

Compilerbau

Aufgabensammlung

1 Lexikalische Analyse

Beispiel 1.1 Konstruieren Sie einen möglichst einfachen deterministischen Automaten für folgende reguläre Ausdrücke:

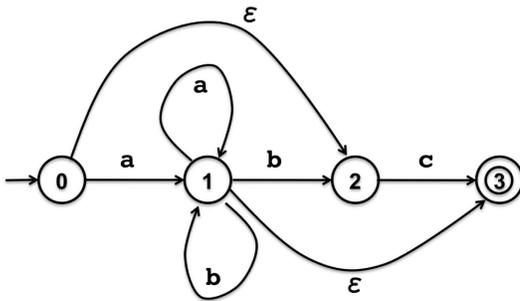
1. $(((a b) | (a c)) (c | a)^+)^* a$
2. $a (a | b)^* ((c (d | \epsilon) a^*) | \epsilon)$
3. $((a b)^* c (b a | \epsilon))^+$

Beispiel 1.2 Angenommen in einer Programmiersprache gibt es Konstanten, die eine Liste von Integerzahlen darstellen können. Diese sind wie folgt definiert:

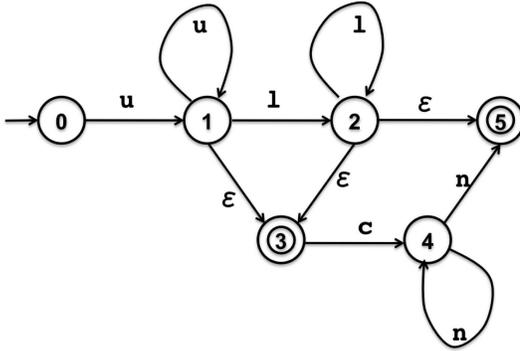
Ein Liste von Integer beginnt mit einer geschwungenen Klammer auf '{' und endet mit einer geschwungenen Klammer zu '}'. Dazwischen können beliebige Integers stehen, die durch Beistriche ',' getrennt sind. Ein Integer in der Sprache besteht aus einer Folge von Ziffern '0' ... '1', die am Anfang und am Ende durch '#' abgeschlossen werden.

1. Beschreiben Sie Integers und Integerliste in der Sprache mittels regulärer Ausdrücke (bzw. regulärer Definitionen).
2. Konstruieren Sie einen äquivalenten deterministischen endlichen Automaten (DFA).

Beispiel 1.3 Wandeln Sie den folgenden nichtdeterministische endlichen Automaten (NFA) mittels Subset Construction Algorithmus in einen deterministischen endlichen Automaten (DFA) um. Geben Sie zusätzlich einen äquivalenten regulären Ausdruck an.



Beispiel 1.4 Wandeln Sie den folgenden nichtdeterministischen endlichen Automaten (NFA) mittels Subset Construction Algorithmus in einen deterministischen endlichen Automaten (DFA) um. Geben Sie zusätzlich einen äquivalenten regulären Ausdruck an.



Beispiel 1.5 Im SML Standard¹ werden *real constants* folgendermaßen beschrieben:

An *integer constant* (in decimal notation) is an optional negation symbol (\sim) followed by a non-empty sequence of decimal digits $0, \dots, 9$. [...] A *real constant* is an integer constant in decimal notation, possibly followed by a point ($.$) and one or more decimal digits, possibly followed by an exponent symbol (E or e) and an integer constant in decimal notation; at least one of the optional parts must occur, hence no integer constant is a real constant. Examples: 0.7 $3.32E5$ $3E\sim7$. Non-examples: 23 $.3$ $4.E5$ $1E2.0$.

- Definieren Sie einen passenden **regulären Ausdruck** für die oben beschriebene *real constant*
- Konstruieren Sie mit Hilfe des in der Vorlesung beschriebenen Algorithmus (*Thompson construction*) den dazugehörigen **nichtdeterministischen endlichen Automaten (NFA)**. Bitte führen Sie auch die Konstruktionsschritte an und führen Sie keine Vereinfachungen durch.
- Wandeln Sie den gefundenen NFA in einen äquivalenten **deterministischen endlichen Automaten (DFA)** um.

Beispiel 1.6 Gegeben sei der folgende reguläre Ausdruck: $(a \mid b)^+ c (b \mid \epsilon)$.

1. Wandeln Sie diesen unter Verwendung der Thompson Construction in einen äquivalenten nicht-deterministischen Automaten (NFA) um.
2. Wandeln Sie den erhaltenen NFA in einen äquivalenten deterministischen Automaten (DFA) um. Verwenden Sie dabei den Subset Construction Algorithmus.

Beispiel 1.7 Gegeben sei der folgende reguläre Ausdruck: $c (a \mid (b c b)^*)^*$.

1. Wandeln Sie diesen unter Verwendung der Thompson Construction in einen äquivalenten nicht-deterministischen Automaten (NFA) um.
2. Wandeln Sie den erhaltenen NFA in einen äquivalenten deterministischen Automaten (DFA) um. Verwenden Sie dabei den Subset Construction Algorithmus.

¹The Definition of Standard ML (revised), by Robin Milner et al., MIT Press 1997

Beispiel 1.8 Gegeben sei folgender regulärer Ausdruck, der das Token *Filename* beschreibt:

filename $\rightarrow [_ /] \text{file} (_ \text{file})^*$

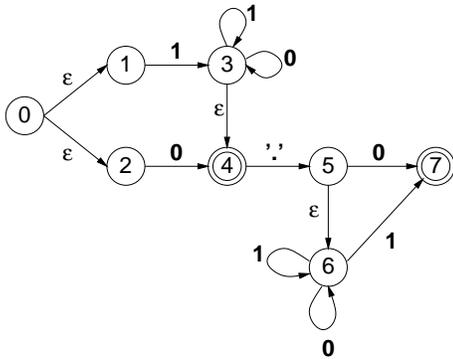
file $\rightarrow (\text{char} + (_ \text{char} +)^*) | (_ [_])$

char $\rightarrow \underline{x}$

D.h. $.. / . / x / x . x$ ist ein gültiger Ausdruck aber $/ . . / . x . x$ ist kein gültiger Filename.

1. Konstruieren Sie einen NFA (Nicht-deterministischen Finiten Automaten) unter Verwendung der Thompson-Construction.
Tipp: Sie müssen nicht den in der Vorlesung gebrachten Algorithmus verwenden. Vermeiden Sie wenn immer möglich ϵ -Übergänge.
2. Wandeln Sie den NFA in einen DFA um. Wenden Sie dabei den Subset Construction Algorithmus an und berechnen Sie die jeweiligen ϵ -closure und move-Operationen.

Beispiel 1.9 Wandeln Sie folgenden NFA (Nicht-deterministischen Finiten Automaten) in einen DFA (Deterministischen Finiten Automaten) um. Benutzen Sie dabei den Sub-set Construction Algorithmus:



Der Startzustand ist der Zustand 0 und die Endzustände sind 4 und 7. Neben ϵ sind die Übergänge mit **0**, **1** und **'.'** (für einen Punkt) bezeichnet.

1. Geben Sie einen regulären Ausdruck an, der die selbe Sprache wie obiger DFA/NFA definiert. Welche Sprache wird durch obige Automaten und den regulären Ausdruck definiert? Geben Sie Beispiele an.
2. Verwenden Sie die Thompson-Konstruktion um aus dem regulären Ausdruck von (b) einen NFA zu erzeugen. Wie unterscheiden sich die beiden NFA?

Beispiel 1.10 Gegeben ist folgender regulärer Ausdruck: $(\underline{a} \underline{b})^* \underline{a} [\underline{b}] .$

1. Verwenden Sie die Thompson Construction um diesen in einen nichtdeterministischen finiten Automaten (NFA) umzuwandeln.
2. Verwenden Sie die Subset Construction um den in (a) erhaltenen NFA in einen äquivalenten deterministischen finiten Automaten (DFA) umzuwandeln.

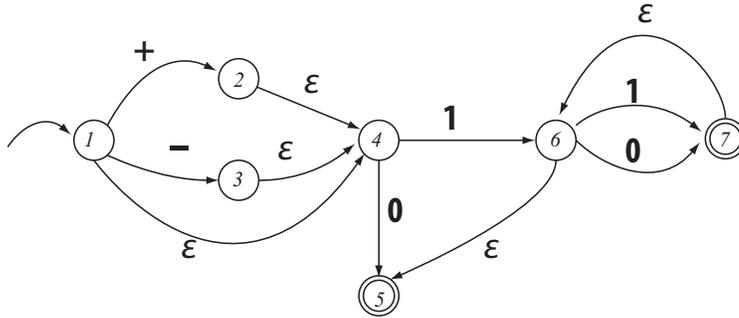
Beispiel 1.11 Gegeben ist folgender regulärer Ausdruck: $(\underline{0}^* \underline{1}) | (\underline{1}^+) .$

1. Verwenden Sie die Thompson Construction um diesen in einen nichtdeterministischen finiten Automaten (NFA) umzuwandeln.
2. Verwenden Sie die Subset Construction um den in (a) erhaltenen NFA in einen äquivalenten deterministischen finiten Automaten (DFA) umzuwandeln.

Beispiel 1.12 Gegeben ist folgender regulärer Ausdruck: $\underline{a} (\underline{b} | \underline{a})^+ (\underline{c} | \epsilon)$.

1. Verwenden Sie die Thompson Construction um diesen in einen nichtdeterministischen finiten Automaten (NFA) umzuwandeln.
2. Verwenden Sie die Subset Construction um den in (a) erhaltenen NFA in einen äquivalenten deterministischen finiten Automaten (DFA) umzuwandeln.

Beispiel 1.13 Gegeben ist folgender nichtdeterministischer finiter Automaten (NFA) mit Startzustand 1 und den Endzuständen 5 und 7.



1. Verwenden Sie die Subset Construction um den NFA in einen äquivalenten deterministischen finiten Automaten (DFA) umzuwandeln.
2. Schreiben Sie einen regulären Ausdruck, für den in (a) erhaltenen DFA, der die selbe Sprache akzeptiert.

Beispiel 1.14 Gegeben ist folgender regulären Ausdruck: $(\underline{a}^* | (\underline{b} \underline{c})^+)^* \underline{\$}$

1. Verwenden Sie die Thompson Construction zur Konstruktion eines äquivalenten nicht-deterministischen finiten Automaten (NFA).
2. Verwenden Sie die Subset Construction um den NFA in einen äquivalenten deterministischen finiten Automaten (DFA) umzuwandeln.

2 Syntaktische Analyse

Beispiel 2.1 Gegeben ist folgender Auszug aus der Grammatik einer Programmiersprache:

$$\begin{aligned}
 S &\rightarrow \textit{identifierList statList} \\
 \textit{identifierList} &\rightarrow \textit{identifier} \mid \textit{identifierList} _ \textit{identifier} \\
 \textit{statList} &\rightarrow \textit{stat} \mid \textit{statList} _ \textit{stat} \\
 \textit{stat} &\rightarrow \textit{identifier} _ \equiv \textit{expression} \mid \\
 &\quad \underline{\textit{if expression then stat else stat}} \mid \\
 &\quad \underline{\textit{if expression then stat}} \\
 \textit{expression} &\rightarrow \textit{identifier} _ \pm \textit{identifier}
 \end{aligned}$$

(bei den *kursiv* gesetzten Bezeichnern handelt es sich um Nicht-Terminals, die Zeichen $_$, \equiv und $\underline{\quad}$ sowie die nicht kursiv gesetzten Bezeichner sind Terminals).

1. Entfernen Sie alle Links-Rekursionen und wenden Sie *left factoring* an. Führen Sie diese zwei Arbeitsschritte in einer geeigneten Reihenfolge durch.
2. Konstruieren Sie einen Recursive Descent Parser für die gegebene Grammatik.
3. Berechnen Sie die LL(1) Tabelle für die gegebene Grammatik.
4. Berechnen Sie die SLR(1) Tabelle für die gegebene Grammatik.

Beispiel 2.2 Gegeben ist folgende Grammatik:

$$\begin{aligned}
 \textit{formula} &\rightarrow \textit{formula} \supseteq \textit{expr} \mid \textit{formula} \equiv \textit{expr} \mid \textit{expr} \\
 \textit{expr} &\rightarrow \textit{expr} _ \vee \textit{term} \mid \textit{term} \\
 \textit{term} &\rightarrow \textit{term} _ \wedge \textit{factor} \mid \textit{factor} \\
 \textit{factor} &\rightarrow \neg \textit{factor} \mid \underline{(\textit{formula})} \mid \perp
 \end{aligned}$$

1. Eliminieren Sie alle *Links Rekursionen* in dieser Grammatik
2. Konstruieren Sie einen Recursive Descent Parser für die gegebene Grammatik.
3. Berechnen Sie die LL(1) Tabelle für die gegebene Grammatik.
4. Berechnen Sie die SLR(1) Tabelle für die gegebene Grammatik.

Beispiel 2.3 Gegeben ist folgende Grammatik:

$$\begin{aligned}
 W &\rightarrow \underline{a}W\underline{a} \\
 W &\rightarrow \underline{b}W\underline{b} \\
 W &\rightarrow \epsilon
 \end{aligned}$$

1. Schreiben Sie einen Predictive Recursive Descent Parser für die gegebene Grammatik.
2. Berechnen Sie die LL(1)-Parser Tabelle für die Grammatik und zeigen Sie durch Verwendung der Tabelle, dass abba ein Wort der durch die Grammatik gegebenen Sprache ist.

Beispiel 2.4 Gegeben ist folgende Grammatik:

$$\begin{aligned} S &\rightarrow \underline{b} W S \\ S &\rightarrow \underline{b} \\ W &\rightarrow W \underline{a} \\ W &\rightarrow \epsilon \end{aligned}$$

1. Berechnen Sie die LL(1)-Parser Tabelle für die Grammatik. Formen Sie diese falls erforderlich um!
2. Berechnen Sie die SLR(1)-Parser Tabelle für die Grammatik. In diesem Fall führen Sie keine Umformungen durch.

Beispiel 2.5 Berechnen Sie die LL(1)-Parser Tabelle für folgende Grammatik. Führen Sie falls erforderlich die notwendigen Umformungen durch:

$$\begin{aligned} C &\rightarrow \underline{/} T \\ C &\rightarrow \underline{/} * TC \underline{/} \\ T &\rightarrow \underline{char} T \\ T &\rightarrow \underline{cr} \\ TC &\rightarrow \underline{char} T \\ TC &\rightarrow \underline{cr} T \\ TC &\rightarrow \epsilon \end{aligned}$$

Beispiel 2.6 Gegeben ist folgende Grammatik:

$$\begin{aligned} L &\rightarrow (\underline{LO}) \\ L &\rightarrow \underline{atom} \\ LO &\rightarrow L LR \\ LO &\rightarrow \epsilon \\ LR &\rightarrow \underline{L} LR \\ LR &\rightarrow \epsilon \end{aligned}$$

1. Berechnen Sie die LL(1)-Parser Tabelle für die Grammatik. Formen Sie diese falls erforderlich um!
2. Berechnen Sie die SLR(1)-Parser Tabelle für die Grammatik. In diesem Fall führen Sie keine Umformungen durch.

Beispiel 2.7 Berechnen Sie die SLR(1)-Parser Tabelle für folgende Grammatik:

$$\begin{aligned} C &\rightarrow \underline{if} P S S \underline{fi} \\ C &\rightarrow \underline{if} P S \underline{fi} \\ S &\rightarrow \underline{C} \\ S &\rightarrow \underline{assign} \\ P &\rightarrow \underline{pred} \end{aligned}$$

Beispiel 2.8 Berechnen Sie die SLR-Parsertabelle für folgende Grammatik. Gibt es einen Shift-Reduce-Konflikt? Wenn ja, kann dieser durch geeignete Annahmen (z.B. Operatoren sind linksassoziativ) aufgelöst werden? Was bedeutet es wenn ein Auflösen des Konflikts nicht möglich ist?

- (1) $P \rightarrow P C$
- (2) $P \rightarrow C$
- (3) $C \rightarrow \underline{comp} (\underline{id} \underline{I} O)$
- (4) $I \rightarrow \underline{id} \underline{I}$
- (5) $I \rightarrow \epsilon$
- (6) $O \rightarrow \underline{id}$

Beispiel 2.9 Berechnen Sie eine LL(1)-Parsertabelle (inklusive der notwendigen FIRST- und FOLLOW-Mengen) für folgende Grammatik.

- (1) $LA \rightarrow A$
- (2) $LA \rightarrow L$
- (3) $A \rightarrow \underline{\text{id}}$
- (4) $L \rightarrow (\underline{LL})$
- (5) $LL \rightarrow LA \underline{LR}$
- (6) $LL \rightarrow \epsilon$
- (7) $LR \rightarrow \underline{,} LA LR$
- (8) $LR \rightarrow \epsilon$

Beispiel 2.10 Berechnen Sie eine LL(1)-Parsertabelle (inklusive der notwendigen FIRST- und FOLLOW-Mengen) für folgende Grammatik.

- (1) $E \rightarrow F$
- (2) $E \rightarrow \underline{\text{id}}$
- (3) $F \rightarrow N (\underline{E})$
- (4) $N \rightarrow E \underline{\&}$
- (5) $N \rightarrow \underline{\text{id}}$

Anmerkung: Falls Grammatiktransformationen dazu notwendig sind, führen Sie diese durch.

Beispiel 2.11 Berechnen Sie eine LL(1)-Parsertabelle (inklusive der notwendigen FIRST- und FOLLOW-Mengen) für folgende Grammatik mit E als Startsymbol.

- (1) $E \rightarrow E \underline{\text{op}} E$
- (2) $E \rightarrow F$
- (3) $E \rightarrow \underline{\text{id}}$
- (4) $F \rightarrow E (\underline{E})$

Anmerkung: Falls Grammatiktransformationen dazu notwendig sind, führen Sie diese durch.

Beispiel 2.12 Schreiben Sie einen LL(1)-Parser für folgende Grammatik. Anmerkung: Führen sie notwendige Umformungen vor der Erstellung der Parsing-Tabelle durch.

$$\begin{aligned}
 S &\rightarrow S \underline{,} S \\
 S &\rightarrow F \\
 S &\rightarrow \underline{\text{id}} R \\
 R &\rightarrow \underline{:} \underline{\text{num}} \\
 R &\rightarrow \epsilon \\
 F &\rightarrow \underline{\text{id}} (\underline{S})
 \end{aligned}$$

Beispiel 2.13 Schreiben Sie eine LR(1)-Tabelle für folgende Grammatik. Geben Sie die Schritte für die Berechnung der Tabelle an!

$$\begin{aligned}
 S &\rightarrow S \underline{,} S \\
 S &\rightarrow \underline{\text{char}}
 \end{aligned}$$

Beispiel 2.14 Berechnen Sie die LL(1)-Parsertabelle für folgende Grammatik. Anmerkung: Führen sie notwendige Umformungen vor der Erstellung der Parsing-Tabelle durch. T bezeichnet das Startsymbol der Grammatik.

$$\begin{aligned}
 T &\rightarrow \underline{\text{@}} (\underline{T \text{ , } T}) \\
 T &\rightarrow \underline{\text{@}} \underline{L} \\
 L &\rightarrow \underline{\text{id}} \\
 L &\rightarrow \underline{\text{num}}
 \end{aligned}$$

Beispiel 2.15 Gegeben sei folgende Grammatik mit Startsymbol S .

$$\begin{aligned} S &\rightarrow S \# S \\ S &\rightarrow S \% S \\ S &\rightarrow \mathbf{char} \end{aligned}$$

1. Berechnen Sie die SLR(1)-Tabelle für die Grammatik. Geben Sie die notwendigen Schritte für die Berechnung der Tabelle an!
2. Ist die Grammatik eine SLR(1)-Grammatik? Wenn dies nicht der Fall ist, lösen Sie die Shift-Reduce-Konflikte auf. Nehmen Sie dabei an, dass $\#$ stärker bindet als $\%$.
3. Zeigen Sie, dass das Wort $\mathbf{char \% char \# char}$ mit der Tabelle aus (b) als korrekt erkannt werden kann.

Beispiel 2.16 Berechnen Sie die LL(1)-Parsertabelle für folgende Grammatik. Anmerkung: Führen sie notwendige Umformungen vor der Erstellung der Parsing-Tabelle durch. E bezeichnet das Startsymbol der Grammatik.

$$\begin{aligned} E &\rightarrow E @ E \\ E &\rightarrow A \\ E &\rightarrow \mathbf{n} \\ E &\rightarrow \mathbf{z} \\ A &\rightarrow \mathbf{n} [E] \end{aligned}$$

Beispiel 2.17 Gegeben sei folgende Grammatik mit Startsymbol S .

$$\begin{aligned} S &\rightarrow S \# S \\ S &\rightarrow S \% S \\ S &\rightarrow ?S \\ S &\rightarrow \mathbf{bn} \end{aligned}$$

1. Berechnen Sie die SLR(1)-Tabelle für die Grammatik. Geben Sie die notwendigen Schritte für die Berechnung der Tabelle an!
2. Ist die Grammatik eine SLR(1)-Grammatik? Wenn dies nicht der Fall ist, lösen Sie die Shift-Reduce-Konflikte auf. Nehmen Sie dabei an, dass $?$ stärker bindet als $\#$ und dieses stärker bindet als $\%$.
3. Zeigen Sie, dass das Wort $\mathbf{bn \% bn \# ? bn}$ mit der Tabelle aus (b) als korrekt erkannt werden kann.

Beispiel 2.18 Berechnen Sie die LL(1)-Parsertabelle für folgende Grammatik. Anmerkung: Führen sie notwendige Umformungen vor der Erstellung der Parsing-Tabelle durch. E bezeichnet das Startsymbol der Grammatik.

$$\begin{aligned} E &\rightarrow E R \\ E &\rightarrow A \\ E &\rightarrow \mathbf{id} \\ R &\rightarrow E \pm \\ R &\rightarrow \epsilon \\ A &\rightarrow \mathbf{id} \# E \end{aligned}$$

Beispiel 2.19 Gegeben sei folgende Grammatik mit Startsymbol S .

$$\begin{aligned} S &\rightarrow S \# S \\ S &\rightarrow S \underline{:} E \\ S &\rightarrow E \\ E &\rightarrow \mathbf{0} \\ E &\rightarrow \underline{\mathbf{1}} \\ E &\rightarrow E E \end{aligned}$$

1. Berechnen Sie die SLR(1)-Tabelle für die Grammatik. Geben Sie die notwendigen Schritte für die Berechnung der Tabelle an!
2. Ist die Grammatik eine SLR(1)-Grammatik? Wenn dies nicht der Fall ist, lösen Sie die Shift-Reduce-Konflikte auf. Nehmen Sie dabei an, dass $\underline{:}$ stärker bindet als $\#$.
3. Zeigen Sie, dass das Wort $\mathbf{0} \# \mathbf{1} \underline{:} \mathbf{1}$ mit der Tabelle aus (b) als korrekt erkannt werden kann.

Beispiel 2.20 Gegeben sei folgende Grammatik mit Startsymbol E .

$$\begin{aligned} E &\rightarrow Z \\ E &\rightarrow Z \underline{--} \\ Z &\rightarrow OS \underline{N} OE \\ OS &\rightarrow \underline{-} \\ OS &\rightarrow \epsilon \\ N &\rightarrow N \underline{\mathbf{0}} \\ N &\rightarrow N \underline{\mathbf{1}} \\ N &\rightarrow \underline{\mathbf{0}} \\ N &\rightarrow \underline{\mathbf{1}} \\ OE &\rightarrow \underline{e} OS N \\ OE &\rightarrow \epsilon \end{aligned}$$

Berechnen Sie die LL(1)-Parsertabelle für die gegebene Grammatik. Anmerkung: Führen sie notwendige Umformungen vor der Erstellung der Parsing-Tabelle durch.

Beispiel 2.21 Gegeben sei folgende Grammatik wobei S das Startsymbol bezeichnet:

$$\begin{aligned} S &\rightarrow L \underline{cr} \\ L &\rightarrow L \underline{|} C \\ L &\rightarrow C \\ C &\rightarrow \underline{cmd} \underline{int} \\ C &\rightarrow \underline{cmd} AL \\ AL &\rightarrow \underline{str} AL \\ AL &\rightarrow \epsilon \end{aligned}$$

Berechnen Sie die LL(1)-Parsertabelle für die gegebene Grammatik. Anmerkung: Führen sie notwendige Umformungen vor der Erstellung der Parsing-Tabelle durch.

Beispiel 2.22 Berechnen Sie die SLR(1)-Tabelle für folgende Grammatik mit Startsymbol L .

$$\begin{aligned} L &\rightarrow L \mid C \\ L &\rightarrow C \\ C &\rightarrow \underline{\mathbf{cmd}} AL \\ AL &\rightarrow \underline{\mathbf{str}} AL \\ AL &\rightarrow \epsilon \end{aligned}$$

1. Ist die Grammatik eine SLR(1)-Grammatik? Wenn dies nicht der Fall ist, lösen Sie die Shift-Reduce-Konflikte unter Annahmen auf.
2. Zeigen Sie, dass das Wort $\mathbf{cmd str str \mid cmd}$ mit der Tabelle aus (b) als korrekt erkannt werden kann.

Beispiel 2.23 Gegeben sei folgende Grammatik wobei P das Startsymbol bezeichnet:

$$\begin{aligned} P &\rightarrow P D \\ P &\rightarrow D \\ D &\rightarrow L \underline{\mathbf{o}} L \\ D &\rightarrow L \\ L &\rightarrow \underline{\mathbf{\$}} L \underline{\mathbf{\$}} \\ L &\rightarrow \underline{\mathbf{n}} \\ L &\rightarrow L \underline{\mathbf{;}} L \end{aligned}$$

Berechnen Sie die LL(1)-Parsertabelle für die gegebene Grammatik. Anmerkung: Führen sie notwendige Umformungen vor der Erstellung der Parsing-Tabelle durch.

Beispiel 2.24 Berechnen Sie die SLR(1)-Tabelle für folgende Grammatik mit Startsymbol P .

$$\begin{aligned} P &\rightarrow E \underline{\mathbf{\%}} \\ E &\rightarrow E \underline{\mathbf{uop}} \\ E &\rightarrow E E \underline{\mathbf{bop}} \\ E &\rightarrow Z \\ Z &\rightarrow Z \underline{\mathbf{1}} \\ Z &\rightarrow \underline{\mathbf{1}} \end{aligned}$$

1. Ist die Grammatik eine SLR(1)-Grammatik? Wenn dies nicht der Fall ist, lösen Sie die Shift-Reduce-Konflikte unter Annahmen auf.
2. Zeigen Sie, dass das Wort $\mathbf{11 1 bop uop \%}$ mit der Tabelle aus (b) als korrekt erkannt werden kann.

3 Attributierte Grammatiken

Beispiel 3.1 Geben Sie eine Attributierte Grammatik an, die folgendes Problem löst. Gegeben ist ein Text in der englischen Sprache der wiederum aus einfachen Sätzen besteht. Jeder Satz besteht aus einem Subjekt, einem Prädikat und einem oder mehrere Objekte (durch Konnektive getrennt). Subjekt, Prädikat und Objekt sind aus den entsprechenden Mengen frei auszuwählen. Das Problem besteht nun darin zu erkennen ob ein Satz auch Semantisch Sinn macht. Der Satz *Twetty is a bird* macht Sinn, der Satz *Bird is a Twetty* jedoch nicht (unter der Annahme, daß *Twetty* keine Unterart bzw. Instanz von *Bird* ist).

$$\begin{aligned} T &\rightarrow S \\ T &\rightarrow S, T \\ S &\rightarrow \text{subj pred } O . \\ S &\rightarrow \text{pred subj } O ? \\ O &\rightarrow \text{obj} \\ O &\rightarrow \text{obj } CO \\ C &\rightarrow \text{and} \mid \text{or} \end{aligned}$$

Zur Vereinfachung nehmen Sie an, daß es eine Funktion $compatible(S, P, O)$ gibt, die **true** zurückliefert, wenn das Subjekt S , Prädikat P und Objekt O zueinander kompatibel sind. Weiters haben die Terminal **subj**, **pred**, **obj** jeweils das Attribut *val*, das den jeweiligen String zurückliefert.

Beispiel 3.2 Gegeben ist eine Wissensbasis, die in Aussagenlogik beschrieben ist. Wandeln Sie die aussagenlogischen Sätze in **if-then-else** Konstrukte um. Definieren Sie eine Attributierte Grammatik zu diesem Zweck. Die Syntax der Aussagenlogik ist wie folgt gegeben:

$$\begin{aligned} L &\rightarrow S \\ L &\rightarrow SL \\ S &\rightarrow A \rightarrow U . \\ A &\rightarrow O \\ A &\rightarrow U \\ O &\rightarrow \text{id} \\ O &\rightarrow \text{id} ; O \\ U &\rightarrow \text{id} \\ U &\rightarrow \text{id} , U \end{aligned}$$

Die Wissensbasis L besteht aus Implikation S , die entweder Und-verknüpfte Aussagen U oder Oder-verknüpfte Aussagen O auf der linken Seite haben. Die rechte Seite der Implikation ist eine Menge von Und-verknüpften Aussagen.

Die Umwandlung erfolgt nach folgenden Regeln:

- Ein aussagenlogischer Satz der Form $P_1, \dots, P_n \rightarrow S$ kann als einfache **if-then-else** Anweisung **if** P_1 **and** \dots **and** P_n **then** S **end**.
- Sätze der Form $P_1; \dots; P_n \rightarrow S$ können als n Sätze der Form $P_1 \rightarrow S \dots P_n \rightarrow S$ dargestellt werden, wobei danach jeder einzelne umgewandelt werden muß.
- Besteht die rechte Seite einer Implikation aus einem Und-verknüpften Satz, so kann dieser ebenfalls vor dem Umwandeln in Teilsätze aufgespalten werden. Aus $A \rightarrow S_1, \dots, S_k$ wird $A \rightarrow S_1 \dots A \rightarrow S_k$. Anmerkung: A kann selber wieder aus mehreren Elementen (Und- oder Oder-verknüpft sein).

Beispiel 3.3 Ein Constraint Satisfaction Problem (kurz CSP) ist durch eine Menge von Constraints gegeben, die Einschränkungen auf Variablen darstellen. Jedes Constraint besteht aus einem Scope, das ist eine Menge von Variablen, die in diesem Constraint verwendet werden, und einer Menge von Wertetupel. Angenommen wir haben die folgende Sprache zur Beschreibung von CSPs:

$$\begin{aligned}
 S &\rightarrow DC \\
 D &\rightarrow L : T . \\
 L &\rightarrow \mathbf{id} \\
 L &\rightarrow \mathbf{id} , L \\
 T &\rightarrow \mathbf{bool} \\
 T &\rightarrow \mathbf{int} \\
 C &\rightarrow \mathbf{constraint id LV end} \\
 V &\rightarrow (VE , V) \\
 V &\rightarrow \epsilon \\
 VE &\rightarrow (VL) \\
 VL &\rightarrow E \\
 VL &\rightarrow E , VL \\
 E &\rightarrow \mathbf{bvalue} \\
 E &\rightarrow \mathbf{num}
 \end{aligned}$$

Damit ein Programm in der obigen Sprache auch korrekt ist, muß folgendes gelten:

1. Bei jedem Constraint **id** muß die Anzahl der Variablen L gleich der Anzahl der Elemente in V sein.
2. Für jede Variable, die an einer Stelle i in L definiert ist, gibt es Werte in den Elementen von V . Diese Werte stehen in jedem Element ebenfalls an der Stelle i . Jeder dieser Werte muß mit dem vorher definierten Typ übereinstimmen.

Beispiel 3.4 Schreiben Sie eine attributierte Grammatik, die Ausdrücke in Postfixnotation in Ausdrücke in Infixnotation umwandelt. Zum Beispiel soll aus $1\ 2\ +\ 4\ *$ der Ausdruck $(1 + 2) * 4$ werden. Definieren Sie zu diesem Zweck eine geeignete Grammatik für Postfixausdrücke.

Beispiel 3.5 Gegeben sei folgende Grammatik mit Startsymbol L , die die Sprache der Listen beschreibt. Schreiben Sie eine attributierte Grammatik, die die Länge einer Liste in der ersten Ebene berechnet. Beispiel: $((...),(...),(...))$ hat Länge 3, unabhängig davon was in den drei Elementen noch vorkommt. Die attributierte Grammatik soll dabei möglichst einfach sein.

$$\begin{aligned}
 L &\rightarrow (\underline{E}) \\
 E &\rightarrow \underline{L R} \\
 E &\rightarrow \epsilon \\
 R &\rightarrow \underline{L R} \\
 R &\rightarrow \epsilon
 \end{aligned}$$

Beispiel 3.6 Gegeben sei folgende Grammatik, die Ausdrücke in der Prefixnotation beschreibt. Schreiben Sie eine attributierte Grammatik, die diese Ausdrücke in eine entsprechende Postfixnotation umwandelt.

$$\begin{aligned}
 E &\rightarrow \underline{num} \\
 E &\rightarrow \underline{id} \\
 E &\rightarrow \underline{bop} E E \\
 E &\rightarrow \underline{uop} E
 \end{aligned}$$

Beispiel 3.7 Gegeben sei folgende Grammatik mit Startsymbol B , die Binärzahlen definiert. Gesucht ist eine attributierte Grammatik, die den Dezimalzahlwert einer Binärzahl berechnet.

$$\begin{aligned}
 B &\rightarrow OP \ BB \ OB \ OE \\
 OP &\rightarrow \pm \\
 OP &\rightarrow \underline{\quad} \\
 OP &\rightarrow \epsilon \\
 BB &\rightarrow \underline{0} \ R \\
 BB &\rightarrow \underline{1} \ R \\
 R &\rightarrow \underline{0} \ R \\
 R &\rightarrow \underline{1} \ R \\
 R &\rightarrow \epsilon \\
 OB &\rightarrow \cdot \ BB \\
 OB &\rightarrow \epsilon \\
 OE &\rightarrow \underline{\quad} \ OP \ BB \\
 OE &\rightarrow \epsilon
 \end{aligned}$$

Beispiel 3.8 Gegeben sei folgende einfache Grammatik:

$$\begin{aligned}
 W &\rightarrow [\ W \] \\
 W &\rightarrow \epsilon
 \end{aligned}$$

1. Schreiben Sie eine attributierte Grammatik, die die Anzahl der Klammernpaare zurückliefert.
2. Berechnen Sie die SLR-Tabelle und zeigen Sie wie man dort die attributierte Grammatik implementieren kann.
3. Implementieren Sie die attributierte Grammatik als rekursive Funktion.
4. Für beide Parserimplementierungen zeigen Sie, wie diese den Wert für den Ausdruck "[[]]" berechnen.

Beispiel 3.9 Erweitern Sie folgenden Grammatik durch Attribute und geben Sie die semantischen Regeln an, die es erlauben einen gegebenen Ausdruck zu evaluieren. Die vorliegende Grammatik beschreibt einen Operator, der auf alle Elemente einer Liste angewandt werden soll. Die Elemente der Liste können selbst wieder Listen (ohne Operator sein). Das Ergebnis soll ebenfalls eine Liste sein. Welche Attribute sind synthetisiert und welche Attribute sind inherited?

$$\begin{aligned}
 E &\rightarrow O \ L \\
 L &\rightarrow (\ LE \ LR \) \\
 LR &\rightarrow \ , \ LE \ LR \\
 LR &\rightarrow \epsilon \\
 LE &\rightarrow \mathbf{num} \\
 LE &\rightarrow L \\
 O &\rightarrow \mathbf{inc} \mid \mathbf{dec}
 \end{aligned}$$

Beispiel: $\mathbf{inc}(1 \ (1 \ 2) \ 3)$ soll $(2 \ (2 \ 3) \ 4)$ liefern.

Beispiel 3.10 Schreiben Sie eine attributierte Grammatik, die Expressions in Assembler-Code umwandelt. Nehmen Sie an, daß die Zielmaschine die folgenden Instruktionen implementiert:

ADD R_1, R_2 $R_1 := R_1 + R_2$
 MUL R_1, R_2 $R_1 := R_1 * R_2$
 MOV M, R $R := \text{memory}(M)$
 LOA N, R $R := N$ (Setzt den Wert des Registers direkt)

Weiters nehmen Sie an, daß die Maschine beliebig viele Register hat und es eine Funktion `nextRegister()` gibt, die den nächsten freien Register zurückliefert.

Die Sprache der Expressions ist durch folgende Grammatik gegeben:

$E \rightarrow (E)$
 $E \rightarrow E + E$
 $E \rightarrow E * E$
 $E \rightarrow \mathbf{id}$
 $E \rightarrow \mathbf{num}$

Zeigen Sie, wie Ihre attributierte Grammatik den Ausdruck

$(1 + (x * 5))$

in Assembler-Code umwandelt.

Beispiel 3.11 Schreiben Sie eine attributierte Grammatik, die Expressions in Assembler-Code umwandelt. Nehmen Sie an, daß die Zielmaschine die folgenden Instruktionen implementiert:

MUL R_1, R_2 $R_1 := R_1 * R_2$
 MOV M, R $R := \text{memory}(M)$
 LOA N, R $R := N$ (Setzt den Wert des Registers direkt)

Dabei bezeichnet R_i einen Register, M eine Stelle im Hauptspeicher und N eine Zahl.

Weiters nehmen Sie an, daß die Maschine beliebig viele Register hat und es eine Funktion `nextRegister()` gibt, die den nächsten freien Register zurückliefert.

Die Sprache der Expressions ist durch folgende Grammatik gegeben:

$E \rightarrow (E E *)$
 $E \rightarrow (E \mathbf{num} **)$
 $E \rightarrow \mathbf{id}$
 $E \rightarrow \mathbf{num}$

Die Multiplikation wird über den Operator `*` angesprochen. Der Operator `**` bezeichnet die Potenzfunktion. $(2\ 3\ **)$ bedeutet 2^3 . Beachten Sie, daß das 2. Argument laut Grammatik immer eine Zahl sein muß.

Beispiel 3.12 Erweitern Sie folgenden Grammatik durch Attribute und geben Sie die semantischen Regeln an, die es erlauben die Ausdrücke in Prefix-Notation in Ausdrücke in Postfix-Notation umzuwandeln wobei Ausdrücke nach ihrer maximalen Tiefe angeordnet werden sollen. Bei gleicher Tiefe ist die Reihenfolge der Terme beizubehalten. Welche Attribute sind synthetisiert und welche Attribute sind inherited?

$E \rightarrow (O E E)$
 $E \rightarrow \underline{\mathbf{id}}$
 $E \rightarrow \underline{\mathbf{num}}$
 $O \rightarrow \pm | =$

Beispiele:

$(+ (+ 1\ 2)\ 3) \Rightarrow (3\ (1\ 2\ +)\ +)$
 $(+ 1\ (+ 2\ 3)) \Rightarrow (1\ (2\ 3\ +)\ +)$

Beispiel 3.13 Schreiben Sie eine Grammatik für Listen, die wie folgt definiert sind. Eine Liste beginnt mit einer Zahl gefolgt von einem Doppelpunkt und der eigentlichen Listendefinition. Eine Liste besteht aus Atomen oder Listen. Jede Liste beginnt mit (und endet mit). Die Zahl am Anfang der Listendefinition gibt die Anzahl der Atome einer Liste wieder. Achtung: Eine Liste kann wieder aus Listen bestehen. D.h. die Atome sind nicht unbedingt in einer Ebene zu finden. Schreiben Sie eine attributierte Grammatik, die diese Listeneigenschaft überprüft. Welche Attribute sind inherited und welche synthesized?

Beispiele: $3 : ((\text{atome}) ((\text{atom}) \text{atom}))$ ist eine gültige Definition. $1 : ()$ ist zwar grammatikalisch richtig aber hinsichtlich der semantischen Regel falsch.

Beispiel 3.14 Schreiben Sie eine attributierte Grammatik, die arithmetische Ausdrücke in Assembler-Code umwandelt. Die Zielmaschine die folgenden Instruktionen implementiert:

```
ADD R1 , R2   R1 := R1 + R2
MOV M , R       R := memory(M)
LOA N , R       R := N (Setzt den Wert des Registers direkt)
```

Dabei bezeichnet R_i einen Register, M eine Stelle im Hauptspeicher und N eine Zahl. Weiters nehmen Sie an, daß die Maschine beliebig viele Register hat und es eine Funktion `nextRegister()` gibt, die den nächsten freien Register zurückliefert.

Die Sprache der Expressions ist durch folgende Grammatik gegeben, wobei P das Startsymbol ist:

```
E → ± E E
E → id
E → num
P → E ; AL
AL → AE ; AL
AL → ε
AE → id := num
```

Ein Programm besteht somit aus einem Ausdruck (E) und einer Liste von Wertzuweisungen (AL) für alle Variablen, die im Ausdruck vorkommen. Beachten Sie, dass die Werte der Variablen zuerst bestimmt werden müssen bevor der Wert eines Ausdrucks bestimmt werden kann. Weiters vergeben Sie sinnvolle Namen für die Attribute und stellen für jedes Attribut fest, ob es sich um ein synthetisiertes oder inherited (vererbtes) Attribut handelt.

Beispiel 3.15 Schreiben Sie eine attributierte Grammatik, die alle Variablen in einem gegebenen Satz durch das entsprechende Wort ersetzt. Behandeln Sie auch Fehler, die auftreten können, wenn eine Variable in einem Satz vorkommt, danach aber nicht definiert wird.

Beispiel: `word id word $ id = word` Je nachdem ob `id` im vorderen oder hinteren Teil des Wortes die selbe Variable bezeichnet, wird es eine Fehlermeldung geben oder auch nicht.

Die Sprache der Sätze ist durch folgende Grammatik gegeben, wobei S das Startsymbol ist:

```
S → E RS $ V
RS → E RS
RS → ε
E → word
E → id
V → id = wordR
R → 2 V
R → ε
```

Ein Satz besteht somit aus einer Abfolge von Wörtern und Variablen (E) gefolgt von einer Liste von Wertzuweisungen (V) für Variablen. Beachten Sie, dass die Werte der Variablen zuerst bestimmt werden müssen bevor diese verwendet werden können. Vergeben Sie sinnvolle Namen für die Attribute. Stellen Sie für jedes Attribut fest, ob es sich um ein synthetisiertes oder inherited (vererbtes) Attribut handelt.

Beispiel 3.16 Gegeben sei folgende einfache Programmiersprache, die nur Sequenz von Assignment Statements als Programm zuläßt:

$$\begin{aligned}
 S &\rightarrow St ; S \\
 S &\rightarrow \epsilon \\
 St &\rightarrow \underline{id} := E \\
 E &\rightarrow \underline{id} \\
 E &\rightarrow (E \underline{op} E) \\
 E &\rightarrow (\underline{uop} E)
 \end{aligned}$$

- (a) Schreiben Sie eine Attributierte Grammatik, die den Datenflußgraphen für solche einfachen Programme zurückliefert. Definieren Sie die notwendigen Attribute und bestimmen Sie deren Typ (inherited bzw. synthetisiert).
- (b) Schreiben Sie eine Attributierte Grammatik, die die Abhängigkeiten zwischen Variablen für jedes Statement berechnet und in einem Attribut speichert. Definieren Sie die notwendigen Attribute und bestimmen Sie deren Typ (inherited bzw. synthetisiert). Anmerkung: In einem Statement $x := y + z$ ist x von y und z abhängig. In diesem Fall wird die Abhängigkeit als Menge von Tupeln der Form $\{ (x, y), (x, z) \}$ dargestellt.

Beispiel 3.17 Gegeben sei folgende einfache Programmiersprache, die die Sprache aus Beispiel 3.16 um If-then-else Statements und While-Statements erweitert.

$$\begin{aligned}
 S &\rightarrow St ; S \\
 S &\rightarrow \epsilon \\
 St &\rightarrow \underline{id} := E \\
 St &\rightarrow \underline{if} E \underline{then} \{ S \} \underline{Oelse} \\
 St &\rightarrow \underline{while} E \underline{do} \{ S \} \\
 Oelse &\rightarrow \underline{else} \{ S \} \\
 Oelse &\rightarrow \epsilon \\
 E &\rightarrow \underline{id} \\
 E &\rightarrow (E \underline{op} E) \\
 E &\rightarrow (\underline{uop} E)
 \end{aligned}$$

Schreiben Sie eine Attributierte Grammatik, die den Kontrollflußgraphen (CFG) für Programme zurückliefert. Definieren Sie die notwendigen Attribute und bestimmen Sie deren Typ (inherited bzw. synthetisiert).

Beispiel 3.18 Gegeben ist folgende Grammatik mit Startsymbol L , das die Sprache der geschachtelten Listen definiert. Die leere Liste wird als **nil** dargestellt. **atom** bezeichnet beliebige Elemente einer Liste, die nicht geteilt werden können.

$$\begin{aligned} L &\rightarrow \mathbf{nil} \\ L &\rightarrow \mathbf{atom} \\ L &\rightarrow [LR] \\ LR &\rightarrow L ; LR \\ LR &\rightarrow L \end{aligned}$$

Schreiben Sie eine attributierte Grammatik, die eine Liste von Atomen zurückliefert, die in einer Liste einer gegebenen Schachtelungstiefe N enthalten sind.

Beispiel:

\mathbf{atom} mit $N = 0$ liefert die Liste \mathbf{atom} .
 \mathbf{atom} mit $N = 1$ liefert die leere Liste.
 \mathbf{atom} mit $N = 1$ liefert die Liste \mathbf{atom} .
 $\mathbf{atom}, \mathbf{atom}, \mathbf{atom}$ mit $N = 1$ liefert die Liste \mathbf{atom} .
 $\mathbf{atom}, \mathbf{atom}, \mathbf{nil}, \mathbf{atom}$ mit $N = 1$ liefert die Liste $\mathbf{atom}, \mathbf{atom}$.
 $\mathbf{nil}, \mathbf{atom}, \mathbf{nil}, \mathbf{nil}$ mit $N = 2$ liefert die Liste \mathbf{atom} .

Vergeben Sie sinnvolle Namen für die Attribute. Stellen Sie für jedes Attribut fest, ob es sich um ein synthetisiertes oder inherited (vererbtes) Attribut handelt.

Beispiel 3.19 Gegeben ist folgende Grammatik mit Startsymbol D , die die Sprache der binären Bäume inklusive eines Typs definiert. Die Wurzelknoten werden als **char** bzw. **int** dargestellt.

$$\begin{aligned} D &\rightarrow TY T \\ TY &\rightarrow \mathbf{char_tree} \\ TY &\rightarrow \mathbf{int_tree} \\ T &\rightarrow \mathbf{char} \\ T &\rightarrow \mathbf{int} \\ T &\rightarrow \mathbf{tree} (T ; T) \end{aligned}$$

Schreiben Sie eine attributierte Grammatik, die die Anzahl der Integer bzw. Character Wurzelknoten abhängig vom Typ zurückliefert.

Beispiele:

$\mathbf{char_tree tree(int, char)}$ liefert 1.
 $\mathbf{int_tree tree(int, char)}$ liefert 1.
 $\mathbf{char_tree tree(int, int)}$ liefert 0.
 $\mathbf{int_tree tree(int, tree(int, char))}$ liefert 2.

Vergeben Sie sinnvolle Namen für die Attribute. Stellen Sie für jedes Attribut fest, ob es sich um ein synthetisiertes oder inherited (vererbtes) Attribut handelt.