Skriptsprache

„by example“

Stand: 3. Mai 2000

Autor: Jens Elei - Copyright: IT-Entwicklung Klaus Schaarmann

Inhalt

1 Aufbau der Sprache

Ausgangspunkt für den Sprachentwurf des Konverters sind die zentralen Festlegungen der Verarbeitungsstrategie. Sie führen zu folgenden Beschreibungselementen, die jeweils einzelnen Erkennungsbereichen zugeordnet werden:

- Jeder Bereich wird zunächst durch Zuordnung eines Bezeichners eindeutig im Skript definiert. Dieser ***Bereichsdeklaration*** werden dann alle folgenden Informationen zugeordnet.
- Jedem Bereich wird eine ***Verarbeitungsmethode*** zugeordnet. Diese Methode entspricht einer Vorgehensweise, die geeignet ist typische Strukturen in der Eingabe zu verarbeiten. So erhalten Tabellen z.B. eigene Methoden, da sie besondere Strukturen aufweisen. Einer Methode werden dann innerhalb eines Blocks ***Verarbeitungsanweisungen*** zugeordnet, welche die Ausführung der Methode für den Anwendungsfall parametrisieren:
  - Es werden Zeichenfolgen oder Spaltenbreiten einer Tabelle festgelegt.
  - Der Eingabezeiger kann versetzt werden.
  - Einzelne Teile der Bereiche können als Unterbereiche definiert werden, wodurch die Strukturierung in einen Bereichsbaum erfolgt. Aus den Bereichsdeklarationen und ihrer Strukturierung in Unterbereiche entsteht so letztendlich das *Erkennungsmodell*.
- Jedem Bereich werden ***Merkmale*** zugewiesen, die eine Erkennung der einzelnen Bereiche ermöglichen und die ihren Umfang bestimmen. Paßt eine folgende Eingabe auf eine Bereichsbeschreibung, dann ist deren Verarbeitung zulässig, wenn sie durch eine Anweisung ausgelöst wird.
- Die Ausgabe wird durch das Metamodell einer Datenstruktur in einer DTD beschrieben. Für eine ***Abbildung der Eingabe auf die Ausgabe*** werden Entsprechungen zwischen Erkennungs- und Datenmodell identifiziert. Dies geschieht auf verschiedene Arten:
  - Einzelne Fragmente können durch Verarbeitungsanweisungen direkt Bestandteilen der Ausgabestruktur zugeordnet werden.
  - Verarbeitungsmethoden können ebenfalls Ausgabeelemente zugewiesen werden, wenn sie z.B. in jeder Iteration eine Ausgabe erzeugen.
  - Bereichen können Ausgabeelemente zugeordnet werden.

  Die Aufteilung der Abbildung auf verschiedene Stellen ist notwendig, um eine flexible Abbildung zu gewährleisten. So können Daten zwischen Tags oder als Attributwerte ausgegeben werden, und die Struktur von Ein- und Ausgabe wird nicht zu restriktiv gekoppelt.
- Um in flexibler Weise sowohl Positions- als auch Parseransatz verwenden zu können, wird eine Menge von ***Grundfunktionen*** definiert. Diese ermöglichen eine flexible Anwendung beider Ansätze für unterschiedliche Aufgaben:
  - Sie können zur Formulierung der *Bereichsmerkmale* genutzt werden, indem sie enthaltene Zeichen und den Bereichsumfang beschreiben.
  - Im Rahmen der *Verarbeitungsstrategie* werden sie genutzt um einzelne Fragmente zu beschreiben.

1.1 Verbindung der Elemente im Sprachaufbau

Alle Elemente der Beschreibung werden durch die Skriptsprache zusammengefaßt. Dabei ist das Skript in einzelne Blöcke für die einzelnen Bereiche der Eingabe aufgeteilt. Jedem dieser Bereiche werden dann die erläuterten Informationen zugeordnet:

- *Merkmale* für die Erkennung des Bereiches.
- Eine *Verarbeitungsmethode*, deren einzelne *Anweisungen* dann den Rumpf der Bereichsbeschreibung bilden.
- Ein XML Element der Ausgabe, wenn der Bereich diesem entspricht. Auf dieses Element erfolgt dann ggf. die *Abbildung der Eingabe*.

Es entsteht insgesamt ein Skript das aus einzelnen Blöcken besteht, die jeweils einem Bereich entsprechen. Ein solcher Block hat die folgende Form:

```
<Bereichsdeklaration>:
     <Merkmale> [, <Abbildungselement>]: <Verarbeitungsmethode>
     {    <Verarbeitungsanweisungen>   };
```

Die Definition der Bestandteile dieses Blocks wird in den folgenden Kapiteln beschrieben.

1.2 Datenstrukturen der Verarbeitung

Bei der Verarbeitung wird sequentiell eine Eingabedatei durchlaufen, die im Rahmen einzelner Bereiche in eine strukturierte Ausgabe überführt wird. Um den Status der Verarbeitung zu beschreiben werden zu jeder Zeit bestimmte Datenstrukturen verwaltet:

1.2.1 Globale Daten

Die einzige globale Größe ist ein *Bereichsstack*, der jederzeit die aktuelle Hierarchie der Bereiche, zusammen mit deren lokalen Informationen, enthält. Jedes Mal wenn in einen untergeordneten Bereich verzweigt wird, dann wird der Ausgangsbereich auf diesem Stack abgelegt. Wenn er wieder verlassen wird, dann wird er entsprechend wieder vom Stack geholt.

Untergeordnete Bereiche greifen grundsätzlich nicht auf die lokalen Informationen übergeordneter Bereiche zu. Allerdings werden zwei Größen beim Betreten eines untergeordneten Bereiches direkt an diesen übergeben:

- Die *kleinste Bereichsbegrenzung des übergeordneten Bereichs* wird übergeben, damit überwacht werden kann, daß untergeordnete Bereiche nicht Grenzen ihrer Elternknoten überschreiten. Praktisch reicht es die Grenze des direkt übergeordneten Bereiches oder die Grenze einer ausgeführten Anweisung zu übergeben, da keine andere übergeordnete Grenze kleiner sein kann.
- Der *Eingabezeiger* verweist immer auf eine Position innerhalb einer Eingabe. Diese Position wird immer als Ausgangspunkt für die Verarbeitung eines Bereiches benutzt, d.h. er gilt als Startpunkt für die Suche nach Bereichsmerkmalen. Beim übergebenen Zeiger sind zwei Fälle zu unterscheiden:
  - Es wird der aktuelle Eingabezeiger der Eingabedatei übergeben.
  - Eine Methode kann möglicherweise bereits Transformationen der Eingabe vornehmen. In diesem Fall wird ein Zeiger in eine temporär erzeugte Eingabe übergeben.

1.2.2 Lokale Daten eines Bereiches

Die lokalen Daten eines Bereiches bestehen aus verschiedenen Komponenten, die je nach zugeordneter Verarbeitungsmethode variieren können. In jedem Fall bestehen sie aber zumindest aus den folgenden Teilen:

- Der *Eingabezeiger* wird beim Betreten des Bereiches vom übergeordneten Bereich initialisiert. Er wird auf drei verschiedene Arten manipuliert:
  - Treffen die Merkmale des Bereiches auf die Eingabe zu, dann wird der Zeiger auf den Beginn des Bereiches gesetzt.
  - Während der Verarbeitung können verschiedene Anweisungen den Zeiger erhöhen, aber niemals in der Eingabe zurück setzen.
  - Ist die Verarbeitung des Bereiches abgeschlossen, dann wird der Eingabezeiger nur dann auf das Ende des Bereiches gesetzt, wenn dieser Bereich selbst ein Bereichsende definiert.
- Die *Bereichsobergrenze* wird ebenfalls beim Betreten des Bereiches vom übergeordneten Bereich initialisiert. Wenn dem neuen Unterbereich eine eigene Bereichsgrenze zugewiesen ist, dann wird dieser Wert als neuer Wert der Obergrenze gesetzt. Es sei darauf hingewiesen, daß diese neue Grenze immer kleiner oder gleich der alten ist, also niemals weiter hinten im Eingabestrom liegen kann.

1.3 Betreten und Verlassen von Bereichen

Die Informationen eines Bereiches werden in einer bestimmten Reihenfolge genutzt, wenn er als nächster Bestandteil der Verarbeitung referenziert wird. Um das Zusammenwirken der Komponenten zu verdeutlichen, seien hier zunächst die einzelnen Phasen erläutert, die beim Betreten und Verlassen eines Bereiches durchlaufen werden:

0. Das Betreten eines Bereiches wird durch eine Verarbeitung angestoßen. Dabei wird die Bereichsbegrenzung des übergeordneten Bereiches und ein Eingabezeiger übergeben, sowie lokale Informationen auf dem Stack abgelegt. Erhält die aufrufende Anweisung die Kontrolle zurück, dann ist sie auch für die Wiederherstellung ihrer Informationen verantwortlich.
1. Die Merkmale des Bereiches werden überprüft. Ist ein Merkmal nicht erfüllt, dann wird die Kontrolle zurückgegeben. Je nach Anweisung wird dort ein Fehler ausgelöst, oder eine alternative Verarbeitung gewählt.
2. Treffen die Merkmale zu, dann wird der Eingabezeiger an den Anfang des beschriebenen Bereiches gesetzt.
3. Ist dem Bereich ein XML Element der Ausgabe zugewiesen, dann wird dieses in einer Zwischendarstellung der Ausgabe erzeugt (Start-Tag).
4. Die eigentliche Verarbeitung wird ausgeführt.
5. Ist dem Bereich ein XML Element der Ausgabe zugewiesen, dann wird dieses in einer Zwischendarstellung der Ausgabe abgeschlossen (End-Tag).
6. Ist dem Bereich ein Bereichsende zugewiesen, dann wird der Eingabezeiger auf dieses Ende gesetzt.
7. Die Kontrolle wird an die aufrufende Anweisung zurückgegeben.

2 Grundfunktionen

Aufbauend auf der Strategie hierarchischer Bereiche werden immer wieder die beiden beschriebenen Ansätze verwendet:

- Positionsansatz
- Parseransatz

Dies schlägt sich zum einen in bestimmten Sprachkonstrukten nieder, und ist zum anderen auch die Idee einer Menge von Grundfunktionen.

Die Idee dieser Funktionen entsteht aus der Notwendigkeit für die verschiedensten Aufgaben des Konverters immer wieder mit Positionen oder Mustern bestimmte Informationen zu erzeugen, die dann in verschiedener Weise genutzt werden:

- Es ist notwendig Zeichenketten zu beschreiben und zu extrahieren.
- Es ist notwendig Erkennungsmerkmale einzelner Bereiche zu beschreiben.
- Es ist notwendig Start- und Endpunkte im Zeichenstrom zu beschreiben. Sie können z.B. den Start und das Ende von Zeichenketten oder Spalten in Tabellen festlegen.

Diese Informationen lassen sich jeweils mit Hilfe des Positions- und des Parseransatzes formulieren. Da sie an unterschiedlichen Stellen Verwendung finden, werden im folgenden Grundfunktionen aufgebaut, die immer diese gleiche Menge von Informationen erzeugen. Damit ist es möglich auf Basis einer kleinen Funktionsmenge verschiedene Aufgaben einheitlich zu beschreiben.

2.1 Ergebnis der Funktionen

Alle nachfolgenden Funktionen erzeugen die gleiche Datenstruktur als Ergebnis. Dies ist notwendig, um sie in der gleichen Weise und an den gleichen Stellen flexibel verwenden zu können. Sie besteht aus den oben motivierten Elementen, abgelegt in:

- `str:string` enthält eine gefundene Zeichenkette. Für reguläre Ausdrücke ist das die zum Muster passende Zeichenfolge. Bei absoluten Angaben die Zeichen von der aktuellen Eingabeposition bis zu einem angegebenen Punkt.
- `found:bool` gibt an ob ein entsprechender String gefunden wurde. Kann für reguläre Ausdrücke falsch sein, wenn ein Muster nicht in einem Bereich gefunden wird, und für absolute Angaben, wenn eine Bereichsbegrenzung überschritten wird.
- `start:number`, `end:number` enthält die absolute Start- und Endposition der Zeichenkette im Eingabestrom. Die Werte sind relativ zur aktuellen Position, können aber ggf. umgerechnet werden. Der Wert in `end` wird als exklusiver Zeiger des Strings verwendet, d.h. das Zeichen auf das er verweist ist nicht mehr Element des Strings.

Diese Datenstruktur wird im folgenden als `DescRes` für *„description result“* bezeichnet.

2.2 Grundfunktionen mit Positionsangaben

Eine Verwendung absoluter Angaben muß zunächst im Rahmen zweier Eigenschaften konkretisiert werden:

- *Bezugspunkt der Angabe*

  Eine angegebene Entfernung kann sich auf verschiedene Ausgangspunkte im sequentiellen Strom beziehen, wie z.B. auf den Dateianfang oder einen Bereichsanfang. Der hier verwendete Bezugspunkt wird immer der aktuelle globale Eingabezeiger (Cursor) sein, da diese Größe der Vorstellung einer schrittweisen Verarbeitung der Eingabe entspricht.

- *Dimensionen der Angabe*

  Angaben können sowohl auf den sequentiellen Eingabestrom, als auch auf die zweidimensionale Darstellung bezogen sein. Dies motiviert drei verschiedene Funktionen, jeweils eine bezogen auf den sequentiellen Strom (`posStream`) und auf die zweidimensionale Darstellung (`posML`), sowie eine für den Sonderfall der zweidimensionalen Darstellung, bezogen auf eine einzelne Zeile (`posSL`).

Die Aufgaben der drei Funktionen lassen sich innerhalb ihrer Signatur beschreiben:

- `posStream(numChars: number)` → `DescRes`

  Mit der Funktion `posStream` lassen sich Entfernungen im sequentiellen Strom beschreiben. Es handelt sich um eine eindimensionale Angabe bei der Zeilenumbrüche *immer* als 1 Zeichen (*`CR`*) gezählt werden. Der Parameter `numChars` beschreibt die Anzahl der Zeichen als Entfernung vom aktuellen Eingabezeiger. In der Ergebnisstruktur wird der aktuelle Zeiger als `start` und der beschriebene Punkt als `end` eingetragen. Der String `str` spiegelt die Grenzen entsprechend wider. Der Wert von `found` kann nur dann `false` werden, wenn durch die Beschreibung eine Bereichsgrenze überschritten wird. Der Eingabezeiger wird nicht versetzt.

- `posSL(numChars: number)` → `DescRes`

  Die Funktion `posSL` (*positional single line*) trägt dem Umstand Rechnung, daß oft Entfernungen in der gleichen Zeile beschrieben werden müssen. Zeilenumbrüche stellen eine rechte Grenze dar, und es gilt `found=false`, wenn die Position in der Zeile oder dem Bereich nicht vorhanden ist. Der Parameter `numChars` hat die gleiche Bedeutung wie zuvor, ebenso wie `start`, `end` und `str` wieder entsprechend aktualisiert werden.

- `posML(numLines: number, numCols: number)` → `DescRes`

  Mit `posML` (*positional multiple lines*) lassen sich zweidimensionale Angaben machen, wobei wieder `found=false` gilt, wenn eine Position nicht existiert. Der Parameter `numLines` gibt die Anzahl der Zeilen an, um die das Ende vertikal verschoben ist, und der Parameter `numCols` beschreibt die Entfernung vom Beginn der beschriebenen Endzeile zu einem Zeichen in dieser Zeile. Es handelt sich in diesem Fall also um eine absolute Position innerhalb einer Zeile. Wiederum werden `start`, `end` und `str` gesetzt, der Eingabezeiger aber nicht weitergesetzt.

Keine dieser Funktionen nimmt von sich aus eine Verschiebung des Eingabezeigers vor. Eine Verschiebung bewirken erst einige Funktionen, die diese als Beschreibungen einsetzen.

Beispiele:

Exemplarische Eingabe

a	**b**	c	d	*cr*				
e	f	g	h	i	j	*cr*		
k	l	m	n	o	*cr*			
p	q	r	s	t	u	v	w	*cr*

Der Eingabezeiger steht zu Beginn auf **b**.

- `posStream(2)`

  In `DescRes` zeigt `start` auf das „b“ und `end` auf das „d“. Der String `str` enthält „bc“.

- `posSL(2)`

  Erzeugt das gleiche Ergebnis.

- `posStream(4)`

  Setzt den `end` Wert in `DescRes` um `4` Zeichen verschoben auf „e“. Zudem zeigt `start` auf das „b“. Der String `str` enthält „bcd↵“.

- `posSL(4)`

  In `DescRes` zeigt `start` auf das „b“ und `end` auf den Zeilenumbruch („CR“). Zusätzlich ist `found=false`.

- `posML(0, 2)`

  Versetzt `end` um `0` Zeilen und auf das `2`-te Zeichen, also wieder genau auf das „b“. Der Wert in `start` zeigt auf das „b“. Der String `str` enthält nichts.

- `posML(0, 4)`

  Versetzt `end` um `0` Zeilen und auf das `4`-te Zeichen, also auf „d“. In `DescRes` zeigt `start` weiterhin auf „b“. Der String `str` enthält „bc“.

- `posML(1, 1)`

  Verschiebt `end` um `1` Zeile und auf das `1`-te Zeichen, also auf das „e“. In `DescRes` zeigt `start` auf das „b“. Der String `str` enthält „bcd↵“.

- `posML(2, 7)`

  Verschiebt `end` auf den Zeilenumbruch hinter dem „o“, da diese Zeile kein `7`-tes Element enthält. Deshalb ist auch `found=false`. Der Wert in `start` weist weiterhin auf das „b“.

In allen Fällen in denen `found` nicht näher beschrieben ist gilt `found=true`. Reicht eine Angabe über das Ende eines begrenzten Erkennungsbereichs heraus, dann gilt auch `found=false`, da es unzulässig ist eine Bereichsgrenze zu überschreiten. In Fällen in denen `found=false` ist, wird ein Ergebnis für die anderen Datenelemente errechnet, da deren Informationen teilweise trotzdem genutzt werden kann, nicht zuletzt für entsprechende Fehlermeldungen.

2.3 Grundfunktionen mit regulären Mustern

Die zweite Menge von Grundfunktionen stützt sich auf die Ausdrucksmöglichkeiten regulärer Ausdrücke. Deren Mächtigkeit ist ein elementarer Einflußfaktor für die Anwendbarkeit von Beschreibungen.

Der Begriff „Mächtigkeit“ ist dabei bereits durch das Konzept dieser regulären Ausdrücke mit Sequenz, Fallunterscheidung und Iteration beschrieben. In einer entsprechenden textuellen Darstellung erweitert um eine Klammerung.

2.3.1 Spezifikation regulärer Ausdrücke

Auch wenn der Konverter nicht die Menge der Sprachen erweitert, die mit regulären Ausdrücken beschrieben werden können, so werden doch einige Schreibweisen eingeführt, welche dem besonderen Verwendungszweck Rechnung tragen.

Zunächst sind einige verkürzte Schreibweisen für häufig verwendete Kombinationen der drei Operatoren Sequenz (`ab`), Iteration (`a*`) und Fallunterscheidung (`a|b`) sinnvoll. Die hier zu implementieren Konstrukte werden um folgendes erweitert (definiert über Σ={a,b,c,d,e,f}):

Schreibweise	Beschreibung	formal
`a+`	Iteration, mindestens ein Element	`aa*`
`a?`	optionales Auftreten, Element kommt einmal oder keinmal vor.	`a | ε`
`a{<MIN>, <MAX>}`	frei beschränktes Auftreten, Element kommt mindestens `MIN` und höchstens `MAX` mal vor. Für `MAX=0` existiert keine obere Grenze. Der Fall `MIN>MAX`, `MAX>0` erzeugt einen Fehler.	$\underbrace{a\ldots a}_{<MIN>}$ `a*` für `MAX=0` $\underbrace{a\ldots a}_{<MIN>}$ `|` … `|` $\underbrace{a\ldots a}_{<MAX>}$, sonst.
`[abc]`	Menge möglicher Zeichen	`a | b | c`
`[^abc]`	Komplement der Menge möglicher Zeichen	`d | e | f`
`[a-d]`	Bereich möglicher Zeichen	`a | b | c | d`
`[^a-d]`	Komplement des Bereichs möglicher Zeichen	`e | f`

Zusätzlich werden noch einige besondere Zeichenmengen definiert, die oftmals Verwendung finden werden. Dies sind zunächst besonders kurze Schreibweisen für ein beliebiges Zeichen:

Schreibweise	Beschreibung
`.`	einzelnes beliebiges Zeichen inklusive Zeilenumbruch
`_`	einzelnes beliebiges Zeichen exklusive Zeilenumbruch

Alle anderen Zeichenfolgen werden immer durch einen kurzen Code beschrieben, eingeleitet von einem umgekehrten Schrägstrich („\“):

Code	Beschreibung
`\cr`	Zeilenumbruch
`\num`	Ziffer, d.h. gleichbedeutend mit: `[0-9]`.
`\lit`	Literal, d.h. gleichbedeutend mit: `[a-zA-Z]` (ohne Umlaute)
`\slit`	kleine Literale, d.h. gleichbedeutend mit: `[a-z]`
`\clit`	große Literale, d.h. gleichbedeutend mit: `[A-Z]`
`\x`*nnn*	Das Zeichen mit der hexadezimalen Kodierung *nnn*.
`\fill`	Füllzeichen, d.h. Leerstellen und Tabulatoren

Für alle verwendeten Sonderzeichen gilt, daß sie nicht als Steuerzeichen, sondern selbst als Zeichen interpretiert werden, wenn man ihnen einen umgekehrten Schrägstrich voranstellt. Dadurch wird z.B. „`\.`“ als das Zeichen „Punkt“ und nicht als beliebiges Zeichen interpretiert. Das gleiche gilt auch für den umgekehrten Schrägstrich selbst, „`\\`“ erzeugt also „\“.

2.3.1.1 Suche nach regulären Mustern

Die Suche nach einem Muster innerhalb einer Zeichenfolge beginnt immer genau am Start der Folge, d.h. das Muster „`cd`“ angewendet auf die Zeichenfolge „`abcdef`“ findet keine Übereinstimmung, weil die Zeichenfolge nicht mit „`cd`“ beginnt. Die Suche nach der Zeichenfolge an einer beliebigen Position wird dann beschrieben durch: „`.*cd`“ bzw. „`_*cd`“.

Dies fordert auch eine andere Konsequenz, die eine Expansion regulärer Ausdrücke von anderen Implementationen unterscheidet: Es wird nicht das längste Präfix, sondern das kürzeste mögliche Präfix gematcht. Andernfalls würde eine Angabe von „`.*`“ immer automatisch auf die gesamte Datei expandiert.

2.3.1.2 Anwendungsspezifische Erweiterungen

Letztlich existieren noch zwei anwendungsspezifische Erweiterungen, die in keiner Weise den Aufbau eines Ausdrucks beeinflussen, sondern als zusätzliche Informationen für den Konverter genutzt werden. Es handelt sich um zwei Markierungen in den Expansionen der Ausdrücke:

„`\lb`“ kennzeichnet eine Start- und „`\rb`“ eine Endmarkierung innerhalb eines regulären Ausdrucks. Sie dient dazu nur einen Teil einer erkannten Zeichenfolge zu separieren, die in einem größeren Kontext beschrieben werden soll (`lb` für „*left border*“ und `rb` für „*right border*“). Ein Beispiel dafür ist:

> „`Datum: \lb[\num\.]+\rb\cr`“ separiert ein Datum bestehend aus Ziffern und Punkten vor dem die Zeichenkette „`Datum: `“ steht, gefolgt von einem Zeilenumbruch.

Diese Marken werden innerhalb verschiedener Aufgaben dafür genutzt werden linke und rechte Grenzen in der Eingabe zu definieren, mit deren Hilfe dann z.B. festgelegt wird:

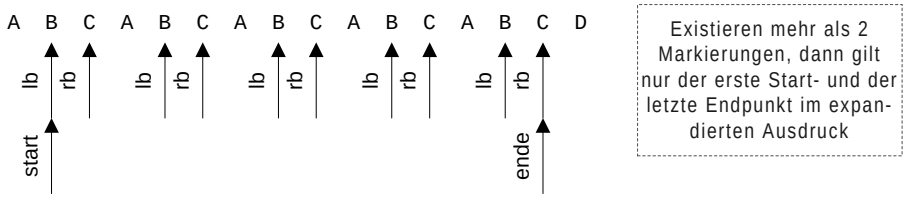
- In welchem Bereich einer Zeichenfolge eine Teilfolge extrahiert werden soll.
- An welcher Stelle Erkennungsbereiche beginnen oder enden.
- An welcher Stelle Anfang bzw. Ende einer Tabelle liegen.

Für das Auftreten dieser Markierungen in einer erkannten Expansion gilt:

- Wenn beide Marker vorkommen, dann bestimmt `\lb` die Position von `start` und `\rb` die Position von `end`.
- Wenn kein Marker vorkommt, dann wird angenommen sie befinden sich direkt vor dem Anfang und nach dem Ende.
- Ein einzelner Marker wird entsprechend als Start oder Ende interpretiert. Der fehlende befindet sich wieder automatisch am Anfang bzw. am Ende.
- Kommt mehr als ein Marker vom gleichen Typ vor, entweder durch mehrfache Angabe oder durch seine Wiederholung in einer Iteration, dann gilt die erste auftretende Startmarkierung und die letzte auftretende Endmarkierung.

  Beispiel: „`(A\lbB\rbC)*D`“ kann in seiner Expansion beliebig viele Markierungen aufweisen. Es gilt dann nur die erste und letzte Markierung:

Beispiel für mehr als 2 Markierungen



Die markierte Position befindet sich immer direkt hinter der Marke. Kann eine Position nicht eindeutig bestimmt werden, wie z.B. in „`A.*\lb.*B`“, dann ist das Ergebnis undefiniert. Sind durch Alternativen (`|`) mehrere Positionen möglich, wie z.B. in „`(A\lbB\rbC)|(\lbABC\rb)`“, dann wird das Ergebnis erzeugt, das zuerst erfolgreich expandiert werden kann. Die Auswertung erfolgt dabei von links nach rechts, d.h. hier wird „`A\lbB\rbC`“ erzeugt. Zur leichteren Handhabung sollten solche Fälle aber vermieden werden, was durch Ausdrucksumstellung möglich ist.

2.3.2 Spezifikation der Funktionen

Die folgenden Funktionen nutzen diese regulären Ausdrücke innerhalb ihrer Signatur. Der reguläre Ausdruck steht dabei immer im Parameter „pattern“:

- `regStream(pattern: string, maxChars: number) → DescRes`

  Versucht ein Muster im sequentiellen Strom zu finden, wobei die Suche auf `maxChars` Zeichen beschränkt werden kann. Für `maxChars=0` ist die Suche unbegrenzt. Die Werte von `start` und `end` werden auf die beschriebenen Positionen von `\lb` und `\rb` gesetzt, der String `str` enthält die entsprechende Zeichenfolge. Der Wahrheitswert `found` wird dann `false`, wenn das Muster `pattern` nicht innerhalb der `maxChars` Zeichen gefunden werden kann, oder das Muster über oder hinter einer Bereichsgrenze expandiert wird. Der Eingabezeiger wird nicht versetzt.

- `regSL(pattern: string) → DescRes`

  Versucht das Muster ab der aktuellen Position zu finden, begrenzt auf die aktuelle Zeile. Die Werte von `DescRes` werden wieder genauso gesetzt wie zuvor, wobei nun die Suche aber auf die aktuelle Zeile beschränkt ist. Wieder wird der Eingabezeiger nicht versetzt.

- `regML(pattern: string, maxRows: number) → DescRes`

  Beschränkt die Expansion eines Musters auf maximal `maxRows` Zeilen, d.h. die Funktion ist für `maxRows=1` gleich der Funktion von `regSL`. Wieder werden die Werte von `DescRes` entsprechend gesetzt, der Eingabezeiger aber nicht verändert.

Rückgabe der Funktionen ist also immer die beschriebene Datenstruktur. Dabei enthalten `start` und `end` die Positionen der beschriebenen `\lb und \rb` Marker, und `str` die Zeichenfolge zwischen den beiden Marken. Der Wert von `found` signalisiert, ob die spezifizierte Sprache innerhalb der beschriebenen Grenzen zur Eingabe gepaßt hat.

Beispiele:

- `regStream(“Datum:”, 0)` untersucht ob an der aktuellen Position des Eingabezeigers die Zeichenkette „`Datum:`“ steht.
- `regStream(“.*Datum:”, 0)` untersucht den gesamten Rest der Eingabe, bis zum ersten Mal die Zeichenkette „`Datum:`“ gefunden wird.
- `regSL(“Datum:”)` funktioniert wie `regStream(“Datum:”, 0)`.
- `regSL(“.*Datum:”)` sucht die Zeichenkette „`Datum:`“ im Rest der aktuellen Zeile.
- `regML(“.*Datum:”, 1)` funktioniert exakt wie `regSL(“.*Datum:”)`.
- `regML(“.*Datum:”, 3)` sucht die Zeichenkette „`Datum:`“ im Rest der aktuellen und den beiden folgenden Zeilen.

3 Bereichsbeschreibung

3.1 Bereichsdeklaration

Der Block jedes Bereiches beginnt immer mit dem Schlüsselwort `area`, gefolgt von der Deklaration eines Namens für diesen Bereich:

`area` <*`Bereichsname`*>**:**

Durch dieses Statement wird der Name des Bereiches deklariert, d.h. er kann in anderen Beschreibungen zur Verzweigung in Unterbereiche verwendet werden. Auf diese Deklaration folgen dann alle weiteren Beschreibungselemente, die diesem Bereich zugeordnet werden.

3.2 Bereichsmerkmale

Die Bereichsmerkmale bestehen aus folgenden Informationen:

- Einer Beschreibung welche Eigenschaften die folgende Eingabe erfüllen muß, um als dieser Bereich akzeptiert zu werden. Dies geschieht durch die Angabe einer Grundfunktion, dessen Ergebnis im Wert `found` bestimmt, ob die Eingabe zu diesem Bereich passen kann.
- Einer Beschreibung der Startposition für die Verarbeitung dieses Bereiches. Diese Startposition wird durch den Wert von `end` aus der Ergebnisstruktur einer Grundfunktion bestimmt.
- Die Beschreibung einer Endposition kann durch die Angabe einer zweiten Beschreibung mit einer entsprechenden Grundfunktion erfolgen. In diesem Fall bestimmt `end` dann die hintere Bereichsgrenze.

Es werden also Informationen aus der Anwendung von Grundfunktionen verwendet, um zwei verschiedene Arten von Bereichen zu beschreiben:

- *Geschlossene Bereiche* besitzen sowohl eine definierte Start- als auch eine definierte Endposition. Für die Beispielrechnung wird z.B. beschrieben, daß der Bereich der Rechnungspositionen mit einer Linie („`----------*`“) beginnt und mit einer zweiten Linie endet.
- *Offene Bereiche* besitzen nur eine definierte Startposition, d.h. ihr Ende wird nicht explizit beschrieben, sondern ergibt sich aus der Verarbeitung. Für die Beispielrechnung würde dann zunächst nur die erste Linie als Start beschrieben, das Ende bleibt zunächst unspezifiziert.

Es kann nicht nur ausreichend sein auf eine Endbeschreibung zu verzichten, sondern sogar erforderlich, wenn sich das Ende eines Bereiches erst durch seine Verarbeitung ergibt.

In diesem Fall kann bzw. muß der Konverter auf die Suche einer Endbeschreibung verzichten, was zusätzlich den positiven Effekt hat, daß der Verarbeitungsaufwand verringert wird. Andererseits wird die vorherige Beschreibung des Endes an einigen Stellen notwendig sein. In diesen Fällen wird im folgenden explizit darauf hingewiesen.

Die Merkmale eines Bereiches bestehen letztlich aus einer notwendigen Startbeschreibung und einer optionalen Endbeschreibung:

- **start(***Merkmale: DescRes***)**

  Mit dem Schlüsselwort `start` wird eine Startbeschreibung eingeleitet, die für jeden Bereich notwendig ist. Sie nutzt Informationen aus einer `DescRes` Struktur, d.h. es wird eine der Grundfunktionen verwendet, um den Start zu beschreiben. Genutzt werden dabei einerseits der Wert `end`, um die exakte Startposition des Bereiches festzulegen, und andererseits `found`, um festzustellen ob dieser Bereich überhaupt an dieser Stelle passen kann.

  Die Startposition wird bei der Verarbeitung des Bereiches dazu genutzt den Eingabezeiger zu Beginn der Verarbeitung auf diese Position zu setzen.

- **end(***Merkmale: DescRes***)**

  Diese optionale Beschreibung nutzt den Wert von `end` aus einer weiteren Anwendung einer Grundfunktion, um eine hintere Grenze eines Bereiches festzulegen. Zusätzlich dient der Wert `found` auch wieder als Erkennungsmerkmal. Startpunkt für die Anwendung ist der Bereichsbeginn.

  Die festgelegte Endposition wird zum einen innerhalb einiger Verarbeitungsstrategien benutzt, und legt andererseits eine neue Position des Eingabezeigers beim Verlassen dieses Bereiches fest. Aus diesem Grund wird auch ein Fehler ausgelöst, wenn das Bereichsende hinter dem Bereichsende eines übergeordneten Bereiches liegt.

Soll der Anfang eines Bereiches also z.B. anhand des enthaltenen Wortes „`RECHNUNG`“ erkannt werden, das innerhalb der nächsten 3 Zeilen auftaucht, dann wird dies z.B. wie folgt angegeben:

```
start(regML(".*\rbRECHNUNG", 3))
```

Durch die Angabe der `\rb` Markierung wird festgelegt an welcher Stelle der Bereich genau beginnt. Beim Betreten dieses Bereiches wird dann der Eingabezeiger automatisch auf den so beschriebenen Anfang gesetzt. Analog funktioniert dies auch für Endbeschreibungen:

Beispiel: (Endbeschreibung des Headers, expandierter regulärer Ausdruck markiert)

```
end(regML(".*-{10,100}\cr\rb", 10))
```

Endbeschreibung des Headers in der Beispielrechnung

```
RECHNUNG↵
↵
Jens Elei↵
Kastorpfaffenstr. 1↵
56068 Koblenz↵
↵
Datum: 1.12.1999↵
↵
Anz ArtName                      Einzelpreis[DM] Gesamtpreis[DM]↵
-------------------------------------------------------------↵
 1  PS/2 Adapter                    10,00          10,00↵
...
```

Die Markierung des rechten Randes durch `\rb` signalisiert das genaue Ende, und wird beim Verlassen des Bereiches auch als neue Eingabeposition festgelegt.

Zusammenfassend beginnt ein Bereichsblock damit mit einer Folge der folgenden Form:

area <*Name*>**: start(**<*Funktion*>**)** [**, end(**<*Funktion*>**)**]

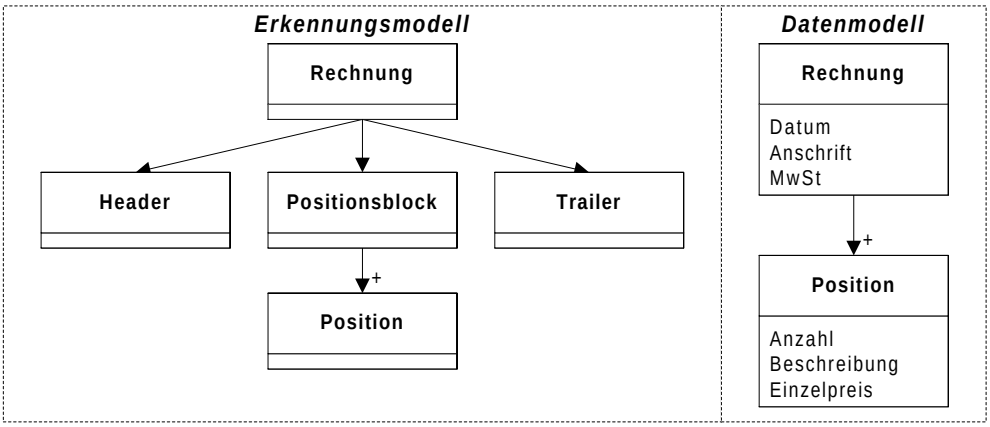
3.3 Abbildung von Bereichen auf die Ausgabe

Den einzelnen Bereichen des Erkennungsmodells können einzelne Knoten des Ausgabemodells entsprechen, wie zuvor beschrieben.

Wenn nun ein solcher Bereich der Eingabe nach Jackson „in gleicher Sequenz“ erkannt wird, wie seine Knoten im Ausgabemodell erzeugt werden müssen, dann kann man diese Erkennung direkt an die Erzeugung von Knoten in der Ausgabe koppeln.

Beispiel:

Erkennungsmodell und Datenmodell der Beispielrechnung



Immer wenn während der Erkennung ein Element „`Position`“ erkannt wird, dann muß auch in der Ausgabe eine Instanz des Typs „`Position`“ erzeugt werden. Erkennung und Ausgabe laufen „in gleicher Sequenz“. Folglich kann die Erzeugung einer „`Position`“ in der Ausgabe in drei Schritten erfolgen:

1. Wird der Erkennungsbereich betreten, dann wird eine neue Instanz in der Ausgabe erzeugt.
2. Während einer Verarbeitung innerhalb des Bereiches werden den Attributen Werte zugewiesen, oder Daten hinter das Start-Tag geschrieben.
3. Wird der Erkennungsbereich wieder verlassen, dann werden alle Daten zusammen mit dem Abschluß des Tags in der Ausgabe abgelegt.

Um diesen Fall zu beschreiben, kann man einem Eingabebereich durch ein weiteres optionales Konstrukt am Anfang des Blocks ein entsprechendes XML Element der Ausgabe zuweisen:

`node(`<*`XML-Element-Name`*>**`)`**

Damit wird einem deklarierten Erkennungsbereich ein Ausgabeknoten zugewiesen, was dafür sorgt, daß während der Verarbeitung verschiedene Schritte automatisiert werden:

- Beim Betreten des Bereiches wird in der Ausgabe ein Start-Tag des XML Elementes erzeugt, d.h. es wird ein Ausgabeknoten initiiert.
- Beim Verlassen des Bereiches wird in der Ausgabe ein End-Tag des XML Elementes erzeugt, d.h. der Ausgabeknoten wird abgeschlossen.

Alle Ausgaben die zwischen Betreten und Verlassen des Bereiches gemacht werden, sind somit in dieses XML Element eingebettet.

Der mögliche Anfang eines Bereichsblocks wird damit erweitert zu:

`area` <*`Name`*>**`: start(`**<*`Funktion`*>**`)`** `[`**`, end(`**<*`Funktion`*>**`)`**`] [`**`, node(`**<*`XMLName`*>**`)`**`]:`

Beispiel:

Dem Eingabebereich „`Position`“ wird ein Bereich zugewiesen der genau einer Zeile entspricht. Der Anfang der Zeile ist daran zu erkennen, daß innerhalb der ersten drei Zeichen der Zeile eine Zahl steht. Das Eingabeelement wird auf das Ausgabeelement `PositionAusgabe` abgebildet:

```
area Position: start(regStream(“\rb\fill*\num+”, 3)),
               end(regSL(“_*\cr\rb”)),
               node(“PositionAusgabe”):
```

3.4 Zusammenfassendes Beispiel

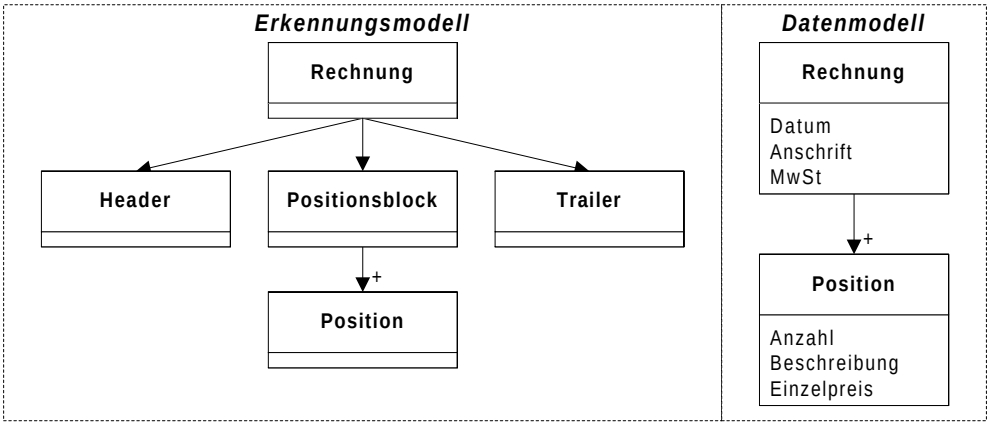
Zum Abschluß der Bereichsbeschreibung nochmals ein umfassendes Beispiel für eine mögliche Beschreibung unserer Beispielrechnung:

```
RECHNUNG

Jens Elei
Kastorpfaffenstr. 1
56068 Koblenz

Datum: 1.12.1999


Anz ArtName                      Einzelpreis[DM] Gesamtpreis[DM]
-------------------------------------------------------------
 1  PS/2 Adapter                    10,00           10,00
 2  Mousepad                        10,00           20,00
 1  Computer                        1.989,00        1.989,00
    Powerline X 500
-------------------------------------------------------------
Gesamtsumme                                        2.019,00
16% MwSt.                                          323,04
Gesamtbetrag                                       2.342,04
```

Rechnung: Header, Positionsblock, Trailer

Es gelte folgendes Erkennungsmodell und Datenmodell:

Erkennungsmodell und Datenmodell der Beispielrechnung



Dann können folgende Bereichsbeschreibungen formuliert werden:

```
area Rechnung: start(regStream(“.*\rbRECHNUNG”, 100)),
               node(“Rechnung”):
```

Der Bereich `Rechnung` wird nur durch seinen Anfang beschrieben und wird in der Ausgabe mit einem gleichnamigen XML Element synchronisiert. Beim Betreten des Bereiches wird der Eingabezeiger per `\rb` auf den ersten Buchstaben des Wortes gesetzt.

```
area Header:   start(posSL(0)),
               end(regML(“.*\rb-{10,100}\cr”, 10)):
```

Die Beschreibung des Starts von `Header` bleibt praktisch funktionslos, da er an der gleichen Stelle beginnen soll wie der übergeordnete Bereich `Rechnung`. Der Bereich endet vor der ersten Trennlinie, die innerhalb der ersten 10 Zeilen der Rechnung liegen muß. Die Linie muß mindestens 10 Zeichen und maximal 100 Zeichen lang sein.

```
area Positionsblock: start(regML(“.*-{10,80}\cr\rb”, 10)),
                     end(regML(“.*\cr\rb-{10,80}”, 100))
```

Der `Positionsblock` liegt zwischen beiden Trennlinien und darf maximal 100 Zeilen umfassen.

```
area Trailer:  start(regStream(“-{10,100}\cr\rbGesamtsumme”), 120):
```

Der `Trailer` beginnt dann nach der zweiten Trennlinie vor dem Wort „`Gesamtsumme`“.

```
area Position: start(regStream(“\rb\fill*\num+”, 3)),
               node(“Position”):
```

Eine Position ist daran zu erkennen, daß innerhalb der ersten drei Zeichen eine Zahl steht. Das Ende ergibt sich durch die Verarbeitung der einzelnen Zeilen einer Tabelle. Es würde an dieser Stelle nicht ausreichen das Zeilenende als Ende der `Position` zu definieren, da ein mehrzeiliger Eintrag vorhanden ist.

Ein entsprechendes Konstrukt für diese Verarbeitung wird im folgenden Kapitel beschrieben.

3.5 Grammatik der Bereichsbeschreibung

Die zuvor erläuterten Elemente der Bereichsbeschreibung werden nun im folgenden formal in einer Grammatik formuliert:

```
AREADEFINTION ::= ‘area’ AREANAME ‘:’ S AREADEF AREANODE
```

Der `AREANAME` referenziert einen Bereichsbezeichner der Eingabestruktur und `S` wird auf ein oder mehrere Trennzeichen expandiert. Darauf folgt direkt die Spezifikation von Start- und Endbedingungen:

```
AREADEF ::= ‘start(’ BASEFUNC ‘)’ S AREAEND
```

Im Gegensatz zur Anfangsbeschreibung ist die Beschreibung des Bereichsendes optional:

```
AREAEND ::= (ε) | (‘,’ S ‘end(’ BASEFUNC ‘)’)
```

Um diesem Bereich optional einem XML Element der Ausgabestruktur (`XMLELEMENTNAME`) zuzuweisen kann die folgende Klausel angebracht werden:

```
AREANODE ::= ((‘:’) | (‘,’ S ‘node(”’ XMLELEMENTNAME ‘”):’)) PROCESSING
```

Die Basisfunktionen (`BASEFUNC`) können dann durch reguläre Ausdrücke oder Positionen die jeweils benötigten Daten der Beschreibung liefern.

An der Stelle des hier noch nicht aufgelösten Elementes `PROCESSING` wird im nächsten Kapitel dann die Beschreibung um Konstrukte zur Verarbeitung der einzelnen Bereiche erweitert.

4 Bereichsverarbeitung

Für einen einzelnen Bereich kann immer spezifiziert werden, wie dieser verarbeitet werden soll. Eine solche Angabe besteht immer aus einer Verarbeitungsmethode, der in einem folgenden Anweisungsblock zum Teil einzelne Ausführungsschritte zugeordnet werden.

Für die Verarbeitung der Bereiche existieren verschiedene Konstrukte, die häufig auftretenden Organisationsformen der Daten entsprechen. Sie wurden im Laufe der Eingabeanalyse beschrieben, und werden nun klassifiziert behandelt nach:

- Bereichsstrukturierungen

  Ein Bereich kann automatisch vom Konverter in Unterbereiche aufgegliedert werden, indem ein regulärer Ausdruck angegeben wird, der die möglichen Strukturierungen in Unterbereiche beschreibt.

- Bereichszuweisungen

  Entspricht ein Bereich auf unterster Ebene einem gesuchten Datenelement, oder können einzelne Datenelemente innerhalb dieses Bereiches direkt mit einer der Grundfunktionen beschrieben werden, dann kann diese Information einem entsprechenden Element in der Ausgabestruktur zugewiesen werden.

- Tabellen

  Bereiche die einer Tabelle entsprechen, können mit eigenen Konstrukten verarbeitet werden. Dabei wird es dann z.B. möglich mehrzeilige Tabelleneinträge zu verwalten etc. .

4.1 Bereichsstrukturierungen

Um einen Bereich vom Konverter in Unterbereiche aufspalten zu lassen, die jeweils einzeln verarbeitet werden, reicht es prinzipiell einen regulären Ausdruck über das Alphabet der Bereichsbezeichner anzugeben. Der Konverter versucht dann zur Laufzeit einzelne Bereiche nach Maßgabe des Ausdrucks zu finden, und ruft jeweils die Verarbeitung der Unterbereiche auf.

Beispiel:

Für die Verarbeitung einer Rechnung wird im Erkennungsbereich `Rechnung` der folgende reguläre Ausdruck für eine automatische Auflösung angegeben:

```
(Header, Position+, Trailer)
```

Der Konverter ruft nun zur weiteren Verarbeitung zunächst den Bereich `Header` auf, gefolgt von mindestens einer Iteration durch `Position`, gefolgt von einer einzelnen Verarbeitung in `Trailer`.

Da der Ausdruck über Bereichsbezeichner formuliert wird, die nicht direkt auf die Eingabe gematcht werden können, ist hier ein eigenes *Match-Kriterium* notwendig. Für dieses Match-Kriterium können die schon bekannten Bereichsmerkmale verwendet werden, d.h. eine folgende Eingabe kann auf einen Eingabebereich gematcht werden, wenn diese die Merkmale erfüllt.

Zusätzlich sind auch nicht alle zuvor definierten Erweiterungen der regulären Ausdrücke in diesem Kontext sinnvoll, weshalb die Operatoren für diese Beschreibung beschränkt werden. Zulässig sind nur Sequenz (`,`), Iterationen (`*` und `+`), Alternative (`|`) und optionale Elemente (`?`). Für die Expansion der Ausdrücke gelten folgende Regelungen:

- Die Operatoren `*`,`+` und `?` werden auf das längste gültige Präfix gematcht.
- Alternativen werden in ihrer Reihenfolge von links nach rechts gematcht.

In Analogie zu klassischen prozeduralen Programmiersprachen implementieren die Operatoren praktisch Funktionsaufrufe in Sequenzen, Fallunterscheidungen und Schleifen.

Damit eine korrekte Funktionsweise des Prozesses sichergestellt werden kann, sollten der Ausdruck und die Unterbereiche einige Anforderungen erfüllen:

- Da die Erkennung der Bereiche prädiktiv, d.h. ohne Backtracking erfolgt, sollte das sich ergebende Inhaltsmodell unzweideutig sein. Für die Auflösung eines nicht prädiktiv zu erkennenden Ausdrucks ist der Entwickler des Skripts verantwortlich. (vgl. hierzu: [RECXMLEN], Anhang E)
- Um die korrekte Funktionsweise sicherzustellen, sollte der Entwickler des Skripts zusätzlich darauf achten, daß die Merkmale der einzelnen Bereiche stark genug beschränkt sind, um sich gegenseitig auszuschließen.

Um dann die Anwendung dieser Methode der Bereichsverarbeitung zu spezifizieren, wird der Bereichsbeschreibung ein Konstrukt der folgenden Art zugeordnet:

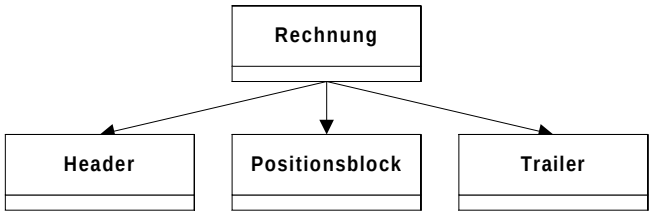
`asSubAreas(``<`*`regulärer Ausdruck über die Bereichsbezeichner`*`>`**`);`**

Beispiel: (bezogen auf die Beispielrechnung)

```
area Rechnung: start(regStream(“.*\rbRECHNUNG”, 100)),
               node(“Rechnung”):
               asSubAreas(Header, Positionsblock, Trailer)
```

Damit wird dann der folgende Teilbaum des Erkennungsmodells beschrieben, wobei die Bereiche `Header`, `Positionsblock` und `Trailer` natürlich noch zusätzlich im Skript definiert werden müssen:

Durch* asSubAreas *beschriebenes Erkennungsmodell



Der genaue Ablauf des Aufrufs der einzelnen Bereiche wurde bereits in Kapitel 1.3 beschrieben. Wichtig ist in diesem Kontext, daß nicht unbedingt ein Fehler ausgelöst wird, wenn die Merkmale eines Bereiches nicht zutreffen. Dies spiegelt einfach den Fall wieder, daß dieser Bereich nicht auf die Eingabe gematcht werden konnte. Ein Fehler wird nur dann ausgelöst, wenn nicht der gesamte Ausdruck als Bereiche über die Eingabe expandiert werden kann.

4.2 Bereichszuweisungen

Eine Bereichszuweisung erfolgt immer auf Ebene der Blätter des Erkennungsbaumes. Wurde ein Bereich so weit eingeschränkt, daß er genau einem Datenelement der Ausgabe entspricht, oder können einzelne Datenelemente innerhalb dieses Bereiches direkt mit einer der Grundfunktionen beschrieben werden, dann kann man für dieses Blatt folgendes angeben:

```
asSingleValues {
    (AddElement  (<XMLElementName>   , <DescRes> [, <trimStyle>]); |
     AddAttribute(<XMLAttributeName> , <DescRes> [, <trimStyle>]); |
     Skip        (<DescRes>);                                       )+
}
```

Das Schlüsselwort `asSingleValues` leitet einen Anweisungsblock ein, dessen Semantik die Menge der zulässigen integrierten Anweisungen auf drei Bestandteile beschränkt, die in ihrer angegeben Reihenfolge einmalig und sequentiell ausgeführt werden:

- **`AddElement(`**`<XMLElementName>, <DescRes> [, <trimStyle>]`**`);`**

  Der durch eine der Grundfunktionen in `<DescRes>` beschriebene Text wird zwischen ein Start- und End-Tag eines XML Elementes mit dem Namen aus `<XMLElementName>` in die Ausgabe übernommen. Durch die optionale Angabe eines Wertes von `<trimStyle>` können Manipulationen am Text erfolgen, wie z.B. das Eliminieren führender bzw. folgender Leerstellen[1].

  Der Eingabezeiger wird durch diese Anweisung nicht manipuliert, allerdings wird ein Fehler ausgelöst, wenn in `<DescRes>` angezeigt wird, daß der Wert nicht gefunden wurde (`found=false`). Der globale Bereichsstack wird nicht beeinflußt, d.h. die Ausführung bleibt in jedem Fall lokal.

- **`AddAttribute(`**`<XMLAttributeName>, <DescRes> [, <trimStyle>]`**`);`**

  Diese Anweisung funktioniert genauso wie `AddElement`, mit dem einzigen Unterschied, daß die Ausgabe nun nicht zwischen ein Start- und End-Tag erfolgt, sondern der Wert einem Attribut zugeordnet wird. Dazu steht im Parameter `<XMLAttributeName>` ein, durch einen XML Elementnamen qualifizierter, Bezeichner eines Attributes.

  Der Eingabezeiger und der Bereichsstack werden nicht beeinflußt. Wenn der Wert nicht gefunden wurde (`found=false`), wird ein Fehler ausgelöst.

- **`Skip(`**`<DescRes>`**`);`**

  Der durch eine Grundfunktion beschriebene Bereich wird ohne Datenübernahme übersprungen. Dies geschieht dadurch, daß der Eingabezeiger auf den Wert von `end` aus `<DescRes>` gesetzt wird. Dabei wird wiederum ein Fehler ausgelöst, wenn `found=false` gilt.

  Der globale Bereichsstack wird nicht verändert.

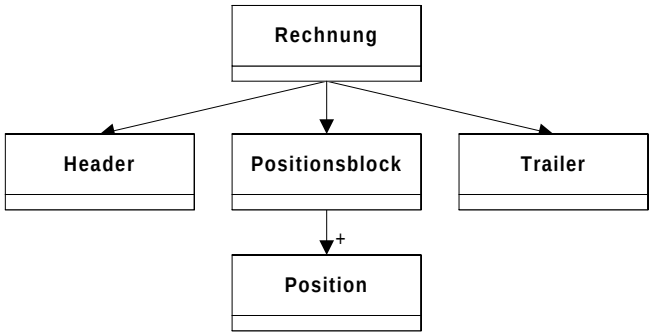
  Die `Skip` Funktion ist letztendlich ein Hilfskonstrukt, das es ermöglicht einzelne Datenelemente nacheinander aus einem Bereich zu extrahieren, wenn sie nicht einfach vom Bereichsanfang aus beschrieben werden können.

[1] Eine Menge von Werten und Manipulationen ist noch zu definieren.

Beispiel: (bezogen auf die Beispielrechnung)

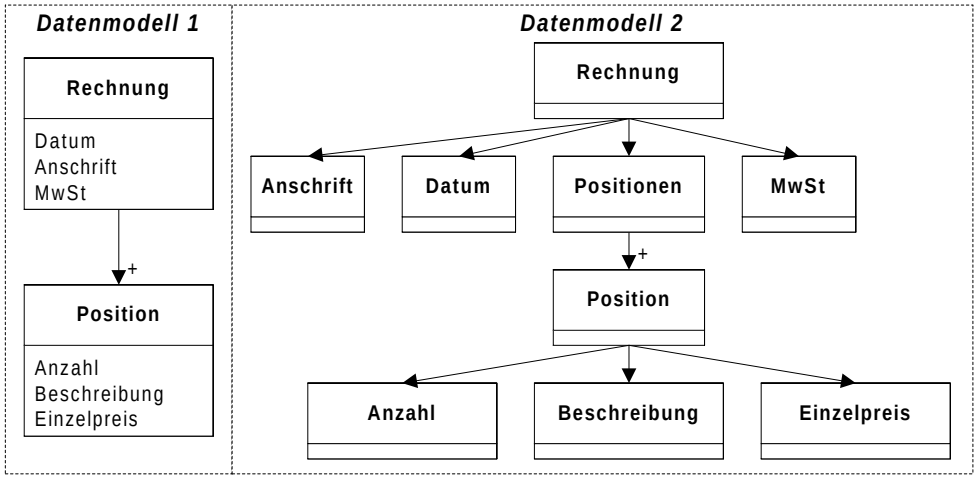
Benutzt man für die Beispielrechnung das folgende Erkennungsmodell:

Erkennungsmodell der Beispielrechnung



Dann kann man mit Hilfe von `AddAttribute` und `AddElement` prinzipiell zwei verschiedene Varianten einer Ausgabe beschreiben:

Datenmodelle der Beispielrechnung für eine Bereichszuweisung



Im ersten Fall entsprechen die extrahierten Datenelemente Attributen eines XML Elementes, und im zweiten Fall wird jedem Datenelement selbst ein XML Element zugeordnet. Die inneren Knoten fassen im zweiten Fall Elemente zusammen. Die Markups der Blätter werden in diesem Fall nicht durch ein `node` Konstrukt erzeugt, sondern durch Angabe der `AddElement` Operation.

Es wird deutlich, daß die eigentlichen Daten sowohl als Attribute, als auch als Text zwischen Markups, abgelegt werden können. Welche Variante verwendet wird bleibt offen. Es erscheint aber sinnvoll immer nur eine Variante in einem Skript zu verwenden.

Der Erkennungsbereich `Header` sei nun durch eine Beschreibung auf folgendes eingegrenzt:

Header der Beispielrechnung

```
RECHNUNG

Jens Elei
Kastorpfaffenstr. 1
56068 Koblenz

Datum: 1.12.1999
```

Der Eingabezeiger steht zu Beginn auf dem „`R`“ von Rechnung, da er beim Betreten eines Bereiches automatisch auf den Bereichsanfang gesetzt wird.

Dann können Extraktionen exemplarisch wie folgt beschrieben sein:

- Für das erste Datenmodell sind in der DTD der Ausgabe entsprechend folgende Definitionen enthalten:

```
<!ELEMENT Rechnung (Position+)>
<!ATTLIST Rechnung
      Datum CDATA #IMPLIED
      Anschrift CDATA #IMPLIED
>
```

  Im Skript kann als Bereichsblock für `Header` dann formuliert werden:

```
area Header: start(<DescRes>), end(<DescRes>): asSingleValues {
  AddAttribute("Rechnung.Datum", regML(".*Datum:\lb[\num\.]+", 10));
  Skip(posML(3,0));
  AddAttribute("Rechnung.Anschrift", regSL("\lb_*\rb\cr"));
}
```

  Dies erzeugt innerhalb der Ausgabe das folgende Fragment:

```
<Rechnung Datum="1.12.1999" Anschrift="Kastorpfaffenstr. 1">
```

- Für das zweite Datenmodell erfolgt sei entsprechend folgendes innerhalb der DTD definiert:

```
<!ELEMENT Rechnung  (Datum, Anschrift, Positionen, Trailer)>
<!ELEMENT Datum     (#PCDATA)>
<!ELEMENT Anschrift (#PCDATA)>
```

  Im Skript wird für `Header` nun formuliert:

```
area Header: start(<DescRes>), end(<DescRes>): asSingleValues {
  AddElement("Datum", regML(".*Datum:\lb[\num\.]+\rb", 10));
  Skip(posML(3,0));
  AddElement("Anschrift", regSL("\lb_*\rb\cr"));
}
```

  Die Ausgabe ist dann zwischen einzelne Tags verschachtelt:

```
<Rechnung>
    <Datum>1.12.1999</Datum>
    <Anschrift>Kastorpfaffenstr. 1</Anschrift>
    …
</Rechnung>
```

4.3 Tabellen

Bei Tabellen kann man zwei mögliche Anordnungsweisen einzelner Zellen beobachten:

Name	Nummer
Klaus	1
Peter	3
Karl	5
…	…

↓

Name	Klaus	Peter	Karl	…
Nummer	1	3	5	…

→

Tabellen können also entweder im zweidimensionalen Raum vertikal oder horizontal unbegrenzt viele Elemente enthalten, und haben dann eine feste Anzahl von Spalten bzw. Zeilen.

Dies macht ihre Verarbeitung innerhalb des Konverters so weit unterschiedlich, daß zwei verschiedene Mengen von Konstrukten, eine für vertikal wachsende und eine für horizontal wachsende Tabellen sinnvoll werden.

4.3.1 Vertikal wachsende Tabellen

Die Beschreibung vertikal wachsender Tabellen, durch feste Spaltenbreiten per Positionsansatz, oder durch flexible Spaltenbreiten per Musterbeschreibung, erscheint zunächst relativ einfach. Erschwert wird diese Aufgabe aber durch folgende Fälle der Eingabeanalyse:

- ***Umgebrochene Eingabezeilen***: Lange Tabellenzeilen werden möglicherweise auf mehrere Zeilen umgebrochen, um sie einer maximalen Darstellungsbreite anzupassen. Bei festen Spaltenbreiten entsteht dann auch eine feste Anzahl von Zeilen, nicht aber bei flexiblen Spalten, die z.B. durch Trennzeichen wie Kommata, Semikolons etc., beschrieben werden.

  Beispiele umgebrochener Eingabezeilen

```
Anz ArtName                          Anz;ArtName;Einzelpreis[DM];
Einzelpreis[DM]   Gesamtpreis[DM]    Gesamtpreis[DM]
-----------------------------------  --------------------------
 1  PS/2 Adapter                     1;PS/2 Adapter;10,00;10,00
10,00             10,00              2;Mousepad;10,00;20,00
 2  Mousepad                         1;Computer Powerline X 500;
10,00             20,00              1.989,00;1.989,00
 1  Computer Powerline X 500
1.989,00          1.989,00
```

- ***Mehrzeilige Datenelemente***: Dabei umfaßt eine Zeile der Eingabe alle Spalten der Tabellen, allerdings können bei fester Breite einer Spalte zu lange Datenelemente in Folgezeilen umgebrochen werden. In folgendem Beispiel ist so der Artikelname „`Computer Powerline X 500`“ mehrzeilig:

  Beispiel mehrzeiliger Datenelemente

```
Anz ArtName                     Einzelpreis[DM] Gesamtpreis[DM]
-------------------------------------------------------------
 1  PS/2 Adapter                   10,00           10,00
 2  Mousepad                       10,00           20,00
 1  Computer                       1.989,00        1.989,00
    Powerline X 500
```

  In diesem Fall ist die Anzahl der Zeilen für einen Datensatz flexibel. Diese Anzahl kann aber nicht wie bei flexiblen Spalten zuvor durch die Anzahl der Elemente erkannt werden, sondern erfordert ein flexibles Kriterium. Darüber hinaus sind die Teile einer Zelle im sequentiellen Strom fragmentiert.

Erneut muß die Verarbeitung auf sehr unterschiedliche Weise erfolgen, weshalb nochmals eine Aufteilung in zwei verschiedene Verarbeitungsfunktionen vertikaler Tabellen erfolgt, die in den folgenden beiden Kapiteln beschrieben werden.

4.3.1.1 Tabellen mit umgebrochenen Eingabezeilen

Bei Tabellen mit umgebrochenen Eingabezeilen sind einzelne Zellen der Tabelle fortlaufende Zeichenketten, aber eine Tabellenzeile (ein Datensatz) kann auf mehrere Zeilen der Eingabe umgebrochen sein. Dies entspricht dem Fall *umgebrochener Eingabezeilen*, kann aber genauso für Tabellen genutzt werden, deren Datensätze immer genau einer Zeile entsprechen.

Für diese Variante einer Tabelle ist prinzipiell nur eine Beschreibung erforderlich, wie jede einzelne Spalte erkannt werden kann, und welchem Datenelement der Ausgabe die extrahierte Zeichenkette zugeordnet werden soll.

Die Beschreibung eines einzelnen Elementes kann wieder mit Hilfe der bekannten Grundfunktionen erfolgen, die dann in einer Aufzählung einzelner Spalten einem Attribut, einem XML Element oder einem weiteren Bereich zugeordnet werden.

Die Verarbeitungsmethode `asMLRVertTable` („as multiple lines per row vertical table“) verschachtelt insgesamt eine Folge von Anweisungen in der folgenden Form:

```
asMLRVertTable ([<OutputElementName1>]) {
     (AddAttribute(<OutputAttributeName>, <DescRes> [, <trimStyle>]); |
      AddElement  (<OutputElementName2> , <DescRes> [, <trimStyle>]); |
      AddArea     (<InputAreaName>      , <DescRes>);                 |
      Skip        (<DescRes>);                                         )+
}
```

Diese Methode sorgt nun dafür, daß die Anweisungen innerhalb des Blocks solange sequentiell in Iterationen durchlaufen werden, bis durch die Verarbeitung der Anweisungen das Bereichsende erreicht wird. In jeder Iteration kann durch die Angabe eines optionalen XML Elementes in *<OutputElementName1>* das bezeichnete Element in der Ausgabe erzeugt werden. Dieser Bestandteil der Abbildung von Eingabe auf Ausgabe spiegelt die Tatsache wider, daß eine Tabelle meist aus einzelnen Datensätzen besteht, die durch ein Markup zusammengefaßt werden.

Innerhalb des Blocks beschränkt die Methode die Anweisungsfolge auf folgende Bestandteile:

- **`AddAttribute(`***`<OutputAttributeName>`*`, <DescRes> [, <trimStyle>]);`**

  Die Zeichenfolge `str` aus `<DescRes>` wird nach optionalen Manipulationen per `<trimStyle>` als Attribut mit dem Namen *`<OutputAttributeName>`* in der Ausgabe abgelegt, wenn `found=true` gilt. Für `found=false` wird ein Fehler ausgelöst.

  Danach wird der Eingabezeiger auf den Wert von `end` gesetzt. Der globale Bereichsstack wird nicht verändert.

- **`AddElement(`***`<OutputElementName2>`*`, <DescRes> [, <trimStyle>]);`**

  Die Zeichenfolge `str` aus `<DescRes>` wird nach optionalen Manipulationen per `<trimStyle>` zwischen Start- und End-Tag des XML Elementes mit dem Namen *`<OutputElementName2>`* in der Ausgabe abgelegt, wenn `found=true` gilt. Für `found=false` wird ein Fehler ausgelöst.

  Danach wird der Eingabezeiger auf den Wert von `end` gesetzt. Der globale Bereichsstack wird nicht verändert.

- **`AddArea(`***`<InputAreaName>`*`, <DescRes>);`**

  Durch diese Anweisung wird die Verarbeitung in einen untergeordneten Bereich mit Namen *`<InputAreaName>`* verzweigt. Dazu werden lokale Daten auf den Bereichsstack gelegt, und dann Werte an den untergeordneten Bereich übergeben. Diese sind zum einen der Wert aus `start` als Eingabezeiger des Unterbereiches, und zum anderen der Wert von `end` als übergeordnete Bereichsgrenze (vgl. Kapitel 1.2).

  Ist die Verarbeitung im Unterbereich abgeschlossen, dann werden die lokalen Daten wiederhergestellt und der Eingabezeiger wird auf `end` gesetzt.

- **`Skip(`***`<DescRes>`*`);`**

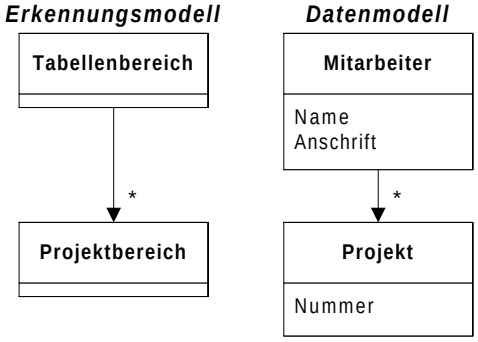
  Diese Anweisung setzt den Eingabezeiger auf den errechneten Wert von `end` aus *`<DescRes>`* und nimmt ansonsten keine Änderungen oder Ausgaben vor.

  Sie dient dazu einzelne Teile der Eingabe zu überspringen, wenn diese nicht extrahiert werden, oder um den Eingabezeiger nach dem letzten Element auf den Startpunkt für die Iteration des nächsten Datensatzes zu setzen.

Beispiel: (*Variante 4* aus der Eingabeanalyse)

Mitarbeitername	Anschrift	arbeitet an Projekt Nummer
Müller	Dorfplatz	4711, 0815
Schmidt	Rheinau 2	4711, 0815
Mayer	Rosenstraße 3	0815, 42, 007

Jede Spalte sei in der Eingabe 30 Zeichen breit. Die letzte Spalte nimmt eine Sonderrolle ein, da in ihr mehrere Datenelemente angeordnet sein können und sie bis zum Zeilenende reicht. Ihre Verarbeitung wird deshalb in einem gesonderten Bereich vorgenommen. Das Erkennungsmodell wird durch die Verarbeitung implementiert, und das Datenmodell sei wie folgt definiert:



Durch eine passende Bereichsbeschreibung sei die Tabelle auf die drei eigentlichen Datenzeilen (also ohne Kopfzeile) eingeschränkt. Es ergibt sich eine Spezifikation der folgenden Art, wobei aus hier Gründen der Übersichtlichkeit teilweise auf die Angabe der Bereichsmerkmale verzichtet wird:

```
area Tabellenbereich: start(<DescRes>), end(<DescRes>):
asMLRVertTable (“Mitarbeiter”) {
    AddAttribute(“Mitarbeiter.Name”     , posSL(30));
    AddAttribute(“Mitarbeiter.Anschrift”, posSL(30));
    AddArea(“Projektbereich”            , regSL(“\lb_*\rb\cr”));
    Skip(regStream(“\cr”, 1);
}
```

Die Deklarationen von `AddAttribute` und `AddArea` ordnen je einer Spalte der Tabelle ein Ausgabeattribut oder einen untergeordneten Erkennungsbereich zu. Mit Hilfe der `Skip` Angabe wird der Umbruch am Zeilenende für die nächste Iteration übersprungen. In jeder Iteration wird ein XML Element `Mitarbeiter` erzeugt.

Durch den Verweis der Verarbeitung in Unterbereiche werden auch komplexe Strukturen handlebar. So kann hier der Unterbereich `Projektbereich` dazu genutzt werden, eine Aufzählung von Projektnummern zu beschreiben, die dann in einem weiteren Unterbereich jeweils in ein Attribut des XML Elementes `Projekt` abgelegt werden. Dies wird durch folgendes Skriptfragment realisiert:

```
area Projektbereich: start(<DescRes>), end(<DescRes>):
asSubAreas(Projektnummer+);

area Projektnummer: start(regSL(“\rb\num+”)),
                    end(regSL(“\num+(, |\cr)”)),
                    node(“Projekt”):
asSingleValues {
    AddAttribute(“Projekt.Nummer”, regSL(“\num+”));
}
```

Wenn flexible Spaltenbreiten benutzt werden, dann kann nicht vorausgesagt werden, nach welcher Spalte ein Zeilenumbruch erfolgt, d.h. die Verarbeitung muß flexibel auf auftretende Umbrüche reagieren. Dies gilt in anderer Ausprägung auch für Sonderfälle bei festen Spalten:

Probleme bei fester Breite der letzten Spalte

1	2	3	4	5	6	7	8	
a	b	c	d	↵				
a	b	c	d	e	f	g	h	↵

Spaltenbreite = 8 Zeichen

1.) Die letzte Spalte kann durch einen Umbruch kürzer sein.

2.) Die letzte Spalte kann mit dem Umbruch verlängert sein.

Als Lösungen für diese Fälle können folgende Strategien angewendet werden:

- Taucht im sequentiellen Zeichenstrom nach einer festen Spalte ein Zeilenumbruch auf, dann kann dieser durch eine `Skip` Deklaration übersprungen werden. Dadurch werden umgebrochene Zeilen fester Breite verarbeitet.
- Das Problem bei flexiblen Breiten kann entweder genauso gelöst werden, oder ein möglicher Umbruch wird ins Muster integriert.
- Für feste Spalten, die durch einen Umbruch verkürzt sein können, ist an Stelle der festen Breite ein Muster zu spezifizieren, das den Umbruch integriert. Im obigen Beispiel geschieht dies mit: `regSL("\lb_*\rb\cr")`.

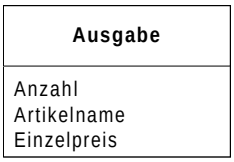
Allgemein verlangt diese Methode immer, daß sowohl Anfang als auch Ende so beschrieben sind, daß nur die eigentlichen Datenzeilen in dem Bereich vorhanden sind. Genau dieser Bereich wird dann auch bearbeitet, d.h. das Tabellenende wird durch das Bereichsende erkannt.

Im folgenden werden zusätzliche Beispiele für Spezifikationen erfaßt.

Beispiel: (mehrzeilige Tabelle mit fester Spaltenbreite)

```
Anz ArtName
Einzelpreis[DM]   Gesamtpreis[DM]
---------------------------------
 1  PS/2 Adapter
10,00             10,00
 2  Mousepad
10,00             20,00
 1  Computer Powerline X 500
1.989,00          1.989,00
```

Datenmodell (DTD)




```
area Eingabebereich: start(<DescRes>), end(<DescRes>):
asMLRVertTable ("Ausgabe") {
    AddAttribute("Ausgabe.Anzahl"     , posSL(3));
    AddAttribute("Ausgabe.Artikelname", regSL("\lb_*\rb\cr"));
    Skip(regStream("\cr", 1)
    AddAttribute("Ausgabe.Einzelpreis", posSL(16));
    Skip(regStream("_*\cr", 0));
}
```

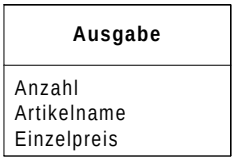
Aus einer Tabelle mit festen Spaltenbreiten werden drei Attribute extrahiert, das letzte wird übersprungen. Als Breite der ersten Spalte werden `3` Zeichen, als Breite der dritten `16` Zeichen, festgelegt. Die Zeilenumbrüche wurden jeweils in entsprechende Muster integriert. Der durch `start(<DescRes>)` und `end(<DescRes>)` beschriebene Bereich umfasse den grau hinterlegten Teil.

Am Eingabemodell besteht lediglich aus dem Bereich `Eingabebereich`, das Datenmodell sei in einer DTD definiert.

Beispiel: (mehrzeilige Tabelle mit variabler Spaltenbreite)

```
Anz;ArtName;Einzelpreis[DM];
Gesamtpreis[DM]
---------------------------
1;PS/2 Adapter;10,00;10,00
2;Mousepad;10,00;20,00
1;Computer Powerline X 500;
1.989,00;1.989,00
```

Datenmodell (DTD)




```
area Eingabebereich: start(<DescRes>), end(<DescRes>):
asMLRVertTable (“Ausgabe”) {
    AddAttribute(“Ausgabe.Anzahl”     , regStream(“\lb\num+\rb;”, 0));
    AddAttribute(“Ausgabe.Artikelname”, regStream(“;\cr?\lb_*\rb;”,0));
    AddAttribute(“Ausgabe.Einzelpreis”, regStream(“;\cr?\lb_*\rb;”,0));
    Skip(regStream(“;\cr?_*\cr”, 20));
}
```

Aus einer Tabelle mit flexiblen Spaltenbreiten werden wiederum die gleichen drei Attribute extrahiert. In die Muster wurden jeweils optionale Zeilenumbrüche integriert (`\cr?`), und es wurde bewußt `regStream` und nicht `regSL` verwendet, um die Umbrüche handhaben zu können.

Am Eingabemodell besteht wieder nur aus dem Bereich `Eingabebereich`, das Datenmodell sei wieder in einer DTD definiert.

4.3.1.2 Tabellen mit mehrzeiligen Datenelementen

Bei Tabellen mit mehrzeiligen Datenelementen können einzelne Zellen mehrere Zeilen in der Eingabe haben, aber eine Tabellenzeile (ein Datensatz) ist genau so breit wie der Erkennungsbereich. Dies entspricht dem beschriebenen Fall *mehrzeiliger Datenelemente*.

Wenn ein Element auf mehrere Zeilen umgebrochen wird, dann ergibt sich die spezielle Aufgabenstellung dieser Methode aus verschiedenen Aspekten:

- Eine Zelle muß aus mehreren, im sequentiellen Strom getrennten, Fragmenten zusammengesetzt werden:

Beispiel zur Fragmentierung einer einzelnen Zelle

```
1  Computer                    1.989,00        1.989,00
   Powerline X 500
```

sequentieller Strom:

```
1 Computer →→ 1.989,00 → 1.989,00 ↵→ Powerline X 500 ...
```

→= Tabulator (in der Vorverarbeitung expandiert, hier nur als vereinfachte Darstellung genutzt)

- Es muß ein Kriterium definiert werden, mit dem es möglich ist festzustellen, wie viele Zeilen ein Datensatz umfaßt:

Beispiel zur Bestimmung der letzten Eingabezeile einer Zelle

```
 2  Mauspad,
    ergo, blau          10,00          20,00
 1  Computer
    Powerline
    X 500            1.989,00       1.989,00
```

Anhand der enthaltenen oder eben nicht enthaltenen Daten in der aktuellen oder der folgenden Zeile kann erkannt werden, welche Zeilen zusammen gehören. Hier sind z.B. folgende Bedingungen denkbar:
- Solange in der ersten Spalte keine Zahl in der folgenden Zeile steht, gehört die folgende Zeile noch zum aktuellen Datensatz.
- Die letzte Zeile wird daran erkannt, daß in der letzten zugehörigen Zeile in der vierten Spalte eine Zahl steht.

(das jeweils beschriebene Fragment ist markiert)

Ein Fragment einer Zelle signalisiert immer anhand der enthaltenen Daten, wann der nächste Datensatz beginnt[2]. Für die formale Definition eines Kriteriums sind dabei i.A. folgende Informationen entscheidend:

- Auf Basis welcher Spalte das Ende eines Datensatzes zu erkennen ist.
- Ob das Zutreffen einer Bedingung für die nächste Zeile das Ende des aktuellen oder den Anfang des folgenden Datensatzes signalisiert.
- Die eigentliche Bedingung, hier der Vergleich mit einem Muster.

[2] Wenn nicht immer eine feste Anzahl von Zeilen pro Datensatz vorliegt, oder einzelne Datensätze mit einer zusätzlichen Trennzeile etc. visualisiert werden. Diese Fälle können aber in anderer Form ebenfalls spezifiziert werden.

Die Verarbeitungsmethode `asMLEVertTable` („as multiple lines per element vertical table“) berücksichtigt nun diese Besonderheiten durch eine veränderte Arbeitsweise, die durch zusätzliche Parameter gesteuert wird. Sie verschachtelt wiederum die gleiche Menge von Anweisungen in der folgenden Form:

```
asMLEVertTable(<PatternCol>, <Condition>, <Pattern> [, <OutputElementName1>])
{
      (AddAttribute(<OutputAttributeName>, <DescRes> [, <trimStyle>]); |
       AddElement  (<OutputElementName2> , <DescRes> [, <trimStyle>]); |
       AddArea     (<InputAreaName>, <DescRes>);                       |
       Skip        (<DescRes>);                                         )+
}
```

Die Methode sorgt nun im Gegensatz zu `asMLRVertTable` nicht nur für Iterationen durch die Anweisungsfolgen, sondern faßt einzelne Iterationen noch in besonderer Weise zusammen:

Die auszugebenden Datenelemente der einzelnen Iterationen werden so lange immer wieder um das Fragment der Folgezeile erweitert, bis durch das spezifizierte Kriterium der Beginn eines neuen Datensatzes erkannt wird. Die Ausgabe in das optional angegebene XML Element mit Namen *<OutputElementName1>* erfolgt damit erst dann, wenn ein Datensatz abgeschlossen ist. Die Iteration und der letzte Datensatz werden durch das Bereichsende terminiert.

Die im Vergleich zu `asMLRVertTable` zusätzlichen drei Parameter entsprechen genau den zuvor motivierten Bestandteilen des Kriteriums für eine Trennung einzelner Datensätze. Ihre genaue Bedeutung ist wie folgt festgelegt:

- Mit *<PatternCol>* wird die Spalte festgelegt, auf die das Kriterium angewendet werden soll. Es handelt sich um eine Ganzzahl mit `1` Basis. Die per `Skip` übersprungenen Elemente werden mitgezählt, damit auch auf sie ein Muster angewendet werden kann. Die Angabe von `0` für *<PatternCol>* faßt alle Zeilen des Bereiches zu einem Datensatz zusammen.
- Für *<Condition>* existieren zwei zulässige Werte:
  - Eine `0` signalisiert, daß die letzte Zeile eines Datensatzes mit dem Muster übereinstimmt.
  - Eine `1` signalisiert, daß die erste Zeile eines neuen Datensatzes mit dem Muster übereinstimmt.
- Der Parameter *<Pattern>* enthält die Zeichenfolge eines regulären Ausdrucks, der innerhalb der Zeichenkette der Spalte expandiert wird. Kann er expandiert werden, dann stimmt das Muster mit der Spalte überein, und die Zeile entspricht je nach *<Condition>* der letzten Zeile eines Datensatzes oder der ersten Zeile des folgenden Datensatzes.

Die einzelnen Anweisungen funktionieren wieder genau wie bei `asMLRVertTable`, mit der bereits beschriebenen Erweiterung, daß Ausgaben erst vorgenommen werden, wenn ein kompletter Datensatz gelesen wurde. Lediglich für `AddArea` ergibt sich dabei eine Besonderheit. Zu Gunsten der lokalen Vollständigkeit der Beschreibung werden aber nochmals alle Anweisungen im folgenden zusammen beschrieben.

Die Anweisungen innerhalb des Blocks der Methode erfüllen folgende Aufgaben:

- **AddAttribute(**<*OutputAttributeName*>, <*DescRes*> [, <*trimStyle*>]**);**

  Die Zeichenfolge str aus <DescRes> wird nach optionalen Manipulationen per <trimStyle> als Attribut mit dem Namen <*OutputAttributeName*> für die Ausgabe gespeichert, wenn found=true gilt. Für found=false wird ein Fehler ausgelöst. Die eigentliche Ablage erfolgt nach Konkatenation mit folgenden Fragmenten, bis das Ende eines Datensatzes erkannt wurde.

  Nach jeder Ausführung wird der Eingabezeiger auf den Wert von end gesetzt. Der globale Bereichsstack wird nicht verändert.

- **AddElement(**<*OutputElementName2*>, <*DescRes*> [, <*trimStyle*>]**);**

  Die Zeichenfolge str aus <DescRes> wird nach optionalen Manipulationen per <trimStyle> für die Ausgabe zwischengespeichert, wenn found=true gilt. Für found=false wird ein Fehler ausgelöst. Die eigentliche Ablage erfolgt nach Konkatenation mit folgenden Teilen zwischen Start- und End-Tag des in <*OutputElementName2*> angegebenen XML Elementes.

  Nach jeder Ausführung wird der Eingabezeiger auf den Wert von end gesetzt. Der globale Bereichsstack wird nicht verändert.

- **AddArea(**<*InputAreaName*>, <*DescRes*>**);**

  Durch diese Anweisung wird die Verarbeitung in einen untergeordneten Bereich mit Namen <*InputAreaName*> verzweigt. Dazu werden lokale Daten auf den Bereichsstack gelegt, und dann Werte an den untergeordneten Bereich übergeben.

  Genau dabei ergibt sich ein entscheidender Unterschied im Vergleich zur vorherigen Datenübergabe: Da eine Zelle nun aus mehreren Fragmenten im Eingabestrom besteht, kann nicht einfach der Eingabezeiger übergeben werden. Vielmehr werden die einzelnen Fragmente, jeweils durch einen Zeilenumbruch getrennt, zu einem temporären Eingabestrom konkateniert.

  An den Unterbereich wird dann als Eingabezeiger ein Zeiger auf den Start dieses temporären Bereiches übergeben. Als Wert für die übergeordnete Bereichsgrenze wird entsprechend das Ende des temporären Bereiches gewählt (vgl. Kapitel 1.2).

  Ist die Verarbeitung im Unterbereich abgeschlossen, dann werden die lokalen Daten wiederhergestellt und der Eingabezeiger wird auf den Wert von end im ursprünglichen Eingabestrom gesetzt. Der temporär erzeugte Eingabebereich wird verworfen.

- **Skip(**<*DescRes*>**);**

  Diese Anweisung setzt den Eingabezeiger auf den errechneten Wert von end aus <*DescRes*> und nimmt ansonsten keine Änderungen oder Ausgaben vor.

  Sie dient dazu einzelne Teile der Eingabe zu überspringen, wenn diese nicht extrahiert werden, oder um den Eingabezeiger nach dem letzten Element auf den Startpunkt für die Iteration des nächsten Datensatzes zu setzen.

Die Manipulationen per <trimStyle> sind noch genau zu beschreiben. Da hier einzelne Fragmente zusammengesetzt werden, ist mindestens an folgende Varianten zu denken:

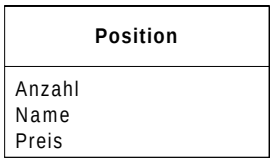
- Führende und folgende Füllzeichen werden eliminiert.
- Führende und folgende Füllzeichen werden eliminiert, aber zwischen einzelne Teile wird jeweils eine Leerstelle eingefügt.

Beispiel:

***Beispiel zu* asMLEVertTable**

```
  2  Mauspad,
     ergo, blau          10,00        20,00
  1  Computer
     Powerline
     X 500            1.989,00     1.989,00
```

***Datenmodell* (DTD)**




```
area EingabePosition: start(<DescRes>), end(<DescRes>):
asMLEVertTable(1, 1, "_?\num+_?", "Position") {
    AddAttribute("Position.Anzahl", posSL(3));
    AddAttribute("Position.Name"  , posSL(15);
    AddAttribute("Position.Preis" , posSL(10));
    Skip(regStream("_*\cr", 0));
}
```

In diesem Fall wird das Ende über die erste Spalte erkannt. Ein Datensatz endet nach Maßgabe dieser Parameter genau dann, wenn das Muster (mehrere Ziffern) in der nächsten Zeile expandiert werden kann. Dies ist hier genau dann der Fall, wenn in der Folgezeile innerhalb der ersten drei Zeichen eine Zahl steht.

Das skizzierte Datenmodell sei entsprechend in einer DTD definiert. Als Ausgabe in einer XML Datei erzeugt der Konverter dann folgendes:

```
<Position Anzahl="2" Name="Mauspad, ergo, blau", Preis="10,00"/>
<Position Anzahl="1" Name="Computer Powerline X 500", Preis="1.989,00"/>
```

Durch die besondere Arbeitsweise von `AddArea`, die nun durch einen Zeilenumbruch verbundene Fragmente als Unterbereiche übergibt, ergibt sich eine wichtige Neuerung für die Beschreibung von Bereichen:

Die durch Spalten beschriebenen Bereiche entsprechen benachbarten Bereichen in der zweidimensionalen Darstellung der Eingabe. Damit erhält man die Möglichkeit durch die Definition einer Spalte auch vertikale Teilungen der Eingabebereiche zu verwenden.

Beispiel:

Innerhalb einer Eingabedatei sei die Kundennummer und Anschrift des Kunden neben einer Tabelle mit den einzelnen Rechnungspositionen angeordnet:

***Beispiel benachbarter Bereiche durch* asMLEVertTable**

```
                 Anz;ArtName;Einzelpreis[DM];
KundenNr: 4711   Gesamtpreis[DM]
                 --------------------------
Jens Elei        1;PS/2 Adapter;10,00;10,00
Kastorpfaffen 1  2;Mousepad;10,00;20,00
56068 Koblenz    1;Computer Powerline X 500;
                 1.989,00;1.989,00
```

horizontal benachbarter Bereich eigener Verarbeitung

Tabelle mit variablen Elementen

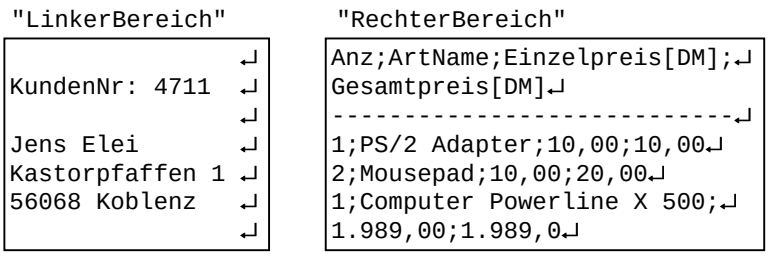
Um diese Eingabe zu verarbeiten kann nun nicht direkt eine der Verarbeitungsmethoden angewendet werden. Mit Hilfe von `asMLEVertTable` ist es aber möglich zunächst beide Teile innerhalb der Eingabe zu trennen, obwohl ihre einzelnen Fragmente gemischt im sequentiellen Eingabestrom auftauchen.

Dies geschieht durch folgende Formulierung im Skript:

```
area Gesamtbereich: start(<DescRes>), end(<DescRes>):
asMLEVertTable(0, 0, "") {
    AddArea("LinkerBereich" , posSL(16));
    AddArea("RechterBereich", regStream("_*\cr", 0);
}
```

Durch die Angabe von `PatternCol=0` wird der gesamte vertikale Umfang des Bereiches im Form von zwei temporären Unterbereichen an die Verarbeitung in anderen Bereichen weitergeleitet. Die Unterbereiche erhalten dann jeweils neue Bereiche, die adäquat mit den beschriebenen Methoden behandelt werden können.

Beispiel zusätzlicher Zeilenumbrüche



Der rechte Teil kann dann z.B. als eine Tabelle verarbeitet werden, und der linke mit Hilfe von `asSingleValues`.

4.3.2 Horizontal wachsende Tabellen

Die besondere Verarbeitung horizontal wachsender Tabellen ergibt sich aus der Tatsache, daß ein Datensatz der Tabelle nun aus einzelnen Fragmenten im sequentiellen Strom besteht, die aber nicht zu einem Datenelement verknüpft werden sollen, sondern in verschiedenen Datensätzen der Ausgabe abgelegt werden.

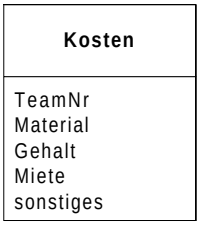
Da diese Variante im Rahmen der Fallstudie sehr selten auftritt, wird sie im folgenden auch durch ein recht einfaches Konstrukt behandelt, das folgende Einschränkung macht: Jede Spalte der Daten darf nur eine Zeile der Eingabe umfassen.

Als Beispiel wird im folgenden eine Tabelle verwendet, die per Definition häufig im Eingabestrom auftaucht, und immer die gleichen Kostenarten für eine flexible Anzahl von Teams auflistet. Exemplarisch hat sie folgende Form:

Beispiel einer horizontal wachsenden Tabelle

Kostenart	Team 1	Team 2	Team 3	Team 4	Total
Material	10.478	1.570	37.679	2.500	52.225
Gehalt	150.500	87.000	95.000	457.500	789.500
Miete	5.500	3.500	1.450	12.450	22.900
sonstiges	760	2.710	1.250	5.600	10.320

Jeder Datensatz der Tabelle soll im folgenden Ausschnitt des Datenmodells abgelegt werden:



Zunächst ist wieder ein geeigneter Bereich für die Extraktion zu definieren, wobei sich direkt eine typische Problemstellung dieser Art von Tabellen ergibt: Auch hier sind wieder Kopf und Fuß der Tabelle zu eliminieren, allerdings können diese nicht mehr wie zuvor im sequentiellen Strom am Beginn bzw. Ende abgeschnitten werden, da es sich nun um Spalten handelt.

Diese Aufgabe kann durch die zuvor erläuterte Tabellenart `asMLEVertTable` erledigt werden. Der `TabellenBereich` entspricht dann der obigen Tabelle ohne die grau hinterlegten Teile:

```
area Gesamtbereich: start(<DescRes>), end(<DescRes>):
asMLEVertTable(0, 0, "") {
    Skip(posSL(15));
    AddArea("TabellenBereich", regSL("\lb_*\rb    Total");
    Skip(regStream("_*\cr", 0));
}
```

Die restliche Aufgabe beschränkt sich darauf einzelne Spalten zu beschreiben, die in der Eingabe jeweils einer Zeile entsprechen. Die Umsetzung wird dann entsprechend automatisiert. Dies geschieht mit Hilfe der Verarbeitungsmethode `asHorzTable` in folgender Form:

```
asHorzTable ([<OutputElementName1>]) {
     (AddAttribute(<OutputAttributeName>, <DescRes> [, <trimStyle>]); |
      AddElement  (<OutputElementName2> , <DescRes> [, <trimStyle>]);  )+
}
```

Die Arbeitsweise der Methode besteht wieder aus einzelnen Iterationen, in denen jeweils optional ein Ausgabeelement *<OutputElementName1>* erzeugt wird, wobei sich aber einige wichtige Unterschiede ergeben:

- Die Ausführung jeder Anweisung einer Iteration erfolgt in einer eigenen Zeile der Eingabe, d.h. es wird lokal für jede Zeile ein eigener Eingabezeiger verwaltet, der in jeder Iteration nur genau einmal weitergesetzt wird. Die Anzahl der untersuchten Zeilen ergibt sich dann direkt aus der Anzahl der Anweisungen im Block. Die Reihenfolge der Zuordnung entspricht der sequentiellen Reihenfolge der Anweisungen im Block.
- Die Iteration einzelner Datensätze wird nicht durch das Erreichen des Bereichsende, sondern durch das Erreichen eines Zeilenumbruchs am rechten Rand jeder Zeile, terminiert. Dabei gilt, daß die gesamte Iteration erst dann beendet ist, wenn in jeder Zeile dieser Umbruch erreicht wurde.

Die Verarbeitungsmethode beschränkt die Menge der zulässigen Anweisungen nun auf nur noch zwei Elemente, da Analogien zu den anderen aufgrund der Vereinfachung nicht sinnvoll sind:

- **`AddAttribute(`***`<OutputAttributeName>`*`, <DescRes> [, <trimStyle>]);`**

  Die Zeichenfolge `str` aus `<DescRes>` wird nach optionalen Manipulationen per `<trimStyle>` als Attribut mit dem Namen *`<OutputAttributeName>`* in der Ausgabe abgelegt, wenn `found=true` gilt. Für `found=false` wird nur dann ein Fehler ausgelöst, wenn nicht das Ende der Zeile erreicht wurde.

  Danach wird der Eingabezeiger der entsprechenden Zeile auf den Wert von `end` gesetzt. Der globale Bereichsstack wird nicht verändert.

- **`AddElement(`***`<OutputElementName2>`*`, <DescRes> [, <trimStyle>]);`**

  Die Zeichenfolge `str` aus `<DescRes>` wird nach optionalen Manipulationen per `<trimStyle>` zwischen Start- und End-Tag des XML Elementes mit dem Namen *`<OutputElementName2>`* in der Ausgabe abgelegt, wenn `found=true` gilt. Für `found=false` wird kein Fehler am Ende der Zeile ausgelöst.

  Danach wird der Eingabezeiger der entsprechenden Zeile auf den Wert von `end` gesetzt. Der globale Bereichsstack wird nicht verändert.

Für obiges Beispiel kann die folgende Spezifikation verwendet werden, wenn für jeden Teamnamen nur die Zahl extrahiert werden soll und eine feste Breite von `10` Zeichen verwendet wird:

```
area TabellenBereich: start(posSL(0)), end(posML(5, 0)):
asHorzTable (“Kosten”) {
     AddAttribute(“Kosten.TeamNr”   , regSL(“( )*Team \lb\num+\rb”));
     AddAttribute(“Kosten.Material” , posSL(10));
     AddAttribute(“Kosten.Gehalt”   , posSL(10));
     AddAttribute(“Kosten.Miete”    , posSL(10));
     AddAttribute(“Kosten.sonstiges”, posSL(10));
}
```

Als Ausgabe wird dann folgendes erzeugt, wenn das oben skizzierte Datenmodell in einer DTD beschrieben ist:

```
<Kosten TeamNr=“1”, Material=“10.478”, Gehalt=“150.000”, Miete=“5.500”,
sonstiges=“760”/>
<Kosten TeamNr=“3”, Material=“1.570”, Gehalt=“87.000”, Miete=“3.500”,
sonstiges=“2.710”/>
<Kosten TeamNr=“3”, Material=“37.679”, Gehalt=“95.000”, Miete=“1.450”,
sonstiges=“1.250”/>
<Kosten TeamNr=“4”, Material=“2.500”, Gehalt=“457.000”, Miete=“12.450”,
sonstiges=“5.600”/>
```