Übersetzerbau

WS 2002/2003 Version 0.3

Ruckert

11. Januar 2012

Inhaltsverzeichnis

1 Einführung

Übersetzerbau beschäftigt sich mit der Übersetzung von Programmiersprachen in Maschinencode (klassischer Fall), daneben gibt es in der Praxis viele große und kleine Übersetzungsaufaben bei denen Daten in einer Form (am bequemsten in Text Form) vorliegen und in einer leicht anderen Form gebraucht werden (praktischer Fall). Die Theorie und die Werkzeuge, die für den klassischen Fall entwickelt wurden können Nutzbringend in beiden Fällen verwendet werden.

Lernziele:

→ Was ist Übersetzerbau?
→ Wozu braucht man Übersetzer?
→ Wozu braucht man die Theorie?

1.1 Werkzeuge

Um nur einige Werkzeuge zu nennen: sed, der Stream Editor; lex (flex), für die lexikalische Analyse; yacc (bison), für die Generierung des Parsers; oder awk, für Patterndirected Texttransformationen.

Fürs Backend gibt es eine Vielzahl von Werkzeugen. Etwa die GNU Compiler Collection (gcc), die sich auf viele Programmiersprachen übertragen lässt und viele andere mehr (siehe etwa `www.first.gmd.de/cogent/catalog/`).

Lernziele:

→ Welche Werkzeuge gibt es?
→ Was sind die Vor- und Nachteile einzelner Werkzeuge?

1.2 Phasen eines Übersetzers

Lexikalische Analyse

Zerlegung der Eingabe in "atomare" Einheiten (Token), d.h. kleinste Sinn tragende Einheiten. Entfernung von "bedeutungslosem" Beiwerk (Leerzeichen, Kommentare).

Syntax Analyse

Extraktion der Programmstruktur aus der Folge von Eingabe Tokens (Konstruktion eines Parsetrees) typischerweise gesteuert von einer Grammatik.

Semantische Analyse

Extraktion und Überprüfung von weiteren Eigenschaften der Eingabe, wie z.B. Extraktion von Typinformation. Aufbau eines AST, d.h eine kompakten symbolischen Repräsentation der Eingabe.

Symboltabellen

Neben syntaktischen Eigenschaften, die durch die Zugehörigkeit durch eine Tokenklasse gegeben sind, haben viele Token außerdem noch weitere Semantische Attribute. Zahlen etwa haben einen Wert, Namen haben einen Typ oder sind selbst Typen. Diese Zusatzinformation, die bei der lexikalischen und syntaktischen Analyse anfällt wird für die spätere Verwendung in Symboltabellen festgehalten.

Optimierung

Vor der Erzeugung des Codes werden verschiedene Optimierungen am AST vorgenommen um einen möglichst kompakten und schnellen Code zu erzeugen.

Code Erzeugung

Die Code Erzeugung erfolgt oft in zwei Schritten: zuerst wird ein Zwischencode erzeugt, dieser wird dann weiter optimiert und dann wird der Code spezifisch für die gegebene Zielmaschine erzeugt, gefolgt von Maschinenspezifischen Optimierungen.

Fehlerbehandlung

Fehler treten hauptsächlich in den ersten Phasen: lexikalische, syntaktische und semantische Analyse auf. So ist zum Beispiel "§" kein gültiges Symbol in einem C Programm (lexikalischer Fehler), Die Kombination "int = double;" ist syntaktisch nicht möglich und "const int x; x=0;" ist semantisch nicht korrekt.

Eine gute Fehlerbehandlung ist eine Kunst. Da die genaue Ursache von Fehlern oft nicht aus den Programmtext zu erschließen ist.

Lernziele:

→ Welches Phasen gibt es?
→ Was wird in den einzelne Phasen gemacht?
→ Wozu braucht man eine Symboltabelle?
→ Welche Arten von Fehlern können (in den verschiedenen Phasen) auftreten?

2 Sprachen

2.1 Definitionen

Alphabet: *Eine endliche Menge von Zeichen heißt Alphabet.*

Beispiel: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ oder $\{0, 1\}$ oder $\{+, -\}$ oder $\{\mathit{True}, \mathit{False}\}$ oder die Menge aller ASCII Zeichen.

String: *Eine endliche Folge von Zeichen eines Alphabets heißt ein String über diesem Alphabet.*

Die leere Folge, geschrieben ϵ ist auch eine Folge und somit ein String.

Die Erwähnung des Alphabets lässt man meist weg, wenn aus dem Zusammenhang klar ist um welches Alphabet es sich handelt.

In der mathematischen Theorie spricht man oft auch von einem *Wort* statt einem String.

Sprache: *Eine Menge von Strings über einem Alphabet heißt eine Sprache über diesem Alphabet.*

Es ist bei dieser Definition nicht gesagt, dass es hier irgend eine Bedeutung der Sprache gibt. Die Bedeutung liegt wie so oft im Auge des Betrachters.

Selbst die leere Menge $\{\}$ ist eine Sprache, die sich im übrigen von der Sprache $\{\epsilon\}$, die den leeren String enthält, unterscheidet. Aber nun genug mit den Spitzfindigkeiten.

Während Alphabete und Strings immer endliche sind, sind Sprachen oft auch unendlich. So ist z.B. Die Menge aller binärzahlen Zahlen über dem Alphabet $\{0, 1\}$ unendlich, während die Menge aller Bytes, d.h. die Menge aller genau 8-stelligen binären Zahlen über demselben Alphabet endlich ist (256 Stück).

2.2 Operationen auf Strings

Aneinanderhängen (concatenation): aus den Strings $x =$"Hunde" und $y =$"Hütte" entsteht $xy =$"HundeHütte" (nicht "Hundehütte"). Der leere String ϵ ist das das Neutralelement dieser Operation, wie die 1 bei der Multiplikation. Für einen beliebigen String x gilt nämlich $x\epsilon = \epsilon x = x$. Wenn das nicht sonnenklar ist besteht hier ein Verständnisdefizit und man lese den Abschnitt nochmal.

Wenn das Aneinanderhängen der Multiplikation entspricht, was ist dann die Exponentiation?

Exponentiation: Die Exponentation x^n eines Strings x ist der String, der durch n-maliges aneinanderhängen des Strings x entsteht. Es gilt $x^0 = \epsilon$ für jeden String x.

Ist also $x =$"bla", so ist $x^3 = xxx =$"blablabla". Es ist also genau wie bei der normalen Multiplikation.

2.3 Operationen auf Sprachen

Da Sprachen Mengen sind, gibt es die üblichen Mengenoperationen auf Sprachen, zusätzlich aber gibt es Operationen, die durch Operationen auf den Elementen der Menge, also den Strings definiert sind. Im Einzelnen:

Vereinigung: *Sind A und B Sprachen, dann ist $A \cup B$, die Vereinigung von A und B, wieder eine Sprache. $A \cup B$ enthält alle Strings, die entweder in A oder in B sind.*

Aneinanderhängen: *Sind A und B Sprachen, dann ist AB wieder eine Sprache. AB enthält alle Strings der Form ab wobei a in A ist und b in B.*

Exponentiation: *Ist A eine Sprache und $n \geq 0$ eine ganze Zahl, dann ist A^n wieder eine Sprache. Dabei ist $A^0 = \{\epsilon\}$ und $A^n = AA^{n-1}$ für $n > 0$.*

Strings aus A^n entstehen also dadurch, daß man n Strings aus A hintereinander hängt.

Wiederholung: *Ist A eine Sprache, dann ist A^* wieder eine Sprache. A^* ist die Vereinigung aller A^n für alle $n \geq 0$.*

Strings aus A^* entstehen also dadurch daß man beliebig viele Strings aus A hintereinander hängt.

Klammern: Kombiniert man verschiedene Operationen miteinander so können Unklarheiten entstehen. So könnte AB^2 für A mal B-Quadrat stehen oder für AB zusammen im Quadrat. Wir legen deshalb fest, dass Exponentiation stärker bindet als Aneinanderhängen, und Aneinanderhängen stärker als Vereinigung. Die Operationen Vereinigung und Aneinanderhängen sind links-assoziativ. Im Zweifel, oder um einer andere Kombination zu erzwingen, verwenden wir Klammern.

Erweiterungen: Man kann noch weitere Operationen auf Sprachen definieren, etwa A^+ (für AA^*) oder $A?$ (für $A \cup A^0$). Diese zusätzlichen Operationen lassen sich jedoch durch die schon vorhandenen Operationen ausdrücken und dienen lediglich der bequemeren Schreibweise.

Beispiel: Es sei $A = \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}\}$ die Menge aller einelementigen Strings über dem Alphabet, das aus den Ziffern von `0` bis `9` besteht.

Es sei $B = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}\}$ die Menge aller einelementigen Strings über dem Alphabet, das aus den Großbuchstaben von `A` bis `F` besteht.

Dann gilt:

$A \cup B$ ist die Menge aller hexadezimalen Ziffern.

AB ist die Menge aller zweielementigen Strings, die zuerst eine Ziffer und dann einen Großbuchstaben zwischen `A` und `F` haben.

A^4 ist die Menge aller vierstelligen Zahlen (einschließlich führender Nullen).

$(A \cup B)^2$ ist die Menge aller zweistelligen Hexadezimal-Zahlen.
A^* ist die Menge aller Zahlen (beliebiger Länge).

Lernziele:

→ Kenntnis der Definitionen.
→ Zusammenhang zwischen String- und Sprachoperationen.

3 Reguläre Ausdrücke

Aufbauend auf den vorher definierten Sprachoperationen sind reguläre Ausdrücke ein Formalismus zur Definition von Sprachen. Nicht alle Sprachen lassen sich durch reguläre Ausdrücke definieren. Die Sprachen, die sich so definieren lassen heißen reguläre Sprachen.

3.1 Definition

Wir definieren reguläre Ausdrücke rekursiv. Parallel dazu definieren wir zu jedem Ausdruck r die Sprache $\mathcal{S}(r)$, die vom Ausdruck r dargestellt wird. Wir setzen ein festes aber ansonsten beliebiges Alphabet voraus.

Epsilon: ϵ ist ein regulärer Ausdruck. Die Sprache $\mathcal{S}(\epsilon) = \{\epsilon\}$ enthält nur den leeren String.

Symbole: Ist a ein Element des Alphabets, so ist a auch ein regulärer Ausdruck. Die Sprache $\mathcal{S}(a) = \{a\}$ enthält nur den String der aus dem Symbol a besteht.

Vereinigung: Sind r und s reguläre Ausdrücke, so ist auch $r|s$ (sprich: r oder s) ein regulärer Ausdruck. Die Sprache $\mathcal{S}(r|s) = \mathcal{S}(r) \cup \mathcal{S}(s)$ ist die Vereinigung beider Sprachen.

Aneinanderhängen: Sind r und s reguläre Ausdrücke, so ist auch rs ein regulärer Ausdruck. Die Sprache $\mathcal{S}(rs) = \mathcal{S}(r)\mathcal{S}(s)$ entsteht durch das Aneinanderhängen von Strings der beiden Sprachen.

Wiederholung: Ist r ein regulärer Ausdruck, so ist auch r^* ein regulärer Ausdruck. Die Sprache $\mathcal{S}(r^*) = (\mathcal{S}(r))^*$ entsteht durch beliebiges Wiederholen von Strings aus $\mathcal{S}(r)$.

Klammern: Kombiniert man verschiedene Operationen miteinander so können Unklarheiten entstehen. So könnte $rs|t$ für rs oder t stehen oder für r gefolgt von s oder t. Wir legen deshalb fest, dass Exponentiation stärker bindet als Aneinanderhängen, und Aneinanderhängen stärker als Vereinigung. Die Operationen Vereinigung und Aneinanderhängen sind links-assoziativ. Im Zweifel, oder um einer andere Kombination zu erzwingen, verwenden wir Klammern.

3.2 Erweiterungen

Erweiterungen: Man kann noch weitere Operationen auf regulären Ausdrücken definieren. Diese zusätzlichen Operationen lassen sich jedoch durch die schon vorhandenen Operationen ausdrücken und dienen lediglich der bequemeren Schreibweise. Hier werden nicht alle möglichen Erweiterungen angeführt, die je nach dem verwendeten Programmierwerkzeug auch etwas verschieden ausfallen können, sondern es werden nur die gebräuchlichsten aufgeführt.

Symbolbereiche: Sind $a_1, a_2, \ldots a_i$ Symbole des Alphabets, so ist $[a_1, a_2, \ldots a_i]$ ein regulärer Ausdruck. Er steht für $a_1|a_2|\ldots|a_i$, also für einen einelementigen String bestehend aus einem der Symbole a_1 bis a_i.

Ist das Alphabet geordnet, wie zum Beispiel Buchstaben, Ziffern oder ASCII Codes, kann man zusammenhängende Symbolbereiche einfacher spezifizieren. Man schreibt übersichtlicher $[a_1 - a_i]$ anstatt alle Symbole von a_1 bis a_i explizit aufzulisten.

Bei sehr großen Symbolbereichen ist es oft einfacher anzugeben welche Symbole nicht zum Bereich gehören. Man schreibt $[\hat{}a_1, a_2, \ldots a_i]$ oder $[\hat{}a_1 - a_i]$ anstatt alle Symbole außer a_1 bis a_i einzeln aufzulisten.

Echte Wiederholung: Will man ausdrücken, daß ein regulärer Ausdruck r mindestens einmal wiederholt werden muß schreibt man r^+ statt rr^*, was insbesondere für längere Ausdrücke r übersichtlicher ist.

Optional: Will man ausdrücken, daß ein Ausdruck r optional ist, das heißt, daß man ihn auch weglassen kann, so schreibt man $r?$ statt $\epsilon|r$.

3.3 Transition Diagrams

Ein Transition Diagramm (TD) ist ein gerichteter Graph, d.h. eine Menge von Knoten, die durch gerichtete Kanten miteinander verbunden sein können. Eine Kante kann mit einem Zeichen aus dem Alphabet beschriftet sein. Weiter gibt es einen besonderen Knoten, den Start Knoten und mindestens einen End oder Stop Knoten. Die Knoten sind oft ebenfalls beschriftet, um sie einzeln identifizieren zu können.

Ein Weg durch ein TD ist eine Folge von Kanten, wobei die erste Kante am Start-Knoten beginnt, jede weitere Kante an dem Knoten anfängt an dem die vorige endete und die letzte Kante an einem Stop Knoten endet.

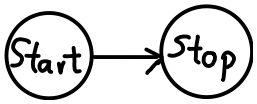
Zu jedem Weg gibt es ein zugeordnetes Wort, nämlich die Folge der Zeichen mit denen die Kanten längs des Weges beschriftet sind. Ist keine dieser Kanten beschriftet erhält man einfach das leere Wort.

Die Menge aller möglichen Wege durch ein TD definiert somit eine Menge von Worten. Diese Menge von Worten ist die Sprache, die durch ein TD definiert ist.

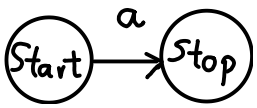
Es ist einfach zu sehen, dass sich jede reguläre Sprache auch durch ein TD beschreiben lässt. Interessant ist, dass auch das umgekehrte gilt: Die Sprachen, die durch TD beschrieben werden sind alle regulär! Wir verzichten hier (leider) auf einen Beweis dieser Tatsache und zeigen lediglich wie jeder reguläre Ausdruck in ein Transition Diagramm übersetzt werden kann. Dies ist auch der eigentliche Grund, warum wir uns mit TD beschäftigen: Es ist bequem eine Sprache durch reguläre Ausdrücke zu spezifizieren aber bei der Implementierung sind TD günstiger.

Da reguläre Ausdrücke induktiv definiert sind kann man die entsprechenden Transition Diagramms auch induktiv definieren:

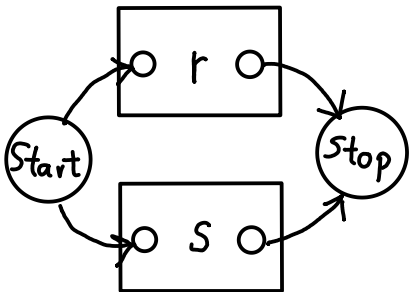
Epsilon: ϵ ist ein regulärer Ausdruck. Das entsprechende Transition Diagram ist:



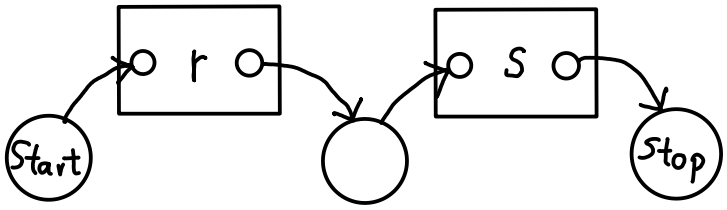
Symbole: Ist a ein Element des Alphabets, so ist a auch ein regulärer Ausdruck. Das entsprechende Transition Diagram ist:



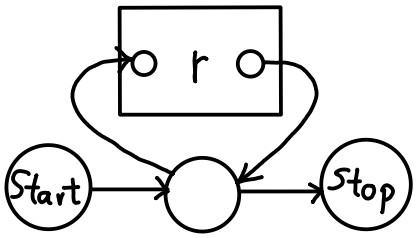
Vereinigung: Sind r und s reguläre Ausdrücke, so ist auch $r|s$ (sprich: r oder s) ein regulärer Ausdruck. Das entsprechende Transition Diagram ist:



Aneinanderhängen: Sind r und s reguläre Ausdrücke, so ist auch rs ein regulärer Ausdruck. Das entsprechende Transition Diagram ist:



Wiederholung: Ist r ein regulärer Ausdruck, so ist auch r^* ein regulärer Ausdruck. Das entsprechende Transition Diagramm ist:



3.4 Endliche Automaten

Eine Darstellung von Sprachen, die der Darstellung durch TD sehr verwandt ist, ist die Darstellung mittels endlicher Automaten.

Ein endlicher Automat wird beschrieben durch ein Eingabealphabet, d.h. eine Menge von möglichen Eingabezeichen, einer Menge von Zuständen, und einer Funktion die abhängig von einem Eingabezeichen und einem Zustand einen Nachfolgezustand angibt, der Übergangsfunktion, oder Maschinenfunktion. Die Maschine hat auch ein Ausgabealphabet, das nur die beiden Zeichen: `Fehler` oder `Akzeptiert` enthält sowie einen besonderen Zustand, den Startzustand und eine Reihe von Ausgezeichneten Zuständen, den Akzeptierenden Zuständen, und einem Fehler-Zustand.

Eine solche Maschine liest einen Eingabestring und spuckt spätestens wenn das Ende des Strings erreicht ist ein Ausgabezeichen aus. Ist dieses Zeichen `Akzeptiert` so gehört der String zur Sprache andernfalls nicht.

Intern funktioniert so eine Maschine so: Die Maschine befindet sich zuerst im Start-Zustand. dann wird jeweils ein Zeichen gelesen und abhängig von diesem Zeichen und dem Zustand der Maschine wird ein neuer Zustand berechnet. Die Maschine nimmt dann den neuen Zustand an. Befindet sich die Maschine im Fehler-Zustand gibt sie das Fehler Zeichen aus und hält. Andernfalls wiederholt sich dieser Vorgang bis das Ende der Eingabe erreicht ist. Befindet sich die Maschine dann in einem Akzeptierenden Zustand, so gibt sie das Akzeptiert-Zeichen aus, andernfalls das Fehler-Zeichen.

Der Zusammenhang zwischen TD und endlichen Automaten ist unmittelbar erkennbar. man kann leicht das eine in das andere übersetzten. Die Zustände entsprechen den Knoten, die Kanten entsprechen der Übergangsfunktion. Zum Hinmalen ist natürlich ein TD schöner, zum Programmieren ist ein endlicher Automat die geeignete Form. Beim Beschreiben einer Spracherkennung ist oft das Bild des endlichen Automaten günstig und man gebraucht eine entsprechende Wortwahl.

Ein endlicher Automat ist natürlich erst dann effizient zu implementieren, wenn die Übergangsfunktion eindeutig ist, wenn es also zu einem Zustand und einem Eingabezeichen höchstens einen Nachfolgezustand gibt — andernfalls muss man nämlich nach dem richtigen Ablauf der Maschine suchen. Ein solcher Automat mit einer eindeutigen Übergangsfunktion heißt deterministisch. Entsprechend heißen die entsprechenden TD's deterministische TD's. Von ihnen handelt der nächste Abschnitt.

3.5 Nichtdeterministische und Deterministische Transition Diagramms

Ein TD heißt deterministisch (DTD), wenn von jedem Knoten aus, entweder nur eine unbeschriftete Kante wegführt, oder wenn alle ausgehenden Kanten mit jeweils verschiedenen Zeichen beschriftet sind. Ist ein TD nicht deterministisch so heißt es Nichtdeterministisches TD (NDTD). Logisch oder?

Deterministische TD erlauben einen ganz einfachen Test, ob ein gegebenes Wort zur Sprache gehört: Man beginnt beim Startknoten und nimmt die Buchstaben des Wortes als eine Wegbeschreibung. Jeder Buchstabe im Wort von links auch rechts gelesen benennt die Kante, die als nächstes den Weg verlängert. Außer im Falle einer nicht beschrifteten Kante, die auf jeden Fall dem Weg hinzugefügt wird. Einen Zweifel um die Auswahl der nächsten Kante kann es nicht geben, die Auswahl ist deterministisch, da es entweder nur eine einzige unbeschriftete Kante gibt, oder aber es gibt zum nächsten Buchstabe höchstens eine einzige Kante.

Gibt es beim zusammenstellen des Weges einen Knoten, der für den nächsten Buchstaben keine Kante enthält, so gehört das Wort nicht zur Sprache. Ist man am Ende des Wortes angelangt und ist der letzte Knoten des Weges kein Endknoten, so gehört das Wort auch nicht zur Sprache. In allen anderen Fällen gehört das Wort nach Definition zur Sprache.

Da DTD so einfach in einen Algorithmus zur Spracherkennung zu übersetzen sind, ist es interessant zu wissen, dass sich jedes NDTD in ein DTD übersetzten lässt. Dies geschieht mit dem sogenannten Teilmengen Verfahren:

Gegeben sei ein TD durch:

- Eine Menge K von Knoten. Wir verwenden die Buchstaben k und l für Knoten dieser Maschine.
- Ein Startknoten, den wir mit `Start` bezeichnen.
- Eine Menge von Endknoten.
- Eine Menge G von gerichteten Kanten. Wir schreiben $k \xrightarrow{a} l$ für die Kante von k nach l mit Beschriftung a.

Wir konstruieren ein neues DTD gegeben durch:

- Eine Menge K' von Knoten. Diese neuen Knoten sind Teilmengen von K. Wir benutzen die Bezeichnung k' und l' für solche Knoten.
- Einen Startknoten.
- Eine Menge von Endknoten.
- Eine Menge G' von gerichteten Kanten. Die Notation entspricht der Notation von Kanten aus G.

Die Mengen K' und G' werden schrittweise aufgebaut. Zu diesem Zweck müssen wir in der Lage sein fertige Knoten von noch unfertigen Knoten zu unterscheiden. Wir markieren einen noch unfertigen Knoten deshalb irgendwie und nehmen diese Markierung weg, wenn er fertig ist.

Um das Verfahren zu verstehen sind vielleicht einige Vorbemerkungen hilfreich:

Das Problem bei den nichtdeterministischen TD ist, dass man oft die Wahl zwischen verschiedenen Kanten hat und nicht weiß welche man wählen soll. Im neuen TD bilden deshalb Mengen von alten Knoten die neuen Knoten. Eine solche Menge $\{k, l, \ldots\}$ von alten Knoten hat anschaulich die Bedeutung: "Im alten TD befände ich mich jetzt in Knoten k oder l oder ...". Man muss sich hier also nicht mehr zwischen verschiedenen Alternativen entscheiden sondern kann alle Alternativen gleichzeitig betrachten.

Verfahren:

1. Wir beginnen mit $K' = \{\}$.
2. Wir nehmen die Knotenmenge {Start}, vervollständigen sie (siehe unten),
3. markieren sie als unfertig,
4. und fügen sie zu K' hinzu. Dieser Knoten ist der Startknoten des neuen TD.
5. Solange es noch unfertige Knoten in K' gibt wiederhole die folgenden Schritte:
   (a) Wähle einen unfertigen Knoten $k' \in K'$ aus,
   (b) bearbeite k' (siehe unten) und
   (c) markiere k' als fertig.
6. Alle neuen Knoten, die einen alten Endknoten enthalten sind neue Endknoten.

Eine Knotenmenge k' wird durch folgendes Verfahren vervollständigt.
Wiederhole die folgenden Schritte solange als möglich:

1. Wähle ein Element k aus der Menge k' aus.
2. Wähle eine unbeschriftete Kante der Form $k \longrightarrow l$ in G aus, so dass l noch nicht Element von k' ist.
3. Füge l zu k' hinzu.

Bei der Vervollständigung handelt es sich also darum, dass man die Menge der erreichbaren Knoten noch um all die Knoten auffüllt, die mittels einer unbeschrifteten Kante erreichbar sind.

Eine Knotenmenge k' wird wie folgt bearbeitet:
Für alle Zeichen a des Alphabets führe die folgenden Schritte aus:

1. Nimm eine leere Knotenmenge $l' = \{\}$.

2. Für alle Elemente k von k' und für alle Kanten der Form $k \xrightarrow{a} l$ aus G füge l zu l' hinzu.

3. Falls l' immer noch leer sind wir hier fertig, ansonsten

4. vervollständige l',

5. markiere l' als unfertig und

6. füge l' zu K' hinzu.

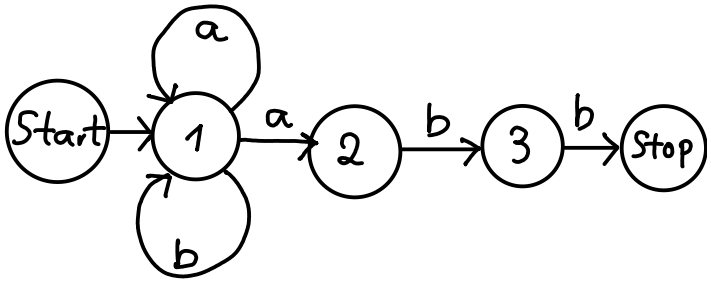
7. Weiter füge die Kante $k' \xrightarrow{a} l'$ zu G' hinzu.

Das heißt also, wir untersuchen für jedes Zeichen a, ob es einen Nachfolgeknoten gibt und wie dieser Aussehen muss. Dazu schauen wir uns alle Knoten aus k', an denen wir sein könnten und bilden die Menge aller Knoten, die wir von einem der möglichen Knoten über eine mit a beschriftete Kante erreichen könnten. Diese Menge wird dann noch vervollständigt und bildet dann den Nachfolgeknoten mit der entsprechend beschrifteten Kante.

3.6 Beispiele

Beispiel 1: Nehmen wir den regulären Ausdruck: $(a|b)^*abb$

Nach obigem Verfahren erhält man daraus zunächst einen NDTD. Man kann dieses Diagramm etwas vereinfachen indem man überflüssige leere Kanten eliminiert und erhält dann:

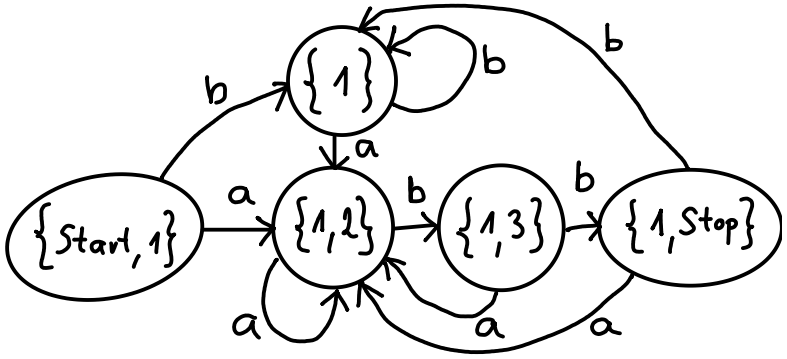


Mit dem Teilmengenverfahren: ergibt sich nun:

- Beginn mit Zustand {Start}.
- {Start} vervollständigt ergibt $\{\text{Start}, 1\}$.
- $\{\text{Start}, 1\}$ wird bearbeitet: Übergang mit a führt zu $\{1, 2\}$. $\{1, 2\}$ vervollständigt ergibt wieder $\{1, 2\}$. Übergang mit b führt zu $\{1\}$. $\{1\}$ vervollständigt ergibt wieder $\{1\}$.
- $\{1, 2\}$ wird bearbeitet: Übergang mit a führt wieder zu $\{1, 2\}$. Übergang mit b führt zu $\{1, 3\}$. $\{1, 3\}$ vervollständigt ergibt wieder $\{1, 3\}$.

- $\{1\}$ wird bearbeitet: Übergang mit a führt zu $\{1, 2\}$. Übergang mit b führt wieder zu $\{1\}$.
- $\{1, 3\}$ wird bearbeitet: Übergang mit a führt zu $\{1, 2\}$. Übergang mit b führt zu $\{1, \text{Stop}\}$. $\{1, \text{Stop}\}$ vervollständigt ergibt wieder $\{1, \text{Stop}\}$.
- $\{1, \text{Stop}\}$ wird bearbeitet: Übergang mit a führt zu $\{1, 2\}$. Übergang mit b führt zu $\{1\}$.
- $\{1, \text{Stop}\}$ ist der einzige Endknoten.

Wir erhalten:



Man überzeugt sich leicht, dass dieses DTD funktioniert. Der Zustand $\{1, 2\}$ wird immer dann erreicht, wenn ein a im Input ist. Zwei darauf folgende b bringen uns zum Knoten $\{1, \text{Stop}\}$, wo dann ein Ende des Input möglich wäre. Ein weiteres b bringt uns zum Knoten $\{1\}$ bei dem man bleibt, bis wieder ein a auftritt.

3.7 Grenzen

Endliches Gedächtnis Über dem Alphabet $\{a, b\}$ betrachten wir die Menge aller Wörter, die erst lauter a und dann genausoviele b enthalten. Diese Menge ist eine Sprache, aber keine reguläre Sprache. Das ergibt sich einfach daraus, dass ein Automat, der eine solche Sprache erkennen wollte ein unendliches Gedächtnis bräuchte um sich die Anzahl der a's zu merken, da es ja beliebig viele a's sein können. Endliche Automaten haben aber nur endlich viele Zustände als ihr Gedächtnis und können sich nicht beliebig viel merken. Also kann diese Sprache nicht regulär sein.

Man sieht ähnlich leicht, dass ganz gewöhnliche Anforderungen an eine Programmiersprache, wie etwa, dass ein Ausdruck gleich viele öffnende und schließende Klammern enthalten muss nicht durch reguläre Ausdrücke beschreibbar sind.

Anzahl der Zustände In der Praxis kommen für effizientes Arbeiten nur deterministische TD in Frage. Der Übergang von Zuständen zu Mengen von Zuständen kann die Anzahl der Zustände drastisch erhöhen. Zu n Zuständen, gibt es 2^n verschiedene Mengen.

Das im Extremfall diese Anzahl von Zuständen auch ausgeschöpft wird zeigt das folgende Beispiel:

Wir nehmen den Regulären Ausdruck $(a|b)^*a(a|b)^n$ für eine beliebige feste natürliche Zahl n. Nach dem Muster der vorigen Beispiels lässt sich dafür ein NDTD konstruieren mit $n+3$ Knoten.

Ein deterministisches TD muss sich einfach die letzten $n+1$ Buchstaben merken, wozu man, anschaulich gesprochen, $n+1$ Bit braucht (nur zwei Buchstaben). Erreicht man das Ende kann man den $n+1$ letzten Buchstaben anschauen und daraus ableiten ob der String zur Sprache gehört oder nicht (dort muss nämlich ein a stehen). Falls man nicht am Ende des Strings ist kann man den $n+1$ letzten Buchstaben vergessen und merkt sich nun den neuen Buchstaben und die n letzten. Aus $n+1$ Bit ergeben sich 2^{n+1} viele Kombinationen. Also braucht ein DTD mindestens 2^{n+1} verschiedene Knoten.

Erfreulicherweise treten solche Fälle in der Praxis eigentlich nicht auf.

Lernziele:

- → Welche Sprachen lassen sich mit Regulären Ausdrücken definieren?
- → Welche Sprachen lassen sich mit Regulären Ausdrücken nicht definieren?
- → Was ist ein Transition Diagramm und welcher Zusammenhang besteht zwischen Regulären Ausdrücken, Transition Diagramms und endlichen Automaten?
- → Was ist ein deterministischer/nicht deterministischer endlicher Automat?
- → Wie definiert ein Automat eine Sprache?
- → Wie werden Reguläre Ausdrücke in nicht deterministische endliche Automaten übersetzt?
- → Wie übersetzt man nicht deterministische Automaten für reguläre Sprachen in deterministische Automaten? Geht das immer?
- → Fertigkeit im Schreiben von Regulären Ausdrücken für einfache Sprachen.

  Beispielaufgabe: Schreibe einen regulären Ausdruck für die Sprache aller Gleitkommazahlen. Zum Beispiel sind “`1.3`” “`1.03`” “`-1.3`” “`+1.3`” “`1.3e3`” “`1.3E-3`” “`-1e+3`” gültige Gleitkommazahlen, aber “`1.3.4`” “`++1.3`” “`1.`” “`.3`” “`1.3e-3.1`” nicht.

4 Kontext Freie Sprachen

Reguläre Ausdrücke haben ihre Vorteile, aber auch ihre Grenzen bei der Sprachbeschreibung. Ein weitaus stärkerer Formalismus zur Beschreibung von Sprachen, der außerdem den typischen Anforderungen bei der Definition von Programmiersprachen gut gerecht wird ist die Definition einer Sprache durch eine kontextfreie Grammatik.

4.1 Definition

Kontextfreie Grammatik Eine Kontextfreie Grammatik ist gegeben durch:

- ein Alphabet, d.h. eine endliche Menge von Symbolen.
- ein ausgezeichnetes Symbol aus dem Alphabet, genannt Startsymbol.
- eine endliche Menge von Regeln. Wobei jede Regel aus einer linken Seite und einer rechten Seite besteht, wobei die linke Seite aus einem einzigen Symbol besteht und die rechte Seite aus einem beliebigen String von Symbolen.

Wir schreiben eine Regel mit linker Seite a und rechter Seite bcd, der Konvention von yacc folgend, als:
$a : bcd ;$

Symbole des Alphabets, die als linke Seite einer Regel vorkommen heißen "Nicht Terminale Symbole", oder "Nonterminals", alle anderen Symbole des Alphabets heißen "Terminale Symbole" oder "Terminals". Diese Bezeichnungsweise versteht man, wenn man sich Herleitungen anschaut. Diese werden wir im nächsten Abschnitt einführen.

Oft lässt man die Angabe des Alphabets ganz weg, da es sich sowieso aus den Regeln ergibt. Man vereinbart auch, dass normalerweise das Symbol auf der linken Seite der ersten Regel das Startsymbol ist und verzichtet damit oft auf die explizite Angabe des Startsymbols.

Herleitung Mittels der Grammatik kann man aus einem String einen neuen String herleiten. Dazu wählt man erst ein Symbol des Strings aus, dann wählt man eine Regel aus, die das betreffende Symbol als linke Seite besitzt und ersetzt schließlich das ausgewählte Symbol im String durch die rechte Seite der Regel. Der so erhaltene String ist aus dem ursprünglichen String mittels der Grammatik hergeleitet.

Dieser Vorgang kann mit dem neuen String wiederholt werden solange der String noch Symbole enthält für die es entsprechende Grammatik Regeln gibt. Eine solche Folge von Strings heißt eine Herleitung.

Symbole, für die es keine Regeln gibt können nicht ersetzt werden und heißen deswegen terminale Symbole.

Kontextfreie Sprache Durch eine Grammatik wird eine Sprache definiert. Dazu betrachtet man alle Herleitungen maximaler Länge, die mit dem Startsymbol beginnen. Also Herleitungen, an deren Anfang das Startsymbol steht und an deren Ende ein String steht, der nur noch terminale Symbole enthält. Die Menge dieser Strings aus terminalen Symbolen, die aus dem Startsymbol abgeleitet werden können ist die Sprache, die durch die Grammatik definiert wird.

Eine Sprache die durch eine kontextfreie Grammatik definiert werden, kann heißt kontextfreie Sprache.

Beispiel: Das Alphabet sei $\{0, 1, s\}$, s sei das Startsymbol, und die Regeln seien:

$$s : ; \qquad \text{und} \qquad s : 0s1 ;$$

Offensichtlich sind 0 und 1 Terminale Symbole und s ein Nicht Terminales Symbol.

Wie sieht die Sprache aus? Betrachten wir dazu eine Herleitung:

Wir beginnen mit s, wählen das Symbol s (welches auch sonst) und dazu die zweite Regel. Wir ersetzen also s durch $0s1$. Wir wählen wieder das Symbol s und wieder die zweite Regel und ersetzen s in $0s1$ durch $0s1$ und erhalten $00s11$. Wir wählen nochmal das Symbol s, aber diesmal die erste Regel und ersetzten s durch den leeren String. Wir erhalten 0011. Weiter geht die Herleitung nicht, denn 0 und 1 sind terminal, d.h. es gibt keine Regeln dafür. Die Herleitung sieht also insgesamt so aus:

s
$0s1$
$00s11$
0011

Man sieht leicht, dass die Sprache, die durch diese Grammatik definiert ist gerade alle Null-Eins Folgen enthält bei denen auf eine Anzahl von Nullen ebensoviele Einsen folgen.

Man bemerke, dass diese Sprache, so einfach sie ist, nicht mittels regulärer Ausdrücke definierbar ist.

Beispiel: Arithmetische Ausdrücke Ein wichtiges Beispiel einer Sprache, die sich durch eine kontextfreie Grammatik beschreiben lässt ist die Sprache der arithmetischen Ausdrücke.

Wir nehmen folgende Regeln (die, der Übersicht wegen, durchnummeriert sind):

1 E: $E + F$
2 E: F
3 F: $F * A$
4 F: A
5 A: n
6 A: (E)

Das Startsymbol ist E, die terminalen Symbole sind $+$, $*$, $($, $)$, und n. Man stelle sich vor, dass n für "number" steht, weiter E für "expression", F für "factor" und A für "atomic".

Eine Herleitung wäre zum Beispiel:

E
$E + F$
$E + F * A$
$E + A * A$
$F + A * A$
$A + A * A$
$n + A * A$
$n + A * n$
$n + n * n$

Der String $n + n * n$ ist also ein arithmetischer Ausdruck. Ähnliche Herleitungen gibt es für $n+n+n$, $((n+n))$ oder $(n+n)*(n+n)$ aber nicht für $(((n+n)$ und $n+*(n+n)$.

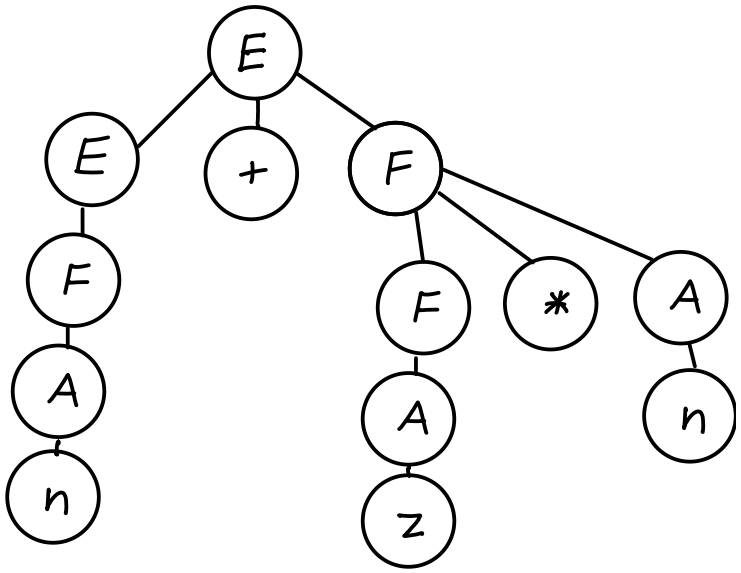
Leftmost/Rightmost Derivation Man beobachtet, dass es für denselben String meist sehr viele Herleitungen gibt, die sich oft aber nur in der Reihenfolge der Regelanwendungen unterscheiden. So ist es bei obiger Herleitung eigentlich gleichgültig, ob ich von $A+A*A$ erst $n+A*A$ und dann $n+A*n$ ableite oder von $A+A*A$ zunächst $A+A*n$ und dann ebenfalls $n+A*n$ ableite. Um dieser Art von Mehrdeutigkeit zu entgehen betrachtet man Standard Herleitungen. Bei einer Leftmost Derivation (Linksableitung) z.B. wird etwa immer das am weitesten links stehende nicht terminale Symbol ersetzt. Bei einer Rightmost Derivation (Rechtsableitung) jeweils das am weitesten rechts stehende nicht terminale Symbol.

Es gilt: hat ein String überhaupt eine Herleitung, dann hat er auch eine Links- und eine Rechtsableitung. Das bedeutet, dass man sich bei der Suche nach Herleitungen auf die eine oder andere Art von Standard Herleitungen beschränken kann. Findet man keine Standard Herleitung, so gibt es überhaupt keine Herleitung.

Die Linksableitung für obiges Beispiel lautet:

$$
\begin{aligned}
E \\
E + F \\
F + F \\
A + F \\
n + F \\
n + F * A \\
n + A * A \\
n + n * A \\
n + n * n
\end{aligned}
$$

Parsetree Eine weitere Möglichkeit die Mehrdeutigkeit zu umgehen, die sich nur aus der Reihenfolge der Regelanwendung ergibt ist die Darstellung einer Herleitung in Form eines Baumes, wobei die Symbole die Knoten des Baumes bilden und ein Knoten und seine Nachfolger gerade eine Regelanwendung darstellen. Das Symbol des Knotens ist die linke Seite der Regel, die Symbole der Nachfolger bilden die rechte Seite der Regel. Ein solcher Baum legt zwar fest welche Regeln auf welche Symbole angewendet werden, lässt aber die Reihenfolge der Regelanwendung offen. Ein solcher Baum heißt parse tree (zu deutsch: Parse Baum). Als Beispiel, der zu obiger Herleitung gehörige parse tree:



4.2 BNF und EBNF

Die Notation für Kontext freie Grammatiken, die wir oben eingeführt haben, fand ihre erste große Anwendung in der Geschichte der Informatik in der Definition der Sprache Algol 60. Zu Ehren der Autoren Backus und Naur wurde diese Notation in der Folge Backus-Naur Form getauft, kurz BNF. Eine erweiterte Form dieser Notation, Extended BNF, oder EBNF, ist auch heute noch in der Definition von Programmiersprachen der allgemein gebräuchliche Standard.

Zur Abkürzung und Verbesserung der Lesbarkeit erlaubt EBNF die folgenden Schreibweisen. Allerdings ist nur die erste dieser Abkürzungen gebräuchlich wenn es um die formale Beschreibung von Grammatiken für einen Parsergenerator geht. Die weiteren Abkürzungen findet man dagegen oft in Definitionen von Programmiersprachen. Hier in diesem Skript wird daher ebenfalls nur die erste dieser Abkürzungen verwendet.

- | für "oder".

  Anstelle der beiden Regeln $A : a_1 a_2 \dots a_n$ und $A : b_1 b2_2 \dots b_m$ schreibt man $A : a_1 a_2 \dots a_n \mid b_1 b2_2 \dots b_m$.

- [ ] Optionaler Bestandteil. (Vorsicht! Eckige Klammern haben bei regulären Ausdrücken eine andere Bedeutung.)

  Anstelle der beiden Regeln $A : a_1 a_2 \dots a_n$ und $A : a_1 a_2 \dots a_n b_1 b2_2 \dots b_m$ schreibt man $A : a_1 a_2 \dots a_n [b_1 b2_2 \dots b_m]$.

- $*$ und $+$ Iteration.

  Der Gebrauch von $*$ und $+$ für Wiederholungen entspricht exakt der Verwendung dieser Symbole bei der Spezifikation von Regulären Ausdrücken.

- ( ) Gliederung durch Klammern.

  Um den Teilstring auf den sich ein $*$ oder $+$ bezieht anzugeben werden gelegentlich Klammern verwendet.

4.3 Grenzen

Wir haben oben gesehen, dass sich die Sprache der symmetrischen 01 Folgen durch eine kontextfreie Grammatik beschreiben lässt: $s : |0s0|1s1$; ist die einzig nötige Regel.

Erstaunlicherweise lässt sich die Sprache der wiederholten 01 Folgen nicht durch eine Kontext freie Sprache beschreiben. Die Sprache der wiederholten 01 Folgen ist dabei definiert als die Menge aller Strings der Form ss wobei s eine beliebige 01 Folge ist. Die Strings 0101 oder 1011110111 gehören also zu dieser Sprache, aber 0110 und 010110 nicht.

Versuchen wir diesem Phänomen auf den Grund zu gehen. Wie könnte denn eine Grammatik für diese Sprache aussehen?

Etwa:
```
s :
  | 11
  | 00
  | 0101
  | 1010
  | 0000
  | 1111
  | ...
```

Hier ist das Problem, dass man nicht endlos so weiter machen kann, da eine Kontext freie Grammatik nur endlich viele Regeln erlaubt. Man braucht also eine rekursive Regel, etwa von der Form: $s : t0t0|t1t1$; Die Idee ist dabei, dass man an zwei gleiche Strings t und t je eine Null oder eine Eins anhängt und dann wieder zwei gleiche Strings erhält. Wie immer man aber die Regeln für t schreibt, man kann nicht verhindern, dass auf die beiden Vorkommen von t im String verschiedene Regeln angewendet werden und somit zwei verschiedene 01 Folgen entstehen. An diesem Punkt scheitert man bei dem Versuch eine passende Grammatik zu schreiben. Daher kommt auch die Bezeichnung "Kontext frei". Die Anwendbarkeit einer Regel hängt eben nur vom Zeichen auf der linken Seite der Regel ab und nicht vom Kontext, in dem das Zeichen innerhalb des Strings steht. Ist eine Regel einmal erlaubt, dann ist sie immer erlaubt.

Betrachtet man die Konsequenzen dieses einfachen Beispiels, dann sieht man, dass es nicht möglich ist mittels einer kontextfreien Grammatik auszudrücken, dass ein Teilstring nur erlaubt ist wenn derselbe Teilstring vorher schon mal vorkam. Übertragen auf Programmiersprachen heißt das etwa, das es z.B. nicht möglich ist mittels der Grammatik auszudrücken, dass eine Variable definiert werden muss bevor man sie verwenden kann. Eine durchaus wichtige, praxisrelevante Forderung, der man aber mittels anderer Mittel nachkommen muss.

Lernziele:

→ Wie ist durch eine Grammatik eine Sprache definiert?
→ Was versteht man unter einer Herleitung eines Strings?
→ Was ist eine leftmost/rightmost Derivation?
→ Was ist ein Parsetree?
→ Welcher Zusammenhang besteht zwischen Parsetree und Herleitung?
→ Was ist eine Kontext freie Sprache?
→ Was ist BNF, welche Erweiterungen bietet EBNF gegenüber BNF und wie kann man solche Erweiterungen auf einfache BNF zurückführen?
→ Welche Sprachen lassen sich mit Kontext freien Grammatiken definieren?
→ Welche Sprachen lassen sich mit Kontext freien Grammatiken nicht definieren?
→ Fertigkeit im Schreiben von Kontext freien Grammatiken in BNF Form für einfache Sprachen.

  Beispielaufgabe: Schreiben Sie eine Grammatik für Listen von Zahlen, die durch Kommata getrennt sind. Zum Beispiel sind "`1,2,3`" oder "`1`" oder ϵ (leere Liste) gültige Listen von Zahlen, aber "`1,,2`" oder "`1,2 3`" oder "`1,2,3,`" nicht. Gehen Sie von den terminalen Symbolen *Zahl* und *Komma* aus.

5 Parsing

5.1 Top-Down

Noch erklären: LL(1), predictive, topdown, recursive descent.

5.1.1 Definition der Mengen First und Follow

Für ein (nonterminales) Symbol A ist First(A) eine Menge, die neben terminalen Symbolen, auch noch den leeren String ϵ enthalten kann. Anschaulich gesprochen ist First(A) die Menge aller Symbole mit denen ein aus A hergeleiteter String anfangen kann. ϵ als Element von First(A) bedeutet, dass sich aus A auch der leere String herleiten lässt.

Definition:

- Ein terminales Symbol a ist Element von First(A) genau dann, wenn es einen String β gibt so dass sich aus A der String $a\beta$ herleiten lässt.
- Der leere String ϵ ist Element von First(A) genau dann, wenn sich aus A der leere String herleiten lässt.

Für ein (nonterminales) Symbol A ist Follow(A) eine Menge von terminalen Symbolen. Anschaulich gesprochen enthält Follow(A) alle terminalen Symbole, die in einem Wort nach einem A auftreten können.

Definition: Ein terminales Symbol a ist ein Element von Follow(A), genau dann, wenn es Strings α und β und eine Herleitung von $\alpha A a \beta$ aus dem Startsymbol gibt.

5.1.2 Konstruktion der Mengen First und Follow

Beide Mengen kann man durch ein iteratives Verfahren bestimmen.

Die Konstruktion von First: Für alle Symbole X werden die Mengen First(X) gemeinsam berechnet. Dazu wendet man die folgenden Regeln solange an, bis sich an den Mengen First(X) keine Änderungen mehr ergeben.

1. Ist X ein terminales Symbol, so ist First(X)=$\{X\}$.
2. Ist X ein Nonterminal und gibt es eine Regel $X : \epsilon$ so füge ϵ zu First(X) hinzu.
3. Ist X ein Nonterminal und gibt es eine Regel $X : A_1 A_2 \cdots A_n$, so füge alle Elemente von First($A_1 A_2 \cdots A_n$) zu First(X) hinzu.

Man bemerke, dass im letzten Schritt First auf einen String $A_1 A_2 \cdots A_n$ angewendet wurde und nicht auf ein Symbol. Wir müssen dies also noch definieren.

Definition: First($A_1 A_2 \cdots A_n$) ist First(A_1) falls ϵ kein Element von First(A_1) ist, andernfalls ist First($A_1 A_2 \cdots A_n$) = (First(A_1) $\setminus$ $\{\epsilon\}$) $\cup$ First($A_2 \cdots A_n$).

Die Konstruktion von Follow: Für alle nonterminalen Symbole X werden die Mengen Follow(X) gemeinsam berechnet. Man berechnet Follow(X), indem man die folgenden Regeln solange anwendet, bis sich an den Follow Mengen keine Änderungen mehr ergeben.

1. Füge $\dashv$ zu Follow(S) hinzu, wobei S das Startsymbol ist und $\dashv$ das Symbol für das Ende des Inputs.
2. Für jede Regel der Form $A : \alpha B \beta$, mit $\beta \neq \epsilon$, füge alle Elemente von First(β), mit Ausname von ϵ, zu Follow(B) hinzu.
3. Für jede Regel der Form $A : \alpha B$ füge alle Elemente von Follow(A) zu Follow(B) hinzu.
4. Für jede Regel der Form $A : \alpha B \beta$, wobei ϵ ein Element von First(β) ist, füge alle Elemente von Follow(A) zu Follow(B) hinzu.

5.2 Bottom-Up Parsing/Handle Pruning

Das gebräuchlichste Verfahren für das Bottom-Up Parsing ist parsing durch "handle pruning". Dies funktioniert so: Wie nehmen an, wir haben eine Folge von Eingabesymbolen für die es eine Herleitung aus dem Startsymbol gibt. Diese Herleitung gilt es zu finden.

Wir benutzen wieder obiges Beispiel und nehmen den String "n+n*n". Wir nehmen die Rechtsherleitung (warum wird später noch klar), die dann so aussieht

$$
\begin{aligned}
E \\
E + F \\
E + F * A \\
E + F * n \\
E + A * n \\
E + n * n \\
F + n * n \\
A + n * n \\
n + n * n
\end{aligned}
$$

Betrachten wir die Herleitung in umgekehrter Reihenfolge, so sieht man, dass immer ein Teilstring, der gerade die rechte Seite einer Regel darstellt durch die linke Seite dieser Regel ersetzt wird. So einen Teilstring, zusammen mit der Regel nennt man ein "handle", das Ersetzen des Handles durch das entsprechende Symbol "handle pruning" (prune=zuschneiden). Da wir eine Rechtsherleitung suchen, genügt es immer das erste (von links nach rechts) Handle im String zu bestimmen und zu ersetzen. Dieses Vorgehen heißt dann: "parsing durch handle pruning"; da man dabei die Herleitung, bzw den Parsetree von unten nach oben konstruiert auch: "bottom up parsing".

In unserem Beispiel beginnen wir mit $n+n*n$ und identifizieren das erste n als Handle: $\underline{n} + n * n$. (Im Folgenden markieren wir stets das handle durch unterstreichen.) Die Regel dazu lautet 5: $A : n$: wir ersetzen also $\underline{n}$ durch A und erhalten $A + n * n$. Nun lautet das handle $\underline{A}$ mit Regel 4 und handle pruning bringt uns von $\underline{A} + n * n$ zu $F + n * n$. Bisher war das recht einfach, denn zu n gibt es nur eine Regel, nämlich Regel 5 und da in keiner weiteren Regel ein n vorkommt, war klar, dass unser erstes handle $\underline{n}$ ist mit Regel 5. Beim zweiten handle pruning, betrachtet man $F+$ und stellt fest, dass es keine Regel gibt, die ein $F+$ enthält, wenn man also irgend etwas mit dem F machen will, so muss man Regel 4 nehmen, und schon hat man wieder das handle gefunden. Jetzt haben wir $F + n * n$ und es ist ebenso klar, dass wir mit Regel 2 von $\underline{F} + n * n$ zu $E + n * n$ kommen, dann mit Regel 5 von $E + \underline{n} * n$ zu $E + A * n$ und mit Regel 4 von $E + \underline{A} * n$ $E + F * n$.

Hier wird es erstmals schwierig, wir können nämlich mit Regel 1 von $\underline{E + F} * n$ zu $E * n$ oder mit Regel 5 $E + F * \underline{n}$ zu $E + F * A$. Die Beobachtung, dass es keine Regel gibt, die $E*$ enthält lässt uns vermuten, dass es besser ist die zweite Möglichkeit zu wählen. Dann geht es mit Regel 3 von $E + \underline{F * A}$ zu $E + F$ und zuletzt mit Regel 1 von $\underline{E + F}$ zu E. Fertig.

Das ist also soweit ganz einfach, man muss nur irgendwie wissen, wo das handle ist. Bevor wir uns daran machen, zuerst noch die exakte Definition von handle und handle pruning.

Definition: Ein handle ist ein Substring, zusammen mit einer Regel, so dass der Substring gerade die rechte Seite der Regel ist und die Ersetzung der rechten Regelseite durch die linke Seite gerade ein Schritt in der Herleitung des ganzen Strings aus dem Startsymbol.

Definition: Handle pruning ist das Ersetzen eines handles durch das Symbol auf der linken Seite der zugehörigen Regel.

Parsing durch Handle pruning ist also nichts anderes, als fortgesetztes handle pruning, beginnend mit dem gegebenen Eingabestring, solange bis nur noch das Startsymbol übrig bleibt.

5.3 Shift-Reduce Parsing

Das Finden eines Handles, und damit der Herleitung lässt sich vereinfachen, wenn man sich auf das Finden einer Rechtsherleitung beschränkt. Gibt es überhaupt eine Herleitung, so gibt es ja auch immer eine Rechtsherleitung. Bei einer Rechtsherleitung wird, wie man oben sieht, immer das erste Handle von links identifiziert und durch handle pruning ersetzt.

Shift-Reduce Parsing vereinfacht die Entscheidungsfindung noch weiter. Es arbeitet mit einem Parsestack auf den mittels einer Push-Operation das nächste Symbol aus dem Eingabestring abgelegt werden kann (Shift) und mittels mehrere Pop-Operationen die rechte Seite der Regel oben vom Stack herunter genommen wird und mittels einer Push-Operation durch das Symbol auf der linken Seite der Regel ersetzt wird (Reduce) Es bleibt also nur noch die Entscheidung, ob eine Shift oder Reduce Operation erfolgen soll.

Mit einem Parse Stack sieht das obige Beispiel so aus:

Parse Stack	Input	Aktion
	$n + n * n$	
n	$+n * n$	Shift
A	$+n * n$	Reduce mit Regel 5
F	$+n * n$	Reduce mit Regel 4
E	$+n * n$	Reduce mit Regel 2
$E+$	$n * n$	Shift
$E + n$	$*n$	Shift
$E + A$	$*n$	Reduce mit Regel 5
$E + F$	$*n$	Reduce mit Regel 4
$E + F*$	n	Shift
$E + F * n$		Shift
$E + F * A$		Reduce mit Regel 5
$E + F$		Reduce mit Regel 3
E		Reduce mit Regel 1

5.3.1 LR(k)

Beim LR(k) parsing wird die Entscheidung, ob Reduce oder Shift, von einem Endlichen Automaten aufgrund der nächsten k Eingabezeichen, des sogenannten "Lookahead", ent-

schieden. Da in der Praxis nur der Wert $k = 1$, also ein einzelnes Zeichen look ahead verwendet wird beschränken wir uns auch hier darauf. Des weiteren legt dieser Automat seinen Zustand auf dem Parsestack ab. Die Funktionsweise des Automaten kann damit durch eine Tabelle seiner Aktionen und Zustandsübergänge beschrieben werden.

Für die obige Grammatik erhält man folgende Tabelle:

	Aktion						Goto		
Zustand	n	$+$	$*$	$($	$)$	$\dashv$	E	F	A
0	s5			s4			1	2	3
1		s6				Accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	4
5		r5	r5		r5	r5			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r6	r6		r6	r6			

Ein Eintrag si steht für die Aktion "Shift mit neuem Zustand i. Ein Eintrag ri steht für die Aktion "Reduce mit Regel i". Das Zeichen $\dashv$ steht für das Ende der Eingabe (EOF). Fehlende Einträge bedeuten einen Übergang in den Fehler Zustand (Error).

Der Parsing Algorithmus sieht nun so aus:

1. Initialisiere den Zustand durch den Start-Zustand (0).
2. Initialisiere den Parsestack mit einem einzigen Eintrag, der der Zustand enthält.
3. Initialisiere das Lookahead Token durch das erste Token der Eingabe.
4. Wiederhole solange bis der Zustand entweder "Accept" oder "Error" ist.
   (a) Lese Tabelleneintrag (Aktion) für den gegebenen Zustand und das gegebene Lookahead Token.
   (b) Ist der Tabelleneintrag si (Shift), so wird der Zustand auf i gesetzt und das Lookahead Token zusammen mit dem neuen Zustand i auf den Parse Stack geschrieben (Push). Ein neues Lookahead Token wird vom Input gelesen.
   (c) Ist der Tabelleneintrag ri (Reduce), so entsprechen die Zeichen oben auf dem Parsestack gerade der rechten Seite der Regel i.
      i. Es werden nun n Einträge vom Parsestack heruntergenommen (Pop), wobei n die Länge der rechten Seite der Regel i ist.

ii. Der Zustand der nun oben auf dem Parsestack liegt wird zum aktuellen Zustand.

iii. Lese den Tabelleneintrag k für den aktuellen Zustand und das Symbol auf der linken Seite von Regel i.

iv. Setze den aktuellen Zustand auf k und speichere k und das Symbol von der linken Seite der Regel i auf dem Parse Stack (Push).

Wir benutzen den obigen Algorithmus und die gegebene Tabelle um uns das Beispiel $n + n * n$ nochmal genau anzuschauen.

Parse Stack	Lookahead	Input	Aktion
0	n	$+n*n\dashv$	Initialisierung
0	n	$+n*n\dashv$	Tabelle$[0,n]$ = s5
$0\ 5/n$	$+$	$n*n\dashv$	Shift
$0\ 5/n$	$+$	$n*n\dashv$	Tabelle$[5,+]$ = r5, Regel 5 ist $A:n$
0	$+$	$n*n\dashv$	Pop
0	$+$	$n*n\dashv$	Tabelle$[0,A]=3$
$0\ 3/A$	$+$	$n*n\dashv$	Push
$0\ 3/A$	$+$	$n*n\dashv$	Tabelle$[3,+]$ = r4, Regel 4 ist $F:A$
0	$+$	$n*n\dashv$	Pop
0	$+$	$n*n\dashv$	Tabelle$[0,F]=2$
$0\ 2/F$	$+$	$n*n\dashv$	Push
$0\ 2/F$	$+$	$n*n\dashv$	Tabelle$[2,+]$ = r2, Regel 2 ist $E:F$
0	$+$	$n*n\dashv$	Pop
0	$+$	$n*n\dashv$	Tabelle$[0,E]=1$
$0\ 1/E$	$+$	$n*n\dashv$	Push
$0\ 1/E$	$+$	$n*n\dashv$	Tabelle$[1,+]$ = s6
$0\ 1/E\ 6/+$	n	$*n\dashv$	Shift
$0\ 1/E\ 6/+$	n	$*n\dashv$	Tabelle$[6,n]$ = s5
$0\ 1/E\ 6/+\ 5/n$	$*$	$n\dashv$	Shift
$0\ 1/E\ 6/+\ 5/n$	$*$	$n\dashv$	Tabelle$[5,*]$ = r5, Regel 5 ist $A:n$
$0\ 1/E\ 6/+$	$*$	$n\dashv$	Pop
$0\ 1/E\ 6/+$	$*$	$n\dashv$	Tabelle$[6,A]=3$
$0\ 1/E\ 6/+\ 3/A$	$*$	$n\dashv$	Push
$0\ 1/E\ 6/+\ 3/A$	$*$	$n\dashv$	Tabelle$[3,*]$ = r4, Regel 4 ist $F:A$
$0\ 1/E\ 6/+$	$*$	$n\dashv$	Pop
$0\ 1/E\ 6/+$	$*$	$n\dashv$	Tabelle$[6,F]=9$
$0\ 1/E\ 6/+\ 9/F$	$*$	$n\dashv$	Push
$0\ 1/E\ 6/+\ 9/F$	$*$	$n\dashv$	Tabelle$[9,*]$ = s7
$0\ 1/E\ 6/+\ 9/F\ 7/*$	n	$\dashv$	Shift
$0\ 1/E\ 6/+\ 9/F\ 7/*$	n	$\dashv$	Tabelle$[7,n]$ = s5
$0\ 1/E\ 6/+\ 9/F\ 7/*\ 5/n$	$\dashv$		Shift
$0\ 1/E\ 6/+\ 9/F\ 7/*\ 5/n$	$\dashv$		Tabelle$[5,\dashv]$ = r5, Regel 5 ist $A:n$
$0\ 1/E\ 6/+\ 9/F\ 7/*$	$\dashv$		Pop
$0\ 1/E\ 6/+\ 9/F\ 7/*$	$\dashv$		Tabelle$[7,A]=10$
$0\ 1/E\ 6/+\ 9/F\ 7/*\ 10/A$	$\dashv$		Push
$0\ 1/E\ 6/+\ 9/F\ 7/*\ 10/A$	$\dashv$		Tabelle$[10,\dashv]$ = r3, Regel 3 ist $F:F*A$
$0\ 1/E\ 6/+$	$\dashv$		Pop, Pop, Pop
$0\ 1/E\ 6/+$	$\dashv$		Tabelle$[6,F]=9$
$0\ 1/E\ 6/+\ 9/F$	$\dashv$		Push
$0\ 1/E\ 6/+\ 9/F$	$\dashv$		Tabelle$[9,\dashv]$ = r1, Regel 1 ist $E:E+F$
0	$\dashv$		Pop, Pop, Pop
0	$\dashv$		Tabelle$[0,E]=1$
$0\ 1/E$	$\dashv$		Push
$0\ 1/E$	$\dashv$		Tabelle$[1,\dashv]$ = Accept

5.3.2 SLR

5.3.3 Konflikte

Lernziele:

→ Wie funktioniert Shift-Reduce Parsing?
→ Was ist der Unterschied zwischen SLR und LR(k)?
→ Gibt es Grammatiken, die man Top-Down aber nicht Bottom-Up parsen kann?
→ Gibt es Grammatiken, die man Bottom-Up aber nicht Top-Down parsen kann?
→ Was ist ein Shift-Reduce Konflikt?
→ Was ist ein Reduce-Reduce Konflikt?
→ Wie kann man Konflikte beheben?
→ Was ist der Zusammenhang zwischen regulären Sprachen und kontextfreien Sprachen?
→ Gibt es für jede Kontext freie Sprache eine LR(k) Grammatik?
→ Gibt es zu einer Sprache eine, höchstens eine, oder mehrere Grammatiken?
→ Wie funktioniert Shift-Reduce parsing?
→ Wie erstellt man Parse-Tabellen für SLR Grammatiken?
→ Was sind "Punktierte Regeln"?

6 Code Erzeugung

6.1 AST

6.2 Intermediate Code

6.3 Optimierung

Constant Propagation

Common Subexpression Elimination

Copy Propagation

Dead Code Elimination

Algebraic Simplification

Strength Reduction

Basic Blocks

Dead Variables Elimination

Alias Analysis

Constant Code Movement

Induction Variables

Loophole Optimization

Lernziele:

→ Was ist ein AST?
→ Wozu braucht man intermediate Code?
→ Was ist Alias Analysis?
→ Was ist ein Basic Block?
→ Welche Optimierungen gibt es? Erläutere sie jeweils durch ein Beispiel.

7 Aufgaben

Die folgenden Aufgaben entstammen der Klausur im Wintersemester 2002.

Aufgabe 1 (16 Punkte): Erklären Sie die folgenden Begriffe jeweils anhand eines Beispiels.

a) Common Subexpression Elimination

b) Strengt Reduction

c) Basic Block

d) Induction Variable

Aufgabe 2 (14 Punkte): Die Sprache $\mathcal{R}$ ist definiert durch den regulären Ausdruck a*bc* und die Sprache $\mathcal{S}$ durch a*b*c entsprechend.

a) Ist der Durchschnitt $\mathcal{R} \cap \mathcal{S}$ von $\mathcal{R}$ und $\mathcal{S}$ wieder eine Sprache? (Begründung!)

b) Falls ja, ist $\mathcal{R} \cap \mathcal{S}$ eine reguläre Sprache? (Begründung!)

Aufgabe 3 (20 Punkte): Problem: Gegeben ist ein Textfile das einen einzigen mathematischen Term enthält. Ein Term besteht hier nur aus ganzen Zahlen, Klammern und dem Plus Zeichen in der üblichen Weise. Gesucht ist die Anzahl der Summanden, d.h. die Anzahl der Plus Zeichen, wobei das Plus als Vorzeichen und die Plus Zeichen innerhalb von Klammern nicht gerechnet werden. Beispiele:

Input	Ausgabe
2	1
+2	1
2+3	2
+2 +3	2
+(+2)+(+3)	2
+(2+3)	1
(2+3)+(+4+(5+6))	2
+(2+3)+(4)+5	3

Input	Ausgabe
ϵ	parse error
3+	parse error
(+3(	parse error
(+2)+ +3	parse error
()	parse error

a) Schreiben Sie ein lex Programm, das die Token: Zahl, Plus, Klammer-Auf und Klammer-Zu erkennt sowie Leerzeichen ignoriert.

b) Schreiben Sie ein yacc Programm, das einen Term erkennt und die Summanden zählt. Hinweis: Speichern Sie die Anzahl der Summanden eines Terms als den semantischen Wert des Terms.

Aufgabe 4 (15 Punkte): Ergänzen Sie die nachstehende Tabelle, indem Sie jedes Feld genau dann ankreuzen, wenn der String am Anfang der Spalte zur Sprache des reguläre Ausdruck am Anfang der Zeile gehört (siehe erste Zeile).

RE/String	ϵ	a	abc	ac	aa	b
a*	X	X			X	
a*(bc|ac)?						
a[bc]*						
(a|b|c)+						
[abc][bc]						
[bc]|[ac]						

Aufgabe 5 (15 Punkte): Durch ([ab](x[ab])*)? ist eine reguläre Sprache gegeben.

a) Schreiben Sie, ausgehend von den terminalen Symbolen a,b und x, eine kontextfreie Grammatik in reiner BNF für diese Sprache.

b) Geben Sie ein Transition Diagramm für diese Sprache an und entscheiden Sie, mit Begründung, ob Ihr Transition Diagramm deterministisch oder nichtdeterministisch ist.

Aufgabe 6 (25 Punkte): Gegeben ist der folgende Versuch eines yacc Programms. Der Input ist eine beliebige Folge von a's und b's, der Rückgabewert von main sollte die Anzahl der Tripel "aaa" im Input sein.

```
%{ int count=0;
%}
%token a b
%%
input: | input blob;
blob: a | b | triple;
triple: eins zwei drei {count++;};
eins: a;
zwei: a;
drei: a;
%%
int main(){yyparse(); return count;}
```

Diese Grammatik hat einen Reduce/Reduce Konflikt und liefert nicht das richtige Ergebnis.

a) Erklären Sie was ein Reduce/Reduce Konflikt ist.

b) Wo tritt bei der gegebenen Grammatik dieser Konflikt auf?

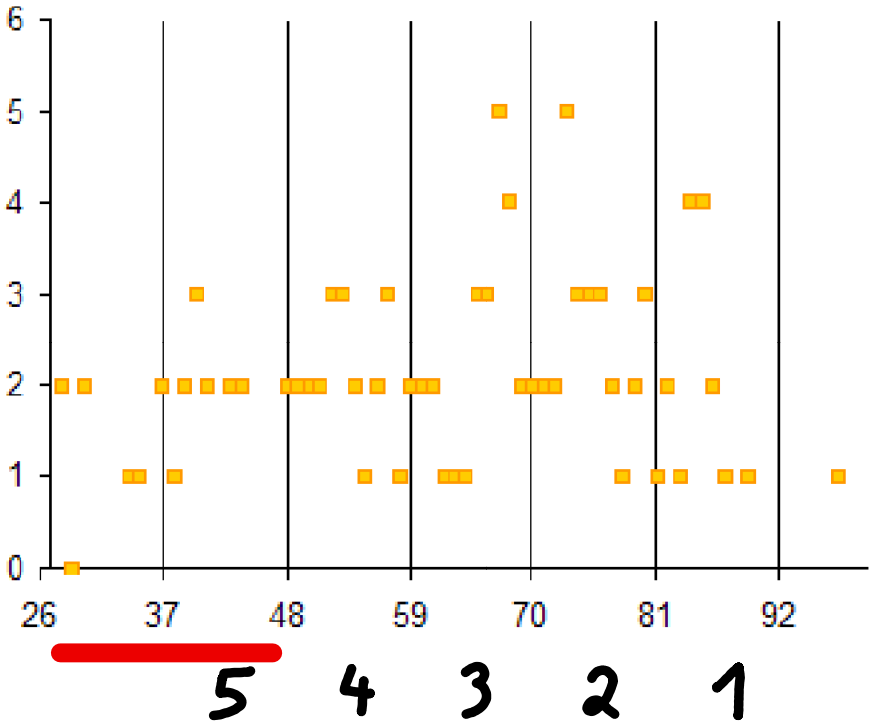
c) Schreiben Sie die Grammatik um, so dass der Reduce/Reduce Konflikt vermieden wird.

   Hinweis: Es ist relativ leicht hier nur den Reduce/Reduce Konflikt zu beseitigen, meist bekommt man dann aber einen Shift/Reduce Konflikt und gelegentliche parse errors. Es ist viel schwerer eine konfliktfreie und funktionierende Grammatik zu schreiben. Sie haben die Wahl zwischen beiden Varianten. Im ersten Fall haben Sie dann etwas mehr Arbeit bei den nächsten beiden Fragen, die Sie im zweiten Fall dann einfach mit "Nein" beantworten können.

d) Hat Ihre Grammatik Shift/Reduce Konflikte? Wenn ja, welche?

e) Produziert Ihre Grammatik parse errors? Wenn ja, welche? und warum?

Ergebnisse: Damit man auch einen Eindruck davon bekommt wie die Klausur ausgefallen ist, hier ein Histogram:



Ab 48 Punkten gab es eine 4, ab 59 Punkten eine 3, ab 70 Punkten eine 2 und ab 81 Punkten eine 1. Insgesamt gab es 20 Fünfer, 23 Vierer, 26 Dreier, 26 Zweier und 17 Einser.