JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
Maximilian Arthofer

Angefertigt am
**Institut für
Systemsoftware**

Beurteiler / Beurteilerin
**o.Univ.-Prof. Dr. Dr.h.c.
Hanspeter Mössenböck**

August 2023

A PARSER GENERATOR FOR LALR(1) GRAMMARS

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

im Bachelorstudium

Informatik (UK 033/521)

**JOHANNES KEPLER
UNIVERSITÄT LINZ**
Altenberger Straße 69
4040 Linz, Österreich
jku.at
DVR 0093696

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Bachelorarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Ort, Datum

Unterschrift

Inhaltsverzeichnis

1. Einleitung/Motivation

Am Institut für Systemsoftware der JKU Linz findet jährlich die Lehrveranstaltung Compilerbau statt. Gegen Ende dieser Lehrveranstaltung wird in der Regel, die sogenannte Bottomup-Syntaxanalyse besprochen, welche über einige wünschenswerte Eigenschaften verfügt, für große Sprachen praktisch, aber sehr komplex ist. In den zugehörigen Übungseinheiten erstellen die Studierenden dann Tabellen, welche als Grundlage für diese Parser Variante dienen.

Ziel dieser Arbeit ist es, den bereits vorhandenen Compilergenerator Coco/R so zu adaptieren, dass er anstatt eines Top-Down-Parsers einen solchen Bottom-Up-Parser generiert und den Prozess dieser Generierung als Ausgabe bereitstellt. Diese soll dann von den Übungsleitern verwendet werden können, um Aufgaben zu erstellen oder gegebenenfalls zu korrigieren.

Die entstehende Software heißt Coco/LR in Anlehnung an die verwendeten LR-Grammatiken Die Implementierung von Coco/LR soll sich so nah wie möglich an jene von Coco/R halten. Dementsprechend wurden in dieser Arbeit große Teile von Coco/R übernommen, andere adaptiert oder ausgetauscht.

Im ersten Kapitel („Background“) wird auf die, für das Verständnis der Arbeit notwendige, Theorie eingegangen. Diese orientiert sich an den Vorlesungsfolien der Compilerbau LVA (Mössenböck, Unterlagen zur Vorlesung Compilerbau, 2022).

In den darauffolgenden Kapiteln wird die Implementierung und Verwendung von Coco/LR beschrieben, sowie auf etwaige Probleme bei der Entwicklung und deren Lösung eingegangen.

2. Background

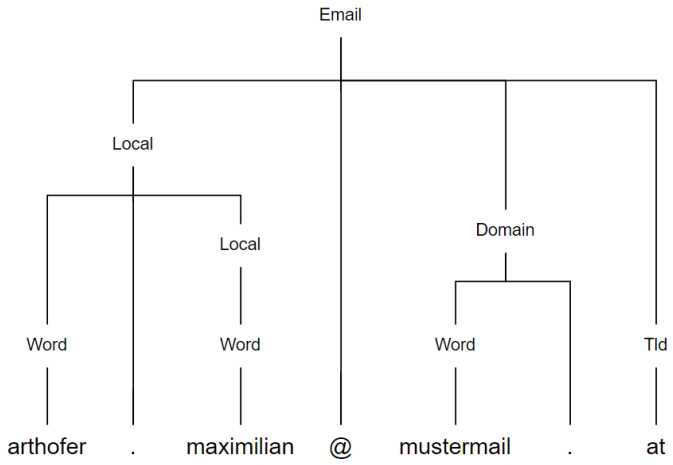
2.1. Compiler & Grammatiken

Ein Compiler ist ein Programm, welches Code, geschrieben in einer Hochsprache liest und in eine andere Sprache (meist Maschinensprache/Assembler) umwandelt. Als Hochsprachen bezeichnet man Programmiersprachen, welche die Komplexität der unterliegenden Systeme & Hardware abstrahieren, um das Entwickeln von Software zu vereinfachen.

Die kleinsten Einheiten, in welche wir eine Sprache unterteilen, nennen wir Terminalsymbole. Hierbei kann es sich um Zeichen und Wörter handeln. Durch das Zusammensetzen von Terminalsymbolen erhalten wir Nonterminal-Symbole. Ein Nonterminal-Symbol kann wiederum auch andere Nonterminal-Symbole enthalten. Die genaue Zusammensetzung letzterer wird durch sogenannte Produktionen beschrieben. Eine Produktion besteht aus zwei Seiten. Die linke Seite gibt das Nonterminal-Symbol an, welches beschrieben wird. Die rechte Seite gibt in einer Sequenz aus Terminal- und Nonterminal-Symbolen die Zusammensetzung an. Gibt es für ein Symbol mehrere Optionen der Zusammensetzung schreibt man diese durch „|“ getrennt an. Man nennt diese Form auch Backus-Naur-Form (im Folgenden kurz BNF).
Durch mehrere solcher Produktionen lässt sich nun, ausgehend von einem Startsymbol, systematisch die Grammatik einer Sprache beschreiben. Eine einfache Grammatik zur Erkennung einer E-Mail-Adresse könnte in BNF etwa wie folgt aussehen (Anm.: stark vereinfacht; vollständige Definition für Emails definiert in RFC 5322):

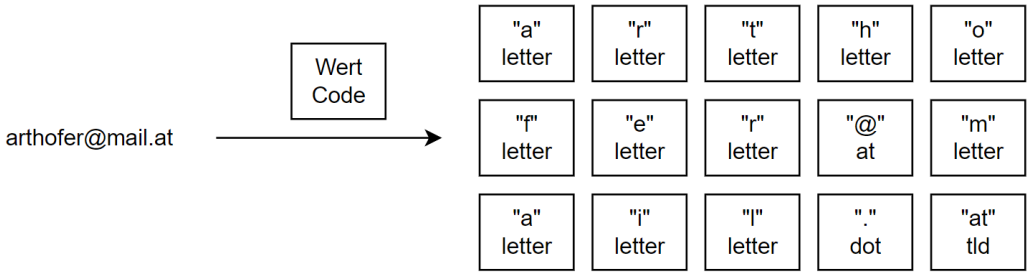
Word = letter | Word letter.
Local = Word | Word “.” Local.
Tld = “at” | “de” | “com”.
Domain = Word “.” | Word “.” Domain.
Email = Local “@” Domain Tld.

Die Grammatik beschreibt die Syntax einer Sprache, also wie man Symbole zu gültigen Sätzen kombiniert. Die Zusammensetzung eines Satzes lässt sich auch als Baum darstellen, was wir passenderweise Syntaxbaum nennen:



Ein Compiler nimmt also eine Eingabe mit einer bestimmten Grammatik, und wandelt diese um. Der erste Schritt einer Kompilierung ist immer die sogenannte Lexikalische Analyse. Die Komponente eines Compilers, welche diese Aufgabe übernimmt, wird Scanner genannt. Der Scanner liest die Eingabezeichen und wandelt diese in einen Tokenstrom um. Ein Token hat einen eindeutigen Code sowie einen Wert und stellt ein Terminalsymbol in einer Produktion dar.

Bleiben wir beim Beispiel eine E-Mail-Adresse könnte diese Umwandlung wie folgt aussehen:



Auf Basis dieser Token erfolgt nun der nächste Schritt, die Syntaxanalyse, welche prüft, ob ein Satz der Grammatik entspricht. Diese Aufgabe übernimmt der Parser. Dieser versucht nun mit den Tokens, unter Zuhilfenahme der Grammatik, einen gültigen Syntaxbaum aufzubauen, ein Beispiel eines solchen haben wir bereits gesehen.

Die Frage ist nun: Aus welcher Richtung bauen wir ihn auf? Die erste Variante ist „Top-Down“, also von oben nach unten. Im Programm erfolgt dies durch ein Verfahren namens „rekursiver Abstieg“. Dabei definieren wir für alle Nonterminal-Symbole Methoden und wenn wir an irgendeiner Stelle ein Symbol erkennen wollen, rufen wir einfach dessen Methode auf. Diese kann dann wiederum andere Methoden aufrufen, um die Zeichen zu erkennen aus welchem sich das zugehörige Symbol zusammensetzt usw. Ein solcher Parser muss immer beim obersten Symbol (in unserem Bsp. E-Mail) starten und arbeitet sich dann nach unten.

Eine alternative Möglichkeit ist, das Parsen von unten zu starten. Solch ein Parser wird BU-Parser genannt und verfügt über ein paar Vorteile im Gegensatz zum rekursiven Abstieg.

Die Analyse erfolgt durch einen Kellerautomaten, welcher die Funktionalität eines „herkömmlichen“ endlichen Automaten erweitert. Ein endlicher Automat verfügt über eine bestimmte Anzahl an verschiedenen Zuständen (States). Darüber hinaus gibt es eine Zustandsübergangsfunktion, die den Input liest und durch Kombination von aktuellem State und Input den nächsten State bestimmt. Der Automat startet in einem vordefinierten Startzustand und akzeptiert einen Input, wenn er an irgendeinem Punkt einen Endzustand erreicht. Ist nach Abarbeitung des gesamten Inputs kein solcher Zustand erreicht, wird die Eingabe abgelehnt.

Ein Kellerautomat verfügt darüber hinaus über eine Speicherstruktur, den Keller, der es ihm ermöglicht, einen Teil des gegangenen Weges (also die durchlaufenen States) „zurückzugehen“. Des Weiteren bietet er die folgenden Analysefunktionen:

- Shift
  - Liest ein Zeichen vom Input, und wechselt in einen neuen Zustand.
- Reduce
  - Reduziert eine Menge von Symbolen auf ein bestimmtes Nonterminal-Symbol. Des Weiteren wird der Zustand um die Länge der verwendeten Produktion minus eins Schritte zurückgesetzt.
- Shift-Reduce
  - Findet eine Shift-Operation in einen State statt, in welchem nur mit einer einzigen Produktion reduziert wird, so kann diese Shift-Operation durch eine Shift-Reduce Operation ersetzt und der Reduce-State gelöscht werden.
- Accept
  - Der Input wird akzeptiert.
- Error
  - Der Input wird abgelehnt.

In jedem State ist für jedes möglicherweise auftretende Symbol eine dieser Aktionen definiert. Wir wollen diesen Sachverhalt nun anhand eines Beispiels genauer betrachten.

Die folgenden Produktionen sind gegeben:

1. A = 'a' B C.
2. B = 'b'.
3. C = 'c'.

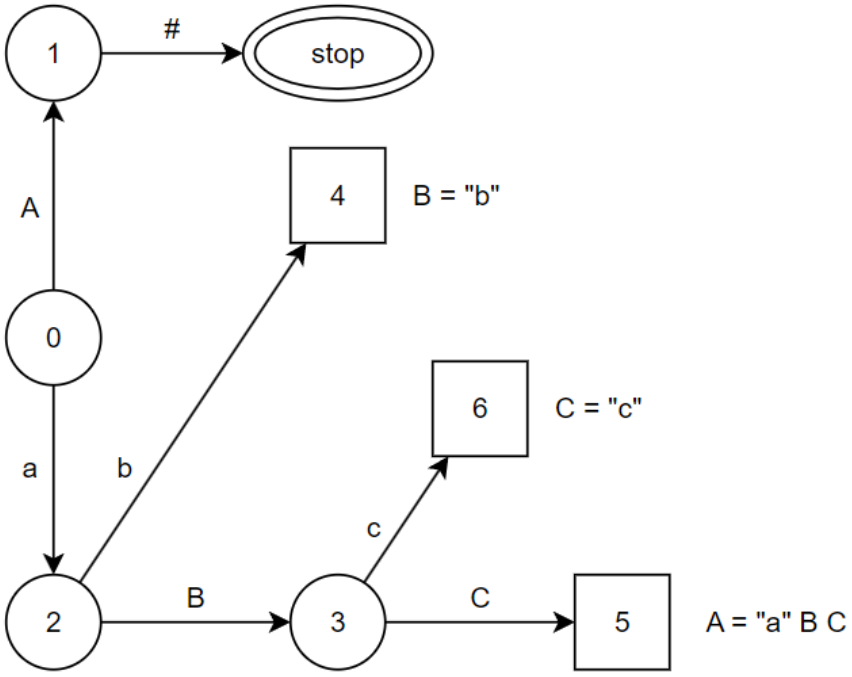
Die zugehörige Parser Tabelle:

	a	**b**	**c**	**#**	**A**	**B**	**C**
0	s2	-	-	-	s1	-	-
1	-	-	-	accept	-	-	-
2	-	s4	-	-	-	s3	-
3	-	-	s6	-	-	-	s5
4	-	-	r2	-	-	-	-
5	-	-	-	r1	-	-	-
6	-	-	-	r3	-	-	-

Im Folgenden nutzen wird die vorgestellten Operationen, gemeinsam mit einem Keller, um die Eingabe „abc“ zu erkennen. Die Raute am Ende der Eingabe signalisiert EOF (End of File) also das Ende der Eingabe:

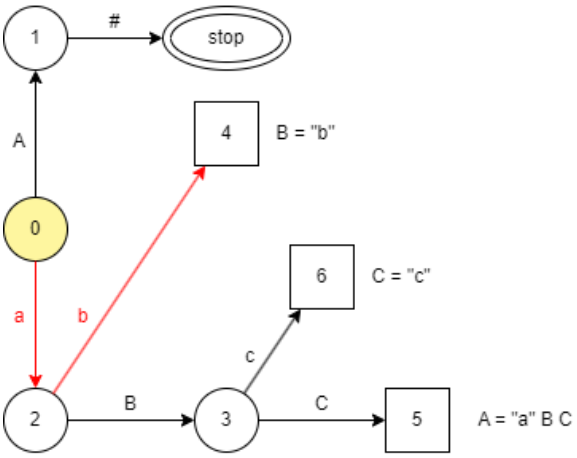
Keller	**Eingabe**	**Aktion**
0	a b c #	shift 2
0 2	b c #	shift 4
0 2 4	c #	reduce 2
0 2	B c #	shift 3
0 2 3	c #	shift 6
0 2 3 6	#	reduce 3
0 2 3	C #	shift 5
0 2 3 5	#	reduce 1
0	A #	shift 1
0 1	#	accept

Der zugehörige Automat sieht wie folgt aus:

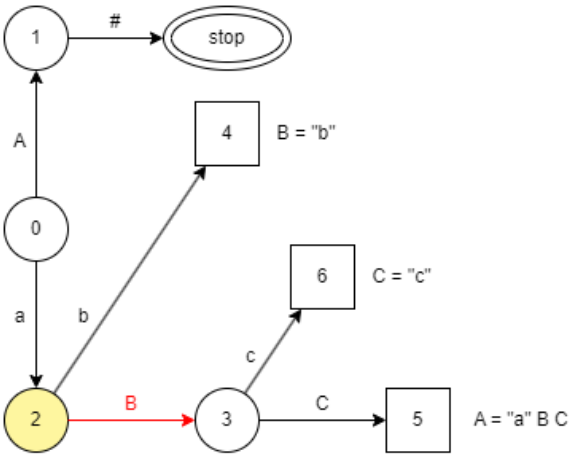


Hier sehen wir die Erkennung derselben Eingabe („abc“) am Automaten:

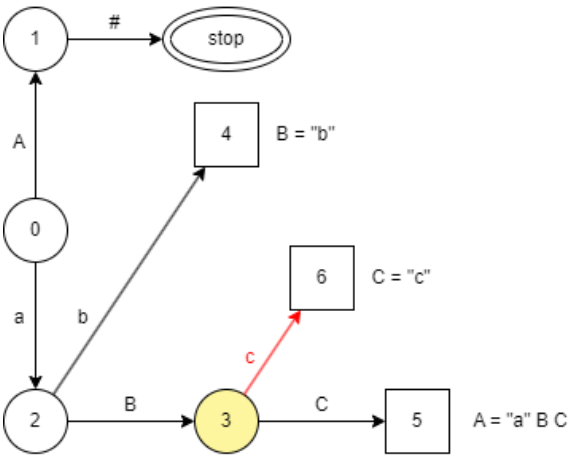
abc#



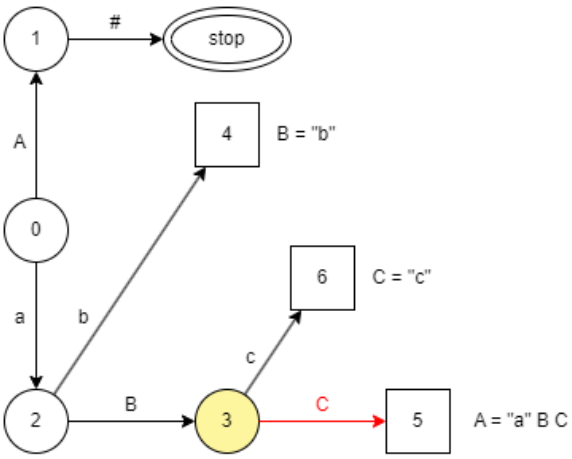
Bc#



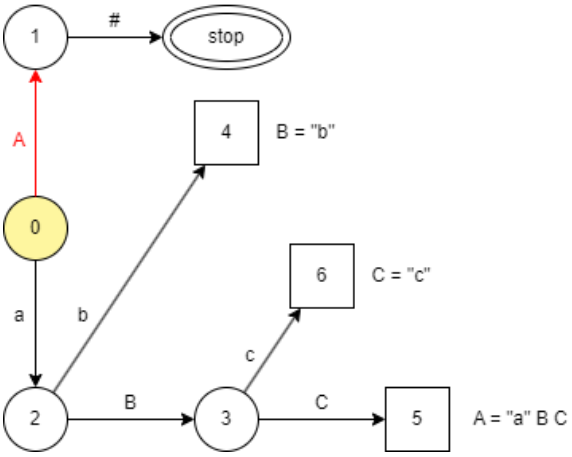
c#



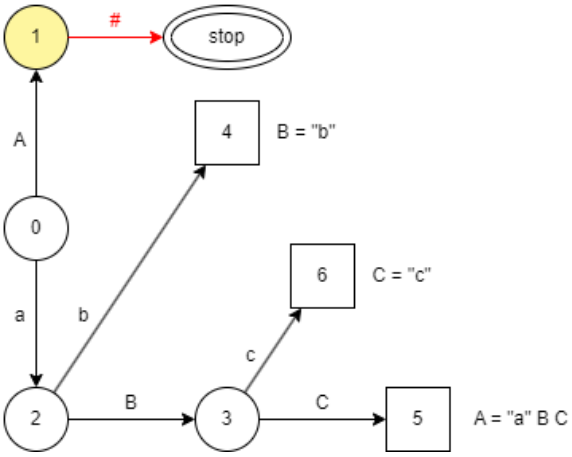
C#



A#



#



Die Grammatiken, welche wir mit unseren BU-Parsern erkennen möchten werden LALR(1) Grammatiken genannt. Wir nennen eine Grammatik LALR(1), wenn in jedem Zustand des Automaten für jedes Symbol eindeutig festgestellt werden kann, ob eine Shift- oder eine Reduce Operation ausgeführt werden soll. Darüber hinaus ist es uns mit LALR(1) Sprachen erlaubt, Zustände mit dem selben Kern aber unterschiedlichen Nachfolgern zu einem Zustand zusammenzufassen (siehe später).

Ein BU-Parser ist im Gegensatz zu einem Top-Down-Parser mächtiger, da er z.B. auch in der Lage ist, Alternativen mit den gleichen terminalen Anfängen zu verarbeiten, was auch notwendig ist, da man im Syntaxbaum von unten kommend vorher nie genau wissen kann, zu welchem Nonterminal-Symbol ein Token gehört. Da die Funktionalität als Tabelle dargestellt werden kann, ist der Parser an sich auch deutlich kleiner und universell einsetzbar (lediglich die Tabelle muss ausgetauscht werden). Darüber hinaus ist es auch möglich, Fehler selbstständig zu korrigieren (auch wenn man nicht unbedingt das gewünschte Ergebnis erhält). Auf der anderen Seite ist ein BU-Parser deutlich schwerer zu konstruieren bzw. zu debuggen, da die Abarbeitung der Syntaxanalyse aus Sicht des Parsers, im Gegensatz zum Rekursiven Abstieg, nicht intuitiv verständlich ist.

Für komplexe Sprachen ist die Erstellung dieser Compiler-Komponenten unter Umständen sehr aufwendig, weshalb sogenannte Compilergeneratoren entwickelt wurden, die Teile dieser Generierung übernehmen können. Ein Beispiel für einen solchen Compilergenerator ist Coco/R, auf welchem diese Arbeit basiert. Ausgehend von einer Beschreibung der Sprache kann Coco/R den Scanner sowie den Parser selbstständig generieren.

In einem vollständigen Compiler folgen auf die Syntaxanalyse noch die semantische Analyse (was bedeutet ein Satz?), sowie eine Optimierung mit anschließender Codeerzeugung. Aufgrund der Komplexität des Themas beschränken wir uns hier jedoch auf die Lexikalische Analyse (gegeben durch Coco/R), sowie die Syntaxanalyse basierend auf einem BU-Parser. Das bedeutet auch, dass die durch Coco/LR generierten Parser nur in der Lage sind festzustellen, ob eine Eingabe aus gültigen Sätzen einer Grammatik besteht.

2.2. Tabellengenerierung

Wir verstehen nun was einen BU-Parser grundsätzlich ausmacht und im folgenden wird beschrieben, wie man von einer Grammatik zu einer Parser Tabelle kommt. Vereinfacht gesagt, bewegt sicher der Parser Symbol für Symbol durch die Produktionen. Die Kombination aus Produktion und aktueller Position des Parsers in dieser nennen wir in diesem Kontext ein Item. Für die Grammatik A = B B a. B = x. könnte das erste Item daher wie folgt aussehen:

A = . B B a

Da es sich bei B um ein Nonterminal-Symbol handelt ist es in diesem Item effektiv so, als stünden wir direkt vor der B-Produktion:

A = . B B a
B = . x

Des Weiteren merkt sich jedes Item seine Nachfolger, also die Menge der Terminalsymbole, die auf die Produktion folgen können:

A = . B B a /…
B = . x / x

Für diese beiden Items gäbe es jetzt zwei mögliche Aktionen: zum einen ein shift mit „B“ und zum anderen einen shift mit „x“.

Der Zustand oder State eines Automaten ist durch den Abarbeitungs-Status seiner Produktionen also durch eine Menge von Items definiert. Führen wir mit den obigen Items einen Shift mit B durch Erhalten wir einen neuen Zustand:

A = B . B a /…

Alle Items eines Zustands, deren Position nicht am Beginn der Produktion ist, werden als Kern eines States bezeichnet. Stehen wir in einem der Items vor einem Nonterminal-Symbol, können wir den Kern, wie oben, um die entsprechenden Produktionen erweitern:

A = B . B a /…
B = . x / a

Diese erweiterte Form des Kerns wird Hülle genannt. Ziel ist es nun durch Shift- und Reduce-Aktionen alle möglichen Zustände des Automaten zu generieren. Wir beginnen immer damit, einen zusätzlichen Start State einzufügen. Beginnend an diesem Startpunkt im Zustand 0 wiederholen wir für jeden State die folgenden Schritte:

1. Hülle bilden
2. Mögliche Aktionen suchen
    a. Stehen wir am Anfang oder inmitten einer Produktion -> Shift
    b. Stehen wir am Ende einer Produktion -> Reduce
3. Neue States bilden
    a. Führt eine shift Operation zu einem Kern, der in irgendeinem State bereits vorgekommen ist, so führt dieser shift in diesen State. In diesem Fall müssen die Nachfolger in diesem State aktualisiert werden.
    b. Ansonsten wir ein neuer State gebildet
4. Start bei 1 für nächsten State

Bei einer Shift-Operation schreiben wir immer zuerst „shift“, dann das Symbol mit welchem geshiftet wird gefolgt vom Zielzustand.
Bei einer Reduce-Operation schreiben wir immer „reduce“, dann die Nachfolger des aktuellen Items, gefolgt von der Nummer der für die Reduktion verwendeten Produktion.

Zustand Nr.	**Items**	**Nachfolger**	**Aktion**
0	A‘ = . A #	/	shift A 1
	A = . B B a	/ #	shift B 2
	B = . x	/ x	shift x 3
1	A’ = A . #	/	accept
2	A = B . B a	/ #	shift B 4
	B = . x	/ a	shift x 3
3	B = x .	/ a,x	reduce a,x 2
4	A = B B . a	/ #	shift a 5
5	A = B B a .	/ #	reduce # 1

Aus den States lässt sich nun die folgende Parser Tabelle ablesen:

State/Symbol	**a**	**x**	**#**	**A**	**B**
0	-	s3	-	s1	s2
1	-	-	acc	-	-
2	-	s3	-	-	s4
3	r2	r2	-	-	-
4	s5	-	-	-	-
5	-	-	r1	-	-

Alle Symbole, für welche in einem State keine Aktionen definiert sind, sind error-Aktionen.
Diese Tabelle lässt sich noch etwas vereinfachen. Führt eine Shift-Operation in einen Zustand in welchem nur reduziert werden kann, kann diese Shift-Operation durch eine Shift-Reduce Operation ersetzt werden. Hierbei schreiben wir immer „shiftred“ gefolgt vom Symbol mit welchem geshiftet wurde und der verwendeten Produktion. Der Zustand in welchen das Shift geführt hat kann dann gelöscht werden. Für unser Beispiel führt das zu folgendem Ergebnis:

State/Symbol	**a**	**x**	**#**	**A**	**B**
0	-	sr2	-	s1	s2
1	-	-	acc	-	-
2	-	sr2	-	-	s4
4	sr1	-	-	-	-

2.3. Fehlerbehandlung

Kommt es während des Parsens zu einem Fehler, so wird dieser wie folgt behandelt:

1. Ausgehend von der aktuellen Position, werden sogenannte „Guides“ oder „Wegweiser“, also vordefinierte Symbole, verwendet, um so schnell wie möglich zum Endzustand zu gelangen. Auf dem Weg, den diese Guides definieren, werden alle gültigen Symbole gesammelt. Diese bilden unsere Anker-Menge.
2. Nun überlesen wir so lange Token in der Eingabe, bis wir auf ein Symbol stoßen, welches Teil der Anker-Menge ist.
3. Ist das aktuelle Symbol Teil der Ankermenge (spätestens EoF), werden so lange die Guides eingefügt, bis wir in den Zustand kommen, in welchem das aktuelle Symbol gültig, also eine Aktion dafür definiert ist.

Dieser Vorgang stellt nicht sicher, dass das Ergebnis erzielt wird, das vom Benutzer beabsichtigt war, aber es verhindert einen Absturz des Parsers und schlägt eine mögliche Korrektur vor.

Beachtet man bei der Erstellung der Tabelle ein paar Regeln, lassen sich die Wegweiser ganz einfach daraus ablesen. Erstens sind Produktionen des gleichen Nonterminal-Symbols nach Größe zu sortieren (dazu müssen Nonterminal-Symbole möglicherweise aufgelöst werden). Zweitens ist beim Bilden der Hülle darauf zu achten, dass wenn man in einem Item vor einem Nonterminal-Symbol steht, das Item mit der zugehörigen Produktion unmittelbar unterhalb platziert.

Produktionen:

A = x y z.		1. A = B x.
A = B x.	sortieren + nummerieren	2. A = x y z.
B = y z.		3. B = y.
B = y.		4. B = y z.

ZstdNr.	Items	Nachfolger	Aktion	Guide
0	A‘ = . A #	/	shift A 1	y
	A = . B x	/ #	shift B 2	
	B = . y	/ x	shift y 3	
	B = . y z	/ x		
	A = . x y z	/ #	shift x 4	
1	A’ = A . #	/	accept	#
2	A = B . x	/ #	shift x 5	x
3	B = y .	/ x	reduce x 3	x
	B = y . z	/ x	shift z 6	z
4	A = x . y z	/ #	shift y 7	y
5	A = B x .	/ #	reduce # 1	#
6	B = y z .	/ x	reduce x 4	x
7	A = x y . z	/ #	shift z 8	z
8	A = x y z .	/ #	reduce # 2	#

Wurde die Tabelle richtig aufgebaut, ist die erste erzeugte Terminal-Aktion der Wegweiser.
Wir wollen uns die Fehlerkorrektur anhand eines Beispiels nochmals genauer ansehen. Wir verwenden die obige Grammatik mit der fehlerhaften Eingabe: „xaz“

Keller	Eingabe	Aktion
0	x a z #	shift x 4
0 4	a z #	error

1.) Fluchtweg suchen indem wir den Wegweisern folgen, dabei bilden wir die Ankermenge:

Keller	Aktion	Ankermenge
0 4	shift y 7	{y}
0 4 7	shift z 8	{y, z}
0 4 7 8	reduce # 2	{y, z, #}
0 1	accept	{y, z, #}

2.) Fehlerhafte Token löschen, solange das aktuelle Symbol auf der Eingabe nicht in der Ankermenge ist.

a z # „a“ ist nicht Teil der Ankermenge und wird somit gelöscht.
z # „z“ ist Teil der Ankermenge und muss somit nicht gelöscht werden.

3.) Fehlende Token einfügen, bis ein Zustand erreicht wird, in welchem mit dem aktuellen Eingabesymbol weitergemacht werden kann.

„z“ ist in Zustand 4 nicht definiert. Wir folgen dem Wegweiser und setzen „y“ ein und führen „shift y 7“ aus.

„z“ ist in Zustand 7 definiert, der Wiederaufsatz ist geglückt!

Keller	Eingabe	Aktion
0 4	y z #	shift y 7
0 4 7	z #	shift z 8
0 4 7 8	#	reduce # 2
0	A #	shift A 1
0 1	#	accept

Im Grunde haben wir das „a“ in der Eingabe durch das korrekte Symbol „y“ ersetzt.

3. Die Eingabesprache

Die Eingabesprache von Coco/LR basiert auf der zugehörigen Grammatik, welche in der Datei „cocolr.atg“ (verfügbar im Abgabefolder) definiert ist. Diese wurde zu Beginn der Arbeit durch den Betreuer zu Verfügung gestellt. Sie orientiert sich stark der Grammatik von Coco/R und übernimmt einige Stellen auch direkt. Entsprechend basiert auch die folgende Beschreibung auf jener von Coco/R, zu finden im Coco/R Benutzerhandbuch (Mössenböck, Coco/R User Manual, 2010).

Coco/LR definiert 4 verschiedene Token, welche als Grundbausteine für die Eingabe verstanden werden können:

- ident = letter { letter | digit }.
- number = digit { digit }.
- string = '"' { stringCh | '\\' printable } '"'. //String-Konstante
- char = '\'' (charCh | '\\' printable { hex }) '\''. //Character-Konstante
-

Darüber hinaus reserviert Coco/LR die folgenden Schlüsselwörter: ANY, CHARACTERS, COMMENTS, COMPILER, CONTEXT, END, FROM, IGNORE, NESTED, PRODUCTIONS, TO

Einzeilige Kommentare in der Eingabedatei werden durch „//“ bzw. mehrzeilige durch „/*“ mit anschließendem „*/“ beschrieben.

Das Coco/R Benutzerhandbuch beschreibt die Grammatik der Eingabedatei als Produktionen, was hier für Coco/LR adaptiert wird. Die Eingabedateien für Coco/LR folgen dieser Struktur:

```
CocoLR =    "COMPILER" ident
            ScannerSpecification
            ParserSpecification
            "END" ident.
```

Der Identifier der auf das COMPILER Schlüsselwort folgt wird auch Grammar-Name genannt und muss mit dem Identifier nach END, am Ende der Eingabedatei übereinstimmen. Darüber hinaus muss immer eine Produktion mit dem Grammar-Name vorhanden sein. Alle Produktionen müssen sich in eine Sequenz von Terminal-Symbolen zerteilen lassen und für jedes verwendete Nonterminal-Symbol (also alles, was kein Token, Character- oder String-Literal ist) muss zumindest eine Produktion existieren.

Die Scanner-Spezifikation ist wie folgt aufgebaut:

```
ScannerSpecification =  "CHARACTERS"{ SetDecl }
                        "TOKENS"{ TokenDecl }
                        "COMMENTS"
                        "FROM" TokenExpr
                        "TO" TokenExpr
                        "IGNORE" IgnoreSet
```

Das Schlüsselwort “CHARACTERS” erlaubt die Definition von Zeichenmengen, die dann zur Definition von Tokens verwendet werden können. Dabei sind Operationen wie das Angeben von Bereichen (z.B. 'A'..'Z') oder die Aggregation mehrerer Mengen durch „+“ möglich. Des Weiteren können wir das Schlüsselwort „ANY“ verwenden, um alle möglichen Symbole zuzulassen bzw. mit „-“ Symbole oder Teilmengen wieder abziehen.

Auf „TOKENS“ folgt die Definition der Tokens, die der Scanner später erkennen und an den Parser liefern soll. Tokens werden als EBNF-Produktionen unter Verwendung der Character-Sets definiert.

Kommentare werden mit einem Start- und einem Endsymbol nach „COMMENTS“ definiert. Sollen Kommentare geschachtelt werden können, folgt das „NESTED“ Schlüsselwort auf die Definition.

Soll der Scanner bestimmte Sequenzen in der Eingabe komplett ignorieren, können diese folgend auf das „IGNORE“ Keyword angegeben werden.

Auf die Scanner-Spezifikation folgt die Parser-Spezifikation. Diese hat folgende Struktur:

```
Parser-Specification = "PRODUCTIONS"
                                { ident '=' { Symbol } '.' }
```

Die Produktionen werden in Coco/LR ausschließlich per BNF definiert, da die Generierung einer LALR(1) Tabelle mit dem in der Vorlesung vorgestellten Algorithmus, ausschließlich mit dieser Form möglich ist. Auf der linken Seite der Produktion steht der Name des zu definierenden Nonterminal-Symbols. Auf der rechten steht eine beliebige Sequenz von Terminal- und Nonterminal-Symbolen, abgeschlossen durch einen Punkt. Alternativen in den Produktionen werden im Falle von Coco/LR, ausschließlich durch die Definition mehrerer Produktionsregeln abgebildet.

Der Inhalt der Eingabedatei für eine Sprache „D“ könnte wie folgt aussehen:

```
COMPILER D
CHARACTERS
        letter    = 'A'..'Z' + 'a'..'z' + '_'.
        digit     = '0'..'9'
        cr        = '\r'.
        lf        = '\n'.
        tab       = '\t'.
TOKENS
        an        = letter | digit.
COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO lf
IGNORE cr + lf + tab
PRODUCTIONS
        D         = ...
END D.
```

4. Implementierung

4.1. Datenstrukturen

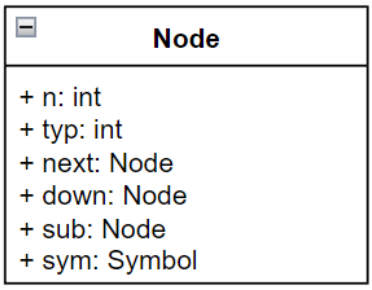
4.1.1. Symboltabelle

Die grundlegendste Datenstruktur, welche in Coco/LR Verwendung findet, ist die Klasse Symbol (Tab.java). Sie wurde aus Coco/R übernommen und repräsentiert Terminal- sowie Nonterminal-Symbole. Diese werden in zwei getrennten Listen in der Tab-Klasse gespeichert.
Für jedes Symbol (abhängig vom Typ) wird z.B. gespeichert ob es löschbar ist, welche terminalen Anfänge es hat, welche Symbole folgen können usw.

4.1.2. Syntax-Graph

Auch diese Datenstruktur wurde aus Coco/R übernommen, wenngleich Coco/LR nur auf einen Teil seiner Funktionalität zurückgreift.
Die einzelnen Produktionen der Parser-Spezifikation der Eingabedatei werden in Coco als Graph repräsentiert. Der Anfang des Graphen wird als Teil des entsprechenden Nonterminal-Symbols gespeichert.

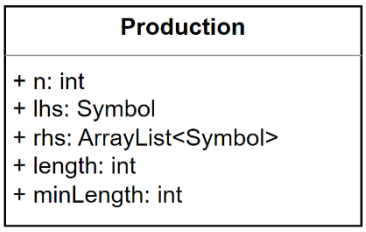
Der Grundbaustein des Syntax-Graphen ist die Klasse Node. Jeder Knoten im Graphen verfügt über eine Nummer n sowie einen Typ. Coco/R definiert 8 verschiedene Knoten-Typen. Für uns sind nur 3 davon relevant:



- Terminal-Symbol
- Nonterminal-Symbol
- Alternativen („alt“)

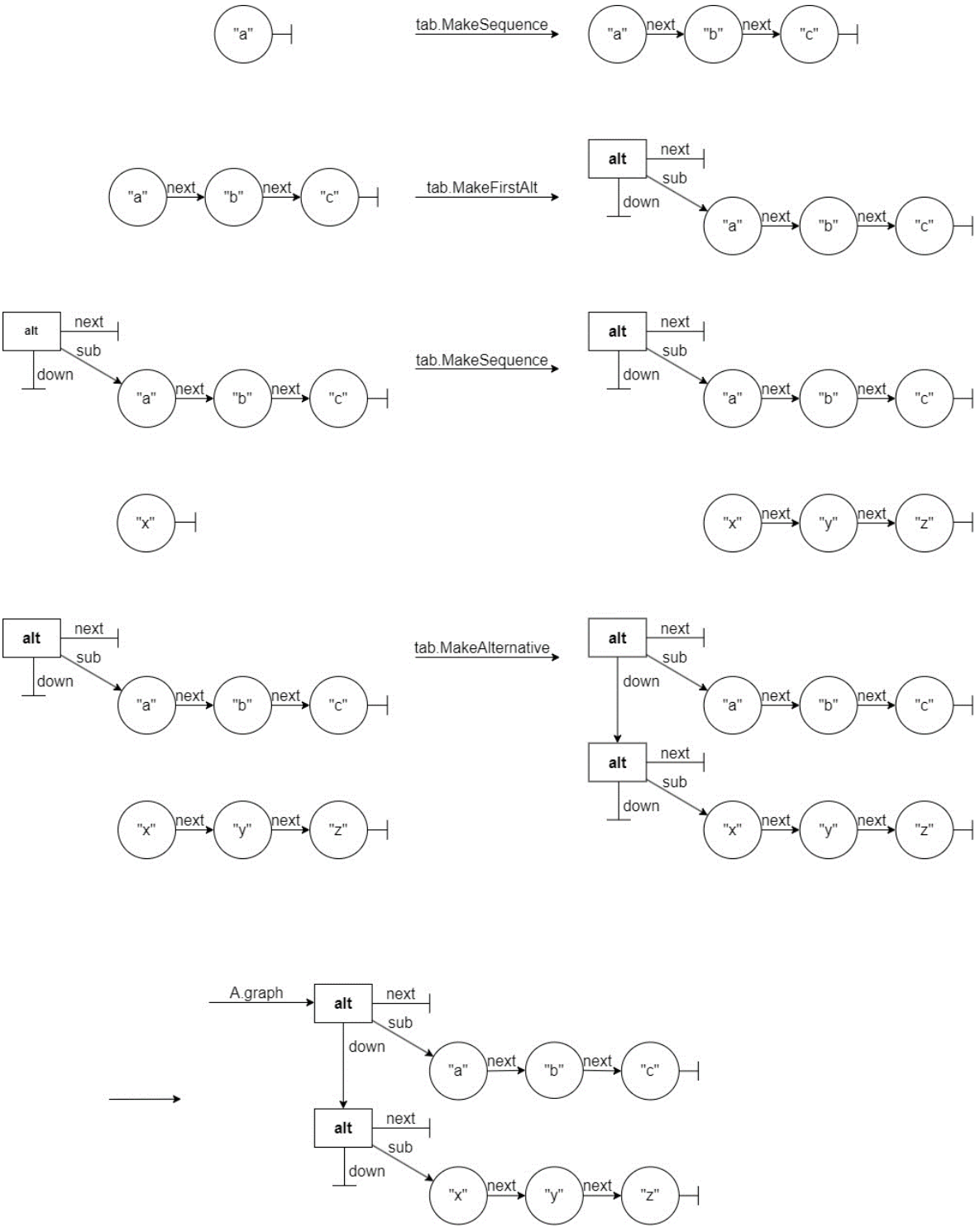
Das Feld „next“ zeigt auf den nächsten Knoten einer Sequenz. Die Felder „down“ und „sub“ sind nur für „alt“-Knoten relevant und zeigen jeweils auf den folgenden Knoten bzw. auf den ersten Knoten einer Alternative. Repräsentiert der Knoten ein Symbol, so ist dieser in „sym“ referenziert.

Aus dieser Struktur lesen wir zu Beginn der Parsergenerierung alle Produktionen in unsere eigene Datenstruktur, die Klasse „Production“ ein. In dieser ist es für uns etwas leichter mit den Produktionen zu arbeiten. Produktionen bestehen aus einem Nonterminal-Symbol auf der linken Seite der Produktion (left-hand-side = lhs) und einer beliebigen Sequenz von Symbolen auf der rechten Seite (right-hand-side = rhs).



Im Folgenden wird beispielhaft illustriert, wie der Parser aus den Produktionen unten, den entsprechenden Syntaxgraphen aufbaut:

A = „a“ „b“ „c“.
A = „x“ „y“ „z“.



Der erste Alternativ-Knoten wird in Coco/LR immer erstellt. Zeiger, die auf keinen anderen Knoten zeigen, sind auf „null“ gesetzt.

4.1.3. Items & Nachfolgermengen

Item
+ production: Production + pos: int + successors: ArrayList<Symbol> + dependents: ArrayList<Item>

Wie im Background bereits besprochen, sind Items, sich in Bearbeitung befindende Produktionen. Dementsprechend halten sie eine Referenz auf ein Produktions-Objekt sowie die aktuelle Position. Die „successor“-Liste speichert alle terminalen Nachfolger dieses Items.
Bei der Tabellengenerierung kommt es öfter vor, dass diese Nachfolger aktualisiert werden müssen. Dies geschieht in der Regel, wenn:

- eine shift-Operation in einen Zustand führt, der durch ein anderes Shift bereits erreicht wurde.
- beim Bilden der Hülle zwei Items vor demselben Nonterminal-Symbol stehen und unterschiedliche Nachfolger haben.
- Das Item durch das Expandieren eines Nonterminal-Symbols am Ende eines anderen Items entstanden ist, und somit die Nachfolger dieses Items teilt. Werden die Nachfolger des anderen Items verändert, so muss dies auch für dieses Item nachgezogen werden.

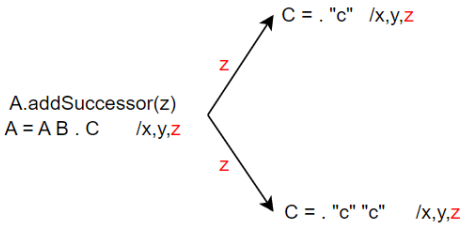
Der letzte Punkt, also die Abhängigkeiten zwischen den Nachfolgermengen unterschiedlicher Items ist ein durchaus komplexes Problem. Der erste Lösungsansatz war, dass sich solche Items einfach eine Nachfolgerliste teilen. Wird diese durch eines der Items verändert, bekommen alle anderen Items diese Veränderung ganz natürlich mit. Das Problem hierbei war jedoch, dass es zu Situationen kam, in welchen sich die Nachfolger des abhängigen Items verändert haben und diese Änderungen dann fälschlicherweise auch im übergeordneten Item sichtbar waren. Das Resultat war eine Parser Tabelle bei der mit demselben Symbol, im gleichen Zustand, sowohl eine Shift- als auch eine Reduce-Operation durchgeführt werden konnte. Davon abgesehen war die Darstellung im Trace-Output falsch. Glücklicherweise gibt es für derlei Abhängigkeiten eine sehr elegante Lösung: das Observer Pattern. Jedes Item führt die Liste „dependents“ in welchem es sich merkt, welche Items informiert werden müssen wenn sich die Nachfolger ändern. Auch hierzu ein kleines Beispiel:

Hier entstehen zwei Items durch Expandieren des Kerns. Da C am Ende der Produktion steht, sind die Nachfolger der entstehenden Items abhängig vom ersten Item, sie werden also in die „dependents“-Liste aufgenommen.

A = A B . C /x,y

↓

C = . "c" /x,y
C = . "c" "c" /x,y



Wird nun Item A ein Nachfolger hinzugefügt, wird dieser wie oben beschrieben allen abhängigen Items auch angefügt.

Auf diese Weise sind Aktualisierungen der Nachfolger nur „nach unten hin“ sichtbar, und die Parser-Tabelle sowie der Trace-Output sind korrekt.

Ebenfalls ist zu beachten, dass Nonterminal-Symbole löschbar sein können, was in der Regel bedeute, dass es eine leere Alternative gibt. Wir wollen uns auch diese Problematik anhand eines kleinen Beispiels genauer ansehen. Die folgenden Produktionen seien gegeben:

A = „a“ B C.
B = „b“.
C = „c“.
C = .

Das Symbol C ist hier löschbar. Man stelle sich nun den folgenden Kern vor:

A = „a“ . B C / #

Bilden wir die Hülle erhalten wir:

A = „a“ . B C / #
B = . „b“ / „c“, #

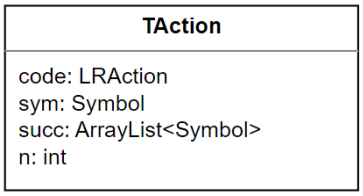
Da C löschbar ist, müssen wir die auf C in Produktion A folgenden Symbole auch als Nachfolger des neuen Items eintragen.

4.1.4. Produktions- & Zustandsliste

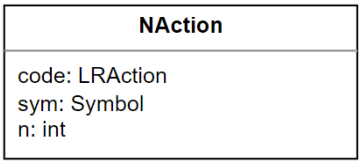
Der Parser Generator (ParserGen.java) bzw. die „WriteParser“ Methode hält jeweils eine sequenzielle Liste für alle Produktionen sowie die generierten Zustände. Diese sind trivial und werden nicht weiter behandelt.

4.1.5. Aktionsklassen & Aktionstabelle

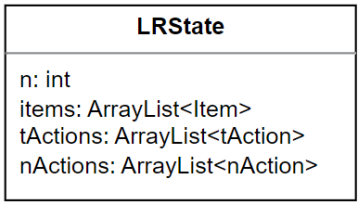
Zur Darstellung von Aktionen verwendet Coco/LR zwei getrennte Klassen, „NAction“ und „TAction“, sowie eine Enumeration für den Aktionscode.



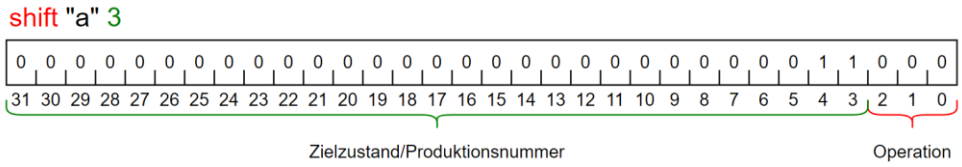
Terminal-Aktionen können sowohl shift- als auch Reduce-Operationen sein und benötigen folglich eine Nachfolger Liste für Reduce-Operationen. Nonterminal-Aktionen können nur shift-Aktionen sein und benötigen diese somit nicht.



Die Aktionen sowie alle zugehörigen Items werden in Objekten der Klasse „LRState“ (State ist in DFA.java, einer übernommenen Komponente, bereits definiert) gespeichert.



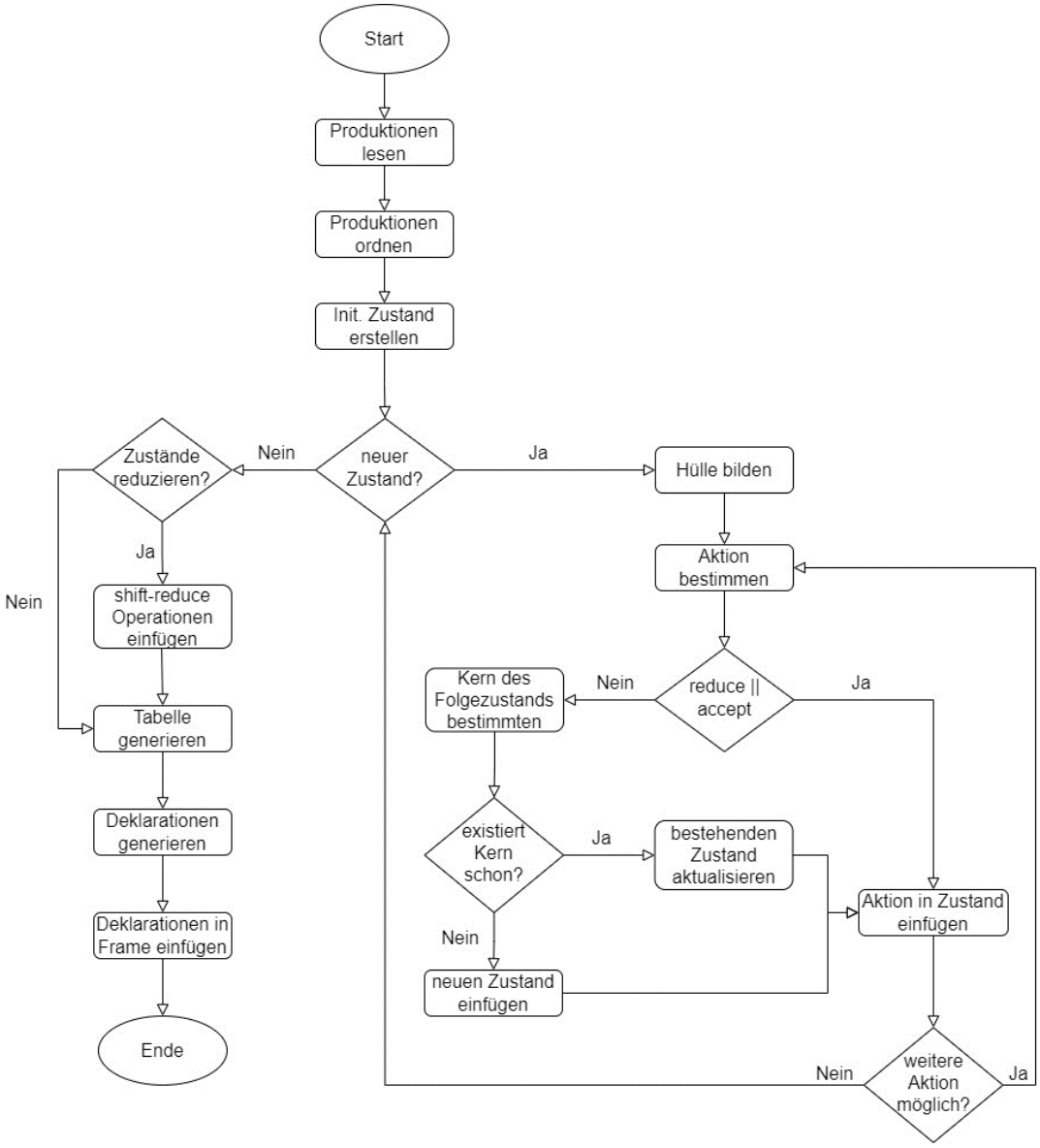
Wurden alle Zustände generiert/bearbeitet, werden die Aktionen in die Parser Tabelle (hier Aktionstabelle) geschrieben. Hierbei handelt es sich um ein zweidimensionales Integer-Array, wobei die Symbole (Terminal & Nonterminal) die Spalten und die Zustände die Zeile darstellen. Ist in irgendeinem Zustand X z.B. eine shift-Aktion für „a“ definiert, wird dies in der entsprechenden Zelle wie folgt codiert:



Die drei niederwertigsten Bits werden für die Codierung der Operation verwendet. Die restlichen 29 Bit codieren den Zielzustand bzw. die bei einer Reduktion verwendete Produktion.

4.2. Programmablauf Parsergenerierung

Das folgende Flussdiagramm zeigt den vereinfachten Ablauf der Parsergenerierung:



Der Parsergenerator beginnt damit, die Startproduktion (Grammarname‘ = Grammarname #.) in die Produktionsliste einzufügen. Danach iteriert er über alle Nonterminal-Symbole in der Symboltabelle und liest die Produktionen aus den jeweiligen Syntax-Graphen ein. Die Produktionen werden der Größe nach sortiert und durchnummeriert.
Sind die Produktionen fertig, nimmt er die Startproduktion und erstellt den ersten Zustand. Nun führt er, solange er noch nicht jeden Zustand bearbeitet hat, folgende Abläufe für jeden einzelnen Zustand durch:

1. Der aktuelle Zustand wird expandiert bzw. die Hülle gebildet.
2. Es werden alle möglichen Aktionen bestimmt und abgearbeitet:
    a. Handelt es sich um eine Reduce- oder Accept-Operation, wird die Aktion direkt in den aktuellen Zustand eingefügt.

   b. Handelt es sich um eine shift-Operation, muss der entstehende Kern berechnet werden. Ist dieser Ident zu einem der bereits existierenden Kerne, so führt die Shift-Operation in einen Zustand, der bereits existiert und die Nachfolger der Items in diesem müssen entsprechend aktualisiert werden.
3. Wurden so alle möglichen Aktionen durchgearbeitet, kommt der nächste Zustand dran usw.

Sind alle Zustände durchlaufen, werden diese (je nach gewählten Startparametern) durch Einfügen von Shift-Reduce Operationen reduziert:

1. Es werden alle Zustände und deren Aktionen (Terminal & Nonterminal) durchlaufen. Für jede shift-Operation wird festgestellt, ob der Zielzustand nur Reduce-Operationen enthält. Tut er das, kann die shift-Operation durch eine Shift-Reduce-Operation ersetzt und der betreffende Zustand gelöscht werden.
2. Wurden so alle Zustände durchgearbeitet, müsse sie neu nummeriert werden. Für jeden Zustand müssen daher alle shift Operationen in diesen Zustand gefunden und geändert werden.

An diesem Punkt ist die Zustands-Liste vollständig und die Parser Tabelle kann daraus generiert werden (das zweidimensionale Array, welches oben bereits genauer beschrieben wurde).
Der Output von Coco/LR erfolgt durch Einfügen der Deklaration dieser Tabelle in ein vorgefertigtes Gerüst: die Dateien Scanner.frame und Parser.frame. Damit das funktioniert, muss das Integer-Array zuvor in einen Deklarations-String umgewandelt werden, welcher dann in das entsprechende Frame eingefügt wird. Neben der Parser-Tabelle werden auf diese Weise noch einige andere Informationen mitgegeben, welche genau ist weiter unten ersichtlich, wenn wir auf die Struktur des generierten Parsers eingehen.

An diesem Punkt ist die Generierung des Parsers abgeschlossen. Die Verwendung der generierten Dateien ist weiter unten beschrieben.

4.3. Struktur des generierten Parsers

Der generierte Parser hat die folgende Struktur:

```
public class Parser {
        static int[][] action;
        static int[] prod_lhs;
        static int[] prod_lengths;
        static int[] guide;
        static int noTerminals;
        static String[] terminalNames;

        private Scanner scanner;
        private Token t;
        private Stack<Integer> stack;

        public Parser(Scanner scanner)
        public void parse();
        private void next();
        private int getOp(int a);
        private int getN(int a);
}
```

Zunächst verfügt der Parser über das 2-dimensionale „action“-Array, wie es bei den Datenstrukturen beschrieben wurde. Die Methoden „getOp“ sowie „getN“ können dazu verwendet werden die Informationen aus den in der Tabelle gespeicherten Integern zu extrahieren.

Damit wir den Parser als eine einzelne Datei ausliefern können, muss die Parser Tabelle in der Parser-Klasse initialisiert werden. Für Hochsprachen wie C ist die Parser Tabelle sehr groß und stößt an das Limit für die Größe einer Methode in Java, welches bei 65536 Bytes liegt. Daher müssen wir die Initialisierung auf mehrere, statische Initialisierungsmethoden aufteilen. Hier ist es wichtig zu wissen, wann eine Initialisierungsmethode beendet und die nächste gestartet werden muss. Wir müssen also in der Lage sein die Größe der Initialisierungsmethoden zu berechnen oder zumindest sinnvoll zu approximieren.

Der erste Eintrag in der Parser Tabelle für C sieht z.B. wie folgt aus:

```
action[0] = new int[] {4,48,4,4,56,4,4,4,… ,4,4,4,8,16,24};
```

Die Größe dieser Zeile berechnen wir mit:

```
GrZeile = GrErstellung + Spalten*GrEintrag
```

Multiplizieren wir die Größe einer Zeile mit der Anzahl der Zeilen in einer Initialisierungsmethode, erhalten wir logischerweise die Größe der Methode.

Durch Analysen des von Java generierten Byte-Codes (javap -c Parser.class) wissen wir, dass das Erstellen des Arrays im obigen Eintrag 10 Byte kostet:

```
0: getstatic    #1
3: iconst_0
4: sipush       145
7: newarray     int
9: …
…: aastore
```

Müssen wir für den Index auch einen Short verwenden, wie im folgenden Beispiel, kann die Größe der Erstellung bis zu 12 Byte betragen:

```
0: getstatic    #1
3: sipush       300
6: sipush       145
9: newarray     int
11: …
…: aastore
```

Auch die Größe der einzelnen Tabelleneinträge variiert. Für den ersten Eintrag im Beispiel oben (action[0][0] = 4), benötigen wir nur 4 Bytes:

```
0: dup
1: iconst_0
2: iconst_4
3: iastore
4: …
```

Müssen Index und Wert des Eintrags als Short geschrieben werden, benötigen wir hier bis zu 8 Bytes:

```
0: dup
1: sipush       356
4: sipush       1521
7: iastore
8: …
```

Wir gehen davon aus, dass alle Indizes sowie Tabelleneinträge in einem Short Platz haben. Dann wählen wir für beide Größen den jeweils größtmöglichen Wert und erhalten:

GrInit = Zeilen * 12 + Zeilen*Spalten*8

Mit dieser Formel können wir nun die Größe der Initialisierungsmethoden approximieren und so sicherstellen, dass der Parser keine zu großen Methoden enthält. Klar ist aber auch, dass die tatsächliche Größe der Methode im Regelfall deutlich unter dieser Approximation liegen wird, da viele Tabelleneinträge Error-Aktionen und somit sehr klein sind. Auf der anderen Seite lässt sich so sicherstellen, dass der generierte Parser immer funktioniert, ungeachtet der Sprache und ihrer Grammatik.

Im Array „prod_lhs“ (Productions Left-Hand-Side) sind die jeweiligen Nonterminal-Symbole für die Produktionen gespeichert. Diese werden für die Reduce- und Shift-Reduce-Operationen benötigt.
Die Variable „noTerminals“ speichert die Anzahl der Terminal-Symbole. Der Parser selbst sieht für die Symbole nur Nummern und kann so unmöglich wissen, wo die Nonterminal-Symbole beginnen. Wissen wir aber wie viele Terminal-Symbole existieren, können wir auch die Nonterminal-Symbole identifizieren.
Um sinnhafte Fehlerausgaben zu generieren ist es darüber hinaus notwendig die entsprechenden Symbol-Namen zu speichern. Ein Nutzer weiß mit der Nummer eines Symbols logischerweise nichts anzufangen!
Des Weiteren halten wir eine Referenz auf den Scanner, den aktuellen Token sowie unseren Keller (hier „stack“).

5. Bedienungsanleitung

5.1. Aufruf und Argumente

Der Abgabe dieser Bachelorarbeit wird ein JAR-Archiv beigelegt, welche alle Programmkomponenten von Coco/LR enthält (Mit Ausnahme der Frame-Dateien). Der Aufruf über die Kommandozeile erfolgt mittels:

java -jar CocoLR.jar Dateiname Optionen

Beim Aufruf des Programmes stehen die folgenden Optionen zur Verfügung (Beschreibung der aus Coco/R übernommenen Optionen basierend auf: (Mössenböck, Coco/R User Manual, 2010)):

- -package <Paketname>: Legt die beiden generierten Klassen in das angegebene Paket. Wird diese Option nicht angegeben, verbleiben sie im Default-Paket.
- -frames <Frame-Verzeichnis>: Gibt das Verzeichnis an, in welchen sich die Frame-Dateien befinden, welche für die Generierung verwendet werden. Standardmäßig werden die Frames in jenem Verzeichnis gesucht, in welchem sich das Grammatik-File befindet.
- -o <Ausgabe-Verzeichnis>: Erlaubt es ein Ausgabe-Verzeichnis anzugeben, in welches die generierten Dateien (Scanner.java & Parser.java) abgelegt werden. Standardmäßig wird das Verzeichnis verwendet, in welchem sich das Grammatik-File befindet.
- -trace <Trace-String>: Erlaubt es bestimmte interne Zustände von Coco/LR in ein Text-File (trace.txt) auszugeben. Die Definition, was genau auszugeben ist, wird mittels eines Strings definiert. Folgende Zeichen sind hierfür definiert:
    - A: die Zustände des Scanner-Automaten
    - F: die First- und Follow-Sets aller Nonterminal-Symbole
    - G: die Syntax-Graphen für alle Produktionen
    - I: die Berechnung der First-Sets
    - P: Performance Metriken bzgl. der Ausführung
    - S: die Symboltabelle sowie eine Liste aller deklarierten Literale
    - X: die gelesenen Produktionen (sortiert & nummeriert)
    - Y: alle generierten States, mit allen generierten Items und Aktionen
- -reduce: Mit dieser Option wird gesteuert, ob die generierte Tabelle mittels Shift-Reduce vereinfacht werden soll oder nicht. Wird sie nicht angegeben, wird kein Shift-Reduce durchgeführt.

5.2. Interface der generierten Klassen

Der generierte Scanner verfügt über das folgende Interface (Beschreibung des Scanner Interfaces basierend auf: (Mössenböck, Coco/R User Manual, 2010)):

```
public class Scanner {
        public Buffer buffer;

        public Scanner (String fileName);
        public Scanner(InputStream s);

        public Token Scan ();
        public Token Peek ();
        public void ResetPeek ();
}
```

Die Main-Klasse des Compilers, welcher diesen Scanner verwendet, muss, um diesen zu erstellen, den Namen der Input-Datei oder einen entsprechenden Input-Stream übergeben. Über „buffer“ kann auf die Eingabe direkt zugegriffen werden.

Die Methode „Scan()“ verrichtet die Hauptaufgabe des Scanners. Wird sie aufgerufen, liefert der Scanner den nächsten gelesenen Token.

„Peek()“ liefert ebenfalls den nächsten Token, entfernt diesen aber nicht von der Eingabe. Will man „Peek()“ auf die aktuelle Position zurücksetzen muss „ResetPeek()“ aufgerufen werden.

Der generierte Parser verfügt über das folgende Interface:

```
public class Parser {
   public Scanner scanner;
   public Token t;
   public Stack<Integer> stack;

   public Parser(Scanner scanner);

   public void parse();
}
```

Das Feld „scanner“ hält eine Referenz auf das verwendete Parser-Objekt, welches im Konstruktor übergeben werden muss. Im Feld „t“ wird eine Referenz auf das zuletzt gelesene Token gehalten und in „stack“ stellt den Keller des implementierten Automaten dar.

Die Methode „parse()“ startet den Parse-Vorgang.

5.3. Beispielsprache

Als Beispielsprache zur Demonstration der Funktionalität dieses Compilergenerators, verwenden wir ein abgewandeltes Beispiel für BU-Parser, aus der Compilerbau Übung. Hierbei geht es um die Erkennung von Linux-Pfaden. Gegeben sei die folgende EBNF-Grammatik:

Path = Dir {Dir} Name.
Dir = (Name | „.“ [„.“]) „/“.
Name = an {an}.

Um diese Grammatik in Coco/LR verwenden zu können, müssen wir sie zunächst in eine BNF-Grammatik umwandeln. Wir erhalten also:

Path = Dirs Name.
Dirs = Dir.
Dirs = Dirs Dir.
Dir = Name „/“.
Dir = „.“ „/“.
Dir = „.“ „.“ „/“.
Name = an.
Name = Name an.

Die Eingabedatei für diese Grammatik sieht dann wie folgt aus:

```
COMPILER Path
CHARACTERS
  letter    = 'A'..'Z' + 'a'..'z' + '_'.
  digit     = '0'..'9'.
  cr        = '\r'.
  lf        = '\n'.
  tab       = '\t'.
TOKENS
  an = letter | digit.
IGNORE cr + lf + tab
PRODUCTIONS
  Path = Dirs Name.
  Dirs = Dir.
  Dirs = Dirs Dir.
  Dir = Name "/".
  Dir = "." "/".
  Dir = "." "." "/".
  Name = an.
  Name = Name an.
END Path.
```

5.4. Beispiel für Ausgabe

Rufen wir Coco/LR für diese Grammatik mit der Y Option auf, erhalten wir eine Übersicht aller generierten Zustände, deren Items mit Nachfolger sowie mögliche Aktionen und Wegweiser:

States:

0	Path' = .Path EOF	[]	shift Path 1	an
	Path = .Dirs Name	[EOF]	shift Dirs 2	
	Dirs = .Dir	[an, "."]	shiftred Dir 2	
	Dir = .Name "/"	[an, "."]	shift Name 3	
	Name = .an	["/", an]	shiftred an 4	
	Name = .Name an	["/", an]		
	Dir = "." "/"	[an, "."]	shift "." 4	
	Dir = "." "." "/"	[an, "."]		
	Dirs = .Dirs Dir	[an, "."]		
1	Path' = Path.EOF	[]	accept	EOF
2	Path = Dirs.Name	[EOF]	shift Name 5	an
	Name = .an	[EOF, an, "/"]	shiftred an 4	
	Name = .Name an	[EOF, an, "/"]		
	Dirs = Dirs.Dir	[an, "."]	shiftred Dir 3	
	Dir = .Name "/"	[an, "."]		
	Dir = "." "/"	[an, "."]	shift "." 4	
	Dir = "." "." "/"	[an, "."]		
3	Dir = Name."/"	[an, "."]	shiftred "/" 6	"/"
	Name = Name.an	["/", an]	shiftred an 5	
4	Dir = "."."/"	[an, "."]	shiftred "/" 7	"/"
	Dir = "."."." "/"	[an, "."]	shift "." 6	
5	Path = Dirs Name.	[EOF]	reduce [EOF] 1	EOF
	Name = Name.an	[EOF, an, "/"]	shiftred an 5	
	Dir = Name."/"	[an, "."]	shiftred "/" 6	
6	Dir = "." "."/"	[an, "."]	shiftred "/" 8	"/"

Mit der X Option erhalten wir darüber hinaus eine sortierte Übersicht aller Produktionen gemeinsam mit der automatisch eingefügten „Anfangsproduktion“:

```
0 Path' = Path EOF
1 Path  = Dirs Name
2 Dirs  = Dir
3 Dirs  = Dirs Dir
4 Name  = an
5 Name  = Name an
6 Dir   = Name "/"
7 Dir   = "." "/"
8 Dir   = "." "." "/"
```

Die von Coco/LR generierten Klasse in Scanner.java und Parser.java müssen dann nur noch durch eine geeignete Main-Methode aufgerufen bzw. deren Objekte erstellt werden. Auch hier ein Beispiel für eine sehr einfache Variante hierfür:

```
public class Main {
    public static void main (String[] arg) {
        String srcName = arg[0];
        if (srcName != null) {
            try {
                Scanner scanner = new Scanner(srcName);
                Parser parser = new Parser(scanner);
                parser.parse();
            } catch (FatalError e) {
                System.out.println(e.getMessage());
            }
        } else {
            System.out.println("No Input File specified!");
        }
    }
}
```

Wählen wir eine gültige Eingabe wie etwa „Das/ist/ein/Pfad“ erhalten wie dir folgende Ausgabe:

Accept!

Für fehlerhafte Eingaben wie etwa „Das/ist/kein//Pfad/“ erhalten wir die folgenden, möglichen Korrekturen als Ausgabe:

l:1 c:14: an injected
l:1 c:20: an injected
2 errors occurred

5.5. Beispiel für komplexere Sprachen

Wie in der Abgabe gefordert, wurde Coco/LR auch für eine Hochsprache, in unserem Fall Java, angewandt. Die Grammatik, stammt aus der offiziellen Java Sprachspezifikation aus 1996 (Sun Microsystems, Inc., 1996), wo eine LALR(1) Version veröffentlicht wurde. Für die Verwendung in Coco/LR mussten die Produktionen in reine BNF gebracht werden. Die generierten Klassen sowie eine Reihe von Testfällen sind im Abgabearchiv enthalten.

6. Literaturverzeichnis

Mössenböck, H. (2005). Data Structures in Coco/R. Johannes Kepler Universität Linz.

Mössenböck, H. (2010). Coco/R User Manual. Johannes Kepler Universität Linz, SSW.

Mössenböck, H. (2022). Unterlagen zur Vorlesung Compilerbau. Johannes Kepler Universität Linz, SSW.

Sun Microsystems, Inc. (1. August 1996). *The Java Language Specification*. Von https://www.cs.cornell.edu/andru/javaspec/19.doc.html abgerufen