) curr,d:next),True)
                         else (cn,False)
        (cn2,changed2) = check_deps cn1 (a,d) deps
check_deps cn _ _ _      = (cn,False)
```

Coderzeugung für eine imperative Programmiersprache

Jetzt soll ein Übersetzer für eine imperative Programmiersprache im einzelnen definiert werden.
Wir beginnen mit den Ausdrücken einer imperativen Sprache:

konkrete Syntax			**abstrakte Syntax**	
exp	$\longrightarrow$	ident(exps)	APPLY	ident $\times$ exps $\longrightarrow$ exp
exp	$\longrightarrow$	int	CI	int $\longrightarrow$ exp
exp	$\longrightarrow$	boolean	CB	boolean $\longrightarrow$ exp
exp	$\longrightarrow$	ident	ID	ident $\longrightarrow$ exp
exp	$\longrightarrow$	**fun** ident	FID	ident $\longrightarrow$ exp
exp	$\longrightarrow$	exp$^\wedge$	deref	exp $\longrightarrow$ exp
exp	$\longrightarrow$	exp[exp]	AF	exp $\times$ exp $\longrightarrow$ exp
exp	$\longrightarrow$	exp.ident	RF	exp $\times$ ident $\longrightarrow$ exp
exp	$\longrightarrow$	$\neg$exp	NOT	exp $\longrightarrow$ exp
exp	$\longrightarrow$	exp $\leq$ exp	LEQ	exp $\times$ exp $\longrightarrow$ exp
exp	$\longrightarrow$	exp + exp	ADD	exp $\times$ exp $\longrightarrow$ exp
exp	$\longrightarrow$	exp - exp	SUB	exp $\times$ exp $\longrightarrow$ exp
exp	$\longrightarrow$	exp * exp	MUL	exp $\times$ exp $\longrightarrow$ exp
exps	$\longrightarrow$	exp, exps	_:_	exp $\times$ exps $\longrightarrow$ exps
exps	$\longrightarrow$	exp	[_]	exp $\longrightarrow$ exps

Symboltabelle

Das wichtigste Attribut bei der Übersetzung einer imperativen Sprache ist die **Symboltabelle**. Wir stellen sie dar als Liste von Quadrupeln

$$q = (\text{id}, \text{td}, \text{adr}, \text{depth}).$$

q ordnet dem Identifier id einen **Typdeskriptor** td, eine **relative Adresse** adr und die **Schachtelungstiefe** depth seiner Deklaration zu.

Zugriff auf die Symboltabelle

get sucht für einen Identifier id in der Symboltabelle nach dem ersten q der Form (id,td,adr,depth).
Die Symboltabelle wird demnach als Keller verwaltet: get liefert den jeweils letzten Eintrag für id.

```
get ((id,td,adr,depth):st) id'
        = if id == id' then (td,adr,depth) else get st id'
get _ _ = error "declaration missing"
```

Datentyp für Typdeskriptoren

Für Typdeskriptoren führen wir einen Datentyp ein, der den Typen der Quellsprache Konstruktoren zuordnet:

```
data TypeDescr = INT | BOOL | Pointer TypeDescr |
                 Func Int TypeDescr | Formalfunc TypeDescr |
                 Name String | Array Int Int TypeDescr |
                 Record [(String,TypeDescr,Int,Int)]
```

- 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*.

Datentyp für Typdeskriptoren

- *Name* liefert den Deskriptor neu deklarierter Typen. Das Stringargument von *Name* ist der jeweilige Typidentifier id.
- Die Argumente des Feldtypkonstruktors *Array* sind die untere Schranke, die Länge und der Elementtypdeskriptor des jeweiligen Feldes.
- Das Argument von *Record* ist eine ganze Symboltabelle, die den Attributen des jeweiligen Records deren Typdeskriptoren und relative Adressen zuordnet.

Die Funktion **Offset**

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 id) st                    = offset td st
                                        where (td,_,_) = get st id

 offset _ _                             = 1
```

substType und CompExp

Wir stellen noch eine Funktion *substType* zur Verfügung, die Typidentifier "expandiert":

```
substType (Name id) st = td where (td,_,_) = get st id
substType td _         = td
```

$compExp$ übersetzt einzelne Ausdrücke, $compExps$ übersetzt Listen von Ausdrücken.

$compExp : B(G)_{exp} \to (\text{Symboltabelle} \times \mathbb{Z} \times \mathbb{Z} \to (\text{[Command]} \times \text{Typdeskriptor}))$
$compExps : B(G)^*_{exp} \to (\text{Symboltabelle} \times \mathbb{Z} \times \mathbb{Z} \to (\text{[Command]} \times F))$

substType und CompExp

$compExp$ und $compExps$ haben die gleichen vererbten Attribute:

- die aktuelle Symboltabelle,
- die Schachtelungstiefe des aktuellen **Scopes**, das ist der innerste Block, in dem der Ausdruck auftritt,
- die nächste freie Befehlsnummer.

Das Attribut *nächste freie Befehlsnummer* bezieht sich auf die Zielsprache. Es soll sich um eine einfache Assemblersprache handeln, deren Programme Listen numerierter Befehle sind.

Target Language

```
                        ----- TARGET LANGUAGE -----

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
```

Target Language

```
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
```

Zielmaschine

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)]
```

Zielmaschine

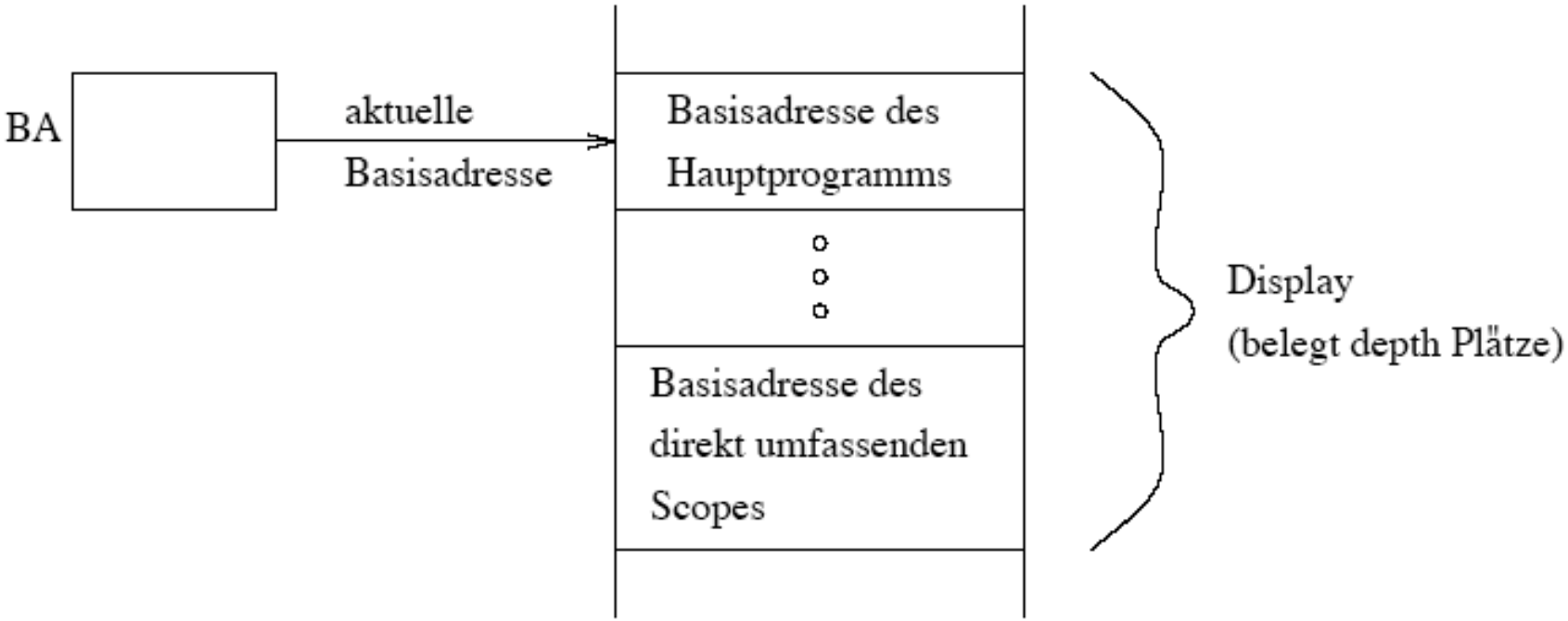
```
readL n r = foldl f [READ (IND r)] [1..n]
            where f code i = code++[READ (DEX i r)]

writeL n = replicate n (WRITE (ADR (IND TOP)))

derefpointer(offset) = MOV (ADR (IND TOP))
                           (REG ACC):pop:loadL (offset-1) ACC

baseAddress declDepth depth = if declDepth == depth
                              then ADR (REG BA)
                              else ADR (DEX declDepth BA)
```

Kellerauszug mit Display



Abstrakte Syntax der Quellsprache

Als Haskell-Datentyp aufgeschrieben, lautet die abstrakte Syntax der Ausdrücke unserer Quellsprache wie folgt:

```
data exp = APPLY String [Exp] | CI Int | CB Bool |
           ID String | FID String | Deref Exp |
           AF Exp Exp | RF Exp String | NOT Exp | LEQ Exp Exp |
           Add Exp Exp | Sub Exp Exp | Mul Exp Exp
```

Konstanten

```
compExp (CI i) _ _ _ = (code,INT)
                       where code = [push, MOV (CON i) (IND TOP)]

compExp (CB b) _ _ _ = (code,BOOL)
                       where code = [push, MOV (CON i) (IND TOP)]
                             i = if b then 1 else 0
```

`push` erhöht den Stacktop. `MOV` kellert die Konstanten.

Applikationen einfacher Identifier

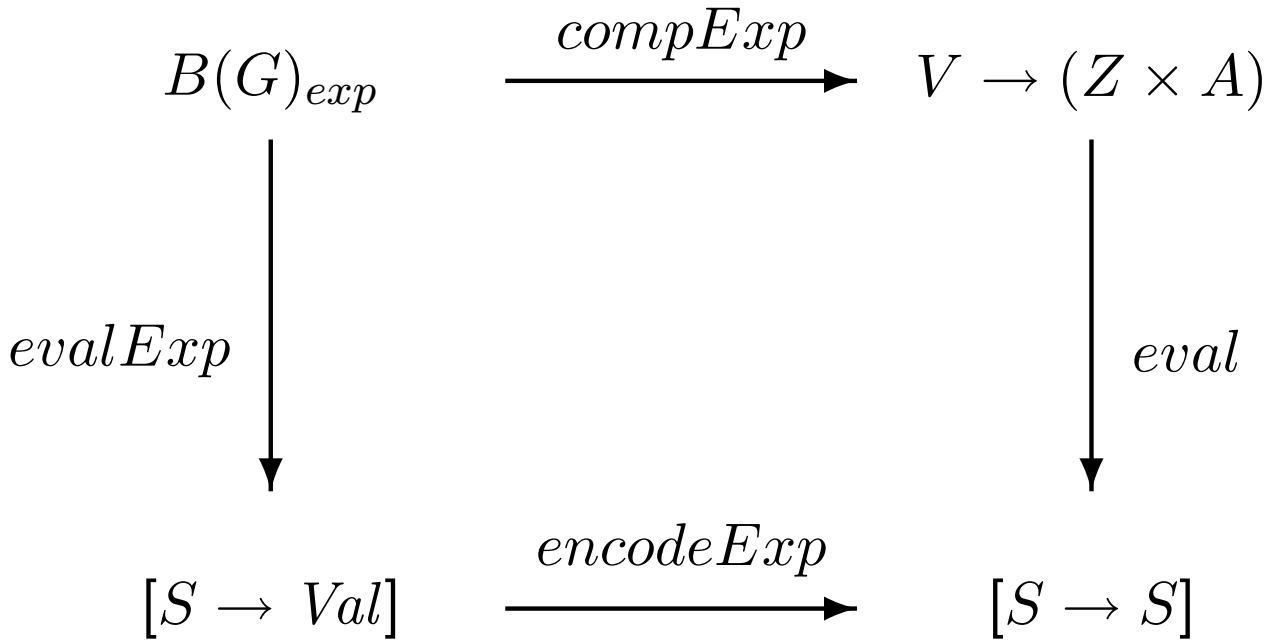
```
compExp (ID id) st depth _ = (code,Pointer td)
                where (td,adr,declDepth) = get st id
                      ba = baseAddress declDepth depth
                      code = [push, MOV ba (IND TOP),
                              ADD (IND TOP) (CON adr)]
```

- In der Symboltabelle st steht unter id die Relativadresse adr und die Schachtelungstiefe declDepth des Scopes der Deklaration von id.
- Aus declDepth und der Tiefe depth des aktuellen Scopes wird die Basisadresse ba des Gültigkeitsbereiches von id berechnet.
- ba ist Parameter des Zielcodes, der daraus durch das Kellern von ba und das Addieren der Relativadresse adr die absolute Adresse von id berechnet und kellert.

Der Zielcode kellert also die Adresse, nicht den Wert von id. Deshalb liefert $compExp$ den Deskriptor des Typs von Zeigern auf Objekte des Typs von id.

compExp und evalExp

Ein mit $compExp$ verträglicher Interpreter $evalExp$ sieht dann so aus:
Seien Val die Menge der speicherbaren Werte, $S = [Adr \to Val]$ die Menge der Speicherinhalte, $Z = [Command]$ und $eval : V \to (Z \times A)$ eine adäquate Fortsetzung des Interpreters von Z auf den gesamten Bildbereich von $compExp$.



compExp und evalExp

$compExp$ wird im Folgenden so definiert, daß für alle $e \in B(G)_{exp}$, $f \in V \to (Z \times A)$, $s \in S$ und $x \in Adr$ gilt:

$$compExp(e) = f \overset{(*)}{\Rightarrow} eval(f)(s)(x) = \begin{cases} evalExp(e)(s) & \text{falls } x = IND(TOP) \\ s(x) & \text{sonst.} \end{cases}$$

Definiert man $encodeExp$ für alle $f \in [S \to Val]$ durch

$$encodeExp(f)(s)(x) = \begin{cases} f(s) & \text{falls } x = IND(TOP) \\ s(x) & \text{sonst,} \end{cases}$$

dann entspricht die Gültigkeit von $(*)$ der Kommutativität des obigen Diagramms.

Applikationen von Funktionsidentifiern

Der Ausdruck `(FID fid)` kommt nur als aktueller Parameter einer Funktion vor.

```
compExp (FID fid) st depth lab = (code,Func codeadr td)
                 where code = [push, MOV ba (IND TOP),
                               push, MOV (CON codeadr) (IND TOP),
                               push, MOV ba (IND TOP),
                                     ADD (IND TOP) (CON resadr)]
                       ba = baseAddress declDepth depth
                       (Func codeadr td,resadr,declDepth) = get st fid
```

Bei der Ausführung des Zielcodes werden drei Adressen gespeichert:

- `ba` = Basisadresse des **statischen Vorgängers** von fid
- `codeadr` = Codeadresse von fid,
- `ba + resadr` = absolute Adresse des Resultates eines Aufrufs von fid.

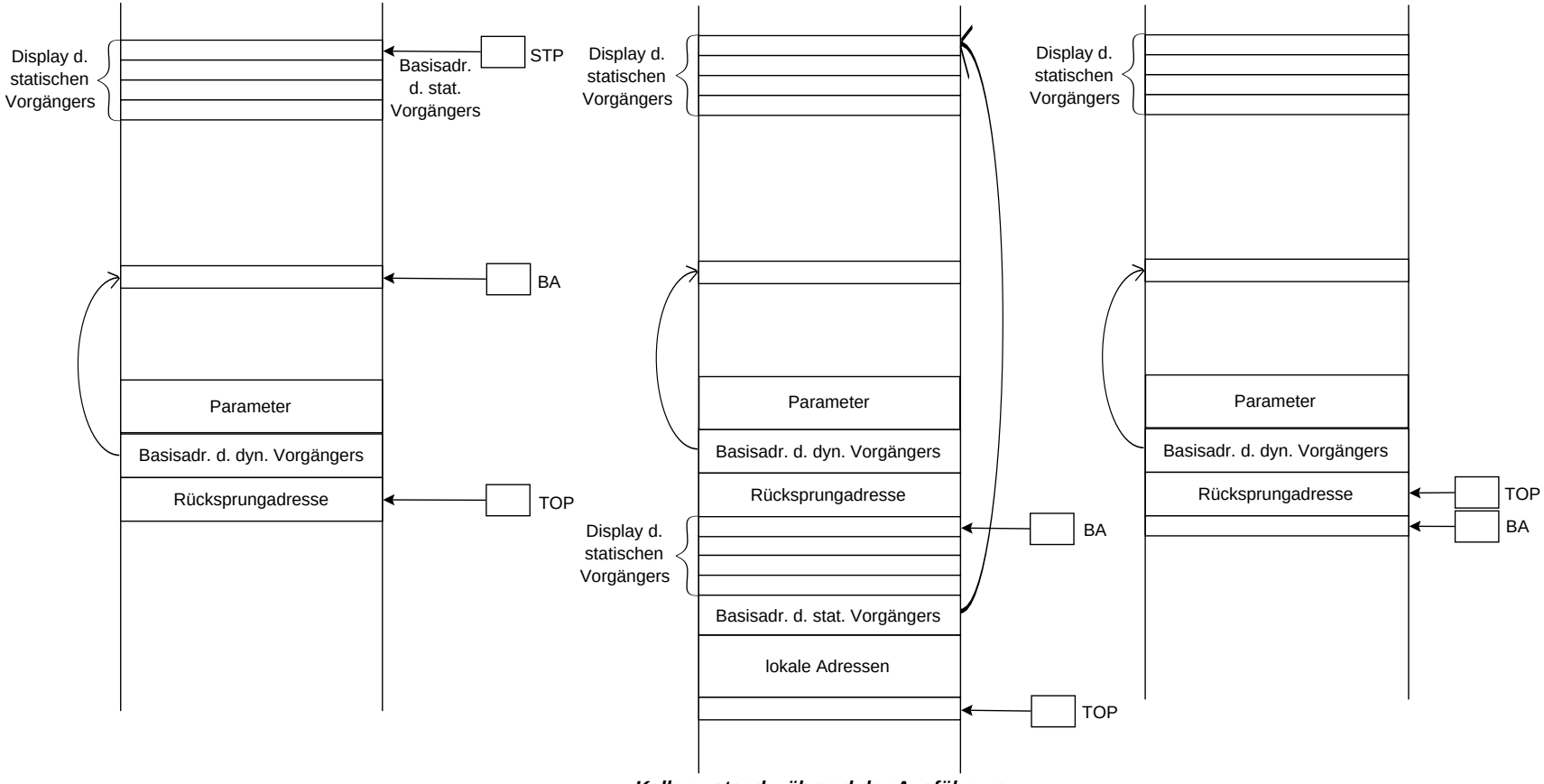
Der Typdeskriptor eines Ausdrucks bestimmt die Anzahl der Kellerplätze, die für seinen Wert reserviert werden.

Funktionsaufrufe

```
compExp (APPLY fid es) st depth lab
       = case ftd of Func codeadr td
                      ->  (code++applyFunc codeadr paroffset resadr ba lab',
                           Pointer td)
                     Formalfunc td
                      ->  (code++applyFormalfunc paroffset resadr ba lab',
                           Pointer td))
         where (code,paroffset) = compExps(es) st depth lab
               (ftd,resadr,declDepth) = get st fid
               ba = baseAddress declDepth depth
               lab' = lab+length code
```

Es werden zwei Fälle unterschieden. Im ersten hat fid den Typdeskriptor `Func codeadr td`, im zweiten Fall den Typdeskriptor `Formalfunc td`.

Auf- und Abbau des Kellers bei der Ausführung eines Funktionsaufrufs



Kellerzustand beim Sprung zur Codeadresse

Kellerzustand während der Ausführung des Funktionsrumpfes

Kellerzustand beim Rücksprung

1. Fall

fid hat den Typdeskriptor `Func codeadr td`. Dann liefert `applyFunc` den restlichen Zielcode:

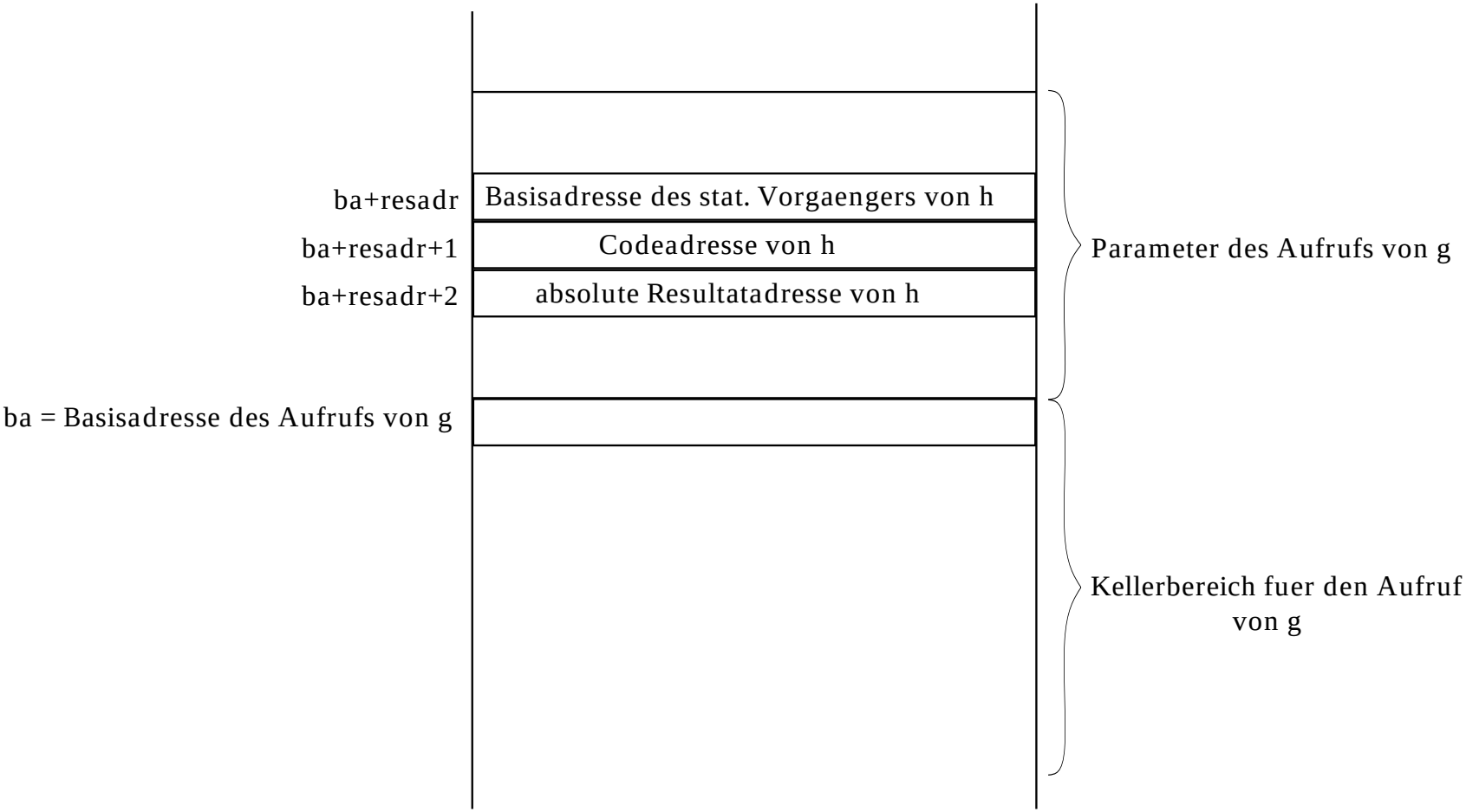
```
applyFunc codeadr paroffset resadr ba lab
       = push :
         MOV (ADR (REG BA)) (IND TOP) :
         MOV ba (REG STP) :
         push :
         MOV (CON retadr) (IND TOP) :
         GOTO (CON codeadr) :
retadr:  pop :
         MOV (ADR (IND TOP)) (REG BA) :
         popL paroffset ++
         [MOV ba (IND TOP),
         ADD (IND TOP) (CON resadr)]
        where retadr = lab+6
```

2. Fall

fid hat den Typdeskriptor `Formalfunc td`. Dann liefert `applyFormalfunc` den restlichen Zielcode:

```
applyFormalfunc(paroffset,resadr,ba,lab)
        = push :
          MOV (ADR (REG BA)) (IND TOP) :
          MOV ba (REG ACC) :
          MOV (ADR (DEX resadr ACC)) (REG STP) :
          push :
          MOV (CON retadr) (IND TOP) :
          GOTO (ADR (DEX (resadr+1) ACC)) :
retadr:   pop :
          MOV (ADR (IND TOP)) (REG BA) :
          popL paroffset ++
          [MOV ba (REG ACC),
           MOV (ADR (DEX (resadr+2) ACC)) (IND TOP)]
          where retadr = lab+7
```

Zur Adressrechnung bei der Übersetzung von fid(es)



Boolesche Funktionen

```
compExp (NOT e) st depth lab = (code',BOOL)
                 where (code,primitive BOOL) = compExp (deref e) st depth lab
                       code' = code++[INV (IND TOP)]


compExp (LEQ e e') st depth lab = (code",BOOL)
                 where (code,_) = compExp (deref e) st depth lab
                       lab' = lab + length code
                       (code',_) = compExp (deref e') st depth lab'
                       code" = code++code'++
                               [pop,
                                LE (ADR (IND TOP)) (ADR (DEX 1 TOP)),
                                MOV (ADR (REG BIT)) (IND TOP)]
```

Boolesche Funktionen

Zeigerargumente (z.B. Objektidentifier; s.o.) werden vor Anwendung einer Booleschen oder arithmetischen Funktion dereferenziert:

```
compExp (deref e) st depth lab
       = case td of Pointer td -> let offset = offset td st
                                  in (code++derefpointer offset,td)
                    _ -> (code,td)
         where (code,td) = compExp e st depth lab
```

Operationen auf ganzen Zahlen

Sei $(Op, OP) \in \{(Add, ADD), (Sub, SUB), (Mul, MUL)\}$.

```
compExp (Op e e') st depth lab = (code",INT))
            where code" = code++code'++
                          [pop,OP (IND TOP) (ADR (DEX 1 TOP))]
                  (code,primitive INT) = compExp (deref e)
                                                 st depth lab
                  lab' = lab + length code
                  (code',primitive INT) = compExp (deref e')
                                                  st depth lab'
```

Feldzugriffe e[e']

```
compExp (AF e e') st depth lab = (code",Pointer td')
      where code" = code++code'++[SUB (IND TOP) (CON lwb),
                                  MUL (IND TOP) (CON offset),
                                  pop,
                                  ADD (IND TOP) (ADR (DEX 1 TOP))]
                    (code,Pointer td) = compExp e st depth lab
                    Array lwb _ td' = substType td st
                    offset = offset td' st
                    lab' = lab + length code
                    (code',primitive INT) = compExp (deref e')
                                                    st depth lab'
```

Feldzugriffe `e.id`

```
compExp (RF e id) st depth lab = (code',Pointer td)
                  where code' = code++[ADD (IND TOP) (CON adr)]
                        (code,Pointer td) = compExp e st depth lab
                        Record st' = substType td st
                        (td,adr,_) = get st' id
```

Aktuelle Parameter

```
compExps [] _ _ _ = ([],0)

compExps (e:es) st depth lab = (code++code',offsets+offset)
                where (code,offsets) = compExps es st depth lab
                      lab' = lab + length code
                      (code',td) = compExp (deref e) st depth lab'
                      offset = offset td st
```

Übersetzung von Anweisungen

konkrete Syntax		
stat	$\longrightarrow$	ε
stat	$\longrightarrow$	exp := exp
stat	$\longrightarrow$	**read** ident
stat	$\longrightarrow$	**write** exp
stat	$\longrightarrow$	stat; stat
stat	$\longrightarrow$	**if** exp **then** stat **else** stat **fi**
stat	$\longrightarrow$	**while** exp **do** stat **od**
stat	$\longrightarrow$	**type** ident = type
stat	$\longrightarrow$	**var** ident: type
stat	$\longrightarrow$	**new** ident
stat	$\longrightarrow$	**begin** stat **end**
stat	$\longrightarrow$	**fun** ident(pars):type **is** stat **end**

abstrakte Syntax	
SKIP	$\longrightarrow$ stat
ASSIGN	exp $\times$ exp $\longrightarrow$ stat
read	ident $\longrightarrow$ stat
write	exp $\longrightarrow$ stat
SEQ	stat $\times$ stat $\longrightarrow$ stat
COND	exp $\times$ stat $\times$ stat $\longrightarrow$ stat
LOOP	exp $\times$ stat $\longrightarrow$ stat
TYPVAR	ident $\times$ type $\longrightarrow$ stat
VAR	ident $\times$ type $\longrightarrow$ stat
NEW	ident $\longrightarrow$ stat
BLOCK	stat $\longrightarrow$ stat
FUN	ident $\times$ pars $\times$ type $\times$ stat $\longrightarrow$ stat

Übersetzung von Anweisungen

Auch Deklarationen werden hier als Anweisungen (Statements) betrachtet. Als Datentyp lautet die abstrakte Syntax von Statements wie folgt:

```
data stat = SKIP | ASSIGN Exp Exp | SEQ Stat Stat |
            COND Exp Stat Stat | LOOP Exp Stat |
            BLOCK Stat | Read String | Write Exp  |
            VAR String TYPE | TYPVAR String TYPE |
            FUN String [Par] TYPE Stat | NEW String
```

Hier ist die Übersetzungsfunktion für Statements:

$$compStat : B(G)_{stat} \to (\text{Symboltabelle} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to ([\text{Command}] \times \text{Symboltabelle} \times \mathbb{Z}))$$

compStat

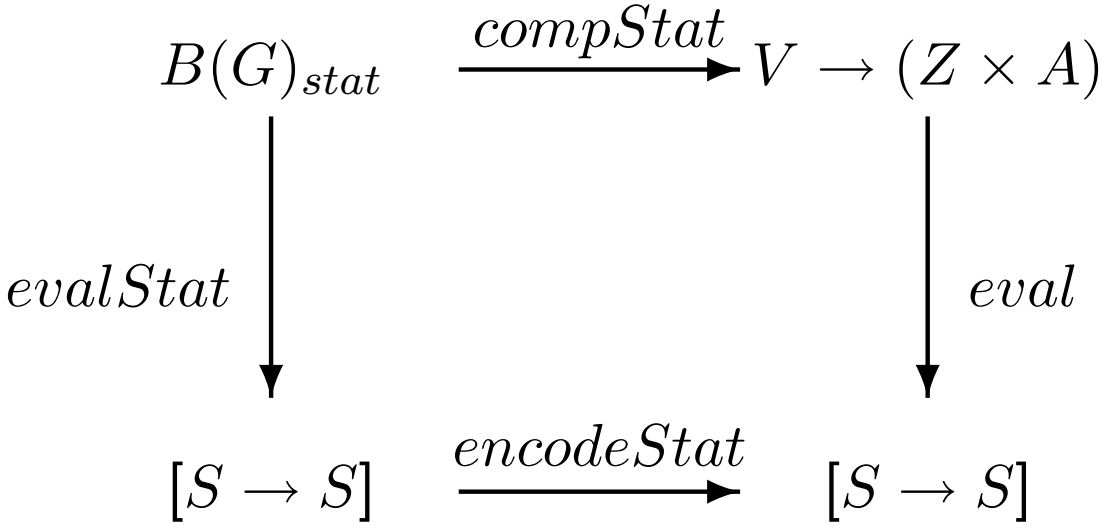
Im Gegensatz zu $compExp$ ist die Symboltabelle bei $compStat$ zugleich vererbtes und abgeleitetes Attribut: Sie wird zwar gelesen, aber bei der Übersetzung von Deklararationen auch verändert.
Weitere vererbte Attribute von $stat$ sind

- die Schachtelungstiefe des Scopes der Anweisung,
- die nächste freie Relativadresse,
- die nächste freie Befehlsnummer.

Anstelle eines Typdeskriptors liefert $compStat$ die nächste freie Relativadresse.

Diagrammkomponenten

Die Komponenten des Diagramms aus §1.1 lauten jetzt wie folgt:
Seien Val wieder die Menge der speicherbaren Werte,
$S = [Adr \to Val]$ die Menge der Speicherinhalte,
$Z = [Command]$ und $eval : V \to (Z \times A)$ eine adäquate Fortsetzung des Interpreters von Z auf den gesamten Bildbereich von $compStat$.



Übersetzung von Anweisungen

$compStat$ wird im Folgenden so definiert, daß für alle $st \in B(G)_{stat}$ und $f \in V \to (Z \times A)$ gilt:

$$compStat(st) = f \Rightarrow eval(f) = evalStat(st).$$

Definiert man $encodeStat$ als Identität auf $[S \to S]$, dann entspricht die Gültigkeit der Gleichung der Kommutativität des obigen Diagramms.

compStat (ASSIGN e e')

```
compStat SKIP st _ adr lab = ([],st,adr)

compStat (ASSIGN e e') st depth adr lab = (code",st,adr)
                where (code,Pointer _) = compExp e st depth lab
                      lab' = lab + length code
                      (code',td) = compExp (deref e') st depth lab'
                      offset = offset td st
                      code" = code++code'++popL offset++
                              MOV (ADR (IND TOP)) (REG ACC):
                              storeL (offset-1) ACC++popL (offset+1)
```

compStat (ASSIGN e e')

- Bei der Übersetzung einer Zuweisung e:=e' werden zunächst die Ausdrücke e und e' mit $compExp$ übersetzt. Da e' auf eine niedrigere Referenzstufe als e gehört, wird e' im Gegensatz zu e vor der Übersetzung 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 dem Adresswert von e beginnenden Kellerbereich,
  - Entkellern des Wertes von deref(e') und des Adresswertes von e.

compStat (Read id)

```
compStat (Read id) st depth adr lab = (code,st,adr)
                        where code = MOV ba (REG ACC) :
                                     ADD (REG ACC) (CON idadr) :
                                     readL (offset-1) ACC
                              (td,idadr,declDepth) = get st id
                              ba = baseAddress declDepth depth
                              offset = offset td st
```

Der Zielcode für `Read id` bewirkt das

- Laden von ACC mit der Basisadresse des Scopes des Lesebefehls,
- Addieren der Relativadresse idadr von id zur Berechnung der absoluten Adresse von id,
- Einlesen in den mit der absoluten Adresse von id beginnenden Kellerbereich.

compStat (Write e)

```
compStat (Write e) st depth adr lab = (code',st,adr)
                                  where code' = code++popL offset++
                                                writeL offset++popL offset
                                        (code,td) = compExp (deref e)
                                                            st depth lab
                                        offset = offset td st
```

Bei der Übersetzung von `Write e` wird zunächst der Ausdruck `e` dereferenziert und übersetzt. Dann wird der Wert von `deref e` ausgegeben und entkellert.

Iterative Sprachkonstrukte

```
compStat (SEQ s s') st depth adr lab = (code++code',st'',adr'')
    where (code,st',adr') = compStat s st depth adr lab
          lab' = lab + length code
          (code',st'',adr'') = compStat s' st' depth adr' lab'

compStat (COND e s s') st depth adr lab = (code',st'',adr'')
    where code' = code ++
                  MOV (ADR (IND TOP)) (REG BIT) :
                  pop :
                  CJF (CON labF) :
 labT:            codeT ++
                  GOTO (CON exit) :
 labF:            codeF
 exit:
          (code,primitive(BOOL)) = compExp (deref e) st depth lab
          labT = lab + length code + 3
          (codeT,st',adr') = compStat s st depth adr labT
          labF = labT + length codeT + 1
          (codeF,st'',adr'') = compStat s' st' depth adr' labF
          exit = labF + length codeF
```

Iterative Sprachkonstrukte

Der Zielcode eines Konditionals `COND e s s'`

- kellert den Wert des Booleschen Ausdrucks e,
- transportiert ihn nach BIT,
- springt im false-Fall zur Marke labF, also hinter den Befehl `GOTO (CON exit)`, und führt den Code codeF von s' aus.
- Im true-Fall werden der Code codeT von s ausgeführt und die Marke exit hinter codeF angesprungen.

Iterative Sprachkonstrukte

```
compStat (LOOP e s) st depth adr init = (code',st',adr')
    where code' =
init:               code ++
                    MOV (ADR (IND TOP) (REG BIT) :
                    pop :
                    CJF (CON exit) :
labT:               codeT ++
                    [GOTO (CON init)]
exit:
          (code,primitive(BOOL)) = compExp (deref e) st depth init
          labT = init + length code + 3
          (codeT,st',adr') = compStat s st depth adr labT
          exit = labT + length codeT + 1
```

Iterative Sprachkonstrukte

Der Zielcode einer Schleife `LOOP e s`

- kellert den Wert des Booleschen Ausdrucks e,
- transportiert ihn nach BIT,
- springt im false-Fall zur Marke exit, also hinter den Befehl `GOTO (CON init)`.
- Im true-Fall werden der Code codeT von s ausgeführt und die Marke init vor dem Befehl `CJF (CON exit)` angesprungen.

typ tid = t

```
compStat (TYPVAR tid t) st _ adr lab = ([],(tid,compType t,0,0):st,adr)
```

tid wird mit dem Typdeskriptor td von t in die Symboltabelle eingetragen. Zielcode wird nicht erzeugt.

var id : t

```
compStat (VAR id t) st depth adr lab = (pushL offset,st',adr+offset)
                                       where td = compType t
                                             offset = offset td st
                                             st' = (id,td,adr,depth):st
```

id wird mit dem Typdeskriptor td von t in die Symboltabelle eingetragen. Der Zielcode `pushL offset` reserviert Platz für den Wert von id. `adr+offset` liefert die nächste freie Relativadresse.

new id

```
compStat (NEW id) st depth adr lab = (code,st,adr)
                where code = pushHeap offset ++
                             [MOV ba (REG ACC),
                              ADD (REG ACC) (CON idadr),
                              MOV (ADR (REG HEAPTOP)) (IND ACC)]
                      (Pointer td,idadr,declDepth) = get st id
                      offset = offset td st
                      ba = baseAddress declDepth depth
```

Bei der Übersetzung einer Objektdeklaration NEW(id) muß der Typ von id ein Zeigertyp sein. Der entsprechende Deskriptor Pointer(td) hat als Argument den zugehörigen Objekttypdeskriptor. ba liefert die Basisadresse des Scopes der Deklaration von id.

`pushHeap offset` reserviert auf dem Heap Platz für das Objekt, auf das id zeigt. Der restliche Zielcode berechnet die absolute Adresse von id und legt den neuen Heaptop unter dieser Adresse ab.

Blockdeklaration

```
compStat (BLOCK s) st depth adr lab = (code',st,adr)
    where (code,_,locadr) = compStat s st
                                    (depth+1) (depth+1) (lab+2*depth+8)
          code' = 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 ++
                  push :
                  MOV (ADR (REG STP)) (IND TOP) :
                  code ++
                  popL locadr ++
                  [MOV (ADR (IND TOP)) (REG BA),
                   pop]
```

Blockdeklaration

Der Zielcode von `BLOCK s` bewirkt folgende Laufzeitaktionen:

- Kellern der aktuellen Basisadresse (= Basisadresse des umfassenden Scopes; an dieser Kellerposition steht beim Funktionsaufruf die Rücksprungadresse),
- 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,

Blockdeklaration

- Kellern des Displays des umfassenden Scopes,
- Kellern der Basisadresse des umfassenden Scopes und damit Vervollständigen des Displays des neuen Scopes,
- Ausführen des Blockrumpfes s,
- Entkellern der lokalen Adressen und des Displays des neuen Scopes (s.u.),
- Laden von BA mit der Basisadresse des umfassenden Scopes.

Blockdeklaration

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 s.

Blockdeklaration

- Die *nächste freie Befehlsnummer* lab+2*depth+8 ergibt sich aus den 8 Einzelbefehlen vor dem Code von s und der Tatsache, daß der Ladebefehl `loadL (depth-1) STP` zum Kellern des alten Displays in jedem Iterationsschritt aus 2 Einzelbefehlen besteht.
- `locadr` liefert die nächste freie Relativadresse nach Übersetzung von s. 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

```
compStat (FUN fid ps t s) st depth resadr lab = (code',st',resadr+offset)
    where code' = pushL offset ++
                  GOTO (CON exit) :
codeadr:           push :
                  MOV (ADR (REG TOP)) (REG BA) :
                  pop ++
                  loadL (depth-1) STP ++
                  push :
                  MOV (ADR (REG STP)) (IND TOP) :
labS:             code ++
                  popL locadr ++
                  [GOTO (ADR (IND TOP))]
exit:
          td = compType t
          offset = offset td st
          codeadr = lab+offset+1
          st' = (fid,Func codeadr td,resadr,depth):st
          st'' = compPars ps st' (depth+1) (-2)
          labS = codeadr+2*depth+5
          (code,_,locadr) = compStat s st'' (depth+1) (depth+1) labS
          exit = labS+length code+locadr+1
```

Übersetzung von Typen und Parametern

konkrete Syntax

type	$\longrightarrow$	**int**
type	$\longrightarrow$	**bool**
type	$\longrightarrow$	**^1**type
type	$\longrightarrow$	**array** [Int..Int] type
type	$\longrightarrow$	**record** stat
par	$\longrightarrow$	ident:type
par	$\longrightarrow$	**fun** ident:type
pars	$\longrightarrow$	par, pars
pars	$\longrightarrow$	par

abstrakte Syntax

int	$\longrightarrow$ type
bool	$\longrightarrow$ type
POINTER	type $\longrightarrow$ type
ARRAY	int $\times$ int $\times$ type $\longrightarrow$ type
RECORD	stat $\longrightarrow$ type
PAR	ident $\times$ type $\longrightarrow$ par
FPAR	ident $\times$ type $\longrightarrow$ par
_:_	par $\times$ pars $\longrightarrow$ pars
[_]	par $\longrightarrow$ pars

Übersetzung von Typen und Parametern

Für die Übersetzung eines Resultattyps bzw. einer Liste formaler Parameter verwenden wir entsprechende **Compile-Funktionen** $compType$ und $compPars$. $compType$ liefert den zu einem Typ gehörigen Typdeskriptor.

```
data Type = INT | BOOL | POINTER Type | ARRAY Int Int Type |
            RECORD Stat | NAME String

compType :: Type -> Typdeskriptor

compType INT                = INT
compType BOOL               = BOOL
compType (POINTER t)        = Pointer (compType t)
compType (ARRAY lwb upb t) = Array lwb (upb-lwb+1) (compType t) |
compType (RECORD s)         = Record st where (_,st,_) = compStat s [] 0 0 1
compType (NAME id)          = Name id |
```

Übersetzung von Typen und Parametern

```
data  Par = PAR String Type | FPAR String Type

compPars :: [Par] -> Symboltabelle -> Int -> Int -> Symboltabelle
compPar  :: Par -> Symboltabelle -> Int -> Int -> (Symboltabelle,Int)

compPars [] st _ _ = st |
compPars (p:ps) st depth adr = compPars ps st depth adr'
                               where (st,adr') = compPar p st depth adr

compPar (PAR id t) st depth adr = (st',adr')
                               where td = compType t
                                     adr' = adr-offset td st
                                     st' = (id,td,adr',depth):st

compPar (FPAR fid t) st depth adr = (st',adr')
                               where td = compType t
                               adr' = adr-3
                               st' = (fid,Formalfunc td,adr',depth):st
```

Übersetzung von Typen und Parametern

Damit ist der Compiler einer imperativen Sprache in eine Assemblersprache vollständig definiert—bis auf die Übersetzung der Startproduktion:

```
data prog = PROG Stat

compProg (PROG s) = code where (code,_,_) = compStat s [] 0 0 1
```

Wir beginnen also mit der leeren Symboltabelle, der Schachtelungstiefe 0, der Kelleradresse 0
und der Befehlsnummer 1.

Übersetzer funktionaler Sprachen

Wir betrachten einige Haskell-Konstrukte zum Aufbau funktionaler Ausdrücke::

1. Variable oder Konstante x
2. Applikation $(e\ e')$
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** $x = e$
8. Fixpunkt **fix** $x_1 = e_1 | \ldots | x_n = e_n$

Übersetzer funktionaler Sprachen - 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*

Inferenzregeln

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 Typvariablen enthalten.
  - Ist x eine Konstante, dann sind alle Typvariablen von $state(x)$ gebunden.

Inferenzregeln

- Der Ausdruck $state \vdash exp : value$ bedeutet, daß im Zustand $state$ dem Ausdruck exp ein Wert $value$ zugeordnet wird.
  Gilt das für die Konklusionen, dann auch für die Prämissen einer Regel.

Das Compilerschema aus § 4.1 entspricht z.B. Regeln der Form:

$$\frac{a_0 \vdash p(x_1, \ldots, x_n) : e_n(a_0, \ldots, a_n)}{e_0(a_0) \vdash x_1 : a_1, \ldots, e_n(a_0, \ldots, a_{n-1}) \vdash x_n : a_n} \Uparrow$$

Regeln zur Herleitung der Typen

1. $$\frac{state \vdash x : t}{True} \Uparrow$$ t ist eine *Instanz* des Typs $state(x)$

2. $$\frac{state \vdash (e\ e') : t}{state \vdash e : t' \to t,\ state \vdash e' : t'} \Uparrow$$

3. $$\frac{state \vdash (e_1, \ldots, e_n) : (t_1, \ldots, t_n)}{state \vdash e_1 : t_1,\ \ldots,\ state \vdash e_n : t_n} \Uparrow$$

4. $$\frac{state \vdash \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : t}{state \vdash e : bool,\ state \vdash e_1 : t,\ state \vdash e_2 : t} \Uparrow$$

Regeln zur Herleitung der Typen

5. $$\frac{state \vdash \textbf{let } x = e' \textbf{ in } e : t}{state \vdash e' : t',\ update(state)(x, \forall\alpha_1, \ldots, \alpha_n t') \vdash e : t} \Uparrow$$

$\alpha_1 \ldots \alpha_n$ sind die Typvariablen von t',
die in $state$ nicht frei vorkommen

6 $$\frac{state \vdash \lambda x.e : \alpha \to t}{update(state)(x, \alpha) \vdash e : t} \Uparrow$$ α ist eine neue Typvariable

7 $$\frac{state \vdash \textbf{fix } x = e : t}{update(state)(x, \alpha) \vdash e : t} \Uparrow$$ α ist eine neue Typvariable

8 $$\frac{state \vdash \textbf{fix } x_1 = e_1 | \ldots | x_n = e_n : (t_1, \ldots, t_n)}{state' \vdash e_1 : t_1,\ \ldots,\ state' \vdash e_n : t_n} \Uparrow$$

$\alpha_1, \ldots, \alpha_n$ sind neue Typvariablen,
$state' = update(\ldots(update(state)(x_1, \alpha_1), \ldots))(x_n, \alpha_n)$

Typen und Instanzen

- Typen sind hier aus Typvariablen wie α und β, den Typkonstruktoren $(\_, \_)$ (Produkt) und $\rightarrow$ sowie Datentypnamen zusammengesetzte Ausdrücke.
  Typvariablen können durch den Allquantor $\forall$ gebunden sein.
  Der Typ der map-Funktion ist z.B. durch den Ausdruck

$$\forall\alpha, \beta\ (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$$

  gegeben. α und β sind hier gebunden.
- 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

$$\forall\beta\ ([\gamma] \rightarrow \beta) \rightarrow [[\gamma]] \rightarrow [\beta].$$

Typunifikation

- 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.

☞ Unifikation der Typen t und t' bedeutet, daß die Variablen von t und t' solange instanziiert werden, bis t und t' identisch sind.

Unifikation - Beispiel

- $t = (\alpha \to (\text{Int} \to \text{Bool}))$ und $t' = ((\beta, \gamma) \to \delta)$ haben z.B. den **Unifikator** $\sigma = \{(\beta, \gamma)/\alpha,\ (\text{Int} \to \text{Bool})/\delta\}$
- Für die **Instanzen von t und t' durch σ**: $(\beta, \gamma) \to (\text{Int} \to \text{Bool}))$ bzw. $(\beta, \gamma) \to (\text{Int} \to \text{Bool}))$ schreibt man kurz $t\sigma$ bzw. $t'\sigma$.
- Die Typen $\alpha \to (\text{Int} \to \text{Bool})$ und $\alpha' \to (\beta, \gamma)$ haben keinen Unifikator, weil $Int \to Bool$ mit (β, γ) *nicht* unifizierbar ist.

Implementierung

Die folgende Funktion `unify` berechnet den *allgemeinsten* Unifikator zweier Typterme.

```
isin :: a -> Term a -> Bool

x `isin` V y    = x == y
x `isin` F _ ts = any (isin x) ts

for :: Term a -> a -> Term a

(t `for` x) y = if x == y then t else V y

(>>>) :: Term a -> (a -> Term a) -> Term a

V x >>> f    = f x
F x ts >>> f = F x (map (>>> f) ts)
```

Implementierung

```
andThen :: (a -> Term a) -> (a -> Term a) -> a -> Term a

(f `andThen` g) x = f x >>> g

unify    :: Term a -> Term a -> Maybe (a -> Term a)

unify (V x) (V y)       = if x == y then Just V
                                    else Just (V y `for` x)
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
```

Implementierung

```
unifyall :: [Term a] -> [Term a] -> Maybe (a -> Term 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
```

Occurs Check

`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.

Die Funktion **Analyze**

Entsprechend dem Aufbau der funktionalen Ausdrücke unseres Mini-Haskell bilden wir aus den Typinferenzregeln die Funktion

`analyze` : Ausdruck $\rightarrow$ Zustand $\rightarrow$ Unifikator $\rightarrow$ (Typ,Unifikator)

mit folgender Eigenschaft:

`analyze e state` $id = (t, \sigma)$ $\Rightarrow$ 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.

Die Funktion **Analyze** - Implementierung

```
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
```

Die Funktion **Analyze** - Implementierung

$$\texttt{analyze } (\texttt{e}_1,\ldots,\texttt{e}_n) \texttt{ state } \sigma = ((t_1\sigma_n,\ldots,t_n\sigma_n),\ \sigma_n)$$
$$\texttt{where } (t_1,\sigma_1) = \texttt{analyze e}_1 \texttt{ state } \sigma$$
$$(t_2,\sigma_2) = \texttt{analyze e}_2 \texttt{ state } \sigma_1$$
$$\vdots$$
$$(t_n,\sigma_n) = \texttt{analyze e}_n \texttt{ state } \sigma_{n-1})$$

$$\texttt{analyze } (\textbf{if } \texttt{e } \textbf{then } \texttt{e}_1 \textbf{ else } \texttt{e}_2) \texttt{ state } \sigma = (t_2\sigma_4,\ \sigma_3\sigma_4)$$
$$\texttt{where } (t,\sigma_0) = \texttt{analyze e state } \sigma$$
$$\sigma_1 = \texttt{unify } t \textbf{ bool}$$
$$(t_1,\sigma_2) = \texttt{analyze e}_1 \texttt{ state } \sigma_0\sigma_1$$
$$(t_2,\sigma_3) = \texttt{analyze e}_2 \texttt{ state } \sigma_2)$$
$$\sigma_4 = \texttt{unify } t_1\ t_2$$

Die Funktion **Analyze** - Implementierung

analyze (**let** x = e' **in** e) state σ = $(t',\ \sigma_1)$
    where (t, σ_0) = analyze e' state σ
          (t', σ_1) = analyze e (update state (x,$\forall\alpha_1, \ldots, \alpha_n t$)) σ_0
          wobei $\alpha_1, \ldots, \alpha_n$ die Typvariablen von t sind,
               die in state nicht frei vorkommen

analyze (\x$\rightarrow$ e) state σ = $(\alpha\sigma_0 \rightarrow t,\ \sigma_0)$
    where (t, σ_0) = analyze e (update state (x,α)) σ
          wobei α eine neue Typvariable ist

analyze (**fix** x=e) state σ = $(t\sigma_1,\ \sigma_0\sigma_1)$
    where (t, σ_0) = analyze e (update state (x,α)) σ
          σ_1 = unify t $(\alpha\sigma_0)$
          wobei α eine neue Typvariable ist

Die Funktion **Analyze** - Implementierung

$$
\begin{aligned}
&\texttt{analyze (fix } x_1\texttt{=}e_1\texttt{|...|}x_n\texttt{=}e_n\texttt{) state } \sigma \texttt{ = } ((t_1\tau_n, \ldots, t_n\tau_n),\ \sigma_n\tau_n)\\
&\quad \texttt{where state' = foldl update state [(}x_1, \alpha_1\texttt{),...,(}x_n, \alpha_n\texttt{)]}\\
&\qquad (t_1, \sigma_1) \texttt{ = analyze } e_1 \texttt{ state' } \sigma\\
&\qquad \tau_1 \texttt{ = unify } t_1\ (\alpha_1\sigma_1)\\
&\qquad (t_2, \sigma_2) \texttt{ = analyze } e_2 \texttt{ state' } \sigma_1\tau_1\\
&\qquad \tau_2 \texttt{ = unify } t_2\ (\alpha_2\sigma_2)\\
&\qquad \vdots\\
&\qquad (t_n, \sigma_n) \texttt{ = analyze } e_n \texttt{ (state' } \sigma_{n-1}\tau_{n-1}\texttt{)}\\
&\qquad \tau_n \texttt{ = unify } t_n\ (\alpha_n\sigma_n)\\
&\qquad \texttt{wobei } \alpha_1, \ldots, \alpha_n \texttt{ neue Typvariablen sind}
\end{aligned}
$$