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 10. Februar 2012.
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. Wir haben uns dabei stark am Vorlesungsskriptum orientiert. Das Vorlesungsskriptum und folglich auch dieses Skriptum bezieht die angeführten Sprachen und Konzepte großteils aus dem Skriptum "Einführung in die Theorie der Informatik" von A. Leitsch, TU Wien, 1989.

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 Grundlegende 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.

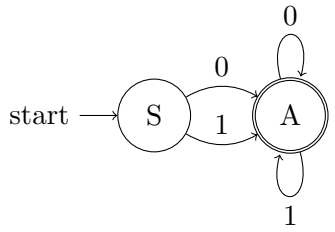


Abb. 1.1: Binärzahlen FSM

Wir haben eine Eingabe (mglw. Wort der Sprache) und 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. Um zu verifizieren wann ein Wort von einer Grammatik erzeugt werden kann müssen wir nun zuerst definieren was eine Ableitung ist.

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

Abb. 1.2: Binärzahlen Grammatik

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.

- 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.

Beispiel 1.1: 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
```

Beispiel 1.2: 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 Grammatik 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:

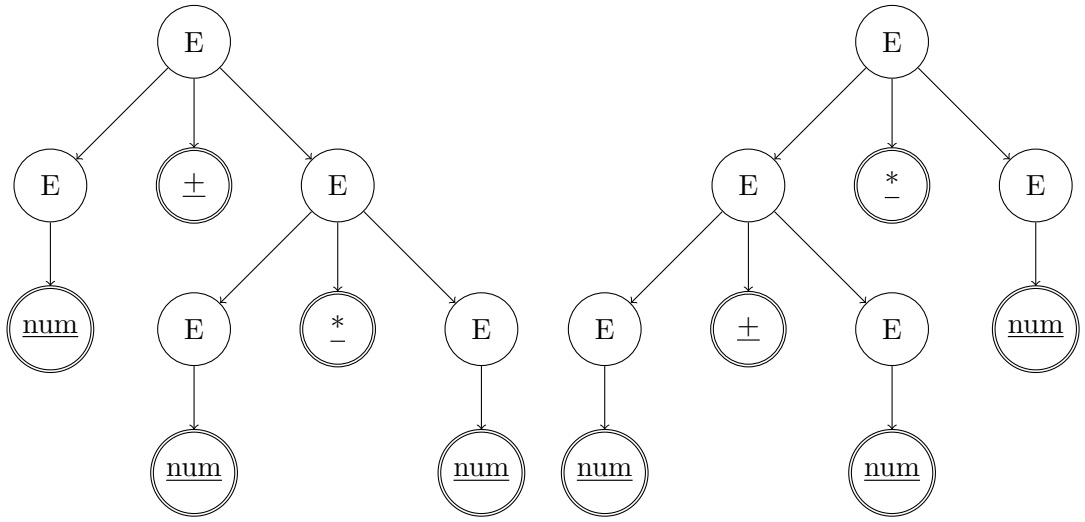
1. $E \rightarrow E\underline{+}E$
2. $E \rightarrow E\underline{*}E$
3. $E \rightarrow \underline{(}E\underline{)}$
4. $E \rightarrow \underline{\text{num}}$

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).

1.8 LL(1)-Grammatiken

Definition 1.16 (LL(1)-Grammatik)**:** Eine kontextfreie Grammatik G ist eine LL(1)-Grammatik (ist in LL(1)-Form) wenn sie

- keine Linksrekursionen enthält
- keine Produktionen mit gleichen Prefixen für die selbe linke Seite enthält
- ermöglicht immer in einem Schritt (d.h. nur mit Kenntnis des nächsten Tokens) zu entscheiden, welche Produktionsregel zur Ableitung verwendet werden muss.

LL(1) steht für "**L**eft to right", "**L**eftmost derivation", "**1** Token Look-ahead".

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.

Mit diesen Regeln können wir oftmals Grammatiken in äquivalente LL(1)-Grammatiken umformen. Es gibt natürlich Grammatiken bei denen dies nicht möglich ist. In diesem Fall bleibt nur die Möglichkeit eine andere Parsing-Methode zu verwenden oder die Sprache zu verändern.

Beispiel 1.3:

Überlegen wir uns einmal anhand eines kleineren Beispiels was eine direkte Linksrekursion eigentlich bedeutet und wie man diese intuitiv auflöst. Wir werden anhand dieses Beispiels eine Herleitung der Regel für die direkte Linksrekursion skizzieren.

1. $A \to \underline{b}$
2. $A \to A\underline{a}$

Wir erkennen auf den ersten Blick dass jedes Wort der Sprache dieser Grammatik wohl mit $\underline{b}$ beginnen muss. Dann folgen eine beliebige Anzahl von $\underline{a}$. Als regulären Ausdruck könnte man schreiben `ba*` Versuchen wir nun die Sprache intuitiv umzubauen. Dazu ändern wir zunächst wie folgt die 2. Regel um:

1. $A \to \underline{b}$
2. $A \to \underline{a}A$

Jetzt haben wir `a*b` gebaut, also bauen wir ein neues Non-Terminal für die Produktionen der $\underline{a}$, die man erst nach einem $\underline{b}$ produzieren können sollte.

1. $A \to \underline{b}$
2. $B \to \underline{a}B$

Nun können wir nur noch `b` produzieren, daher fügen wir ein B in die 1. Produktion ein. Außerdem würde die Rekursion von B nie terminieren.

1. $A \to \underline{b}B$
2. $B \to \underline{a}B$

Nun produzieren wir zuerst ein $\underline{b}$ und dann beliebig viele $\underline{a}$, allerdings terminiert die Rekursion nie. Wir fügen also eine regel $B \to \varepsilon$ ein und erhalten die richtig umgeformte Grammatik:

1. $A \to \underline{b}B$
2. $B \to \underline{a}B|\varepsilon$

Es ist auffällig dass unsere Umformung der Form entspricht die in den Regeln beschrieben wurde.

Beispiel 1.4:

In diesem Beispiel möchten wir nun auch die Regeln der Linksfaktorisierung und indirekte Linksrekursion durch intuitive Herangehensweise herleiten. Gegeben Sei die folgende Grammatik die arithmetische Ausdrücken einer bestimmten Form beschreibt:

1. $E \to G$
2. $G \to E\underline{+}E$
3. $G \to E\underline{*}E$
4. $E \to \underline{(}E\underline{)}$
5. $E \to \underline{(}\underline{\text{num}}\underline{)}$
6. $E \to \underline{(}G\underline{)}\underline{/}\underline{(}G\underline{)}$

Versuchen wir nun zu Bestimmen: Gibt es Linksrekursionen oder gemeinsame Prefixe? Gibt es weitere Probleme bei dieser Grammatik?
In der Grammatik sind keine direkten Linksrekursionen. Regeln 2, 5 und 6 haben einen gemeinsamen Prefix ($\underline{(}$). Man sieht auch recht schnell dass durch die Produktionen $E \to G \to E\underline{*}E$ eine indirekte Linksrekursion entsteht.

Wenn wir uns überlegen wie man die gemeinsamen Prefixe entfernen kann wird man intuitiv auf die gleiche Idee kommen die oben in den Regeln festgehalten wurde: Wir führen ein neues Non-Terminal ein welches die Produktion von ($\underline{(}$) übernimmt und anschließend einen der Reste produziert. Wir erhalten dann die nebenstehende Grammatik.

1. $E \to G$
2. $G \to E\underline{+}E$
3. $G \to E\underline{*}E$
4. $E \to \underline{(}B$
5. $B \to E\underline{)}$
6. $B \to \underline{\text{num}}\underline{)}$
7. $B \to G\underline{)}\underline{/}\underline{(}G\underline{)}$

Wie können wir nun die indirekte Linksrekursion lösen? Wir hatten sie durch die Produktionen $E \to G \to E*E$ erkannt. Versuchen wir nun diesen Zwischenschritt zu eliminieren. Dazu können wir die Regel G einfach "einsetzen" - d.h. aus Regel 1 werden 2 neue Regeln, aus Regel 7 werden sogar 4 neue Regeln (alle Kombinationen).

1. $E \to E+E$
2. $E \to E*E$
3. $E \to (B$
4. $B \to E)$
5. $B \to (\text{num})$
6. $B \to E+E)/(E*E)$
7. $B \to E+E)/(E+E)$
8. $B \to E*E)/(E*E)$
9. $B \to E*E)/(E+E)$

Lösen wir zunächst wieder die gemeinsamen Prefixe (Regeln 6-9). Dadurch entstehen wieder neue gemeinsame Prefixe die dann gelöst werden müssen (Regeln 7-10):

1. $E \to E+E$
2. $E \to E*E$
3. $E \to (B$
4. $B \to E)$
5. $B \to (\text{num})$
6. $B \to EC$
7. $C \to +E)/(E*E)$
8. $C \to +E)/(E+E)$
9. $C \to *E)/(E*E)$
10. $C \to *E)/(E+E)$

1. $E \to E+E$
2. $E \to E*E$
3. $E \to (B$
4. $B \to E)$
5. $B \to (\text{num})$
6. $B \to EC$
7. $C \to +E)/(ED$
8. $C \to *E)/(ED$
9. $D \to +E)$
10. $D \to *E)$

1. $E \to E+E$
2. $E \to E*E$
3. $E \to (B$
4. $B \to EF$
5. $F \to)$
6. $B \to (\text{num})$
7. $F \to C$
8. $C \to +E)/(ED$
9. $C \to *E)/(ED$
10. $D \to +E)$
11. $D \to *E)$

Nun haben wir nur noch die direkte Linksrekursionen (Regeln 1 und 2). Wir wissen aus dem vorherigen Beispiel bereits wie wir diese auflösen und erhalten die nebenstehende Grammatik. Wir sehen dass die Regeln genau unser intuitives Vorgehen festhalten und verallgemeinern. Außerdem garantieren korrekte Anwendungen der Regeln dass die Grammatik nach der Umformung noch die gleiche Sprache beschreiben.

1. $E \to (BR$
2. $R \to +E$
3. $R \to *E$
4. $R \to \varepsilon$
5. $B \to EF$
6. $F \to)$
7. $B \to (\text{num})$
8. $F \to C$
9. $C \to +E)/(ED$
10. $C \to *E)/(ED$
11. $D \to +E)$
12. $D \to *E)$

1.9 LL(1)-Tabellen

LL(1)-Tabellen (auch: LL(1)-Parser-Tabellen) ermöglichen es uns einen generischen LL(1)-Parser zu schreiben, der mit einer beliebigen Tabelle (und damit Sprache) arbeiten kann. Wir werden uns nun erarbeiten wie eine solche Tabelle berechnet werden kann. Für die Definition der FIRST- und Follow-Mengen stellen wir uns Nonterminale wieder als Zustände in einem Graph oder eine Maschine vor.

Definition 1.17 (FIRST-Menge)**:** Die FIRST-Menge eines Nonterminals X ist die Menge aller Terminalsymbole die im Zustand X **als erstes** geparst werden können. Die FIRST-Menge eines Terminals x ist immer das Terminalsymbol selbst.

Formal bedeutet dies:

1. Wenn x ein Terminal ist: $\text{FIRST}(x) = \{x\}$
2. Wenn die Grammatik Produktionsregeln enthält so dass $X \to \ldots \to \varepsilon$, dann ist: $\varepsilon \in \text{FIRST}(X)$
3. Für jede Produktionsregel $X \to Y_1 Y_2 \ldots Y_n$, ist $x \in \text{FIRST}(X)$ wenn $x \in \text{FIRST}(Y_i)$ und für alle Y_j mit $j < i$ gilt, dass $\varepsilon \in \text{FIRST}(Y_j)$.

Für aufeinanderfolgende Terminal- bzw. Nonterminalsymbole $X_1 X_2 \ldots X_n$:

1. Wenn $x \in \text{FIRST}(X_i)$ und für alle X_j mit $j < i$ gilt, dass $\varepsilon \in \text{FIRST}(X_j)$, dann ist $x \in \text{FIRST}(X_1 X_2 \ldots X_n)$.
2. Wenn für alle X_i $\varepsilon \in \text{FIRST}(X_i)$ ist, dann ist auch $\varepsilon \in \text{FIRST}(X_1 X_2 \ldots X_n)$.

Aus dieser Definition folgt beispielsweise für einfache Regeln $A \to B$, dass $\text{FIRST}(A) = (\text{FIRST}(B) \setminus \{\varepsilon\}) \cup \ldots$ ist.

Definition 1.18: Zwecks Übersichtlichkeit definieren wir die FIRST*-Menge als FIRST-Menge ohne ε:

$$\text{FIRST}^*(X) = \text{FIRST}(X) \setminus \{\varepsilon\}\,.$$

Beispiel 1.5: Betrachten wir folgendes Beispiel:

$$\begin{aligned}
\text{FIRST}(\underline{b}) = \quad & \{\underline{b}\} && \text{(wird meist nicht aufgeschrieben da trivial)} \\
\text{FIRST}(A) = \quad & \{\underline{b}\} && \text{(durch die 1. Produktion)} \\
\cup \quad & \text{FIRST}^*(A) && \text{(durch die 2. Produktion)} \\
\cup \quad & \{\underline{a}\} && \text{(2. Produktion kann wegfallen d.h. } \varepsilon \text{ werden)} \\
\cup \quad & \text{FIRST}^*(A) && \text{(durch die 3. Produktion)} \\
\cup \quad & \text{FIRST}^*(B) && \text{(3. Produktion, wenn } A \text{ wegfällt)} \\
\cup \quad & \{\varepsilon\} && \text{(durch die 3. Produktion)}
\end{aligned}$$

Wir wissen aus der Mengenlehre dass $M \cup M' = M$ mit $M' \subseteq M$ und können daher FIRST*(A) weglassen:

$$\begin{aligned} &= \{\underline{b}\} \cup \{\underline{a}\} \cup \text{FIRST*}(B) \cup \{\varepsilon\} \\ &= \{\underline{b}, \underline{a}, \varepsilon\} \cup \text{FIRST*}(B) \end{aligned}$$

Um FIRST*(B) zu erhalten müssen wir also zuerst FIRST(B) ausrechnen:

$$\begin{aligned} \text{FIRST}(B) &= \{\underline{b}\} \cup \{\underline{q}\} \\ &= \{\underline{b}, \underline{q}\} \\ \text{FIRST}(A) &= \{\underline{b}, \underline{a}, \varepsilon\} \cup \{\underline{b}, \underline{q}\} \\ &= \{\underline{b}, \underline{a}, \varepsilon, \underline{q}\} \end{aligned}$$

$$\begin{aligned} \text{FIRST}(C) &= \text{FIRST*}(A) \cup \{\underline{c}\} && \text{(da } A \to \varepsilon \text{ werden kann)} \\ &= \{\underline{b}, \underline{a}, \underline{q}\} \cup \{\underline{c}\} \\ &= \{\underline{b}, \underline{a}, \underline{q}, \underline{c}\} \end{aligned}$$

Definition 1.19: Jede Eingabe des Parsers endet mit dem "End of Input" Symbol \$.

Definition 1.20 (Follow-Menge)**:** Die Follow-Menge eines Nonterminals X ist die Menge aller Terminalsymbole die direkt auf Zustand X **folgen** können. Das heißt, alle Terminalsymbole die nach Abarbeitung des Zustands X als erstes geparst werden können.

Formal bedeutet dies:

1. Wenn X das Startsymbol ist, dann ist \$ $\in$ FOLLOW(X)
2. Für alle Regeln der Form $A \to \alpha B \beta$ ist FIRST*$(\beta) \subseteq$ FOLLOW(X)
3. Für alle Regeln der Form $A \to \alpha B$ bzw. $A \to \alpha B \beta$ mit $\varepsilon \in$ FIRST(β) ist FOLLOW$(A) \subseteq$ FOLLOW(B)

Beispiel 1.6: Betrachten wir das gleiche Beispiel wie zuvor:

$A \to \underline{b}$
$A \to A\underline{a}$
$A \to ABC$
$A \to \varepsilon$
$B \to \underline{b} | \underline{q}$
$C \to A\underline{c}$

$$\begin{aligned} \text{FOLLOW}(A) = &\quad \{\$\} && \text{(da } A \text{ das Startsymbol ist)} \\ \cup &\quad \{\underline{a}\} && \text{(durch die 2. Produktion)} \\ \cup &\quad \text{FIRST*}(B) && \text{(durch die 3. Produktion)} \\ \cup &\quad \{\underline{c}\} && \text{(durch die 6. Produktion)} \\ = &\quad \{\$, \underline{a}\} \cup \{\underline{b}, \underline{q}\} \cup \{\underline{c}\} \\ = &\quad \{\$, \underline{a}, \underline{b}, \underline{q}, \underline{c}\} \\ \text{FOLLOW}(B) = &\quad \text{FIRST*}(C) && \text{(durch die 3. Produktion)} \\ = &\quad \{\underline{b}, \underline{a}, \underline{q}, \underline{c}\} \\ \text{FOLLOW}(C) = &\quad \text{FOLLOW}(A) && \text{(durch die 3. Produktion)} \\ = &\quad \{\$, \underline{a}, \underline{b}, \underline{q}, \underline{c}\} \end{aligned}$$

Algorithmus (Parse-Table)**:** Der Parse-Table Algorithmus berechnet aus einer Grammatik eine Parse-Table (auch LL(1)-Tabelle) M.

1. Für jede Produktion $X \to \alpha$:
   (a) Für jedes Element $y \in \text{FIRST}^*(\alpha)$ bzw. wenn $\alpha = y$:
      i. Füge $X \to \alpha$ in $M(X, y)$ ein.
   (b) Wenn $\varepsilon \in \text{FIRST}(\alpha)$:
      i. Für alle $y \in \text{FOLLOW}(X)$:
         A. Füge $X \to \alpha$ in $M(X, y)$ ein.

Leere Einträge in der Tabelle sind Fehlerfälle. Diese können auch explizit mit ERROR beschriftet werden.

Beispiel 1.7: Betrachten wir das gleiche Beispiel wie zuvor:

$A \to \underline{b}$
$A \to A\underline{a}$
$A \to ABC$
$A \to \varepsilon$
$B \to \underline{b} | \underline{q}$
$C \to A\underline{c}$

Wir haben bereits die FIRST- und FOLLOW-Mengen berechnet, wir können also gleich den Algorithmus ausführen.

	FIRST	FOLLOW
A	$\{\underline{b}, \underline{a}, \varepsilon, \underline{q}\}$	$\{\$, \underline{a}, \underline{b}, \underline{q}, \underline{c}\}$
B	$\{\underline{b}, \underline{q}\}$	$\{\underline{b}, \underline{a}, \underline{q}, \underline{c}\}$
C	$\{\underline{b}, \underline{a}, \underline{q}, \underline{c}\}$	$\{\$, \underline{a}, \underline{b}, \underline{q}, \underline{c}\}$

Wir beginnen mit der leeren Parse-Tabelle. Wir haben eine Zeile pro Non-Terminal und eine Spalte je Terminal sowie eine Spalte für "End of Input". Ausführung für $A \to \underline{b}$:

	$\underline{a}$	$\underline{b}$	$\underline{c}$	$\underline{q}$	\$
A					
B					
C					

$\to$

	$\underline{a}$	$\underline{b}$	$\underline{c}$	$\underline{q}$	\$
A		$A \to \underline{b}$			
B					
C					

Schritt 1b trifft auf diese Produktionsregel nicht zu. Ausführung für $A \to A\underline{a}$ und für $A \to ABC$ (auch hier wird die Bedingung aus Schritt 1b noch nicht erfüllt):

$\to$

	$\underline{a}$	$\underline{b}$	$\underline{c}$	$\underline{q}$	\$
A	$A \to A\underline{a}$	$A \to \underline{b}$ $A \to A\underline{a}$		$A \to A\underline{a}$	
B					
C					

$\to$

	$\underline{a}$	$\underline{b}$	$\underline{c}$	$\underline{q}$	\$
A	$A \to A\underline{a}$ $A \to ABC$	$A \to \underline{b}$ $A \to A\underline{a}$ $A \to ABC$		$A \to A\underline{a}$ $A \to ABC$	
B					
C					

Ausführung für $A \to \varepsilon$ (Schritt 1b):

		$\underline{a}$	$\underline{b}$	$\underline{c}$	$\underline{q}$	\$
	A	$A \to A\underline{a}$ $A \to ABC$ $A \to \varepsilon$	$A \to \underline{b}$ $A \to A\underline{a}$ $A \to ABC$ $A \to \varepsilon$	$A \to \varepsilon$	$A \to A\underline{a}$ $A \to ABC$ $A \to \varepsilon$	$A \to \varepsilon$
	B					
	C					

Ausführung für $B \to \underline{b}|\underline{q}$ und $C \to A\underline{c}$

		$\underline{a}$	$\underline{b}$	$\underline{c}$	$\underline{q}$	\$
	A	$A \to A\underline{a}$ $A \to ABC$ $A \to \varepsilon$	$A \to \underline{b}$ $A \to A\underline{a}$ $A \to ABC$ $A \to \varepsilon$	$A \to \varepsilon$	$A \to A\underline{a}$ $A \to ABC$ $A \to \varepsilon$	$A \to \varepsilon$
	B		$B \to \underline{b}$		$B \to \underline{q}$	
	C	$C \to A\underline{c}$	$C \to A\underline{c}$	$C \to A\underline{c}$	$C \to A\underline{c}$	

Definition 1.21: Eine Grammatik ist eine LL(1)-Grammatik, wenn die berechnete Parse-Tabelle keine Mehrfacheinträge hat.

Definition 1.22: Eine LL(1)-Parsing Tabelle (oder auch Parsing-Tabelle) stellt einen vollständigen Parse-Vorgang dar. Jede Zeile entspricht einem Bearbeitungsschritt im Parse-Vorgang. In Spalten werden Stack, Eingabe und die angewandte Produktionsregel aufgetragen.

Algorithmus (LL(1)-Parsing mit Tabelle)**:** Sei X das oberste Stack-Element, t das aktuelle Token der Eingabe w und $\mathcal{L}$ die von der Grammatik akzeptierte Sprache.

1. Wenn X ein Non-Terminal ist:
   (a) Nimm den Wert von $M(X, t)$
   (b) Ist der Eintrag leer oder ein Fehlereintrag: Abbruch ($w \notin \mathcal{L}$).
   (c) Sonst: Ersetze das oberste Stack-Element X durch Produktion in umgekehrter Reihenfolge (WVU wenn $M(X, t) = X \to UVW$).
2. Andernfalls (X ist ein Terminal):
   (a) Wenn $X = t =$ \$: Parsing erfolgreich ($w \notin \mathcal{L}$).
   (b) Sonst, wenn $X = t \neq$ \$, dann nimm X vom Stack und gehe zum nächsten Token im Input.
   (c) Sonst: Abbruch ($w \notin \mathcal{L}$).

Diese Definition kann für andere Parser leicht angepasst werden.

Beispiel 1.8: Gegeben Sei die folgende Grammatik (ähnlich der Grammatik aus dem vorherigen Beispiel):

$$\begin{aligned} A &\to A\underline{a} \\ A &\to ABC \\ A &\to \varepsilon \\ B &\to \underline{b}|\underline{q} \\ C &\to A\underline{c} \end{aligned}$$

Zeigen oder widerlegen Sie dass das Wort $\underline{abbqa}$ von der Grammatik akzeptiert wird. Zeigen oder widerlegen Sie dass das Wort $\underline{abc}$ von der Grammatik akzeptiert wird.

Lösung: Wir formen die Grammatik zu einer LL(1)-Grammatik um, berechnen dann die LL(1)-Tabelle und beweisen mittels einer Parsing-Tabelle dass das gegebene Wort von der Grammatik akzeptiert wird.

$$\begin{aligned} A &\to A\underline{a} \\ A &\to ABC \\ A &\to \varepsilon \\ B &\to \underline{b}|\underline{q} \\ C &\to A\underline{c} \end{aligned} \quad \Rightarrow \quad \begin{aligned} A &\to R \\ R &\to \underline{a}R \\ R &\to BCR \\ R &\to \varepsilon \\ B &\to \underline{b}|\underline{q} \\ C &\to A\underline{c} \end{aligned} \quad \Rightarrow \quad \begin{aligned} A &\to \underline{a}A \\ A &\to BCA \\ A &\to \varepsilon \\ B &\to \underline{b}|\underline{q} \\ C &\to A\underline{c} \end{aligned}$$

Anhand der FIRST- und FOLLOW-Mengen ist bereits erkennbar dass die Mehrdeutigkeiten behoben sind.

$$\begin{aligned} \text{FIRST}(B) &= \{\underline{b}, \underline{q}\} \\ \text{FIRST}(A) &= \{\underline{a}\} \cup \text{FIRST}^*(B) \cup \{\varepsilon\} &&= \{\underline{a}, \underline{b}, \underline{q}, \varepsilon\} \\ \text{FIRST}(C) &= \text{FIRST}^*(A) \cup \{\underline{c}\} &&= \{\underline{a}, \underline{b}, \underline{q}, \underline{c}\} \end{aligned}$$

$$\begin{aligned} \text{FOLLOW}(A) &= \{\$\} \cup \text{FOLLOW}(A) \cup \{\underline{c}\} &&= \{\$, \underline{c}\} \\ \text{FOLLOW}(B) &= \text{FIRST}^*(C) &&= \{\underline{a}, \underline{b}, \underline{q}, \underline{c}\} \\ \text{FOLLOW}(C) &= \text{FIRST}^*(A) \cup \text{FOLLOW}(A) &&= \{\underline{a}, \underline{b}, \underline{q}, \$, \underline{c}\} \end{aligned}$$

Wir können nun die LL(1)-Tabelle berechnen. Dazu führen wir im ersten Schritt die Regeln für die FIRST-Menge des Non-Terminals A durch:

	$\underline{a}$	$\underline{b}$	$\underline{c}$	$\underline{q}$	\$
A					
B					
C					

$\to$

	$\underline{a}$	$\underline{b}$	$\underline{c}$	$\underline{q}$	\$
A	$A \to \underline{a}A$	$A \to BCA$		$A \to BCA$	
B					
C					

Da ε in der FIRST-Menge ist, führen wir im zweiten Schritt die Regeln für die FOLLOW-Menge des Non-Terminals A durch:

$\to$

	$\underline{a}$	$\underline{b}$	$\underline{c}$	$\underline{q}$	\$
A	$A \to \underline{a}A$	$A \to BCA$	$A \to \varepsilon$	$A \to BCA$	$A \to \varepsilon$
B					
C					

Nun noch B und C:

		$\underline{a}$	$\underline{b}$	$\underline{c}$	$\underline{q}$	\$
$\rightarrow$	A	$A \rightarrow \underline{a}A$	$A \rightarrow BCA$	$A \rightarrow \varepsilon$	$A \rightarrow BCA$	$A \rightarrow \varepsilon$
	B		$B \rightarrow \underline{b}$		$B \rightarrow \underline{q}$	
	C	$C \rightarrow A\underline{c}$	$C \rightarrow A\underline{c}$	$C \rightarrow A\underline{c}$	$C \rightarrow A\underline{c}$	

Diesmal haben wir keine Mehrfach-Einträge, die Grammatik ist also eine LL(1)-Grammatik. Die Parsing-Tabelle für $\underline{abbqa}$ sieht dann wie folgt aus:

Stack	**Input**	**Produktion**
\$$A$	$\underline{abbqa}$\$	$A \rightarrow \underline{a}A$
\$$AA\underline{a}$	$\underline{abbqa}$\$	
\$$AA$	$\underline{bbqa}$\$	$A \rightarrow BCA$
\$$AAACB$	$\underline{bbqa}$\$	$B \rightarrow \underline{b}$
\$$AAAC\underline{b}$	$\underline{bbqa}$\$	
\$$AAAC$	$\underline{bqa}$\$	$C \rightarrow A\underline{c}$
\$$AAA\underline{c}A$	$\underline{bqa}$\$	$A \rightarrow BCA$
\$$AAA\underline{c}ACB$	$\underline{bqa}$\$	$B \rightarrow \underline{b}$
\$$AAA\underline{c}AC\underline{b}$	$\underline{bqa}$\$	
\$$AAA\underline{c}AC$	$\underline{qa}$\$	$C \rightarrow A\underline{c}$
\$$AAA\underline{c}A\underline{c}A$	$\underline{qa}$\$	$A \rightarrow BCA$
\$$AAA\underline{c}A\underline{c}ACB$	$\underline{qa}$\$	$B \rightarrow \underline{q}$
\$$AAA\underline{c}A\underline{c}AC\underline{q}$	$\underline{qa}$\$	
\$$AAA\underline{c}A\underline{c}AC$	$\underline{a}$\$	$C \rightarrow A\underline{c}$
\$$AAA\underline{c}A\underline{c}A\underline{c}A$	$\underline{a}$\$	$A \rightarrow \underline{a}A$
\$$AAA\underline{c}A\underline{c}A\underline{c}A\underline{a}$	$\underline{a}$\$	
\$$AAA\underline{c}A\underline{c}A\underline{c}A$	\$	$A \rightarrow \varepsilon$
\$$AAA\underline{c}A\underline{c}A\underline{c}$	\$	FAIL

Wir haben somit **widerlegt** dass das Wort $\underline{abc}$ von der gegebenen Grammatik akzeptiert wird.

Stack	**Input**	**Produktion**
\$$A$	$\underline{abc}$\$	$A \rightarrow \underline{a}A$
\$$AA\underline{a}$	$\underline{abc}$\$	
\$$AA$	$\underline{bc}$\$	$A \rightarrow BCA$
\$$AAACB$	$\underline{bc}$\$	$B \rightarrow \underline{b}$
\$$AAAC\underline{b}$	$\underline{bc}$\$	
\$$AAAC$	$\underline{c}$\$	$C \rightarrow A\underline{c}$
\$$AAA\underline{c}A$	$\underline{c}$\$	$A \rightarrow \varepsilon$
\$$AAA\underline{c}$	$\underline{c}$\$	
\$$AAA$	\$	$A \rightarrow \varepsilon$
\$$AA$	\$	$A \rightarrow \varepsilon$
\$$A$	\$	$A \rightarrow \varepsilon$
\$	\$	ACCEPT

Und mit dieser Parsing-Tabelle haben wir **gezeigt** dass das Wort *abc* von der gegebenen Grammatik akzeptiert wird.

Kapitel 2

Semantik von Programmiersprachen

Im ersten Kapitel haben wir uns damit beschäftigt wie ein Wort (Programm) einer Sprache eindeutig geparst werden kann. Die Wörter (Programme) haben jedoch noch keine Bedeutung. Wir wollen uns nun damit beschäftigen Sprachen eine Bedeutung zu geben und Sprachen anhand der Bedeutung der Wörter zu unterscheiden. Im Kontext der Semantik verwenden wir vermehrt den Begriff "Programm einer Programmiersprache" anstatt "Wort einer Sprache".

Im zweiten Kapitel betrachten wir nur noch syntaktisch korrekte Eingaben, d.h. wir betrachten den Fall nachdem der Parser bereits entschieden hat, dass eine Eingabe ein syntaktisch gültiges Programm ist.

Wir teilen dazu Sprachen hauptsächlich in funktionale, imperative und logische Sprachen. Zu jedem dieser drei Sprachparadigmen werden wir Sprachen konstruieren und deren Semantik definieren.

Sowohl für die Definition der Semantik als auch für die Interpretation eines konkreten Programms in einer Sprache, werden wir mathematische Funktionen definieren: die Interpretationsfunktion sowie weitere Hilfsfunktionen. Diese mathematische Definition wird es uns erlauben die Korrektheit unserer Programme zu beweisen.

Um Syntax und Semantik zu unterscheiden werden wir Programme einer Sprache wie bisher unterstreichen. Die Beschreibung der Semantik ist kein Programm und wird daher auch keinesfalls unterstrichen.

Beispiel 2.1: Was drückt der Ausdruck $\underline{\text{a = b + c}}$ aus? (vgl. Vorlesungsskriptum Seite 42)

Es gibt einige mögliche Interpretationen, hier eine Auswahl davon:

1. Imperative Interpretation: Eine Zuweisung wie in *C*. $\underline{\text{a}}$ hat nach der Ausführung

des Ausdrucks den Wert der Summe der Werte von b und c. Andere Variante: Der Wert von a ist nach der Zuweisung die Zeichenfolge b + c.

2. Funktionale Interpretation: Eine Funktion a wird mit den 4 Parametern = b + c aufgerufen.

3. Logische Interpretation: Ein logischer Ausdruck, beispielsweise ist der Ausdruck Wahr wenn der Wert von a gleich der Summe der Werte von b und c ist. Andere Variante: Der Wert von 2 der 3 Variablen ist bekannt, der Wert der 3. Variable wird so festgelegt.

Wir sehen anhand dieses Beispiels dass es wichtig ist exakt zu definieren wie ein Ausdruck zu interpretieren ist.

In funktionalen Programmiersprachen besteht jedes Programm aus einer oder mehreren Funktionen.

Definition 2.1 (Funktion)**:** Eine Funktion ist eine Relation zwischen einer Menge A und einer Menge B. Jedem Element aus der Menge A wird genau ein Element der Menge B zugeordnet. Das heißt: Für jeden möglichen Eingabewert gibt es genau einen Ausgabewert.

2.1 Sprache $\mathcal{A}$ - einfache arithmetische Ausdrücke

Arithmetische Ausdrücke sind Funktionen. Wir können beispielsweise die Funktionen Addition, Subtraktion und Multiplikation von zwei Zahlen in $\mathbb{R}$ definieren mit einem Eingabewert in $\mathbb{R} \times \mathbb{R}$ und einen Ausgabewert in $\mathbb{R}$. Auch die Division können wir als Funktion definieren von $\mathbb{R} \times \mathbb{R} \setminus \{0\}$ (Division durch 0 schließen wir damit aus, da die Division in diesem Fall nicht als Funktion definiert ist) auf Ausgabewerte in $\mathbb{R}$.

Definition 2.2: Die Sprache $\mathcal{A}$ definieren wir mit Alphabet $\Sigma = \left\{\underline{0}, \ldots, \underline{9}, \underline{(}, \underline{)}, \underline{+}\right\}$. Zwecks Einfachheit definieren wir Ziffern (D, digits) und Zahlen:

- $\mathcal{A}_D = \{\underline{0}, \ldots, \underline{9}\}$
- ZAHL $= (\mathcal{A}_D \setminus \{\underline{0}\}\, \mathcal{A}_D^*) \cup \{\underline{0}\}$

Wir definieren die Sprache A nun nicht mehr über eine Grammatik sondern durch eine induktive Beschreibung (Basisfall und allgemeine Fälle):

1. ZAHL$\subset \mathcal{A}$
2. Wenn $x, y \in \mathcal{A}$, dann ist auch $\underline{(}x\underline{)}\underline{+}\underline{(}y\underline{)} \in \mathcal{A}$.

An dieser Stelle sei noch einmal darauf hingewiesen dass wir x, y nicht unterstreichen dürfen, da sie keine Sprachelemente sind sondern Platzhalter, mathematisch würde man sie auch als Variablen bezeichnen. Wir wollen nun mit unserer Sprache $\mathcal{A}$ Ausdrücke

berechnen können. Dazu definieren wir eines unserer mächtigsten Werkzeuge im zweiten Kapitel: Die Interpretationsfunktion I (auch genannt Semantikfunktion). Man kann sich diese Funktion vorstellen wie einen Interpreter einer Scriptsprache: Wir geben ein Programm ein und führen es aus, abhängig vom aktuellen Zustand liefert uns der Interpreter ein Ergebnis zurück. Genau so soll unsere Interpretationsfunktion arbeiten. Wir erwarten einen Eingabewert aus $\mathcal{A}$ und bilden auf $\mathbb{N}_0$ ab, d.h. geben einen Wert aus $\mathbb{N}_0$ zurück. Genau wie die Syntax werden wir nun die Semantik induktiv durch die Interpretationsfunktion definieren.

Definition 2.3: Die Semantik der Sprache $\mathcal{A}$ definieren wir durch:

1. $I_{\mathcal{A}}(x) = \langle x \rangle$ wenn $x \in \mathcal{A}_N$. x ist dabei eine Zeichenkette im Programm, $\langle x \rangle$ die entsprechende Repräsentation in $\mathbb{N}_0$.
2. $I_{\mathcal{A}}(\underline{(}x\underline{)} + \underline{(}y\underline{)}) = I_{\mathcal{A}}(x) + I_{\mathcal{A}}(y)$ wenn $x, y \in \mathcal{A}$.

Beispiel 2.2: Das Programm $\underline{((10) + (9)) + (3)}$ (vgl. Vorlesungsskriptum Seite 45) können wir wie folgt interpretieren:

$$\begin{aligned} I_{\mathcal{A}}(\underline{((10) + (9)) + (3)}) &= I_{\mathcal{A}}(\underline{(10) + (9)}) + I_{\mathcal{A}}(\underline{3}) \quad \text{(entsprechend 2. Fall der Definition)} \\ &= I_{\mathcal{A}}(\underline{10}) + I_{\mathcal{A}}(\underline{9}) + 3 \quad \text{(beim } \underline{3} \text{ nun der 1. Fall der Definition)} \\ &= 10 + 9 + 3 = 22 \end{aligned}$$

Auch hier sehen wir wieder deutlich die Unterscheidung zwischen Zeichenketten im Programmcode (unterstrichen) und den Werten auf der semantischen Ebene (nicht unterstrichen). Um den Unterschied weiter zu verdeutlichen definieren wir nun die Sprache der einfachen arithmetischen Ausdrücke von Binärzahlen $\mathcal{B}$.

Definition 2.4: Die Sprache $\mathcal{B}$ definieren wir mit Alphabet $\Sigma = \left\{\underline{0}, \underline{1}, \underline{(}, \underline{)}, \underline{+}\right\}$.

1. $(\underline{1}\,\{\underline{0}, \underline{1}\}^*) \cup \{\underline{0}\} \subset \mathcal{B}$
2. Wenn $x, y \in \mathcal{B}$, dann ist auch $\underline{(}x\underline{)}\underline{+}\underline{(}y\underline{)} \in \mathcal{B}$.

Die Semantik der Sprache $\mathcal{B}$ definieren wir durch:

1. $I_{\mathcal{B}}(x) = \langle x \rangle$ wenn $x \in \mathcal{B}_N$. $\langle x \rangle \in \mathbb{N}_0$ ist nun die durch die Binärzahl (exakt: die Binärziffernfolge) dargestellte Zahl auf semantischer Ebene, in diesem Fall also im mathematischen Sinne.
2. $I(\underline{(}x\underline{)} + \underline{(}y\underline{)}) = I(x) + I(y)$ wenn $x, y \in \mathcal{B}$.

Beispiel 2.3: $I(\underline{1001}) = 9$ aber $\underline{1001} \neq 9$.

Betrachten wir das Beispiel wie zuvor, nun in Binärdarstellung $\underline{((1010) + (1001)) + (11)}$:

$$\begin{aligned} I_{\mathcal{B}}(\underline{((1010) + (1001)) + (11)}) &= I_{\mathcal{B}}(\underline{(1010) + (1001)}) + I_{\mathcal{B}}(\underline{11}) \\ &= I_{\mathcal{B}}(\underline{1010}) + I_{\mathcal{B}}(\underline{1001}) + 3 \\ &= 10 + 9 + 3 = 22 \end{aligned}$$

Wir versuchen nun der Sprache $\mathcal{A}$ eine zweite Funktion, die Multiplikation hinzuzufügen.

Definition 2.5: Die Sprache $\mathcal{C}$ ist definiert durch:

1. ZAHL$\subset \mathcal{C}$
2. $\underline{(}x\underline{)}\underline{+}\underline{(}y\underline{)} \in \mathcal{C}$, wenn $x, y \in \mathcal{C}$.
3. $\underline{(}x\underline{)}\underline{*}\underline{(}y\underline{)} \in \mathcal{C}$, wenn $x, y \in \mathcal{C}$.

Die Semantik der Sprache $\mathcal{C}$ definieren wir durch:

1. $I_{\mathcal{C}}(x) = \langle x \rangle$ wenn $x \in \mathcal{C}_N$.
2. $I_{\mathcal{C}}(\underline{(}x\underline{)} * \underline{(}y\underline{)}) = I_{\mathcal{C}}(x) \cdot I_{\mathcal{C}}(y)$ wenn $x, y \in \mathcal{A}$.
3. $I_{\mathcal{C}}(\underline{(}x\underline{)} + \underline{(}y\underline{)}) = I_{\mathcal{C}}(x) + I_{\mathcal{C}}(y)$ wenn $x, y \in \mathcal{A}$.

Mit dieser Definition ist $I_{\mathcal{C}}$ keine Funktion.

Beweis: Laut Definition 2.1 ist eine Relation eine Funktion wenn es für jeden möglichen Eingabewert genau einen Ausgabewert gibt.

Möchte man eine Aussage über "alle" Werte bzw. "jeden" Wert widerlegen so gestaltet sich ein Beweis oft relativ einfach. In so einem Fall müssen wir nur ein Gegenbeispiel finden, denn dann gilt die Aussage offensichtlich nicht für alle Werte, wir haben ja einen gefunden für den es nicht gilt. Diese Beweistechnik nennt man "Beweis durch Widerspruch".

Wir werden nun zeigen dass $I_{\mathcal{C}}$ für das Programm $\underline{1+2*3}$ verschiedene Interpretationsmöglichkeiten zulässt da nicht festgelegt ist ob der 2. oder 3. Fall der Definition die höhere Priorität hat.

$$
\begin{aligned}
I_{\mathcal{C}}(\underline{1+2*3}) &= I_{\mathcal{C}}(\underline{1+2}) \cdot I_{\mathcal{C}}(\underline{3}) && \text{(2. Fall der Definition)} \\
&= (I_{\mathcal{C}}(\underline{1}) + I_{\mathcal{C}}(\underline{2})) \cdot 3 && \text{(3. Fall der Definition)} \\
&= (1+2) \cdot 3 = 3 \cdot 3 = 9 \\
I_{\mathcal{C}}(\underline{1+2*3}) &= I_{\mathcal{C}}(\underline{1}) \cdot I_{\mathcal{C}}(\underline{2*3}) && \text{(3. Fall der Definition)} \\
&= 1 + (I_{\mathcal{C}}(\underline{2}) \cdot I_{\mathcal{C}}(\underline{3})) && \text{(2. Fall der Definition)} \\
&= 1 + (2 \cdot 3) = 1 + 6 = 7 \neq 9
\end{aligned}
$$

Wir haben gezeigt dass für einen Eingabewert 2 unterschiedliche Ausgabewerte möglich sind. Folglich gibt es nicht für jeden Eingabewert genau einen Ausgabewert, daher kann $I_{\mathcal{C}}$ keine Funktion sein. □

Wir müssten also die Interpretationsfunktion $I_{\mathcal{C}}$ anders definieren. Eine Lösung wäre beispielsweise zu definieren dass der 3. Fall der Interpretationsfunktion nur angewendet werden darf wenn in den beiden Operanden x und y kein $\underline{*}$ vorkommt.

2.2 Sprache $\mathcal{VA}$ - arithmetische Ausdrücke mit Variablen

Wir erweitern die Sprache $\mathcal{A}$ durch Variablen und schaffen so eine mächtigere Sprache $\mathcal{VA}$. Um mit Variablen umgehen zu können brauchen wir nun einerseits eine Menge zulässiger Variablennamen und andererseits eine Funktion die von Variablennamen auf eine Wertemenge der semantischen Ebene (z.B. $\mathbb{N}_0$) abbildet. Die Menge der zulässigen Variablennamen nennen wir IVS (Individuenvariablensymbole).

Definition 2.6: Zwecks Einfachheit erlauben wir nur wenige Variablennamen und definieren daher

$$\text{IVS} = \{\underline{a}, \underline{b}, \ldots, \underline{z}\} \cup \{\underline{x1}, \underline{x2}, \ldots\}\,.$$

Die Funktion die von Variablennamen auf eine Wertemenge abbildet nennen wir ω-Environment, (Variablen-)Umgebung. Man kann sich diese Funktion auch als Tabelle vorstellen bzw. in einem Interpreter als Tabelle implementieren.

Definition 2.7: Die Menge aller Environments sei

$$\text{ENV} = \bigcup_{x \in \text{IVS}, y \in \Lambda} \{(x, y)\}\,,$$

das heißt, die Vereinigung über alle Tupel Variablenname $x \in$ IVS und Wert auf semantischer Ebene $y \in \Lambda$.

Für die Sprache $\mathcal{VA}$ ist $\Lambda = \mathbb{N}_0$.

Definition 2.8: Die Syntax der Sprache $\mathcal{VA}$ ist definiert durch:

1. ZAHL$\subset \mathcal{VA}$
2. IVS$\subset \mathcal{VA}$
3. $\underline{(}x\underline{)}\underline{+}\underline{(}y\underline{)} \in \mathcal{C}$, wenn $x, y \in \mathcal{VA}$.

Die Interpretation eines Programms hängt nun nicht mehr allein vom Programm selbst ab, sondern auch von den Werten der Variablen im ω-Environment.

Definition 2.9: Die Interpretationsfunktion $I_{\mathcal{VA}} : \text{ENV} \times \mathcal{VA} \to \Lambda$ weist jedem Tupel aus Environment und Programm einen Wert in Λ zu.

1. $I_{\mathcal{VA}}(\omega, k) = \langle k \rangle$ wenn $k \in$ ZAHL, $\omega \in$ ENV.
2. $I_{\mathcal{VA}}(\omega, v) = \omega(v)$ wenn $vk \in$ IVS, $\omega \in$ ENV.
3. $I_{\mathcal{VA}}(\underline{(}x\underline{)} \underline{+} \underline{(}y\underline{)}) = I_{\mathcal{VA}}(\omega, x) + I_{\mathcal{VA}}(\omega, y)$ wenn $x, y \in \mathcal{VA}$, $\omega \in$ ENV.

Beispiel 2.4: Gegeben Sei das Environment $\omega(\underline{x}) = 0$, $\omega(\underline{y}) = 1$, $\omega(\underline{z}) = 2$. Interpretieren Sie das Programm $\underline{(((x) + (2)) + (y)) + (z)}$.

$$\begin{aligned} I_{\mathcal{VA}}(\omega, \underline{(((x) + (2)) + (y)) + (z)}) &= I_{\mathcal{VA}}(\omega, \underline{((x) + (2)) + (y)}) + I_{\mathcal{VA}}(\omega, \underline{z}) \\ &= I_{\mathcal{VA}}(\omega, \underline{(x) + (2)}) + I_{\mathcal{VA}}(\omega, \underline{y}) + \omega(\underline{z}) \\ &= I_{\mathcal{VA}}(\omega, \underline{x}) + I_{\mathcal{VA}}(\omega, \underline{2}) + \omega(\underline{y}) + 2 \\ &= \omega(\underline{x}) + 2 + 1 + 2 \\ &= 0 + 2 + 1 + 2 = 5 \end{aligned}$$

Beachten Sie auch, dass nach wie vor $I_{\mathcal{VA}}(\omega, \underline{2}) = 2$. Es ist ein bei den Übungen weit verbreiteter Fehler $I_{\mathcal{VA}}(\omega, \underline{2}) = \omega(\underline{2})$ zu schreiben. Die Interpretationsfunktion wurde so nicht definiert und außerdem ist $\underline{2}$ auch kein gültiger Variablenname.

2.3 Datentypen

Bisher haben wir eine Sprache nur für einen Datentypen definiert. Dieser war implizit in der Definition der Sprache drin (beispielsweise die natürlichen Zahlen). Derartige Definitionen erlauben kein Ersetzen des Datentyps ohne die Definition der Sprache wesentlich zu überarbeiten. Da wir dies aber häufig wollen werden wir nun zuerst Datentypen auf der semantischen Ebene und anschließend die Repräsentation von Datentypen auf der syntaktischen Ebene definieren.

Definition 2.10 (Datentyp)**:** Ein Datentyp ist ein Tupel $\Psi = (A, F, P, C)$ mit

- A: Grundmenge (Wertebereich)
- F: Menge von Funktionen $f_i : A^{k_i} \to A^{l_i}$.

  f_i ist die i-te Funktion in der Menge, k_i die Dimension vom Urbild (Dimension des Inputs, Anzahl der Funktionsargumente) und l_i die Dimension vom Bild der i-ten Funktion (Dimension des Outputs, Anzahl der Funktionsrückgabewerte).
- P: Menge von Prädikaten $p_i : A^{k_i} \to \{T, F\}$.

  p_i ist die i-te Funktion in der Menge und k_i die Dimension vom Urbild (Dimension des Inputs, Anzahl der Funktionsargumente) der i-ten Funktion.
- C: Menge von Konstanten c_i wobei $c \subseteq A$.

Die Mengen F^Σ, P^Σ und C^Σ enthalten die entsprechenden Symbole für die syntaktische Repräsentation:

- Funktionssymbole F^Σ: je ein Symbol f_i^Σ (z.B. Name der Funktion) für jede Funktion f_i
- Prädikatensymbole P^Σ: je ein Symbol p_i^Σ (z.B. Name des Prädikats) für jedes Prädikat p_i

- Konstantensymbol C^{Σ}: je ein Symbol c_i^{Σ} (z.B. ausgeschriebene Form der Konstante) für jede Konstante c_i

Konstanten sind eigentlich nur spezielle Funktionen (0 Argumente) und wir unterscheiden nur zwecks Übersicht.

Beispiel 2.5: Definieren Sie den Datentyp Integer mit Funktionen für Addition, Subtraktion und Multiplikation sowie Prädikaten für "kleiner" und Gleichheit.

Lösung: Im Fall der Integer ist die Definition der Funktionen und Prädikate trivial, da alle Funktionen und Prädikate durch die entsprechenden Operationen auf den ganzen Zahlen $\mathbb{Z}$ definiert sind. Daher genügt es zu definieren welche Funktion welcher Operation entspricht.

- Grundmenge $A = \mathbb{Z}$
- Funktionen $f_1 : +$, $f_2 : -$, $f_3 : *$

  Die Funktionen $+, -, *$ sind auf $\mathbb{Z}$ definiert. Auf der syntaktischen Ebene definieren wir: $f_1^{\Sigma} : \underline{\text{plus}}$, $f_2^{\Sigma} : \underline{\text{minus}}$, $f_3^{\Sigma} : \underline{\text{mult}}$.
- Prädikate $p_1 :<$, $p_2 :=$

  Die Prädikate $<, =$ sind auf $\mathbb{Z}$ definiert. Syntaktische Ebene: $p_1^{\Sigma} : \underline{\text{lt?}}$, $p_2^{\Sigma} : \underline{\text{eq?}}$.
- Konstanten $c_1 : 0$, $c_2 : 1$

  Syntaktische Ebene: $c_1^{\Sigma} : \underline{\text{null}}$, $c_2^{\Sigma} : \underline{\text{eins}}$.

Hinzunahme von Division ist problematisch / da das Ergebnis nicht unbedingt $\in A$ ist.

Wir können bei der Definition eines Datentyps oft (z.B. bei den verschiedenen Datentypen für Zahlen) auf bekannte algebraische Strukturen (Halbgruppen, etc.) zurückgreifen.

Beispiel 2.6: Definieren Sie den Datentyp String mit der Funktion Konkatenation und dem Prädikat "Präfix".

Lösung: Hier können wir nun nicht mehr auf eine vorhandene mathematische Definition zurückgreifen.

- Grundmenge $A = V^*$ mit V einem endlichen Alphabet $\{v_1, \ldots, v_n\}$ (z.B. dem ASCII-Alphabet).
- Funktionen
  1. $f_1 : \circ$ (Konkatenation)
     - Wenn $x \in V^*$ ist, dann ist $x \circ \varepsilon = x$
     - Wenn $x \in V^*$ und $a \in V$ ist, dann ist $x \circ a = xa$
     - Wenn $x, y \in V^*$ und $a \in V$ ist, dann ist $x \circ (y \circ a) = (x \circ y) \circ a$

- Prädikate
  1. $p_1 :<<$ (Präfix)
     - Wenn $x \in V^*$ ist, dann ist $\varepsilon << x$
     - Wenn $x \in V^*$ ist, dann ist $x << x$
     - Wenn x Präfix von y ist, dann ist x auch Präfix von $y \circ z$ $((x << y) \rightarrow (x << (y \circ z)))$.
- Konstanten $c_i : v_i$, $c_{n+1} : \varepsilon$ (eine Konstante für jeden Buchstaben des Alphabets und ε für den Leerstring).

Die syntaktische Ebene überlassen wir dem Leser.

Beispiel 2.7: Definieren Sie den Datentyp des binären Stacks mit Funktionen um Elemente auf den Stack zu legen oder herunterzunehmen, sowie Prädikaten zur Überprüfung des obersten Elementes.

Lösung:

- Grundmenge $A = \{0, 1\}^* \cup \{\varepsilon\}$, d.h. Ziffernfolgen aus 0 und 1 oder ε (leerer Stack).
- Funktionen
  1. f_1 : add0 liefert den Stack mit einer 0 daraufgelegt.
     - $\text{add0}(\varepsilon) = 0$
     - $\text{add0}(x) = 0x$ (mit $x \neq \varepsilon$)
  2. f_2 : add1 liefert den Stack mit einer 1 daraufgelegt.
     - $\text{add1}(\varepsilon) = 1$
     - $\text{add1}(x) = 1x$ (mit $x \neq \varepsilon$)
  3. f_3 : sub liefert den Stack ohne das oberste Element.
     - $\text{sub}(\varepsilon) = \varepsilon$
     - $\text{sub}(ax) = x$ (mit $a \neq \varepsilon$)
- Prädikate
  1. p_1 : ist0? testet ob das oberste Element 0 ist.
     - $\text{ist0?}(x) \Leftrightarrow \exists z : x = 0z$
  2. p_2 : ist1? testet ob das oberste Element 1 ist.
     - $\text{ist1?}(x) \Leftrightarrow \exists z : x = 1z$
  3. p_3 : istLeer? testet ob das oberste Element ε ist.

– istLeer?$(x) \Leftrightarrow x = \varepsilon$

- Konstanten $c_1 : \varepsilon$.

Auf der syntaktischen Ebene werden die gleichen Bezeichnungen wie auf der semantischen Ebene verwendet.

Beispiel 2.8: Berechnen Sie den Ausdruck add0(add1(sub(011))) im Datentyp des binären Stacks.

Lösung:

$$\begin{aligned} &\text{add0(add1(sub(011)))} \\ =\ &\text{add0(add1(11))} \\ =\ &\text{add0(111)} \\ =\ &\text{0111} \end{aligned}$$

$$\begin{aligned} \text{ist0?(0111)} &= T \\ \text{ist1?(0111)} &= F \\ \text{istLeer?(0111)} &= F \end{aligned}$$

2.4 Sprache der Terme $\mathcal{T}$

Wir müssen nun um den Datentyp verwenden zu können eine grundlegende Sprache definieren die diesen Datentyp verwendet. Auf dieser Sprache können dann weitere Sprachen aufgebaut werden.

Definition 2.11 (Sprache der Terme $\mathcal{T}$)**:** Sei $\Psi = (A, F, P, C)$ ein Datentyp. Das Alphabet Σ ist dann eine Vereinigung aus den Mengen der

- IVS (Individuenvariablensymbole)
- Funktionssymbole F^Σ
- Prädikatensymbole P^Σ
- Konstantensymbol C^Σ
- $\underline{(}$, $\underline{)}$ und $\underline{,}$
- Sondersymbole (Keywords): if, then, else, begin, end, ...

Die Syntax von $\mathcal{T} \subseteq \Sigma$ über einem beliebigen Datentypen ist dann definiert durch:

1. $C^\Sigma \subseteq \mathcal{T}$, d.h. Konstantensymbole sind Terme
2. IVS $\subseteq \mathcal{T}$, d.h. Individuenvariablensymbole sind Terme

3. Wenn f_i^{Σ} ein n-stelliges Funktionssymbol ist und $t_1, \ldots, t_n$ Terme, dann ist auch $f_i^{\Sigma}\underline{(}t_1\underline{,}\ldots\underline{,}t_n\underline{)}$ ein Term (Unterstreichungen beachten!).

Die Semantik von $\mathcal{T}$ definieren wir durch die Interpretationsfunktion $I_{\mathcal{T}} : \text{ENV} \times \mathcal{T} \to A$.

1. $I_{\mathcal{T}}(\omega, c_i') = c_i$ mit $c_i \in C$ (Semantik-Ebene), $c_i' \in C^{\Sigma}$ (Syntax-Ebene) und $\omega \in$ ENV.

2. $I_{\mathcal{T}}(\omega, v) = \omega(v)$ mit $v \in$ IVS und $\omega \in$ ENV.

3. $I_{\mathcal{T}}(\omega, f_i'\underline{(}t_1\underline{,}\ldots\underline{,}t_n\underline{)}) = f_i(I_{\mathcal{T}}(\omega, t_1), \ldots, I_{\mathcal{T}}(\omega, t_n))$ mit $f_i \in F$ (Semantik-Ebene), $f_i' \in F^{\Sigma}$ (Syntax-Ebene) und $\omega \in$ ENV.

Beispiel 2.9: Führen Sie das Programm $\underline{\text{plus(plus(x,y),plus(eins,z))}}$ mit dem Environment $\omega(\underline{x}) = 0$, $\omega(\underline{y}) = 1$, $\omega(\underline{z}) = 2$ aus.

Lösung:

$$
\begin{aligned}
I_{\mathcal{T}}\left(\omega, \underline{\text{plus}\left(\text{plus}(x,y), \text{plus}(\text{eins}, z)\right)}\right) &= +\left(I_{\mathcal{T}}\left(\omega, \underline{\text{plus}(x,y)}\right), I_{\mathcal{T}}\left(\omega, \underline{\text{plus}(\text{eins}, z)}\right)\right) \\
&= +\left(+\left(I_{\mathcal{T}}\left(\omega, \underline{x}\right), I_{\mathcal{T}}\left(\omega, \underline{y}\right)\right), +\left(I_{\mathcal{T}}\left(\omega, \underline{\text{eins}}\right), I_{\mathcal{T}}\left(\omega, \underline{z}\right)\right)\right) \\
&= +\left(+\left(\omega(\underline{x}), \omega(\underline{y})\right), +\left(1, \omega(\underline{z})\right)\right) \\
&= +\left(+\left(0,1\right), +\left(1,2\right)\right) = +\left(1,3\right) = 4
\end{aligned}
$$

Laut Definition von $\mathcal{T}$ gibt es nur die Konstanten 0 und 1. Variablen können natürlich jeden beliebigen Wert in $\mathbb{Z}$ annehmen. Es kann aber auch gezeigt werden dass alle ganzen Zahlen durch einen variablenfreien Term dargestellt werden können. Terme sind rekursiv definiert. Die **vollständige Induktion** ist die übliche Beweistechnik für Beweise über rekursive bzw. rekursiv definierte Ausdrücke.

Beispiel 2.10: Wir wollen zeigen, dass alle ganzen Zahlen durch einen variablenfreien Term dargestellt werden können. Die **vollständige Induktion** besteht aus 3 Einzelschritten: In der **Induktionsbasis** werden ein oder mehrere Basisfälle direkt bewiesen. In der **Induktionshypothese** wird versucht eine allgemeine Aussage (eine Hypothese) zu treffen von der angenommen wird dass sie bis zum n-ten Fall gilt. Im **Induktionsschritt** gehen wir einen Schritt weiter, also in den Fall $n+1$ und versuchen diesen zu beweisen. Hier muss unbedingt auf die Induktionshypothese zurückgegriffen werden, sonst wurde keine vollständige Induktion durchgeführt.

Um eine Induktion durchführen zu können müssen die Elemente unbedingt aufgezählt werden können (d.h. man muss eine eindeutige Reihenfolge/Sortierung für die Elemente angeben können).

Bevor wir die Induktion durchführen versuchen wir ein Muster zu erkennen.

$$I_{\mathcal{T}}(\omega, \underline{\text{null}}) = 0 \qquad \text{(trivial)}$$
$$I_{\mathcal{T}}(\omega, \underline{\text{eins}}) = 1 \qquad \text{(trivial)}$$
$$I_{\mathcal{T}}\Big(\omega, \underline{\text{plus}(\text{eins}, \text{eins})}\Big) = 2 \qquad \text{(eine Addition)}$$
$$I_{\mathcal{T}}\Big(\omega, \underline{\text{plus}(\text{eins}, \text{plus}(\text{eins}, \text{eins}))}\Big) = 3 \qquad \text{(verschachtelte Addition)}$$

Wir erkennen das Muster: Alle Zahlen $\in \mathbb{N}$ können durch rekursive Addition von 1 dargestellt werden. Diese Rekursion kann beliebig tief werden, hat allerdings immer die gleiche Form. Wir definieren uns einen Platzhalter t_k um beliebig lange solcher Ausdrücke einfach darzustellen:

$$\begin{aligned} t_2 &= \underline{\text{plus}(\text{eins}, \text{eins})} \\ t_3 &= \underline{\text{plus}(\text{eins}, \text{plus}(\text{eins}, \text{eins}))} \\ &\vdots \\ t_{k+1} &= \underline{\text{plus}(\text{eins},} t_k \underline{)} \end{aligned}$$

k entsprichtder Anzahl der $\underline{\text{eins}}$ im Ausdruck.

- **Induktionsbasis:**
  In der Induktionsbasis beweisen wir einen oder mehrere Basisfälle. Das sind in unserem Fall die beiden Konstanten sowie der Fall t_2:

$$\begin{aligned} I_{\mathcal{T}}(\omega, \underline{\text{null}}) &= 0 \\ I_{\mathcal{T}}(\omega, \underline{\text{eins}}) &= 1 \\ I_{\mathcal{T}}(\omega, t_2) = I_{\mathcal{T}}\Big(\omega, \underline{\text{plus}(\text{eins}, \text{eins})}\Big) &= 2 \end{aligned}$$

- **Induktionshypothese:**
  In der Induktionshypothese treffen wir eine Aussage über ein Element n oder mehrere (evtl. alle) Elemente bis zu einem gewissen Element n. In unserem Fall sagen wir dass

$$I_{\mathcal{T}}(\omega, t_n) = n$$

- **Induktionsschritt:**
  Im Induktionsschritt erfolgt zeigen wir nun dass unter Annahme der Korrektheit der Induktionshypothese der Beweis auch für das Element $n + 1$ (also das erste Element über das wir keine Annahme getroffen haben) erbracht werden kann.

(Häufiger Fehler: Wird im Induktionsschritt die Induktionshypothese nicht verwendet so handelt es sich nicht um eine vollständige Induktion!) Wir müssen also zeigen dass:

$$
\begin{aligned}
I_{\mathcal{T}}(\omega, t_{n+1}) &= n+1 && \text{(Einsetzen von } t_{n+1}\text{)} \\
I_{\mathcal{T}}\left(\omega, \underline{\text{plus(eins,}t_n\text{)}}\right) &= n+1 && \text{(Interpretationsfunktion durchführen)} \\
+\left(I_{\mathcal{T}}(\omega, \underline{\text{eins}}), I_{\mathcal{T}}(\omega, t_n)\right) &= n+1 \\
+\left(1, I_{\mathcal{T}}(\omega, t_n)\right) &= n+1
\end{aligned}
$$

Unter Verwendung der Induktionshypothese (d.h. wir setzen die Induktionshypothese $I_{\mathcal{T}}(\omega, t_n) = n$ ein):

$$+(1, n) = n+1$$

Damit ist der Beweis erbracht.