Bachelorarbeit

Integration von Auszeichnungssprachen in Curry

Max Amadeus Deppert

31. März 2014

Betreut durch:
Prof. Dr. Michael Hanus
Dipl.-Inf. Jan Tikovsky

Arbeitsgruppe für
Programmiersprachen und Übersetzerkonstruktion

Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Weiterhin versichere ich, dass diese Arbeit noch nicht als Abschlussarbeit an anderer Stelle vorgelegen hat.

31. März 2014

Datum Unterschrift

Zusammenfassung

Ein Ausdruck einer domänenspezifischen Sprache verliert schnell an Prägnanz, sobald er durch einen Ausdruck einer Programmiersprache repräsentiert wird. Die funktional-logische Programmiersprache Curry überwindet dieses Problem durch die Möglichkeit sogenannter Codeintegration domänenspezifischer Sprachen. In dieser Arbeit wird ein Parser entwickelt, der die Codeintegration in Curry um die Auszeichnungssprachen XML und HTML erweitert. Inspiriert durch das shakespearische Haskell-Template Hamlet, wird dabei einerseits eine an Curry angelehnte Layout-Regel umgesetzt und andererseits die sogenannte Interpolation von Curry-Ausdrücken ermöglicht.

Inhaltsverzeichnis

1. Einleitung

Auszeichnungssprachen werden heute in den verschiedensten Anwendungsgebieten eingesetzt. Die *HTML* und die *XML* wurden unter ihnen besonders populär. Über das Internet bzw. das *World Wide Web*, welches auf der HTML begründet wurde, verbreitete sich auch die XML, die grundlegend für die Spezifikation vieler weiterer Sprachen und Standards werden sollte, auf der ganzen Welt.
Der große Erfolg dieser *domänenspezifischen Sprachen* (auch *DSL*) lässt sich nicht zuletzt dadurch erklären, dass Daten mit ihnen in einem Dokumentformat formuliert und strukturiert werden können, welches für Mensch und Maschine gleichermaßen lesbar ist. Leider geht dieser Vorteil für den Menschen schnell verloren, sobald Ausdrücke einer DSL in einer Programmiersprache repräsentiert werden. Zwar werden die erforderlichen Datenstrukturen bzw. Datentypen in den meisten Programmiersprachen durch eigene Module oder Programmbibliotheken zur Verfügung gestellt, jedoch leidet die Prägnanz der DSL in der Syntax der Programmiersprache meist erheblich.
Erfreulicherweise ist dieser Missstand überwindbar. Native Ausdrücke einer DSL können im Programm-Quelltext durch einen *Präprozessor* – mittels eines entsprechenden *Parsers* – in einen Ausdruck der Programmiersprache umgewandelt werden, bevor die eigentliche Übersetzung des Programms beginnt. Diese sogenannte *Codeintegration* sorgt dafür, dass die Prägnanz der domänenspezifischen Sprache auch beim Programmieren erhalten bleibt.
Im Rahmen der Bachelorarbeit von Jasper Paul Sikorra an der Universität Kiel, wurde die universelle, funktional-logische Programmiersprache *Curry* um die Möglichkeit einer solchen Codeintegration erweitert.
In dieser Arbeit soll die Codeintegration in Curry durch Parser für XML und HTML erweitert werden. Inspiriert durch das aus der funktionalen Programmiersprache *Haskell* bekannte Paket *Hamlet*, wird hierfür außerdem eine Curry- bzw. Haskell-ähnliche *Layout-Regel* entworfen und implementiert werden. Schließlich sollen Curry-Ausdrücke durch sogenannte *Interpolation* in den Quelltext der Auszeichnungssprache einbetten werden können.
Dafür werden zunächst die notwendigen Grundlagen in *XML*, *HTML*, *Curry* und *Hamlet* besprochen, bevor einige motivierende Worte in den Entwurf und die Implementierung überleiten.

Teil I.

Grundlagen

2. Extensible Markup Language (XML)

Die Extensible Markup Language (XML) ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten durch Textdateien. Die 1998 erstmalig vom *World Wide Web Consortium* (auch *W3C*) herausgegebene Spezifikation ist heute ein weltweit verbreitetes Format für die Speicherung und den Austausch von plattform- und implementierungsunabhängigen Daten, insbesondere über das Internet.

2.1. Struktur

XML-Dokumente werden durch sogenannte *Elemente* aufgebaut und strukturiert. Ein *Element* besitzt einen alphanumerischen Namen, eventuell einige *Attribute* (siehe 2.2.1) und einen *Elementinhalt*. Der Elementinhalt kann dabei aus (Text-)Daten und weiteren Elementen zusammengesetzt sein:

```
<thesis>
  <titel>
    Integration von Auszeichnungssprachen in Curry
  </titel>
  <datum>31. März 2014</datum>
  <autor>Max Amadeus Deppert</autor>
</thesis>
```

Dieses Beispiel spezifiziert ein Element `thesis`, das weitere Elemente enthält, welche die eigentlichen Daten enthalten.
Durch die beliebig tiefe *Verschachtelung* von Elementen, werden die enthaltenen Daten hierarchisch strukturiert. Beispielsweise könnte das Element `datum` folgendermaßen noch genauer spezifiziert und ersetzt werden:

```
<datum>
  <tag>31</tag>
  <monat>03</monat>
  <jahr>2014</jahr>
</datum>
```

Diese Struktur eines Kalenderdatums ist für einen Computer deutlich einfacher zu verarbeiten, aber auch für uns Menschen noch gut lesbar.
Es ist zu sehen, dass jeder Elementinhalt syntaktisch begrenzt wird. Diese Begrenzungen werden *Tags* genannt und spezifizieren das Element.

2.2. Tags

Da XML-Dokumente auf Klartext (*plain text*) aufgebaut sind, müssen Elemente bzw. Elementinhalte eindeutig begrenzt werden. Dafür dienen sogenannte *Tags*, welche man in *Start-, End-* und *Leer-Tags* unterteilt. Ein Element kann dann formuliert werden, indem der Elementinhalt zwischen einem vorangehenden Start-Tag und einem abschließenden End-Tag steht.
Ein einfaches Start-Tag hat in der XML die folgende Form:

```
<E>
```

Dabei ist E ein Name, der den Namen des Elements spezifiziert. Außerdem muss das abschließende End-Tag den gleichen Namen besitzen und wird in der Form

```
</E>
```

geschrieben.
Ein Element E wird dem zur Folge so spezifiziert:

```
<E>X</E>
```

Dabei ist X ein gültiger XML-Ausdruck und kann also selbst wieder Elemente enthalten. Ist ein Element leer, i.e. <E></E>, so kann stattdessen auch abkürzend <E/> geschrieben werden. Ein solches *Tag* wird als *Leer-Tag* bezeichnet, ist aber ebenfalls ein *Start-Tag* und kann somit Attribute (siehe 2.2.1) enthalten.
Wie in Abb. 2.1 zu sehen ist, nutzt man der Übersicht halber wahlweise gerne die eingerückte Form:

```
<E>
   X
</E>
```

Zusätzlich zu Elementinhalten, können Elementen sogenannte *Attribute* zugeordnet werden.

2.2.1. Attribute

Wenn Daten keine weiteren Elemente enthalten, kann man sie auch als Elementattribute formulieren. Diese werden im Start-Tag notiert und genügen der folgenden Form:

```
<E  a1=V1  ...  an=Vn>X</E>
```

Dem Element E werden also die Attribute $a_1, \ldots, a_n$ mit dem jeweiligen V_i zugeordnet. Ein V_i enthält einen Attributswert v_i entweder in der Form $V_i =$ 'v_i' oder $V_i =$ "v_i" für alle $1 \leq i \leq n$.
Attributswerte müssen also in einfache oder doppelte Anführungszeichen gesetzt werden.

Auch diese sind in Abb. 2.1 als Städtenamen zu sehen.

```
<stadtstaaten>
  <stadt name="Berlin">
    <qkm>891,70</qkm>
    <dichte>3785</dichte>
  </stadt>
  <stadt name="Bremen">
    <qkm>419,24</qkm>
    <dichte>1562</dichte>
  </stadt>
  <stadt name="Hamburg">
    <qkm>755,30</qkm>
    <dichte>2296</dichte>
  </stadt>
</stadtstaaten>
```

Abbildung 2.1.: Ein XML-Dokument, welches die Fläche in km^2 und die Bevölkerungsdichte der deutschen Stadtstaaten in Menschen pro km^2 beschreibt

2.2.2. Spezifikation

Um ein wenig genauer zu werden, schreibt die XML-Spezifikation für ein *Start-Tag* etwa das folgende Gerüst vor:

1. Das erste Zeichen eines *Start-Tags* muss ein < (U+003C) sein.
2. Die nächsten Zeichen definieren den alphanumerischen *Tagnamen*, der nicht mit einer Ziffer beginnt.
3. Wenn es Attribute geben soll, muss jetzt mindestens ein Leerzeichen folgen.
4. Nun können Attribute folgen, die jeweils durch mindestens ein Leerzeichen getrennt werden:
   a) Ein *Attribut* beginnt mit einem *Attributsnamen*, der keine Leerzeichen, Anführungsstriche, Gleichheitszeichen oder den Schrägstrich / enthält.
   b) Optional folgt eine Anzahl von Leerzeichen.
   c) Es folgt ein Gleichheitszeichen = (U+003D).
   d) Optional folgt eine Anzahl von Leerzeichen.
   e) Am Ende des *Attributs* steht der *Attributswert*, der entweder in einfachen oder in doppelten Anführungszeichen steht und das jeweilige Anführungszeichen selbst nicht enthält.
5. Nach den Attributen – oder nach dem *Tagnamen*, falls es keine Attribute gibt – können ein oder mehrere Leerzeichen stehen.

6. Wenn das *Tag* ein *Leer-Tag* ist, folgt nun ein einzelner Schrägstrich / (U+002F).
7. Das letzte Zeichen eines *Start-Tags* muss ein einzelnes > (U+003E) sein.

Entsprechend wird ein *End-Tag* spezifiziert:

1. Das erste Zeichen eines *End-Tags* muss ein < (U+003C) sein.
2. Das zweite Zeichen eines *End-Tags* muss ein Schrägstrich / (U+002F) sein.
3. Die nächsten Zeichen definieren den alphanumerischen *Tagnamen*, der nicht mit einer Ziffer beginnt.
4. Optional folgt eine Anzahl von Leerzeichen.
5. Das letzte Zeichen eines *End-Tags* muss ein einzelnes > (U+003E) sein.

3. Hypertext Markup Language (HTML)

Die Hypertext Markup Language (HTML) ist die bekannteste unter den Auszeichnungssprachen. Seit sie 1990 von Tim Berners-Lee erfunden wurde, der zu dieser Zeit auch den ersten Webbrowser programmierte, bildet sie bis heute die Grundlage für die Strukturierung von Inhalten im *World Wide Web*. Wie die XML ist sie eine textbasierte, hierarchische Auszeichnungssprache, die zur Strukturierung von digitalen Inhalten wie Texten, Bildern, Videos und vor allem den sogenannten *Hyperlinks*, über die ein Webbrowser von einem Dokument zu einem nächsten navigieren kann, genutzt wird.

Zeitlich ist leicht zu erkennen, dass die HTML lange vor der XML entstand. So sollte die HTML ein Vorbild für eine allgemeinere, semantikfreie und dadurch vielseitig einsetzbare Auszeichnungssprache werden. Daher sind sich HTML und XML sehr ähnlich, weshalb wir uns hier auf die Unterschiede konzentrieren wollen.

3.1. Struktur

HTML-Dokumente sind wie XML-Dokumente strukturiert und werden darüber hinaus in nahezu identischer Syntax formuliert. Allerdings können Namen in der HTML – anders als in der XML – nicht länger frei gewählt werden, d.h. es existiert eine endliche Menge erlaubter Elemente und Attribute. Dieser Umstand gründet darin, dass ein HTML-Dokument nicht nur eine Datenstruktur spezifiziert, sondern darüber hinaus allen enthaltenen Daten eine individuelle Bedeutung zukommt, welche ein Webbrowser interpretieren kann. Die Spezifikation bestimmt insbesondere eindeutig, welche Attribute und Inhalte in einem Element erlaubt sind (siehe 3.3).
Ein *Hyperlink*, der auf die Webseite der Universität Kiel verweist, kann in HTML beispielsweise so formuliert werden:

Beispiel 3.1: Ein HTML-Hyperlink auf die Webseite der Universität Kiel

```
<a href="http://www.uni-kiel.de/">
  Christian-Albrechts-Universität zu Kiel
</a>
```

Das Element `a` definiert einen Hyperlink, wobei über das Attribut `href` das *Sprungziel* des Hyperlinks angegeben wird. Der Elementinhalt legt einen Ausdruck fest, der *verlinkt* werden soll – in diesem Fall ist dies eine Zeichenkette, wobei dort beispielsweise auch eine Grafik eingebunden werden könnte.

3.2. Attribute

Im Gegensatz zur XML erlaubt die HTML auch nicht-angeführte Attributswerte. Der Formalisierung aus 2.2.1 folgend, darf demnach in einem Element

```
<E  a1=V1  ...  an=Vn>X</E>
```

ein jedes V_i auch in der Form $V_i = v_i$ geschrieben werden, falls v_i mindestens ein Zeichen, aber weder Leerzeichen, Gleichheitszeichen noch jegliche Anführungszeichen enthält. Außerdem darf die rechte Seite eines Attributs a_i inklusive des Gleichheitszeichens ganz ausgelassen werden, falls v_i die leere Zeichenkette bezeichnet und in Anführungszeichen steht, d.h. $V_i =$ '' oder $V_i =$ "". Es sind also äquivalent:

```
<E  a1="">X</E>
```

```
<E  a1=''>X</E>
```

```
<E  a1>X</E>
```

Nicht erlaubt ist hingegen:

```
<E  a1=>X</E>
```

3.3. Content Model

Die HTML-Spezifikation schreibt für jedes HTML-Element genau vor, welche Inhalte bzw. Elemente es enthalten darf. Zu diesem Zweck wird die Menge aller HTML-Elemente in *Inhaltsklassen* eingeteilt, welche aber nicht notwendigerweise disjunkt sind. Beispielsweise ist der folgende Ausdruck nicht *HTML-konform*:

```
<em>
  Das ist
  <p>
    verboten!
  </p>
</em>
```

Das HTML-Element `em` gehört zur Inhaltsklasse des sogenannten *Phrasing content* und darf selbst nur *Phrasing content* enthalten. Das Element `p` hingegen wird einer Oberklasse, dem sogenannten *Flow content*, zugeordnet – allerdings nicht dem *Phrasing content*.

4. Curry

Curry ist eine international entwickelte, universelle Programmiersprache, welche die beiden wichtigsten deklarativen Programmierparadigmen der funktionalen und logischen Programmierung integriert. So werden die Möglichkeiten und Vorteile der funktionalen und logischen Programmierung in Curry nahtlos vereinigt und darüber hinaus sogar noch erweitert. Wie die funktionale Programmiersprache Haskell, verdankt Curry ihren Namen dem US-amerikanischen Logiker und Mathematiker *Haskell Brooks Curry*. Dass dies kein Zufall ist, ist bereits daran zu erkennen, dass Curry syntaktisch weitgehend durch Haskell inspiriert ist, und ein Curry-Programm einem Haskell-Programm sehr ähnlich sieht:

```
square :: Integer -> Integer
square x = x * x
```

Hier wird eine Selbstabbildung `square` auf den ganzen Zahlen definiert, welche das Quadrat einer ganzen Zahl berechnet. Beispielsweise wird der Ausdruck `square 4` mit der *linken Regelseite* `square x` durch die *rechte Regelseite* `x * x` zu `4 * 4` ausgewertet, was weiter ausgewertet wird zu `16`, dem Quadrat von `4`.

4.1. Datentypen und Typsynonyme

Ein Datentyp wird in Curry wie in Haskell in folgender Form deklariert:

$$\texttt{data}\ T\ \tau_1\ \ldots\ \tau_n\ \texttt{=}\ C_1\ \sigma_{1,1}\ \ldots\ \sigma_{1,k_1}\ \texttt{|}\ \ldots\ \texttt{|}\ C_m\ \sigma_{m,1}\ \ldots\ \sigma_{m,k_m}$$

Dabei werden durch $C_1, \ldots, C_m$ die sogenannten (Daten-)Konstruktoren des n-stelligen Datentypkonstruktors T deklariert. Diese haben den Typ

$$C_i\ \texttt{::}\ \sigma_{i,1}\ \texttt{->}\ \ldots\ \texttt{->}\ \sigma_{i,k_i}\ \texttt{->}\ T\ \tau_1\ \ldots\ \tau_n$$

für alle $1 \leq i \leq m$.
Dabei sind die $\tau_1, \ldots, \tau_n$ sogenannte Typvariablen oder Typbezeichner, durch die die Typen $\sigma_{i,j}$ für alle $1 \leq i \leq m$, $1 \leq j \leq k_i$ bestimmt werden.
Vordefiniert ist beispielsweise der einfache Datentyp *Bool*:

```
data Bool = True | False
```

Eine Liste könnte man so deklarieren:

```
data List a = Empty | Cons a (List a)
```

Allerdings sind Listen ebenfalls vordefiniert, wobei `List a` geschrieben wird als `[a]`, `Empty` als `[]` und `Cons` durch einen Infixoperator `(:)` dargestellt wird. Ist also `x` ein Element und `xs` eine Liste gleichen Elementtyps, so lässt sich durch `x:xs` (statt `Cons x xs`) eine neue Liste konstruieren.

Das folgende Beispiel zeigt die Deklaration eines Datentyps für nicht-leere Binärbäume mit einem festen Typ für Knoten und Blätter:

```
data BinTree a = Leaf a
               | Node (BinTree a) a (BinTree a)
```

Ein Binärbaum des Typs a ist demnach entweder ein Blatt (Leaf), dessen Wert vom Typ a ist, oder ein Knoten (Node), bestehend aus einem linken (Teil-)Binärbaum des Typs a, einem (Knoten-)Wert des Typs a und einem rechten (Teil-)Binärbaum des Typs a. Ein nicht-leerer Binärbaum, dessen Blätter und Knoten ganze Zahlen sind, ist demnach durch den Typ `Tree Int` gegeben.

Um für mehr Übersicht und weniger Schreibarbeit zu sorgen, kann man sich sogenannter Typsynonyme bedienen. Ein Typsynonym wird wie folgt deklariert:

```
type S λ1 ... λk = σ
```

So wird ein k-stelliger Typkonstruktor S definiert, der im Typausdruck σ die paarweise verschiedenen Typvariablen $\lambda_1, \ldots, \lambda_k$ durch die übergebenen Typen ersetzt. Sind also Typen $t_1, \ldots, t_k$ gegeben, so kann man S als die Abbildung verstehen, die in σ für alle $1 \leq i \leq k$ gerade λ_i auf t_i abbildet. Damit ist $(S\ t_1\ \ldots\ t_k)$ durch den Typ gegeben, der durch die Anwendung von S auf den Typausdruck σ entsteht.
Damit können wir die oben bereits genannten Binärbäume ganzer Zahlen auch durch ein Typsynonym IBinTree deklarieren:

```
type IBinTree = BinTree Int
```

Sollten wir beispielsweise oft mit Binärbaumen von Listen arbeiten, so könnten wir LBinTree deklarieren:

```
type LBinTree a = BinTree [a]
```

Typsynonyme stellen keine Typerweiterung dar, sondern lassen sich zu jeder Zeit durch den lediglich neu benannten Typ ersetzen.

4.2. Funktionen, Guards und lokale Deklarationen

Der Typ einer Funktion wird in Curry folgendermaßen deklariert:

```
f :: τ1 -> ... -> τn -> τ
```

Dabei ist f der Bezeichner der zu deklarierenden Funktion und $\tau_1, \ldots, \tau_n, \tau$ Typausdrücke.
Die Funktion f wird durch Gleichungen bzw. (Auswertungs-)Regeln definiert. Die einfachste Form einer solchen Regel sieht wie folgt aus:

```
f t1 ... tn = e
```

Dabei sind $t_1, \ldots, t_n$ Terme oder auch Pattern und e ein Ausdruck. Es ist durchaus legitim mehrere solcher Gleichungen anzugeben, um eine Funktion vollständig zu definieren.
Beispielsweise können wir für die bereits besprochenen Binärbäume eine Funktion `leafs` definieren, die die Anzahl der Blätter eines gegebenen Binärbaums bestimmt:

```
leafs :: BinTree a -> Int
leafs (Leaf _) = 1
leafs (Node lt _ rt) = (leafs lt) + (leafs rt)
```

Die Unterstriche `'_'` heißen *Wildcards*, welche als *Pattern* einen beliebigen Wert bzw. Typ erlauben (auch *matchen*). Außerdem lässt sich mit ihrer Hilfe mehr Übersicht erlangen, da es in diesem Beispiel offensichtlich unerheblich ist, welche Werte die Knoten und Blätter besitzen. Wollte man `leafs` dennoch in nur einer Regel definieren, bediente man sich eines `case`-Ausdrucks, der ebenfalls *Pattern* zulässt:

```
leafs t = case t of
              Leaf _ -> 1
              Node lt _ rt -> (leafs lt) + (leafs rt)
```

Durch sogenannte *Guards* ist es zudem möglich, bedingte Gleichungen zu formulieren, welche der Form

```
f t1 ... tn | c1 = e1
            | ...
            | ck = ek
```

genügen, wobei $c_1, \ldots, c_k$ Bedingungen und $e_1, \ldots, e_k$ Ausdrücke sind. Im Falle der Auswertung wird aufsteigend nach einem $i \in \{1, \ldots, k\}$ gesucht, bis eine erfüllende Bedingung c_i gefunden ist, woraufhin die Suche abbricht und e_i als rechte Regelseite bestimmt wird. Die durch den senkrechten Strich eingeleiteten Bedingungen werden *Guards* genannt.

Oft werden Deklarationen nur innerhalb einer bestimmten Funktion benötigt oder sollen außerhalb auch gar nicht sichtbar sein. In diesem Fall können wir das Schlüsselwort `where` oder auch `let ... in` verwenden, um eine Liste lokaler Deklarationen anzugeben:

```
f t1 ... tn = e
  where φ1 = e1
        ⋮
        φk = ek
```

Dabei sind die Deklarationen $\varphi_1, \ldots, \varphi_k$ nur in den Ausdrücken $e_1, \ldots, e_k$ und e sichtbar. Auf diese Weise lassen sich beispielsweise die Fibonnaci-Zahlen als eine Funktion `fib` mit einer lokalen Funktion `fib'` definieren:

```
fib n = fib' 0 1 n
  where fib' a _ 0 = a
        fib' a b i = fib' b (a+b) (i-1)
```

Außerdem wird beispielsweise

```
let a = 10
    b = 4*a
 in b+2
```

zu `42` ausgewertet.

4.3. Freie Variablen

Bisher haben wir nur rein funktionale Konzepte aus Curry besprochen. Wie zu Beginn erwähnt, lassen sich mit Curry allerdings auch logische Probleme lösen. Freie Variablen und *Constraints* stellen dabei wesentliche formale Mittel dar.
Anders als in Prolog, wo sämtliche Variablen – insbesondere also freie Variablen – im globalen Namensraum implizit eingeführt sind, müssen freie Variablen in Curry explizit als solche deklariert werden. Dafür wird das Schlüsselwort `free` verwendet:

```
let x free in [1,x,3] =:= [1,2,3]
```

In diesem Ausdruck ist `x` eine freie Variable und so wird Curry versuchen, die Gleichung `[1,x,3] =:= [1,2,3]` nach `x` aufzulösen, und die Lösung `x = 2` bestimmen.

Das *Constraint* `=:=` fordert dabei die *strikte Gleichheit* beider Seiten. Dieses *Gleichheits-Constraint* ist erfüllt, falls beide Seiten zu einem identischen *Grundterm* reduziert werden können.

So können wir etwa eine Funktion `last` definieren, welche das letzte Element einer Liste liefert:

```
last :: [a] -> a
last xs | let ys,y free in ys ++ [y] =:= xs = y
```

Curry sucht hier also eine initiale Liste `ys` und ein `y`, deren Konkatenation in *strikter Gleichheit* zu `xs` steht, so dass `y` das letzte Element von `xs` sein muss und zurückgegeben wird.

4.4. Layout

In Curry kann und muss die Struktur von Quelltext-Blöcken ganz ohne Klammern, Semikola o.Ä. festgelegt werden. Wie in Haskell, bedient man sich dazu einem Quelltext-Layout. Die assoziierte *Layout-Regel* verbindet oder trennt einzelne syntaktische Einheiten aufgrund ihrer Einrückung:

```
f a b = case a of
              b -> c
                 c -> d
                   _ -> b
  where
   c = d + 1
        d = a - b
```

So ist der `case`- bzw. `where`-Ausdruck in der Definition von `f` zwar ungewöhnlich, aber dennoch legitim eingerückt.

Die Einrückung eines Symbols, welches also kein Tabulator, Leerzeichen oder Zeilenumbruch ist, ist dabei gegeben durch die Spalte seines ersten Zeichens im Quelltext. Die Einrückung einer ganzen Zeile wiederum durch die Einrückung ihres ersten Symbols.

Die Layout-Regel wird nach jedem Vorkommen der Schlüsselwörter `let`, `where`, `do` oder `of` relevant. Für das erste darauf folgende Symbol wird die Einrückung gespeichert, und mit ihm eine Liste zugehöriger Einheiten für das jeweilige Schlüsselwort-Symbol begonnen. Die Einrückung der nächsten Zeile wird mit der gespeicherten verglichen. Sollte sie größer oder gleich der gespeicherten Einrückung sein, so wird sie der Liste hinzugefügt. Ist die Einrückung jedoch kleiner, so wird die Liste beendet.

4.5. XML und HTML

In Curry können XML- und HTML-Ausdrücke nativ dargestellt werden. Die gleichnamigen Module stellen für XML-Ausdrücke den Datentyp

```
data XmlExp = XText String
            | XElem String [(String,String)] [XmlExp]
```

und für HTML-Ausdrücke den Datentyp

```
data HtmlExp =
       HtmlText String
     | HtmlStruct String [(String,String)] [HtmlExp]
     | ...
```

bereit. Es ist deutlich zu sehen, dass `XmlExp` und `HtmlExp` äquivalente Typen beschreiben, wenn man `HtmlExp` auf `HtmlText` und `HtmlStruct` einschränkt.

So kann etwa das Beispiel 3.1 in Curry wie folgt dargestellt werden:

Beispiel 4.1: Ein Hyperlink zur Universität Kiel als `HtmlExp` in Curry

```
HtmlStruct "a"
  [("href","http://www.uni-kiel.de/")]
  [HtmlText "Christian-Albrechts-Universität zu Kiel"]
```

Die Curry-Module `XML` und `HTML` stellen darüber hinaus viele Operationen für das Einlesen sowie die Verarbeitung und Ausgabe der Daten zur Verfügung. Um aus einer Liste von `HtmlExp`-Ausdrücken einen HTML-Quelltext zu generieren, empfiehlt sich die Prozedur `showHtmlExps`:

```
showHtmlExps :: [HtmlExp] -> String
```

4.6. Codeintegration

In Curry existiert ein *Codeintegrator*[1], der es erlaubt, Quelltexte fremder, *domänenspezifischer* Sprachen in Curry-Quelltexte einzubetten, die vor der eigentlichen Übersetzung eines Curry-Programms in nativen Curry-Code übersetzt werden. Dies geschieht im Rahmen einer festen syntaktischen Vorschrift:

```
[ CURRY CODE ] ``L CL''[ CURRY CODE ]
```

Dabei ist L der Name bzw. Bezeichner einer formalen Sprache (z.B. *regex* für reguläre Ausdrücke) und C_L der zu übersetzende Quelltext der Sprache L.
Der *Codeintegrator* ist ein *Präprozessor*, der den Quelltext C_L an einen Parser für die Sprache L weitergibt. Der entsprechende Parser liefert dann einen repräsentativen Curry-Ausdruck für C_L, woraufhin dieser eingesetzt wird und die eigentliche Übersetzung des Curry-Programms beginnen kann:

```
import Regex
isLowerAlpha :: String -> Bool
isLowerAlpha s = s ``regex [a-z]*''
```

Hier wird der *Codeintegrator* den regulären Ausdruck `[a-z]*` *parsen* lassen und vor der Übersetzung des Programms folgenden Code erzeugen:

```
import Regex
isLowerAlpha :: String -> Bool
isLowerAlpha s = s `match` ([Star ([Bracket [Right (('a')
   ,('z'))]])])
```

[1] vgl. Bachelorthesis von Jasper Paul Sikorra: Codeintegration in Curry, 2014

5. Hamlet

Hamlet ist ein Haskell-Paket und gehört zur Familie der sogenannten *shakespearischen Templatesysteme*. Diese implementieren insbesondere die sogenannten Haskell-*QuasiQuotes*, mit denen sich *domänenspezifische* Sprachen in Haskell integrieren lassen, indem ihre Quelltexte zur Übersetzungszeit in Haskell-Quelltext umgewandelt werden. Das Paket Hamlet stellt *QuasiQuoter* zur Verfügung, mit denen HTML-Ausdrücke und ganze HTML-Dokumente in intuitiver Syntax formuliert werden können:

Beispiel 5.1: Beispiel eines Hamlet-Ausdrucks in Haskell

```
{-# LANGUAGE QuasiQuotes #-}
import Text.Hamlet (shamlet)
import Text.Blaze.Html.Renderer.String (renderHtml)

title = "Wau␣Holland"
color = "lightblue"
content = "&quot;...␣deshalb␣mag␣ich␣Binärtechnik.\
          \Da gibt es nur drei Zustände:\
          \High, Low und Kaputt.&quot;"

main :: IO ()
main = putStrLn $ renderHtml [shamlet|
<html>
  <head>
    <title>#{title}
  <body bgcolor=#{color}>
    <h1>#{title}
    #{content}
|]
```

Dabei ist `shamlet` vom Typ `QuasiQuoter` und liefert einen Ausdruck des Datentyps `Html`, der durch die Funktion `renderHtml` in einen HTML-Quelltext umgewandelt und schließlich mit `putStrLn` ausgegeben wird:

```
<html><head><title>Wau Holland</title>
</head>
<body bgcolor="lightblue"><h1>Wau Holland</h1>
&amp;quot;... deshalb mag ich Binärtechnik.Da gibt es nur
   drei Zustände:High, Low und Kaputt.&amp;quot;</body>
</html>
```

Es ist zu sehen, dass der durch `[shamlet|` und mit `|]` beendete Quelltext zwar *Start-Tags* der HTML enthält, die damit begonnenen Elemente jedoch nicht durch entsprechende *End-Tags* beendet werden. Außerdem können offenbar Haskell-Ausdrücke bzw. Haskell-Zeichenketten eingebettet werden.

5.1. Layout

Dass die Elemente bzw. ihr Inhalt trotz fehlender *End-Tags* eindeutig ausgezeichnet werden, wird in Hamlet durch eine eigene Layout-Regel sichergestellt. *End-Tags* sind daher optional für sogenannte *inline-Elemente*, w.z.B. `em`, `b` oder *Hyperlinks* `a`, und werden für alle anderen Elemente sogar ignoriert, d.h. sie werden als Text interpretiert.
Hamlets Layout-Regel ist der von Curry oder Haskell zwar ähnlich, hat jedoch einen markanten Unterschied. Sie greift zunächst immer unmittelbar nach einem *Start-Tag*. Anstatt dann aber nach einem nächsten, nicht-einrückenden Symbol zu suchen, wird der *Einzug* allein durch die singulär inkrementierte Einrückung des *Start-Tags* bestimmt:

```
   <body>
     Hello
    world
```

Die Layout-Regel ordnet dem Element `body` beide Zeichenketten zu, da ihre Einrückungen echt größer als die des Start-Tags sind. Andererseits wird der Absatz `p` in

```
<body>
  <h1>Title
  <p>
    Hello world
```

dem `body` und nicht mehr dem `h1` zugeordnet, weil es auf *gleicher Höhe*, d.h. in der gleichen Spalte wie das `h1` beginnt.

5.2. Interpolation

Wie in Beispiel 5.1 zu sehen ist, können in Hamlet bzw. im Quelltext der *QuasiQuoter* Haskell-Ausdrücke eingebettet werden. Dieser Vorgang ist *typsicher* und wird *Interpolation* genannt. Eingeleitet mit einer Raute, wird der zu interpolierende Ausdruck in geschweiften Klammern geschrieben:

```
[shamlet|<p>#{"Hello"++"␣"++p++"!"}|]
```

Für die Haskell-Variable `p = "Bob"` wäre der assoziierte HTML-Quelltext also:

```
<p>Hello Bob!</p>
```

Ebenso ist zu erkennen, dass Interpolation beispielsweise auch in Attributen stattfinden kann. Auch dies kann in vielen Situation sehr hilfreich sein und gewährleistet einen hohen Grad an Flexibilität.

Teil II.

Implementierung

6. Motivation

In Beispiel 4.1 haben wir einen `HtmlExp`-Ausdruck in Curry gesehen, der noch recht lesbar anmutete. Schauen wir uns nun ein weiteres Beispiel an:

```
<h1>Stadtstaaten</h1>
<ul>
  <li><em>Berlin</em></li>
  <li>Bremen</li>
  <li>Hamburg</li>
</ul>
```

Dieser Quelltext wird in Curry durch folgenden `HtmlExp`-Ausdruck repräsentiert:

```
[HtmlStruct "h1" [] [HtmlText "Stadtstaaten"],
 HtmlStruct "ul" []
   [HtmlStruct "li" [] [HtmlStruct "em" []
                          [HtmlText "Berlin"]],
    HtmlStruct "li" [] [HtmlText "Bremen"],
    HtmlStruct "li" [] [HtmlText "Hamburg"]]
```

Die Formulierung des spracheigenen Curry-Ausdrucks bedeutet also bereits in der Anzahl der verwendeten Zeichen einen deutlichen Mehraufwand. Hier noch wohlüberlegt eingerückt, kann diese Notation außerdem schnell demotivierend unübersichtlich ausufern.
Eine erste Erleichterung schafft das Modul HTML.curry durch viele abkürzende Funktionen. Dazu gehören etwa `h1` oder `emphasize`, die beide den Typ `[HtmlExp] -> HtmlExp` besitzen und eine Liste von `HtmlExp`-Ausdrücken in ein neues Element einbetten. Außerdem existieren `ulist :: [[HtmlExp]] -> HtmlExp` und `htxt :: String -> HtmlExp`. Es gelten beispielsweise `emphasize = HtmlStruct "em"[]` und `htxt = HtmlText` und wir können ein wenig vereinfachen:

```
[h1 [htxt "Stadtstaaten"],
 ulist [[emphasize [htxt "Berlin"]],
        [htxt "Bremen"],
        [htxt "Hamburg"]]
```

Dieser Ausdruck ist erheblich besser zu lesen – allerdings müssen wir die Abkürzungen kennen, können keine Attribute festlegen und müssen auf die richtige Klammerung achten. Auch werden Elementnamen in der XML frei gewählt, und so sind spracheigene Abkürzungen dort gar nicht möglich bzw. sinnvoll. Auch diese Form reicht also an die Prägnanz des ursprünglichen Ausdrucks nicht heran.

Wie wir in Kapitel 5 gesehen haben, darf in *Hamlet* ein nahezu native Syntax verwendet werden, die sich sogar weiter abkürzen lässt. Unübersichtliche Ausdrücke in spracheigenen Datenstrukturen gehören damit der Vergangenheit an.

Inspiriert durch *Hamlet* in Haskell und die *Codeintegration* in Curry (siehe 4.6) wäre es von Nutzen, wenn wir etwa folgendes schreiben könnten:

```
``html
  <h1>Stadtstaaten
  <ul>
    <li><em>Berlin
    <li>Bremen
    <li>Hamburg''
```

Da wir im Allgemeinen aber nicht bloß konstante Ausdrücke generieren, sondern außerdem auch in Curry berechnete Ergebnisse in die Ausgabe mit einfließen lassen wollen, sollen Zeichenketten aus Curry (i.e. `String`s) in geschweiften Klammern eingebettet bzw. *interpoliert* werden können. So soll der folgende Ausdruck das uns bekannte Ergebnis repräsentieren,

```
``html
  <h1>Stadtstaaten
  <ul>
    <li><em>{reverse xs}
    <li>Bremen
    <li>Hamburg''
```

falls `xs` im Namensraum zu `xs = "nilreB"` aufgelöst werden kann. In geschweiften Klammern dürfen also beliebige Curry-Ausdrücke geschrieben werden, die den Typ `String` besitzen.

Äquivalente Beispiele lassen sich natürlich in der XML formulieren und so wollen wir den Codeintegrator in Curry um die folgenden Formen erweitern:

```
[ CURRY CODE ] ``html C_html''[ CURRY CODE ]
```

```
[ CURRY CODE ] ``xml C_xml''[ CURRY CODE ]
```

Dabei werden in C_{html} Quelltexte erlaubt, welche beliebige HTML-Ausdrücke enthalten, wobei auf abschließende End-Tags verzichtet und eine Layout-Regel verwendet werden darf, die wir später spezifizieren. Außerdem sollen die besprochenen Curry-Quelltexteinbettungen erlaubt sein. Ebenso soll sich C_{xml} zur XML verhalten.

7. Entwurf

Die Auszeichnungssprachen XML und HTML bzw. `xml` und `html` sind sich syntaktisch und strukturell so ähnlich, dass es sich lohnt, einen Parser zu entwerfen, der beide Sprachen bzw. Quelltexte verarbeiten kann. Grundsätzlich soll er jede Eingabe akzeptieren. Anders als Quelltexte in Programmiersprachen, spezifizieren Quelltexte in Auszeichnungssprachen keine Programmlogik, sondern strukturieren Daten. Deshalb ist es üblich, während der Analyse von Auszeichnungssprachen lediglich Warnungen auszugeben und auf abbrechende Fehlermeldungen gänzlich zu verzichten.

Wir beginnen mit der Phase der *lexikalischen Analyse*:

1. *Zerleger*
   Der Zerleger zerlegt die Eingabezeichenkette an signifikanten Stellen in eine erste, flache Struktur von Zeichenketten, mit ihren Positionen (Zeile, Spalte) aus der Eingabe.

2. *Tokenizer*
   Der Tokenizer bestimmt nun die Art der einzelnen Zeichenketten und konvertiert sie in eine jeweils praktische Struktur, woraufhin sie die Bezeichnung *Token* verdienen. Dabei bedient er sich gesonderter *Tag-* und *Daten-Tokenizer*.

3. *Layouter*
   Der Layouter stellt fest, in welchen syntaktischen Abständen die *Token* zueinander stehen und hält diese fest.

Unmittelbar danach folgt die Phase der *syntaktischen Analyse*:

1. *Parser*
   Der Parser verarbeitet die Liste der *Token* in einen Baum bzw. in eine Liste von Bäumen, welche der Hierarchie der Auszeichnungssprache entsprechen. Sollte $L =$ `xml` gelten, wird das *Content-Model* (siehe 3.3) beachtet.

2. *Umsetzer*
   Der Umsetzer wandelt die entstandenen Bäume in `XmlExp`- bzw. `HtmlExp`-Ausdrücke um und gibt einen gültigen Curry-Ausdruck der entstandenen Liste als `String` zurück.

In Abb. 7.1 ist zu sehen, dass die Eingabe $C_L \in \{C_{xml}, C_{html}\}$ während der *lexikalischen Analyse* verarbeiten werden kann, ohne Kenntnis der Sprache $L \in \{\texttt{xml}, \texttt{html}\}$. Das liegt gerade an der großen Ähnlichkeit zwischen XML und HTML.

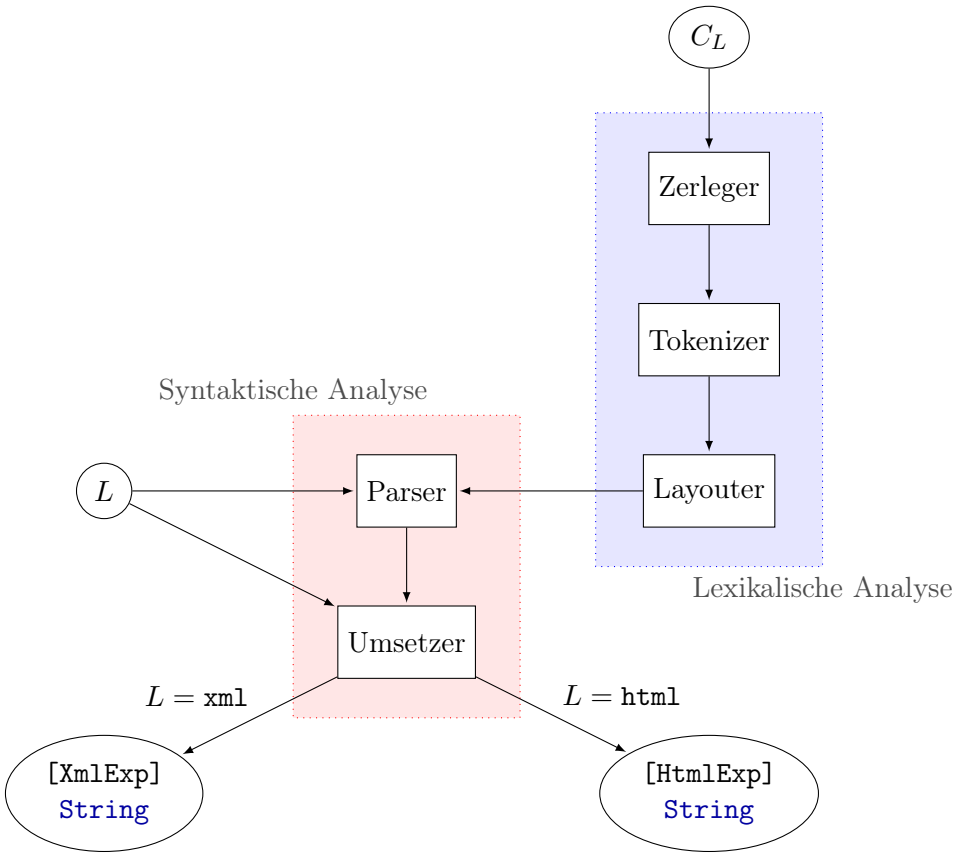


Abbildung 7.1.: Die Phasen der lexikalischen und syntaktischen Analyse

7.1. Tags

Wir verwenden grundsätzlich die Spezifikation der XML (siehe 2.2.2), da diese ebenso für HTML-Ausdrücke zutreffend ist. D.h. es werden sowohl *Start-*, als auch *Leer-* und *End-Tags* erlaubt. Ihre Attribute dürfen darüber hinaus jedoch auch mittels der *nicht angeführten Attributswerte* der HTML geschrieben werden (siehe 3.2).

7.2. Layout

Gewiss hat Hamlets Layout-Regel ihre Vorteile. Insbesondere ist ihre Implementierung sehr einfach und wir könnten sie auch zu unserem Zwecke einsetzen. Aus Gründen der Sprachkonsistenz wollen wir uns dennoch an eine Layout-Regel halten, die analog zu der von Curry formuliert wird:

Nach jedem Vorkommen eines *Start-Tags* eines Elements E, bestimmen wir dessen *Einzug* ϵ als die Einrückung[1] des nächsten Symbols, welches selbst keine Einrückung darstellt. Folgende Zeilen, deren Einrückung größer oder gleich ϵ ist, werden E zugeordnet, bis eine Zeile eine Einrückung besitzt, die echt kleiner als ϵ ist.

Bis hier her ist dies die transferierte Layout-Regel aus Curry. Zusätzlich werden jedoch durch ein *End-Tag* eines Elements E' rückwärts alle Elementinhalte vorhergehender *Start-Tags* beendet, bis ein *Start-Tag* des Elements E' gefunden und ebenfalls beendet wird. Hierfür betrachten wir ein Beispiel:

```
<div>
      <div>  <em>Inhalt
                 </div>
                 mehr
```

Die zusätzliche Vorschrift für *End-Tags* besagt hier, dass dem `em` nur die Zeichenkette `Inhalt` zugeordnet wird, bevor es durch das *End-Tag* `</div>` geschlossen wird, woraufhin auch das innere `div` geschlossen und die Zeichenkette `mehr` also dem äußeren Element `div` einbeschrieben wird. Der gesamte Ausdruck wird daher auch durch diesen, *echten* HTML-Ausdruck beschrieben:

```
<div>
  <div>
    <em>Inhalt</em>
  </div>
  mehr
</div>
```

[1] Der Begriff der Einrückung wird weiterhin wie im Layout von Curry verstanden (siehe 4.4).

7.3. Interpolation

Für die Interpolation von Curry-Ausdrücken wollen wir die geschweiften Klammern aus Hamlet übernehmen, aber auf die vorangestellte Raute[2] verzichten.
Beispielsweise soll der Ausdruck

```
``html <h1>{reverse "noisserpxE"}''
```

umgewandelt werden in den Ausdruck

```
[HtmlStruct "h1" [] [HtmlText (reverse "noisserpxE")]]
```

Damit öffnende geschweifte Klammern im *Codeintegrator* dennoch als einzelne Zeichen geschrieben werden können, die *nicht* den Beginn eines Ausdrucks spezifizieren, muss ihnen ein *Backslash* vorangestellt werden. So sollte

```
``html <h1>Opening Brace: \{''
```

umgeformt werden zu

```
[HtmlStruct "h1" [] [HtmlText "Opening␣Brace:␣{"]]
```

Andersherum müssen schließende geschweifte Klammern in Curry-Ausdrücken ebenfalls mit einem vorhergehenden *Backslash* geschützt werden, damit sie nicht als Ende des Ausdrucks interpretiert werden. Entsprechend sollte

```
``html <h1>Closing Brace: {"\}"}''
```

das Ergebnis

```
[HtmlStruct "h1" [] [HtmlText "Closing␣Brace:␣}"]]
```

liefern. Ebenfalls wie in Hamlet, soll Interpolation auch innerhalb aller Attributswerte möglich sein. So fordern wir, dass

```
``html <img src="images/{filename}" />''
```

in den Ausdruck

```
[HtmlStruct "img" [("src","images/"++filename)] []]
```

umgesetzt wird.

[2] Hamlet besitzt mehrere *Interpolationsoperatoren*, die mit unterschiedlichen Zeichen beginnen.

8. Implementierung

Im Folgenden werden die Phasen der *lexikalischen* und der *syntaktischen Analyse* in ihren einzelnen Schritten entwickelt.

8.1. Lexikalische Analyse

In dieser Phase werden wir Schritt für Schritt einen *Lexikalischen Scanner* (auch *Lexer*) entwickeln, der die Eingabe in eine Liste lexikalischer Einheiten, sogenannter *Token*, zerlegen und sie mit ihrer ursprünglichen Position `TPos` aus der Eingabezeichenkette versehen wird:

```
type SimplePos = (Int,Int)
type TPos = (SimplePos,Int)
```

`SimplePos` beschreibt ein geordnetes Paar, bestehend aus einem Zeilen- und einem Spaltenindex. Darauf aufbauend kann in `TPos` zusätzlich die Anzahl aller – in der gleichen Zeile – vorangestellten Tabulatoren gespeichert werden[1].
Nun deklarieren wir den Lexer wie folgt:

```
lex :: String -> TPos -> ([Symbol],[Warning])
```

Ein `Symbol` ist dabei ein geordnetes Paar, bestehend aus einem `Token` und einer Position `TPos`, was wir später genauer besprechen. Eine *Warnung* ist wiederum ein geordnetes Paar, bestehend aus einer Position und einer Zeichenkette, die die Warnung bzw. ihre Ursache formuliert. Sie entspricht damit dem folgenden Typsynonym:

```
type Warning = (TPos,String)
```

Es ist zu sehen, dass der Lexer außer der eigentlichen Eingabe (`String`) noch eine Position (`TPos`) verlangt. Diese teilt uns den Beginn des Quelltexts C_L im ursprünglichen Curry-Quelltext mit, damit wir absolute Positionen berechnen und dadurch auch positionsexakte Warnungen erzeugen können. Diese *Startposition* bezeichnen wir von nun an mit `start :: TPos`.

Üblicherweise wird ein *Lexer* durch *Deterministische endliche Automaten* modelliert.

[1] Die vorangestellten Tabulatoren sind später für die Layout-Regel von Interesse, da Tabulatoren dort schwerer gewichtet werden als andere Zeichen.

Definition. *Ein Deterministischer endlicher Automat (DEA) ist ein Quintupel* $\mathfrak{A} = (Q, \Sigma, \delta, q_0, F)$ *mit den Eigenschaften:*

- Q *ist eine endliche Zustandsmenge*
- Σ *ist ein endliches Eingabealphabet*
- $\delta : Q \times \Sigma \to Q$ *ist eine Transitionsfunktion*
- $q_0 \in Q$ *ist ein Startzustand*
- $F \subseteq Q$ *ist eine Menge akzeptierender Zustände (auch Endzustände)*

Solche Automaten werden wir nutzen, um Zeichenketten in einen Zieltyp T umzuwandeln:

```
deaGenerate :: String -> T
```

Hierfür setzen wir $Q := \{0, \ldots, n\}$ für ein $n \in \mathbb{N}$, $q_0 := 0$ und $\Sigma :=$ `Char` $= \{c_0, \ldots, c_m\}$ für ein $m \in \mathbb{N}$, wobei Σ also die endliche Menge aller Zeichen (in Curry) darstellt. Da wir eine Auszeichnungssprache übersetzen und jede denkbare Eingabe ein Ergebnis liefern soll, definieren wir alle Zustände als akzeptierend, i.e. $F = Q$. So können wir mit einer Funktion `f :: Int -> Char -> T -> T` eine allgemeine Form bestimmen:

```
deaGenerate xs = dea 0 xs
  where dea :: Int -> String -> T -> T
        dea _ [] y = y
        dea q (c:cs) y
          | q == 0 && c == c_0 = dea δ(0,c_0) cs (f q c y)
          | ...
          | q == 0 && c == c_m = dea δ(0,c_m) cs (f q c y)
          | ......
          | q == n && c == c_0 = dea δ(n,c_0) cs (f q c y)
          | ...
          | q == n && c == c_m = dea δ(n,c_m) cs (f q c y)
```

Das Ergebnis eines Aufrufs wird also *zustandsabhängig* in der Variablen `y` akkumuliert, und so können wir einen Automaten verwenden, um eine Zeichenkette in einen Typ unserer Wahl umzusetzen. In konkreten Implementierungen wird man im Allgemeinen für einen Zustand q aber nicht alle $c \in \Sigma$ unterschiedlich behandeln – die Anzahl der *Guards* ist also meist erheblich kleiner als $m \cdot n$.

8.1.1. Zerlegung

Zunächst zerlegen wir die Eingabe an markanten Stellen, um so die Grundlage des *Tokenizers* zu gewinnen. Wir definieren einen *Zerleger* folgender Art:

```
breakup :: String -> TPos -> [(String,TPos)]
```

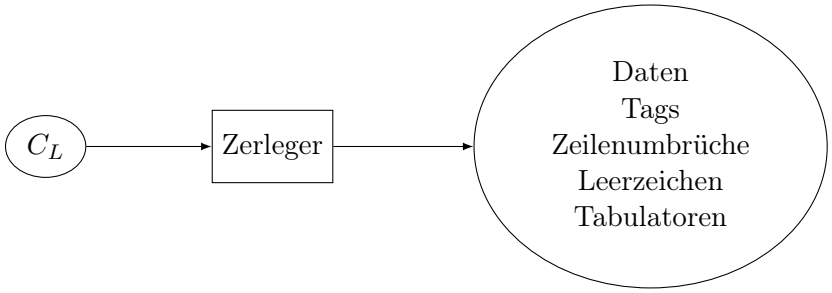


Abbildung 8.1.: Eingabezerlegung als erster Schritt des lexikalischen Scanners

Dieser soll, mathematisch formuliert, eine sogenannte *Partition* der Eingabe bestimmen, wobei jede einzelne Zeichenkette mit der Position ihres ersten Zeichens versehen wird. Die Konkatenation all dieser Zeichenketten ergäbe also wieder die Eingabezeichenkette. Auf jeden Fall müssen Daten bzw. Text identifiziert werden. Damit wir die Daten dann in die intendierte Struktur bringen können, müssen die sie umschließenden Tags erkannt werden. Ob allerdings ein Start- oder ein End-Tag vorliegt, ist hier noch nicht von Interesse[2]. Außerdem benötigen wir für die Umsetzung der Layout-Regel jegliche Einrückungen, d.h. Leerzeichen und Tabulatoren, falls diese durch einen Zeilenumbruch oder das Ende eines (Start-)Tags eingeleitet werden. Zusätzlich wird jeder Zeilenumbruch isoliert.

Die Eingabe wird also, wie in Abb. 8.1 zu sehen, in Daten, Tags, Leerzeichen, Tabulatoren und Zeilenumbrüche zerlegt. Damit diese wohlunterscheidbar sind, spezifizieren wir:

- *Daten*
  Jegliche zusammenhängende Zeichen, die keine Tags oder Zeilenumbrüche enthalten und nicht nur aus Leerzeichen oder nur aus Tabulatoren bestehen
- *Tags*
  Tags beginnen mit einem < und enden mit einem > – außerdem folgt darin auf jedes einfache oder doppelte Anführungszeichen ein jeweilig gleiches Anführungszeichen
- *Leerzeichen und Tabulatoren*
  Zusammenhängende Leerzeichen oder zusammenhängende Tabulatoren, die unmittelbar nach einem Tag oder Zeilenumbruch stehen

Praktischerweise modellieren wir den *Zerleger* als *Deterministischen endlichen Automaten* (siehe Abb. 8.3). Die gestrichelten Transitionen kennzeichnen, dass das gelesene Zeichen noch nicht konsumiert bzw. verarbeitet, sondern weitergegeben wird, und dessen Verarbeitung also dem Folgezustand verbleibt. Die Sterne bezeichnen die Menge aller Zeichen, für die keine (andere) Transition existiert.

[2]Die Identifizierung von Start- und End-Tags sowie eventuellen Attributen soll erst im nächsten Schritt durch den *Tokenizer* vorgenommen werden.

Name	Dez	ASCII	Name	Dez	ASCII	Name	Dez	ASCII
HT	9	\t	SQ	39	'	GT	62	>
BR	10	\n	SL	47	/	BS	92	\
BL	32	␣	LT	60	<	OB	123	{
DQ	34	"	EQ	61	=	CB	125	}

Abbildung 8.2.: Wichtige Zeichen für die Phase der *lexikalischen Analyse*

Nehmen wir eine simple Beispieleingabe an:

```
<a href="#">
  Hello World!
```

Der *Zerleger* wird nun die folgenden Zustände durchlaufen:

1. q_0 : LT $\rightarrow$ q_1 $\rightarrow$ ein neues *Tag* wird begonnen
2. q_1 : BL,'h','r','e','f','=' $\rightarrow$ q_1
3. q_1 : DQ $\rightarrow$ q_5 $\rightarrow$ ein Attributswert hat begonnen
4. q_5 : '#' $\rightarrow$ q_5
5. q_5 : DQ $\rightarrow$ q_1 $\rightarrow$ der Attributswert ist zuende
6. q_1 : GT $\rightarrow$ q_2 $\rightarrow$ das *Tag* wird beendet

An dieser Stelle wechseln wir in die Zustände der Einrückung, damit später der Einzug der Start-Tags bestimmt werden kann.

7. q_2 : BR $\rightarrow$ q_0 $\rightarrow$ BR wird nicht konsumiert, sondern weitergegeben
8. q_0 : BR $\rightarrow$ q_2 $\rightarrow$ ein Zeilenumbruch wird isoliert
9. q_2 : BL $\rightarrow$ q_3 $\rightarrow$ eine neue Leerzeichensequenz wird begonnen
10. q_3 : BL $\rightarrow$ q_3
11. q_3 : 'H' $\rightarrow$ q_2 $\rightarrow$ die Leerzeichensequenz wird beendet
    $\rightarrow$ 'H' wird nicht konsumiert, sondern weitergegeben
12. q_2 : 'H' $\rightarrow$ q_0 $\rightarrow$ 'H' wird nicht konsumiert, sondern weitergegeben
13. q_0 : 'H' $\rightarrow$ q_0 $\rightarrow$ neues Datum wird begonnen
14. q_0 : 'e','l','l','o',BL,'W','o','r','l','d','!' $\rightarrow$ q_0
15. q_0 : $\rightarrow$ stop $\rightarrow$ das Datum wird beendet

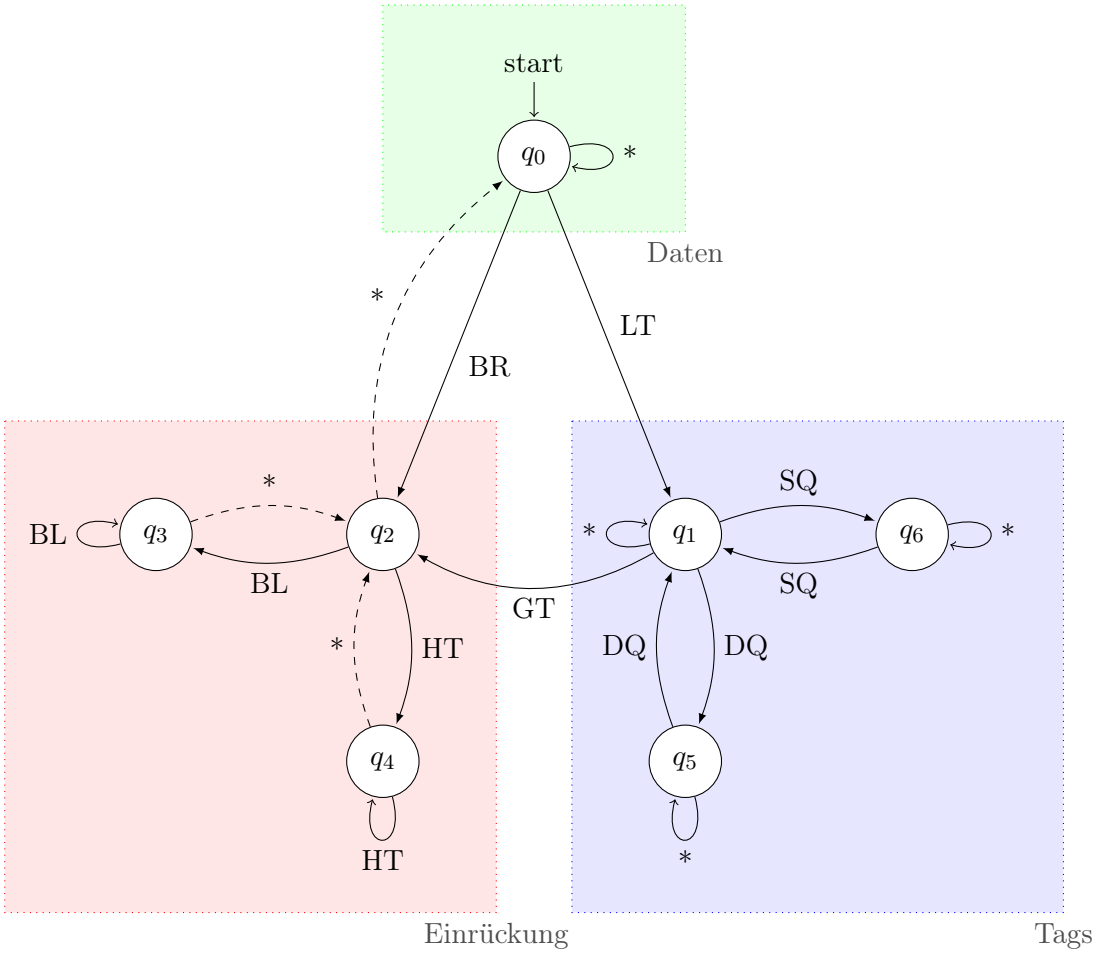


Abbildung 8.3.: Der Zerleger als Deterministischer endlicher Automat (DEA)

Auf diese Weise wird die folgende Ergebnisliste konstruiert:

```
[("<a␣href=\"#\">",p1),("\n",p2),
 ("␣␣",p3),("Hello␣World!",p4)]
```

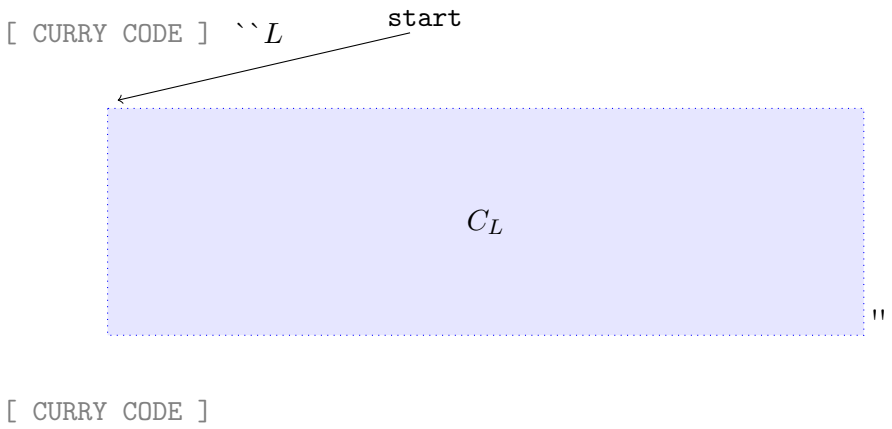
Dabei bezeichnen $p_1, \ldots, p_4$ die Positionen des jeweils ersten Zeichens. Sollte die Startposition `start :: TPos` also beispielsweise `((17,5),0)` sein, so lautete das Ergebnis:

```
[("<a␣href=\"#\">",((17,5),0)),("\n",((29,5),0)),
 ("␣␣",(17,6)),("Hello␣World!",((19,6),0))]
```

Die Positionen werden berechnet, indem eine Laufvariable `next@((i,j),t)`, die die Position des aktuell betrachteten bzw. ersten Zeichens `c` der noch zu verarbeitenden Eingabe bezeichnet, durch eine einfache Funktion und die Startposition `start@((a,b),0)` verändert wird:

$$((i,j),t) \mapsto \begin{cases} ((i,j+1),t+1) & c = HT \\ ((i+1,b),0) & c = BR \\ ((i,j+1),t) & sonst \end{cases}$$

Dass die neue Spalte nach einem Zeilenumbruch wieder die Startspalte `b` ist, liegt daran, dass der *Codeintegrator* selbst die Layout-Regel aus Curry umsetzt, und mehrzeilige Quelltexte daher immer nach folgendem Schema an den zuständigen Parser übergeben werden:



Genauer gesagt, beginnt der Quelltext C_L mit dem ersten Zeichen, das sich von Leerzeichen, Tabulatoren und Zeilenumbrüchen unterscheidet, und die Startposition für den Parser wird als die Position dieses Zeichens festgelegt. Die erste Spalte ist demnach in jeder Zeile von C_L die Startspalte.

Nun haben wir eine Liste des Typs `[(String,TPos)]`, die weitergegeben wird an den *Tokenizer*.

8.1.2. Tokenizer

Der *Tokenizer* bestimmt die Art der Zeichenketten, die der lexikalische Scanner liefert, und übersetzt sie in sogenannte *Token*. Dafür benötigen wir einige Datentypen:

```
data Text = Raw String | Exp String
type Attribute = (String,[Text])
```

Ein `Text` ist also entweder *roh* (`Raw String`), d.h. er bezeichnet nicht mehr und nicht weniger als eine Zeichenkette, oder ein Curry-Ausdruck (`Exp String`), der ebenfalls in einer Zeichenkette steht.

```
data Token = Break
           | Tabs Int
           | Blanks Int
           | Data [Text]
           | StartTag String [Attribute] Int
           | VoidTag String [Attribute]
           | EndTag String
```

Ein Token ist demnach entweder ein Zeilenumbruch (`Break`), eine Anzahl aufeinander folgender Tabulatoren (`Tabs Int`) oder Leerzeichen (`Blanks Int`), ein Datum bzw. eine Liste des Typs `Text`, ein Start-Tag mit einem Namen, einer Liste von Attributen und einem Einzug (`StartTag String [(String,String)] Int`), ein Leer-Tag mit einer Liste von Attributen und einem Namen (`VoidTag String [(String,String)]`) oder ein End-Tag mit einem Namen (`EndTag String`).
Das Ergebnis des *Tokenizers* sind dann Paare des Typs `(Token,TPos)`, die wir *Symbole* nennen:

```
type Symbol = (Token,TPos)
```

Der *Tokenizer* ist die erste Instanz, die Verstöße gegen die *Wohlgeformtheit*[3] identifizieren kann und diese in Form von Warnungen mitteilt. Er wird folgendermaßen beschrieben:

```
tokenize :: [(String,TPos)] -> ([Symbol],[Warning])
```

Da das Beispiel aus 8.1 all unseren Regeln entspricht, soll es keine Warnungen erzeugen und von `tokenize` in den folgenden Ausdruck übersetzt werden:

```
([(StartTag "a" [("href",[Raw "#"])] 0,((17,5),0)),
  (Break,((29,5),0)),(Blanks 2,((17,6),0)),
  (Data [Raw "Hello␣World!"],((19,6),0))],[])
```

Es ist zu sehen, dass der *Einzug* des *Start-Tags* `a` mit 0 angegeben ist. Diese Festlegung soll der *Tag-Tokenizer* auf alle *Start-Tags* anwenden, obwohl es offensichtlich falsch ist; denn das erste nachfolgende `Token`, das keine Einrückung darstellt, ist `Text "Hello␣World!"`, welches in Spalte 19 beginnt. Daher sollte der *Einzug* für `a` eigentlich mit 19 bestimmt werden. Diesen Missstand wird im nächsten Schritt der *Layouter* auflösen.
Das eigentliche Problem dieses Schritts stellen die *Tags* dar – alle anderen Zeichenketten lassen sich sehr leicht konvertieren. Für diese können wir erst einmal einfache Prädikate `isBreak`, `isTabs`, `isBlanks`, und `isData` definieren:

```
isBreak (c:cs) = c == BR && null cs
isTabs = all (HT==)
isBlanks = all (BL==)
isData (c:_) = c /= LT
```

[3] Man nennt Ausdrücke oder auch ganze Dokumente einer Auszeichnungssprache *wohlgeformt*, falls alle formalen Regeln der Sprache eingehalten werden.

Diese haben alle den Typ `[_] -> Bool`, wobei `isTabs` und `isBlanks` durch die vordefinierte Funktion `all :: (a -> Bool)-> [a] -> Bool` definiert werden, die prüft, ob ein Prädikat auf alle Elemente einer Liste zutrifft.
Damit können wir Paare des Typs `(String,TPos)`, die einen Zeilenumbruch, Tabulatoren, Leerzeichen oder einen Curry-Ausdruck repräsentieren, wie folgt umwandeln:

```
tk :: (String,TPos) -> Symbol
tk (s,pos) | isBreak s  = (Break,pos)
           | isTabs s   = (Tabs (length s),pos)
           | isBlanks s = (Blanks (length s),pos)
           | isData s   = (Data [Raw s],pos)
```

Das Prädikat `isData` ist eigentlich nicht korrekt, denn es verifiziert beispielsweise auch einen Zeilenumbruch als Text. Da wir es aber als letztes abfragen, ist das Verhalten wie erwartet. Jeder noch übrige Fall einer Zeichenkette sollte nun ein *Tag* sein. Es fehlt also noch ein *Tokenizer* für *Tags*. Allerdings haben wir eventuelle Curry-Ausdrücke in Daten noch nicht beachtet – stattdessen speichern wir Daten als einzelne *rohe* Zeichenkette. Ebenso wird der *Tag-Tokenizer* mit Attributswerten verfahren, denn dann können wir in einem dritten und letzten *Daten-Tokenizer* sowohl inhaltliche Daten als auch Attributswerte in tatsächlich rohe Daten und Curry-Ausdrücke zerlegen.

8.1.3. Tag-Tokenizer

Der *Tag-Tokenizer* wird vom *Tokenizer* eingesetzt, um zu prüfen, ob eine Zeichenkette ein *Tag* repräsentiert, und sie dann gegebenenfalls in ein entsprechendes `Token` zu übersetzen:

```
tagTokenizer :: String -> TPos -> (Token,[Warning])
```

Die dabei anfallenden Warnungen fügt der *Tokenizer* den bestehenden hinzu.
Diese Aufgabe erledigen wir erneut mit Hilfe eines Deterministischen endlichen Automaten, der in Abb. 8.4 zu sehen ist. Dass dieser Automat ganze 17 Zustände braucht, wurzelt darin, dass die Spezifikation der *Tags* (siehe 2.2) – insbesondere gegenüber Leerzeichen in Verbindung mit Attributen – eine Vielzahl verschiedener Eingaben erlaubt.
So ist etwa folgender HTML-Ausdruck wohlgeformt:

```
<p style    ="color:␣green;"  class name=    joke>
 Sei Epsilon kleiner Null.
</p>
```

Der so spezifizierte Absatz hat einen *leeren* Klassennamen, i.e. `class=""`, während sein Name durch die Zeichenkette `joke`, i.e. `name="joke"`, angegeben ist.

Auch dieser Automat akzeptiert die Eingabe in jedem Zustand und gibt gegebenenfalls Warnungen aus. In diesem Fall gibt es jedoch drei Zustände, die als die eigentlichen Endzustände zu verstehen sind. So ist q_9 etwa der Zustand, in dem ein *Start-Tag* beendet sein sollte, in q_{10} finden *Leer-Tags* ihr Ende und in q_{13} werden *End-Tags* besiegelt. Zusätzlich zur bereits bekannten Notation des *Zerlegers* (siehe 8.1.1), werden alphabetische

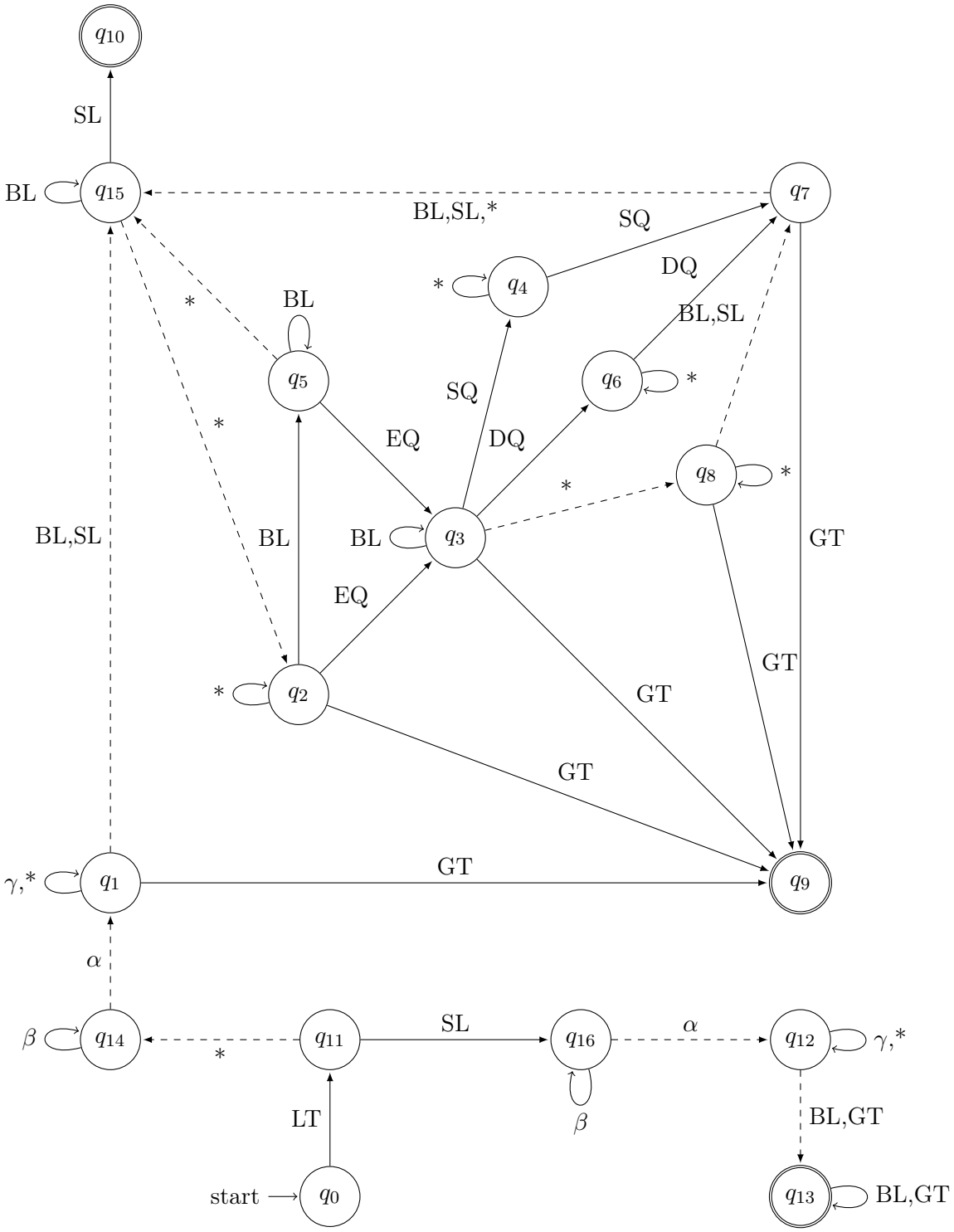


Abbildung 8.4.: Der *Tag-Tokenizer* als Deterministischer endlicher Automat (DEA)

Buchstaben mit einem α, Ziffern mit einem β und alphanumerische Zeichen mit einem γ bezeichnet.
Unsere Beispielinstanz enthielt die Zeichenkette `<a href="#">`, die das *Start-Tag* des Elements `a` spezifiziert. Der *Tag-Tokenizer* verarbeitet diese so:

1. q_0 : LT $\rightarrow$ q_{11}
2. q_{11} : 'a' $\rightarrow$ q_{14} $\rightarrow$ 'a' wird nicht konsumiert, sondern weitergegeben
3. q_{14} : 'a' $\rightarrow$ q_1 $\rightarrow$ 'a' wird nicht konsumiert, sondern weitergegeben
4. q_1 : 'a' $\rightarrow$ q_1 $\rightarrow$ mit a wird der *Tagname* begonnen
5. q_1 : BL $\rightarrow$ q_{15} $\rightarrow$ der *Tagname* wird beendet
6. q_{15} : 'h' $\rightarrow$ q_2 $\rightarrow$ wird nicht konsumiert und weitergegeben
7. q_2 : 'h','r','e','f' $\rightarrow$ q_2 $\rightarrow$ ein Attributsname wird begonnen
8. q_2 : EQ $\rightarrow$ q_3 $\rightarrow$ der Attributsname wird beendet
9. q_3 : DQ $\rightarrow$ q_6
10. q_6 : '#' $\rightarrow$ q_6 $\rightarrow$ mit '#' wird ein Attributswert begonnen
11. q_6 : DQ $\rightarrow$ q_7 $\rightarrow$ der Attributswert wird beendet
12. q_7 : GT $\rightarrow$ q_9
13. q_9 : $\rightarrow$ stop $\rightarrow$ das *Tag* wird als *Start-Tag* beendet

Wie wir gefordert hatten, entsteht dabei dieses `Token`:

```
StartTag "a" [("href",[Raw "#"])] 0
```

Tatsächlich wird der *Tag-Tokenizer* jeden Attributswert als *rohen* Text in einer einelementigen Liste ablegen. Ob sich darin Curry-Ausdrücke verstecken, wird nun der *Daten-Tokenizer* feststellen.

8.1.4. Daten-Tokenizer

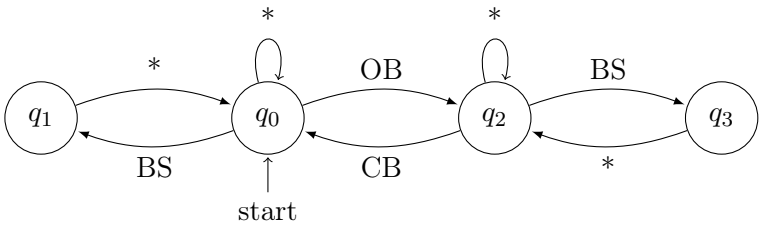
Zu diesem Zeitpunkt sind eigentlich bereits alle Zeichenketten in `Token` umgeformt worden. Allerdings können in Elementinhalten und Attributswerten noch Curry-Ausdrücke enthalten sein, die noch nicht detektiert wurden. Diese gilt es nun zu finden und von von den *rohen* Zeichenketten abzuheben. Da auch der *Daten-Tokenizer* von unserem *Tokenizer* eingesetzt wird, gibt es zu einzelnen Symbolen eventuell bereits Warnungen, die nun noch erweitert werden können:

```
dataTokenizer :: (Symbol,[Warning]) -> (Symbol,[Warning])
```

Die eigentlich interessanten Symbole bzw. Token sind nur *Start-Tags* (`StartTag`), *Leer-Tags* (`VoidTag`) und Elementinhalte in Form von Daten (`Data`). Alle anderen Symbole können ignoriert werden, d.h. sie werden unverändert zurückgegeben. Ferner liegen die relevanten Informationen dieser Symbole immer in einelementigen Liste mit einem *rohen* Text (`Raw String`) vor. Diese Zeichenkette wird nun in eine Liste des Typs `[Text]` umgewandelt, die dann möglicherweise auch Curry-Ausdrücke (`Exp String`) enthält, und die vormals einelementige Liste durch diese neue ersetzt. Dafür definieren wir folgende Funktion:

```
dtk :: String -> [Text] -> [Text]
```

Glücklicherweise haben wir die Syntax für die Interpolation (siehe 7.3) so einfach festgelegt, dass der nun notwendige *DEA* mit nur vier Zuständen auskommt:



In den Zuständen q_0, q_1 wird nun *Rohtext* erkannt, wohingegen q_2 und q_3 für Ausdrücke zuständig sind. Im Zustand q_1 wird dafür gesorgt, dass öffnende geschweifte Klammern (OB) im Rohtext mit einem *Backslash* (BS) geschützt werden können, während q_3 die schließenden, geschweiften Klammern (CB) in Ausdrücken schützt. Das Beispiel

```
[Raw "\{Ausgabe:{output}"]
```

wird folgendermaßen verarbeitet:

1. q_0 : BS $\rightarrow q_1$
2. q_1 : OB $\rightarrow q_0$ $\rightarrow$ mit OB wird ein Rohtext begonnen und das BS verworfen
3. q_0 : 'A','u','s','g','a','b','e',':' $\rightarrow q_0$ $\rightarrow$ der Rohtext wird erweitert
4. q_0 : OB $\rightarrow q_2$ $\rightarrow$ der Rohtext wird beendet und ein Ausdruck wird begonnen
5. q_2 : 'o','u','t','p','u','t' $\rightarrow q_2$ $\rightarrow$ der Ausdruck wird erweitert
6. q_2 : CB $\rightarrow q_0$ $\rightarrow$ der Ausdruck wird beendet
7. q_0 : $\rightarrow$ stop

Entsprechend sieht nun die Ergebnisliste aus:

```
[Raw "{Ausgabe:",Exp "output"]
```

So entsteht die erwünschte Liste, die neben Rohdaten nun auch Ausdrücke enthalten kann. Damit ist der *Daten-Tokenizer* und somit auch der gesamte *Tokenizer* fertiggestellt. Wir können anfangen, uns mit dem *Layout* zu beschäftigen.

8.1.5. Layouter

Der *Tokenizer* hat die *Start-Tags* aus der Eingabe nun zwar in `Token` der Art `StartTag String [(String,String)] Int` umgeformt, allerdings ist dabei der Einzug initial mit 0 bestimmt worden. Damit der *Parser* im nächsten Schritt auch die Layout-Regel wie gewünscht umsetzen kann, ist es die Aufgabe des *Layouters*, den Einzug eines jeden *Start-Tags* richtig zu bestimmen.
Dafür werden nun die *einrückenden* Symbole bzw. Token interessant, die wir überspringen müssen, um das erste *nicht-einrückende* Symbol zu finden, welches mit der Spaltenzahl seiner Position den Einzug des *Start-Tags* bestimmt. Um *einrückende* und *nicht-einrückende* Symbole unterscheiden zu können, definieren wir ein Prädikat `isAlign`:

```
isAlign :: Symbol -> Bool
isAlign sym = case tok sym of
                    Break    -> True
                    Tabs _   -> True
                    Blanks _ -> True
                    _        -> False
```

Wie gewünscht wird `isAlign` alle Zeilenumbrüche, Tabulatoren und Leerzeichen als *einrückend* und alle anderen Symbole als *nicht-einrückend* ausweisen.

Damit können wir den *Layouter* beschreiben, durch eine Funktion

```
layout :: ([Symbol],[Warning]) -> ([Symbol],[Warning])
```

Warnungen wird der *Layouter* keine generieren, weil die Einzüge der *Start-Tags* zwar festgestellt, aber noch nicht beurteilt werden. Deshalb werden die bisher angefallenen Warnungen unverändert weitergegeben:

```
layout (input,ws) = (join input,ws)
  where
    join :: [Symbol] -> [Symbol]
    join [] = []
    join (x:xs)
      | isStartTag x = joiner x xs []
      | otherwise = x : join xs
    joiner :: Symbol -> [Symbol] -> [Symbol] -> [Symbol]
    joiner t [] tmp = t : reverse tmp
    joiner t@(StartTag s a _,p) (x:xs) tmp
      | isAlign x = joiner t xs (x:tmp)
      | otherwise = (:) (StartTag s a (wcol (pos x)),p)
                        (reverse tmp) ++ join (x:xs)
```

Mit der Funktion `join` durchsuchen wir die `Symbol`-Eingabeliste also nach *Start-Tags*, wobei der eigentlich interessante Teil dieses überschaubaren Algorithmus' in der Funktion

`joiner` steckt, da diese nur zum Einsatz kommt, falls ein `StartTag` gefunden wurde. Dieses halten wir im ersten Parameter `t` fest, und suchen daraufhin nach dem ersten Vorkommen eines nicht-einrückenden Symbols in der Symbol-Restliste. In der Liste `tmp` werden alle dabei übersprungenen Symbole festgehalten, damit die Liste wieder richtig zusammengesetzt werden kann.
Ist ein *nicht-einrückendes* `Symbol x` gefunden, so wird das alte `StartTag` durch ein neues ersetzt, welches dem alten bis auf den Einzug gleicht. Dieser wird als die *gewichtete Spaltenzahl der Position* von `x` festgelegt.
Die *gewichtete Spaltenzahl einer Position* wird durch die Funktion `wcol` berechnet:

```
wcol :: TPos -> Int
wcol p = col p + 7 * tbs p
```

Diese Festlegung ist nicht intuitiv. Dahinter steckt die verbreitete Konvention, dass ein Tabulator als Einrückung wie acht gewöhnliche Zeichen interpretiert wird[4]. Haben wir also eine Spaltenzahl s einer Zeile, in der vor dem s-ten Zeichen t Tabulatoren stehen, und bezeichnen wir die *gewichtete* Spaltenzahl mit g, so gilt die Gleichung

$$g = (s - t) + 8t = s + 7t$$

Da nun `col :: TPos -> Int` durch `col = snd . fst` gerade die Spalte einer Position und `tbs :: TPos -> Int` durch `tbs = snd` die Anzahl der in der gleichen Zeile vorhergehenden Tabulatoren bestimmt, berechnet `wcol` (für *weighted column*) die *gewichtete Spaltenzahl einer Position*.

Um das Eingabebeispiel der vorhergehenden Schritte fortzuführen, sollte

```
([(StartTag "a" [("href",[Raw "#"])] 0,((17,5),0)),
  (Break,((29,5),0)),(Blanks 2,((17,6),0)),
  (Data [Raw "Hello␣World!"],((19,6),0))],[])
```

von `layout` nun lediglich umgewandelt werden in:

```
([(StartTag "a" [("href",[Raw "#"])] 19,((17,5),0)),
  (Break,((29,5),0)),(Blanks 2,((17,6),0)),
  (Data [Raw "Hello␣World!"],((19,6),0))],[])
```

Nun ist der letzte Schritt getan, um das *Parsen* zu beginnen.

8.2. Syntaktische Analyse

Alle bisherigen Schritte haben die Eingabe in einer Liste einfacher Entitäten, d.h. in einer *flachen* Struktur belassen. Die *hierarchische* Struktur der Auszeichnungssprachen XML und HTML verlangt jedoch nach einer *Baumstruktur*, d.h. die *Tags* müssen nun zu Elementen zusammengesetzt werden. Dafür muss festgestellt werden, was diese enthalten, und an dieser Stelle kommt die *Layout-Regel* ins Spiel.

[4] Diese Tabulatorengewichtung findet sich beispielsweise im *Haskell Report* oder im *Curry Report*.

8.2.1. Parser

Der Parser soll die Symbolliste in eine Liste von Bäumen verarbeiten, die der *hierarchischen* Struktur der Eingabeausdrücke entsprechen:

```
parse :: Lang -> ([Symbol],[Warning])
         -> ([Tree],[Warning])
```

Der Datentyp `data Lang = Xml | Html` des ersten Parameters `lang` verrät, ob es sich bei der Sprache L um `xml` oder `html` handelt. Der Typ `Tree` hingegen wird folgendermaßen zusammengesetzt:

```
data Node = Content [Text]
          | Element String [Attribute]
data Tree = Tree Node [Tree]
```

Ein Knoten besteht also entweder aus einem *Elementinhalt*, d.h. einer Liste des Typs `[Text]`, oder aus einem *Element* mit einem Namen und einer Liste von Attributen. Ein Baum ist nun ein Knoten mit einer Liste von Teilbäumen. Damit Symbole tatsächlich in Knoten verwandelt werden können, definieren wir die Funktion `sym2node`:

```
sym2node :: Symbol -> Node
sym2node x = case tok x of
               Break          -> Content [Raw "\n"]
               Tabs n         -> Content [Raw (tabs n)]
               Blanks n       -> Content [Raw (blanks n)]
               Data ds        -> Content ds
               VoidTag s a    -> Element s a
               StartTag s a _ -> Element s a
  where blanks :: Int -> String
        blanks = flip take $ repeat BL
        tabs :: Int -> String
        tabs = flip take $ repeat HT
```

Wir implementieren den *Parser* als sogenannten *Kellerautomaten* (auch *Stapelautomat*). Dadurch können wir auch die *Wohlgeformtheit* eines Ausdrucks beurteilen und Verstöße dagegen durch weitere Warnungen anzeigen. Damit dies gelingt, benötigen wir einen Datentyp für *Stapel*. Sehr einfach und effizient ist hier ein Typsynonym für Listen:

```
type Stack a = [a]

push :: a -> Stack a -> Stack a
push = (:)
top :: Stack a -> a
top = head
pop :: Stack a -> Stack a
pop = tail
```

Das oberste Element eines Stapels verändern wir mit `update`:

```
update :: (a -> a) -> Stack a -> Stack a
update f (x:xs) = (f x) : xs
```

Das uns begleitende Beispiel aus der *lexikalische Analyse*

```
([(StartTag "a" [("href",[Raw "#"])] 19,((17,5),0)),
  (Break,((29,5),0)),(Blanks 2,((17,6),0)),
  (Data [Raw "Hello␣World!"],((19,6),0))],[])
```

sollte durch den Parser in die folgende Struktur verarbeitet werden:

```
([Tree (Element "a" [("href",[Raw "#"])])
       [Tree (Content [Raw "Hello␣World!"])
             []]],[])
```

Damit dieses Verhalten erreicht wird, assoziieren wir nun in einem Stapel zu jedem gefundenen *StartTag* eine Liste von Bäumen. Diese Bäume soll das mit einem *Start-Tag* begonnene *Element* später enthalten:

```
type ParseStack = Stack (Symbol,[Tree])
```

Damit können wir den *Kellerautomaten* formulieren, der durch folgende Funktion definiert wird:

```
parseLayout :: Lang -> [Symbol] -> ParseStack -> [Warning]
               -> ([Tree],[Warning])
```

Durch den ersten Parameter wird also wieder die verwendete Sprache, danach eine Symbolliste und ein Stapel übergeben, gefolgt von den bisherigen Warnungen.
Jetzt machen wir uns einen Trick zunutze: Auf dem Boden des Stapels `ParseStack` postulieren wir ein Paar `(root,[])` mit einem *Wurzelsymbol* `root`, der Form:

```
root = (StartTag "" [] 0,start)
```

Dies ist sinnvoll, da C_L ja nicht zwangsläufig einen einzelnen Ausdruck repräsentiert, sondern eventuelle mehrere. Die aus ihnen entstehenden Bäume können von `parseLayout` nun alle in der Liste des Wurzelsymbols abgelegt werden, welches wir sinnvollerweise niemals vom Stapel entfernen. Dies ist bereits wegweisend für die Abbruchbedingung, die sich in der Abbildung des Parseralgorithmus (siehe 8.5) unter Bedingung 1. a) verbirgt. Bevor wir feststellen, dass der Parser nach Abb. 8.5 zwar die wesentliche Problematik löst, leider jedoch mitnichten vollständig ist, wollen wir einige erläuternde Worte finden.

Der Stapel enthält dem Typ nach zwar Symbole, wird tatsächlich aber ausschließlich *Start-Tags* stapeln.
Das *Reduzieren* des Stapels durch die Funktion `reduce`[5], ist nun gleichbedeutend mit dem *Beenden* der Verarbeitung des aktuellen (obersten) *Start-Tags*. Der Inhalt des assoziierten

[5] Dass `list` in der Definition von `reduce` in der Reihenfolge invertiert wird, liegt nur daran, dass der Parser sie in umgekehrter Reihenfolge aufbaut, da dies zunächst effizienter ist.

Elements liegt ja bereits in Bäumen vor und so wird dem vorherig ersten *Start-Tag* des Stapels ein neuer Baum mit dem, in ein `Element` übersetzten, aktuellen *Start-Tag*, hinzugefügt – was gerade durch die Funktion `assign` realisiert wird.
Die *Layout-Regel* kommt besonders in den Bedingungen 3. und 4. a) zum Tragen; denn durch 3. wird festgestellt, dass ein Element aufgrund der Einrückung des nächsten Symbols beendet ist. Außerdem wird durch die Instruktionen in 4. a) verhindert, dass einem *Start-Tag* Inhalte zugewiesen werden, obwohl dessen Einzug durch den *Layouter* echt kleiner als der vorherige Einzug bestimmt wurde. Dem inneren `div` darf hier kein Inhalt zugewiesen werden:

```
<div>
    <div>
  Hallo
```

Darüber hinaus werden in 1. b) am Ende der Symbolliste alle noch offenen *Tags* bzw. Elemente beendet, die durch 4. b) begonnen wurden. In 5. werden die Layout-Regelungen für *End-Tags* umgesetzt und in 6. kann das aktuell betrachtete Symbol nur noch Daten oder ein *Leer-Tag* darstellen. Da beide keine Inhalte besitzen können, werden sie ohne solche in das aktuelle *Tag* bzw. Element eingebettet.

Leider gibt uns die HTML hier noch einige Sonderfälle auf. Beispielsweise dürfen wir den Inhalt eines `pre`-Elements *nicht* mit der Layout-Regel auswerten! Das liegt daran, dass dem `pre`-Element in der HTML eine besondere Bedeutung zukommt. Betrachten wir den folgenden HTML-Ausdruck:

```
<p>
     <b>Wir           stehen
               weit
   auseinander.</b>
</p>
```

Das Element `p` spezifiziert einen *Absatz*, den ein gewöhnlicher *Webbrowser* etwa so darstellen wird:

Wir stehen weit auseinander.

Dass die Ausgabe zwischen den Worten lediglich ein einzelnes Leerzeichen zeigt, liegt daran, dass *mehrere* Leerzeichen oder Zeilenumbrüche in Texten in der HTML nur als *einzelne* Leerzeichen interpretiert werden, damit ihnen – wie zwischen *Tags* – eine *einrückende Rolle* zukommen kann.
Möchte man aber beispielsweise Quelltexte einer formalen Sprache auf einer Webseite darstellen, so ist dieses Verhalten höchst unerwünscht. Dafür gibt es das Element `pre`, dessen Textinhalte in einem Webbrowser *kongruent* dargestellt werden.
Das Problem ist, dass alle Arten von Einrückungen durch unseren bisherigen Parser auch in `pre`-Elementen ignoriert werden. Die logische Konsequenz ist, bestimmte Ausdrücke

Da der Stapel zumindest das Wurzelsymbol `root` enthält, ist er niemals leer und wir bezeichnen das oberste Stapelsymbol mit t. Falls die Symbolliste nicht leer ist, verstehen wir unter x das erste Symbol. Außerdem haben wir folgende Hilfsprozeduren:

```
assign :: (Symbol,[Tree]) -> ParseStack -> ParseStack
assign (sym,ts) =
  update (\(s,trs) -> (s,(Tree (sym2node sym) ts):trs))
reduce :: ParseStack -> ParseStack
reduce ((sym,list):st) = assign (sym,reverse list) st
```

Solange 1. und 1. a) nicht erfüllt sind, verarbeiten wir das erste Symbol und wiederholen nach der ersten erfüllten Bedingung den

Parseralgorithmus

1. *Die Symbolliste ist leer*
   a) *Der Stapel enthält nur noch das Wurzelsymbol* `root`
   Es wird die assoziierte Baumliste zurückgegeben.
   b) Ansonsten wird der Stapel mit `reduce` reduziert.
2. *x ist eine Einrückung*
   Das Symbol x wird ignoriert.
3. *x befindet sich außerhalb des Einzugs von t*
   Der Stapel wird mit `reduce` reduziert und x verbleibt in der Symbolliste.
4. *x ist ein Start-Tag*
   a) *Der Einzug von x ist kleiner oder gleich dem von t*
   x wird gleich wieder *geschlossen*, d.h. es wird t mit `assign` als Teilbaum mit leerer Teilbaumliste hinzugefügt.
   b) Ansonsten wird `(x,[])` ganz oben auf den Stapel gelegt.
5. *x ist ein End-Tag*
   a) *Der Stapel enthält nur noch das Wurzelsymbol* `root`
   x wird ignoriert, da es kein vorhergehendes *Start-Tag* besitzt, und es wird eine Warnung generiert.
   b) *x hat den gleichen Tagnamen wie t*
   Die Liste von t wird geschlossen, d.h. der Stapel wird mit `reduce` reduziert, und das nächste Symbol wird verarbeitet.
   c) *Das assoziierte Start-Tag zu x befindet sich im Stapel*
   Der Stapel wird reduziert aber x verbleibt in der Symbolliste.
   d) Ansonsten wird x ignoriert.
6. Ansonsten wird `(x,[])` mit `assign` in einen Baum umgewandelt und an die Teilbaumliste von t angehängt.

Abbildung 8.5.: Der Parseralgorithmus

bzw. Elemente *strikt* auszuwerten. Praktisch heißt dies, dass innerhalb solcher Ausdrücke auf die Layout-Regel verzichtet wird, und *End-Tags* wieder verpflichtend sind.
Die Implementierung enthält aus diesem Grund einen zweiten, *strikten* Parser, der ausschließlich im Falle L = `html` bzw. `L == H` eingesetzt wird, sobald beispielsweise ein `script-`, `style-` oder `pre-Start-Tag` gefunden wird. In Anlehnung an den bestehenden (Layout-)Parser ist seine Implementierung allerdings recht trivial, so dass wir an dieser Stelle auf die Details verzichten.

Außerdem setzt der tatsächlich implementierte Parser Prädikate für die Ermittlung von Verstößen gegen das *Content Model* (siehe 3.3) ein, um entsprechende Warnung auszugeben. Dies wird mit Prädikaten wie `isHtmlElement :: Symbol -> Bool` oder auch `isPhrasingElement :: Symbol -> Bool` erreicht.

8.2.2. Umsetzer

Dem Umsetzer kommt nun eine überschaubare Aufgabe zu, denn praktischerweise hat der *Parser* ein Ergebnis geliefert, das sich durch eine einfache *Baumrekursion* in das erwünschte Format umsetzen lässt. Allerdings muss natürlich wieder eine Unterscheidung zwischen XML und HTML stattfinden.
Der Umsetzer wird entsprechend einer festgelegten Schnittstelle für die Codeintegration implementiert. Wir können ihn im Wesentlichen aber durch folgende Funktion beschreiben:

```
translate :: Lang -> ([Tree],[Warning])
             -> (String,[Warning])
```

Die Ausgabezeichenkette soll dabei den neuen Curry-Quelltext enthalten, der im Zuge der Codeintegration in das ursprüngliche Curry-Programm eingesetzt wird.
Unsere Beispielinstanz wird nun ein letztes Mal umgewandelt. So wird aus

```
([Tree (Element "a" [("href",[Raw "#"])])
       [Tree (Content [Raw "Hello␣World!"])
             []]],[])
```

ein Paar, bestehend aus der *Zeichenkette* bzw. dem *Curry-Quelltext*

```
[HtmlStruct "a" [("href","#")] [HtmlText "Hello␣World!"]]
```

und einer leeren Liste von Warnungen.
Am Ende ist so ein Curry-Ausdruck bzw. sein Quelltext entstanden, der äquivalent zur ursprünglichen Eingabezeichenkette steht. Der *Präprozessor* der Codeintegration führt nun die Ersetzung durch, und die Übersetzung des Curry-Programms kann beginnen.

9. Fazit

Das Ziel dieser Arbeit war die Implementierung und Integration einer Hamlet-ähnlichen Sprache und Syntax für die Formulierung von XML- und HTML-Ausdrücken in Curry. Dafür sollten neue Parser die bereits existierende Codeintegration domänenspezifischer Sprachen erweitern.
Es hat sich herausgestellt, dass die Entwicklung eines einzigen Parsers für XML und HTML eine sinnvolle Idee war. Während der lexikalischen Analyse können Ausdrücke beider Sprachen vollkommen identisch beurteilt und verarbeitet werden. Lediglich während der syntaktischen Analyse wird die Unterscheidung der Sprache notwendig.
Die Webprogrammierung bzw. das Arbeiten mit den Auszeichnungssprachen XML und HTML im Allgemeinen ist in Curry nun erheblich eleganter geworden. Einmal vertraut mit der Syntax der Codeintegration, sind Programmierer in Curry nicht länger auf die Kenntnis der zugehörigen Datentypen oder abkürzender Funktionen angewiesen, um einzelne Ausdrücke oder ganze Dokumente der jeweiligen Auszeichnungssprache zu formulieren. Durch *Interpolation* können in Curry generierte Zeichenketten mittels kürzester Syntax in Elemente oder ihre Attribute eingebettet werden. Auf diese Weise lassen sich Inhalte bzw. Attributswerte in XML- oder HTML-Ausdrücken sehr dynamisch und übersichtlich gestalten.
Die wesentliche Problematik der Arbeit bestand darin, möglichst vollständige Konzepte für die einzelnen Phasen bzw. Schritte der Implementierung zu finden. Hierfür wäre natürlich die detailgetreue Umsetzung der aktuellen Spezifikationen der W3C eine erfüllende Maßnahme gewesen. Da diese mittlerweile jedoch so umfangreich ausfallen, mussten zwangsläufig Kompromisse gemacht werden. Diese zufriedenstellend auszuhandeln, war eine zeitaufwändige Aufgabe.
Die Implementierung selbst – und insbesondere die erfolgreiche Anbindung an den Codeintegrator von Sikorra – hat sehr viel Freude bereitet.
Natürlich kann dieses Projekt weitergeführt werden. In Hamlet werden über die QuasiQuotes typsichere Parser eingesetzt, die Ausdrücke also typspezifisch überprüfen und interpolieren können. Dies kann selbstverständlich auch in Curry gewährleistet werden. Dafür müssten die zu interpolierenden Ausdrücke jedoch auf ihren Typ hin untersucht werden, was das Parsen jener notwendig machen würde, und also einen Aufwand darstellt. Auch beinhalten die shakespearischen Templates QuasiQuoter für weitere domänenspezifische Sprachen, wie etwa *Cassius* für CSS und *Julius* für Javascript. Vergleichbare Parser in Curry zu implementieren, ist sicher auch eine interessante Aufgabe.

Literatur

[1] Simon Marlow (ed.) *Haskell 2010 Language Report*. 2010. URL: http://www.haskell.org/onlinereport/haskell2010/.

[2] M. Hanus (ed.) *Curry: An Integrated Functional Logic Language (Vers. 0.8.3)*. 2014. URL: http://www.curry-language.org.

[3] Michael Hanus. "High-Level Server Side Web Scripting in Curry". In: *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*. PADL '01. London, UK, UK: Springer-Verlag, 2001, S. 76–92. ISBN: 3-540-41768-0.

[4] Rudolf Schnabel Olaf von Grudzinski. "Mathematische Grundlagen. 2. überarbeitete Auflage". Mathematisches Seminar der Universität Kiel. Wintersemester 2011/2012.

[5] Mark Pilgrim. *HTML5 - Up and Running: Dive Into the Future of Web Development*. O'Reilly, 2010, S. I–XII, 1–205. ISBN: 978-0-596-80602-6.

[6] Michael Snoyman. *Developing Web Applications with Haskell and Yesod*. O'Reilly Media, Inc., 2012. ISBN: 1449316972, 9781449316976.

[7] W3C®. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Nov. 2008. URL: http://www.w3.org/TR/2008/REC-xml-20081126/.

[8] W3C®. *HTML5 - A vocabulary and associated APIs for HTML and XHTML*. Feb. 2014. URL: http://www.w3.org/TR/2014/CR-html5-20140204/.

[9] Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus, 3. Auflage*. Oldenbourg Verlag, 2011. ISBN: 978-3-486-70951-3.

Anhang

Das folgende Curry-Programm generiert aus einer Liste von Personen eine HTML-Tabelle, und speichert diese in einem HTML-Dokument `born0331.html`:

Anhang 1: Eine generierte HTML-Tabelle durch Codeintegration in Curry

```
1  import HTML
2
3  data Person = Person String Int Bool
4
5  genPersonRows :: [Person] -> [HtmlExp]
6  genPersonRows [] = []
7  genPersonRows ((Person name birth alive):ps) = ``html
8   <tr style="background-color:{color};">
9     <td>{name}
10    <td style='text-align:center;'>{show birth}''
11    ++ genPersonRows ps
12   where color | alive     = "white"
13               | otherwise = "lightgray"
14
15 genPersonTable :: Int -> [Person] -> HtmlExp
16 genPersonTable colw ps = HtmlStruct "table" [] rows
17   where header = ``html
18  <tr>
19    <th style="width:{show␣colw}px;">Name
20    <th style="width:{show␣colw}px;">Birth''
21         rows = header ++ genPersonRows ps
22
23 main :: IO ()
24 main = writeFile "born0331.html" $ showHtmlPage p
25   where
26    title = "Born␣On␣March␣31st"
27    p = standardPage title [genPersonTable 220 born0331]
28    born0331 = [Person "René␣Descartes"        1596 False,
29                Person "Al␣Gore"               1948 True ,
30                Person "Joseph␣Haydn"          1732 False,
31                Person "Christopher␣Walken"    1943 True ,
32                Person "Johann␣Sebastian␣Bach" 1685 False]
```

Anhang 2: Der durch die Codeintegration in Anhang 1 entstandene Curry-Quelltext

```
1  import HTML
2
3  data Person = Person String Int Bool
4
5  genPersonRows :: [Person] -> [HtmlExp]
6  genPersonRows [] = []
7  genPersonRows ((Person name birth alive):ps) = [
      HtmlStruct "tr" [("style",("background-color:"++(color
      )++";"))] [HtmlStruct "td" [] [HtmlText ((name)++"\n")
      ],HtmlStruct "td" [("style",("text-align:center;"))] [
      HtmlText ((show birth))]]]
8
9
10
11    ++ genPersonRows ps
12   where color | alive     = "white"
13               | otherwise = "lightgray"
14
15 genPersonTable :: Int -> [Person] -> HtmlExp
16 genPersonTable colw ps = HtmlStruct "table" [] rows
17   where header = [HtmlStruct "tr" [] [HtmlStruct "th" [("
        style",("width:"++(show colw)++"px;"))] [HtmlText ("
        Name\n")],HtmlStruct "th" [("style",("width:"++(show
         colw)++"px;"))] [HtmlText ("Birth")]]]
18
19
20
21         rows = header ++ genPersonRows ps
22
23 main :: IO ()
24 main = writeFile "born0331.html" $ showHtmlPage p
25   where
26    title = "Born␣On␣March␣31st"
27    p = standardPage title [genPersonTable 220 born0331]
28    born0331 = [Person "René␣Descartes"        1596 False,
29                Person "Al␣Gore"               1948 True ,
30                Person "Joseph␣Haydn"          1732 False,
31                Person "Christopher␣Walken"    1943 True ,
32                Person "Johann␣Sebastian␣Bach" 1685 False]
```

Anhang 3: Das durch Anhang 2 erzeugte HTML-Dokument `born0331.html`

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Born On March 31st</title>
    <meta http-equiv="Content-Type" content="text/html;␣
       charset=utf-8"/>
  </head>
  <body>
    <h1>Born On March 31st</h1>
    <table>
      <tr>
        <th style="width:220px;">Name</th>
        <th style="width:220px;">Birth</th>
      </tr>
      <tr style="background-color:lightgray;">
        <td>René Descartes</td>
        <td style="text-align:center;">1596</td>
      </tr>
      <tr style="background-color:white;">
        <td>Al Gore</td>
        <td style="text-align:center;">1948</td>
      </tr>
      <tr style="background-color:lightgray;">
        <td>Joseph Haydn</td>
        <td style="text-align:center;">1732</td>
      </tr>
      <tr style="background-color:white;">
        <td>Christopher Walken</td>
        <td style="text-align:center;">1943</td>
      </tr>
      <tr style="background-color:lightgray;">
        <td>Johann Sebastian Bach</td>
        <td style="text-align:center;">1685</td>
      </tr>
    </table>
  </body>
</html>
```

Ein gewöhnlicher Webbrowser dürfte `born0331.html` aus Anhang 3 etwa so darstellen:

Born On March 31st

Name	Birth
René Descartes	1596
Al Gore	1948
Joseph Haydn	1732
Christopher Walken	1943
Johann Sebastian Bach	1685