Elmar Stellnberger

Ein Simulator für endliche Automaten, Kellerautomaten und Turingmaschinen

Bakkalaureatsarbeit

zur Erlangung des akademischen Grades
Bachelor of Science

Studium
Informatik

Alpen-Adria-Universität Klagenfurt
Fakultät für Technische Wissenschaften

Begutachter
Assoc. Prof. Dr. Peter Schartner

Institut für Angewandte Informatik
Forschungsgruppe Systemsicherheit

Klagenfurt, 21.10.2018

Bei Fragen, Problemen oder Anregungen kontaktieren Sie bitte die Forschungsgruppe Systemsicherheit (info@syssec.at) oder den Autor (estellnb@elstel.org).

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich

- die eingereichte wissenschaftliche Arbeit selbstständig verfasst und andere als die angegebenen Hilfsmittel nicht benutzt habe,
- die während des Arbeitsvorganges von dritter Seite erfahrene Unterstützung, einschließlich signifikanter Betreuungshinweise, vollständig offengelegt habe,
- die Inhalte, die ich aus Werken Dritter oder eigenen Werken wortwörtlich oder sinngemäß übernommen habe, in geeigneter Form gekennzeichnet und den Ursprung der Information durch möglichst exakte Quellenangaben (z.B. in Fußnoten) ersichtlich gemacht habe,
- die Arbeit bisher weder im Inland noch im Ausland einer Prüfungsbehörde vorgelegt habe und
- zur Plagiatskontrolle eine digitale Version der Arbeit eingereicht habe, die mit der gedruckten Version übereinstimmt.

Ich bin mir bewusst, dass eine tatsachenwidrige Erklärung rechtliche Folgen haben wird.

Elmar Stellnberger 21.10.2018, Bodensdorf am Ossiacher See

Danksagung

Vielen Dank an Herrn Prof. Schartner, der mich in dieser Arbeit unterstützt und mir wertvolles Feedback gegeben hat.

Zusammenfassung

Diese Arbeit beschäftigt sich mit dem Themenbereich der theoretischen Informatik und hier speziell mit der programmtechnischen Simulation von nicht-deterministischen Automaten und Turingmaschinen. Bevor ein Automat oder eine Maschine implementiert werden, werden diese zumeist in eine deterministische Variante übersetzt, was die Implementierung effizienter macht und vereinfacht. Bei Kellerautomaten beschränkt man sich dabei im Parserbau zumeist auf Varianten bei welchen eine Übersetzung in deterministische Automaten möglich ist. Bei Turingmaschinen kann man zwar nicht-deterministische mit deterministischen simulieren, dies ist aber für die vorgestellte Arbeitsumgebung aus praktischen Gründen kein gangbarer Weg. Es ist deshalb möglich auch die Ausführung nicht-deterministische Automaten bereits in der Entwurfsphase zu simulieren. Dies ist für den Entwurf neuer Software und zu Lehrzwecken interessant, da diese Automaten somit ausführbar werden und bereits getestet werden können.

Vorwort

Diese Arbeit beschäftigt sich mit einer Anwendung zur Simulation von endlichen Automaten, Kellerautomaten und Turingmaschinen. Alle Automatentypen können sowohl deterministisch als auch nicht-deterministisch simuliert werden. Weiters werden die verschiedenen Typen von Automaten formal definiert und beschrieben.

Die Arbeit beginnt mit einem Kapitel zur Chomsky-Hierarchie, welche die Äquivalenz verschiedener Automatentypen mit entsprechenden Grammatiktypen aus der Menge der Phrasen-Struktur Grammatiken beschreibt. Weiters wird der Sinn unterschiedlicher Automatentypen motiviert, welcher auf die Komplexitätsklasse des Laufzeitverhalten sowie die Analysierbarkeit und die Anwendbarkeit von Algorithmen auf Automaten zurückzuführen ist.

Besonderes Augenmerk haben wir dabei auf die in der Literatur teilweise unterschiedlich ausfallenden Definitionen ein und derselben Automatentype gelegt. Es werden nicht nur unterschiedliche Definitionen beschrieben sondern die Arbeit zeigt auch wie die unterschiedlich definierten Automaten ineinander übergeführt werden können.

Die vorliegende Fassung der Arbeit kann jedoch keinen Anspruch darauf erheben, eine umfassende Einführung in die theoretische Informatik wiederzugeben. Im Gegenteil haben wir versucht den theoretischen Teil möglichst kurz zu halten und uns im wesentlichen auf die Definition unterschiedlicher Automatentypen zu beschränken.

Weitere interessante Themen für die wir auf die reichhaltige Literaturliste verweisen dürfen sind beispielsweise die Äquivalenz von endlichen Automaten und regulären Ausdrücken, der Nachweis der Äquivalenz von Kellerautomaten und kontextfreien Grammatiken sowie die unterschiedlichen Varianten des Bumping Lemma um nachweisen zu können ob eine Sprache eben nicht durch einen endlichen Automaten oder Kellerautomaten beschrieben werden kann.

Exemplarisch haben wir im theoretischen Teil den Algorithmus zur Überführung von deterministischen, endlichen Automaten (DFA – Deterministic Finite Automata) in nicht-deterministische, endliche Automaten (NFA – Non-Deterministic Finite Automata) beschrieben, da dies insbesondere für das Verständnis des praktischen Teils wichtig ist, wenn nicht-deterministische Automaten simuliert werden sollen.

Ebenso haben wir in die Literaturliste alle Bücher aufgenommen, die dafür notwendig waren, dass wir die Anwendung zur Simulation von Automaten und Maschinen tatsächlich haben implementieren können. Die Beschreibung der Implementierung geht dabei auf einen Streifzug durch den Simulator wobei das Gewicht auf den Kernkomponenten des Simulators gesetzt wurde. Die Implementierung der grafischen Benutzeroberfläche sowie der Bibliotheken auf unterster Ebene wird nur gestreift.

Die Implementierung wird dabei nicht nur auf einer übergeordneten, abstrakten Ebene beschrieben, sondern es werden dort, wo dies sinnvoll ist, auch exemplarische Quellcodeausschnitte präsentiert. Das soll den Leser schrittweise mit der Arbeitsweise des Simulators vertraut machen und dann auch an die Quellcodes heranzuführen. Das Lesen dieser Arbeit ist dabei durchaus als Einstieg in die Sourcen gedacht, wenn jemand das Programm weiter programmieren oder um neue Automatentypen erweitern will.

Wer an der grundlegenden Funktionsweise des Programms interessiert ist, wird hingegen auch nicht zu kurz kommen, denn wir beschreiben im Detail wie es möglich ist nichtdeterministische Automaten auf einem deterministischen Rechner zu simulieren. Dabei wird eine gemeinsame Simulationsroutine für verschiedenste Automatentypen verwendet.

Inhaltsverzeichnis

1 Theoretischer Teil

Eine Sprache ist in der theoretischen Informatik definiert als eine Menge von Wörtern. Jede syntaktisch korrekte Zeichenkette ist Element dieser Menge. Im Gegensatz zur natürlich sprachlichen Bedeutung des Wortes Sprache spielt dabei der Sinn oder genauer die Bedeutung des Textes keine Rolle. Weiters werden wir statt dem Wort "Text" einfach nur das Wort "Wort" für eine zu testende Eingabe verwenden.

Wir werden im folgenden einige Typen von Automaten kennenlernen, die so konstruiert werden können, dass diese jeweils über einer Sprache als Teil einer bestimmten Klasse von Sprachen akzeptieren können. Akzeptiert ein Automat oder eine Maschine über einem Eingabewort, so bedeutet dies, dass das Wort zu der vom Automaten erkannten Sprache gehört.

Es ist dabei sinnvoll die Möglichkeiten einer Sprache durch Zugehörigkeit zu einer bestimmten Sprachklasse zu beschränken, da dies einerseits die Konstruktion von Maschinen, welche diese Sprache akzeptieren, vereinfacht und uns andererseits an Analysefähigkeit gewinnen läßt.

Auch wenn Algorithmen zur automatischen Umformung oder Analyse von Automaten und Sprachen nicht Hauptteil dieser Arbeit sind, so werden wir gegebenenfalls auf die Existenz solcher Algorithmen hinweisen oder deren grundlegende Arbeitsweise kurz umreißen.

1.1 Unbeschränkte Grammatiken

Der einfachste Weg Sprachen zu beschreiben ist mittels einer Phrasenstruktur grammatik (*en: Phrase Structure Grammar*). Der Begriff rührt von der Verwendung von Phrasenstrukturgrammatiken zur Beschreibung natürlicher Sprachen her, wird aber genauso für künstliche Sprachen verwendet.

Das prominenteste Beispiel künstlicher Sprachen sind wohl Programmiersprachen, obwohl nach unserer Definition alles was eine Syntax hat als Sprache gilt, inklusive dem Format von Emailadressen oder der Makrosprache LaTeX zur Textverarbeitung, welche zur Erstellung dieser Arbeit verwendet worden ist.

Phrasenstrukturgrammatiken sind sehr einfach aufgebaut. Sie bestehen aus einer Menge von Regeln, mit linker und rechter Seite und einem Startsymbol. Beginnend beim Startsymbol wird in einer Top-Down Ableitung im aktuelle String die linke Seite einer Regel durch eine rechte ersetzt, solange bis nur mehr Terminale überbleiben.

Terminale sind Symbole aus dem Zeichensatz Σ der Sprache, die wir hier durch Kleinbuchstaben darstellen, während Nichtterminale Variablen oder syntakatische Kategorien darstellen, welche zwar zur Herleitung einer korrekten Syntax verwendet werden, sonst aber nicht in der Sprache vorkommen. Nichtterminale werden wir hier der Einfachheit halber mit Großbuchstaben schreiben.

$$
\begin{array}{llll}
S & \rightarrow & aSBC \mid aBC & \\
CB & \rightarrow & BC & \\
aB & \rightarrow & ab, & bB \rightarrow bb \\
bC & \rightarrow & bc, & cC \rightarrow cc
\end{array}
$$

$Ableitungsbeispiel: S \rightarrow aSBC \rightarrow aaBCBC \rightarrow aaBBCC \rightarrow aabBCC \rightarrow aabbCC \rightarrow aabbcC \rightarrow aabbcc$

Abbildung 1.1: Eine einfache Phrasenstrukturgrammatik

Jede linke Seite enthält dabei zumindest ein Nichtterminal. Die Schreibung $S \rightarrow aSBC|aBC$ meint eigentlich zwei Regeln, nämlich $S \rightarrow aSBC$ und $S \rightarrow aBC$. Abbildung 1.1 gibt folglich die Grammatik für $a^n b^n c^n$ mit $n \in \mathbb{N}$ an.

Bei näherer Analyse zeigt sich, dass Grammatiken, welche eine beliebige linke gegen eine beliebige rechte Seite tauschen können turingvollständig sind. Turingvollständig bedeutet, daß diese alles berechnen können, was mit Turingmaschinen berechenbar ist.

Turingmaschinen sind Maschinen, die einen aktuellen Zustand besitzen und abhängig vom aktuellen Zeichen unter dem Schreib-/Lesekopf und ihrem gegenwärtigen Zustand sich um ein Feld nach links, rechts bewegen oder das Zeichen unter dem Schreib-/Lesekopf austauschen können.

Man kann durch Äquivalenz mit anderen Berechenbarkeitsmodellen zeigen, dass Turingmaschinen alles berechnen können, was überhaupt berechenbar ist (siehe auch Kapitel 5.7 [Soch08]).

Wenn wir uns eine unbeschränkte Grammatik (Wir werden noch Teilklassen von Grammatiken mit Einschränkungen auf den erlaubten Regeln kennen lernen.) wie in Abbildung 1.1 anschauen, so müssen wir feststellen, dass man damit auch eine Turingmaschine implementieren könnte. Man schreibe dafür einfach den aktuellen Schreib-/Lesekopf im Zustand Z als Nichtterminal Z, nehme das aktuelle Zeichen unter dem Schreib-/Lesekopf als Kontext hinzu und beschreibe wie dieses verändert wird.

Es hat sich gezeigt, dass es praktisch unmöglich ist turingvollständige Berechnungssysteme programmatisch zu analysieren. Das Halteproblem, das zeigt, dass es unmöglich ist einen Algorithmus für ein solches Berechnungssystem zu schreiben, der feststellt, ob ein Programm überhaupt terminiert oder nicht, dient hier als exemplarisches Beispiel.

Um umgekehrt zu zeigen, dass Turingmaschinen jede beliebige unbeschränkte Grammatik akzeptieren können, reicht es uns vorerst festzustellen, dass eine Ableitung auch in umgekehrte Richtung erzeugt werden kann (Bottom-Up Parsing). Akzeptieren bedeutet dabei als Wort der Sprache erkennen. Im fraglichen Eingabestring werden dabei Terminale durch Nichtterminale auf der linken Regelseite ersetzt, solange bis das Startsymbol entweder ableitbar ist oder nicht.

Nun stellt sich dem interessierten Leser sicherlich die Frage, was ein Automat oder eine Maschine in einer Situation tun soll, in der mehrere Regeln anwendbar sind. Die einfachste Antwort darauf ist, dass die Maschine für beide Fälle weiter rechnet und akzeptiert, wenn ein Folgezustand akzeptiert. Das wäre zumindest eine praktische Realisierung von nicht-deterministischen Automaten.

Konzeptuell gesehen braucht ein nicht-deterministischer Automat aber von Anfang an nur zielführende Wege zu beschreiten; ein richtiger Weg wird dabei wie von Zauberhand gewählt. Das würde das Testen aller Möglichkeiten ersparen. Tatsächlich gilt für nicht-deterministischen Automaten, der kürzeste Weg vom Start zum akzeptierenden Ziel als Ausführungszeit.

Das Gebiet der künstlichen Intelligenz zeigt uns zwei Möglichkeiten auf, wie eine solche Berechnung von einer physikalischen Maschine geleistet werden kann: Breitensuche und Tiefensuche. In der Breitensuche wächst der Speicherinhalt mit jeder Fallunterscheiung, während sich die Tiefensuche für später die aktuell getroffene Entscheidung in einem Index merkt. Findet die Tiefensuche keine Übereinstimmung, so geht diese mit der Ausführung wieder an den Punkt, den sie sich vorher gemerkt hat zurück (Backtracking). Im schlimmsten Fall rechnet die Tiefensuche aber sehr lange, während sie ein kürzeres Ergebnis übersieht. Auch die Breitensuche kann im schlimmsten Fall für ein kurzes Ergebnis sehr lange brauchen, wenn der Entscheidungsbaum sehr breit wird und viele Alternativen in Betracht gezogen werden müssen.

Aus Kosten- und Effizienzgründen versucht man daher immer deterministische Automaten zu konstruieren, welche für einen Ausgangszustand höchstens einen Folgezustand kennen. Das Parsing von Programmiersprachen funktioniert dabei ausschließlich mit deterministischen Automaten (LL- und LR-Grammatiken sind dabei gängig verwendete Teilklassen von Grammatiken, für welche ein deterministischer Parser effizient konstruiert werden kann. [AhSU99]).

1.2 Die Sprachklassen der Chomsky-Hierarchie

Eine einfache Einschränkung für unbeschränkte Grammatiken (Typ 0) besteht darin zu fordern, dass die rechte Seite einer Regel stets länger als deren linke Seite sein muss. Diese Bedingung ist dafür hinreichend um sicherzustellen, dass ein für die Erkennung dieser Sprachklasse designter Automat auch terminiert. Da der Text durch neue Ersetzungen immer länger wird braucht nur so lange gerechnet werden, bis die aktuelle Länge des Eingabestrings erreicht wird.

Es ist wenigstens aus theoretischer Sicht sogar hinreichend zu fordern, dass die Länge des Textes nicht abnimmt, denn bei gleichbleibender Länge kann mit einem endlichen Zeichenvorrat auf endlicher Bandlänge nur eine endliche Zahl an Wörtern generiert werden. Die Maschine müsste dafür allerdings prüfen, ob das aktuell berechnete Wort schon einmal vorgekommen ist und falls dem so ist auf ein Weiterrechnen zu verzichten.

Eine Grammatik, deren Länge durch neue Ersetzungen stets gleich bleibt oder zunimmt nennt man monoton wachsende Grammatik (*en: monotonically increasing grammar, [Nagp13]*) oder Typ 1 Grammatik. Unabgesehen vom hohen Speicheraufwand für das Testen von Zyklen nennt man eine Turingmaschine, die eine monoton wachsende Grammatik akzeptiert, linear beschränkten Automaten (*en: linear bound automaton*).

Monoton wachsende Grammatiken nennt man auch kontextsensitiv. Die zusätzlichen Zeichen zum Nichtterminal auf der linken Seite nennt man dabei Kontext. Das läßt uns gleich auf kontextfreie Grammatiken (Typ 2) zurückkommen, welche auf der linken Seite nur ein einziges Nichtterminal ohne sog. Kontext haben dürfen. Die Sprache $a^n b^n$ ist kontextfrei ($S \rightarrow aSb|ab$), nicht aber die Sprache $a^n b^n c^n$, die wir im letzten Abschnitt kennen gelernt haben.

Deutsche Relativsätze haben beispielsweise eine kontextfreie Struktur (siehe Abbildung 1.2). Kontextfreie Grammatiken können von Stapel- oder Kellerautomaten geparst werden (Keller ist

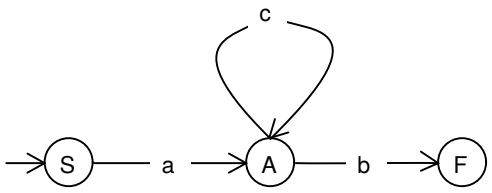
Der Mann arbeitet.
, welcher das Kind sieht,
, welches die Katze fütter,

$Satz \rightarrow Subjekt\ Verbphrase.$
$Relativsatz \rightarrow welcher\ Verbphrase \mid welches\ Verbphrase$
$Verbphrase \rightarrow Verb \mid Objekt\ Verb \mid Objekt,\ Relativsatz,\ Verb$
$Subjekt \rightarrow der\ mann,\ Objekt \rightarrow das\ kind \mid die\ katze$
$Verb \rightarrow f\ddot{u}ttert \mid sieht \mid arbeitet$

Abbildung 1.2: Deutsche Relativsätze

ein anderes Wort für Stapel; parsen bedeutet so viel wie von einem Kellerautomaten analysieren lassen). Der Kontext für die übergeordnete Ebene kommt dabei auf einen Stapel, das ist eine Datenstruktur die nach dem LIFO – Prinzip (Last In – First Out, zuletzt gemerkt – als erstes wieder gelesen) funktioniert. Der Name Keller kommt von Stapeln von Objekten, von welchen man immer nur das oberste nehmen kann, wie man diese häufig im Keller vorfindet.

Schließlich gibt es noch lineare Grammatiken, welche nur ein Nichtterminal auf der rechten Seite erlauben ([MoAK88]). Dabei sind aber vor allem linkslineare und rechtslineare Grammatiken interessant (Typ 3), welche das optionale Nichtterminal neben einem Terminal ganz auf der linken bzw. ganz auf der rechten Seite stehen haben.



$S \rightarrow aA,\ A \rightarrow cA \mid bF,\ F \rightarrow \varepsilon$
$S \rightarrow \varepsilon,\ A \rightarrow Sa \mid Ac,\ F \rightarrow Ab$

Abbildung 1.3: Endliche Automaten und Typ 3 Grammatiken

Über die Konstruktion eines endlichen Automaten lassen sich dabei linkslineare in rechtslineare Grammatiken übersetzen und umgekehrt (Abbildung 1.3). "ε" steht dabei für das leere Wort oder den String der Länge null.

ε-Produktionen sind Produktionen welche die linke Seite durch das leere Wort mit Null Buchstaben ersetzen. ε-Übergänge bei Automaten entsprechen ε-Produktionen bei Grammatiken. Der Automat wird dann so ausgeführt, als ob er sich in allen durch ε-Übergänge erreichbaren Knoten gleichzeitig befinden würde. Das ist dadurch bedingt, daß ε-Übergänge ohne das Lesen eines Zeichens vom Eingabewort aktiviert werden.

ε-Produktionen können entfernt werden, indem jede rechte Seite mit einem Nichtterminal das Epsilon ableitet verdoppelt wird: einmal mit dem Nichtterminal das Epsilon ableitet und einmal ohne diesen (Beispiel: $S \rightarrow sAb;\ A \rightarrow a \mid \varepsilon.$ wird zu $S \rightarrow sAb \mid sb;\ A \rightarrow a$. Ausnahme

bleiben Nichtterminale deren einzige Produktion eine ε-Produktion ist; diese kann man getrost ganz weglassen.

Mit der Elimination von ε-Produktionen ist die Bedingung, dass die rechte Seite wächst oder gleich bleibt auch für Typ 2 und Typ 3 Grammatiken mit ε-Produktionen erreicht.

Sprach-klasse	Name	akzeptierender Automat	erzeugende Grammatik
Typ0	rekursiv aufzählbar	Turing-Maschine	unbeschränkte Grammatik
Typ1	kontextsensitiv	linear beschränkte Turingmaschine	monoton wachsende Grammatik
Typ2	kontextfrei	nicht-deterministischer Kellerautomat	kontextfreie Grammatik
Typ3	regulär	endlicher Automat	linkslineare bzw. rechtslineare Grammatiken

Abbildung 1.4: Die Sprachklassen der Chomsky-Hierarchie

Die Tabelle aus Abbildung 1.4 faßt die bisher besprochenen Sprachklassen von Phrasenstrukturgrammatiken übersichtlich zusammen. Man könnte zwischen Typ 2 und Typ 3 Grammatiken sogar noch eine Ebene für deterministische Kellerautomaten einfügen, denn diese können eine echte Teilmenge der Sprachen nicht-deterministischer Kellerautomaten parsen. Voraussetzung aber keine hinreichende Bedingung für deterministische Kellerautomaten ist eine eindeutige (also eine unzweideutige) Grammatik.

Palindrome, das sind Wörter die in beide Richtungen gelesen das gleiche Wort ergeben (wie bspw. Marktkram), können nur von nicht-deterministischen Kellerautomaten erkannt werden, da der Automat nicht weiß wo die Mitte des Wortes liegt und daher in jedem Schritt beide Möglichkeiten testet: das Anhängen (*en: Push*) und das Herunternehmen (*en: Pop*) vom Stapel.

Interessant wird es, wenn wir auf die für die Spracherkennung benötigte Laufzeit, schauen. Kontextsensitive Grammatiken haben die sehr schlechte Komplexitätsklasse $O(2^n)$, sind also NP-vollständig, was bedeutet das ein Algorithmus nur alle Regelanwendungen der Reihe nach durchprobieren kann. Solche Algorithmen sind nur für kleine n effektiv lösbar, für größere brauchen diese so lange, dass ein Ende nicht absehbar ist.

Besser ist da schon die Laufzeit zum Parsen mehrdeutiger Grammatiken. Der CYK-Algorithmus (Cocke-Younger-Kasami-Algorithmus) vollbringt dies in einer Zeit von $O(n^3)$ ([Soch08], [MoAK88]). Dafür muss die Grammatik allerdings zuerst in Chomsky-Normalform gebracht werden (alle Regeln der Form $A \to BC$ oder $A \to b$).

Zum Parsen von Programmiersprachen werden allerdings in der Praxis LL(k) und LR(k) Grammatiken verwendet, welche eine echte Teilmenge aller deterministisch parsebaren Grammatiken darstellen [AhSU99]. Dies ist in annähernd linearer Zeit möglich, da für jedes Zeichen eine konstante Verarbeitungszeit einberechnet werden kann.

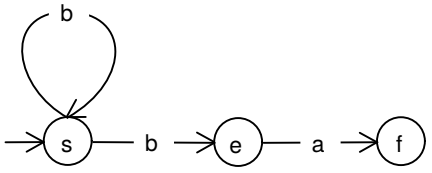
Für Programmiersprachen kommt es aber nicht nur darauf an, dass eine Sprache erkannt wird, sondern auch auf deren Semantik (Generierung von Objektcode). Die Grammatik bildet dabei die Struktur der Sprache ab. Es ist erwünscht, dass diese vor Verwendung im Parser nur begrenzten Umformungen unterzogen werden muss.

Schließlich bleibt uns noch die Klasse regulärer Sprachen, welche sehr effizient in linearer Zeit gescannt werden können. Im Gegensatz zu kontextfreien Grammatiken existiert für jede reguläre Grammatik ein deterministischer, endlicher Automat, der diese akzeptiert.

Abschließend wollen wir anmerken, dass es neben den Phrasenstrukturgrammatiken noch andere Arten von Grammatiken gibt, welche u.a. in der Computerlinguistik von Bedeutung sind. Es gibt nämlich in gewissen Sprachen auch natürlichsprachliche Konstrukte, welche nicht kontextfrei sind (Kapitel 9.2 [Haus00], Beispiel zu schweizer Deutsch). Mit sog. Linksassoziativen Grammatiken (LAG) können Sprachen wie $a^n b^n c^n$ in linearer Zeit gescannt werden.

1.3 Endliche Automaten

Ein endlicher Automat sei definiert als $M = (S, \Sigma, \Delta, s, F)$ wobei S eine Menge von Zuständen, Σ ein Eingabealphabet, Δ eine Überführungsrelation über $S \times \Sigma^* \times S$, $s \in S$ ein Startzustand und $F \subseteq S$ eine Menge von Endzuständen ist. Ein Eingabewort w ist nun Teil der Sprache $L(M)$, wenn der Automat M w akzeptiert. In einem solchen Fall gibt es Übergänge $(q_i, \tau_i, p_i) \in \Delta$ vom Startzustand s bis zu einem Enduzstand $f \in F$ mit $p_i = q_{i+1}$, sodass alle der Reihe nach gelesenen Teilworte τ_i zusammengefügt dem Eingabewort w entsprechen. Ansonsten akzeptiert M das Wort w nicht und $w \notin L(M)$.



$M_{b^*ba} = (\{s, e, f\}, \{a, b\}, \Delta, s, \{f\})$
$\Delta = \{(s, b, s), (s, b, e), (e, a, f)\}$

Abbildung 1.5: Nicht-deterministischer, endlicher Automat

Wir erkennen zunächst, dass es in Abbildung 1.5 mehrere Wege vom Startzustand s bis zum einzigen Zielzustand f gibt; beispielsweise *s-e-f, s-s-e-f* und *s-s-s-e-f*. Jeder dieser Wege, entspricht einer möglichen Eingabe *(hier: ba, bba und bbba)*, die der Automat akzeptiert. Will man also ein Wort finden, das in der Sprache $L(b^*ba)$ ist, braucht man nur einen Weg vom Start- zum Zielzustand zu suchen.

$$\underset{\Sigma^*}{\forall} v: \quad (q, \tau v) \vdash_M (p, v) \quad \Leftrightarrow \quad (q, \tau, p) \in \Delta$$

$$(s, bba) \vdash_{M_{b^*ba}} (s, ba) \vdash_{M_{b^*ba}} (e, a) \vdash_{M_{b^*ba}} (f, \varepsilon)$$
$$(s, ba) \vdash_{M_{b^*ba}} (e, a) \vdash_{M_{b^*ba}} (f, \varepsilon)$$

Soll nun aber umgekehrt ein gewisser Eingabestring erkannt werden, ist es so, als ob der Automat schon vorher wissen würde, welche Übergänge mit gelesenen Zeichen er als nächstes zu beschreiten hat. Die Beschreibung als „nichtdeterministisch“ soll aber genau dem entgegentreten. Der Automat verhält sich so, als ob er einfach so lange alle möglichen Wege ausprobiert, bis er einen Weg zum Ziel findet. Die obige Relation $\vdash_{M_{b^*ba}}$ zeigt wie eine vom Startzustand ausgehende, mit einem gewissen Eingabestring konfigurierte Maschine zwei der vorher angeführten Wörter

erfolgreich abarbeiten kann. „ε“ steht dabei für das leere Wort mit null Buchstaben. Am Ende müssen alle Buchstaben gelesen sein.

Auch wenn ein „echt“ nichtdeterministisch konstruierter Automat irgendwann auf einer richtigen Eingabe akzeptieren würde, so kann es bei komplizierteren Automaten zumindest für den menschlichen Betrachter schwierig oder umständlich sein, alle möglichen Wege der Reihe nach durchzutesten. Ein weiteres Problem wäre hier, dass wir ebenso wissen wollen wenn ein Wort $w \notin L(M)$ ist, ohne dass wir endlos warten müssen bis unser Automat vielleicht doch noch akzeptiert. Würde der Automat seine Wege echt zufällig auswählen, könnte es nämlich zumindest theoretisch vorkommen, dass der Automat beliebig lange nicht akzeptiert, obwohl er über einer gültigen Eingabe steht, denn es ist ja nicht ausgeschlossen, dass dieser denselben Weg mehrfach testet, wenn er sich nicht an alle bisher untersuchten Wege erinnert.

$$(\{s\}, bba) \underset{M_{b^*ba}}{\vdash} (\{s, e\}, ba) \underset{M_{b^*ba}}{\vdash} (\{s, e\}, a) \underset{M_{b^*ba}}{\vdash} (\{f\}, \varepsilon)$$

Tatsache bleibt, dass der Automat akzeptieren soll, sobald einer von vielen möglichen Wegen durch den Automaten zu einem Endzustand führt. Es ist dabei eher so, als ob sich der Automat bei der Ausführung in mehreren Zuständen gleichzeitig befindet und erst an gegebener Stelle, sobald entsprechende Zeichen eingelesen werden sollen, entscheidet für welche Strings er die Erkennung fortsetzen will. Das soll obige Relation $\underset{M_{b^*ba}}{\vdash}$ verdeutlichen.

Wäre es denn nicht viel einfacher, wenn unser Automat sich stets bloß in einem Zustand befinden könnte oder genauer gesagt, wenn jeder Zustand für jedes noch zu lesende Eingabewort nur einen möglichen Folgezustand kennen würde?

Ein endlicher Automat heißt deterministisch, wenn für Δ gilt, dass $S \times \Sigma^* \to S$ eine Funktion ist, d.h. beim Lesen der gleichen Eingabe nur ein einziger Folgezustand „aktiv“ wird und wenn kein Wort $\tau \in \Sigma^*$ ein Präfix eines anderen ist, d.h. $(p, \tau, q) \in \delta \ \wedge \ (p, \rho, r) \in \delta \ \wedge \ q \neq r \ \Rightarrow \ \neg\exists \nu : \tau = \rho\nu$. Dann könnte der Automat nämlich nach Lesen derselben Eingabe ebenso, über wie viele Schritte auch immer, in mehreren Zuständen landen. Es sei also in einem DFA, auch geschrieben als $(S, \Sigma, \delta, s, F)$, immer nur ein Zustand gleichzeitig aktiv. Damit gilt $L(DFA) \subseteq L(NFA)$ o.b.d.A. .

Die meisten Definitionen von DFAs sind sogar noch restriktiver, alsdass sie einfach $\delta : S \times \Sigma \to S$ voraussetzen, dass also jeweils genau ein Zeichen gelesen wird, was man durch geeignetes Einführen von Zwischenzuständen erreichen kann (siehe auch Kapitel 1.6). Epsilonübergänge (Übergänge, die kein neues Zeichen einlesen oder verbrauchen) sind in DFAs jedoch so oder so verboten, da das leere Wort ε mit null Buchstaben Präfix jedes anderen Wortes wäre. Das gilt zwar nicht automatisch wenn, wir beispielsweise nur einen einzigen Übergang von einem Zustand p aus haben, der ein Epsilonübergang ist. Dennoch sollte immer nur ein Zustand gleichzeitig aktiv sein können, was für Epsilonübergänge nicht gilt (Wir könnten dann auch beide Zustände, ohne dass sich an $L(M)$ etwas ändern würde, verschmelzen.).

Weiters setzen wir voraus, dass die Mengen S und Σ *(und damit auch Δ und F)* endlich sein müssen, dass also $|S| \in \mathbb{N}, |\Sigma| \in \mathbb{N}$. M soll ja ein Modell für eine wenn auch abstrakte Maschine sein und die ist eben in ihrer „Merkfähigkeit“ hier also in der Anzahl ihrer möglichen Zustände begrenzt. Noch wichtiger ist die Endlichkeit eines endlichen Automaten aber in Hinblick auf seine Analysierbarkeit und die Anwendbarkeit von Algorithmen.

Es gibt einen Algorithmus, der einen DFA in einen NFA überführt *(somit gilt dann: L(DFA)=L(NFA))*. Der einem NFA entsprechende DFA arbeitet dabei einfach über der Potenzmenge der Zustände des NFA. Jeder Zustand des DFA entspricht dabei einer Teilmenge der Zustände des NFA. Als Überführungsrelation dient $\Vdash$ statt $\vdash$.

Die Folgezustandsmenge einer Zustandsmenge ist dabei definiert als die Menge aller Zustände, die über Kanten, die ein gewisses Zeichen lesen, mit einem Zustand aus der ursprünlgichen Menge verbunden sind. Wird dabei ein neuer Zustand in die Zustandsmenge aufgenommen, so werden immer auch gleichzeitig alle mit diesem über ε verbundene Zustände mit aufgenommen (i.e. die Epsilonumgebung des Zustandes). Der Einfachheit halber können wir annehmen, dass alle Kanten nur ein einziges Zeichen einlesen, da mehrschrittige Übergänge über das Einfügen von Zwischenzuständen in mehrere einbuchstabige Kanten verwandelt werden können.

Der Algorithmus konstruiert dabei konkret die Zustandsmenge des Folgeautomaten ausgehend von der Epsilonumgebung des Startzustandes durch Nachgehen aller Kanten, was solange anhält bis kein neuer Zustand, als Menge von Teilzuständen aus dem NFA, mehr hinzugefügt werden kann.

Weiters gibt es noch einen Algorithmus, der die Zustände eines deterministischen endlichen Automaten minimiert. Dieser arbeitet so, dass er eine Äquivalenzrelation über den Knoten des Automaten definiert, bei der anfangs zwei Knoten als verschieden markiert werden, wenn man beim Nachgehen einer Kante mit bestimmtem Buchstaben einmal in einem Akzeptorzustand landet und einmal nicht [Soch08]. In den Folgeschritten werden weitere Knoten als verschieden erkannt, wenn zwei Wege mit gleicher Zeichensequenz von diesem Knoten aus einmal akzeptieren und einmal verwerfen. Das ist der einzige Unterschied auf den es dabei ankommt.

Solche Algorithmen gibt es nur dann, wenn sichergestellt ist, dass diese auch wirklich terminieren, was ja Voraussetzung für die Existenz eines Algorithmus ist. Folglich müssen diese auch über einer endlichen Zustandsmenge arbeiten. Ohne Endlichkeitsaxiom wären unsere eben definierten Automaten turingvollständig[1] und daher auch nicht programmatisch analysierbar.

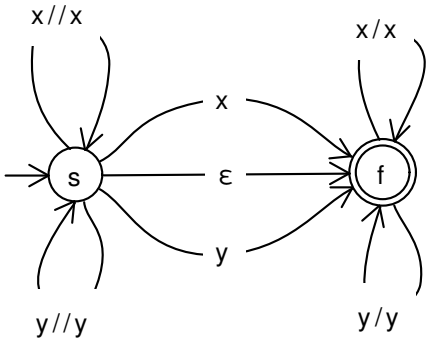
1.4 Kellerautomaten

Ein Stapel- oder Kellerautomat ist definiert als Sextupel von $M = (Z, \Sigma, \Gamma, \Delta, s, F)$, wobei Γ das Stapelalphabet ist und Δ als Teilmenge von $(Z \times \Sigma^* \times \Gamma^*) \times (Z \times \Gamma^*)$ definiert ist. Z entspricht S in endlichen Automaten und ist eine Menge von Zuständen. Alle anderen Komponentn haben dieselbe Bedeutung wie bei endliche Automaten.

Die beiden zusätzlichen Felder über Γ^* in Δ geben die in einem Ausführungsschritt zu poppenden und danach wieder zu pushenden Stapelsymbole wieder. Ein Stapel ist eine Datenstruktur, bei welcher Lese- und Schreibzugriffe immer an der Spitze stattfinden. Ein “Push” legt dabei Symbole oben auf den Stapel auf während ein “Pop” die zuletzt auf die Spitze des Stapels gelegten Symbole wieder herunternimmt.

Stapelautomaten akzeptieren, wenn die gesamte Eingabe gelesen ist, alle Symbole vom Stapel geholt worden sind und sich der Automat in einem Endzustand aus F befindet.

[1] d.h. könnten alles berechnen was berechenbar ist.



$M_{xyx} = (\{s, f\}, \{x, y\}, \Delta, s, \{f\})$
$\Delta = \{\ ((s, x, \varepsilon), (s, x)),\ ((s, y, \varepsilon), (s, y)),$
$((s, \varepsilon, \varepsilon),\ (f, \varepsilon)),\ ((s, x, \varepsilon),\ (f, \varepsilon)),\ ((s, y, \varepsilon),\ (f, \varepsilon)),$
$((f, x, x), (f, \varepsilon)),\ ((f, y, y), (f, \varepsilon))\ \}$

Abbildung 1.6: Palindromscanner, nichtdeterministischer Kellerautomat

Der in Abbildung 1.6 dargestellte Kellerautomat legt, solange er sich in Zustand s befindet alle gelesenen Eingabesymbole auf den Stapel. Die Annotation "x//x" des gezeigten Übergangs bedeutet dabei das Lesen eines "x" aus der Eingabe und das pushen eines "x" auf den Stapel. Der Automat kann dabei stets über einen ε-Schritt oder durch das Lesen eines einzelnen Zeichens in den Zustand f wechseln. Einmal in f angelangt werden alle ankommenden Eingabesymbole mit dem obersten Stapelzeichen verglichen und falls diese übereinstimmen, kann das Eingabesymbol gelesen und das oberste Stapelsymbol entfernt werden. Die Annotation "x/x" bedeutet dabei wiederum das Lesen eines "x" aus der Eingabe und das holen (poppen) eines "x" vom Stapel.

Da der Automat inhärent nichtdeterministisch arbeitet muss dieser nicht wissen wo die Mitte des von ihm erkannten Palindroms liegt. Es gibt viele Wege durch den Automaten. Erfolg hat aber nur jener der die Eingabe voll liest und den Stapel leer hinterläßt; das ist dann eben jener Weg der in der Mitte des Palindroms den Zustand auf f wechselt.

Beispiele für Palindrome in der deutschen Sprache sind "Kajak", "Regallager" und "Marktkram". Sie ergeben von vorne nach hinten wie von hinten nach vorne gelesen dasselbe Wort.

Im folgenden zeigen wir einen akzeptierenden und einen nicht akzeptierenden Weg durch den Automaten für das Wort "xyxyx":

$$(s, xyxyx, \varepsilon) \vdash_M (s, yxyx, x) \vdash_M (s, xyx, yx) \vdash_M (f, yx, yx) \vdash_M (f, x, x) \vdash_M (f, \varepsilon, \varepsilon)$$
$$(s, xyxyx, \varepsilon) \vdash_M (s, yxyx, x) \vdash_M (s, xyx, yx) \vdash_M (s, yx, xyx) \vdash_M (f, yx, xyx)$$

dabei gilt:

$$((p, u, \beta), (q, \gamma)) \in \Delta \quad \Leftrightarrow \quad \forall\, \alpha, x : (p, ux, \beta\alpha) \vdash (q, x, \gamma\alpha)$$

In der Praxis versucht man aber stets deterministische Kellerautomaten zu konstruieren. Auch obigen Palindromscanner würde man auf einem PC wohl anders implementieren, nämlich indem

man gleich das erste mit dem letzten und dann das zweite mit dem vorletzten Zeichen vergleicht und so fort.

Damit ein Kellerautomat deterministisch arbeitet darf er keine zwei kompatiblen Transitionen haben. Zwei Transitionen heißen kompatibel wenn ein zu lesendes Eingabewort Präfix eines anderen ist und gleichzeitig ein zu poppenden Stapelwort Präfix eines anderen.

Also beispielsweise $((p, ab, \alpha), (...))$ und $((p, a, \alpha\beta), (...))$: Kommt es hier dazu, dass als zu lesende Eingabe ab ansteht und $\alpha\beta$ am Stapel liegt, so werden beide Transitionen gleichzeitig aktiv. Das ist bei deterministischen Automaten zu vermeiden.

Bieten hingegen entweder das zu poppende Stapelwort oder das zu lesende Eingabewort einen Unterschied, so ist ausgeschlossen, dass der Automat in zwei verschiedenen Zuständen gleichzeitig landet.

Weiters bieten deterministische Stapelautomaten noch einen definitionsmäßigen Unterschied zu ihren nicht-deterministischen Verwandten: Das zuletzt gelesene Eingabezeichen ist stets ein \$, auch wenn dieses nicht tatsächlich Teil der Eingabe ist. \$ kann dabei nicht in der Mitte auftauchen, ist also nicht Teil des Eingabealphabetes.

$$
\begin{aligned}
M_{ab} &= (\{s, q, f\}, \{a, b\}, \Delta, s, \{f\}) \\
\Delta &= \{\ ((s, \varepsilon, \varepsilon), (q, \$)), \\
&\qquad ((q, a, \$), (q, a\$)),\ ((q, b, \$), (q, b\$)), \\
&\qquad ((q, a, a), (q, aa)),\ ((q, b, b), (q, bb)), \\
&\qquad ((q, a, b), (q, \varepsilon)),\ ((q, b, a), (q, \varepsilon)), \\
&\qquad ((q, \$, \$), (f, \varepsilon))\ \}
\end{aligned}
$$

Abbildung 1.7: Deterministischer Kellerautomat der testet ob $|w|_a = |w|_b$

Will ein deterministischer Automat am Ende und nur am Ende der Eingabe etwas vom Stapel nehmen, so kann er sich dabei auf das terminierende \$ berufen. Das ist etwa in einem Automaten praktisch, der abzählt, dass ein String aus gleich vielen “a”s wie “b”s besteht. Dieser braucht eine Bodenmarkierung, damit er weiß wann er beginnen soll zu zählen. Ist hingegen keine Bodenmarkierung greifbar können “a”s gegen “b”s aufgerechnet werden. d.h. ein “a” löscht ein am Stapel liegendes “b” und ein “b” ein am Stapel liegendes “a” bis der Stapel leer wird. Bei leerem Stapel mit Bodenmarkierung “\$” kann das aktuelle Eingabesymbol am Stapel gelegt oder das Ende der Eingabe erkannt werden.

Die Bodenmarkierung soll nun aber nicht entfernt werden ehe die ganze Eingabe gelesen ist, da sonst ein Weiterzählen nicht möglich ist. Das ist immer dann wichtig wenn zufällig in der Mitte der Eingabe die Anzahl der “a”s gleich jener der “b”s ist.

Es bleibt zu zeigen, dass ein nicht-deterministischer Stapelautomat mit einem die Eingabe terminierendem \$ gleichmächtig ist mit einem der dies nicht hat. Dann sind deterministische Stapelautomaten eine echte Teilklasse von nicht-deterministischen.

Wir können uns sehr einfach verdeutlichen, dass dies gilt indem wir folgendermaßen vorgehen:

Wir erfinden zu jedem Endzustand einen zweiten Endzustand. Wenn vom ursprünglichen Endzustand der Stapel ohne Lesen eines Eingabewortes ab- und aufgebaut werden kann, so geben wir auch dem neuen Endzustand diese Fähigkeit. Schließlich entfernen wir den ursprünglichen

Endzustand aus der Menge der Endzustände. Solange das abschließende $ nicht gelesen worden ist soll unser Automat auch nicht terminieren.

Haben wir nun einen Übergang, der die Endemarkierung $ aus der Eingabe liest, so lassen wir diesen in den neuen Endzustand wechseln. Das $ wird bei der Automatentransformation aus dem zu lesenden Eingabewort entfernt. Wir probieren also aus ob der Automat terminiert, wenn er sich schon am Ende der Eingabe befindet. Ansonsten bleibt er im neu geschaffenen Endzustand stecken ohne eine weitere Eingabe lesen zu können.

In der Literatur finden sich unterschiedliche Terminationsbedingungen für Kellerautomaten. Eine davon ist eine Terminierung ausschließlich nach Endzustand und eine weitere die Terminierung bei leerem Stapel in beliebigem Zustand.

Die Terminierung nach Endzustand kann mit unserem Modell dadurch simuliert werden, dass in jedem Endzustand einfach alle Zeichen vom Stapel geholt werden. Die Terminierung mit leerem Stapel in jedem Zustand kann man einfach erreichen indem man alle Zustände zu Endzuständen macht.

Umgekehrt kann man mit der Terminierung bei leerem Stapel auch wieder unser Modell simulieren. Man muss dabei einfach ein Zeichen am Anfang auf den Boden des Stapels legen und dieses zum Schluß nur in bestimmten dedizierten Endzuständen vom Stapel holen.

Auch die Terminierung nur durch Endzustand kann um das Abtesten eines leeren Stapels erweitert werden. Dafür ist wiede eine Bodenmarkierung des Stapels erforderlich. Man geht dabei nur bei Vorhandensein der Bodenmarkierung in den Endzustand über.

1.5 Turingmaschinen

Turingmaschinen lassen sich als Quadrupel von (Z, Σ, Δ, s) darstellen, wobei Δ eine Relation über $(Z \times \Sigma) \times (Z \times (\Sigma \cup \{L, R\}))$ ist. Wir werden uns hier vor allem mit deterministischen Turingmaschinen beschäftigen bei denen Δ eine Funktion ist und dann als $\delta : Z \times \Sigma \to Z \times (\Sigma \cup \{L, R\})$ geschrieben wird. Eine deterministische Turingmaschine ist somit durch (Z, Σ, δ, s) definiert.

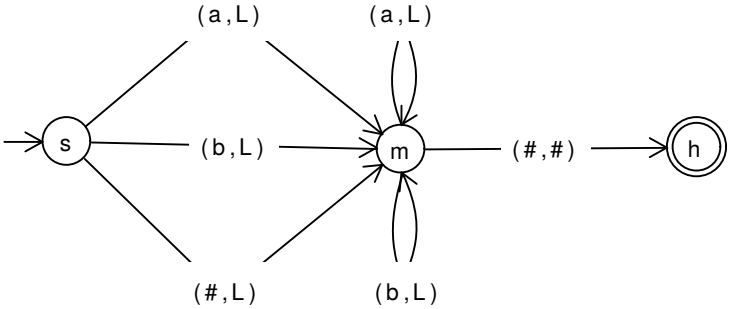
Eine Turingmaschine wechselt, abhängig vom gegenwärtigen Zeichen unter dem Schreib-/Lesekopf den Zustand und kann dabei wahlweise ein anderes Zeichen schreiben oder den Schreib-/Lesekopf um eins nach links oder rechts bewegen.

Eine Turingmaschine haltet, sobald diese in den Zustand “h” wechselt. Die Ausgabe der Turingmaschine, besteht dann in dem Wort das zu diesem Zeitpunkt am Band steht. Vor dem Starten wird die Maschine üblicherweise mit einem durch das Leersymbol “#” begrenzten Wort also beispielsweise “#abc#” befüttert, wobei der Schreib-/Lesekopf anfangs am rechten “#” steht.

Das Band beginnt dabei beim ersten “#” und erstreckt sich unendlich lange nach rechts. Alle nicht gezeigten Positionen am Band seien dabei anfangs mit dem Leersymbol “#” initialisiert. Passiert es, dass der Schreib-/Lesekopf versehentlich links vom Band fällt, so sagt man, dass die Turingmaschine hängt.

Die Konfiguration einer Turingmaschine schreibt man als (q, u, a, v) wobei q der aktuelle Zustand, u die Zeichen vor dem Schreib-/Lesekopf, a das Zeichen direkt unter dem Schreib-/Lesekopf und

v die Zeichen rechts vom Schreib-/Lesekopf sind.



$M_{L_\#} = (\{s, m, h\}, \{a, b, \#\}, \Delta, s)$
$\delta = \{\ ((s,a),(m,L)),\ ((s,b),(m,L)),\ ((s,\#),(m,L)),$
$((m,a),(m,L)),\ ((m,b),(m,L)),\ ((m,\#),(h,\#))\ \}$

Abbildung 1.8: Turingmaschine: Linkssuchmaschine $L_\#$

$$(s, \#ab, \#, \#) \vdash_M (m, \#a, b, \#) \vdash_M (m, \#, a, b) \vdash_M (h, \varepsilon, \#, ab)$$

dabei gilt:

$$\begin{aligned}
&((p,a),(q,b)) \in \Delta \quad \Leftrightarrow \quad \forall\, u, v : (p, u, a, v) \vdash (q, u, b, v)\\
&((p,a),(q,L)) \in \Delta \quad \Leftrightarrow \quad \forall\, u, v \in \Sigma^*, v \neq \#, c \in \Sigma : (p, uc, a, v) \vdash (q, u, c, av)\\
&\quad \wedge \quad \forall\, u \in \Sigma^*, c \in \Sigma : (p, uc, a, \#) \vdash (q, u, c, a)\\
&((p,a),(q,R)) \in \Delta \quad \Leftrightarrow \quad \forall\, u, v \in \Sigma^*, v \neq \varepsilon, c \in \Sigma : (p, u, a, cv) \vdash (q, ua, c, v)\\
&\quad \wedge \quad \forall\, u \in \Sigma^*, c \in \Sigma : (p, u, a, c) \vdash (q, ua, c, \#)
\end{aligned}$$

Die in Abbildung 1.8 gezeigte Linkssuchmaschine $L_\#$ geht bis zum nächsten # nach links. Als Einzelmaschine eher uninteressant, so kann diese erst in einem sogenannten Maschinenschema ihre Wirkung voll entfalten.

Ein Maschinenschema ist nichts anderes als die Hintereinanderausführung mehrerer einzelner Turingmaschinen. Der Haltezustand der vorhergehenden Maschine wird dabei zum Startzustand der nächsten.

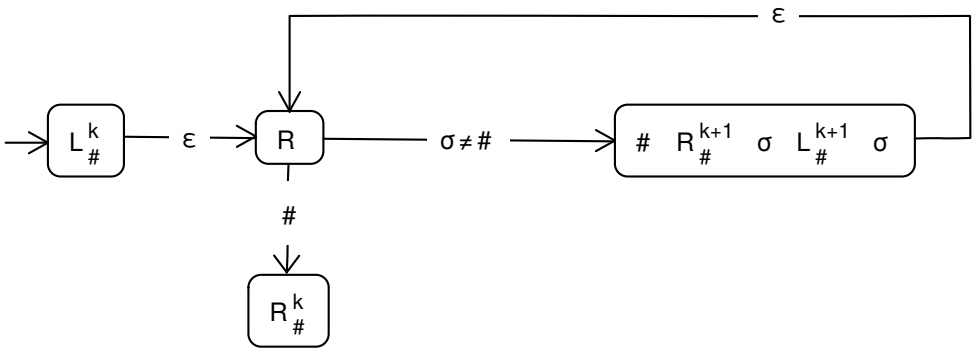


Abbildung 1.9: k-Kopiermaschine

Zwischen verschiedenen Knoten, kann es zudem Übergänge geben, welche auf das Vorhandensein eines bestimmten Zeichens am Band testen (Verbindung des Haltezustands mit dem Startzustand der nächsten Maschine über entsprechende Testkante).

Es ist sogar möglich, dass sich ein Maschinenschema, das aktuell geprüfte Zeichen in einer Variable merkt. Auf unterer Ebene wird dies so implementiert, dass sich die Maschine das aktuelle Zeichen in ihrem Zustand merkt. Für jedes mögliche Eingabezeichen und jeden Zustand in der Ausgangsmaschine muss es dabei einen Zustand in der übersetzten Maschine geben.

Die k-Kopiermaschine in Abbildung 1.9 kopiert dabei das k-te Wort vom rechten Ende des Bandes gesehen am Ende noch einmal dazu. Eine 2-Kopiermaschine macht also aus "#ab#cd#" ein "#ab#cd#ab#". Die Hochzahlen in Abbildung 1.9 bedeuten dabei die k bzw. $k+1$-fache Hintereinanderausführung der entsprechenden Maschine.

Die k-Kopiermaschine arbeitet so, dass diese k-mal nach links geht (oder k+1 mal, wenn das neue Token am Ende des Bandes bereits existiert), sich dort das Zeichen am Band merkt, die dortige Position mit einem # markiert, dann die gleiche Strecke wieder nach rechts geht, dort das gemerkte Zeichen schreibt um dann wieder nach links zu gehen und das Zeichen am linken Band statt der Merk-Markierung # wiederherzustellen. Das R geht danach ein Zeichen nach rechts um das Kopieren des nächsten Zeichens in Angriff zu nehmen.

Wer darüber erstaunt ist, dass mit einem relativ einfachen Berechenbarkeitsmodell wie der Turingmaschine alles berechenbar ist, was überhaupt berechnet werden kann, dem werden wir dies nun kurz plausibel machen. Wir werden hingegen darauf verzichten die Äquivalenz zu anderen Berechenbarkeitsmodellen formal nachzuweisen ([Soch08]).

Am einfachsten lassen sich natürliche Zahlen auf einer Turingmaschine durch eine entsprechende Anzahl an i-s darstellen (oft wird auch das große I dafür genutzt). Vergleichen Sie das Zählen mittels i-s dabei mit dem Aufrechnen von a-s gegen b-s des in Abbildung 1.7 gezeigten Kellerautomaten. Zwei Zahlen können nun addiert werden, indem das trennende # entfernt und durch ein i ersetzt wird und zudem noch am Ende ein i weggenommen wird. Eine Multiplikation von m mal n läßt sich beispielsweise so realisieren, dass n m-mal an das Ende des Bandes kopiert wird. Dabei kann durch # eine Positionsmarkierung in m angebracht werden, ganz so wie die k-Kopiermaschine dies tut.

Die von uns hier bevorzugte Definition von Turingmaschinen ist allerdings nicht die einzig mögliche. Eine weitere Definition ist so beschaffen, dass eine Turingmaschine in einem Schritt gleichzeitig ein neues Zeichen schreibt und den Schreib-/Lesekopf bewegt.

Ein solches Verhalten kann ganz einfach simuliert werden, indem man nach dem Schreiben eines Zeichens in einen dedizierten Zustand übergeht, der dann immer um eins nach links bzw. rechts weitergeht.

Umgekehrt kann man auch für das gleichzeitige Schreiben und Bewegen die Bewegung durch Einfügen eines zusätzlichen Zustandes wieder rückgängig machen, wobei der Schreib-/Lesekopf dann für jedes beliebige Zeichen zurück bewegt werden muss. Soll nur bewegt werden, kann man ganz einfach das aktuell vorhandene Zeichen noch einmal zurückschreiben.

Was den Umstand betrifft, wann Turingmaschinen akzeptieren oder verwerfen, ist die einfachste hier favorisierte Methode zum Erreichen eines solchen Endzustandes das Zurückschreiben von

“#Y#” oder “#N#” auf das Band. Das ist für das Lösen von Entscheidungsproblem, welche nur ein ja oder nein als Antwort haben von Bedeutung.

Es wird aber auch grundsätzlich das Akzeptieren und Verwerfen in Abhängigkeit des Endzustands unterstützt. Dabei gibt es zwei Haltezustände h_0 und h_1. Soll ein Maschinenschema jedoch abfragen, wie eine Untermaschine terminiert hat, ist es notwendig das Ergebnis auf dem Band vorzuhalten, da der Automatensimulator bisher noch keinen Mechanismus bereitstellt um auf das Akzeptieren oder Verwerfen einer Untermaschine explizit zu testen.

Von Turingmaschinen gibt es eine Vielzahl an Varianten, beispielsweise eine mit einem in beide Richtungen unendlichem Band. Ein solches kann simuliert werden, indem man in alle ungeraden Positionen Zeichen des Bandes links vom Ursprung einträgt und in alle geraden Positionen Zeichen rechts des Ursprunges. Die Programmierung der Maschine muss dann noch entsprechend geändert werden, damit diese mit dem neuen Datenlayout arbeitet. Auch Mehrbandturingmaschinen lassen sich auf dieselbe Art und Weise mit einem Band simulieren.

1.6 Erweiterte Notation

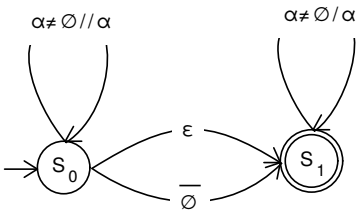


Abbildung 1.10: Palindromscanner

Wenn wir uns an den Palindromscanner von Abbildung 1.6 zurückerinnern, so hat dieser für jedes Zeichen unseres Eingabealphabetes jeweils einen Übergang erfordert. Das ist in der Praxis für große Alphabete sehr mühsam, wenn nicht gar aus praktischen Gründen unmachbar. Der im Rahmen dieser Arbeit vorgestellte und in C++ geschriebene Automatensimulator ist dafür ausgelegt über dem vollen Eingabealphabet von 256 Zeichen zu arbeiten und akzeptiert dafür sogar mit Backslash geschützte Zahlencodes für Steuerzeichen.

Wenn wir uns nun Abbildung 1.10 anschauen, so stellen wir fest, dass der Kellerautomat durch Einführung einer Variablen, die für den Übergang lokal bleibt und nach diesem nicht mehr gilt, das Aufzählen aller Zeichen vermeidet.

Etwas gewöhnungsbedürftig ist vielleicht die Notation mit der leeren Menge $\emptyset$. Wir schreiben hier die volle Menge des gesamten Zeichensatzes als Komplement (i.e. Negation) der leeren Menge. Bei den rekursiven Übergängen wird dies durch ein Ungleichheitszeichen erreicht “$\neq$”, während der Zustandswechsler für das Komplement der leeren Mengen deren Symbol mit einem Oberstrich versieht.

Die Marken für die Übergänge können dabei natürlich nicht so eingegeben werden wie sie schließlich dargestellt werden. Eingegeben wird für den ersten rekursiven Übergang “$\backslash\{alpha\}! = //\backslash\{alpha\}$” und für die Zwischenzustandsübergänge einfach nur “! =” bzw. “”. Das Symbol für die leere Menge braucht dabei nicht explizit eingegeben zu werden, da dieses in den gezeigten Beispielen durch den Zeichenstring der Länge Null gegeben ist. Ein leerer String für einen

Übergang wird dabei zu einem Epsilonübergang, während ein leerer String mit vorangestelltem Ungleichheitszeichen das Komplement der leeren Menge darstellt.

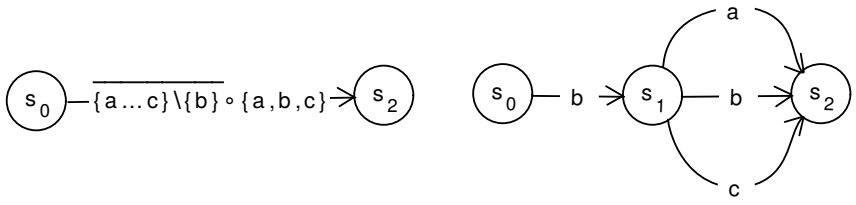


Abbildung 1.11: Erweiterte Notation für endliche Automaten

In Abbildung 1.11 sehen wir was die erweiterte Notation für endliche Automaten leisten kann. Wir nehmen an, dass der Automat unter dem Eingabealphabet $\{a, b, c\}$ arbeitet. Mehrschrittige Übergänge werden durch ein "$\circ$" getrennt, wobei jedes "$\circ$" zu einem neuen Zustand ersetzt wird (s_1), wie in der rechten Hälfte der Abbildung zu sehen ist. Weiters können durch "..." und "$\setminus$" Zeichenmengen mit Bereichsangaben und Mengenabzug gebildet werden. Für jedes Zeichen in der Zeichenmenge wird einer von mehreren parallelen Übergängen generiert.

Beim Editieren der Übergangslabels des linken Übergangs erfolgt die Eingabe folgendermaßen: "$! = a\backslash..c\backslash - b\backslash + abc$". "$\backslash..$" generiert dabei ein Unicode Sonderzeichen für Bereichsangaben, "$\backslash-$" das Mengenabzugszeichen, welches vom Backslash zu unterscheiden ist, und "$\backslash+$" den Ring-Operator.

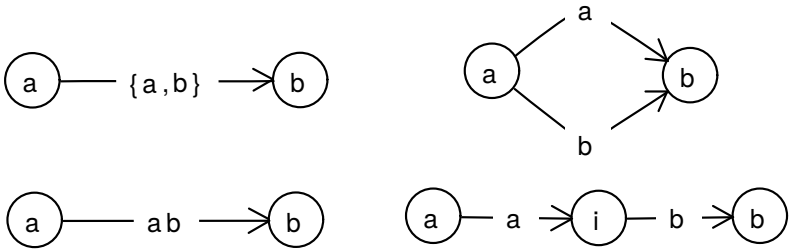


Abbildung 1.12: Bei der erweiterten Notation zu beachten

Abbildung 1.12 oben zeigt das Ergebnis von "$ab\circ$", während unten das Ergebnis von einfach nur "ab" dargestellt ist. Mehrbuchstabige Übergänge werden nämlich standardmäßig als mehrschrittig interpretiert. Erst wenn sich ein Mengenaufzählungs oder Mengenabzugszeichen unter den Zeichen befindet oder eben die Eingabe mit "$\circ$" terminiert wird, dann schaltet der Automatensimulator um zu parallel anzuwendenden Übergängen. Das Terminieren der Eingabe mit "$\circ$" für eine einfache Zeichenmenge mag zwar etwas gewöhnungsbedürftig sein, dafür erspart man sich aber die Verwendung des Beistriches, der dann für die volle Verwendbarkeit des Zeichensatzes wieder mit einem Backslash geschützt werden müsste, so er im Eingabealphabet auftaucht.

Wichtig ist, dass in Maschinenschemata bei Übergängen mit Variablen das Mengenabzugszeichen genutzt werden sollte, um mit Sicherheit deterministisch zu bleiben. Gibt es nämlich zwei Übergänge, einen mit "α" und einen mit "β" und wird die Variablenbelegung so gewählt, dass beide Variablen denselben Wert haben, so kommt es zu einer nichtdeterministischen Verzweigung,

wobei die zwei neuen Folgezustände parallel vorgehalten werden. Deshalb immer "α", "$\beta\backslash\alpha$" und "$\gamma\backslash\alpha\beta$' schreiben, um unabsichtliche Nichtdeterminismen zu vermeiden.

Abschließend sei hier noch eine Zusammenfassung zur Verwendung von Variablen im Automatensimulator gegeben. Bei Kellerautomaten gibt es wie bereits gezeigt sog. übergangslokale Variablen, bei Maschinenschemata Variablen in Übergängen, die danach für die restliche Ausführung des Automaten definiert bleiben. Variablen in Turingmaschinen können gleich wie in Maschinenschemata verwendet werden.

Neben diesen dynamischen Programmvariablen gibt es noch global vordefinierte Variablen in allen Automaten. Diese werden als Parameter von einem Maschinenschema übergeben oder können vor der Simulation von Automaten zusammen mit dem Bandinhalt festgelegt werden.

Weitere Hinweise zur Bedienung des Automatensimulators entnehmen Sie bitte der "Quick Help Page" bzw. den Kurzinformationen des Programms.

2 das Programm

2.1 Einführung

qCoAn ist vollständig unter C++ implementiert, verwendet Qt als grafische Bibliothek und übersetzt damit auf Windows, MacOS und Linux. Eine ältere Vesion des Simulators (v1.0) war hingegen mit Delphi unter Object Pascal implementiert worden und ist nur unter Windows lauffähig. Der Name qCoAn steht übrigens für Compiler und Automaten Netzwerk Simulator und das vorangestellte Q weist darauf hin, dass es sich um ein Qt-Programm handelt. Auch wenn es bis dato keine explizite Unterstützung für Compiler, also für LR oder LL - Parser gibt, so bleibt doch der Name. Vielleicht wäre der Name qTuAn besser gewählt gewesen, da neben Automaten auch noch Turingmaschinen und Maschinenschemata unterstützt werden.

Die Software von qCoAn ist im wesentlichen in drei Schichten aufgebaut. Das hat einerseits den Vorteil, dass die grundlegenden Ein-/Ausgabe und Stringverwaltungsroutinen auf Ebene eins grundsätzlich auch mit jedem anderen Programm verwendet werden können und andererseits, dass CoAn auch völlig ohne Qt nur als Konsolenanwendung übersetzt werden kann. Qt ist das Toolkit, das CoAn für seine grafische Oberfläche verwendet [BlSu11].

1. grundlegende Datentypen & Ein- Ausgaberoutinen
2. Definitionen von Automaten und Routinen zu deren Ausführung/Simulation
3. grafische Benutzerobefläche

Die grundlegende Eigenschaft solche Schichten ist, dass jede Schicht zwar alle darunterliegende Schichten voraussetzt und erfordert, jedoch auch ohne die darüber liegenden Schichten auskommt also übersetzt und verwendet werden kann.

Derzeit können als Konsolenanwendung nur verschiedene Testprogramme, welche die korrekte Funktionsweise der grundlegenden Bibliotheken und der Automatdefinitionen überprüfen, übersetzt werden. Die Automatendefinitionen machen das Herzstück des Simulators aus: Diese regeln die Speicherdarstellung von Automaten und verantworten die Ausführung oder Simulation der Automaten. Mehr dazu ab Kapitel 2.3. Es ist jedoch geplant auch eine echte Konsolenanwendung für qCoAn namens CoAn zu schreiben, welche dann die Ausführung der zuvor über der GUI (grafische Benutzeroberfläche) definierten Automaten auf beliebigen Eingaben im Batch-Betrieb (Stapelverarbeitungsbetrieb) garantieren kann. So könnte man beispielsweise automatisierte Tests über Maschinen ohne GUI laufen lassen oder das Programm einfach als RegExp – Matcher (Matcher für reguläre Ausdrücke) und Mini-Parser für Konsolenanwendungen benutzen.

Es ist aber bereits ein Gewinn verschiedene Testprogramme auf den Bibliotheken und dem Herzstück des Simulators laufen lassen zu können ohne zeitaufwendig die Komponenten für

die grafische Oberfläche mitübersetzen zu müssen. Außerdem ist in manchen Fällen eine Konsolenausgabe einfacher zu überprüfen. Soll zum Beispiel etwas im Herzstück des Simulators restrukturiert werden (i.e. Refactoring betrieben werden), so kann man einfach die Konsolenausgabe von main.cpp vor und nach der Restrutkturierung vergleichen. Diese beinhaltet eine textuelle Repräsentation aller Ausführungsschritte verschiedener im Programmcode definierter Automaten.

2.1.1 make

Während das Qt-Programm qCoAn mit der von Qt bereitgestellten Make-Infrastruktur arbeitet, hat die Konsolenanwendung ihr eigenes Makefile: Dieses ist in den Archiven als makefile.console benannt und kann durch Weglassen von .console im Dateinamen zum Hauptmakefile gemacht werden. Es tut nicht viel außer Makefile.build aufzurufen, welches Code für die Cross-Compilierung und die Abhängigkeiten zwischen den Modulen enthält, damit nur die neu veränderten Dateien auch neu übersetzt werden müssen. Beide Makefiles, von denen eines sauber im entsprechenden build-Verzeichnis arbeitet inkludieren Makefile.includes. Mehr über die Funktionsweise und Programmierung von GNU Make unter Linux erfährt man im entsprechenden Buch [Meck04]. Die Cross Compilierung ist sowohl für Linux als auch Windows, für 32bit sowie 64bit x86 Systeme ausgelegt.

Meistens wird man aber das ganze Programm inklusive GUI übersetzen wollen. Dafür generiert man zuerst ein Makefile über “qmake -project qcoan.pro” und ruft dann “make” auf. Unter Windows kann man auch direkt mit der Microsoft Visual Studio Community Edition übersetzen ohne GNU Make zu verwenden, sofern Qt installiert ist. Eine Visual Studio Projektdatei generiert man dabei mit “qmake -tp vc qcoan-windows.pro”.

Egal ob man mit dem Makefile für Qt oder der Konsolenanwendung arbeitet, das Ergebnis wird immer sauber in einem eigenen Unterverzeichnis bereitgestellt, sodass Sourcecodes und Objektcodes nicht unübersichtlich vermischt werden können.

2.1.2 Portierung

Die Portierung des entsprechend unter Linux entwickelten Programms auf Windows hat hauptsächlich ein Umschreiben der class/struct Konstruktoren für Konstanten erfordert. Unter Visual Studio ist es leider nicht möglich die Feldnamen bei der Initialisierung anzugeben. Da der Code aber mit Feldnamen leichter lesbar ist, findet man oft eine bedingte Kompilierung für beide Compiler.

Weiters kann man unter Visual Studio nicht Arrays variabler Länge als Lokalvariablen definieren. Hier wird dann einfach mit alloca auf dem Stack ein groß genuges Feld alloziert und ein Zeiger darauf eingerichtet, der in C wie ein Array verwendet werden kann.

Auch ist es nicht möglich Wertebereiche in switch – Strukturen anzugeben; diese müssen dann zuvor mit einem if abgefangen werden, was hauptsächlich im Modul für arithmetische Ausdrücke notwendig war (arithexpr).

Schließlich kompiliert das Programm mit zwei verschiedenen Compilern unter Linux: clang und g++. Einige Änderungen waren bereits für clang notwendig, wie beispielsweise, dass es keine Arraykonstruktoren für Klassen gibt:

```
class TextExtentCallStruct { public:
  QFontMetricsF *metric[3], metric0,metric1,metric2; QFont font[3];
...
  TextExtentCallStruct( QPaintDevice *device, QFont elmLabelFont, QFont sub1Font, QFont sub2Font ) :
    metric0(QFontMetricsF(elmLabelFont,device)), metric1(QFontMetricsF(sub1Font,device)), metric2(
        QFontMetricsF(sub2Font,device)),
...
     metric[0] = &metric0; metric[1] = &metric1; metric[2] = &metric2;
```

Src. 2.1: Keine ArrayKonstruktoren für Klassen

Statt `QFontMetricsF metric[3]` und `TextExtentCallStruct(...): metric( { QFontMetricsF( elmLabelFont,device), QFontMetricsF(sub1Font,device), QFontMetricsF(sub2Font,device)} )` muss daher wie in obigem Listing der Umweg über `QFontMetricsF *metric[3]` gegangen werden (siehe coan-v0.8.1.4).

2.1.3 Eigene IO und Stringverarbeitung

Aufgrund der Schichtenarchitektur und weil das Konsolenprogramm auch ohne Qt übersetzt werden können soll, ergibt sich die Notwendigkeit eines eigenen String- oder Texttyps. Es stellt zwar auch die Standardbibliothek von C++ einen Typ namens "string" bereit, doch ist dieser nicht gleich vielseitig einsetzbar wie die neu definierten xstr – Typen, die u.a. Referenzen auf Teilstrings erlauben, ohne den ganzen String kopieren zu müssen, was u.a. auch für die Parameterübergabe nützlich ist.

Ein Vorteil eigener IO – Routinen ist, dass der Zeichensatz vor der Ausgabe noch konvertiert werden kann. Text-Konstanten im Simulator sind ausnahmslos als Utf-8 gegeben, können aber on-the-fly in Latin-1 konvertiert werden, wenn die Konsole mit diesem Zeichensatz läuft. Ein weiterer Zeichensatz wird für das Laden aus Datei verwendet, der ähnlich Latin-1 nur 256 Zeichen umfaßt aber an gewisser Stelle eine Einblendung für griechische Zeichen hat. Der alte unter Windows laufende CoAn-Simulator, der noch unter Delphi / Object Pascal geschrieben war, kannte nämlich keinen Unicode und wechselte für einen gewissen Zeichenbereich innerhalb der verfügbaren 256 Zeichen den Zeichensatz. Zeichen, des neuen unter C++ geschriebenen Simulators werden hingegen gleich als Utf-8 gesichert und zurückgeladen.

Weiters haben die spezifischen xstr – Typen eigene IO-Routinen oder besser gesagt Operatoren. Die Syntax für die Ausgabe erfolgt ähnlich der Syntax der Standardbibliothek, also "cout << xstr << endl". Diese könnten mit den Streams der Standardbibliothek von C++ sonst nämlich gar nicht ausgegeben werden.

2.2 Aufteilung in Klassen und Module

Jede Headerdatei (*.h) mit zugehöriger Implementierung (*.cpp) kann man grob gesehen als eigenes Modul interpretieren. Bis auf eine Ausnahme hat jede Headerdatei nur ein sie implementierendes Modul. Für automatdef.h gibt es jedoch zwei: eines für die eigentliche Automatendefinition (automatdef.cpp) und eines für grafische Ein-, Ausgaberoutinen, die hier schon eine Ebene unterhalb der eigentlichen GUI implementiert sind (ElementGraphics.cpp).

Alle grafischen Klassen und Routinen auf Ebene 2 (GraphicalNode, ElementGraphics.cpp) werden bedingt compiliert, das heißt diese werden nur mitübersetzt wenn Qt in Verwendung ist, also wenn

Ebene 3 vorhanden ist. Immer wenn eine Headerdatei (.h) mit zugehöriger Implementierung (.cpp) gemeint ist, haben wir untenstehend die Dateiendung weggelassen (wie bspw. GraphicalNode statt GraphicalNode.cpp und GraphicalNode.h). Meistens umfaßt die bedingte Compilierung nur einige Codeblöcke in Headern und Implementierungen. GraphicalNode wird hingegen ohne Qt erst gar nicht eingebunden und die beiden dort definierten Klassennamen EllipticalNode und RectangularNode werden einfach über ein Makro auf Node zurückgeführt während die Datei ElementGraphics.cpp zwar immer eingebunden wird ohne Qt aber im wesentlichen leer bzw. unübersetzt bleibt.

Ebenfalls nicht verfügbar ist die Routine SaveAllAtmts ohne Qt in LoadSaveAtmt, da Automaten derzeit nur über die grafische Oberfläche verändert werden können. Die Tatsache, dass Automatensammlungen aus Dateien (*.atm) auch ohne GUI geladen werden können müssen, erfordert, dass die zugehörigen Datenstrukturen auf dem intern verwendeten Stringtypen (xstrbuf, xstr_const) aufbauen und nicht auf dem Typ QString von Qt. Da aber die grafische Oberfläche darüber ausnahmslos den Typen QString für Text verwendet, müssen die Datenstrukturen für den Automatenbaum in TreeWidget und für die Bandauswahl in InputSelector doppelt vorhanden sein (als QString und als xstr). Beide Strukturen müssen folglich mit entsprechendem Mehraufwand stets synchron gehalten werden. Das Laden und Speichern erfolgt nämlich ausschließlich über den internen Typen xstrbuf und xstr_const.

In folgendem Listing sind alle Dateien aller drei Ebenen zusammengefaßt sowie die Namen der darin enthaltenen Klassen, bis auf einige nur lokal in der entsprechenden Datei verwendete Klassen. Es ist zudem eine Einschränkung von Qt, dass nicht beliebig viele Klassen in einer Header-Datei implementiert werden können. Sobald nämlich eine Klasse eigene sog. Signals oder Slots enthält, darf diese nicht mit anderen Klassen, welche ebenfalls eigene Signale und Slots enthalten, sich einen Header teilen. Signale und Slots sind ein Kommunikationsmechanismus von Qt der das Signal, das beispielsweise beim Drücken eines Knopfes ausgelöst wird, mit einem Slot, das ist eine Routine, die daraufhin ausgeführt wird, verbindet. Da die Verbindung erst zur Laufzeit dynamisch hergestellt wird, erfordert dies eigene Metainformationen zur späten Bindung, welche von moc dem Meta Object Compiler von Qt durch Analyse der Sourcecodes bereitgestellt werden. Daß moc für alle Header mit Q_OBJECT Makro automatisch augerufen wird erledigt die von qmake ausgehende Makefile-Infrastruktur.

Auflistung aller Module:

1. Ebene: Bibliotheken
   a) basedefines.h: Datentypen int8/uint8 – int64/uint64, Kompilerdirektiven wie likely und ___noinline, Präprozessorvariablen wie OSTYPE
   b) auxtypes: verschiedene Stringtypen (bspw. estr_const), IOStream – Typ
   c) symtab: Symboltabelle
   d) arithexpr: arithemtische Ausdrücke parsen und auswerten: für Wertparameter von Maschinenschemata
   e) charset: StaticCharSet als Bitfeld mit 256 bit, ExtCharset: gegeben als estr_const mit Bereichsangaben und Abzugsmengen; collectSymbolsAndVars, alwaysDisjoint

f) Testprogramme: test_aux.cpp, test_symtab.cpp, test_charset.cpp, qauxtest.cpp

2. Ebene: Automatendefinition und Simulation

a) automatdef: Definition der abstrakten Typen Runtime, State, Element, ElementProxy, GraphElement, Node, Edge, Automaton

b) GraphicalNode: EllipticalNode, RectangularNode

c) ElementGraphics.cpp: weitere Implementierung von automatdef.h – Textausgabe mit Sub- und Superscript, Kanten (Edges) zeichnen und einzelne Eckpunkte von Kanten verschieben (trackEdges)

d) FiniteAutomata: endliche Automaten, Kellerautomaten

e) TuringMachine: Turingmaschinen, Maschinenschemata

f) LoadSaveAtmt: sichern und laden von Automaten

g) main.cpp: Testprogramm

3. grafische Benutzeroberfläche

a) qcoan.cpp: Hauptprogramm

b) MainWindow: Hauptfenster, ButtonSidebar, ButtonDockWidget, SubscriptedPushButton

c) MainAuxDialogs: ViewCFSL, ViewQuickHelp, SelectNewAtmt

d) Settings: AtmtSettings, QuadrPushButton, DisplayVars

e) TreeWidget: AtmtTreeWidget

f) SVGExport: createSVG – Routine

g) AtmtCanvas: editieren von Automaten: AtmtCanvas (abgeleitet von AtmtWatch)

h) AtmtExec: AtmtWatch (Automat bloß zeichnen; ohne weiter Funktionalität), TapeWatch (Band darstellen, benutzt InputSelector zur Auswahl eines Bandinhaltes, entpsricht der unteren und linken Ansicht im Ausführungsmodus), OutcomeView, OutcomeDialog (verwendet OutcomeView) – Fenster zum Darstellen der Ergebnisbänder nichtdeterministischer Turingmaschinen und Maschinenschemata

i) InputSelector: für die Ausführung von TuringMaschinen: Bandinhalt in eigenem Fenster festlegen

j) ExecEnv(Datei): Klasse ExecEnv: für die Ausführung von Automaten, hält die Klassen AtmtWatch und TapeWatch zusammen, ist neben dem Modul AtmtExec für die Ausführung von Automaten zuständig

Wenn Sie das Programm qCoAn einmal starten, befinden Sie sich zunächst im Übersichtsmodus (MainWindow: AtmtWatch, TreeWidget); das heißt das auf der linken Seite ein Baum mit verschiedenen Automaten angezeigt wird und rechts der aktuell gewählte Automat dargestellt wird. Es ist notwendig mehrere Automaten in einer Datei zu speichern, da sich Maschinenschemata aus verschiedenen untergeordneten Turingmaschinen, die ebenfalls benötigt werden, zusammensetzen. Drückt man F4 oder wechselt den Modus über das Menü, so kommt man in den Editiermodus (AtmtCanvas), wo mit der Maus neue Knoten und Übergänge erstellt oder bestehende verändert

werden können. Danach kommt man mit F4 in den Ausführungsmodus, in dem Automaten simuliert werden können (AtmtExec, ExecEnv). MainWindow hält alles zusammen und regelt auch was geschieht wenn man den Modus mit F4 wechselt.

2.3 Definition von Automaten

Die grundlegenden Datentypen, die für die Definition eines Automaten vonnöten sind, sind in automatdef.h definiert. Entsprechende Automaten wie etwa endliche Automaten oder Turingmaschinen werden von der Klasse Automaton abgeleitet. Diese ist bloß eine Containerklasse, welche jeweils für die Klasse spezifische Knoten und Kanten beinhaltet. Weiters fungiert die Automtenklasse als Factory Klasse für Knoten, Kanten und Runtime-Objekte, kann also Objekte des für den aktuellen Automaten richtigen Typs erzeugen. Für endliche Automaten heißen die Klassen für Knoten und Kanten beispielsweise `FA_Node` und `FA_Edge`. Auch das Löschen und Deallozieren von Knoten und Laufzeitobjekten wird mit den Automaton-Methoden `removeEdge/Node/Element` und `freeRuntime` vorgenommen.

`Element` ist dabei die Superklasse von `Node` und `Edge`. Hier werden einige grundlegende Methoden definiert, die Knoten und Kanten gemeinsam haben. In der direkt von Element abgeleiteten Klasse `GraphElement` wird zudem der Modus (`modus`), eine Fehlerposition (errpos = -1 wenn kein Fehler) und neben den Bildschirmkoordinaten das sog. CursorSet definiert.

Zuerst zum Modus: Dieser regelt die Bildschirmdarstellung des Objekts, also ob dieses markiert oder für die Ausführung wie beim aktuellen Zustand speziell hervorgehoben werden soll. Der Modus ist dabei ein Index in das Array `paintsettings`.

Weiters gibt es noch einen sogenannten Cursor, der bei mehrschrittigen Übergängen anzeigt an welcher Position man sich in der Ausführung befindet. Die Variable `cursorSet` ist ein Bit-Array, das mehrere Cursor für die aktuelle Ausführung des aktuellen Elements speichern kann. Bei nichtdeterministischen Automaten können nämlich nicht nur unterschiedliche Elemente gleichzeitig aktiv sein, sondern auch das aktuelle Element mehrmals mit unterschiedlicher Cursorposition.

Erlaubt ein Element mehr als 7 (für 32bit Computer) oder mehr als 15 (für 64bit Computer) Zwischenschritte, so ist in der Union `cursorSet` nicht die Variable `val` aktiv sondern `ref`, welches dann eine Referenz auf ein Bitfeld enthält. `cursorSet` ist so implementiert, dass jeweils 2 Bit für eine Ausführungsposition reserviert sind. Damit können aktuell in Ausführung befindliche Elemente noch einmal hervorgehoben werden.

Runtime Objekte haben im Gegensatz zu Knoten und Kanten keine Rückreferenz des Automaten auf das Runtimeobjekt, weshalb für einen Automaten zur gleichen Zeit mehrere Runtime Objekte unabhängig voneinander existieren können. Es ist jedoch nicht ganz so, dass während der Ausführung eines Automaten alles was die Laufzeit betrifft nur im Runtimeobjekt schön gekapselt bleibt.

Für die aktuell sichtbare Ausführung werden nämlich Knoten und Kanten direkt im Automaten markiert, weshalb es immer nur eine sichtbare Simulation eines Autoamten geben kann. Untergeordnete Automaten eines Maschinenschemas können hingegen unsichtbar mehrmals gleichzeitig ausgeführt werden. Wenn mehrere Fenster geöffnet sind, können natürlich auch mehrere Auto-

maten gleichzeitig ausgeführt werden. Dies ist jedoch kein Problem, da jedes Fenster auf einer eigenen Kopie der Automaten arbeitet.

Weiters werden in `automatdef.h` noch die Datentypen `Status`, `Data`, `State` und `IntermedState` definiert. `Status` ist ein `enum` und gibt den aktuellen Status eines Automaten wieder, also ob sich dieser gerade in Ausführung befindet (`stepDone`), ob dieser hält, verwirft oder akzeptiert. Wenn es mehrere parallele Ausführungen eines Automaten gibt, wird der aktuelle Status zudem auf den höherwertigsten, also den zuletzt definierten Status kondensiert.

`Data` ist eine Superklasse für Datenobjekte, welche den aktuellen Bandinhalt von Turingmaschinen oder den aktuellen Stapelinhalt von Kellerautomaten speichert. Der Bandinhalt von Kellerautomaten und endlichen Automaten wird hingegen im Runtime-Objekt gespeichert, da dieser nur einmal vorhanden und nur-lesbar ist.

Datenobjekte sind darauf ausgelegt statische Objekte zu sein, das heißt ohne virtuelle Methoden auszukommen. Das beschleunigt die Implementierung und vereinfacht die Speicherdarstellung. Man muss sich veranschaulichen, das aufgrund der Unterstützung nichtdeterministischer Automaten, wie wir noch sehen werden, in jedem Schritt ein neues Datenobjekt mit verändertem Stapel- bzw. Bandinhalt erstellt werden muss.

Demnach sind alle virtuellen Methoden der Datenobjekte auf die virtuelle Klasse Runtime ausgelagert worden: `duplicateData`, `freeData`, `compareData` und etwa. `printData` wie in der Klasse `Runtime_Base` nachzulesen ist. `compareData` ist für das Sortieren von Datenobjekten und die Duplikaterkennung notwendig, wie dies von C++ Mengen-Containern erfordert wird. Weil ein `Data`-Objekt immer von seinem zugehörigen Laufzeitobjekt deallociert werden muss, enthält der Standard-Destruktur von `Data`-Objekten die Ausgabe einer Fehlermeldung sowie ein `abort()` zum Programmabbruch mit Core-Dump. [1]

Um den aktuellen Ausführungszustand eines Automaten zu merken, sind aber Paare von Daten- und Elementobjekten zu speichern, was die Klasse `State` (Zustand) übernimmt. Daneben existiert noch eine Klasse `IntermedState`, welche für den Lookahead (wörtl.: Vorausschau) des aktuellen Zustands, das sind die bei aktueller Eingabe aktiven Übergänge, verantwortlich ist. Diese enthält noch zusätzlich eine Symboltabelle, falls der aktuelle Übergang (i.e. Kante) eine neue Variable definieren sollte. Diese ist deshalb notwendig weil die zuvor in den Lookahead übernommenen Daten aus einem Zustand im Lookahead weder verändert werden können noch freigegeben werden. Doch dazu später mehr.

`ElementProxy` enthält schließlich die abstrakte Definition eines Elements ohne spezifischer Bildschirmdarstellung. Bei mehrschrittigen Übergängen werden die virtuellen Zwischenübergänge und Zwischenknoten als `Elementproxy` gespeichert. Ebenso enthalten die Knoten eines Maschinenschemas mehrere Ausführungsschritte welche über einen `ElementProxy` gespeichert werden. Der `ElementProxy` enthält eine Referenz auf das Hauptobjekt und tut im wesentlichen nichts weiteres als eine fest kodierte Ausführungsposition (`relpos`) zu speichern.

Es war eine Entwurfsentscheidung sog. `ElementProxies` zuzulassen und dafür auf ein `relpos` – Feld in `State` und `IntermedState` zu verzichten. Diese Felder wären derzeit jeweils auch nur von

[1] Die Erzeugung einer core-Datei bei Programmabbruch kann mit `ulimit -c unlimited` ermöglicht werden. Core-Datei dann mit `gdb qcoan core` zum Debuggen verwenden.

einer spezifischen Automatentype genutzt worden: endliche Automaten und Kellerautomaten würden `relpos` in `State` nutzen und Maschinenschemata `relpos` in `IntermedSate`.

Neben `State` und `IntermedSate` gibt es zuguterletzt noch `FinalState`, welches nur dazu da ist haltende, verwerfende oder akzeptierende Zustände für später zu merken. Bei nichtdeterministischen Turingmaschinen und Maschinenschemata können einige der parallel laufenden Maschinen in jedem Schritt ein Ergebnis liefern. Zum Schluß liefern jene Maschinen ein Ergebnis, die dafür am meisten Berechnungsschritte gebraucht haben; dies müssen aber nicht unbedingt die akzeptierenden Maschinen sein oder jene, deren Ergebnis letzten Endes interessant ist. `FinalState` merkt neben Element und Daten noch den `Status` (haltend, akzeptierend, verwerfend) und die Schrittzahl (`step_num`) zu derer die Ausführung der Instanz terminiert hat.

Ein Nichtdeterminismus in Turingmaschinen tritt auf, wenn zu einem Zustand und einer zugehörigen Eingabe zwei verschiedene Berechnungsschritte möglich sind. Der Simulator löst das Problem, indem einfach zwei Maschineninstanzen mit jeweils unterschiedlichem Ergebnis parallel weiterlaufen.

2.4 Vorbereitungen zur Ausführung

Es gibt mehrere Dinge, die vor der Ausführung eines Automaten in Vorbereitung auf diese zu machen sind. Eines davon ist die Bestimmung der Epsilonumgebung aller Knoten. Da nur endliche Automaten und Kellerautomaten echte ε-Übergänge haben, ruft auch nur deren `Autoamton::createRuntime` – Routine `refreshEpsilonClosure` auf. Mit ε beschriftete Kanten von Maschinenschemata hingegen gehen von einem Zustand in einem Ausführungsschritt in einen Folgezustand über, aktivieren aber nicht beide Zustände gleichzeitig wie echte ε-Kanten dies tun.

Zuerst gruppiert `refreshEpsilonClosure` alle Epsilonübergänge mit `std::partition` zusammenhängend am Beginn des `succ`-Feldes, welches die auf einen Knoten folgenden Übergänge enthält. `std::partition` arbeitet dabei wohl so, dass es vom Anfang und vom Ende ausgeht und zur Mitte vorstößt. Immer wenn es links eine nicht Epsilonkante und rechts eine Epsilonkante findet, so können beide Kanten ähnlich dem Vorgehen im Quicksortalgorithmus ausgestauscht werden.

Der eigentliche Algorithmus zum Finden der ε-Hülle arbeitet nun so, dass für jede ε-Kante dem Ausgangsknoten der Folgeknoten zur Hülle hinzugefügt wird. Klar; die Hülle des Ausgangsknotens beinhaltet den Folgeknoten. Danach werden alle ε-Übergänge vom Ausgangsknoten in umgekehrte Richtung beschritten und wieder demselben Folgeknoten von der ursprünglichen Kante zugerechnet. `propagateEpsilon` rechnet also einem Folgeknoten den Epsilonhüllen aller vorhergehenden Knoten zu.

Die vorher mittels `std::partition` gesammelten Epsilonkanten gelten aber nur für Folgeübergänge, nicht für vorhergehende und können daher vom Hüllenalgorithmus nicht genutzt werden. Der Grund warum die Epsilonkanten zusammengesammelt werden ist ja auch, dass diese während der Ausführung einfacher übersprungen bzw. markiert werden können und nicht der Hüllenalgorithmus.

Weiters gibt es neben ε-Übergängen bei Stapelautomaten noch sog. ZeroReadEdges, das sind Kanten die den Stapelzustand verändern aber kein Eingabezeichen lesen. Die optimierten Varianten der Prozedur `computeZeroReadLookAhead` benötigen dabei die Anzahl aller parallelen ZeroRead-Zyklen, die im aktuellen Knoten beginnen und wieder enden. Dieses Datum wird von

`prepareZeroReadTransitions()` in `createRuntime` erhoben und in `max_incoming_zeroRead-Cycles` gespeichert. Mehr dazu im entsprechenden Kapitel.

Zuguterletzt brauchen Maschinenschemata noch einen Backpatching Algorithmus, der die Namen von Automaten durch direkte Referenzen auf das jeweilige Automatenobjekt ersetzt. Die Routine heißt `MS::backpatchAtmtRefs(bool refreshAll)`. Ist der Boolean-Parameter `true` so werden alle Automatenreferenzen erneuert, was insbesondere dann nützlich ist, wenn zwei Automaten so umbenannt worden sind, dass ihre Namen ausgetauscht worden sind.

Der einzige Fall der ausgeschlossen werden muss, ist dass sich ein Automat rekursiv selbst aufzurufen versucht, denn dann kann die Automatenreferenz im Maschinenschema nicht einfach durch die Zustände des Automaten ersetzt werden; i.e. ein solche Ersetzung würde unendlich oft weitergehen, da sich der Automat immer selbst beinhaltet. Auch wenn eine solche Ersetzung zur Ausführung nicht tatsächlich vorgenommen wird, so geht es um die korrekte Definition von Maschinenschemata, welche immer auf eine zusammengesetzte Turingmaschine zurückgeführt werden können müssen.

2.5 Die Ausführung von Automaten

Obwohl viele unterschiedliche Automaten mit jeweils unterschiedlichem Verhalten unterstützt werden, gibt es eine zentral implementierte Routine, welche die Ausführung von Automaten steuert. Automatenspezifisches Verhalten ist dabei in den `Element`-Routinen `lookAhead` und `next` gekapselt.

Grundsätzlich kann man sich die Ausführung von Automaten so vorstellen, dass in einem Schritt von einer Knotenmenge ausgegangen wird. Sodann prüft der Simulator für alle von diesen Knoten ausgehenden Kanten (`lookAhead`), ob diese aktiv sind; d.h. bei aktueller Eingabe und Daten beschritten würden. Ist die Prüfung erfolgreich, so kommen die aktiven Kanten in den Lookahead-Vektor. Wenn die Ausführung einen Schritt weiter rückt, dann werden in die neue Zustandsmenge (`States`) alle Knoten aufgenommen, welche auf die zuvor aktiven Übergänge folgen (next). Zuvor wird bei endlichen Automaten und Kellerautomaten noch die Bandposition mittels `runtime->finalizeStep()` um eine Position weitergerückt.

Ganz so einfach ist es allerdings in der Praxis nicht. Überlegen wir uns doch einmal was bei mehrschrittigen Übergängen passiert. Wir wissen bereits, dass diese weder als Knoten noch als Kanten sondern als Subklassen von `ElementProxy` implementiert sind. Die Implementierung arbeitet deshalb nicht direkt auf Knoten oder Kanten sondern stets auf der Superklasse `Element`.

Ansonsten verhält sich die Implementierung aber genau wie vorhin beschrieben. Zuerst geht man von einer Zustandsmenge aus, ruft dann für jedes Element in der Zustandsmenge die Methode `lookAhead` auf. Die jeweils automatenspezifische Implementierung von `lookAhead` ruft dann für auf den aktuellen Zustand passende Übergänge ihrerseits die `Runtime`-Methode `addLookAhead` auf, welche nichts weiters tut als den übergebenen `IntermedState` dem aktuellen `lookahead`-Vektor hinzuzufügen.

Weiter geht es, indem vom aktuellen `lookahed`-Vektor wieder eine Zustandsmenge gefolgert wird. Dies geschieht, indem für jedes Element im `lookahead`-Vektor die Methode `next` aufgerufen wird. Innerhalb der automtatenspezifischen Implementierung von `next` kann wiederum `Runtime`

`::addState` aufgerufen werden, welches einen neuen Zustand in die folgende Zustandsmenge aufnimmt.

2.5.1 Die Methoden lookAhead und next in endlichen Automaten

```
void FA_Node::lookAhead( Runtime *rg, Data *d, int relpos ) {
  FA_Runtime *r = (FA_Runtime*)rg;
  if( r->pos < 0 || r->pos >= r->tape.length ) return;
  achar curInpChar = r->tape.chars[r->pos];
  for(Edges::const_iterator ie = succ.begin() + epsilonSucc; ie != succ.end(); ie++ ) {
    FA_Edge *edge = (FA_Edge*)*ie;
    if( edge->evalEntryCond( r, curInpChar ) )
      r->addLookAhead( IntermedState( edge, NULL, NULL ) );
  }
}

void FA_Edge::next( Runtime *r, Data *d, SymbolTable *tmpTransitionSymbols, int relpos ) {
  if( relpos + 1 >= conditionLength() )
    r->addState( State( to, NULL ) );
  else
    r->addState( State( &(edge_proxy[relpos]), NULL ) );
}

void FA_Node::next( Runtime *r, Data *d, SymbolTable *tmpTransitionSymbols, int relpos ) {
  cerr << "internal error: FA_Node::next was called; i.e. a node was in the lookahead set." << endl;
  abort();
}
```

Src. 2.2: Beispiele für lookAhead und next: endliche Automaten

In obenstehendem Listing 2.2 sehen wir die Implementierung von `next` und `lookAhead` für endliche Autoamten. Zuerst schaut sich die `lookAhead`-Prozedur eines Knotens alle von ihm ausgehenden Übergänge an. Wenn diese auf die aktuelle Eingabe passen (`edge->evalEntryCond( r, curInpChar )`) wird der Übergang dem Lookahead hinzugefügt. Schließlich kann für Übergänge im Lookahead `next` aufgerufen werden was im Normalfall (`relpos + 1 >= conditionLength()`) den auf den Übergang folgenden Knoten in die Zustandsmenge aufnimmt.

Eine Feinheit von `lookAhead` ist, dass sich dieser nicht um Epsilon-Übergänge kümmert (`succ .begin()+ epsilonSucc`), da diese über einen anderen Mechanismus hinzugefügt werden: Beim Hinzufügen eines neuen Knotens über `Runtime::addState` wird nicht nur der neue Knoten selbst in die neue Zustandsmenge aufgenommen, sondern gleich alle Knoten innerhalb seiner ε-Umgebung.

Grundsätzlich wird für Übergänge im Lookahead stets `next` aufgerufen, während auf Knoten in der Zustandsmenge `states lookAhead` aufgerufen wird.

Eine Ausnahme hierzu bilden mehrschrittige Übergänge. Diese können sowohl in der Zustandsmenge als auch im Lookahead landen, haben jedoch immer nur einen Folgezustand: Wenn noch eine Bedingung des mehrschrittigen Übergangs auszuwerten ist wird `r->addState( State( &( edge_proxy[relpos]), NULL ));` aufgerufen. `relpos` muss dabei nicht inkrementiert werden, da das `edge_proxy` Array bei Index null anfängt und relpos zu Beginn vom Übergang her immer null ist.

Der `edge_proxy` mit Index null hat dann eine `relpos` von eins und so weiter. Der `edge_proxy` ist nichts anderes als ein Wrapper, der `next` oder `lookAhead` seines Elternobjektes mit entsprechender

`relpos` aufruft. Interessant ist, dass `FA_Edge::lookAhead` eines Folgeschrittes, wenn der Übergang also in die Zustandsmenge gekommen ist, bei positiver Evaluierung der Bedingung für den nächsten Schritt `r->addLookAhead( IntermedState( &(edge_proxy[relpos-1]), d, NULL ));` aufruft. Hier muss relpos für den Index dekrementiert bleiben, damit sich die relpos nicht ändert.

Schließlich bleibt anzumerken, dass es unmöglich ist, dass auf einem Knoten eines endlichen Automaten `next` aufgerufen wird, da Knoten niemals im Lookahead landen können. Knoten kommen immer in die Zustandsmenge. Nur falls ein Element über mehrere Schritte hinweg aktiviert bleiben kann, sind beide Methoden zu implementieren: `lookAhead` und `next`.

2.5.2 TraceStep – Zustandsmengen und Lookahead

Die aktuelle Zustandsmenge sowie der aktuelle `lookahead`-Vektor sind Instanzvariablen von `TraceStep`. Jedes Runtime-Objekt enthält mindestens zwei `TraceSteps`: `prev` und `cur`. Gibt es ein mit dem aktuellen Übergang assoziiertes Datenobjekt, welches entweder den Stapelinhalt bei Kellerautomaten oder den Bandinhalt bei Turingmaschinen enthält, so enthält dieses intern einen Referenzzähler. Der Referenzzähler wird jedoch nur bei Aufnahme in die Zustandsmenge erhöht, nicht bei Aufnahme in den Lookahead. Demgemäß jedenfalls der Aufruf der Runtime Methoden `duplicateData` und `freeData`. Ein Datenobjekt gehört also stets einem Zustand (`State`). Daraus folgt, dass es stets mindestens zwei Zustandsmengen geben muss: `prev` und `cur`. Die aktuelle Zustandsmenge kann also nicht direkt aus dem Lookahead heraus überschrieben werden, da sonst die Datenobjekte vor deren erneuter Verwendung dealloziert würden.

Da Übergänge im Lookahead ihr Datenobjekt vom vorhergehenden Zustand nur mitnehmen, aber selber kein neues Datenobjekt allozieren, können Übergänge das Datenobjekt auch nicht direkt ändern. Das ist der Grund warum `IntermedSate` auch eine eigene Symboltabelle hat. Übergänge von Turingmaschinen, Maschinenschemata und sogar Kellerautomaten, können Variablen beinhalten, die während der Ausführung des Übergangs an neue Werte gebunden werden. Im Falle von Turingmaschinen und Maschinenschemata bleibt die Variablenbindung sogar persistent, d.h. sie dauert länger als die Ausführung des Übergangs, sodass die Werte aus der Übergangssymboltabelle bei `next` in die Symboltabelle des Datenobjekts übernommen werden.

```
void MS_Edge::next( Runtime *rg, Data *data, SymbolTable *tmpTransitionSymbols, int relpos ) {
  TM_Runtime *r = (TM_Runtime*)rg; Data *d;
  if( !tmpTransitionSymbols || tmpTransitionSymbols->isEmpty() )
    r->addState( State( to, d = r->duplicateData( data ) ) );
  else r->addState( State( to, d = r->dataAcquireSymTab( data, tmpTransitionSymbols ) ) );
}
```

Src. 2.3: Übernahme der Werte aus der Übergangssymboltabelle in die Symboltabelle des Datenobjektes bei Übergängen von Maschinenschemata

Es ist jedoch vorgesehen, dass sich ein Runtimeobjekt noch mehr als zwei `TraceSteps` merken kann, um in der grafischen Oberfläche nach der Ausführung bis zu einem Ergebnis oder einem Haltepunkt auch noch einmal einen Schritt zurückgehen zu können. Alle Informationen zum aktuellen Ausführungszustand befinden sich nämlich in einem `TraceStep`: Zustandsmenge(`states`), LookAhead(`lookahead`), Status(`status`) und `conditions` welches mehrere Flags über den aktuellen Ausführungszustand speichern kann. Das ganze Array an TraceSteps liegt in `*trace`, hat `tlen` Elemente und die aktuelle Ausführungsposition befindet sich in `tpos`.

Die Zustandsmenge `states` und der `lookahead`-Vektor sind als Standard Template Library (kurz: STL) Container implementiert [Loui96]. Die ganze Implementierung der Core-Funktionalität des Automatensimulators auf Ebene 2 macht Gebrauch von STL Containern. Der Grund warum `states` eine Menge von Zuständen (`std::set<State,State_compare_less>`) ist, ist, dass auf verschiedenen Wegen der gleiche Knoten mit den gleichen Daten aktiv werden kann. Die von einem Knoten ausgehenden Übergänge sind hingegen von Natur aus paarweise verschieden, weshalb hier ein Vektor ohne Duplikaterkennung verwendet wird.

2.5.3 exec & nextStep

Schauen wir nun weiter zu jenen Methoden des Runtime Objektes, welche zur Ausführung des Automaten direkt aufgerufen werden: `nextStep`, `exec` und `exec_traced`.

```
bool Runtime::exec( bool stopOnResult, int maxsteps ) {
  while( !(cur->conditions & all_done) && ( maxsteps != 0 ) ) {
    nextStep(false);
    if( cur->conditions & stopOnErrorMask ) break;
    if( stopOnResult && cur->status > Status::stepDone ) break;
    if( maxsteps > 0 ) maxsteps--;
  }
  return maxsteps != 0 || ( cur->conditions & ( all_done | stopOnErrorMask) ) || ( stopOnResult && cur->
      status > Status::stepDone );
}
```

Src. 2.4: Runtime: die Methode `exec`

Die Routine exec in Listing 2.4 ist im Gegensatz zu `exec_traced`, welche jeden Ausführungsschritt auf der Konsole ausgibt, sehr kompakt. Wenn einmal das Bit `all_done` gesetzt ist, so ist keine weitere Ausführung mehr möglich; es würde bestenfalls eine leerer Zustandsmenge folgen, sodass man die Ansicht des Endergebnisses wieder verlieren würde. Weiters nimmt `exec` noch einen Parameter mit der Maximalzahl an zu beschreitenden Ausführungsschritten entgegen.

Der Grund, warum die Abfrage bezüglich `stopOnErrorMask` und `stopOnResult` mit einem `break` erst nach dem ersten `nextStep` erfolgt, ist, dass bei mehrmaligen Aufrufen von `exec` immer bis zum nächsten Fehler oder bis zum nächsten Ergebnis ausgeführt wird anstatt auf dem aktuellen Ergebnis stehen zu bleiben. Standardmäßig bricht die Ausführung bei jedem Fehler ab. Es können aber mit `setIgnoreErrorMask` Fehler wie beispielsweise `undefined_variable_during_lookahead` ignoriert werden. Die `enum Condition` in der Klasse `Runtime` listet alle möglichen Fehlerzustände und weitere Flags für den aktuellen Ausführungszustand wie `all_done` auf.

Der Rückgabewert von `exec` gibt wieder, ob die Ausführung erfolgreich bis zum nächsten Zwischenergebnis vorangeschritten ist, was immer der Fall ist wenn `maxsteps` $\neq 0$, also wenn der Ausführungsschrittzähler `maxsetep` noch weitere Ausführungsschritte zugelassen hätte. Bricht die Ausführung mit dem zuletzt erlaubten Schritt ab, so muss noch einmal geprüft werden ob der Abbruch nur wegen des letzten Schrittes vonstatten gegangen ist, oder ob es dafür auch noch einen anderen Grund gegeben hat.

```
void ExecEnv::start_exec() { if(rt) return;
  rt = atmt->createRuntime( viewTape->tape,-1, viewTape->charparams,viewTape->valparams, tracelen,true );
  viewTape->rt = rt; addTapeView->rt = rt;
}

void ExecEnv::run( bool stopOnResult ) {
  simulateTimer->stop(); if(!rt) start_exec();
  if(rt) {
    unmark();
    bool quitted_before = !rt->exec(stopOnResult,100000);
    updateStatusBar(); if(quitted_before) statusBar->showMessage("quitted after 100.000 steps.",2000);
    mark(); viewTape->update(); addTapeView->update();
  }
}
void ExecEnv::run_through() { run(false); }
void ExecEnv::run_till_result() { run(true); }
```

Src. 2.5: Ausführung von Automaten im Modul ExecEnv

Kommen wir nun zum Modul `ExecEnv`, das in der GUI für die Ausführung von Automaten zuständig ist. Beispielsweise ruft `ExecEnv::run exec` direkt auf (Listing 2.5). Wenn wir `ExecEnv ::start_exec` anschauen, so benutzt dieses das aktuelle Automatenobjekt als Factory um ein passendes Runtime-Objekt zu erzeugen. Der dabei übergebene Standardparameter -1 sagt, dass die Bandposition auf den dafür vorgegebenen Wert gesetzt wird (bei Turingmaschinen am Ende, bei endlichen Automaten und Kellerautomaten an den Anfang).

`viewTape` ist eine Instanzvariable der Klasse `TapeWatch` (Modul `AtmtExec`), welche im Konstruktor von `MainWindow` an den Konstruktor von `ExecEnv` übergeben wird. `viewTape` ist dabei der Bereich unterhalb des Automaten, der bei der Simulation von Automaten das Band anzeigt. Klickt man auf `viewTape`, so kann man vor der Ausführung eines Automaten einen neuen Bandinhalt auswählen (Dialog der Klasse `InputSelector`).

Ist für `viewTape` ein Runtimeobjekt definiert, so zeigt `viewTape` den Bandinhalt der aktuell ausgeführten Instanz des Automaten an; ansonsten die letzte Auswahl des Benutzers von `viewTape-> inputSelector`, welche in den `viewTape`-Instanzvariablen `tape`, `charparams` und `valparams` gespeichert wird.

Kommen wir nun nochmals zu `ExecEnv::run`. Dieses wird bei Drücken von F9 mit `stopOnResult true` und bei Drücken von Alt/Umschalt/Strg-F9 mit `stopOnResult false` ausgeführt. Dafür existieren zwei Wrapper-Prozeduren `ExecEnv::run_through` und `ExecEnv::run_till_result`, weil die Klasse `MainWindow` diese nur ohne Parameter als Slots mit dem entsprechenden `triggered()`-Signal der `execRunAction` und `execRunThroughAction` – `QAction` verbinden kann.

Als erstes wird der `simulateTimer` gestoppt, der bei Drücken von F6 alle halbe Sekunden einen Schritt weiter geht. Wenn sich noch keine Instanz der Maschine in Ausführung befindet so wird eine solche Instanz zuerst über `start_exec()` erzeugt. Danach wird die Markierung für die Elemente des aktuellen Ausführungsschrittes aufgehoben (`unmark()`), falls sich der Automat bereits in Ausführung befunden hat.

Nun wird der Automat bis zum Ende, bis zum nächsten Zwischenergebnis oder bis 100.000 Schritte vergangen sind, ausgeführt. `updateStatusBar()` gibt danach den aktuellen Status in der Statusleiste sowie etwaige Fehlerbedingungen aus, also beispielsweise “akzeptiert”. `statusBar->`

`showMessage(...)` gibt bei vorzeitigem Abbruch für 2 Sekunden eine enstprechende Meldung aus, bis wieder `stepDone` von `updateStatusBar()` angezeigt wird.

Abschließend müssen noch alle zum Schluß aktiven Zustände markiert werden (`mark()`) und ein Repaint muss für `viewTape` anberaumt werden, da sich durch die Ausführung zumindest die Bandposition geändert hat (`viewTape->update()`, linke Bandansicht: `addTapeView->update()`).

2.5.4 Termination von Maschinen & Lookahead – Berechnung

So unterschiedlich wie die Bedingungen sind unter denen einzelne Automaten terminieren, so unterschiedlich sind auch die Mechanismen um die Termination einer Maschine festzustellen.

Endliche Automaten und Kellerautomaten terminieren sobald die ganze Eingabe gelesen ist. Damit ein solcher Automat akzeptiert muss er sich zum Schluß in einem Endzustand befinden (`elm->isFinalNode()`). Bei Kellerautomaten muss zusätzlich der Stapel leer sein. Weiters gibt es bei beiden Automatentypen noch die Möglichkeit, dass kein Folgezustand definiert ist; dann verwirft der Automat schon bevor er die gesamte Eingabe gelesen hat.

Turingmaschinen hingegen terminieren erst wenn diese einen Haltezustand erreichen. Haltezustände haben als Marke ein “h”. Es gibt Turingmaschinen, die einen akzeptierenden von einem verwerfenden Haltezustand unterscheiden (indexiert mit 0 oder 1 bzw. mit “a” oder “d”) und solche bei denen diese Unterscheidung nur durch den Bandinhalt getroffen wird (“#Y#” oder “#N#” am Ende des Bandes).

Gänzlich anders wird das Halten eines Maschinenschemas festgestellt. Diese halten am Ende eines Knotens der keine ausgehenden Übergänge mehr hat. Das Maschinenschema darf allerdings nicht schon halten wenn ein solcher Knoten zuerst in die Folgemenge aufgenommen wird, sondern erst wenn alle Submaschinen des Endknotens erfolgreich ausgeführt worden sind.

Der erste Terminationsmechanismus geht über die virtuelle Runtime - Methode `getResultStatus ( Element *e, Data *d )`. Diese wird von Runtime::addState, also immer dann wenn ein `next` einen neuen Folgezustand für ein Lookahead-Element hinzufügt, aufgerufen. Stellt sich heraus, dass es sich um einen Status ungleich `stepDone` handelt, so terminiert der Automat, sobald dies für den Status aller neu hinzugefügten Knoten gilt. Dies wird in der Praxis so implementiert, dass das Bedignungsflag `all_done` von `nextStep` in `condition` standardmäßig am Anfang eines Schrittes gesetzt wird und danach beim ersten Auftreten von `stepDone` wieder gelöscht wird.

Bei endlichen Automaten und Kellerautomaten ist dies nun zum Schluß per definitionem der Fall (i.e. kein Folgezustand der `stepDone` meldet), denn diese geben, wenn die Eingabe gelesen ist, immer “accepted” oder “dismissed” zurück.

Der endgültige Status eines Schrittes ist immer der höchstwertige irgendeines Knotens oder Elements für das `getResultStatus` aufgerufen worden ist. Bei endlichen Automaten und generell ist das Akzeptieren höherwertig als das Verwerfen (“dismissed”), weshalb diese akzeptieren sobald nur ein einziger Akzeptorknoten zuletzt vorhanden ist.

```
Status FA_Runtime::getResultStatus( Element *e, Data *d ) {
  if( pos < tape.length ) return Status::stepDone;
  if( e->isFinalNode() && !d ) return Status::accepted; // mayAcccept(d) == !d
  else return Status::dismissed;
}
```

Src. 2.6: getResultStatus für FA und PDAs

Wie in Listing 2.6 zu sehen ist, muss der Datenzeiger zum Akzeptieren `NULL` ergeben was für endliche Automaten immer der Fall ist und für Kellerautomaten genau dann wenn der Stapel leer ist. Obwohl diese Routine für `FA_Runtime` definiert ist, erbt `PDA_Runtime` diese ebenso.

Bei Turingmaschinen kann hingegen weitergerechnet werden bis die letzte parallel laufende Instanz terminiert hat (solange also eine Instanz noch `stepDone` zurückgibt). Wie wir allerdings gesehen haben enthält auch die `exec`-Methode von Runtime eine Option beim ersten zur Verfügung stehenden Ergebnis zu pausieren, was durch einen Status größer als `stepDone` angezeigt wird.

```
Status TM_Runtime::getResultStatus( Element *e, Data *data ) {
  TM_Data *d = (TM_Data*)data;
  if( d->pos < 0 ) return Status::hanging;
  if( e && e->isFinalNode() ) {
    TM_Node *haltnode = (TM_Node*)e; // edge can never be a final node
    if( haltnode->label.length > 2 ) {
      echar indicator = haltnode->label.chars[2];
      if( indicator == '1' || indicator == 'a' ) return Status::accepted;
      else return Status::dismissed;
    } else {
      if( d->tape->right(3) == astr_const((const achar*)"#Y#",3) ) return Status::accepted;
      if( d->tape->right(3) == astr_const((const achar*)"#N#",3) ) return Status::dismissed;
    }
    return Status::halted;
  }
  return Status::stepDone;
}
```

Src. 2.7: getResultStatus für TM

`getResultStatus` für Turingmaschinen hält wenige Überraschungen bereit (Listing 2.7). Wenn die Position der Maschine links vom Band rutscht ergibt das ein "hanging", während ansonsten die Maschine nur auf einem Endknoten haltet. Dieser muss mit einem "h", welches optional von einem Indexierungszeichen gefolgt sein kann, beschriftet sein. Deshalb spart man sich bei `haltnode->label.length > 2` das Prüfen des zweiten Zeichens.

`Runtime::addState` prüft aber nicht nur für jeden neu hinzugefügten Knoten via `getResultStatus`, ob dieser terminiert und ein Ergebnis lieftert, sondern es ist auch dafür verantwortlich, dass `rememberOutcome` gerufen wird um sich das Ergebnis dann zu merken. Ein Ergebnis mit Bandinhalt pro Knoten muss sich der Simulator aber nur für Turingmaschinen und Maschinenschemata merken.

Anders bei endlichen Automaten und Kellerautomaten, welche nur ein einziges Ergebnis nämlich akzeptiert oder verworfen auf dem ursprünglichen Bandinhalt haben. Für diese beiden Automaten gibt die virtuelle Methode `rememberOnceNoState` `true` zurück, sodass `rememberOutcome` nicht in `Runtime::addState` gerufen wird sondern nur einmal zum Schluß von `Runtime::lookAhead` (siehe Listing 2.8).

Bleiben wir bei endlichen Automaten. Wir haben den Fall dafür, wenn es keinen definierten Nachfolgeknoten gibt, noch nicht behandelt. Tatsächlich prüft die Prozedur `Runtime::lookAhead`, welche von `nextStep` gerufen wird um den Lookahead zu berechnen, genau darauf. In folgendem Listing 2.8 geschieht mit der Abfrage von `cur->lookahead.empty()` genau das. Die virtuelle Methode `status_4_no_consecutive_state()` ist für endliche Automaten (und damit auch für Kellerautomaten) überschrieben und tut nichts weiteres als "dismissed" zurückzugeben.

```
inline void Runtime::lookAhead_simple() {
  int prev_lh_size = 0, lh_size;
  for(States::const_iterator s = cur->states.begin(); s!=cur->states.end(); s++ ) {
    s->elm->lookAhead( this, s->data );
    lh_size = cur->lookahead.size();
    if( prev_lh_size == lh_size ) {
      Status cur_status = status_4_no_successor(&*s);
      if( cur_status > cur->status ) cur->status = cur_status;
    };
    prev_lh_size = lh_size;
  }
}

enum Status Runtime::lookAhead( bool prepareStatePos ) {
  assert( !( cur->conditions & at_lookAhead ) );
  cur->zeroReadEdgeNum = cur->viewOnlyElements.size();
  cur->conditions |= at_lookAhead;
  if( !( cur->conditions & Condition::all_done ) ) {
    if( prepareStatePos ) lookAhead_with_StatePos(true); else lookAhead_simple();
    if( cur->lookahead.empty() ) {
      if( cur->status == stepDone ) cur->status = status_4_no_consecutive_state();
      cur->conditions |= all_done;
  } }
  if( cur->conditions & Condition::all_done ) {
    if( rememberOnceNoState() ) rememberOutcome( NULL, NULL, cur->status );
    accruedConditions |= cur->conditions & ~Condition::is_the_initial_step;
  }
  if( cur->status > accruedStatus ) accruedStatus = cur->status;
  return cur->status;
}
```

Src. 2.8: Runtime::lookAhead

Nicht alles was die Routine `Runtime::lookAhead` tut, haben wir bereits besprochen. Sogenannte ZeroReadEdges sind Kanten von Kellerautomaten, welche kein Eingabesymbol lesen aber den Stapel prüfen und verändern können. Diese werden in `Runtime::addState` während `next` hinzugefügt, sind also wenn `lookAhead` gerufen wird schon alle bis zum Schluß eingesammelt worden. Neben ZeroReadEdges werden in `cur->viewOnlyElements` während des Lookaheads noch Kanten mit Fehlerbedingungen gesammelt, weshalb sich die Ausführung die aktuelle Position in dem `viewOnlyElements` Array in `cur->zeroReadEdgeNum` merkt. Der Name kommt übrigens daher, dass diese Elemente nur für die Bildschirmausgabe (“viewOnly”) gesammelt werden, während der Inhalt dieses Feldes keine Auswirkung auf die nachfolgenden Ausführungsschritte hat.

Weiters gibt es neben `lookAhead_simple` noch eine weitere Lookahead-Prozedur, welche sich die Zugehörigkeit von Lookahead-Übergängen im Lookahead-Vektor zu entsprechenden Zuständen in der Zustandsmenge merken kann. Dies ist dafür gedacht, dass wenn man mit der Maus über einen Bandinhalt fährt, dass dann alle zugehörigen Knoten mit folgenden Übergängen aufleuchten. Das ist auch der Grund, warum das Cursor-Bitfeld zwei Bits pro Cursorposition merkt. Es ist allerdings bis dato auf Ebene der GUI noch nicht implementiert worden.

`accruedStatus` merkt sich den höherwertigsten aller Status, die bis dahin überhaupt in der Ausführung aufgetreten sind während `cur->status` nur den höherwertigsten Status für den aktuellen Ausführungsschritt merkt. Gleiches gilt für `accruedConditions` und `cur->conditions`. Das explizite Setzen von `all_done` bei fehlendem Lookahead hat zudem zur Folge, dass die Ausführung stoppt und die Endzustände nicht verloren gehen. Umgekehrt verhindert ein vorheriges Gesetzsein von

`all_done` die Berechnung eines Lookaheads; wenn ein Automat bis zum Ende ausgeführt worden ist, macht es keinen Sinn Elemente zu zeigen, die im nächsten Ausführungsschritt beschritten würden (Lookahead).

Kommen wir nun zu `Runtime::lookAhead_simple`. Die wesentliche Funktionalität dieser Routine besteht darin `s->elm->lookAhead( this, s->data )` für jedes Element aufzurufen; genau so wie `Runtime::nextStep` zuvor `s->elm->next( this, s->data, s->tmpSmbls )` aufruft. Weiters befindet sich noch Code in der Routine, der prüft ob die Größe des Lookahead-Vektors durch den Aufruf von `elm->lookAhead` gewachsen ist. Wir erinnern uns, dass die Implementierung der virtuellen Methode `elm->lookAhead addLookAhead` ein oder mehrmals aufrufen kann, welche jeweils ein Element zu dem Lookahead-Vektor hinzufügt.

Hat nun ein Zustand keinen auf ihn folgenden Übergang, so sehen wir, dass für diesen Zustand die virtuelle Methode `status_4_no_successor(&*s)` aufgerufen wird. Die ungewöhnliche Kombination von Dereferenzierung via "*" und anschließendem Adreßnehmen via "&" hat in C++ den Sinn, dass hier die Adresse eines über einen Iterator referenzierten Objektes abgefragt wird (würde es sich um einen Zeiger handeln wäre dies eine NOOP (No-Operation)).

`status_4_no_successor` ist genau der Punkt, an dem Maschinenschemata ihre Terminierung prüfen wie in Listing 2.9 zu sehen ist.

```
class MS_Node : public RectangularNode {
  ...
  virtual bool isFinalNode() const { return succ.empty(); };
  ...
};

Status MS_Runtime::getResultStatus( Element *e, Data *data ) {
  TM_Data *d = (TM_Data*)data;
  if( d->pos < 0 ) return Status::hanging;
  return stepDone;
}

Status MS_Runtime::status_4_no_successor( const State *state ) {
  TM_Data *d = (TM_Data*)state->data; Status cur_status = Status::no_successor;
  if( d->pos < 0 ) cur_status = Status::hanging;
  else if( state->elm->isFinalNode() ) {
    if( d->tape->right(3) == astr_const((const achar*)"#Y#",3) ) cur_status = Status::accepted;
    else if( d->tape->right(3) == astr_const((const achar*)"#N#",3) ) cur_status = Status::dismissed;
    else cur_status = Status::halted;
  }
  if( cur_status > Status::stepDone ) rememberOutcome( state->elm, state->data, cur_status );
  return cur_status;
}
```

Src. 2.9: Terminierung von Maschinenschemata

Weiters ist `status_4_no_successor` auch der Ort, an dem sich Maschinenschemata das Ergebnis einer Berechnung merken. Die Ergebnisse können zum Schluß durch Klicken auf den Leistenbereich mit den Bandinhalten noch einmal angezeigt werden, sofern wie bei NDTM und nichtdeterministischen Maschinenschemata mehrere Ergebnisse verfügbar sind (Klassen OutcomeDialog und OutcomeView). Weiters sind die Ergebnisse wichtig, wenn ein Maschinenschema untergeordnete Automaten aufruft, da das Maschinenschema diese Ergebnisse dann für sich übernimmt.

`Status::no_successor` ist dabei der niederwertigste existierende Status, da dieser überschrieben wird sobald nur ein einziger Knoten einen aktiven Übergang, der dann einen Folgezustand hervorbringt, hat.

`isFinalNode()` ist eine virtuelle Methode zur Markierung von Endzuständen. Im Falle von Maschinenschemata mit eckigen Knoten sind aber Endzustände grafisch nicht speziell gekennzeichnet; bei runden Knoten hingegen werden diese durch einen Doppelkreis eingerahmt. `isFinalNode()` gibt nun für Maschinenschemata `true` zurück, wenn es keine auf den Zustand folgende Übergänge oder Kanten gibt.

2.5.5 ZeroRead-Edges

Sogenannte ZeroRead-Edges sind bei Kellerautomaten Kanten, welche kein Eingabesymbol lesen aber auf den Stapel zufgreifen. Dies ist beispielsweise bei Automaten praktisch, die im Endzustand mit einer rekursiven ZeroRead- oder Stapelkante noch alle verbleibenden Symbole vom Stapel holen, damit der Automat erfolgreich terminieren kann. Grundsätzlich sind dem Einsatz von Stapelkanten keine Grenzen gesetzt. Man kann diese beispielsweise auch verwenden um am Anfang ein Symbol auf den Stapel zu legen.

Der Automatensimulator implementiert einen eigenen Mechanismus für ZeroRead-Edges oder Stapelkanten. Wenn das Flag `adding_zeroreads` nicht gesetzt ist, so ruft `Runtime::addState`, welches von `Element::next` gerufen wird, für jeden neu hinzugefügten Knoten `node->zeroRead-LookAhead( this, next.data )`, wobei `this` das Runtimeobjekt ist, auf. Das erste, das die virtuelle Methode `zeroReadLookAhead` zu tun hat, ist dann das Flag `adding_zeroreads` zu Prozedurbeginn zu setzten und vor Ende wieder zu löschen. `zeroReadLookAhead` ruft dann seinerseits für alle über Stapelkanten hinzugefügten Knoten `Runtime::addState` auf, kümmert sich dann aber selbst via Rekursion um weitere Stapelkanten, welche von dem neu hinzugefügten Knoten ausgehen.

`node->zeroReadLookAhead` berechnet aber nicht direkt den ZeroReadLookAhead, sondern ruft hierzu je nach Wert der Präprozessor-Direktive `ZERO_READ_MODE` eine der gewrappten Prozeduren `computeZeroReadLookAhead` oder `simpleComputeZeroReadLookAhead`.

```
#define this_node this_chain_node.node
#define this_data this_chain_node.data
#define this_parent this_chain_node.previous

void PDA_Node::simpleComputeZeroReadLookAhead( Runtime *rg, struct SimpleLookAheadChain this_chain_node )
     {
  PDA_Runtime *r = (PDA_Runtime*)rg;

  for( struct SimpleLookAheadChain *cur = this_parent; cur; cur = cur->previous )
    if( cur->node == this_node )
      if( this_data && ( !cur->data || this_data->stack.length > cur->data->stack.length || ( this_data->
          stack.length == cur->data->stack.length && this_data->stack != cur->data->stack ) ) ) {
  // error: stack length stayed the same or has increased by cycle
  r->setCondition( Runtime::had_an_illegal_epsilon_loop );
  r->freeData( this_data );
  return;
      }

  if( this_parent && !r->addState( State( this_node, this_data ) ) )
    return;

  for( Nodes::const_iterator in = this_node->epsilonClosure.begin(); in != this_node->epsilonClosure.end
       (); in++ )
    for( Edges::const_iterator ie = (*in)->succ.begin() + (*in)->epsilonSucc ; ie != (*in)->succ.begin()
         + ((PDA_Node*)*in)->zeroReadSucc; ie++ ) {
      PDA_Edge *edge = (PDA_Edge*)*ie;
      if( edge->evalEntryCond( r, '\000', this_data, NULL ) ) {
  r->addZeroReadEdge( edge );
  simpleComputeZeroReadLookAhead( r, SimpleLookAheadChain( (PDA_Node*)edge->to, edge->execPushPop( r,
       this_data, NULL ), &this_chain_node ) );
    } }

}
```

Src. 2.10: Einfachstmögliche Auswertung von ZeroRead-Kanten

Schauen wir uns zunächst `simpleComputeZeroReadLookAhead` an (Listing 2.10). Diese Routine führt eine einfach verkettete Liste aller bisher besuchten Knoten und ihres Stapelinhaltes (data) durch `struct SimpleLookAheadChain this_chain_node` am Stapel mit. Das ist wichtig um Zyklen im ZeroReadLookAhead zu erkennen. Ist ein Knoten schon einmal besucht worden, prüft die Routine ob sich die Länge des Stapelinhaltes reduziert hat.

Hat sich die Länge eines Stapelinhaltes in einem Durchlauf erhöht und würde das in den Folgedurchläufen so weiter gehen, so hätten wir plötzlich einen endlichen Stapelautomaten mit einer unendlich großen Menge an möglichen Stapelinhalten. Das muss natürlich verboten sein.

Die Routine ist hier etwas restriktiver als unbedingt notwendig, da diese auch ein Gleichbleiben der Länge des Stapelinhaltes für Zyklen verbietet. Das wäre theoretisch erlaubt, denn mit endlicher Länge kann nur eine endliche Zahl an Stapelinhalten generiert werden. Da eine solche Zahl an Stapelinhalten aber relativ schnell unpraktikabel groß werden kann, ist dies hier nicht erlaubt.

Der Simulator hat auch so eine relativ großzügige Unterstützung von Stapelkanten, die über einfach rekursive ZeroRead-Edges hinausgeht (Der alte Windows-Simulator erlaubte nur einfach rekursive Stapelkanten). Es darf nämlich bezweifelt werden, dass es gewinnbringende Anwendungen für Zyklen mit gleichlangem Stapelinhalt gibt. Die einzig erlaubte Möglichkeit für einen Kreis

mit gleichbleibender Stapellänge ist die Identität, das heißt das am Ende des Zyklus dasselbe herauskommt wie am Anfang eingeschleust worden ist.

Wenn der Zyklus nicht erlaubt ist, so wird die Bedingung `had_an_illegal_epsilon_loop` gesetzt und die weitere Verarbeitung des Zyklusses wird abgebrochen. Wenn wir uns den Code anschauen, der diese Bedingung prüft (`this_data && ( !cur->data ...`), so muss dieser den Spezialfall berücksichtigen, dass der Stapel leer ist und durch ein `NULL` repräsentiert wird.

Weiter geht es mit einem Aufruf von `addState` um den neu gewonnen Stapelinhalt samt zugehörigem Knoten zur neuen Zustandsmenge hinzuzufügen. War der neu generierte Zustand noch nicht Teil der aktuellen Zustandsmenge, so sucht `simpleComputeZeroReadLookAhead` nach weiteren ZeroRead-Kanten, die vom neu hinzugefügten Knoten ausgehen und mit dem neuen Stapelinhalt befüttert werden können.

Das geschieht indem von jedem Knoten aus der Epsilonumgebung des neuen Knotens alle ZeroRead-Kanten abgegangen werden. `edge->evalEntryCond( r, '000', this_data, NULL )` ist dabei bereits darauf vorbereitet auch für Stapelkanten gerufen zu werden. Für Stapelkanten, die kein Eingabezeichen lesen, wird demnach der 2.Parameter, der normalerweise das aktuell gelesene Zeichen beinhaltet, ignoriert. Ebenso braucht es keine Symboltabelle um sich das aktuell gelesene Zeichen für die Gültigkeitsdauer des Überganges zu merken (letzter Parameter).

`r->addZeroReadEdge( edge )` dient nur dazu, dass dem Benutzer die aktuelle Stapelkante auch angezeigt wird, ist jedoch für die weitergehende Ausführung des Automaten nicht von Belang.

Danach wird die neue Kante über eine Rekursion von `simpleComputeZeroReadLookAhead` beschritten. `execPushPop` holt dabei, die zuvor mit `evalEntryCond` überprüfte Sequenz vom Stapel und legt optional noch weitere Zeichen auf den Stapel.

Ein einzelner Übergang kann dabei durchaus mehr auf den Stapel legen als er herunterholt, wichtig ist nur, dass sich bei einem Kreis der Stapelinhalt letzten Endes reduziert. Der letzte Parameter für die dem aktuellen Übergang zugerechnete, temporäre Symboltabelle kann dabei wieder leer bleiben. Der letzte Eintrag im neu generierten `SimpleLookAheadChain`–Objekt, das als Parameter auf den Stapel kommt, ist eine Referenz auf das letzte `SimpleLookAheadChain`–Objekt.

Soweit so gut zur Berechnung des ZeroReadLookAheads über `simpleComputeZeroReadLookAhead`. Es gibt jedoch noch zwei weitere Möglichkeiten wie der ZeroReadLookAhead berechnet werden kann. Diese können über die Präprozessordirektive `ZERO_READ_MODE` eingestellt werden: `ZRM_simple`, `ZRM_goback_alloca`, `ZRM_goback_double_recursive`. Unter `gcc` und `clang` macht dies der `"-D"`-Parameter auf der Kommandozeile.

Die beiden weiteren Berechnungsmethoden leisten funktional dasselbe wie `simpleComputeZeroReadLookAhead`, reduzieren dabei allerdings die Menge des notwendigen Stapelspeichers für die Aktivierungsrecords der eingesetzten Routinen. Verdeutlichen wir uns was passiert wenn `simpleComputeZeroReadLookAhead` auf eine rekursive Kante trifft, welche die aktuellen 100 Zeichen vom Stapel holt. Die Aktivierungsrecords von `ZRM_simple` müssten dafür 100 mal hintereinander auf den Stapel geladen werden, weil die Routine dafür 100 rekursive Aufrufe ihrer selbst erfordert. Mit den anderen beiden Routinen braucht aber nur ein einziges Aktivierungsrecord am Stapel Platz zu finden.

Der Trick, der dabei angewendet wird, ist vom Prinzip her einfach: Wenn ein Kreis erkannt wird und ein neuer Stapelinhalt vorliegt, so wird dieser nach untern hin zur ersten Aktivierung der

Routine durchgereicht. Danach reduziert sich die Rekursionstiefe bis zum Ausgangspunkt des Kreises und die Berechnung beginnt erneut mit dem neu gewonnen Stapelinhalt. Das Durchreichen auf die unterste Ebene ist dabei kein Problem, da wir die Parameterobjekte einfach verkettet haben.

Die von `SimpleLookAheadChain` abgeleitete Klasse `LookAheadChain` hält dabei mittels `newData` und `newDataCount` gleich mehrere Slots bereit, mit denen Stapaelinhalte nach unten hin durchgereicht werden können. Das kann auch notwendig werden, wenn es nicht nur einen sondern gleich mehrere sich parallel verzweigende Zyklen gibt, die im aktuellen Knoten beginnen und enden.

Ob und wieviele parallele Zyklen es geben kann, ermittelt bei Instanzierung eines neuen Runtimeobjektes die Prozedur `prepareZeroReadTransitions`. Diese geht von jedem Knoten ebenfalls rekursiv alle Stapelkanten entlang bis ein Kreis erkannt wird.

Besteht der Kreis zwischen dem ursprünglichen Ausgangsknoten, für den `max_incoming_zeroReadCycles` berechnet werden soll, und dem gerade besuchten Knoten, so kann `newest/oldest.node->max_incoming_zeroReadCycles` inkrementiert werden. Weitere am Weg liegende Zyklen werden dabei nicht berücksichtigt, da es sich bei ihnen nicht um parallele sondern um verschachtelte Zyklen handelt.

Generell können die neuen Routinen nur die Rekursion für parallele, nicht aber für verschachtelte Kreise auflösen. Warum das so ist, können wir uns ganz einfach verdeutlichen. Nehmen wir an, das am Weg von Knoten x zu Knoten x noch ein weiterer Zyklus liegt, der wenn man von x nach x geht entweder beschritten werden kann oder nicht. Diesenfalls ist es so, dass der verschachtelte Zyklus aber nicht nur einmal sondern beliebig oft gegangen werden kann bis man wieder zum Knoten x zurückkommt. Das Problem dabei ist, dass es dann beliebig viele Stapelinhalte gibt, die nach unten hin durchgereicht werden müssten. Das geht natürlich nicht.

Die beiden `computeZeroReadLookAhead` Routinen lösen das Problem so, dass wenn alle Slots zum Hinunterreichen von Stapelinhalten bereits belegt sind, dass sie dann einfach `simpleComputeZeroReadLookAhead` aufrufen und ohne diese Optimierung fortfahren. Generell gilt, dass bei mehreren verschachtelten Zyklen, das Hinunterreichen nur für den innersten dieser Zyklen unbegrenzt funktioniert.

Wenn wir uns `computeZeroReadLookAhead` anschauen, dann sehen wir, dass es dort wo `simpleComputeZeroReadLookAhead` aufhört noch weiter geht. Die nach unten hin durchgereichten Datenobjekte mit Stapelinhalten wollen schließlich auch noch behandelt werden.

Solange immer nur ein Staeplinhalt nach unten hin durchgereicht wird, brauch die Routine hier nichts wesentlich anders zu machen als beim ersten Aufruf. Es liegt lediglich ein anderer Ausgangsstapelzustand vor. Der einzige Unterschied besteht darin, dass der Parameter `firstCall4Cycle` jetzt `false` ist, weil die gleiche ZeroRead-Kante bereits mit `addZeroReadEdge` zu den sichtbaren ZeroRead-Kanten hinzugefügt worden ist. Der neue Aufruf unterscheidet sich nur durch den Stapelinhalt.

Was passiert allerdings wenn es mehrere parallele Zyklen gibt und deren Ergebnis in mehreren Slots hinuntergereicht worden ist. Ein erneuter Aufruf würde nur eines dieser hinuntergereichten Ergebnisse auf einmal berücksichtigen. Es wäre also nur ein einziger Slot wieder frei, während `max_incoming_zeroReadCycles` neu besetzt werden könnten. Das Problem wird so gelöst, dass mit `alloca` ein neues Record mit `max_incoming_zeroReadCycles` Plätzen allokiert und mit dem

vorigen Record doppelt verkettet wird. Alte Records können dabei sukzessive von hinten nach vorne wiederverwendet werden, sobald der Zyklus für alle Einträge des Records neu beschickt worden ist.

Zumindest ist dies die Variante, die bei `ZERO_READ_MODE ZRM_goback_alloca` angewendet wird. `ZRM_goback_double_recursive` verwendet statt der doppelt verketteten Zwischenergebnisrecords doppelte Rekursion, ist deshalb aber nicht schneller und eher schwieriger zu lesen. Die ursprüngliche Motivation hinter `ZRM_goback_double_recursive` war es auf `alloca` zu verzichten und nur ein Array variabler Länge als Lokalvariable zu haben. Da der MS Visual Studio Compiler aber keine Arrays variabler Länge (Länge gleich `max_incoming_zeroReadCycles`) als Lokalvariablen erlaubt, muss hier erst wieder auf `alloca` zurückgegriffen werden.

2.6 Bandinhalte abfragen

Verschieden Automaten speichern ihre Bandinhalte auf verschiedene Weise. Für endliche Automaten und Kellerautomaten trägt das Runtime-Objekt das Eingabedatenband. Kellerautomaten haben zudem noch einen Stapelinhalt, der in einem separaten Datenobjekt gespeichert wird. Turingmaschinen und Maschinenschemata hingegen haben ihren Bandinhalt nur im zugehörigen Datenobjekt.

Nun soll es aber auf Ebene der GUI ein einheitliches Interface zur Darstellung von Bandinhalten geben. Damit erspart man sich eine Fallunterscheidung nach Automatentyp sowie den direkten Zugriff auf Datenelemente, welche eigentlich geschützt sein sollten.

Es gibt ganze drei virtuelle Methoden zum Abfragen von Bandinhalten: `Runtime::numTapes`, `Runtime::getTape` und `Runtime::getStatesAndLookAheadForTape`. Da es Maschinen mit mehreren Bändern gibt (i.e. Stapelautomaten haben einen Stapel und ein Eingabedatenband), reicht ein einfacher skalarer Wert nicht als Index. Es kann dann nämlich aufgrund der Unterstützung nichtdeterministischer Maschinen, mehrere parallel laufende Instanzen mit mehreren Bändern geben. Daher muss für zwei Ebenen von Bändern vorgesorgt sein. Jedes Band der Ebene eins kann beliebig viele untergeordnete Bänder der Ebene zwei haben.

Den Index zu einem Band faßt die Klasse `TapeIndex` mit den Feldern `topidx` und `subidx`. Ein Band der Ebene 1 wird mit `topidx` identifiziert wobei `subidx` auf -1 gesetzt bleibt.

Anders für die Funktion `numTapes`. Diese braucht nur ein Elternelement eines `TapeIndex` zu identifizieren, weil sie die Anzahl an untergeordneten Bändern für dieses Elternelement zurückgibt. Demnach nimmt `numTapes` auch nur einen Parameter der Klasse `ParentTapeIndex`, welche Superklasse von `TapeIndex` ist, und als einziges Feld `topidx` besitzt.

Um beispielsweise die Anzahl an Bändern auf Ebene eins abzufragen verwendet man einen ParentTapeIndex mit `topidx` -1, welches ein leerer Konstruktor `ParentTapeIndex()` zurückgibt. Bei Zuweisung eines `ParentTapeIndex` an einen `TapeIndex` wird `subidx` auf -1 gesetzt.

Mit dieser Methode ließen sich auch mehrbandige Turingmaschinen darstellen. Die Bänder auf Ebene 1 wären die Hauptbänder der jeweiligen parallel laufenden Turingmaschineninstanzen. Bänder auf Ebene zwei könnten für zusätzlich der jeweiligen Maschine gehörende Bänder eingesetzt werden.

Schließlich gibt es noch `getStatesAndLookAheadForTape`. Diese Methode ist derzeit zwar implementiert, wird von der GUI aber noch nicht genutzt. Wenn mehrere Instanzen einer nichtdeterministischen Turingmaschine parallel laufen, so können sich diese Instanzen in ganz unterschiedlichen Zuständen befinden. Im Moment hat der Benutzer aber nicht die Übersicht, welche markierten Knoten zu welcher Instanz und daher zu welchem Band der Ebene eins gehören. Es ist gedacht, dass bei überfahren eines Bandes mit der Maus entsprechende Knoten noch einmal speziell hervorgehoben werden können.

Nun stellt sich die Frage wie die Funktionalität von `getStatesAndLookAheadForTape` implementiert werden kann. Entsprechende Vorkehrungen befinden sich bereits in der allgemeinen Simulationsroutine von Runtime, genauer in `lookAhead_with_StatePos(bool remember)`. Die Routine `Runtime ::lookAhead` (Listing 2.8), welche für alle aktuellen Zustände deren Lookahead berechnet hat dabei einen Parameter namens `prepareStatePos`, der wenn er `true` ist `lookAhead_with_StatePos (bool remember)` statt `lookAhead_simple()` aufruft.

Die Zustandsmenge ist zuerst nach Datenelementen (Stapel- oder Bandinhalt) sortiert und erst dann nach Knoten oder Element. Die Sortierung erlaubt schließlich das Ausmustern von doppelten Zuständen. Außerdem ist vorgesehen, dass Zustände mit gleichen Daten gruppiert werden. Damit liegen dann Zustände mit gleichem Bandinhalt direkt nebeneinander.

Die Routine `lookAhead_with_StatePos` merkt sich nun für jeden Zustand welche Lookahead-Elemente aus diesem hervorgehen. Die entsprechenden Positionen in der Zustandsmenge und dem Lookahead-Vektor werden in dem Feld `new_data_pos` von `TraceStep` gespeichert. Der Parameter `remember` gibt zudem an, ob sich der Automat für neu entdeckte Endzustände das Ergebnis mittels `rememberOutcome` merken soll. Dies ist nur beim ersten Aufruf von `lookAhead_with_StatePos` für einen bestimmten `TraceStep` geboten.

Es ist nämlich möglich, dass statt `lookAhead_with_StatePos` `lookAhead_simple()` verwendet wurde, wie dies bei der Ausführung ohne Bildschirmdarstellung von Zwischenschritten der Fall ist. In diesem Fall ist das Array `new_data_pos` leer geblieben. Es kann aber durch Aufruf von `init_new_data_pos()` jederzeit im Nachinein befüllt werden, falls die Zugehörigkeit von Objekten der Zustandsmenge zu Lookahead-Objekten doch gebraucht wird.

`init_new_data_pos()` tut im wesentlichen nichts weiteres als den Lookahaed für den aktuellen `TraceStep` zu löschen und den Lookahead mit `lookAhead_with_StatePos` noch einmal neu zu berechnen. Dabei muss `remember` auf `false` gesetzt sein, da alle dann nochmals erkannten Ergebnisse nicht doppelt gespeichert werden sollen.

2.7 auxtypes – Stringtypen und Bildschirmausgabe für die Konsole

Im wesentlichen definiert das Modul `auxtypes` drei verschiedene Klassen von Stringtypen: `xstr_const`, `xstr_shared` und `xstrbuf`. Die Stringtypen sind dabei stets als Templates definiert und können mit einem von drei Zeichentypen arbeiten: `achar`, `echar` und `wchar`. Diese haben eine Zeichenbreite von 8, 16 bzw 32 Bit.

Für den Automatensimulator werden nur `achar` mit 8 Bit und `echar` mit 16 Bit gebraucht. Eine Ausnahme bildet die `inta_const` mit 32 Bit für Wertparameter von Maschinenschemata. Hier werden allerdings die Elemente als Integer und nicht als Unicodezeichen interpretiert.

Die Vorzeichenbehaftung eines `achar` ist mit Absicht unterschiedlich von der eines gewöhnlichen `char` gewählt, da beide Stringtypen bei der Bildschirmausgabe unterschiedlich interpretiert werden. Ein `achar` wird als Latin1-Zeichen interpretiert während ein `char` Teil eines Utf8-Strings ist.

Utf8 ist eine 8 bittige Kodierung von 16 oder 32 Bit breiten Zeichen, die folgendermaßen aussieht: Zuerst kommt ein Zeichen, dessen k hochwertigste Bits 1 sind (Darauf folgt dann eine Null). Danach kommen k-1 Zeichen mit den hochwertigsten Bits "10". Zeichen die eine Null im hochwertigsten Bit tragen sind direkt als Ascii-Zeichen zu interpretieren. Beispiel für ein Utf8-zeichen wäre 1100 0101, 1001 0101 was zu 01 0101 0101 decodiert wird. Die grundlegende Funktionsweise des Utf-8 Codecs zu kennen ist für das Lesen des Sourcecodes von `auxtypes.cpp` unerläßlich.

2.7.1 xstr_const und utf8str_const

Kommen wir nun zu `xtsr_const`, welches nach Template-Instanzierung in drei Varianten auftritt: `astr_const`, `estr_const` und `wstr_const`. Die entsprechenden Klassen haben zwei Felder, welche auch händisch manipuliert werden dürfen: `chars` für die Zeichen und `length` für die Stringlänge. Ein `xstr_const` kann somit auch Nullzeichen enthalten, was für die Simulation von Automaten wichtig war, welche über dem gesamten Eingabealphabet von 256 Zeichen arbeiten können sollen.

Ein Spezifikum der `xstr_const` Methoden `find_first/last_[not_]of`, welche nach einzelnen Zeichen in einem String suchen, ist, dass diese, falls kein solches Zeichen gefunden worden ist, statt -1 die Position des ersten Zeichens außerhalb des Strings in Suchrichtung zurückgeben.

Wenn man also abfragen will, ob ein Zeichen in einem String ist, so kann man `xstr.find_first_of(..)< xstr.length` schreiben. Die zusätzliche Angabe von `xstr.length` mag bei reinem Abfragen, ob ein Zeichen enthalten ist, den Code etwas verlängern. Dafür ergeben sich aber wesentliche Vereinfachungen für den Fall, dass das Ergebnis von `find_...of` mit `sub` weiterverwendet werden soll, welches einen Teilstring extrahiert. Man spart sich damit auf beiden Seiten, also innerhalb der Prozedur und außerhalb, lästige `if`-Abfragen. Wer eine -1 für "nicht gefunden" als Rückgabe haben will, kann ja zudem `xstr.find_last_of(..)` verwenden.

Ein großer Vorteil von `xstr_const` im Gegensatz zu allen anderen bekannten Stringtypen ist, dass Teilstrings ganz einfach ohne Kopieren des ursprünglichen Strings, genommen werden können. Diese bleiben solange gültig wie der Hauptstring, aus dem die Teilstrings extrahiert worden sind.

Die Routinen `find` und `rfind` suchen nach einem mehrbuchstabigem Teilstring. Dabei erzeugen sie einen Hashwert der nur von den k letzten Zeichen abhängt, wobei k die Länge des zu suchenden Teilstrings ist. Die Implementierung richtet sich nach einer Idee von [OtWi12].

Weiters gibt es noch den eigenen Typ `utf8str_const`, welcher auf `char` statt auf `achar` basiert und einen Utf8-String mit Längenangabe wiedergibt.

Schließlich sollte noch die Klase `XStrTraits` erwähnung finden, welche für einen xstr einen leeren String (`XStrTraits::empty_xstr`) und den ungültigen String (`XStrTraits::invalid_xstr`) definiert. Der leere String hat Länge null aber ein gültiges `chars` Feld, währen beim ungültigen String das Feld `chars NULL` ist.

2.7.2 xstrbuf

Für die meisten Zwecke genug haben dürfte man mit der zusätzlichen Templateklasse `xstrbuf`. Diese implementiert wie der Name schon sagt einen Puffer für Strings. Es stehen Methoden wie `append`, `insert`, `remove` und `replace` bereit.

Interessant ist die Speicherverwaltung von xstrbuf. Per default allokiert ein `xstrbuf`, wenn man ihm einen anderen String zuweist, Platz am Heapspeicher. Das ebenfalls öffentlich lesbare Feld `bufParams` enthält dabei die Länge des allokierten Speicherbereichs.

Standardmäßig wird für einen neuen String ein um eins größerer Speicherbereich reserviert als der String lang ist um wie in C üblich einer abschließenden Null Platz zu bieten. Erst bei Anwendung von Methoden wie `append` kommen die Klassenvariablen `alloc_factor` und `alloc_addspace` zum Einsatz um etwas mehr Platz als derzeit benötigt zu schaffen. Einzelne Zeichen können aber immer zusätzlich am Ende Platz finden da Speicher immer mit der Granularität von `xstrbuf::min_alignment` angefordert wird.

Interessant wird es aber wenn wir uns `setUserAllocedBuf` ansehen. Dieses erlaubt die Verwendung eines vom Benutzer allokierten Puffers, welcher beispielsweise auch am Stapel liegen kann. Benutzerallokierte Strings werden erkannt, weil sie in dem höchstwertigstem Bit von `bufParams` eine Eins tragen (`xstrbuf_bufUserAllocatedBit`).

Der Destruktor eines `xstrbuf` gibt den über `bufParam` allozierten Speicher wieder frei. Das heißt wenn ein `xstrbuf` eine Lokalvariable ist, dass dann alle `xstr_const` – Referenzen auf diesen nach Beendigung der Prozedur ihre Gültigkeit verlieren, weil der entsprechende `xstrbuf` nicht mehr vorhanden ist.

Es gibt in `xstrbuf` noch ein paar weiterer Methoden, die interessant sind. Beispielsweise kann man einen `xstrbuf` mit `referenceFrom` eine einfache Referenz auf einen anderen String tragen lassen, ganz so wie dies ein `xstr_const` tut. In diesem Fall enthält `bufParams` die Konstante `xstrbuf_constBuf`, welche im höchstwertigen Bit eine Eins trägt und sonst null ist. Das bedeutet, dass kein Zeichen des Strings verändert werden darf (Null) und dass der Puffer vom Benutzer allokiert wurde.

Die Methode `xstrbuf::moveFrom` übernimmt konträr zu `referenceFrom` den Puffer eines anderen `xstrbuf` und läßt den alten `xstrbuf` als Referenz auf den neuen Puffer zurück; d.h. alle Zeichen sind weiterhin lesbar; allerdings ist jetzt der neue Puffer für den Erhalt des Basisstrings zuständig.

Weniger empfehlenswert als die beiden vorhin besprochenen Methoden welche mit Stringreferenzen arbeiten sind `sharedRefFrom` und `sharedMoveFrom`. Diese lassen es zu, dass auch die entstandene Referenz noch verändert werden darf. Reicht der vorhandene Puffer aber für die geplante Operation nicht mehr aus, so wird einfach ein neuer alloziert, sodass von da an zwei getrennte `xstrbuf`-s weiterexistieren.

```
PDA_Data* PDA_Edge::execPushPop( PDA_Runtime *r, PDA_Data *d, SymbolTable *tmpTransitionSymbols ) const {
  astrbuf actToPush; bool isTemporary = true; int res;
  if( toPushAsConst.chars ) { actToPush.referenceFrom(toPushAsConst); isTemporary = false; }
  else {
    actToPush.setUserAllocedBuf( alloca( sizeof(achar) * ( toPushAsConst.length + 1 ) ), toPushAsConst.
        length + 1 );
    if(( res = SymbolTable::translate( &actToPush, toPush, tmpTransitionSymbols, r->paramSymTab )) < 0 )
        {
      if( res == -3 ) { r->setCondition(Runtime::out_of_memory); return NULL; }
      else r->setCondition( Runtime::undefined_variable_during_next );
    }
  }
  return (PDA_Data*) r->pushPopStack( d, actToPush, toPopAsConst.length, isTemporary );
}
```

Src. 2.11: Die Methode execPushPop von Kellerautomaten

Obiges Sourcecodelisting 2.11 zeigt die Methode `execPushPop` welche für Kellerautomaten einen vorgegebenen String vom Stapel holt und danach `actToPush` auf den Stapel legt. Enthält der zu pushende String keine Variablenreferenzen, so speichert die Instanzvariable `toPushAsConst` diesen direkt. Ansonst muss `toPush` dekodiert werden, wobei `toPushAsConst` immer noch die Länge des dabei entstehenden Zielstrings aufbewahrt.

Ist `toPushAsConst.chars` definiert reicht es für den Wert der Variablen `actToPush` eine Referenz darauf einzurichten. Mehr noch, ist gesichert, dass diese Referenz über die Lebensdauer des Automaten und des Runtime Objektes bestehen bleibt, sodass `isTemporary` auf `false` gesetzt werden darf. Würde man hier eine normale Zuweisung verwenden, so würde ein eigner Puffer alloziert, der allerdings mit Beendigung von `execPushPop` wieder seine Gültigkeit verlieren würde, sodass `isTemporary` `true` bleiben müsste.

Einziger Wermutstropfen bei der sonst sehr effizienten Implementierung von `xstrbuf` ist, dass diese nur schlecht mit den Containern der Standard Template Library zusammenarbeitet. Speichert man beispielsweise einen `xstrbuf` innerhalb eines Vektors (eines Arrays mit variabler Länge) und muss das Array aufgrund einer Größenänderung realloziert werden, so kopiert die STL alle Strings des vorhergehenden Containerinhaltes mit einer einfach Zuweisung in die neue.

Für einen xstrbuf bedeutet dies, dass der String zuerst am Heap kopiert wird und dann bei der Freigabe des alten Containerinhaltes der ursprüngliche Wert zerstört wird. Viel effizienter wäre es hingegen wenn der Container die Strigs aus dem alten Inhalt mit einem `moveFrom` in den neuen mitnimmt.

Hier schneidet der `QString` von Qt besser ab, da dieser eine String-Referenz mit Referenzzähler beinhaltet. Beim Kopieren von dem alten Container in den neuen erhöht sich der Referenzzähler dabei um eins, während er später beim Deallozieren des alten Inhaltes wieder um eins reduziert wird, ohne dass der Inhalt ein einziges mal kopiert werden müsste.

2.7.3 xstr_shared

Ein `xstr_shared` ist ein xstr mit Referenzzähler (`usage_count`). Das macht nur dann Sinn, wenn man Referenzen auf `xstr_shared` hat, anstatt solche Objekte direkt zu deklarieren. Demnach gibt es auch drei statische Methoden, die Pointers auf neu angelegte `xstr_shared` zurückgeben: `allocate_for_len`, `copyFrom` und `dup_sub`.

Während `copyFrom` mit einem `xstr_const` initialisiert werden kann, ist der Aufrufer von `allocate_for_len` dafür verantwortlich den String mit vorgegebener Länge über `sharedstr ->chars` mit sinnvollen Werten zu befüllen, da der Speicherbereich sonst uninitialisiert bleibt. `dup_sub` ist schließlich eine Kombination von `sub` und `copyFrom`, die einen neuen `xstr_shared` als Teilstring eines alten anlegt. Weiters gibt es noch `xstr_shared* dup_replace_char( strsize pos, xchar write_char )`, welches einen neuen `xstr_shared` mit genau einem ersetzten Zeichen anlegt.

Die Klasse dient in erster Linie den Datenobjekten von Turingmaschinen, welche eine Referent auf einen `xstr_shared` sowie eine Bandposition beinhalten. Bewegt sich in einem Schritt nur der Schreib-/Lesekopf, so kann der `xstr_shared` beibehalten werden, während lediglich dessen Referenzzähler mit `duplicate()` um eins erhöht wird.

Weiters wäre die Klasse `xstr_shared` wohl auch für den Stapelinhalt von Kellerautomaten sinnvoll, doch implementieren diese aus "historischen Gründen" ihre eigene Referenz-Stringklasse. Die Klasse `PDA_Data` hat ebenfalls einen Referenzzähler namens `usage_count`, doch kann dieser neben positiven auch negative Werte annehmen. Ein `PDA_Data` enthält einen Header mit `usage_count` und eine Referenz auf den String. Diesem folgt bei positivem `usage_count` direkt der am Stapel liegende String. Bei negativem `usage_count` enthält die Referenz im Header eine Referenz auf einen String außerhalb des `PDA_Data` Objektes (siehe auch `isTemporary` in Listing 2.11).

Ansonsten werden negative Werte gleich den positiven Werten bzw. deren Absolutwert interpretiert. Die zusätzliche Unterstützung von Referenzen auf externe Strings bringt aber wohl keinen ausreichenden Laufzeitvorteil um eine eigene Klasse von Grund auf zu rechtfertifgen. Letztlich werden externe Referenzen nur einmal am Anfang bei der Initialisierung des Stapelinhaltes gebraucht.

2.7.4 Bildschirmausgabe über IOStream

Die Klasse `IOStream` definiert einen Ein- Ausgabestrom, der wahlweise mit einer Datei oder einem `QString` initialisiert werden kann. Bei Initialisierung mit einem QString wird die gesamte Ausgabe nicht an das Terminal weitergeleitet sondern an den `QString` angehängt. Ansonsten ist es aber einfacher und effizienter für die Konvertierung von xstr-s in QStrings und umgekehrt die Prozeduren `toQString` und `fromQString` zu verwenden.

Ein IOStream hat zwei Teile: ein `IOStream_Attr_Rec`, welches aktuelle Einstellungen für die Bildschirmausgabe beinhaltet und ein `IOStream_Base_Rec`, welches das Underlying (ein Datei-Objekt oder ein Qstring), Fehlercodes sowie Zeilnnummer, Spalte und Tokennummer der aktuellen Ausgabe beinhaltet.

Es gibt nun die Klasse `IOStreamAttr`, welche über ihr eigenes `IOStream_Attr_Rec` verfügt aber für `base` das `IOStream_Base_Rec` eines anderen Datenstroms referenziert. `IOStreamAttr` wird mit einem anderen Datenstrom initialisiert und dient dazu Zeichen auf diesem Datenstrom auszugeben.

Werden dabei Ausgabeattribute wie die Zahlenbasis zur Integerausgabe (10 für Dezimalzahlen, 16 für Hexagesimalzahlen) verändert, so bleiben die Änderungen lokal in der `IOStreamAttr` Instanz. Das ist praktisch wenn eine Routine Ausgaben am Bildschirm machen soll, ohne als Seiteneffekt die Attribute des verwendeten Datenstroms dauerhaft zu ändern.

Schließlich gibt es noch die Klasse `IOStreamRef`. Diese hat zwei Zeiger als Datenfelder: `base` für `IOStream_Base_Rec` und `attr` für `IOStream_Attr_Rec`. Dies ist die Basisklasse von der `IOStreamAttr` und `IOStream` abgeleitet werden und kann statt einem Zeiger auf einen `IOStream` verwendet werden. Dies ist auch die empfohlene Vorgehensweise um eine doppelte Dereferenzierung zu vermeiden, i.e. keine Pointer auf IOStreams zu verwenden sondern nur IOStreamRef-s.

Hier ein paar Beispiele zur Bildschirmausgabe mit IOStream:

cout << IOChangeVal(IOAttr::NumberBaseNextOne,16) << (u_iword)hug.chars;

cout << IOChangeFlags(IOFlag::OverlineNextXStr) << my_estr << endl;

`NumberBaseNextOne` ändert im Gegensatz zu `NumberBase` die Zahelenbasis nur für den als nächstes auszugebenden Integerwert. `u_iword` ist ein vorzeichnloser Integerwert, der genau gleich groß ist wie ein Pointer (definiert in basedefines.h zusammen mit `int32`, `uint64`, ...). Vor allem bei der Compilierung mit Windows kann man nicht darauf vertrauen, dass ein `long` gleich groß ist wie ein `void*`.

IOChangeFlags mit `IOFlag::OverlineNextXStr` zeichnet mittels kombinierbaren Unicodezeichen eine Überstreichung des als nächstes auszugebenden xstrs. Andere Flags sind `XStrSpecialSpace`, `XstrVisibleControlChars`, `NoNumberBaseOverride` und `NumberBaseOverride`.

Die Klasse `IOChangeFlags` hat dabei grundsätzlich drei Parameter: `mkTrue`, `mkFalse` und `mkInverted`. Manche Flags wie `NoNumberBaseOverride` können aber für `mkTrue` angegeben werden und schalten den NumberBaseOverride trotzdem aus. Das liegt daran, dass Einstellungen für `mkTrue` eine sog. Maske mit Bits, die gleichzeitig auf Null gesetzt werden, hat. Das ist bspw. praktisch um verschiedene Einstellungen für die Behandlung von Zeilenumbrüchen mit `IOChangeFlags` statt mit `IOChangeVal` zu machen.

Als `NumberBaseOverride` bezeichnet man die einleitenden Zeichen “0x” für Hexagesimalzahlen “0” für Oktalzahlen und “BASIS_” für jede andere Zalenbasis (also bspw 7_13 = 10). `XstrSpecialSpace` schreibt ein sichtbares Zeichen mit gekantetem Unterstrich für Leerzeichen.

Schlußwort

Wir haben gezeigt, dass es möglich und sinnvoll ist einen Simulator für Automaten zu schreiben, der im Benutzerinterface die grafische Definition von Automaten erlaubt und dabei auch nichtdeterministische Automaten simulieren kann. Das in C++ und Qt geschriebene Programm ist plattformunabhängig und kann sowohl unter Windows als auch unter Linux und MacOS übersetzt werden.

Die Einsatzmöglichkeiten für das vorliegende Programm sind dabei durchaus vielschichtig. Das Programm kann für Lehrzwecke, für wissenschaftliche Zwecke, einfach nur zur Modellierung, wenn es um das Design von Programmen geht, verwendet werden oder direkt zum Scannen und Parsen, wenn man berücksichtigt, dass das Programm auch ohne grafischer Oberfläche auf der Konsole übersetzt werden kann.

Während die vorliegende Anwendung zwar die Simulation von nichtdeterministischen Turingmaschinen erlaubt, so befindet sich diese in der Komplexitätsklasse $O(2^n)$, was die Lösung nur für kleine n praktisch gangbar macht, weil für große n Zeit- und Speicherkomplexität explodieren.

Wir dürfen aber gespannt sein, wie sich das Gebiet der theoretischen Informatik weiterentwickeln wird. Mit der Greifbarkeit von Quantencomputern werden damit auch Berechnungsmodelle in den Vordergrund rücken, welche die Lösung von Problemen aus NP in polynomialer Zeit vorantreiben können.

Falls Sie im beiliegenden Programm namens “qcoan” Fehler entdecken, Sie Wünsche oder Anregungen bzgl. des Programmes haben, so senden Sie diese bitte per eMail an estellnb@elstel.org.

Abkürzungsverzeichnis

DFA	deterministic finite automaton
FA	finite automaton
NDTM	non-deterministic Turing machine
NFA	non-deterministic finite automaton
NPDA	non-deterministic pushdown automaton
PDA	pushdown automaton
TM	Turing machine

Literaturverzeichnis

[AhSU99] A. V. Aho, R. Sethi, J. D. Ullmann: Compilerbau. Oldenburg, München, Wien (1999), 2., durchgesehene Auflage. ISBN 3-486-25294-1.

[BlSu11] J. Blanchette, M. Summerfield: C++ GUI Programming with Qt4. Prentice Hall, Upper Saddle River, NJ, Boston, Indianapolis, San Francisco, New York, Toronto, Montreal, London, Munich, Paris, Madrid, Capetown, Sydney, Tokyo, Singapore, Mexico City (2011), Second Edition. ISBN 0-13-235416-0.

[Haus00] R. Hausser: Grundlagen der Computerlinguistik. Springer, Erlangen (2000), Mensch-Maschine-Kommunikation in natürlicher Sprache. ISBN 3-540-67187-0.

[Hors01] P. Horster: Vorlesungsunterlagen aus Theortischer Informatik (2001), Universität Klagenfurt, Informatik - Systemsicherheit, patrick.horster@uni-klu.ac.at.

[KuSc05] S. Kuhlins, M. Schader: Die C++ Standardbibliothek: Einführung und Nachschlagwerk. Springer, Mannheim (2005), 4. Auflage. ISBN 3-540-65052-0.

[Loui96] D. Louis: C und C++: Programmierung und Referenz. Markt & Technik, Haar bei München (1996), Schnell-Übersicht. ISBN 3-8272-5066-8.

[Love13] R. Love: Linux System Programming. O'Reilly Media Inc., Beijing, Cambridge, Farnham, Köln, Sebastopol, Tokyo (2013), Second Edition. ISBN 978-1-449-33953-1.

[MaSa08] N. Matloff, P. J. Salzman: The art of debugging with GDB, DDD and Eclipse. No Starch Press Inc., San Francisco (2008). ISBN 1-59327-002-X.

[Meck04] R. Mecklenburg: Managing Projects with GNU Make. O'Reilly Media Inc. (2004), 3rd Edition. ISBN 978-81-7366-958-3.

[MoAK88] R. N. Moll, M. A. Arbib, A. J. Kfoury: An Introduction to Formal Language Theory. Springer, New York, Berlin, Heidelberg, London, Paris, Tokyo (1988). ISBN 0-387-96698-6, 3-540-96698-6.

[Nagp13] C. K. Nagpal: Formal Languages and Automata Theory. Oxford University Press, Echelon Institute of Technology Faridabad (2013). ISBN 0-19-807106-X.

[OtWi12] T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen. Spektrum Akademischer Verlag, Heidelberg (2012), 5. Auflage. ISBN 978-3-8274-2803-5.

[Soch08] R. Socher: Theoretische Grundlagen der Informatik. Hanser, Emden, Berlin, Deutschland (2008), 3., aktualisierte und erweiterte Auflage. ISBN 978-3-446-41260-6.

[Voß06] H. Voß: LATEX in Naturwissenschaften & Mathematik. Franzis (2006). ISBN 3-7723-7419-0.