Einführung in die Informatik I (autip)

Dr. Stefan Lewandowski

Fakultät 5: Informatik, Elektrotechnik und Informationstechnik
Abteilung Formale Konzepte
Universität Stuttgart

20.12.2006 + 10.+17.01.2007 / Version 17. Januar 2007

Inhaltsverzeichnis

2 Formale Konzepte

2.1 Grammatiken und Formale Sprachen

Wir haben gesehen, dass wir mit EBNF-Regeln den korrekten formalen Aufbau (die Syntax) eines Ada-Programms beschreiben können. Dass wir dabei manche Eigenschaften (z.B.,

dass verwendete Bezeichner auch immer deklariert werden müssen) nicht durch EBNF-Regeln abbilden konnten, nehmen wir hier nochmals zur Kenntnis.

EBNF-Regeln sind eine Möglichkeit durch sogenannte *kontextfreie Grammatiken* die zugehörigen *kontextfreien Sprachen* zu beschreiben und sind nicht nur dazu da, die Syntax einer Programmiersprache anzugeben. Wir werden nun diesen Mechanismus der Beschreibung von Sprachen durch Grammatiken und verwandte Methoden auf einer formalen Basis betrachten.

2.1.1 Wörter, Alphabete, Sprachen

Die Mengen, aus deren Zeichen wir die Wörter bilden, nennen wir *Alphabete*. Ein Alphabet ist eine endliche Menge $A = \{a_1, a_2, \ldots, a_n\}$.

Ein *Wort* u über einem Alphabet A ist eine endliche Folge von Zeichen aus A:

$$u := u_1 u_2 \ldots u_{n-1} u_n$$

mit $n \geq 0$ und $u_i \in A$, $1 \leq i \leq n$. Das leere Wort ($n = 0$) in obiger Definition macht beim Aufschreiben Probleme: wir schreiben es als ε.

Die Menge A^* aller Wörter über A heißt *Wortmenge* – oder freies Monoid – über A. Eine (formale) *Sprache* ist eine (beliebige) Menge von Wörtern über A, also eine Teilmenge von A^*.

Sprachen, auch solche über endlichen Mengen, sind nicht notwendigerweise endlich (z.B. ist A^*, die Menge aller Wörter, auch eine Sprache). Die Beschreibung einer unendlich großen Sprache kann also insbesondere nicht durch eine Liste aller Wörter erfolgen.

Im Vergleich zu natürlichen Sprachen ist die Verwendung der Begriffe Alphabet, Zeichen und Wort manchmal verwirrend. Den Begriff eines *Satzes* gibt es z.B. nicht. Wir werden also nicht mehrere Wörter einer Sprache miteinander zu etwas Übergeordnetem verknüpfen. Betrachten wir die Sprache aller syntaktisch korrekten Ada-95-Programme, so ist jedes dieser Programme ein Wort aus dieser formalen Sprache. Ein Wort einer formalen Sprache kann also durchaus mit dem übereinstimmen, was in einer natürlichen Sprache ein Satz ist. Genauso kann bei der Beschreibung einer formalen Sprache ein Wort einer natürlichen Sprache plötzlich als Zeichen aufgefasst werden (wir könnten z.B. bei der Beschreibung der Syntax von Ada 95 die reservierten Wörter jeweils als eigenes Zeichen auffassen – das Alphabet könnte dann das Element „begin“ enthalten und dies wäre dann etwas anderes als die Aneinanderreihung der 5 Buchstaben „b“, „e“, „g“, „i“ und „n“ – wir machen von dieser Möglichkeit allerdings hier keinen Gebrauch, d.h. hier werden in der Regel die Zeichen eines Alphabets stets einzelne Zeichen unserer natürlichen Sprache sein – formal gesehen ist dies aber keine Voraussetzung).

2.1.2 Operationen auf Wörtern

Eine wichtige Verknüpfung von Wörtern ist das Hintereinanderschreiben (*Konkatenation*).

Seien $u = u_1 u_2 \ldots u_n, v = v_1 v_2 \ldots v_m \in A^*$ Wörter, dann ist

$$uv := u_1 u_2 \ldots u_n v_1 v_2 \ldots v_m,$$

die Konkatenation ist also eine Abbildung $A^* \times A^* \to A^*$.

Die Definition gilt natürlich auch für das leere Wort ε, hier ergibt sich für beliebiges $u \in A^*$ die Regel

$$u\varepsilon = \varepsilon u = u.$$

Stellen wir uns die Konkatenation als Multiplikation vor, können wir Potenzen u^n eines Wortes u definieren als das n-fache Hintereinanderschreiben von u:

$$u^0 := \varepsilon;\ u^i := u^{i-1}u,\ i = 1, 2, \ldots$$

Hier gelten einige der bekannten Rechenregeln für Potenzen, z.B.

$$u^{(n+m)} = u^n u^m$$

für $u \in A^*$ und $n, m \in \mathbb{N}_0$.

2.1.3 Operationen auf Sprachen

Sprachen sind Mengen, daher können wir die üblichen Mengenoperationen anwenden: Durchschnitt, Vereinigung, Differenz zweier Sprachen, Komplement $A^* \setminus L$.

Eine spezielle Operation auf Sprachen ist die Konkatenation, sie wird über die Konkatenation von Wörtern aus den zu konkatenierenden Sprachen definiert:

$$L_1 \circ L_2 := \{uv \mid u \in L_1, v \in L_2\}$$

(man schreibt oft auch kurz L_1L_2, der „$\circ$“ wird dann ähnlich wie der „$\cdot$“ bei der mathematischen Multiplikation weggelassen).

Analog zu Potenzen bei Wörtern definiert man auch das Potenzieren von Sprachen durch entsprechend häufige Konkatenation: $L^0 := \{\varepsilon\}$, $L^{i+1} := L^i L$ für $i \geq 0$.

Achtung: $\{\varepsilon\}$ ist nicht die leere Menge $\emptyset$ sondern die einelementige Menge, die nur das leere Wort ε enthält.

L^i ist nicht die Menge $\{u^i \mid u \in L\}$. Betrachten wir das Beispiel $L = \{a, b\}$ (die Sprache, die nur aus den Wörtern a und b besteht – wir sehen hier auch, dass a gleichzeitig sowohl ein Element des Alphabets als auch ein Wort einer Sprache sein kann). Dann ist die Sprache $L^2 = L \circ L = \{uv \mid u, v \in L\} = \{aa, ab, ba, bb\}$, also mehr als $\{u^2 \mid u \in L\} = \{aa, bb\}$.

2.1.4 Darstellung von Sprachen

Wir bedienen uns bei der Beschreibung und Definition von formalen Sprachen oft der Mengenschreibweise, etwa für die Sprache über der Menge $A := \{a, b\}$, die aus einer Folge von a besteht, gefolgt von genauso vielen b:

$$\{u \in A^* \mid u = a^n b^n, n \in \mathbb{N}_0\}$$

Diese Form eignet sich nur für halbwegs übersichtliche Beispiele (wir werden uns in diesem Abschnitt auf solche übersichtlichen Beispiele beschränken). Bei komplexeren Sprachen, z.B. der Menge aller korrekt aufgebauten Ada-Programme, benötigen wir mächtigere Darstellungsformen.

Wir geben statt einer Mengenbeschreibung der formalen Sprache nur eine Menge von Konstruktionsregeln an, mit deren Hilfe die Sprache erzeugt werden kann. Zwei Formen dieser Konstruktionsregeln haben wir mit der EBNF und den Syntaxdiagrammen schon kennengelernt. Wir betrachten hier nun noch eine dritte Form dieses Mechanismus: die *kontextfreien Grammatiken.*

Die Idee dahinter ist in allen drei Erscheinungsformen dieselbe: mittels eines Satzes von Regeln werden Wörter Schritt für Schritt aufgebaut, indem gewisse Wortbestandteile (die Nonterminalsymbole) durch andere (normalerweise kompliziertere) Wörter ersetzt werden.

Randbemerkung 1: Eine weitere Methode, Sprachen zu definieren, ist ein Verfahren anzugeben, das feststellt, ob ein gegebenes Wort zur Sprache gehört oder nicht. Derartige Verfahren kann man mittels *Automaten* (im Fall der kontextfreien Sprachen Kellerautomaten, auch Pushdown-Automaten genannt), realisieren; wir nehmen hier nur zur Kenntnis, dass es so etwas gibt.

Randbemerkung 2: Ebenso gibt es effiziente Verfahren, auf Basis von Grammatiken algorithmisch festzustellen, ob ein Wort zu der von einer Grammatik beschriebenen Sprache gehört – auch diese werden wir hier nicht näher betrachten. Stichworte für Interessierte sind der Cocke-Younger-Kasami-Algorithmus, kurz CYK-Algorithmus (dazu werden die Grammatiken in der Chomsky-Normal-Form benötigt, jede kontextfreie Grammatik kann in diese Normal-Form umgewandelt werden) und der Earley-Algorithmus. Ersterer ist Standardstoff im Vordiplom des Hauptfachs Informatik, letzterer wird in Stuttgart eher selten in den Vorlesungen behandelt. Für beide finden sich Beschreibungen unter
`http://en.wikipedia.org/wiki/Category:Parsing_algorithms`

2.1.5 Kontextfreie Grammatiken

Definition Eine kontextfreie Grammatik wird beschrieben durch vier Komponenten: $G = (V, \Sigma, P, S)$ mit

- einer nicht leeren endlichen Menge V von *Nonterminalsymbolen*
- einer nicht leeren endlichen Menge Σ von *Terminalsymbolen* (dies ist unser Alphabet); es muss gelten $V \cap \Sigma = \emptyset$,
- einem *Startsymbol* $S \in V$ und
- einer endlichen Menge von *Produktionen* $P \subset V \times (V \cup \Sigma)^*$.

Die Produktionen haben also die Form (A, x), wobei A ein Nonterminalsymbol ist und x ein Wort aus Terminal- und Nonterminalsymbolen. Um den Ersetzungsaspekt bei den Produktionsregeln zu betonen schreibt man statt (A, x) oft $A \to x$.

Ableitungen und die Sprache $L(G)$ Zu zwei Wörtern $u, v \in (V \cup \Sigma)^*$ definiert man den Begriff der *Ableitbarkeit*:

- Wir nennen v *in einem Schritt ableitbar* aus u, in Zeichen $u \Rightarrow v$, wenn es Zerlegungen $u = xAy$ und $v = xwy$ gibt, so dass $x, y, w \in (V \cup \Sigma)^*$ und $A \to w \in P$.
- Wir nennen v *in k Schritten ableitbar* aus u, in Zeichen $u \Rightarrow^k v$, wenn es $z_0, z_1, \ldots, z_k$ aus $(V \cup \Sigma)^*$ gibt mit $u = z_0$, $v = z_k$, und $z_{i-1} \Rightarrow z_i$, $1 \leq i \leq k$. Die $z_1, z_2, \ldots, z_{k-1}$ sind also die Zwischenergebnisse des Ableitungsprozesses.
- Wir nennen schließlich v *ableitbar* aus u, in Zeichen $u \Rightarrow^* v$, wenn $u = v$ gilt oder es ein k gibt, so dass v in k Schritten aus u ableitbar ist.

Die durch eine kontextfreie Grammatik G definierte Sprache besteht nun aus genau den Wörtern aus Terminalsymbolen, die aus dem Startsymbol abgeleitet werden können:

$$L(G) := \{w \in \Sigma^* : S \Rightarrow^* w\}.$$

Wichtig ist, dass hier $w \in \Sigma^*$ gilt, also nur Wörter zur Sprache gehören, die keine Nonterminalsymbole mehr enthalten. Wörter aus Terminal- und Nonterminalsymbolen nennt man auch *Satzformen*.

Eine Sprache L heißt kontextfrei, wenn es eine kontextfreie Grammatik G gibt mit $L = L(G)$.

Beispiel: Bezeichner Als Beispiel für eine Grammatik betrachten wir die Konstruktionsregeln für Bezeichner in Ada (Buchstabe, gefolgt von beliebig vielen Buchstaben, Ziffern und Unterstrichen; Unterstriche nur einzeln und nicht am Ende des Wortes).

Wir haben bereits Bezeichner in EBNF definiert. Wir beschränken uns hier der Einfachheit halber auf die Buchstaben 'a' und 'b' und die Ziffern '1' und '2'. In EBNF formuliert ist ein

- <Bezeichner> ::= <Buchstabe> { ['_'] <Buchstabe> | ['_'] <Ziffer> }
- <Buchstabe> ::= 'a' | 'b' und <Ziffer> ::= '1' | '2'

Die Erweiterung auf alle Buchstaben und Ziffern ist offensichtlich.

Für die Grammatik $G = (V, \Sigma, P, S)$ müssen wir nun die vier Komponenten definieren:

- Die Terminalsymbole unserer Sprache sind alle darin vorkommenden Zeichen:
  $\Sigma := \{a, b, 1, 2, \_\}$
- Als Nonterminalsymbole verwenden wir (die Wahl ist willkürlich): $V := \{S, R, B, Z\}$,
  - S soll für einen Bezeichner stehen (das wird also das Startsymbol werden, „S“ wie Start),
  - B soll für einen Buchstaben (also a oder b) stehen, Z für eine Ziffer (also 1 oder 2), R für den Rest – nach dem ersten Buchstaben kommt der Rest.
- Da wir S schon zum Startsymbol erklärt haben, fehlt nun noch das Wichtigste: eine Menge von Produktionsregeln.
  - Aus B kann ein Buchstabe werden: $B \to a$, $B \to b$.
  - Aus Z kann eine Ziffer werden: $Z \to 1$, $Z \to 2$.
  - Aus dem Startsymbol wird ein Buchstabe, gefolgt von einem Rest: $S \to BR$.
  - Die Schwierigkeit liegt nun darin, diesen Rest R zu definieren.
    Solange wir den Unterstrich vernachlässigen, reichen die Regeln $R \to BR$, $R \to ZR$, $R \to \varepsilon$, damit aus dem Rest beliebig viele Buchstaben und Ziffern erzeugt werden können; wenn wir fertig sind, dürfen wir wegen der dritten Regel das R zum leeren Wort, dem ε, ableiten – R nach ε ableiten heißt in der Praxis, wir streichen das R weg.
  - Der Unterstrich erfordert die weiteren Produktionen: $R \to \_BR$ und $R \to \_ZR$.

Nun können wir z.B. den Bezeichner ab_b1_a ableiten:

$$S \Rightarrow BR \Rightarrow aR \Rightarrow aBR \Rightarrow abR \Rightarrow ab\_BR \Rightarrow ab\_bR \Rightarrow^* ab\_b1\_aR \Rightarrow ab\_b1\_a$$

wobei wir im letzten Ableitungsschritt $ab\_b1\_aR \Rightarrow ab\_b1\_a$ die Regel $R \to \varepsilon$ verwenden, um das überflüssige R loszuwerden.

2.1.6 Ableitungen und Bäume

Bäume sind in zweierlei Weise für die Veranschaulichung von Sprachen nützlich:

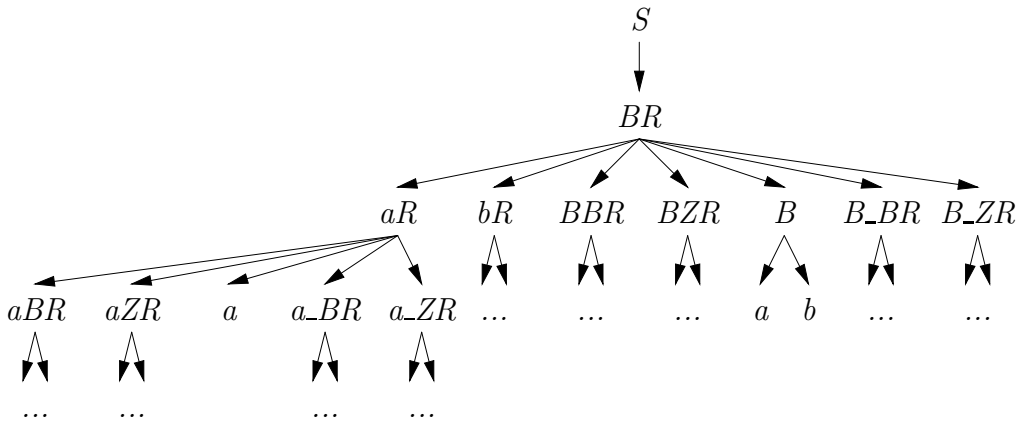
- Zum einen können wir einen Baum konstruieren, der alle Wörter enthält, die aus dem Startsymbol abgeleitet werden können.
- Zum andern werden wir Bäume betrachten, die Ableitungen von einem einzelnen Wort darstellen.

Darstellung von $L(G)$ Zur Darstellung der ganzen Sprache verwenden wir einen Baum, bei dem die Knoten mit Wörtern aus $(V \cup \Sigma)^*$ beschriftet sind und an dem man ablesen kann, was für Wörter abgeleitet werden können.

- Die Wurzel beschriften wir mit dem Startsymbol.
- Ein mit einem Wort u beschrifteter Knoten hat für jedes $v \in (V \cup \Sigma)^*$, für das $u \Rightarrow v$ gilt, einen mit v beschrifteten Sohn – v hat wiederum Söhne v', für die $v \Rightarrow v'$ gilt, usw.

Von diesem — im Allgemeinen unendlich großen — Baum bilden diejenigen Beschriftungen der Blätter, die nur aus Terminalsymbolen bestehen, die Sprache $L(G)$. Ein Blatt kann nur dann noch Nichtterminale enthalten, wenn es für diese Nichtterminale keine Produktionsregel gibt – in diesem Fall lässt sich aus dem betreffenden Nichtterminal kein Wort ableiten, es ist nicht *produktiv*. Man kann sich leicht überlegen, dass man die Grammatik so vereinfachen kann, dass dieser Fall der nicht produktiven Nichtterminale nicht mehr auftritt.

Der Baum aller Ableitungen unserer Grammatik für (eingeschränkte) Bezeichner:



Der Baum wird erwartungsgemäß sehr schnell sehr groß. Trotzdem sollte uns nicht entgehen, dass das Wort a zweimal im Baum erscheint, die meisten anderen Bezeichner erscheinen sogar viel öfter.

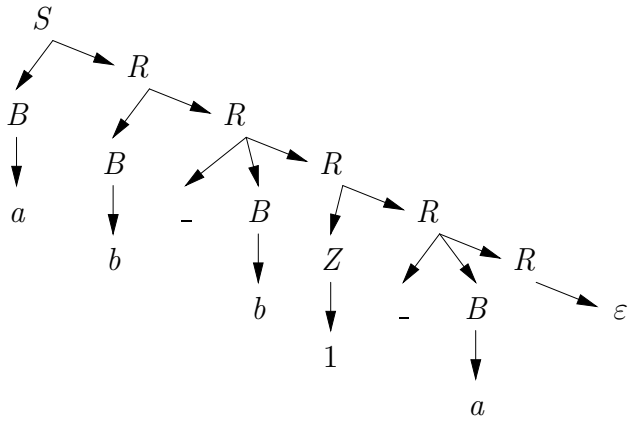
Ableitungsbaum eines Wortes Man kann auch die Ableitung $u \Rightarrow^* v$ für Nonterminale $u \in V$ und Wörter $v \in (V \cup \Sigma)^*$ mittels eines Baums — er heißt dann *Ableitungsbaum* — veranschaulichen.

Hier sind die Knotenbeschriftungen einzelne Symbole aus $V \cup \Sigma$. In die Wurzel schreiben wir das Nichtterminal u – in der Regel interessieren uns Ableitungsbäume für $S \Rightarrow w$ mit Startsymbol S und Wörtern $w \in \Sigma^*$, es steht dann das Startsymbol S in der Wurzel.

Wenn im Laufe der Ableitung $u \Rightarrow^* v$ ein Nonterminalsymbol A aufgrund einer Produktion $A \to w$ ersetzt wird, fügen wir alle Symbole aus w der Reihe nach als Sohnknoten des mit A beschrifteten Knotens ein.

Wie bei Rechenbäumen haben wir hier wieder einen Baum vorliegen, bei dem die Reihenfolge der Sohnknoten wichtig ist! Es ist ein geordneter Baum.

Der Ableitungsbaum zu $S \Rightarrow ab\_b1\_a$ lautet

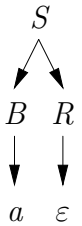


Die Blätter ergeben von „links nach rechts“ gelesen das aus dem in der Wurzel stehenden Nonterminal u abgeleitete Wort v. Wir beobachten hierbei, wie das leere Wort ε bei der Konkatenation verschwindet ($ab\_b1\_a\varepsilon = ab\_b1\_a$ – wir erinnern uns, dass $u\varepsilon v = uv$ für alle Wörter u und v).

Wir sehen, dass die Reihenfolge, in der die Buchstaben hintereinandergereiht werden, große Ähnlichkeit hat mit der Reihenfolge, in der z.B. bei den Fibonacci-Zahlen der Baum der rekursiven Aufrufe durchlaufen wurde. Wir behalten dies schonmal als interessante Eigenschaft im Hinterkopf – wir werden dies im Kapitel über Zeiger, Listen und Bäume noch vertieft behandeln.

Eindeutigkeit der Ableitung

Im obigen Baum aller Ableitungen tauchte das Wort a zwei Mal auf. Betrachten wir die zugehörigen Ableitungen $S \Rightarrow BR \Rightarrow aR \Rightarrow a$ und $S \Rightarrow BR \Rightarrow B \Rightarrow a$, so unterscheiden sich diese nur in der Reihenfolge, in der die letzten beiden Ableitungsschritte durchgeführt wurden. Betrachten wir den zugehörigen Ableitungsbaum, so ist dieser identisch.



Unterscheiden sich die Ableitungen also nur in der Reihenfolge, in der die Produktionen auf Nichtterminale angewendet wurden, entsteht jeweils derselbe Ableitungsbaum. Solche Ableitungen sehen wir als gleich an. (Diese Form von Mehrdeutigkeit tritt immer auf, wenn im Ableitungsprozess wenigstens eine Regel verwendet wurde, auf deren rechter Seite mindestens zwei Nichtterminale vorkommen – dies ist bei kontextfreien Grammatiken fast immer der Fall.)

Ableitungen, die verschiedene Ableitungsbäume aber dasselbe Wort erzeugen, lassen wir hingegen als verschieden gelten. Wir erhalten folgende Definitionen:

- Wörter $u \in L(G)$ heißen *eindeutig*, wenn sie genau einen Ableitungsbaum besitzen, sonst heißen sie *mehrdeutig*.
- Die Grammatik G heißt eindeutig, wenn alle Wörter in $L(G)$ eindeutig sind.

Wichtig: Diese Begriffsbildung ist abhängig von der Grammatik. Für viele Sprachen kann man sowohl eindeutige als auch mehrdeutige Grammatiken angeben – ein einfaches Beispiel ist die Sprache $\{a\}^*$ (also die Sprache, die aus allen Wörtern besteht, die beliebig häufige Wiederholungen des Buchstaben a sind). Eine Grammatik mit den Produktionsregeln $S \to SS,\ S \to a,\ S \to \varepsilon$ erzeugt die Sprache mehrdeutig, mit den Produktionsregeln $S \to aS,\ S \to \varepsilon$ kann man sie aber auch eindeutig erzeugen (selbst überprüfen).

Warum betrachtet man überhaupt Ein-/Mehrdeutigkeit von Grammatiken? Wir skizzieren die Problematik am Beispiel einer modifizierten `if-then-else`-Anweisung. Stellen wir uns vor, Ada hätte kein `end if;` (wir lassen hier auch das `elsif` weg) – die `if`-Anweisung hätte dann folgenden Aufbau:

> `if` <Bedingung> `then` <Anweisung> [`else` <Anweisung>]

(sollen mehrere Anweisungen in einem der beiden Zweige ausgeführt werden, so könnte dies leicht durch einen Block (`begin`...`end`) erreicht werden).

Sehen wir uns dazu eine Grammatik an, die den entsprechenden Ausschnitt aus der Sprachdefinition wiedergibt: Es sei $G := (V, \Sigma, P, A)$ mit

- Nonterminalsymbolen $V := \{A, F, B\}$ (zum Vorstellen: Anweisung, Fallunterscheidung und Bedingung),
- Terminalsymbolen $\Sigma := \{i, t, e, a_1, a_2, b_1, b_2\}$ (zum Vorstellen: `if`, `then`, `else`, zwei Anweisungen und zwei Bedingungen),
- Produktionen: $P := \{A \to F,\ A \to a_1,\ A \to a_2,\ F \to iBtAeA,\ F \to iBtA,\ B \to b_1,\ B \to b_2\}$ (eine sehr künstliche Definition, die nur genügend Regeln zur Verfügung stellt, um das folgende Beispiel rechnen zu können).
- Startsymbol ist das vierte Element des Tupels (V, Σ, P, A), also das A.

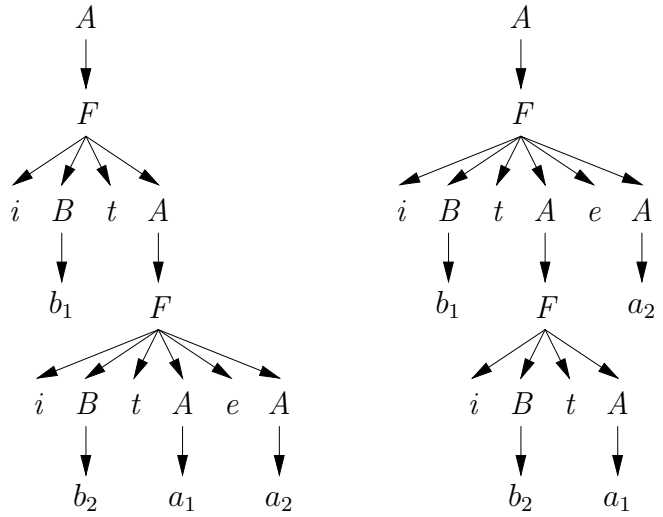
Betrachten wir nun die Ableitbarkeit von

$$A \Rightarrow^* ib_1tib_2ta_1ea_2$$

Auf die Programmiersprachen bezogen betrachten wir

```
if Bed1 then if Bed2 then Anw1 else Anw2
```

Dieses Wort ist ableitbar – und zwar auf wesentlich verschiedene Weisen, hier sind die Ableitungsbäume:



Also: Das Wort ist nicht eindeutig, somit ist auch G nicht eindeutig.

Allgemein (nicht nur bei Grammatiken) versucht man den formalen Aufbau (die Syntax) in Einklang mit der Bedeutung (der Semantik) zu modellieren.

Die Nichteindeutigkeit der Grammatik führt hier nun zu einem Problem, wenn man die Sprachelemente mit ihrer üblichen Bedeutung versieht: Angenommen, b_1 sei `true` und b_2 sei `false`. Gemäß der linken Ableitung würde a_2 ausgeführt werden, gemäß der rechten jedoch nichts.

Wie ist dieser Konflikt zu lösen?

- In Ada 95 gibt es das Problem in dieser Art nicht, da durch das `end if` solche Mehrdeutigkeiten ausgeschlossen sind.
- In Pascal und C wird der Konflikt durch eine Sonderregel gelöst: ein `else` gehört zum innersten `if`, das noch kein `else` hat – hier entspricht dies dem linken Ableitungsbaum.
- Ein weitere Möglichkeit zur Vermeidung der Mehrdeutigkeit wäre, zu jedem `if-then` auch stets ein `else` zu erzwingen, wobei in diesem Zweig dann ggf. nur eine `null`-Anweisung stünde.

Wichtig: die Mehrdeutigkeit resultiert nicht aus der Bedeutung (der *Semantik*) der Fallunterscheidung, sondern tritt losgelöst von der Bedeutung schon bei der Grammatik (der *Syntax*) auf.

Randbemerkung 1: Bei kontextfreien Grammatiken kann man jedes ableitbare Wort so ableiten, dass in jedem Schritt jeweils das am weitesten links stehende Nonterminalsymbol abgeleitet wird. Dies wird als *Linksableitung* bezeichnet. Diese erlauben eine alternative Definition der Mehrdeutigkeit: Ein Wort $u \in L(G)$ heißt *eindeutig*, wenn es genau eine Linksableitung besitzt, sonst heißt es *mehrdeutig* – untersuchen Sie die Linksableitungen im Beispiel des Bezeichners a und im Beispiel der modifizierten `if`-Anweisung $ib_1tib_2ta_1ea_2$.

Randbemerkung 2: Man sieht den Regeln der Grammatiken im Allgemeinen nicht an, ob sie zu Mehrdeutigkeiten führen oder nicht. Regeln, deren rechte Seiten wie oben gleich beginnen, reichen dazu nicht aus – untersuchen Sie als Beispiel die Grammatik mit den Produktionsregeln $S \to A,\ S \to Aa,\ A \to Aaa,\ A \to \varepsilon$, diese Grammatik ist eindeutig, obwohl wir wie im obigen Beispiel Regeln haben, deren rechte Seiten gleich beginnen.

2.1.7 Grammatiken, EBNF oder Syntaxdiagramme?

Grammatiken stellen von den dreien vielleicht die formal sauberste Möglichkeit zur Sprachbeschreibung dar.

Für oft benötigte Konstruktionen (Alternativen, optionale oder wiederholbare Elemente) bietet sich die erweiterte Backus-Naur-Form (EBNF) an. Die zusätzlichen Möglichkeiten lassen sich jedoch leicht mit Grammatiken simulieren.

- $< A > ::= < B > | < C >$ wird zu $A \to B$, $A \to C$,
- $< A > ::= [< B >]$ wird zu $A \to B$, $A \to \varepsilon$,
- $< A > ::= \{< B >\}$ wird zu $A \to BA$, $A \to \varepsilon$.

Ebenso lassen sich Grammatiken sehr leicht in Syntaxdiagramme verwandeln: Jede Regel wird zu einem eigenen Syntaxdiagramm (die Nonterminal- und Terminalzeichen der rechten Seite der Regel werden in runden bzw. eckigen Kästchen mit Pfeilen verbunden hintereinander geschrieben) – gibt es mehrere Regeln mit gleicher linker Seite, so spaltet sich das entsprechende Syntaxdiagramm nur zu Beginn auf. Die Umwandlung beliebiger Syntaxdiagramme in Grammatiken ist etwas komplizierter (im Wesentlichen ist dazu für jedes Zusammentreffen von Pfeilen im Syntaxdiagramm ein eigenes Nonterminalsymbol einzuführen – Details selbst überlegen).

Wem Syntaxdiagramme besser als EBNF-Regeln gefallen, der findet unter
`http://cui.unige.ch/db-research/Enseignement/analyseinfo/Ada95/BNFindex.html`
die komplette Ada-Grammatik außer in der EBNF-Darstellung auch in Form von Sytaxdiagrammen.

2.1.8 Andere Grammatiken

Kontextfreie Grammatiken eignen sich insbesondere zur Darstellung der Syntax von Programmiersprachen, helfen aber auch in anderen Bereichen zur Modellbildung. Die nicht mit kontextfreien Grammatiken darstellbaren Aspekte beschreibt man in der Regel mit zusätzlichen Attributen.

Hier sei nur noch darauf hingewiesen, dass das Attribut „kontextfrei“ unserer Grammatiken wichtig ist, da es auch andere Klassen von Grammatiken und von diesen erzeugte Sprachen gibt, z.B. kontext-sensitive (eine größere Sprachklasse als die kontextfreien Sprachen) oder reguläre (eine Teilmenge der kontextfreien Sprachen).

Näheres dazu findet sich in der Literatur unter dem Stichwort *Chomsky-Hierarchie*.

2.2 Berechenbarkeit – Grenzen der Programmierkunst

Wir haben bereits einiges über Ada 95 und die Möglichkeiten der Programmierung gelernt. Manche Probleme konnten wir bisher noch nicht lösen. Dies hat vielfältige Gründe:

- Wir kennen noch nicht alle Ada-95-Sprachkonstrukte, z.B. das Abfangen von Laufzeitfehlern, Lesen und Schreiben von Dateien, dynamische Datenstrukturen mittels Zeigern.

  Diese können wir uns aber mit etwas Mühe noch erarbeiten.

- Uns fehlen bei manchen Problemen die richtigen Ideen, die zu einer Lösung führen. Hier hilft nur Übung und Erfahrung – wir arbeiten daran.

  Zur Realisierung größerer Softwareprojekte fehlen uns geeignete Methoden – eine ausführliche Behandlung dieser Problematik würde den Rahmen der Vorlesung sprengen. Aber auch diese lassen sich erlernen, es ist nur eine Frage der Zeit und Mühe.

- Es gibt prinzipielle Grenzen für das, was mit Programmen lösbar ist.

Von diesem letzten Punkt handelt dieser Abschnitt. Dabei geht es nicht darum, dass eine bestimmte Programmiersprache wie z.B. Ada 95 die Lösung eines Problems nicht zulässt oder dass der Programmierer einfach die richtige Idee zur Lösung noch nicht gefunden hat, sondern dass es Probleme gibt, die sich prinzipiell algorithmisch nicht lösen lassen, unabhängig davon, welche Programmiersprache man zur Lösung verwendet.

Wozu dann das alles? Keine Panik! Fast alles, was wir uns jemals als Problem ausdenken werden oder was wir heute im Studium oder später im Beruf lösen möchten, lässt sich mit Programmen realisieren.

Dass es Probleme geben soll, die sich nicht algorithmisch lösen lassen, ist auf den ersten Blick eine etwas irritierende Aussage. Um diese genauer fassen zu können, müssen wir uns zunächst ein genaueres Bild davon machen, was die Ausführung eines Programms bewirkt – wie welche Eingabedaten zu welchen Ausgabedaten verarbeitet werden.

Die Darstellung hier ist an Ada 95 angelehnt. In Lehrbüchern wird der Begriff der Berechenbarkeit in der Regel mit Hilfe eines abstrakten Rechenmodells, der Turing-Maschine, erläutert. Um nicht vom Wesentlichen abzulenken, bleiben wir hier im Dunstkreis von Ada 95.

2.2.1 Programme realisieren Abbildungen

Ein Programm setzt einen Algorithmus um, indem es in einer endlichen Zahl wohldefinierter Schritte mit endlichem Speicherplatzbedarf für gegebene Eingabedaten zugehörige Ausgabedaten produziert.

Abstrakt beschreibt ein Programm eine Funktion. In Ada 95 können wir die Ein- und Ausgabedaten als Wörter über einem geeigneten Alphabet Σ auffassen. Ein Programm realisiert somit eine Abbildung $f : \Sigma^* \to \Sigma^*$.

Wir können uns dies vereinfacht so vorstellen, dass die `Get`-Anweisungen im Programm nach und nach die Zeichen des Eingabewortes aus Σ^* einlesen und dass die `Put`-Anweisungen eine u.U. sehr lange Zeichenkette aus Σ^*, das Ausgabewort, erzeugen. Die Details, wie man dies genau definiert, wie Return-Zeichen verarbeitet werden, u.ä. spielen hier keine entscheidene Rolle – wir gehen auf diese nicht weiter ein.

Die von einem Programm P realisierte Funktion f ist in der Regel partiell, d.h., sie ist nicht für alle Eingaben definiert. Die Lücken im Definitionsbereich können folgende Gründe haben:

- Das Programm gerät mit der betreffenden Eingabe in eine Endlosschleife. Es terminiert somit nicht. Wir betrachten die Ausgabe eines Programms nur dann als gültig, wenn das Programm auch irgendwann endet.

- Die Eingabe ist nicht ausreichend lang. Das Programm wartet also irgendwo auf eine Eingabe, hat aber schon alle Zeichen der Eingabe gelesen. Auch in diesem Fall terminiert das Programm nicht und die Funktion f hat mit dieser Eingabe keinen definierten Funktionswert.

- Das Programm terminiert mit einer Fehlermeldung (z.B., wenn bei der Berechnung eine Division durch 0 aufgetreten ist oder einer Variablen ein Wert außerhalb des für den betreffenden Datentyps zulässigen Bereichs zugewiesen wurde).

In allen anderen Fällen terminiert das Programm und liefert uns abhängig von der Eingabe eine Ausgabe.

2.2.2 Berechenbare Funktionen – Definition und Beispiele

Wir nennen eine Funktion $f : \Sigma^* \to \Sigma^*$ *berechenbar*, wenn es ein Programm gibt, das f realisiert.

Um ein Gefühl dafür zu bekommen, was der Begriff *berechenbar* bzw. *nicht berechenbar* meint, betrachten wir einige Beispiele im Zusammenhang mit der Dezimalentwicklung der Zahl π, also der Folge 3, 1, 4, 1, 5, 9, 2, 6, 5, ...

- Die Funktion a_π mit $a_\pi(w) = 1$, falls w Anfang der Dezimalentwicklung von π ist, und 0 sonst (also z.B. $a_\pi(31415) = 1$, $a_\pi(4159) = 0$), ist berechenbar – man kennt Algorithmen, die π mit jeder gewünschten Genauigkeit berechnen, so dass sich der Wert von $a_\pi(w)$ für beliebiges w bestimmen lässt. (Bei der Umsetzung in Ada 95 müssen wir dabei ggf. eigene Zahltypen entwerfen, die eine ausreichende Genauigkeit unterstützen.)

- Von der Funktion b_π mit $b_\pi(w) = 1$, falls w in der Dezimalentwicklung von π vorkommt, und 0 sonst, weiß man nicht, ob sie berechenbar ist.

  Man kennt zwar derzeit noch kein w mit $b_\pi(w) = 0$, aber es ist nicht bewiesen, dass es keins gibt.

- Interessant ist der Status der Funktion $c_{\pi,d}$, wobei d eine Ziffer ist und $c_{\pi,d}(w) = 1$, falls es in der Dezimalentwicklung von π eine Stelle mit wenigstens w Ziffern d in ununterbrochener Reihenfolge gibt, und 0 sonst. (Wir interpretieren w in diesem Beispiel als Zahl.)

  Diese Funktionen sind berechenbar! Wir wissen zwar nicht, wie wir sie berechnen können, aber es gilt folgendes:

  - $c_{\pi,d}$ ist entweder konstant 1 oder
  - es gibt ein $w_0(d)$ mit $c_{\pi,d}(w) = 1$ für alle $w < w_0(d)$ und $c_{\pi,d}(w) = 0$ für alle $w \geq w_0(d)$.

  In beiden Fällen können wir ein Programm angeben, dass diese Abbildung realisiert. Im zweiten Fall wissen wir zwar nicht, welchen Wert die Konstanten $w_0(d)$ haben, wir wissen aber das eines der abhängig von diesen Konstanten definierten Programme das richtige ist.

  Es gibt also Funktionen, von denen wir die Berechenbarkeit zeigen können, ohne in der Lage zu sein, ein Programm explizit anzugeben, das diese Funktion realisiert.

Nun ist es an der Zeit eine konkrete Funktion vorzustellen, die nicht berechenbar ist.

2.2.3 Das Halteproblem – eine nicht berechenbare Funktion

Es wäre praktisch, wenn der Übersetzer uns warnen würde, wenn unser Programm bei bestimmten Eingabewerten in eine Endlosschleife gerät. Dass das leider im Allgemeinen nicht möglich ist, werden wir nun sehen.

Die Funktion `h`, die das *Halteproblem* lösen soll, bekommt als Parameter ein Programm `p` und eine Eingabe `w` für `p` (die Definition geeigneter Datentypen in Ada ist kein Problem, beides sind beliebig lange Zeichenketten – diese werden z.B. im Paket `Ada.Strings.Unbounded` zur Verfügung gestellt).

Es soll gelten:

$$\texttt{h(p,w)} = \begin{cases} 1 & \text{falls } \texttt{p} \text{ bei Eingabe } \texttt{w} \\ & \text{nach endlich vielen Schritten anhält,} \\ 0 & \text{sonst.} \end{cases}$$

Dadurch ist `h` eindeutig definiert. Wir sehen insbesondere, dass `h` für alle Eingaben definiert ist, d.h., `h` muss für beliebige Eingaben stets nach endlich vielen Schritten das Ergebnis liefern.

Nun werden wir durch einen Widerspruch zeigen, dass solch eine Funktion `h` nicht existieren kann, d.h., `h` ist nicht berechenbar.

Angenommen, wir haben eine Funktion `h(p,w)`, dann können wir folgende Prozedur schreiben:

```
procedure fies(p:Unbounded_String) is
begin
  if h(p,p)=1 then
    loop null; end loop; -- Endlos-Schleife
  end if;
end fies;
```

Wir beachten, dass die Prozedur `h` den Programmtext `p` einmal als Programmtext bekommt und einmal als Eingabe für das Programm `p`. Dies ist an sich kein Problem – stellen Sie sich z.B. einen Ada-95-Übersetzer vor, der seinen eigenen Source-Code übersetzt (dies wird im Compiler-Bau übrigens tatsächlich so gemacht).

Die Frage ist nun, was passiert, wenn wir `fies(fies)` aufrufen!?[1]

1. Fall: Annahme `fies(fies)` hält an (d.h., `h(fies,fies)=1`) – in diesem Fall gerät unsere Prozedur `fies` aber in eine Endlos-Schleife, wird also nicht anhalten! Widerspruch zur Annahme `h(fies,fies)=1`.

2. Fall: Annahme `fies(fies)` hält nicht an (d.h., `h(fies,fies)=0`) – in diesem Fall tut unsere Prozedur `fies` nichts und terminiert (mit Ausgabewort ε). Wir erhalten ebenfalls einen Widerspruch zur Annahme `h(fies,fies)=0`.

Was nun? Da beide Annahmen für den Funktionswert `h(fies,fies)` zu einem Widerspruch führen, haben wir damit gezeigt, dass solch eine Funktion `h` nicht existieren kann. Denn, wenn sie existieren würde, muss sie nach endlich vielen Schritten stets als Funktionswert entweder 0 oder 1 ermitteln.

[1] Mit dem Aufruf `fies(fies)` meinen wir hier für den Parameter `fies` den gesamten Programmtext, also `fies("procedure fies(p: ... end fies;")`, genau genommen müsste hier beim Parameter vor dem Schlüsselwort `procedure` auch noch das Paket mit der verwendeten Prozedur `h` mittels `with` und `use` eingebunden werden. Der innerhalb der Prozedur `fies` durchgeführte Aufruf `h(p,p)` entspricht dann `h("procedure ... end fies;","procedure ... end fies;")`.

2.2.4 Praktische Konsequenzen aus dem Halteproblem

Das Halteproblem ist keineswegs nur ein Beispiel von theoretischer Bedeutung: wenn wir ein Programm hätten, dass es löste, würden wir viele Programmierfehler gar nicht mehr machen können.

Man kann natürlich Verfahren erfinden, die viele Fälle abdecken, aber — wie wir gesehen haben — niemals alle.

Nun kann man sich fragen, ob man nicht eine Programmiersprache erfinden kann, in der jedes Programm terminiert.

Wenn wir die Ada-Konstruktionen, die wir kennengelernt haben, daraufhin untersuchen, sehen wir, dass wir nur zwei Möglichkeiten haben, nicht terminierende Programme zu schreiben: die `loop`- bzw. `while`-Schleife (die `for`-Schleife ist in diesem Zusammenhang harmlos, da sie eine zu Beginn festgelegte endliche Anzahl von Schleifendurchläufen durchführt) und den rekursiven Prozeduraufruf.

Wenn wir diese Möglichkeiten verbieten, würde sich das Halteproblem gar nicht erst stellen, da alle Programme nach endlichen vielen Schritten terminieren (oder mangels weiterer Eingaben irgendwo auf diese warten – in diesem Fall würde eine Simulation des Programmablaufs die Antwort bringen).

Wichtig: beim allgemeinen Halteproblem hilft diese Argumentation nicht! Es reicht nicht zu sagen, dass eine Simulation des Programms `p` mit Eingabe `w` nicht zum gewünschten Ergebnis führt (da die Simulation dann ja auch nicht terminieren würde) – mit dieser Argumentation zeigen wir nur, dass man das Halteproblem nicht durch Simulation lösen kann.

Im eingeschränkten Fall (ohne Rekursion und `while`/`loop`-Schleifen) geht dies, da wir in dem Fall wissen, dass das Programm bei ausreichend langer Eingabe in jedem Fall nach endlich vielen Schritten terminiert, was wir durch Simulation dieser höchstens endlich vielen Schritte feststellen können.

Das Leben ohne Rekursion und `while`/`loop`-Schleifen ist jedoch deutlich langweiliger. Es gibt Funktionen, die zwar berechenbar sind, für deren Berechnung Rekursion oder `while`-Schleifen notwendig sind. Das in der theoretischen Informatik in diesem Zusammenhang angeführte Standardbeispiel ist die Ackermann-Funktion. Diese ist definiert durch $a(0, m) = m + 1$, $a(n, 0) = a(n - 1, 1)$ für $n \geq 1$, $a(n, m) = a(n - 1, a(n, m - 1))$ sonst. Diese Funktion wächst sehr sehr schnell (schreiben Sie sich eine Ada-Funktion und berechnen Sie die Werte $a(3, 4)$, $a(3, 5)$, ... – versuchen Sie sich an $a(4, 1)$, so erfolgt die Berechnung noch recht schnell, bei $a(4, 2)$ werden sie das Ergebnis der Berechnung schon nicht mehr erleben).

Das Erkaufen von verbesserter Sicherheit (durch Vermeidung von Endlos-Schleifen) wäre also mit reduzierten Möglichkeiten erkauft.

Andersherum kann man sich fragen, ob man mit mehr Sprachkonstruktionen als wir sie kennen auch mehr Funktionen berechnen kann.

Die Vermutung hier ist: nein (Stichwort Church'sche These) – die unterschiedlichsten Versuche, Programme zu beschreiben, enden stets bei einer Menge berechenbarer Funktionen, die nicht größer ist, als die, die wir auch mit Ada 95 berechnen können.

Zur Beruhigung: Alle Funktionen zu den Problemen aus den Übungsaufgaben zu dieser Vorlesung sind berechenbar (wir haben ja entsprechende Programme geschrieben, die die jeweiligen Funktionen realisieren). Man muss also schon sehr genau suchen, um solche nicht berechenbaren Funktionen zu finden.

2.3 Aufwandsabschätzungen – die O-Notation (Landau-Symbole)

Bisher haben wir den Aufwand eines Algorithmus meist in der Form „proportional zu $f(n)$“ angegeben oder sogar versucht, den Proportionalitätsfaktor noch genau zu bestimmen – bei dem Beispiel Sortieren durch Minimumsuche war dies ok, bei der gerechten Erbschaft bedingt ok, bei Intervallschachtelung (binärer Suche) jedoch weniger geeignet $\rightsquigarrow$ wir betrachten nun die formale Lösung: O-Notation.

Die hauptsächliche Motivation zur O-Notation ist die Bewertung und der Vergleich der Effizienz von Algorithmen.

Die Aufwandsabschätzungen werden in Abhängigkeit der Problemgröße angegeben – dies ist i.A. die Anzahl der zu bearbeitenden Elemente oder z.B. die Länge der eingegebenen Zeichenfolge. Wir konzentrieren uns dabei auf große Problemgrößen – es interessiert also nur das asymptotische Laufzeitverhalten.

Ebenso interessieren uns nur signifikante Laufzeitunterschiede – als signifikant wollen wir alles erachten, was über konstante Faktoren hinausgeht, d.h., wenn ein Algorithmus A ein Problem stets doppelt so schnell löst wie der Algorithmus B, dann werden diese beiden in der O-Notation trotzdem als gleichwertig betrachtet. Die Signifikanz von Laufzeitunterschieden werden wir hier also nur an nichtkonstanten Faktoren festmachen. Die Idee dabei ist, dass wir Laufzeiten, die nur um einen konstanten Faktor länger sind, mit der nächsten (oder übernächsten) Rechnergeneration leicht ausgleichen können. Ist der Faktor jedoch abhängig von der Eingabegröße n, so ist dies i.A. nicht mehr möglich. Wir betrachten als Beispiel jeweils die Problemgrößen, die mit einem Prozessor A, der 10^8 Schritte pro Sekunde ausführen kann, und einem Prozessor B, der $2 \cdot 10^8$ Schritte pro Sekunde ausführen kann, in einer Stunde gelöst werden können, wenn das zu lösende Problem den Aufwand $f(n)$ hat.

Aufwand $f(n)$ des Problems	In 1 Stunde lösbare Problemgröße mit Prozessor A	mit Prozessor B
n	$3.6 \cdot 10^{10}$	$7.2 \cdot 10^{10}$
$2 \cdot n$	$1.8 \cdot 10^{10}$	$3.6 \cdot 10^{10}$
$n \cdot \log(n)$	$\approx 1.2 \cdot 10^9$	$\approx 2.3 \cdot 10^9$
$2 \cdot n \cdot \log(n)$	$\approx 6.2 \cdot 10^8$	$\approx 1.2 \cdot 10^9$
n^2	$\approx 1.9 \cdot 10^5$	$\approx 2.7 \cdot 10^5$
$2 \cdot n^2$	$\approx 1.3 \cdot 10^5$	$\approx 1.9 \cdot 10^5$
n^3	≈ 3300	≈ 4160
$2 \cdot n^3$	≈ 2620	≈ 3300
2^n	≈ 35	≈ 36
3^n	≈ 22	≈ 22

Bzgl. der O-Notation unterscheiden wir die Effizienz der Algorithmen, die sich obiger Tabelle nur um den Faktor 2 unterscheiden, nicht (2^n und 3^n sind bzgl. O-Notation verschieden, wie wir in den Übungen sehen werden). Wir sehen auch, dass uns Laufzeitgewinne von einem konstanten Faktor nicht helfen, wirklich signifikant größere Probleme zu lösen. Dies wird mit der O-Notation ebenfalls abgebildet.

$\rightsquigarrow$ Die O-Notation definiert die Umschreibung „höchstens um einen konstanten Faktor größer und nur endliche viele Ausnahmen“ formal.

Es gilt für Funktionen $f, g : \mathbb{N}_0 \to \mathbb{R}_0^+$:

- $f \in O(g) \;:\Leftrightarrow\; \exists c \in \mathbb{R}^+,\; n_0 \in \mathbb{N} :\; f(n) \leq c \cdot g(n),\; n \geq n_0$,
  d. h., f wächst asymptotisch nicht schneller als g – wir können hier die Eselsbrücke „O“ wie obere Schranke (für die Laufzeit) benutzen
- $f \in o(g) \;:\Leftrightarrow\; \forall c \in \mathbb{R}^+,\; \exists n_0 \in \mathbb{N} :\; f(n) \leq c \cdot g(n),\; n \geq n_0$,
  d. h., f wächst asymptotisch echt langsamer als g
- $f \in \Omega(g) \;:\Leftrightarrow\; g \in O(f)$,
  d. h., f wächst asymptotisch nicht langsamer als g
- $f \in \omega(g) \;:\Leftrightarrow\; g \in o(f)$,
  d. h., f wächst asymptotisch echt schneller als g
- $f \in \Theta(g) \;:\Leftrightarrow\; f \in O(g) \wedge g \in O(f)$,
  d. h., f wächst asymptotisch gleich schnell wie g

Wenn wir für einen Algorithmus die Laufzeit in O-Notation angeben wollen, werden wir stets versuchen, eine möglichst langsam wachsende Funktion zu finden, die den Aufwand des Algorithmus im schlechtesten Fall beschreibt.

Untersuchen wir das Laufzeitverhalten des Sortierens durch Minimumsuche, so werden wir unabhängig von den konkreten Zahlen im sortierenden Array stets etwa $n^2/2$ Vergleiche benötigen und etwa ebenso viele Schritte in den Schleifen. Variabel ist hier lediglich, wie oft wir bei der Suche nach dem Index des kleinsten Elements ein kleineres gefunden haben und uns nun den neuen Index merken. Die Gesamtzahl der Schritte wird also etwa irgendwo zwischen 2 und 2.5 mal n^2 liegen (je nachdem, welche Einheit wir als Schritt auffassen, können die Faktoren auch etwas höher oder niedriger sein).

Wir haben also in jedem Ablauf einen Aufwand, der proportional zu n^2 ist – in O-Notation sagen wir: der Algorithmus hat den Aufwand $O(n^2)$ – wir können sogar genauer sagen: der Algorithmus hat den Aufwand $\Theta(n^2)$, da auch im besten Fall der Aufwand proportional zu n^2 ist.

Betrachten wir als weiteres Beispiel die Intervallschachtelung: Hier kann man Glück haben, dass man das Element zufällig nach dem ersten Vergleich bereits genau in der Mitte gefunden hat (Aufwand konstant viele Schritte), im schlimmsten Fall müssen wir das Intervall jedoch logarithmisch oft halbieren. In O-Notation suchen wir nun für diesen schlechtesten Fall eine möglichst langsam wachsende Funktion und finden als Ergebnis schließlich $O(\log(n))$. Dies entspricht nun auch obiger umgangssprachlicher Formulierung, denn der Aufwand für die Intervallschachtelung ist für beliebige Eingaben „höchstens um einen konstanten Faktor größer als $\log(n)$“.

Formal müssten wir jeweils Konstanten c und n_0 finden, so dass die Ungleichung in der Definition erfüllt ist. In der Praxis hilft uns die Grenzwertbestimmung aus der Mathematik weiter und wir können uns oft folgender Technik bedienen:

Für fast alle Fälle kann man über $\lim_{n\to\infty}(f(n)/g(n))$ bestimmen, in welchem Verhältnis bzgl. O-Notation die beiden Funktionen f und g zueinander stehen.

- Divergiert der Quotient ($\lim \to \infty$), so wächst f echt schneller (also $f \in \omega(g)$ bzw. $g \in o(f)$),
- bei $\lim \to 0$ wächst f echt langsamer (also $f \in o(f)$ bzw. $f \in \omega(f)$),
- gilt $\lim \to c$ für eine Konstante ungleich 0, so unterscheiden sich die beiden Funktionen ab einem n_0 nur noch maximal um einen konstanten Faktor, somit $f \in O(g)$ und $g \in O(f)$ und somit auch $f \in \Theta(g)$ und $g \in \Theta(f)$).

Existiert der Grenzwert nicht, dann sind die Funktionen entweder bzgl. O nicht vergleichbar (in der Praxis selten) oder der Quotient bewegt sich in einem beschränkten Bereich, ist aber nicht konvergent (ebenfalls selten), Beispiele selbst überlegen.

Im weiteren Verlauf werden wir uns im Wesentlichen auf das O-Symbol beschränken (und die weiteren 4 Möglichkeiten der O-Notation nicht weiter beachten).

2.3.1 Programmkonstrukte und deren Laufzeiten

Um für ein Programm oder ein Programmfragment die Laufzeit zu bestimmen, muss man den Aufwand für die einzelnen Teile zusammenaddieren. Folgende Übersicht kann uns dabei helfen.

- Folge von Anweisungen: Aufwand ist die Summe der Einzelaufwände.
- `for`-Schleifen (bedingt auch `while`-Schleifen): Anzahl der Durchläufe mal Aufwand für den Schleifenrumpf – ist der Aufwand abhängig von der Schleifenvariable, so können wir entweder uns mit einer groben Abschätzung begnügen (indem wir für den Aufwand für den Schleifenrumpf das Maximum der möglichen Fälle verwenden) oder die genaue Summe über die Einzelaufwände betrachten (siehe Beispiel weiter unten).
- Fallunterscheidung: hier betrachten wir die Summe(!) der Einzelaufwände (eigentlich reicht das Maximum, überlegen Sie selbst, dass die Summe hier bzgl. O-Notation ebenfalls korrekt ist).
- Rekursion: führt auf Gleichungen für den Aufwand – hier hilft nur Übung um zu erkennen, welcher Aufwand letztlich nötig ist. Hat man eine Idee, was rauskommen könnte, so kann man versuchen, über die Gleichungen und mit vollständiger Induktion die Vermutung zu beweisen.

Führt man die Analyse so durch, erhält man oft ein Polynom der Form $a \cdot n^k + b \cdot n^{k-1} + \cdots + c$, für die Angabe in O-Notation ist dann nur das n^k entscheidend (die anderen Summanden wachsen ja mit mindestens dem Faktor n langsamer, diese werden für größere n also keine Rolle spielen) und wir geben als Aufwand $O(n^k)$ an (ohne den Faktor a). Wir werden dies in den Übungen noch vertiefen.

Als kleines Beispiel betrachten wir das Programmfragment, was bei der Übungsaufgabe zu den Phythagoräischen Zahlentripeln entstanden sein könnte:

```
for a in 1..n loop
   for b in a..n loop
      for c in b..n loop
         ... -- etwas mit konstantem Aufwand k
      end loop;
   end loop;
end loop;
```

Nehmen wir die grobe Abschätzung für die `for`-Schleifen, so erhalten wir den Aufwand $n \cdot n \cdot n \cdot k$. Da k eine Konstante ist, können wir diese in der O-Notation ignorieren und geben den Aufwand mit $O(n^3)$ an.

Wir hätten den Aufwand auch genauer abschätzen können, indem wir mit den exakten Summen gearbeitet hätten: Der Aufwand ist $\sum_{a=1}^{n} \sum_{b=a}^{n} \sum_{c=b}^{n} k$. Nach etwas Umformungsarbeit erhalten wir schließlich den Ausdruck $k \cdot (1/3 \cdot n^3 + \cdots)$. In diesem Fall wären wir mit größerem Aufwand ebenfalls auf das Ergebnis $O(n^3)$ gekommen.

Das dies nicht immer der Fall ist, werden wir spätestens im zweiten Semester bei Kürzeste-Wege-Algorithmen sehen.

2.4 Übungsaufgaben

1. **Grammatiken und Formale Sprachen:** Beim Arbeiten mit formalen Sprachen sind die in dieser Aufgabe vorkommenden Problemstellungen oft mit dabei.

   Gegeben sei die Grammatik $G_1 = (V_1, \Sigma_1, P_1, S_1)$ mit: $V_1 = \{S_1, A, E\}$, $\Sigma_1 = \{a, b, c\}$, $P_1 = \{S_1 \to AE, A \to aA, A \to \varepsilon, E \to bEc, E \to \varepsilon\}$. Die durch diese Grammatik beschriebene Sprache lässt sich auch als Menge charakterisieren: $L(G_1) = \{a^p b^q c^q | p \geq 0, q \geq 0\}$.

   Sei nun eine weitere Grammatik $G_2 = (V_2, \Sigma_2, P_2, S_2)$ gegeben mit: $V_2 = \{S_2, B, C\}$, $\Sigma_2 = \{a, b, c\}$, $P_2 = \{S_2 \to BC, B \to \varepsilon, B \to aBb, C \to \varepsilon, C \to cC\}$.

   Eine weitere Sprache L ist wie folgt gegeben: $L = \{ab^r c^s | r \geq 0, s \geq 0\}$.

   (a) Geben Sie die Mengencharakterisierung der Sprache $L(G_2)$ an.

   (b) Geben Sie eine kontextfreie Grammatik G_0 an, so dass gilt: $L(G_0) = L(G_1) \cup L(G_2)$. Geben Sie die Mengencharakterisierung der Sprache $L(G_0)$ an.

   (c) Geben Sie eine kontextfreie Grammatik G_3 an, so dass gilt: $L = L(G_3)$.

   (d) Geben Sie eine kontextfreie Grammatik G_4 an, so dass gilt $L(G_4) = L(G_1) \cap L(G_3)$. Geben Sie die Mengencharakterisierung der Sprache $L(G_4)$ an.

   (e) Geben Sie eine kontextfreie Grammatik G_5 an, so dass gilt $L(G_5) = L(G_3) \setminus L(G_1)$. Geben Sie die Mengencharakterisierung der Sprache $L(G_5)$ an.

2. **Binärbrüche:** Wie bei Dezimalzahlen können wir auch Binärzahlen in Vor- und Nachkommastellen (getrennt durch einen '.') aufteilen. Ein Binärbruch sei also definiert als eine Sequenz von 0 und 1 mit genau einem Punkt, so dass vor und nach diesem jeweils mindestens eine Ziffer steht. Vor der ersten Ziffer steht optional noch ein Vorzeichen (Minus oder Plus) und eine beliebige Anzahl von Leerzeichen.

(a) Beschreiben Sie die Menge der Binärbrüche als EBNF, Grammatik und Syntaxdiagramm (das Syntaxdiagramm braucht nicht in eClaus abgegeben zu werden).

(b) **Zusatzaufgabe:** Verändern Sie Ihre EBNF und Grammatik so, dass unnötige führende Nullen verboten sind.

3. **Kontextfreie Grammatiken:**

(a) Gegeben sei die Sprache L, die alle korrekten Klammerungen mit runden und eckigen Klammern enthält, mit der Einschränkung, dass innerhalb einer `[]` keine `()` stehen darf. Beispiele: `(())([[]])`$\in L$, aber `([()])`$\notin L$ und ebenso `(()))(()`$\notin L$. Geben Sie die Produktionsregeln einer kontextfreien Grammatik $G = (V, \{$`(`,`)`,`[`,`]`$\}, P, S)$ an, die L erzeugt.

(b) Geben Sie die Produktionsregeln einer kontextfreien Grammatik an, welche die Sprache L' der „natürlichen Polynome zweiten Grades„ erzeugt. Ein „natürliches Polynom zweiten Grades„ ist von der Form `ax^2+bx+c` mit `b`, `c` $\in \mathbb{N} \cup \{$`0`$\}$, `a` $\in \mathbb{N}$, `a`$>$`0`. Dabei soll auf die vorgegebene Reihenfolge geachtet werden und ein Summand nur auftreten, wenn dessen Koeffizient $\neq$`0` ist. Der Faktor `1` muss weggelassen werden. Beispiele: `x^2+1`$\in L'$, `5x^2+6x`$\in L'$, `7x`$\notin L'$, `5x+6x^2`$\notin L'$, `1x^2+3x+1`$\notin L'$, `x^2+0x+3`$\notin L'$.

4. **Bergpanoramen:** Mit den Zeichen `/`, `_` und `\` lassen sich Bergpanoramen zeichnen, z.B.

```
            __
           /  \
     /\  __/\/    \
    /  \/          \
 __/                \______
```

Dabei sollen Anfang und Ende gleich hoch und nicht höher als irgendein anderer Punkt des Panoramas liegen. Wenn wir die Zeichen eines Bergpanoramas einfach von links nach rechts lesen, ohne dabei die vertikale Anordnung auf dem Papier zu beachten, so können wir ein Bergpanorama als Wort einer Sprache auffassen, in unserem Beispiel das Wort `__///\\/__/\//__\\\\_____`.

(a) Beschreiben Sie die Sprache der Bergpanoramen in EBNF. Geben Sie auch eine kontextfreie Grammatik für die Sprache der Bergpanoramen an. (Entwerfen Sie auch ein Syntaxdiagramm, dieses brauchen Sie jedoch nicht in eClaus abzugeben.)

(b) Schreiben Sie ein Ada-95-Programm, bei dem der Benutzer eine Zeichenkette eingeben kann, die in ein Bergpanorama umgesetzt werden soll.

Mit Hilfe von `Get_Line(<string>,<integer>)`, kann man in eine String-Variable eine Zeile Text einlesen. Die Integer-Variable (ein Ausgangsparameter) gibt danach an, wieviel Zeichen eingelesen wurden.

Die eingelesenen Zeichen sollen nun als Bergpanorama ausgegeben werden: Um es uns etwas einfacher zu machen, betrachten wir dabei den linken Bildschirmrand als Grundlinie – der Benutzer muss also den Kopf um 90 Grad drehen. Um es dem Benutzer etwas einfacher zu machen, passen wir die Zeichen entsprechend an. Die Eingabe `/_\_` würde dann zu der Ausgabe

```
\
 |
/
|
```

D.h., ein `'/'` wird als `'\'` ausgegeben und umgekehrt und statt dem `'_'` geben wir ein `'|'` aus. Achten Sie auch darauf, dass die Einrückungen korrekt sind. Entspricht die Eingabe keinem Bergpanorama, so soll dies dem Benutzer mitgeteilt werden.

(c) Geben Sie das Bergpanorama nun auch richtig herum aus. Die Eingabe `/_\_` soll also die Ausgabe

```
 _
/ \_
```

erzeugen. (Hinweis: Es empfiehlt sich hier ein schrittweises Vorgehen: Definieren Sie sich zunächst eine Datenstruktur, die für jede Position im Bergpanorama die zugehörige Höhe angibt; Sie können dann jede Schicht einzeln ausgeben – beginnend mit der maximalen Höhe (definieren Sie sich zur Berechnung eine entsprechende Funktion) bis zum Tal (Höhe 0).)

(d) Schreiben Sie ein Ada-95-Programm, das zufällige Bergpanoramen erzeugt, indem der Ableitungsprozess eines Wortes der Sprache simuliert wird. (Führen Sie für jedes Nichtterminal eine Prozedur ein, die ggf. über eine Zufallszahl eine ihrer Regeln auswählt und die entsprechenden Zeichen in eine globale String-Variable (unser Bergpanorama) schreibt. Das Panorama kann dann mit Hilfe unserer Prozedur aus Teil (b) oder (c) ausgegeben werden. Hinweis: Achten Sie darauf, dass der Ableitungsprozess irgendwann enden muss, d.h., die Wahrscheinlichkeit für die Regeln ohne Nichtterminalsymbole muss ausreichend groß sein.

5. **Eindeutigkeit von Grammatiken:** Gegeben sei die Sprache $L = \{a^i b^j a^k b^l \mid i = j \ \vee \ k = l, \ i,j,k,l \geq 0\}$, bei der entweder zu Beginn eine gleich lange Folge von a und b auftritt (und danach beliebig viele a und danach beliebig viele b) oder zu Beginn beliebig viele a und dann b gefolgt von gleichvielen a wie b.

   Eine mögliche Grammatik für diese Sprache hat die Produktionsregeln
   $P = \{S \to AB, \ S \to BA, \ A \to aAb, \ A \to \varepsilon, \ B \to aB, \ B \to C, \ C \to bC, \ C \to \varepsilon\}$.

   (a) Zeigen Sie, dass diese Grammatik mehrdeutig ist (z.B. durch zwei wesentlich verschiedene Ableitung eines Wortes). Geben Sie eine Mengencharakterisierung der Wörter an, die bei dieser Grammatik mehrdeutig sind.

   (b) Geben Sie eine eindeutige Grammatik für die Sprache L an. Beschreiben Sie Ihre Idee und begründen Sie, warum die Grammatik eindeutig ist.

6. **Virenscanner:** Wir haben gesehen, dass es Probleme gibt, die algorithmisch nicht lösbar sind. In der heutigen Zeit wäre ein universeller Virenscanner V ein sehr hilfreiches Werkzeug. Dieser soll ein Programm P als Eingabe bekommen und ausgeben, ob P das System mit einem Virus infizieren wird oder nicht.

   Zeigen Sie, dass es solch einen universellen Virenscanner V nicht geben kann.

   (Hinweis: die Eingabe für V ist das zu überprüfende Programm P – dies ist vergleichbar mit dem Ada-95-Übersetzer, auch dieser bekommt als Eingabe ein Programm (den von Ihnen geschriebenen Ada-Sourcecode) und liefert eine Ausgabe (das in Maschinensprache übersetzte Programm).)

7. **Definition O-Notation:** Die im Skript angegebenen Definitionen sind nicht einzige Möglichkeit die O-Notation zu definieren. Welche der folgenden Definitionen und Aussagen sind richtig? Begründen Sie Ihre Antworten.

   (a) i. $O(f) := \{g : \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists n_0 \in \mathbb{N}\ \forall n \geq n_0\ \exists c \in \mathbb{R}^+ : g(n) \leq c \cdot f(n)\}$
   ii. $O(f) := \{g : \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists n_0 \in \mathbb{N}\ \forall n \geq n_0\ \exists c \in \mathbb{R}^+ : c \cdot g(n) \leq f(n)\}$
   iii. $O(f) := \{g : \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$
   iv. $O(f) := \{g : \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists n_0 \in \mathbb{N}\ \exists c \in \mathbb{R}^+\ \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$
   v. $O(f) := \{g : \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : c \cdot g(n) \leq f(n)\}$
   vi. $O(f) := \{g : \mathbb{R}^+ \to \mathbb{R}^+ \mid \forall c \in \mathbb{R}^+\ \exists n_0 \in \mathbb{N}\ \exists n \geq n_0 : g(n) \leq c \cdot f(n)\}$
   vii. $O(f) := \{g : \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+\ \forall n_0 \in \mathbb{N}\ \exists n \geq n_0 : c \cdot g(n) \leq f(n)\}$

   (b) i. $f \in O(g) \Rightarrow g \in O(f)$
   ii. $f \in o(g) \Rightarrow g \in o(f)$
   iii. $f \in \Theta(g) \Rightarrow g \in \Theta(f)$
   iv. $f \in o(g) \Rightarrow g \in \omega(f)$
   v. $f \in O(g) \Rightarrow g \in O(f + g)$
   vi. $f \in o(g) \Rightarrow g \in O(f + g)$
   vii. $f \in o(g) \Rightarrow g \in o(f + g)$

   (c) $f \in O(g) \Leftrightarrow \lim_{n\to\infty}(f(n)/g(n)) = c,\ c \in \mathbb{R}$

8. O**-Notation:** Begründen Sie Ihre Antworten.

   (a) Zeigen oder widerlegen Sie folgende Aussagen:
   - $3 \log_{10} n \in O(\log_2 n)$
   - $(n + a)^b \in O(n^b)$, $a, b \in \mathbb{R}$ und $b > 0$
   - $3^n \in O(2^n)$

   (b) Welche Beziehungen $(=, \subseteq, \supseteq)$ gelten zwischen den folgenden Funktionenklassen:
   - $O(n\sqrt{n})$ und $O(n \log^2(n))$
   - $O(3^n)$ und $O(n^n)$
   - $O(2^{n+1})$ und $O(2^n)$
   - $O(2^{2n})$ und $O(2^n)$

   (c) Geben Sie eine Funktion an, die in $o(1)$ (klein-o von 1) liegt, aber nicht konstant 0 ist.

9. O**-Notation II:** Irgendetwas stimmt da nicht ...

   Behauptung: $f(n) = 2^n \in O(1)$
   Beweis: Vollständige Induktion nach n. Die Behauptung ist richtig für $n = 1$, denn $2^1 = 2$ ist konstant, also in $O(1)$. Mit der Induktionsannahme $2^{n-1} \in O(1)$ folgt nun sofort, dass $2^n = 2 \cdot 2^{n-1} = 2 \cdot O(1) = O(1)$.

   Finden Sie heraus, was hier nicht stimmt.