Softwareparadigmen

Dieses Skriptum basiert auf der Softwareparadigmen Übung im Sommersemester 2011 und dem Vorlesungsskriptum 2007.
Vorlesung von Alexander Felfernig

Übungsskriptum verfasst von Daniel Gruß. Version 25. Dezember 2011.
Fehlerfunde bitte melden an gruss@student.tugraz.at.

Inhaltsverzeichnis

Kapitel 1

Syntax

Dieses Skriptum versucht einen kompakten und übersichtlichen Zugang zu den Themen dieser Lehrveranstaltung zu schaffen. Der Vollständigkeit halber verweisen wir in der Fußnote auf die entsprechenden Kapitel des Vorlesungsskriptums.

In der Informatik stehen wir oft vor dem Problem, dass wir eine Art Text auf Fehler überprüfen wollen und in eine andere Darstellungsform übersetzen wollen. Ein Webbrowser prüft beispielsweise HTML-Dokumente auf Fehler und übersetzt diese sofern möglich in eine praktische visuelle Darstellung.

Dieses Problem wollen wir auch lösen können. Dazu brauchen wir zunächst einige grundlegende Definitionen.

1.1 Definitionen

Definition 1.1 (Alphabet)**:** Ein Alphabet Σ ist eine endliche Menge von Symbolen.

Binärzahlen haben das Alphabet $\Sigma = \{0, 1\}$. Eine einfache Variante der Markup-Sprache HTML hat das Alphabet $\Sigma = \{\texttt{<}, \texttt{/}, \texttt{>}, \ldots, a, b, c, \ldots\}$.

Definition 1.2: Σ^* ist die Menge aller beliebigen Konkatenationen von Symbolen aus Σ. Ein Element aus Σ^* nennen wir Wort.

Für $\Sigma = \{0, 1\}$ ist $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$.

Definition 1.3 (Sprache)**:** Eine Sprache $\mathcal{L}$ ist eine Teilmenge ($\subseteq$) von Σ^*.

Sei $\Sigma = \{0, 1\}$. Definieren wir die Sprache der Binärzahlen $\mathcal{B}$, müssen wir zu jedem Wort aus Σ^* entscheiden ob dieses Wort in B enthalten ist. Im Fall der Binärzahlen könnten wir z.B. ε herausnehmen, dann ist die Sprache $\mathcal{B} = \Sigma^* \setminus \{\varepsilon\}$.

Würden wir die Programmiersprache $\mathcal{C}$ als formale Sprache definieren, so wäre ein gesamtes (gültiges) $\mathcal{C}$-Programm ein Wort der Sprache, also ein Element der Menge $\mathcal{C}$.

Definition 1.4 (Compiler)**:** Seien $\mathcal{A}$ und $\mathcal{B}$ Programmiersprachen. Ein Compiler ist ein Programm, welches Programme von $\mathcal{A}$ nach $\mathcal{B}$ übersetzt.

Ein einfacher Compiler würde beispielseweise Binärzahlen zu Dezimalzahlen übersetzen.

1.2 Grammatiken und Sprachen

Oft möchte man nur prüfen ob ein Wort in einer Sprache enthalten ist. Es ist umständlich die Menge aller Wörter einer Sprache aufzuschreiben und darin zu suchen. Daher wollen wir eine kürzere und Methode finden um Sprachen zu beschreiben und diese Überprüfung durchzuführen.

Über Automaten (FSM) ist dies möglich. Für Binärzahlen könnte man dies wie in Abbildung 1.1 machen.

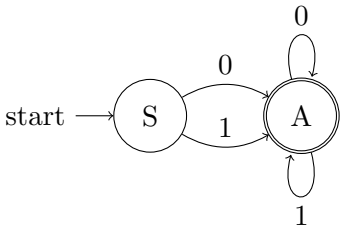


Abb. 1.1: Binärzahlen FSM

Wir haben eine Eingabe w (einem potentiellen Wort der Sprache). Wir beginnen im Zustand S. Die einzigen gültigen Übergänge sind die zu Zustand A. Es muss also entweder eine 0 oder eine 1 gelesen werden (wir schreiben auch gematcht bzw. geparst). Andernfalls gibt es keinen gültigen Übergang und da S kein Endzustand ist wäre die Eingabe nicht gültig. Im Zustand A wurde bereits eine 0 oder 1 gelesen, also ist $w \in \mathcal{B}$. Die Übergänge von A zu A erlauben weitere Zeichen zu lesen und so längere Wörter zu erhalten, die in der Sprache enthalten sind.

Eine ähnliche Herangehensweise stellen Grammatiken dar:

Definition 1.5 (Grammatik)**:** Eine Grammatik ist ein 4-Tupel (V_N, V_T, S, Φ).

- V_N ist eine endliche Menge von Nonterminalen (entsprechen Zuständen der FSM),
- V_T ist eine endliche Menge von Terminalen (entsprechen Alphabet der Sprache),
- $S \in V_N$ das Startsymbol (wie Initialzustand der FSM),
- $\Phi = \{\alpha \to \beta\}$ eine endliche Menge von Produktionsregeln (Zustandsübergänge).

  α ist hierbei eine beliebige Aneinanderreihung von Terminalen und Nonterminalen, die zumindest ein Nonterminal enthält, also $(V_N \cup V_T)^* V_N (V_N \cup V_T)^*$.

  β ist eine beliebige Aneinanderreihung von Terminalen und Nonterminalen, einschließlich der leeren Menge, also $(V_N \cup V_T)^*$.

Anhand der Binärzahlen wollen wir nun genauer betrachten wie die Produktionsregeln funktionieren. Bei den Zustandsübergängen des Automaten hatten wir beispielsweise ein Wort $w = AB$ mit $A \in \{0,1\}$ und $B \in \{0,1\}^*$. Mit dem Zustandsübergang würden wir nun das A weglassen und hätten für das verbleibende zu lesende Wort $w' = B$. Wir ersetzen also AB durch B. Bei den Produktionsregeln der Grammatik halten wir ganz konkret fest was wir ersetzen.

Um die Grammatik der Binärzahlen $G_{\mathcal{B}}$ zu definieren müssen wir die Elemente des oben definierten 4-Tupels beschreiben. Die Menge der Terminale ist das Alphabet $\{0,1\}$. Den Startzustand nennen wir S. In Abbildung 1.2 versuchen wir Produktionsregeln anzugeben um $\mathcal{B}$ zu definieren.

$$S \to 0$$
$$S \to 1$$
$$S \to 0S$$
$$S \to 1S$$

Abb. 1.2: Binärzahlen Grammatik

Um zu verifizieren wann ein Wort von einer Grammatik erzeugt werden kann müssen wir nun zuerst definieren was eine Ableitung ist.

Definition 1.6: Sei (V_N, V_T, S, Φ) eine Grammatik und $\alpha, \beta \in (V_N \cup V_T)^*$. Wenn es 2 Zeichenfolgen τ_1, τ_2 gibt, so dass $\alpha = \tau_1 A \tau_2$, $\beta = \tau_1 B \tau_2$ und $A \to B \in \Phi$, dann kann β direkt (in einem Schritt) von α abgeleitet werden ($\alpha \to \beta$).

Diese Definition ist nur geeignet um eine Aussage darüber zu treffen was in genau einem Schritt abgeleitet werden kann. Wir definieren daher die reflexive Hülle dieses Operators.

Definition 1.7: Sei (V_N, V_T, S, Φ) eine Grammatik und $\alpha, \beta \in (V_N \cup V_T)^*$. Wenn es $n \in \mathbb{N}$ Zeichenfolgen τ_1, τ_n gibt, so dass $\alpha \to \tau_1, \tau_1 \to \tau_2, \ldots, \tau_{n-1} \to \tau_n, \tau_n \to \beta$, dann kann β von α (in n Schritten) abgeleitet werden ($\alpha \xrightarrow{+} \beta$).

Diese Definition ist für $n \geq 1$ geeignet. Wenn $\alpha = \beta$ ist, wäre $n = 0$. Wir definieren die reflexive, transitive Hülle durch eine Verknüpfung dieser beiden Fälle.

Definition 1.8: Es gilt $\alpha \xrightarrow{*} \beta$, genau dann wenn $\alpha \xrightarrow{+} \beta$ oder $\alpha = \beta$ (reflexive, transitive Hülle).

Mit dieser Definition können wir alle Wörter ableiten die diese Grammatik produziert.

Definition 1.9: Sei (V_N, V_T, S, Φ) eine Grammatik G. G akzeptiert die Sprache $L(G) = \left\{ w \mid S \xrightarrow{*} w, w \in V_T^* \right\}$, d.h. die Menge aller Wörter w die in beliebig vielen Schritten aus dem Startsymbol S ableitbar sind und in der Menge aller beliebigen Konkatenationen von Terminalsymbolen V_T enthalten sind.

Die Äquivalenz von zwei Sprachen zu zeigen ist im Allgemeinen nicht trivial. Wenn wir versuchen eine Sprache $\mathcal{L}$ durch eine Grammatik G zu beschreiben ist es im Allgemeinen nicht trivial zu zeigen dass $L(G) = \mathcal{L}$.

An dieser Stelle möchten wir festzuhalten: Um die Gleichheit zweier Mengen (Sprachen) M, N zu zeigen muss gezeigt werden, dass jedes Element (Wort) aus M in N enthalten ist und jedes Element (Wort) aus N in M enthalten ist. Ungleichheit ist daher viel leichter zu zeigen, da es genügt ein Element (Wort) zu finden welches nicht in beiden Mengen (Sprachen) enthalten ist. Der geneigte Leser kann probieren die Gleichheit oder Ungleichheit der Sprache unserer oben definierten Grammatik und der Sprache der Binärzahlen zu zeigen.

Definition 1.10: Ein Programm $P_{\mathcal{L}}$ welches für ein Wort w entscheidet ob es in der Sprache $\mathcal{L}$ enthalten ist (d.h. `true` dann und nur dann zurückliefert wenn es enthalten ist), nennen wir **Parser**.

1.3 Chomsky-Sprachhierarchie

Durch die Grammatik können wir entscheiden ob ein Wort in einer Sprache enthalten ist. Im Falle der Binärzahlen haben wir eine kurze und einfache Grammatik und können einen Parser schreiben welches entscheidet ob ein Eingabewort in der Sprache enthalten ist. Definieren wir eine Programmiersprache wie $\mathcal{C}$ formal, so wird nicht nur die Anzahl der Produktionsreglen höher sein sondern auch die Komplexität der Produktionsregeln ($\alpha \to \beta$ mit komplexen Ausdrücken für α und β). Es ist dann erheblich schwieriger einen Parser zu schreiben. Wir haben also Interesse daran möglichst einfache Grammatiken zu finden. Dazu müssen wir Grammatiken nach ihrer Komplexität vergleichen können.

Wir haben Produktionsregeln definiert durch $\alpha \to \beta$, wobei α zumindest ein Nonterminal enthält.

Definition 1.11: Eine Grammatik ist nach der Chomsky-Sprachhierarchie:

- allgemein/uneingeschränkt (unrestricted)

  Keine Restriktionen

- Kontext-sensitiv (context sensitive): $|\alpha| \leq |\beta|$

  Es werden nicht mehr Symbole gelöscht als produziert werden)

- Kontext-frei (context free): $|\alpha| \leq |\beta|\,, \quad \alpha \in V_N$

  Wie Kontext-sensitiv; außerdem muss α genau **ein** Non-Terminal sein

- regulär (regular): $|\alpha| \leq |\beta|\,, \quad \alpha \in V_N, \quad \beta = aA, \quad a \in V_T \cup \{\varepsilon\}\,, A \in V_N \cup \{\varepsilon\}$

  Wie Kontext-frei; außerdem ist $\beta = aA$ wobei a ein Terminal oder ε ist und A ein Nonterminal oder ε. (Anmerkung: $\varepsilon\varepsilon = \varepsilon$)

Es gilt $\mathbb{L}_{\text{regular}} \subset \mathbb{L}_{\text{context free}} \subset \mathbb{L}_{\text{context sensitive}} \subset \mathbb{L}_{\text{unrestricted}}$ ($\mathbb{L}_{\text{x}}$ Menge aller Sprachen der Stufe x).

Nicht alle Grammatiken können in eine äquivalente Grammatik einer stärker eingeschränkten Stufe umgewandelt werden. Um zu zeigen dass es sich um echte Teilmengen ($A \subset B$) handelt müssen wir zeigen dass alle Elemente aus A in B enthalten sind und mindestens ein Element aus B nicht in A enthalten ist. Eine Beweisskizze dazu findet sich im Vorlesungsskriptum auf Seite 12. Dort wird wird unter gezeigt, dass es für $L = \{a^n b a^n | n \in \mathbb{N}_0\}$ äquivalente reguläre Grammatik G gibt, d.h. $\nexists G \in \mathbb{L}_x : L(G) = L$.

Die Chomsky-Sprachhierarchie unterscheidet Sprachen anhand der Komplexität der produzierten Sprache.

1.4 Parser

Wir überspringen an dieser Stelle den BPARSE-Algorithmus (siehe Vorlesungsskriptum) und betrachten stattdessen einen Recursive Descent Parser (RDP).

Bei einem RDP werden alle Nonterminale in Funktionen übersetzt und diese Funktionen behandeln die verschiedenen Produktionsregeln. Die Eingabe wird in die Terminale unterteilt (auch Tokens genannt). Wir verwenden im Pseudo-Code die Variable `token`, die immer das aktuelle Token enthält, sowie die Funktion `nextToken()`, die `token` auf das nächste Token setzt. Der Parser ruft die Startfunktion auf und gibt `true` zurück wenn `token` leer ist (d.h. das Ende der Eingabe erreicht wurde). `ERROR` führt dazu dass der Parser die Eingabe (das Wort) nicht akzeptiert.

Sei $G_{\text{bin1}} = (\{S, T\}, \{0, 1\}, S, \{S \to 0, S \to 1T, T \to 0T, T \to 1T, T \to \varepsilon\})$. Der Pseudo-Code des RDP zu dieser Grammatik könnte so aussehen:

```
FUNC S()
  IF token == 0
    nextToken()
  ELSE IF token == 1
    nextToken()
    T()
  ELSE
    ERROR
```

```
FUNC T()
  IF token == 0
    nextToken()
    T()
  ELSE IF token == 1
    nextToken()
    T()
  ELSE IF token != epsilon
    ERROR
```

Sei $G_{\text{bin2}} = (\{S, T\}, \{0, 1\}, S, \{S \to 0, S \to 1T, T \to T0, T \to T1, T \to \varepsilon\})$. Wir wissen $L(G_{\text{bin1}}) = L(G_{\text{bin2}})$ (ohne Beweis) und versuchen auch hier einen einfachen rekursiven Parser zu schreiben.

```
FUNC S()
  IF token == 0
    nextToken()
  ELSE IF token == 1
    nextToken()
    T()
  ELSE
    ERROR
```

```
FUNC T()
  IF token == 0
    T()        // Endlos-Rekursion
    nextToken()
  ELSE IF token == 1
    T()        // Endlos-Rekursion
    nextToken()
  ELSE IF token != epsilon
    ERROR
```

Wir erhalten hier im Parser eine Endlos-Rekursion, da in $T \to T0, T \to T1$ ein Nonterminal ganz links steht. Wir nennen dies "Linksrekursion".

Definition 1.12 (Mehrdeutig, Eindeutig)**:** Sei $G = (V_N, V_T, S, \Phi)$ eine Grammatik. Wenn es für ein Wort $w \in L(G)$ mehrere unterschiedliche Ableitungssequenzen ω, ψ, d.h. $S \to \omega_1, \omega_1 \to \ldots, \ldots \to \omega_n, \omega_n \to w$, $S \to \psi_1, \psi_1 \to \ldots, \ldots \to \psi_k, \omega_k \to w$ wobei $\exists \psi_i : \psi_i \neq \omega_i$, ist es mehrdeutig. Wenn eine Grammatik ist genau dann eindeutig, wenn sie nicht mehrdeutig ist.

Definition 1.13 (Linksrekursiv)**:** Eine Grammatik ist direkt linksrekursiv wenn sie eine Produktion der Form $A\alpha \to A\beta$ enthält, wobei A ein Nonterminal ist. Eine Grammatik ist indirekt linksrekursiv wenn sie Produktionen der Form $A\alpha \to A_1\beta_1, A_1\alpha_1 \to A_2\beta_2, \ldots, A_n\alpha_n \to A\beta_n$ enthält, wobei A, A_i Nonterminale sind. Eine Grammatik ist linksrekursiv wenn sie direkt oder indirekt linksrekursiv ist.

Nun können wir mit Sprachen und Grammatiken umgehen und diese nach ihrer Komplexität einstufen.

1.5 Compilerbau

Wir wollen uns nun damit beschäftigen wie wir Compiler für Programmiersprachen bauen können. Wir hatten definiert dass ein Compiler eine Wort w einer Sprache $\mathcal{L}$ entgegennimmt und die Übersetzung des Wortes w' in einer Sprache $\mathcal{L}'$ zurückgibt. Konkreter erhält unser Compiler einen beispielsweise einen ASCII-String, wobei unser Alphabet $\Sigma_{\mathcal{L}}$ meist nicht dem ASCII-Alphabet entspricht. Betrachten wir beispielsweise die Programmiersprache $\mathcal{C}$ so können wir auch Schlüsselwörter wie `int` in $\Sigma_{\mathcal{C}}$ haben. Außerdem gibt es eventuell Whitespaces (Leerzeichen, Tabulatoren, Zeilenumbrüche, etc.; je nach dem positionsabhängig), die keinen Einfluss darauf haben ob das Eingabewort ein Wort der Sprache $\mathcal{C}$ ist, d.h. ein $\mathcal{C}$-Programm ist. Auch Variablen- oder Funktionsnamen werden abgesehen von einer gewissen Form die sie einhalten müssen (z.B. "dürfen nicht nur aus Zahlen bestehen") keinen Einfluss darauf haben ob das Programm in der Sprache enthalten ist. Wenn wir dies berücksichtigen, verkürzen wir die Grammatik und vereinfachen

so unseren Parser sowie eventuelle weitere Rechenschritte.

1.6 Lexikalische Analyse

Wir haben bereits in Abschnitt 1.4 von Tokens geredet. Von Tokens spricht man insbesondere bei einer Sprache die durch Whitespaces getrennt werden. Ein Token ist hierbei die kleinste Sequenz von Zeichen die für die Grammatik eine Bedeutung hat. Der erste Schritt der Kompiliervorgangs ist es die Eingabe in eine Sequenz von Tokens umzuwandeln (dies kann natürlich wie bei unserem rekursiven Parser während dem Parsen passieren).

Wir wollen z.B. den $\mathcal{C}$-Code `int main() { printf("helloworld"); return 123; }` in die Sequenz von Tokens `INT ID ( ) { ID ( STRING ) ; RETURN NUM ; }` umwandeln. Wir nennen dies "Lexikalische Analyse". Es fällt auf dass in der Token-Sequenz keine Namen oder Zahlen mehr vorkommen außerdem sind nun die Tokens voneinander getrennt. Es ist üblich zur lexikalischen Analyse nur reguläre Grammatiken zu verwenden. Diese sind mächtig genug um beispielsweise nur gewisse Variablennamen zu erlauben und können andererseits sehr schnell berechnet werden. Eine reguläre Grammatiken kann auch über einen regulären Ausdruck (Regular Expression/Regex) kompakt dargestellt werden.

Definition 1.14: Ein regulärer Ausdruck A ist wie folgt rekursiv definiert (mit regulären Ausdrücken Q und R):

$$A = \begin{cases} \varepsilon & \text{Leerstring} \\ t & \text{ein Terminal, d.h. } t \in V_T \\ Q|R & \text{entweder } A = Q \text{ oder } A = R \\ QR & \text{Konkatenation zweier regulärer Ausdrücke} \\ Q? & \text{entspricht } Q|\varepsilon\text{, d.h. } Q \text{ ist optional} \\ Q* & \text{beliebige Konkatenation von } Q \text{ mit sich selbst, d.h. } A \in \{\varepsilon, Q, QQ, \ldots\} \\ Q+ & \text{entspricht } QQ*\text{, d.h. mindestens ein } Q \\ (Q) & \text{Klammerung des Ausdrucks } Q \end{cases}$$

Die Sprache der Binärzahlen hatten wir bereits in Abschnitt 1.4 durch die Grammatik $G_{\text{bin1}} = (\{S, T\}, \{0, 1\}, S, \{S \to 0, S \to 1T, T \to 0T, T \to 1T, T \to \varepsilon\})$ beschrieben. Wir können nun diese Grammatik in einen regulären Ausdruck übersetzen indem wir uns nach und nach überlegen welche Werte folgen können. Beginnend bei S erhalten wir den Teilausdruck $0|(1\tau)$. Für τ müssen wir noch einen regulären Ausdruck einsetzen: $\tau = (0|1)*$. Damit erhalten wir den Ausdruck $0|(1(0|1)*)$.

Zur Vereinfachung erlauben wir Variablen (Bezeichner) in regulären Ausdrücken:

Definition 1.15: Sei B die Menge aller Variablen (Bezeichnungen). Ein regulärer Ausdruck der eine Variable b aus B enthält ist ein erweiterter regulärer Ausdruck. Wenn E ein erweiterter regulärer Ausdruck ist und c eine Variable aus B, dann ist $c := E$ ist eine reguläre Definition.

Die Sprache der natürlichen Zahlen inkl. Null beschreiben wir durch den Ausdruck `0|((1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)`. Durch die Definition erweiterter regulärer Ausdrücke können wir nun komplexe Ausdrücke in due einzelnen Bestandteile kapseln und so besser lesbare Ausdrücke schaffen. Diesen Ausdruck können wir nun aufteilen indem wir die Teilausdrücke `digit_not_null := (1|2|3|4|5|6|7|8|9)` und `digit := 0 | digit_not_null` definieren. Dann erhalten wir den wesentlich leichter verständlichen Ausdruck `0|(digit_not_null digit*)`.

Mit erweiterten regulären Ausdrücken können wir nun einfach den Eingabestring in eine Token-Sequenz aufteilen. Übersetzen wir dazu den erweiterten regulären Ausdruck zu einer Funktion so beobachten wir:

- Terminale werden zu `if`-Abfragen,
- `Q*` wird zu einem Schleifenkonstrukt,
- `Q|R` wird zu einer `if, else if, else`-Abfrage wobei der `else` Fall zu einem Ablehnen der Eingabe führt,
- `QR` bedeutet dass zuerst `Q` überprüft wird, danach `R`.

1.7 Grammatikalische Analyse

Der letzte Schritt der Syntaxanalyse ist die grammatikalische Analyse. In diesem Schritt wollen wir anhand der Token-Sequenz prüfen ob das Eingabewort in der Sprache der Grammatik enthalten ist. Den Verarbeitung einschließlich diesen Schrittes nennen wir "Parsing". Als Nebenprodukt wird oft ein Parse-Baum aufgebaut, der zusätzlich zur Reihenfolge der Tokens auch die Kapselung im Programm ausdrückt. Hierzu verwenden wir das Beispiel aus dem Vorlesungsskriptum und werden uns auch weitgehend an diesem orientieren. Wir entwerfen eine Grammatik für einfache arithmetische Ausdrücke, bestehend aus Zahlen, Operatoren (Addition und Multiplikation) und Klammern. Die Bindungsstärke von Operatoren behandeln wir auf Syntax-Ebene nicht.

Bei kontext-freien oder regulären Grammatiken werden wir fortan nur noch die Produktionsregeln mit unterstrichenen Nonterminalen anschreiben, da die Grammatik dadurch vollständig definiert wird:

- V_N ist die Vereinigung über alle Symbole auf der linken Seite der Produktionsregeln (d.h. alle Nonterminale),
- V_T ist die Vereinigung aller unterstrichenen Zeichenfolgen (d.h. alle Terminale),

- S, das Startsymbol ist (soweit nicht anders festgehalten) das erste aufgeführte Nonterminal,
- Φ wird explizit angegeben.

Wir definieren nun die Sprache der einfachen arithmetischen Ausdrücke durch:

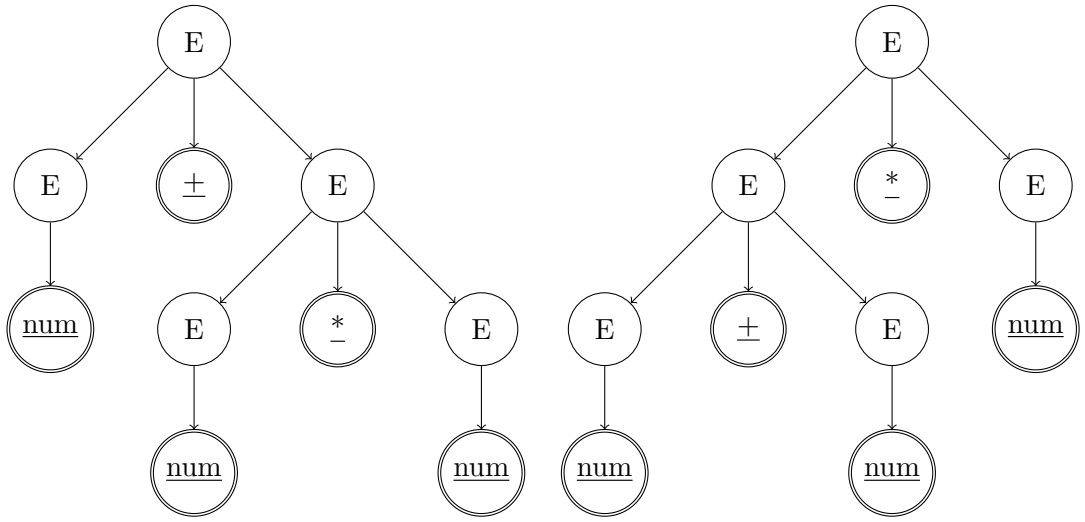
1. $E \rightarrow E\underline{+}E$
2. $E \rightarrow E\underline{*}E$
3. $E \rightarrow \underline{(}E\underline{)}$
4. $E \rightarrow \underline{\text{num}}$

Wir sehen dass diese Grammatik eine kontext-freie Grammatik ist. Außerdem sehen wir anhand von Regel 1 und 2, dass die Grammatik linksrekursiv ist. Weiters können wir für den Satz $\underline{\text{num + num * num}}$ zwei mögliche Ableitungen finden. Wir verwenden dazu eine verkürzte Form der Darstellung aus Abschnitt 1.4. Die Vorgehensweise dabei nennt man auch Top-Down-Parsing.

1. $E \rightarrow E\underline{*}E \rightarrow E\underline{*}E\underline{+}E \rightarrow \underline{\text{num}+}E\underline{*}E \rightarrow \underline{\text{num + num *}}E \rightarrow \underline{\text{num + num * num}}$
2. $E \rightarrow E\underline{+}E \rightarrow E\underline{+}E\underline{*}E \rightarrow \underline{\text{num}+}E\underline{*}E \rightarrow \underline{\text{num + num *}}E \rightarrow \underline{\text{num + num * num}}$

Dies zeigt dass die Grammatik mehrdeutig ist.

Eine weitere Möglichkeit dies anschaulich darzustellen sind Parse-Trees. Ein Parse-Tree stellt einen konkreten Parse-Vorgang dar. Der Wurzel-Knoten das Startsymbol. Die Kinder jedes Knotens sind die einzelnen Terminale und Nonterminale die dieser Knoten produziert (von links nach rechts entsprechend der Grammatik). Die obigen Parse-Sequenzen lassen sich so darstellen:



Das geparste Wort kann in den Blättern des Baumes von links nach rechts abgelesen werden. Hier ist die Mehrdeutigkeit ebenfalls sehr gut erkennbar.

Eine weitere Beobachtung ist, dass es ungünstig ist wenn es für unseren Parser nicht genügt dass aktuelle Token zu kennen sondern für das Parsing auch weitere nachfolgende Tokens ermittelt werden müssen. Ist das Eingabewort `num`, so haben wir 3 Regeln die alle zutreffen könnten (ohne Kenntnis der nachfolgenden Tokens).

Wir definieren einfache Regeln zwecks Auflösung von:

- **Indirekten Linksrekursionen**

  Gibt es Produktionen der Form $A \to \alpha B \beta$ sowie $B \to \gamma$, dann kann man die Produktion $A \to \alpha\gamma\beta$ einfügen. Wenn $\alpha = \varepsilon$ ist können so indirekte Linksrekursionen gefunden werden.

- **Direkten Linksrekursionen**

  Gibt es Produktionen der Form $A \to A\alpha$ und $A \to \beta$, dann kann man diese in Produktionen der Form $A \to \beta R$ sowie $R \to \alpha R|\varepsilon$ umwandeln.

- **Linksfaktorisierungen**

  Gibt es Produktionen der Form $A \to \alpha B$ sowie $A \to \alpha C$, wobei α der größte gemeinsame Prefix von αB und αC ist, so können wir diese Produktion durch $A \to \alpha R$ und $R \to B|C$ ersetzen. Der größte gemeinsame Prefix einer Sequenz von Terminalen und Nonterminalen ist die größte gemeinsame Zeichenkette dieser Sequenzen. Der größte gemeinsame Prefix von if E then E else E und if E then E ist if E then E.

Kapitel 2

Semantik von Programmiersprachen