Stand der Folien: 14.6.04

Grundbegriffe der Informatik

und

Programmieren in Java

Fortbildungskurs für Lehrkräfte der Sek. II

Volker Claus, Stephan Wilczek

Ort: Universität Stuttgart, Neubau Informatik
Zeit: Erstes Halbjahr 2004, 6.3. bis 12.6.04

Aktuelles und Dokumente finden Sie unter
http://www.schule-und-informatik.de

Anschriften:

Prof. Dr. Volker Claus, Universität Stuttgart, FMI, Universitätsstr. 38, 70569 Stuttgart.
Tel.: 0711 / 78 16 - 300, Fax: - 310, e-mail: claus@informatik.uni-stuttgart.de
Dipl.-Kfm. Stephan Wilczek, iMEDIC GmbH, Martinstrasse 42-44, 73728 Esslingen.
Tel.: 0711 / 45 99 98 - 10, Fax: - 29, e-mail: wilczek@imedic.de

Vorgesehene Termine:

6.3., 13.3., 20.3., 27.3., 3.4., 24.4., 8.5., 15.5., 19.5., 9.6. und Abschluss am 12.6.04.
Samstags jeweils 9:30 bis 12:45 Uhr, mittwochs 16:00 bis 19:15 Uhr.
Ort: Hörsaal 38.02 oder ein geeigneter Seminarraum.
Praktische Übungen an Rechnern: Grundstudiumspool.

Bescheinigungen:

Das Oberschulamt hat die Veranstaltung unter dem Az. II-6750.5/187 (Makowsky) als Fortbildungsmaßnahme anerkannt. Eine "vollständige" Teilnahmebescheinigung erhält, wer höchstens zwei Mal gefehlt hat.
Eine erfolgreiche Teilnahme wird bescheinigt, sofern mindestens 8 Programme selbst erstellt, dokumentiert und zum Teil vorgestellt wurden.
Auf Wunsch können zusätzliche qualifizierte Bescheinigungen (Klausurscheine) ausgestellt werden, sofern die Klausur "Grundbegriffe" am 19.6., 9:30 bis 11:00 Uhr und/oder die Klausur "Java" am 26.6., 9:30 bis 11:00 Uhr bestanden wurde(n).

Diese Folien umfassen nur den Teil "Grundbegriffe der Informatik"

Literatur-Auswahl: (Es gibt noch viel mehr, schauen Sie in die Bibliothek)

Appelrath, Hans-Jürgen und Ludewig, Jochen, "Skriptum Informatik - eine konventionelle Einführung", Verlag der Fachvereine Zürich und B.G. Teubner Stuttgart, 4. Auflage 1999

Balzert, Helmut, "Lehrbuch Grundlagen der Informatik", Spektrum Akademischer Verlag, Heidelberg 1999 (enthält auch eine Einführung in Java und UML)

Broy, Manfred, „Informatik. Eine grundlegende Einführung“. Band 1: Programmierung und Rechnerstrukturen, Springer-Verlag, 1998. Band 2: Systemstrukturen und Theoretische Informatik, Springer-Verlag, 1998

Cormen, Leiserson, Rivest, "Introduction to Algorithms", MIT Press, 1996

Goos, Gerhard, "Vorlesungen über Informatik", Band 1 und 2, dritte Auflage, Springer-Verlag, Berlin 2000 und 2001

Güting, R.H., "Datenstrukturen und Algorithmen", B.G.Teubner Stuttgart, Neuauflage 2002

Klaeren, Herbert, "Vom Problem zum Programm", B.G. Teubner Stuttgart, 1990

Ottmann, T., Widmayer, P., "Algorithmen und Datenstrukturen", Spektrum Verlag, Heidelberg, 2002

Schöning, Uwe, "Algorithmik", Spektrum Akademischer Verlag, Heidelberg 2001

Sedgewick, Robert, "Algorithms in C", 3rd Edition, Addison-Wesley, 1998

sowie

Als Einstieg: Appelrath, Boles, Claus, Wegener, "Starthilfe Informatik", Teubner-Verlag, Stuttgart-Leipzig, 2. Auflage, 2001.

Für diverse Definitionen und Erläuterungen: "Duden Informatik", dritte Auflage, Bibliografisches Institut, Mannheim, 2001.

Für Mathematik und Ideen: Meinel, Christoph, Mundhenk, Martin, „Mathematische Grundlagen der Informatik“, Teubner-Verlag, Wiesbaden, 2. Auflage, 2002 (für andere Studiengänge).

Schöning, Uwe, „Ideen der Informatik“, Oldenbourg-Verlag, München, 2002.

Aufbau der Veranstaltung "Grundbegriffe der Informatik"

Es werden mehrere Einzelthemen behandelt. *Geplant sind derzeit:*

0. Objekte und Interaktion. Programme.
1. Aufbau einfacher Algorithmen (Kontrollstruktur, elementare Datentypen).
2. Einfache Datenstrukturen (Feld, Verbund, Vereinigung).
3. Begriff der Sprache, Grammatik/BNF und formale Sprache.
4. Datentypen, Beispiele (Boolean, Keller, Prioritätswarteschlange).
5. Iteration und Rekursion. Prozeduren und Funktionen.
6. Zeit- und Platzkomplexität. Groß-O.
7. Unentscheidbare Probleme, Terminierung.
8. Korrektheit und Beweiskalkül.
9. Speicherstrukturen, Speicherbereiche, Halde.
10. Eigene Definition von Programmiersprachen.
11. Listen, Bäume, Graphen.
12. Suchen und Sortieren.
13. Gültigkeitsbereiche, Polymorphie, Vererbung, Klassen.
14. Graphalgorithmen.

Dieser Kurs kommt mit rund 850 Minuten Vortragsdauer nur bis Kapitel 7.

0. Objekte und Interaktion. Programme.

Wie funktionieren Abläufe im menschlichen Bereich?
Kommunizierende Einheiten, Senden von Nachrichten.
Aufbau einer Einheit?
- Beschreibung über Eigenschaften.
- Zustände.
- Kapselung der Daten.
- Geheimhalten, wie eine Handlung ausgeführt wird.

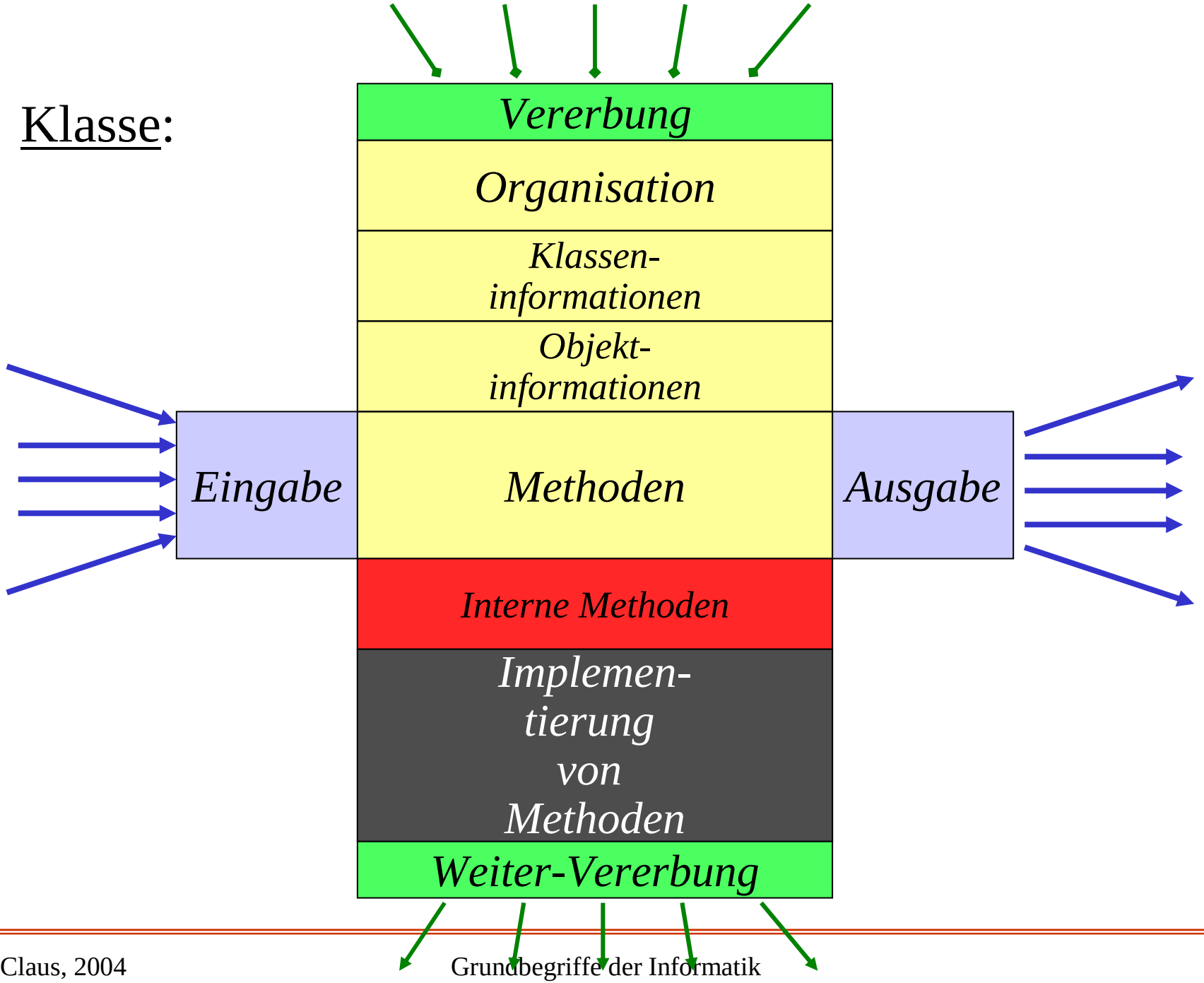
Übernehmen von Definitionen, die es bereits gibt.
Parameter.
Wie kann man hiermit Handlungen ablaufen lassen?
Klasse, Objekt (= Instanz einer Klasse), Programm.

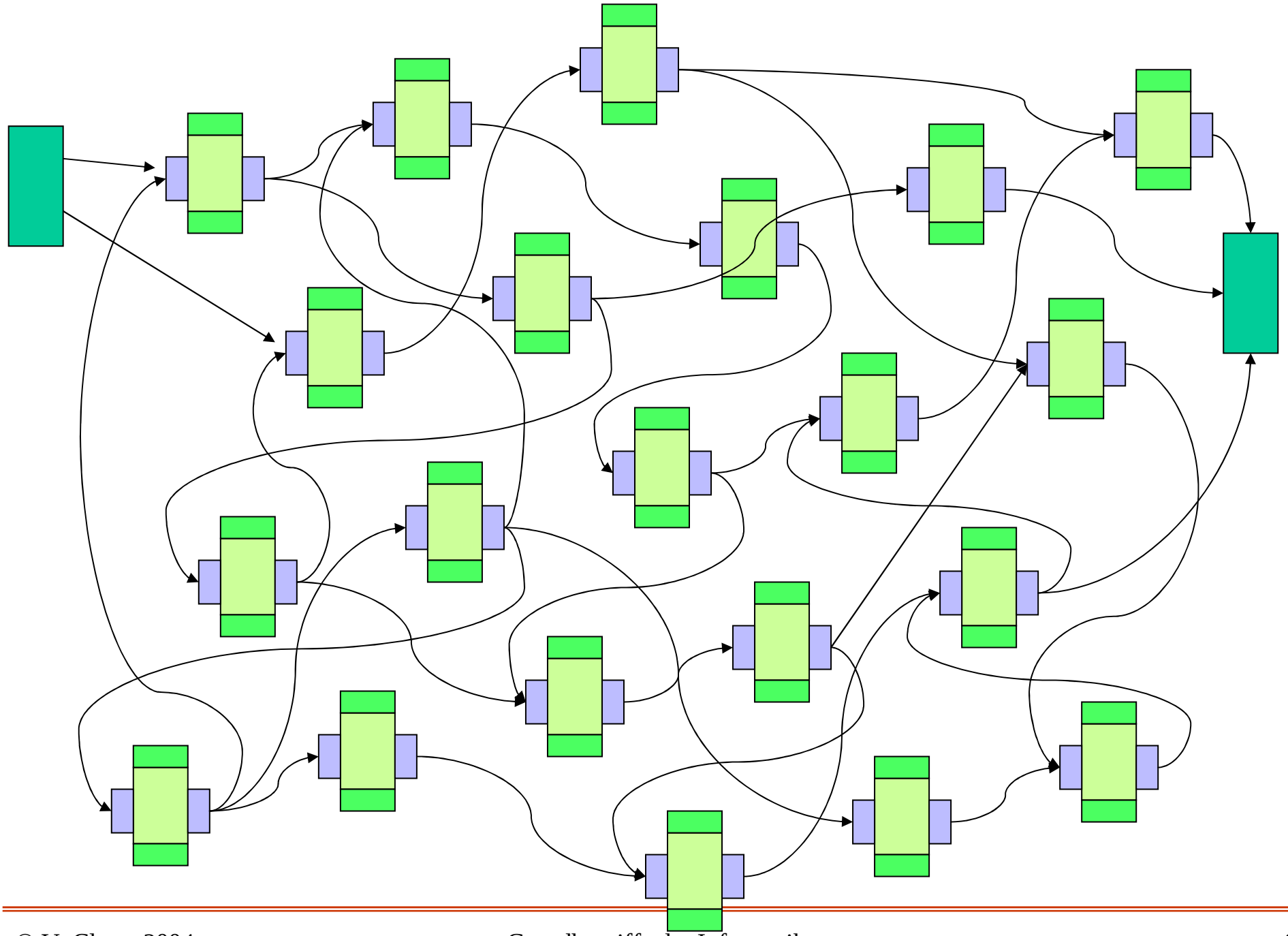
Objekte

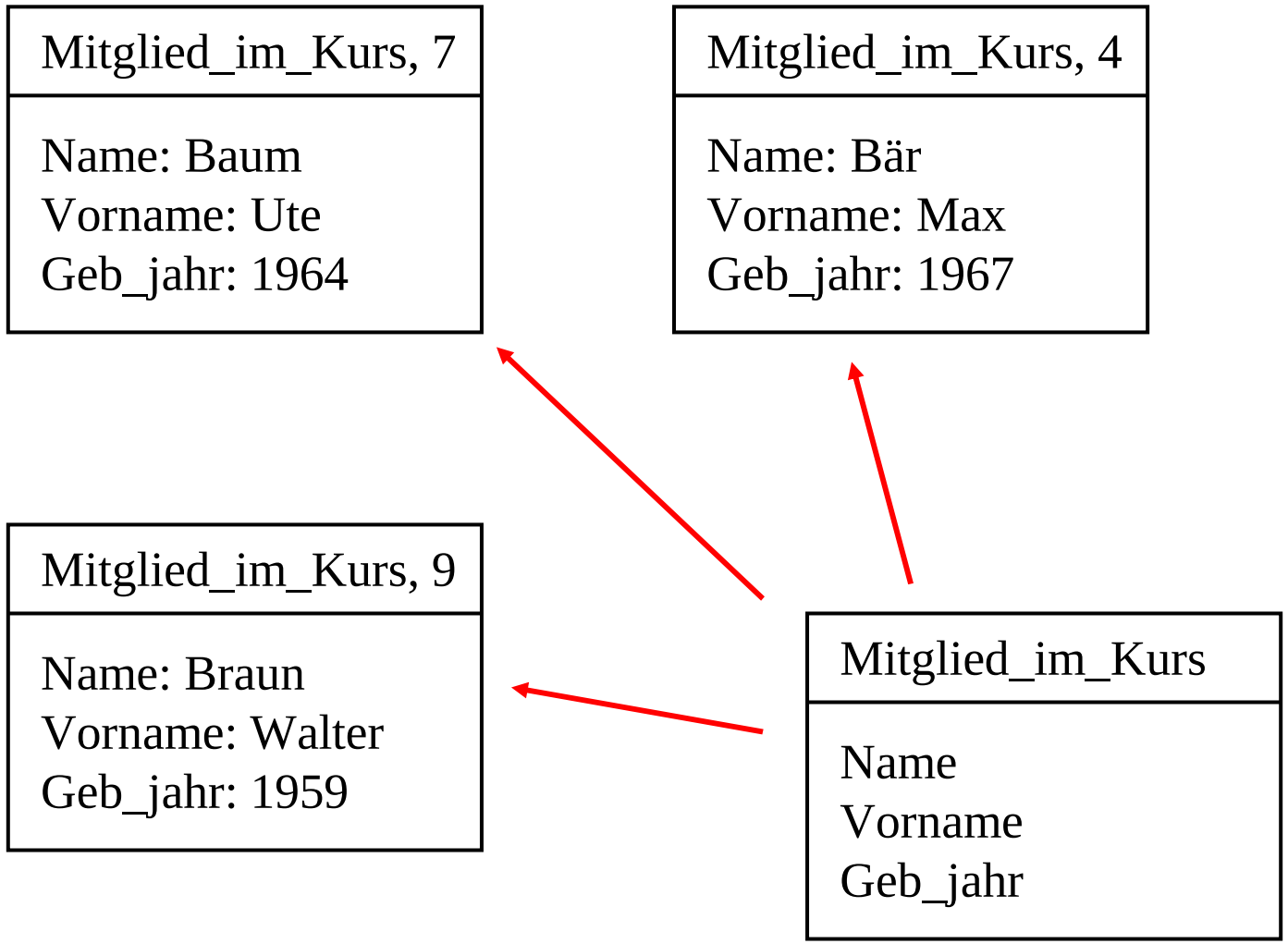
Objekte sind in sich geschossene Einheiten, die

- wie Moduln aufgebaut sind: Es gibt ein Schema, genannt "Klasse", das vor allem aus "Attributen" (das sind die veränderlichen Werte und ihre Strukturen) und "Methoden" (das sind die algorithmischen Teile) besteht und aus dem ein neues Objekt erzeugt werden kann; das Objekt ist eine Instanz (oder ein "Exemplar" oder eine "Ausprägung") dieser Klasse.
- einen individuellen Zustand besitzen (Speicherzustand der Klassen- und Instanzvariablen),
- miteinander kommunizieren können; dies geschieht durch Nachrichtenaustausch ("message passing"),
- durch Vererbung ihre Eigenschaften an neue Objekte bzw. Klassen weitergeben können.

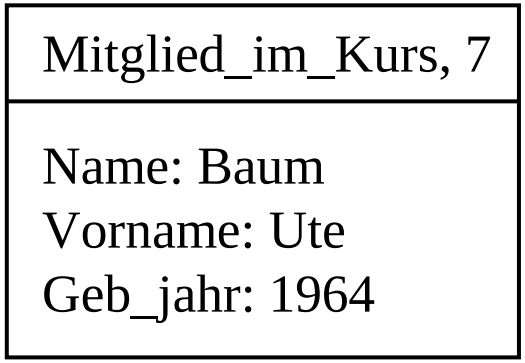
Klasse:


Vererbung
Organisation
Klassen-
informationen
Objekt-
informationen
Eingabe
Methoden
Ausgabe
Interne Methoden
Implemen-
tierung
von
Methoden
Weiter-Vererbung



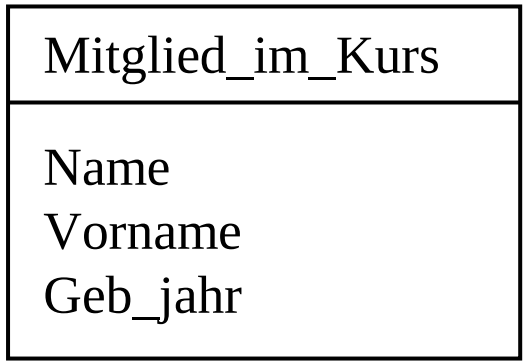


Klasse als Schema für Objekte

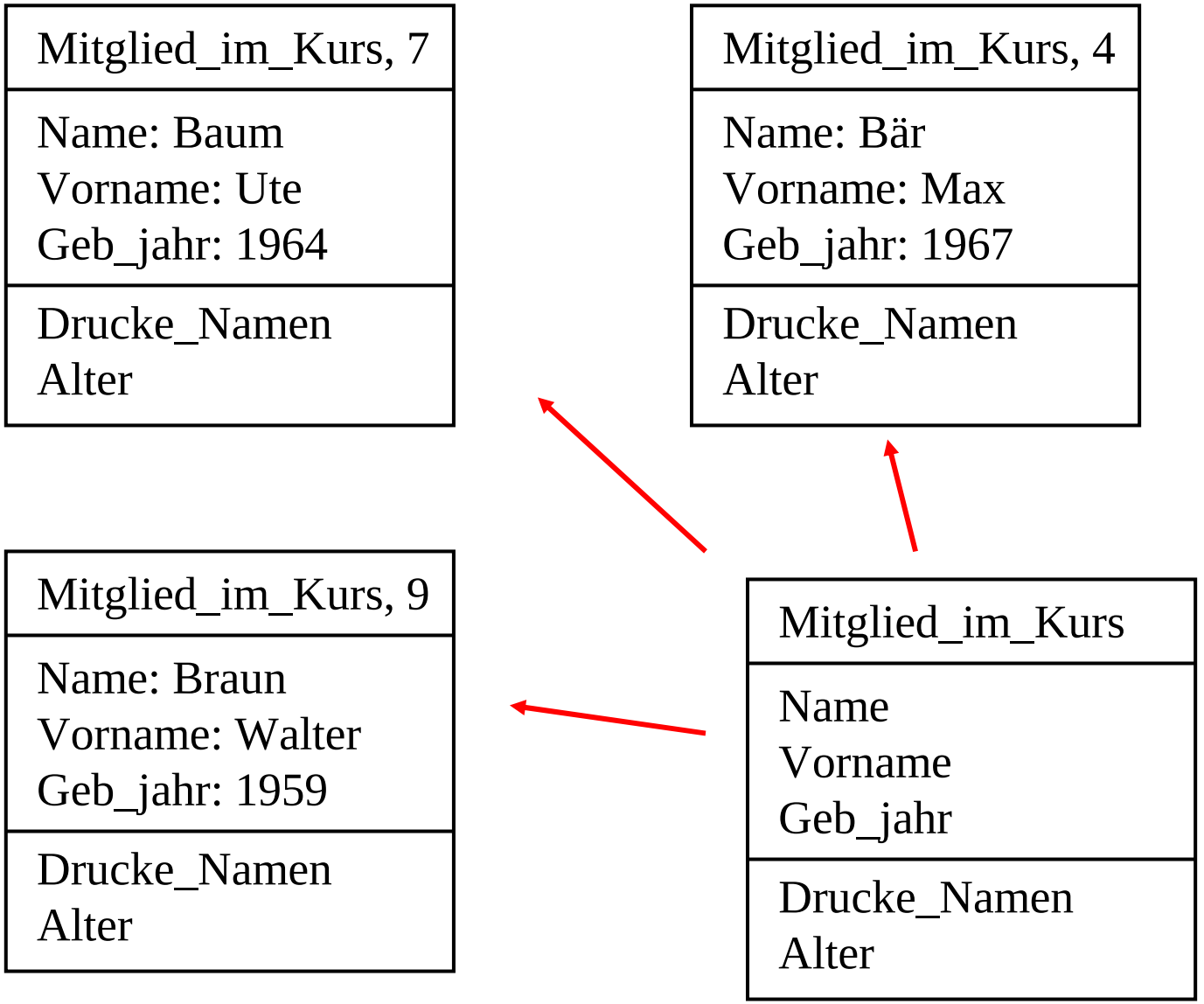


Mitglied_im_Kurs_7 =
new Mitglied_im_Kurs
("Baum", "Ute", 1964);

```
class Mitglied_im_Kurs {
// Attribute
  String Name;
  String Vorname;
  int Geb_jahr;
}
```



Formulierung in einer Sprache

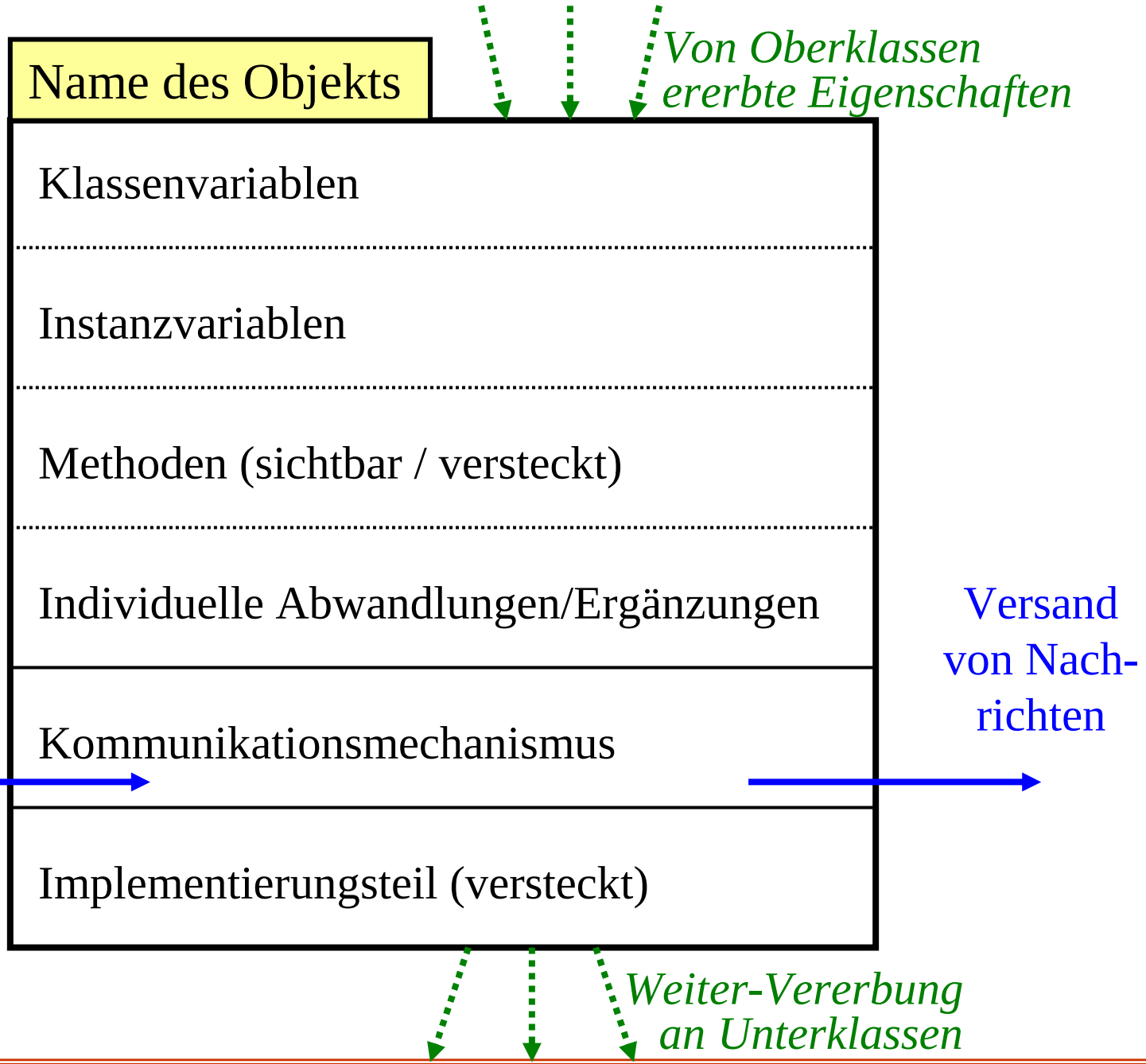


Methoden hinzufügen

Prinzipien der Objektorientierung:

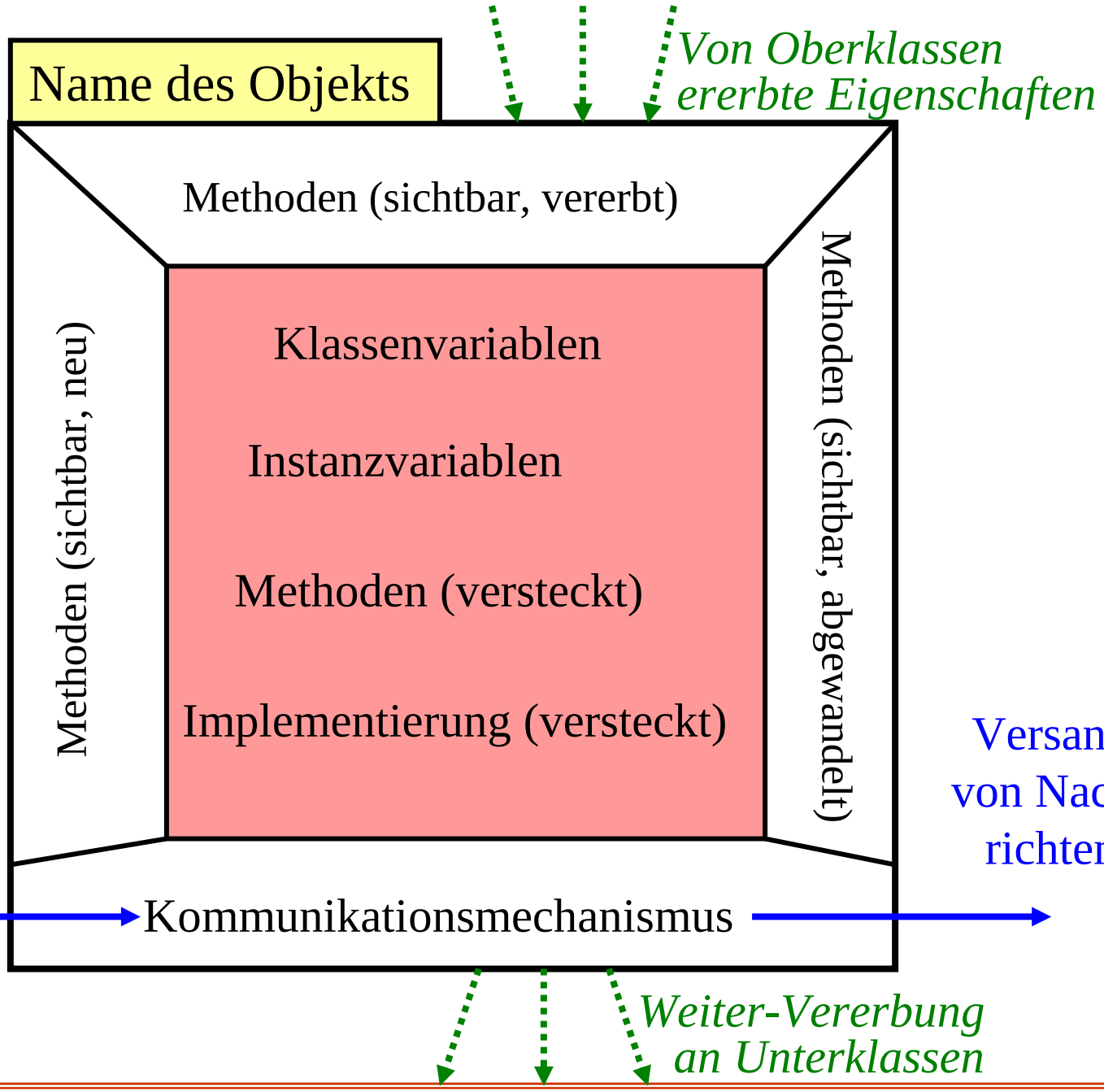
1. Es gibt nur Objekte. Jedes Objekt ist eindeutig identifizierbar über seinen Namen.
2. Alles wird über Klassen, Instanzbildung, Zustände, Methoden, Nachrichten und Vererbung realisiert.
3. Objekte handeln in eigener Verantwortung (und sie geben nur bekannt, *was* sie bearbeiten, niemals, *wie* sie dies tun).
4. Klassen werden in Bibliotheken aufbewahrt und stehen allen Programmen und Klassendefinitionen zur Verfügung.
5. Programmieren bedeutet, Klassen festzulegen, hieraus Objekte zu erzeugen und diesen Aufgaben zu übertragen, indem man ihnen geeignete Nachrichten schickt. Die Auswertung der Objekte erfolgt hierbei erst zur Laufzeit (Polymorphie, dynamische Bindung der Objekte an Variable bzw. Bezeichner).

Allgemeine Darstellung



Bessere Darstellung:

Kapselung von Zustand und Implementierung



Wir definieren eine Klasse "Punkt" (bekannt sei die Klasse real, die Formulierung erfolgt in keiner richtigen Programmiersprache):

Punkt	
real X; real Y;	
procedure Drucken	

procedure Drucken {
< sende an das Objekt *Drucker* die Nachricht drucke_ein_Pixel_an_die_Position (this.X, this.Y) >
}	

Hinweise:

"real" bezeichnet die Klasse der reellen Zahlen.

procedure leitet die Definition einer Handlung ein.

this bezeichnet das Objekt, in dem man sich hier befindet.

Die Punktnotation bewirkt: "ins Innere des Objekts gehen" (z.B. this.X).

Spitze Klammern enthalten Hinweise, was dort zu program-mieren ist.

Wir definieren nun hiermit eine Klasse "Strecke":

Strecke	Punkt
Punkt Anfang; Punkt Ende;	
procedure Drucken, DruckeAnfang, DruckeEnde; function Länge ( ) return real;	

```
procedure Drucken {
< sende an das Objekt Drucker die Nachricht
    drucke_eine_ Strecke_von_bis (this.Anfang.X,
            this.Anfang.Y, this.Ende.X, this.Ende.Y ) >  };
procedure DruckeAnfang { Anfang.Drucken; };
procedure DruckeEnde { Ende.Drucken; };
function Länge ( ) return real
{ return (square_root(quad(this.Anfang.X - this.Ende.X)
            + quad(this.Anfang.Y - this.Ende.Y))); }
```

Wir definieren nun eine Klasse "Polygon" mit dem Parameter N, um N Punkte festzulegen (Oberklassen sind Strecke und Punkt):

Polygon (N: natural, N > 2)	Strecke

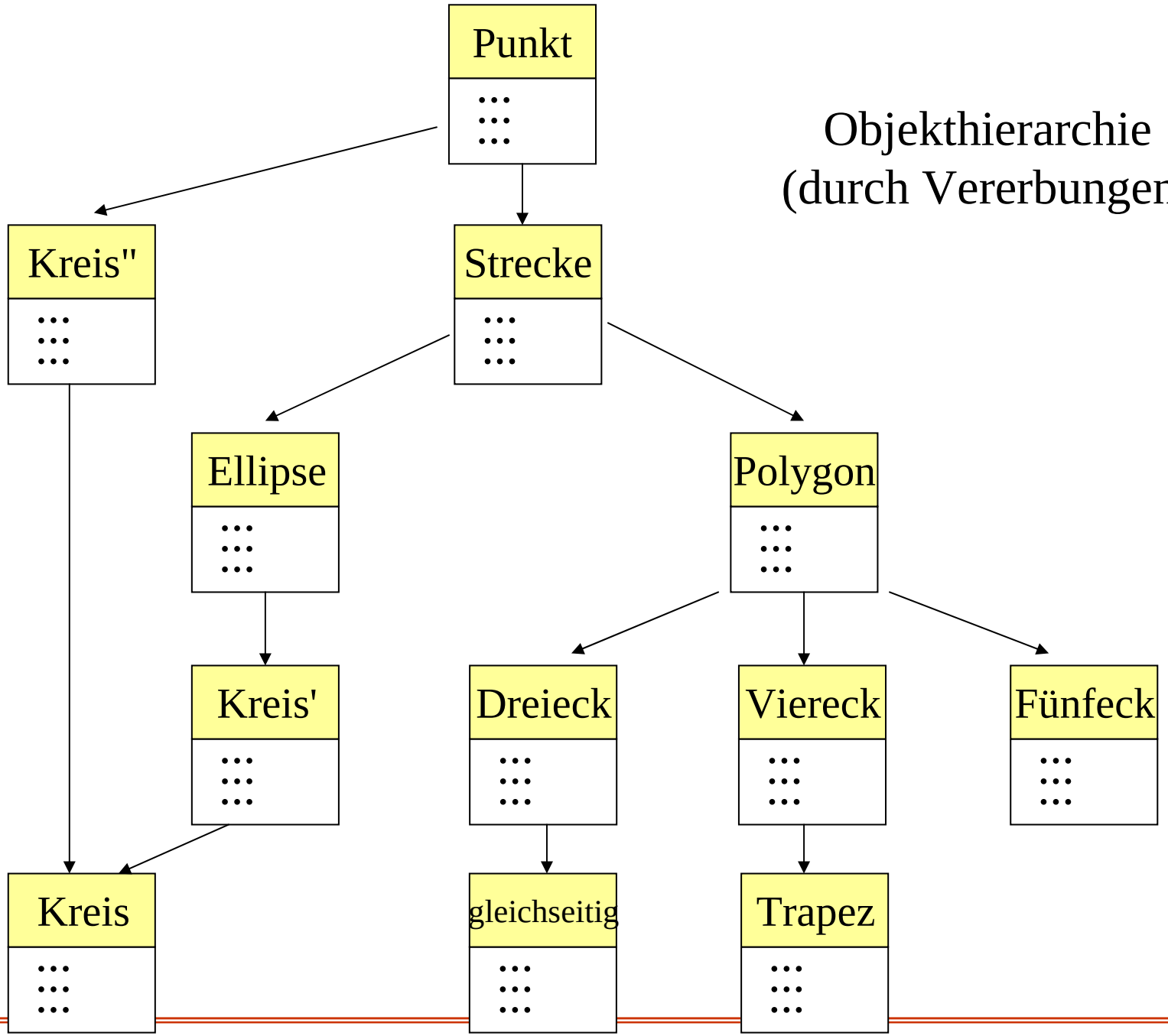
```
Eckpunkte: array [1..N] of Punkt;
private  Streckenzug: array [1..N] of Strecke;
```

```
procedure Drucken; function Länge ( ) return real
```

```
procedure Drucken is  natural I;
begin for I := 1 to N do Streckenzug[I].Drucken od  end;
function Länge ( ) return real is natural I; real L := 0.0;
begin for I := 1 to N do L := L + Streckenzug[I].Länge od; return L end;
```

```
var  natural  I, K;
begin for I := 1 to N do Streckenzug[I] := new Strecke;
            Streckenzug[I].Anfang := Eckpunkte[I];
            if I = N then K:=1 else K:=I+1 fi;
            Streckenzug[I].Ende := Eckpunkte[K]  od
end
```

Initialisierung des Feldes Streckenzug



Objekthierarchie
(durch Vererbungen)

Was geschieht nun in den Methoden?
Wie beschreibt man die auszuführenden Handlungen.
Wie beschreibt man die Daten?
Wie baut man Daten- und Handlungsbereiche auf?

Studieren Sie einen bekannten Algorithmus,
z.B. die Addition von natürlichen Zahlen.
Zunächst: Addiere 1 zu a.
(Hierzu benötigt man eine Nachfolger-Tabelle:
$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9$)
Dann: Addiere b zu a.
(z.B.: b mal "Addiere 1" durchführen oder ziffernweise addieren wie in der Grundschule gelernt; hierzu braucht man eine Additionstabelle für Ziffern.)
Dann: Erweitern auf ganze Zahlen.

1. Aufbau einfacher Algorithmen.

1.1 Motivation

Aufgabe: Addiere 1 zu einer dezimal dargestellten Zahl.

Umgangssprachliche Formulierung eines Lösungsverfahrens:
Eine Zahl sei als eine Folge von Ziffern aus der Menge {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} gegeben. Falls die letzte Ziffer nicht die 9 war, so ersetze sie durch die nächst größere Ziffer und beende das Verfahren, anderenfalls ersetze sie durch 0 und wiederhole das Verfahren für die zweitletzte Ziffer usw.

Kommentar: Hier wird erwartet, dass der, der das Verfahren ausführt, ein wenig mitdenkt oder ein Vorwissen besitzt. So muss man wissen, dass die Ziffern in der angegebenen Reihenfolge $0 < 1 < 2 < ... < 8 < 9$ angeordnet sind, man muss wissen, was das "usw." bedeutet, und man muss wissen, was man zu tun hat, wenn es keine zweitletzte Ziffer gibt.

Als Mensch macht man sich einen Algorithmus an konkreten Beispielen klar. So liefert der obige Algorithmus "+ 1":

2 + 1 = 3, 44 + 1 = 45, 103 + 1 = 104,
0 + 1 = 1, 19 + 1 = 20, 199 + 1 = 200.

Nicht klar ausformuliert wurden die Fälle, in denen die Folge nur aus Neunen besteht, wie 9 + 1 = 10, 99 + 1 = 100. Hier wird unausgesprochen eine "führende Null" angenommen.

Weiterhin entspricht die Aussage, dass eine Zahl eine Folge von Ziffern sei, nicht der üblichen Anschauung, da man i.A. (außer im Falle der Null selbst) keine führenden Nullen zulässt. Das Verfahren liefert zwar 0068 + 1 = 0069, aber in der Regel schreibt man nicht 0068, sondern 68, da man nur eindeutige Darstellungen für Zahlen verwenden möchte.

Das Verfahren beschränkt die Addition der 1 auf natürliche Zahlen (einschl. der Null). Man kann 1 aber auch zu einer ganzen, einer rationalen, einer reellen oder einer komplexen Zahl addieren. Diese Erweiterung wird durch das angegebene Verfahren nicht erfasst, sondern muss durch einen neuen Algorithmus beschrieben werden.

Wie und wo man die Zahlen aufschreibt, bleibt offen. Wir denken sicher sogleich an Papier und Bleistift, doch kämen andere Völker möglicherweise nicht auf diese Realisierung. Beachten Sie, dass die Dezimaldarstellung bzw. allgemein die Darstellung in einem Stellenwertsystem zu einer Basis keineswegs naheliegend ist. Die Römer haben beispielsweise mit einem völlig anderen System gearbeitet: III + I = IV usw.

An diesem Beispiel erkennt man *einige zentrale Sprachelemente, um Algorithmen zu beschreiben.*

- Algorithmen bestehen aus einfachen Handlungen, sog. "elementaren Anweisungen". Im Beispiel sind dies: "ersetze Ziffer durch nächst größere Ziffer" oder "beende das Verfahren".

- Einzelne Handlungen können nacheinander ausgeführt werden. Im Beispiel: "ersetze Ziffer durch nächst größere Ziffer" und danach "beende das Verfahren".

- Die nächste Handlung kann von einer aktuellen Bedingung (Alternative oder Fallunterscheidung) abhängen. Im Beispiel: "Falls die Ziffer nicht die 9 war, dann ..." .

- Handlungen können wiederholt werden. Im Beispiel wird dies durch das "usw." beschrieben, welches besagt, man solle die Ersetzungen solange vornehmen, bis eine von 9 verschiedene Ziffer erreicht wird.
- Durch den Algorithmus werden irgendwelche Gebilde manipuliert. Deren anfängliche Darstellung ist anzugeben. In unserem Beispiel sind dies natürliche Zahlen und deren Darstellung als Ziffernfolgen (zur Basis 10).
- Die Gebilde, die das Ergebnis des Algorithmus sind, ergeben sich durch den Algorithmus selbst. Man sollte sie aber möglichst zuvor beschreiben können. In unserem Beispiel sind dies ebenfalls Dezimaldarstellungen.

Beispiel:
Addieren zweier dezimal dargestellter Zahlen.

Für diese Aufgabe gibt es eine einfache Formulierung eines Algorithmus: Wenn a und b zwei dezimal dargestellte Zahlen (also dargestellt als Ziffernfolgen) sind, so addiere b-mal 1 zu a.

Wie man 1 zu einer Zahl addiert, wissen wir ja bereits.

Hinweis: "addiere b-mal 1 zu a" kann missverstanden werden. Gemeint ist, dass man 1 zu a addiert, dann zum Ergebnis 1 addiert, danach zu dem neuen Ergebnis 1 addiert usw.

An diesem Beispiel erkennt man zwei weitere zentrale Sprachelemente zur Beschreibung von Algorithmen.

- Wiederhole eine Handlung b-mal. Die Anzahl der Wiederholungen ist hier also vor der ersten Ausführung bekannt und hängt nicht von einer aktuellen Bedingung ab.
- Man darf Algorithmen, die bereits anderweitig beschrieben wurden, in anderen Algorithmen verwenden. In unserem Beispiel darf man den Algorithmus "Addiere 1" für die Addition zweier Zahlen benutzen.

Weiterhin muss man sich Gedanken darüber machen, wo Zwischenergebnisse abgelegt und wie sie weiter verwendet werden.

Ein anderes, effizienteres Additionsverfahren lernten wir alle in der Grundschule.

Aufgabe:
Beschreiben Sie diesen Additionsalgorithmus möglichst präzise.

"Möglichst präzise" bedeutet hier folgendes:
Das Verfahren muss so klar und eindeutig beschrieben sein, dass es jeder ohne Rücksprache oder Ausprobieren auf Anhieb korrekt durchführt, sofern er sich genau an die Vorschrift hält.

Aus Beispielen (s. nächste Folie) gewinnt man die Einsicht, dass eine spezielle Sprache sinnvoll ist, um Algorithmen zu beschreiben. Geeignete Sprachelemente (A1) bis (A9) hierfür werden wir im Folgenden angeben.

Einige Beispiele aus dem Alltag:

- Kochrezepte.
- Bastelanleitungen.
- Ermittlung der Abiturnote aus den Leistungen der Oberstufe.
- Bewertungsalgorithmus für Klassenarbeiten (Deutsch, Mathe, ...).
- Ablauf der Gesetzgebungsverfahren.
- Ermittlung kürzester Verbindungen zwischen zwei Orten.
- Feststellen von Rechtschreibfehlern in einem Text.
- In der Industrie eingesetzte Produktionsvorgänge.
- Abläufe in Verwaltungen, z.B. Genehmigung eines Bauantrags.
- Berechnung der Reisekostenerstattung durch eine Verwaltung.
- Erstellung einer Häufigkeitsstatistik aller Wörter, die Goethe in seinem Gesamtwerk verwendet hat (das Gleiche für Noten und Musiker, Farben und Maler, ...).

1.2 Algorithmische Sprachelemente

Festlegungen: (A1) bis (A9)
Diese und weitere Untersuchungen führten zu folgenden Vorgaben und Sprachelementen für die Beschreibung von Algorithmen und den von ihnen manipulierten Daten:

(A1) Ein Algorithmus ist eine Folge von Anweisungen. Eine Anweisung besteht aus elementaren Anweisungen, die nach den folgenden Regeln zu Anweisungen zusammengefügt werden können. Der Algorithmus erhält einen Bezeichner (oder einen Namen).

Ein Bezeichner ist eine Folge von Buchstaben und Ziffern, die mit einem Buchstaben beginnt, z.B. Y, i, B34Z, qqq.
Meist lässt man auch noch den Unterstrich _ zu:
Heute_ist_Samstag, X_1, X_2 .
In der Praxis begrenzt man meist die Länge eines Bezeichners, z.B. auf 32 Zeichen.

(A2) Ein Algorithmus arbeitet auf Daten. Daten werden in "Behältern", genannt Variablen, abgelegt. Variablen sind zu Beginn des Algorithmus aufzulisten einschließlich der Angabe, welche Daten in die Variable gelegt werden dürfen und welche nicht (dies nennt man "Deklaration" oder "Vereinbarung" der Variablen). Diese durch "," oder ";" getrennte Auflistung beginnt mit dem Wort declare.

Variablen werden ebenfalls durch Bezeichner dargestellt. In einem Algorithmus soll man die Bezeichner so wählen, dass man hieraus die Bedeutung der Variablen entnehmen kann, also z.B. Bezeichner wie
Eingabewert, Anzahl, Summe_der_Messwerte, Mittelwert_von_1_bis_100.

Hinweis: In der Mathematik sind Variablen "Platzhalter" oder "Unbestimmte", also etwas anderes als Behälter, in denen man Werte ablegt. Den mathematischen Variablen entsprechen die noch einzuführenden "formalen Parameter".

(A3) Elementare Anweisungen sind von der Form:

skip	*Bedeutung:* Tue nichts.
X := α	"Wertzuweisung". α ist ein Ausdruck. *Bedeutung:* Rechne den Ausdruck α aus und lege den erhaltenen Wert in der Variablen X ab.
read X	Leseanweisung. *Bedeutung:* Lies den nächsten Wert ein und lege ihn in der Variablen X ab.
write α	Schreibanweisung. *Bedeutung:* Drucke den Wert, den der Ausdruck α besitzt, aus.
halt	*Bedeutung:* Beende den Algorithmus.
$F(X_1, ..., X_n)$	*Bedeutung:* Führe den Algorithmus F mit den Werten der Variablen $X_1, ..., X_n$ aus.

(A4) Ausdrücke sind entweder übliche arithmetische Ausdrücke (aufgebaut aus Zahlen, Variablen, Klammern und Operatoren wie +, -, *, /, div, mod) oder logische Ausdrücke (aufgebaut aus den Wahrheitswerten true und false, Variablen, Klammern, Vergleichen und Operatoren wie and, or, not usw.) oder Zeichenausdrücke (aufgebaut aus den Zeichen eines Alphabets, Variablen und Operatoren wie append, empty, remove usw.). Logische Ausdrücke nennt man auch Boolesche Ausdrücke. Wir setzen voraus, dass jede(r) weiß, wie Ausdrücke aufgebaut sind und wie man Ausdrücke auswertet.

(A5) Jede elementare Anweisung ist auch eine Anweisung.

(A6) Hintereinanderausführung oder Sequenz: Wenn C und D Anweisungen sind, dann ist auch C;D eine Anweisung.
Bedeutung: Führe erst C und danach D aus.

(A7) Alternative oder Fallunterscheidung:

Wenn C und D Anweisungen und β ein Boolescher Ausdruck sind, dann ist auch

if β then C else D fi

eine Anweisung.

Bedeutung: Wenn zu dem Zeitpunkt, zu dem man auf diese Anweisung stößt, der Boolesche Ausdruck β den Wert true besitzt (also wahr ist), so führe die Anweisung C aus, anderenfalls die Anweisung D.

Spezialfall: Falls D die Anweisung skip ist, so schreibt man kurz if β then C fi (*einseitige Alternative*).

Hinweis: fi ist wie die "Klammer Zu" zum Symbol if. Die Klammerpaare "(" und ")" oder "[" und "]" entstehen auseinander ebenfalls durch Spiegelung. Auch das in (A8) auftretende "od" bildet mit "do" eine solche Klammerung.

(A8a) while-Schleife:

Wenn C eine Anweisung und β ein Boolescher Ausdruck sind, dann ist while β do C od eine Anweisung.
Bedeutung: Solange der Boolesche Ausdruck β den Wert true ergibt, wiederhole die Anweisung C. Hierbei wird der Ausdruck β stets *vor* der Ausführung von C ausgewertet. Wenn also β zu dem Zeitpunkt, zu dem man auf die while-Schleife stößt, false ist, wird C überhaupt nicht ausgeführt.

(A8b) repeat-Schleife:

Wenn C eine Anweisung und β ein Boolescher Ausdruck sind, dann ist repeat C until β eine Anweisung.
Bedeutung: Solange der Boolesche Ausdruck β den Wert false ergibt, wiederhole die Anweisung C. Hierbei wird der Ausdruck β erst *nach* der Ausführung von C ausgewertet. C wird also stets mindestens einmal ausgeführt.

(A8c) for-Schleife oder Zählschleife:

Wenn C eine Anweisung, i eine Variable, in die ganze Zahlen gelegt werden dürfen, und a und e zwei ganze Zahlen sind, dann ist auch

for i := a to e do C od

eine Anweisung (sofern i und e durch C nicht verändert werden).

Bedeutung: Setze i auf den Wert a. Falls i nicht größer als e ist, führe C aus. Erhöhe nun i um 1 und wiederhole diesen Vorgang, bis i größer als e ist; anschließend führe die Anweisung, die auf die for-Schleife folgt, aus.

Präzisere Festlegung: Die for-Schleife besitzt die gleiche Bedeutung wie folgende Anweisung

i := a; while i ≤ e do C; i := i+1 od

Man verbietet, dass die Variable i oder der Wert von e durch C verändert werden können. Insbesondere darf es in C keine Wertzuweisung der Form i := ... geben. Diese Bedingung ist in der Praxis manchmal kaum nachzuprüfen.

(A9) Ergänzende Vorschriften

Aus Gründen der Übersichtlichkeit und Lesbarkeit macht man meist weitere Vorschriften, z.B. fordern wir:

Die Deklarationen müssen *stets am Anfang* des Algorithmus angegeben werden.

Die auf die Deklarationen folgende Anweisung wird in begin ... end eingeklammert. (halt kann dann entfallen.)

Jeder vorkommende Bezeichner muss vorher in einer Deklaration vereinbart worden sein.

Kommentare trennen wir bis zum Zeilenende durch die Zeichen -- vom Algorithmus ab.

Unsere Algorithmen beginnen stets mit dem Wort program, danach folgt der Name des Algorithmus, danach das Wort is, dann die Deklarationen und schließlich die in begin und end eingeschlossenen Anweisungen.

Einen nach den Vorschriften (A1) bis (A9) aufgeschriebenen Algorithmus bezeichnen wir als **Programm**.

Unsere Programme sind also folgendermaßen aufgebaut:

program <Name des Algorithmus> is
declare <Deklarationen>;
begin <Anweisung> end

Diese Grundgerüst findet sich in den meisten (imperativen) Programmiersprachen. Jede Sprache hat aber viele weitere Sprachelementen und Vorschriften, z.B. im Deklarationsteil und bei der Parametrisierung.

Hinweise:

Zunächst ist die Deklaration von Variablen zu präzisieren, insbesondere wenn die Variablen nicht Zahlen, sondern komplexere Gebilde wie Vektoren, Matrizen oder Graphen als Werte besitzen.

Sodann ist die Wiederverwendung von bereits ausformulierten Programmen festzulegen. Hierfür werden Prozeduren, Funktionen, Moduln und Objekte eingeführt.

Dabei sind die zulässigen Parameter und ihre Ersetzung durch konkrete Werte oder Typen zu klären.

Schließlich muss man das Einfügen bereits bestehender Programm(teil)e und das Zusammenspiel von Programmen regeln (Bibliotheken, Dialoge und andere Interaktionen).

Beispiel: Stelle fest, ob eine natürliche Zahl a eine Quadratzahl ist. Falls ja, drucke 1 aus, sonst drucke 0 aus.

Verfahren: Prüfe für alle Zahlen von 0 bis a, ob deren Quadrat gleich a ist. Falls es eine solche Zahl gibt, dann ist a eine Quadratzahl, anderenfalls nicht. Übertragen in unsere Sprache:

```
program quadratzahl1 is
declare natürliche Zahlen x, i, ergebnis;
begin  read x;                    -- Es wird a eingelesen und in x abgelegt.
       ergebnis := 0;
       for i:=0 to x do           -- prüfe für i von 0 bis a, ob i² = a ist
            if i*i = x then ergebnis := 1 fi  od;
       write ergebnis
end
```

Überprüfe, ob die Regeln (A1) bis (A9) eingehalten wurden:

```
program quadratzahl1 is
declare  natürliche Zahlen  x, i, ergebnis;
begin  read x;                  -- Es wird a eingelesen und in x abgelegt.
       ergebnis := 0;
       for i:=0 to x do         -- prüfe für i von 0 bis a, ob i^2 = a ist
           if i*i = x then ergebnis := 1 fi od;
       write ergebnis
end
```

Also: korrekt gebildetes Programm

Fortsetzung Beispiel:

Rechnet man das Programm für eine Zahl, z.B. für a = 33 durch, so quadriert man alle Zahlen von 0 bis 33, wobei man jedes Mal feststellt, dass i^2 ungleich 33 ist. Man hätte bereits bei i=6 aufhören können, da ab dann $i^2 > 33$ ist. Dies führt zu folgendem "effizienter" arbeitenden Programm:

```
program quadratzahl2 is
declare natürliche Zahlen x, i, ergebnis;
begin  read x;                    -- Es wird a eingelesen und in x abgelegt.
       ergebnis := 0;  i := 0;
       while i*i ≤ x do           -- prüfe nur für i von 0 bis wurzel(a)
            if i*i = x then ergebnis := 1 fi; i := i+1 od;
       write ergebnis
end
```

Fortsetzung Beispiel:
Das Programm quadratzahl2 führt nicht mehr a, sondern nur noch 2·wurzel(a) Multiplikationen durch. Frage: Kann man die lästigen Multiplikationen sparen?

Ja, das geht. Beachte, dass die Differenz zwischen zwei Quadratzahlen immer eine ungerade Zahl ist und dass man die n-te Quadratzahl erhält, indem man die ungeraden Zahlen zwischen 1 und 2n-1 aufsummiert.

1 = 1 4 = 1+3 9 = 1+3+5 16 = 1+3+5+7
25 = 1+3+5+7+9 usw.

Wir notieren daher die nächste ungerade Zahl in der Variablen u (erster Wert ist 1) und das aktuelle Quadrat in der Variablen q (deren erster Wert ist 0). Die nächste Quadratzahl ermitteln wir dann durch die Anweisung q := q+u; u := u+2 .

Fortsetzung Beispiel:
Dies führt zu dem Programm

```
program quadratzahl3 is
declare natürliche Zahlen x, u, q, ergebnis;
begin  read x;                    -- Es wird a eingelesen und in x abgelegt.
       q := 0;  u := 1; ergebnis := 0;
       while q ≤ x do             -- prüfe für i von 0 bis a, ob i² = a ist
            if q = x then ergebnis := 1 fi; q := q+u; u := u+2  od;
       write ergebnis
end
```

Hier werden keine Multiplikationen mehr benötigt, sondern nur noch 2·wurzel(a) Additionen. Dieses Programm wird daher wesentlich schneller durchzurechnen sein als quadratzahl1.

1.3 Ablaufprotokoll

Frage: Wie prüft man nach, was ein Programm macht?
Einfache Antwort: Man vollzieht es schrittweise nach, wobei man die Veränderungen aller Variablen notiert. Ein solches Schema nennt man ein Ablaufprotokoll. Das Schema hierfür lautet (man schreibe die Eingabe und Ausgabe gesondert auf):

Schritt	Aktion	<Var. 1>	<Var. 2>	<Var. 3>	<Var. 4>	...

Definition: Es sei ein Programm mit seinen aktuellen Eingabedaten gegeben. Bilde eine zweidimensionale Tabelle, die für jede im Programm vorkommende Variable eine Spalte, zwei Spalten für die fortlaufende (Zeilen-) Nummerierung und für die aktuelle Aktion (dies ist in der Regel eine elementare Anweisung oder die Auswertung eines Ausdrucks) sowie eventuelle weitere Spalten für Hilfsinformationen besitzt.

Trage in die erste Zeile die Anfangssituation ein, also die erste Aktion des Programms und die Werte der Variablen nach Durchführung dieser Aktion. Trage in die jeweils nächste Zeile mit der Nummer k die im k-ten Schritt durchgeführte Aktion und die Werte der Variablen nach Durchführung dieser Aktion ein, solange bis halt oder end erreicht wird. Ein- und Ausgabe notiere man gesondert. Die so entstandene Tabelle heißt Ablaufprotokoll des Programms für die gegebenen Eingabedaten.

```
program quadratzahl3 is
declare natürliche Zahlen x, u, q, erg;
begin  read x;
       q := 0;  u := 1; erg := 0;
       while q ≤ x do
            if q = x then erg := 1 fi; q := q+u; u := u+2  od;
       write erg
end
```

Eingabe sei 14, ⊥ bedeutet "undefiniert"

Schritt	Aktion	x	u	q	erg	Ausdrücke
1	read x	14	⊥	⊥	⊥	
2	q := 0	14	⊥	0	⊥	
3	u := 1	14	1	0	⊥	
4	erg := 0	14	1	0	0	
5	q ≤ x	14	1	0	0	true
6	q = x	14	1	0	0	false
7	q := q+u	14	1	1	0	
8	u := u+2	14	3	1	0	
9	q ≤ x	14	3	1	0	true

```
program quadratzahl3 is
declare natürliche Zahlen x, u, q, erg;
begin  read x;
       q := 0;  u := 1; erg := 0;
       while q ≤ x do
            if q = x then erg := 1 fi; q := q+u; u := u+2  od;
       write erg
end
```

Eingabe sei 14, ⊥ bedeutet "undefiniert"

Schritt	Aktion	x	u	q	erg	Ausdrücke
10	q = x	14	3	1	0	false
11	q := q+u	14	3	4	0	
12	u := u+2	14	5	4	0	
13	q ≤ x	14	5	4	0	true
14	q = x	14	5	4	0	false
15	q := q+u	14	5	9	0	
16	u := u+2	14	7	9	0	
17	q ≤ x	14	7	9	0	true
18	q = x	14	7	9	0	false

```
program quadratzahl3 is
declare natürliche Zahlen x, u, q, erg;
begin  read x;
       q := 0;  u := 1; erg := 0;
       while q ≤ x do
            if q = x then erg := 1 fi; q := q+u; u := u+2  od;
       write erg
end
```

Eingabe sei 14, ⊥ bedeutet "undefiniert"

Schritt	Aktion	x	u	q	erg	Ausdrücke
19	q := q+u	14	7	16	0	
20	u := u+2	14	9	16	0	
21	q ≤ x	14	9	16	0	false
22	write erg	14	9	16	0	Ausgabe 0
23	"end"	14	9	16	0	

Ausgabe ist 0, d.h., die Eingabe ist keine Quadratzahl.

1.4 Elementare Datentypen

Eine Variable ist in unseren Programmen "getypt", d.h., sie darf nur ganz bestimmte Werte als Inhalt besitzen.
Hierbei gibt es Variablen, die nur ganze Zahlen aufnehmen können; wir sagen, solche Variablen sind "vom Typ integer". Andere dürfen nur Zeichen enthalten; diese sind "vom Typ character". Mit dem Typ wird somit der zulässige Inhalt und zugleich die hierauf zulässigen Operationen festgelegt.

Definition:

Eine Menge (oder mehrere Mengen, "Wertebereich" genannt) zusammen mit den hierauf definierten Operationen nennt man einen (konkreten) Datentyp.

Einen Datentyp, den man nicht auf andere Datentypen zurückführt, nennen wir einen elementaren Datentyp.

Zu den elementaren Datentypen zählen wir die selbst-definierten Aufzählungstypen und die Standarddatentypen.

Aufzählungstypen

Ein neuer Datentyp wird im Deklarationsteil mit Hilfe des Schlüsselwortes type eingeführt in der Form:

type <Name des Datentyps> is (<Liste der Elemente>)

Die Reihenfolge in der Liste bildet zugleich eine Anordnung der Elemente. Der "Name" ist ein selbstgewählter Bezeichner.

Beispiele:

```
type Wochentage is (Mo, Di, Mi, Do, Fr, Sa, So);
type Freie_Tage is (Sa, So);
type Farbe is (weiß, gelb, grün, rot, blau, schwarz);
type Erste_zehn_Primzahlen is (2,3,5,7,11,13,17,19,23,29);
type Nur_die_Null is (0);
```

Ein Datentyp besteht aus dem Wertebereich und den zugehörigen Operationen. Für Aufzählungstypen sind dies:

Wertebereich (= zugrunde liegende Menge): die endliche Menge $M = \{m_1, m_2, ..., m_n\}$, die man selbst definiert hat. Die Menge M wird wie oben angegeben eingeführt, also:

type <Name des Datentyps> is (m_1, m_2, ..., m_n);

Die Menge M ist für diesen Datentyp dann automatisch in dieser Reihenfolge angeordnet: $m_1 < m_2 < ... < m_n$.

Hinweis: Es kommt hierbei prinzipiell nicht zu Namenskonflikten! Das heißt: Wenn ein Wert in mehreren Datentypen verwendet wird, so muss man selbst darauf achten, dass bei jeder Verwendung dieses Elementes eindeutig klar ist, zu welchem Datentyp er gehört. (Dies regelt jede Programmiersprache auf eigene Weise.)

Folgende Operationen seien auf allen Aufzählungstypen definiert:

Vorgänger ("predecessor") und Nachfolger ("successor")
Pred, Succ: M $\rightarrow$ M mit

Pred(X) = das Zeichen vor X in der Auflistungs-Reihenfolge,
Succ(X) = das Zeichen nach X in der Auflistungs-Reihenfolge.
Pred(m_i) = m_{i-1} für i>1 und Pred(m_1) = undefiniert,
Succ(m_i) = m_{i+1} für i<n und Succ(m_n) = undefiniert.

Position in der Anordnung der Zeichen: Pos: M $\rightarrow$ $\mathbb{N}_0$

Pos(X) = n bedeutet: X ist das n-te Zeichen in der Auflistungsreihenfolge (*beginnend mit 0*, d.h., das erste Zeichen hat die Position 0, das zweite die Position 1 usw.).

n-tes Element in der Anordnung: Val: $\mathbb{N}_0 \rightarrow$ M

Val(n) = X bedeutet: X ist das n-te Zeichen in der Auflistungsreihenfolge *(beginnend mit 0).*

Neben diesen vier einstelligen gibt es auch stets folgende zweistellige Operationen (Vergleiche und Min, Max):

Alle sechs Vergleichoperationen bezüglich der Anordnung der Menge "=", "≠", "<", "≤", ">" und "≥" sind zugelassen. Das Ergebnis ist ein Wahrheitswert vom Typ Boolean (siehe später).

Für alle Aufzählungstypen sind die zweistelligen Operationen Minimum und Maximum definiert: Min, Max: $M \times M \rightarrow M$ mit

$Min(X,Y) = X \Leftrightarrow Pos(X) \leq Pos(Y)$,
anderenfalls ist $Min(X,Y) = Y$.

$Max(X,Y) = X \Leftrightarrow Pos(X) > Pos(Y)$,
anderenfalls ist $Max(X,Y) = Y$.

Beachte: Im Falle $X \neq Y$ gilt stets $Min(X,Y) \neq Max(X,Y)$.

Standard-Datentypen:

Boolean	$\mathbb{B}$ = {false, true}
character	A = Latin1-Alphabet aus 256 Zeichen
integer	$\mathbb{Z}$ = Menge der ganzen Zahlen
real	$\mathbb{R}$ = Menge der reellen Zahlen

Unterbereiche von Integer (vgl. Abschnitt 2.1):

natural	$\mathbb{N}_0$ (Natürliche Zahlen mit der Null)
positive	$\mathbb{N}$ (Natürliche Zahlen ohne die Null)

Wichtig: Mit jedem Datentyp sind zugleich die erlaubten Operationen definiert! Diese listen wir für die obigen vier Standarddatentypen nun auf.

Der Datentyp Boolean

Wertebereich: $\mathbb{B}$ = {false, true} (= {falsch, wahr})
Boolean wird oft zugleich als Aufzählungstyp aufgefasst:

type Boolean is (false, true);

Nullstellige Operationen (=besondere Konstanten): false und true.

Einstellige Operationen (NICHT):

Negation $\neg$: $\mathbb{B} \to \mathbb{B}$ (statt $\neg$ schreibt man not).

Zweistellige Operationen (UND, ODER, EXKLUSIVES ODER, ÄQUIVALENZ):

Konjunktion $\wedge$: $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$ (statt $\wedge$ schreibt man and)
Disjunktion $\vee$: $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$ (statt $\vee$ schreibt man or)
Ungleichheit (oder auch "Exklusives Oder" genannt)
$\neq$: $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$ (statt $\neq$ schreibt man xor)
Gleichheit (oder auch "Äquivalenz" genannt)
= : $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$ (statt = schreibt man equiv)

Der Datentyp character

Der Typ character wird meist als Aufzählungstyp mit dem Alphabet "Latin-1" als Wertebereich aufgefasst:
type character is (*NUL*, ..., *US*, ' ', '!', "", '#', '$', ..., 'ÿ');
Die einzelnen Zeichen werden in den meisten Programmier-sprachen in Apostroph eingeschlossen, sofern sie graphisch darstellbar sind (das Apostroph selbst wird hierbei durch zwei Apostrophe dargestellt); sonst schreibt man NUL, SOH, STX, ...
Es stehen somit für character alle allgemeinen Operationen der Aufzählungstypen zur Verfügung. In modernen Sprachen wird statt Latin-1 der aus 16 Bit bestehende Unicode genommen.

Zur Kenntnisnahme geben wir den vollständigen Latin-1 Code, definiert in der ISO-Norm 8859-1, auf den nächsten Folien an. In der Tabelle wird die hexadezimale und die (mit dem Nummerzeichen '#' versehene) dezimale Nummerierung vorangestellt und dann das zugehörige Zeichen genannt.

ISO 8859, Latin-1 Alphabet (festgelegt sind nur die schwarzen Werte; die kursiven blauen bilden eine übliche Ergänzung)

00 #0 NULL	*01 #1 START OF HEADING*	*02 #2 START OF TEXT*	*03 #3 END OF TEXT*	*04 #4 END OF TRANSM.*	*05 #5 ENQUIRY*	*06 #6 ACKN.*	*07 #7 BELL*	*08 #8 BACK-SPACE*	09 #9 HORIZ. TAB.	0A #10 LINE FEED	*0B #11 VERTICAL TAB.*	*0C #12 FORM FEED*	0D #13 CARRIAG RETURN	*0E #14 SHIFT OUT*	*0F #15 SHIFT IN*
10 #16 DATA LINK ESC.	*11 #17 DEVICE CONTR.1*	*12 #18 DEVICE CONTR.2*	*13 #19 DEVICE CONTR.3*	*14 #20 DEVICE CONTR.4*	*15 #21 NEGATIV ACKN.*	*16 #22 SYNCHR. IDLE*	*17 #23 END OF T.BLOCK*	*18 #24 CANCEL*	*19 #25 END OF MEDIUM*	*1A #26 SUBSTI-TUTE*	*1B #27 ESCAPE*	*1C #28 FILE SE-PARATOR*	*1D #29 GROUP SEPAR.*	*1E #30 RECORD SEPAR.*	*1F #31 UNIT SEPAR.*
20 #32 SPACE	21 #33 **!**	22 #34 **“**	23 #35 **#**	24 #36 **$**	25 #37 **%**	26 #38 **&**	27 #39 **‘**	28 #40 **(**	29 #41 **)**	2A # 42 *****	2B #43 **+**	2C #44 **,**	2D #45 **-**	2E #46 **.**	2F #47 **/**
30 #48 **0**	31 #49 **1**	32 #50 **2**	33 #51 **3**	34 #52 **4**	35 #53 **5**	36 #54 **6**	37 #55 **7**	38 #56 **8**	39 #57 **9**	3A #58 **:**	3B #59 **;**	3C #60 **<**	3D #61 **=**	3E #62 **>**	3F #63 **?**
40 #64 **@**	41 #65 **A**	42 #66 **B**	43 #67 **C**	44 #68 **D**	45 #69 **E**	46 #70 **F**	47 #71 **G**	48 #72 **H**	49 #73 **I**	4A #74 **J**	4B #75 **K**	4C #76 **L**	4D #77 **M**	4E #78 **N**	4F #79 **O**
50 #80 **P**	51 #81 **Q**	52 #82 **R**	53 #83 **S**	54 #84 **T**	55 #85 **U**	56 #86 **V**	57 #87 **W**	58 #88 **X**	59 #89 **Y**	5A #90 **Z**	5B #91 **[**	5C #92 ****	5D #93 **]**	5E #94 **^**	5F #95 **_**
60 #96 **`**	61 #97 **a**	62 #98 **b**	63 #99 **c**	64 #100 **d**	65 #101 **e**	66 #102 **f**	67 #103 **g**	68 #104 **h**	69 #105 **i**	6A #106 **j**	6B #107 **k**	6C #108 **l**	6D #109 **m**	6E #110 **n**	6F #111 **o**
70 #112 **p**	71 #113 **q**	72 #114 **r**	73 #115 **s**	74 #116 **t**	75 #117 **u**	76 #118 **v**	77 #119 **w**	78 #120 **x**	79 #121 **y**	7A #122 **z**	7B #123 **{**	7C #124 **|**	7D #125 **}**	7E #126 **~**	*7F #127 DELETE*
80 #128 €	*81 #129*	*82 #130 ‚*	*83 #131 ƒ*	*84 #132 „*	*85 #133 …*	*86 #134 †*	*87 #135 ‡*	*88 #136 ˆ*	*89 #137 ‰*	*8A #138 Š*	*8B #139 ‹*	*8C #140 Œ*	*8D #141*	*8E #142 Ž*	*8F #143*
90 #144	*91 #145 ‘*	*92 #146 ’*	*93 #147 “*	*94 #148 ”*	*95 #149 •*	*96 #150 –*	*97 #151 —*	*98 #152 ˜*	*99 #153 ™*	*9A #154 š*	*9B #155 ›*	*9C #156 œ*	*9D #157*	*9E #158 ž*	*9F #159 Ÿ*
A0 #160 NO-BREAK SPACE	A1 #161 **¡**	A2 #162 **¢** CENT	A3 #163 **£** POUND	A4 #164 **¤**	A5 #165 **¥** YEN	A6 #166 **¦**	A7 #167 **§**	A8 #168 **¨**	A9 #169 **©**	AA #170 **ª**	AB #171 **«**	AC #172 **¬**	AD #173 - SOFT HYPHEN	AE #174 **®** REGIS-TERED	AF #175 **¯** MACRON
B0 #176 **°**	B1 #177 **±**	B2 #178 **²**	B3 #179 **³**	B4 #180 **´** ACCENT	B5 #181 **µ** MICRO	B6 #182 **¶**	B7 #183 **·** MID.DOT	B8 #184 **¸** CEDILLA	B9 #185 **¹**	BA #186 **º**	BB #187 **»**	BC #188 **¼**	BD #189 **½**	BE #190 **¾**	BF #191 **¿**
C0 #192 **À**	C1 #193 **Á**	C2 #194 **Â**	C3 #195 **Ã**	C4 #196 **Ä**	C5 #197 **Å**	C6 #198 **Æ**	C7 #199 **Ç**	C8 #200 **È**	C9 #201 **É**	CA #202 **Ê**	CB #203 **Ë**	CC #204 **Ì**	CD #205 **Í**	CE #206 **Î**	CF #207 **Ï**
D0 #208 **Ð**	D1 #209 **Ñ**	D2 #210 **Ò**	D3 #211 **Ó**	D4 #212 **Ô**	D5 #213 **Õ**	D6 #214 **Ö**	D7 #215 **×**	D8 #216 **Ø**	D9 #217 **Ù**	DA #218 **Ú**	DB #219 **Û**	DC #220 **Ü**	DD #221 **Ý**	DE #222 **Þ**	DF #223 **ß**
E0 #224 **à**	E1 #225 **á**	E2 #226 **â**	E3 #227 **ã**	E4 #228 **ä**	E5 #229 **å**	E6 #230 **æ**	E7 #231 **ç**	E8 #232 **è**	E9 #233 **é**	EA #234 **ê**	EB #235 **ë**	EC #236 **ì**	ED #237 **í**	EE #238 **î**	EF #239 **ï**
F0 #240 **ð**	F1 #241 **ñ**	F2 #242 **ò**	F3 #243 **ó**	F4 #244 **ô**	F5 #245 **õ**	F6 #246 **ö**	F7 #247 **÷**	F8 #248 **ø**	F9 #249 **ù**	FA #250 **ú**	FB #251 **û**	FC #252 **ü**	FD #253 **ý**	FE #254 **þ**	FF #255 **ÿ**

Latin-1 enthält **ASCII** als die ersten 128 Zeichen. Die Zeichen mit den Nummern 128 bis 159 wurden in Latin-1 frei gelassen; sie werden von verschiedenen Softwareherstellern unterschiedlich verwendet. Oben sind die Zeichen von Microsoft Windows eingetragen (die Nummern 128, 129, 141-144, 157, 158 werden dabei nicht festgelegt; 128 wird meist für das Eurozeichen, 142 und 158 für das modifizierte Z benutzt). In vielen Sprachen kann man auf die nicht graphisch darstellbaren Zeichen mit symbolischen Bezeichnungen wie NUL, SOH usw. zugreifen.

Der Datentyp integer

Theoretisch lautet die zugrunde liegende Wertemenge:
Z = { ..., -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, ... } mit üblicher Ordnung.

In der Praxis kann man nur endliche Mengen darstellen, so dass der Datentyp integer dann ein Aufzählungstyp ist, der von der kleinsten (min) bis zur größten (max) darstellbaren Zahl reicht:

type integer is (min, min+1, min+2, ..., max);

Nullstellige Operationen: Alle Zahlen.

Einstellige Operationen:

+	einstelliges Plus (wie Identität, verändert also nichts)
-	einstelliges Minus, Bildung der negativen Zahl, Negation
abs	Absolutbetrag einer Zahl

Zweistellige Operationen:

+	Addition zweier Zahlen
-	Subtraktion (=Differenz) zweier Zahlen
*	Multiplikation zweier Zahlen
mod	Modulo-Funktion (beachte: $0 \leq x \bmod y < \text{abs } y$)
div	ganzzahlige Division (wie bei reellen Zahlen, aber dann nächste ganze Zahl in Richtung zur Null nehmen)
rem	Rest bei der Division mit /, d.h.: $x \text{ rem } y = x - (x \text{ div } y) * y$.
**	Exponentiation (x**y = x^y, nur für $y \geq 0$ definiert)

Hinzu kommen die sechs Vergleichsoperationen "=", "≠", "<", "≤", ">" und "≥" der Funktionalität $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{B}$.
Da integer als Aufzählungstyp aufgefasst wird, sind auch Min, Max, Pred, Succ, Pos und Val für Integer definiert (wobei Pos und Val aber wenig Sinn machen).

Der Datentyp real

Theoretisch lautet die zugrunde liegende Wertemenge:
$\mathbb{R}$ = {x | x ist eine reelle Zahl} mit üblicher Ordnung.

In der Praxis kann man nur endliche Mengen darstellen, so dass der Datentyp real dann ein Aufzählungstyp aller (rationaler) Zahlen ist, die sich mit der jeweiligen Speicherstruktur eines Computers darstellen lassen.

Nullstellige Operationen: Alle darstellbaren reellen Zahlen.

Hinweis: In Programmiersprachen werden die reellen Zahlen durch endlich viele Zeichen angenähert dargestellt und zwar meist entweder als Gleitkommazahl oder als Festkommazahl.

Weitere Operationen auf den reellen Zahlen:

Für alle Darstellungen gibt es die einstelligen Operationen +, - und Absolutbetrag und die zweistelligen Operationen Addition +, Subtraktion -, Multiplikation *, Division / und Exponentiation ** (rechts von ** darf oft nur eine ganze Zahl stehen).

Weiterhin gibt es die Vergleichsoperationen "=", "≠", "<", "≤", ">" und "≥" der Funktionalität $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{B}$.
(In der Praxis sollten wegen der unvermeidlichen Rundungsfehler die Vergleichsoperatoren = und ≠ bei reellen Zahlen *nicht* verwendet werden.)

Darüber hinaus gibt es eine Vielzahl weiterer Funktionen (Sinus, Cosinus, Exponentialfunktion, Logarithmus, ...).

Oft benötigt man das Runden zur nächsten ganzen Zahl: round: $\mathbb{R} \rightarrow \mathbb{Z}$ ("round" = runden) und das Abrunden zur absolut darunter liegenden kleineren Zahl trunc: $\mathbb{R} \rightarrow \mathbb{Z}$.

Prioritäten der gängigen Operationen in Ausdrücken:

Im "Alltag" lauten die Prioritäten meist:

1. Priorität: Absolutbetrag abs
2. Priorität: Exponentiation **
3. Priorität: ·, div, mod, /, rem
4. Priorität: +, - (als einstellige Operationen)
5. Priorität: +, - (als zweistellige Operationen)
6. Priorität: =, ≠, <, ≤, >, ≥
7. Priorität: not
8. Priorität: and
9. Priorität: or
10. Priorität: equiv, xor

(Dies kann in manchen Sprachen auch ganz anders sein!)

1.5 Beispiel Zahldarstellungen

Reelle Zahlen werden meist durch ein Stellenwertsystem mit Vorzeichen beschrieben.

Definition:
Ein Stellenwertsystem ist ein Tripel $S = (b, Z, \beta)$ mit
(1) $b \geq 2$ ist eine natürliche Zahl (die "Basis"),
(2) Z ist eine b-elementige Menge (die Menge der Ziffern),
(3) $\beta: Z \rightarrow \{0, 1, ..., b-1\}$ ordnet jedem Ziffernsymbol umkehrbar eindeutig eine Zahl zwischen 0 und b-1 zu.

Eine natürliche Zahl wird beschrieben durch eine endliche Folge von Ziffern aus Z ohne führende Nullen (außer der Null selbst).
Der Wert $\phi(z)$ einer n-stelligen Zahldarstellung $z = z_{n-1}z_{n-2} \ldots z_1z_0$ mit $n>0$ und $z_i \in Z$ für i=0,1,...,n-1 ist definiert als

$$\phi(z) = \sum_{i=0}^{n-1} \beta(z_i) \cdot b^i$$

Eine relle Zahl wird beschrieben durch eine endliche nicht-leere Folge von Ziffern aus Z ohne führende Nullen (außer der Null selbst). Anschließend folgt ein Punkt und eine unendliche Folge von Ziffern aus Z. Die Darstellung hat also die Form

$$z = \pm z_{n-1}z_{n-2} \ldots z_1z_0 \,.\, z_{-1}z_{-2}z_{-3}z_{-4} \ldots \quad \text{mit } n>0.$$

Gilt ab einem k>0, dass z_{-k}, z_{-k-1}, z_{-k-2} usw. alle 0 sind, so lässt man ab z_{-k} alle Ziffern weg und erhält eine endliche Darstellung (solche Zahlen gehören zu den rationalen Zahlen; allerdings kann man bei vorgegebener Basis b nicht alle rationalen Zahlen durch eine endliche Folge darstellen). Ist k=1, so lässt man auch den Punkt weg (dann liegt eine ganze Zahl vor).

Die relle Zahl $\phi(z)$, die zu $z = \pm z_{n-1}z_{n-2} \ldots z_1z_0 \,.\, z_{-1}z_{-2}z_{-3}z_{-4} \ldots$ mit $n>0$ und $z_i \in Z$ für i=0,1,...,n-1 gehört, ist definiert durch

$$\phi(z) = \pm \sum_{i=-\infty}^{n-1} \beta(z_i) \cdot b^i$$

In der Regel benutzt man keine gesonderten Zeichen für Z, sondern nimmt die Ziffern von 0 bis b-1 (ab 10 fährt man mit den Buchstaben A, B, ... fort, siehe Hexadezimaldarstellung b=16).

Als Basis verwendet man vor allem 10, 2, 8 und 16. Kann es zu Verwechselungen kommen, welche Basis gemeint ist, so schließt man die Zahldarstellung in runde Klammern ein und setzt die Basis b tiefgestellt dahinter:

$(2101)_{\mathbf{3}}$ = "2101 zur Basis 3" = "2·27 + 1·9 + 1" = $(64)_{\mathbf{10}}$

$(2101)_{\mathbf{4}}$ = "2101 zur Basis 4" = "2·64 + 1·16 + 1" = $(145)_{\mathbf{10}}$

$(2101)_{\mathbf{5}}$ = "2101 zur Basis 5" = "2·125 + 1·25 + 1" = $(276)_{\mathbf{10}}$

$(2101.22)_{\mathbf{3}}$ = "2101.22 zur Basis 3"
= "2·27 + 1·9 + 1 + 2/3 + 2/9" = $(64.88888...)_{\mathbf{10}}$

$(-2101.22)_{\mathbf{4}}$ = minus "2·64 + 1·16 + 1 + 2/4 + 2/16" = $(-145.625)_{\mathbf{10}}$

Aufgabe: Gegeben seien eine reelle Zahl x und eine natürliche Zahl b ≥ 2. Gib die Zahl x als Ziffernfolge zur Basis b aus.

Fall 1: Die Zahl x ist eine natürliche Zahl.
Es sei $z_{n-1}z_{n-2} \ldots z_1z_0$ die Zahldarstellung von x zur Basis b.
Dann muss gelten:
z_0 = x mod b (= Rest bei der Division von x durch b)
und die Zahl x div b (= ganzzahlige Division von x durch b)
besitzt die Zahldarstellung $z_{n-1}z_{n-2} \ldots z_1$.
Dies liefert bereits den Algorithmus für x > 0:

```
while x > 0 do
        drucke die Ziffer, die zur Zahl  x mod b  gehört;
        ersetze x durch x div b
od
```

Wenn anfangs x=0 ist, müssen wir nur eine 0 ausdrucken.

Programm hierzu:

Beachte: Dieses Programm liefert die Ziffern in der falschen Reihenfolge, also von hinten nach vorne.

```
program  Darst_zur_Basis_Nat  is
declare  natural x, b;
begin
    read x; read b;
    if b < 2 then  write "Basis ist zu klein"
    else
        if x = 0 then  write '0'
        else
            while x > 0 do  write  x mod b; x := x div b  od
        fi
    fi
end
```

Hinweise: Texte schreiben wir stets in Anführungszeichen "...", einzelne Zeichen in Apostroph. write *x mod b* soll die Ziffer ausdrucken, die zur Zahl x mod b gehört.

Fall 2: Die Zahl x liegt zwischen 0 und 1: $0 < x < 1$.

Es sei $0.z_{-1}z_{-2}z_{-3}z_{-4}$ ... die Zahldarstellung von x zur Basis b.
Dann muss gelten:
z_{-1} = trunc $(x * b)$ (= ganzzahliger Anteil von $x * b$)
und die Zahl $x * b - z_{-1}$ besitzt die Zahldarstellung $0.z_{-2}z_{-3}z_{-4}$ ...
Dies liefert bereits den Algorithmus:

```
while x > 0 do
        drucke die Ziffer, die zur Zahl  trunc (x * b)  gehört;
        ersetze x durch  x * b - trunc (x * b)
od
```

Sobald x=0 wird, können wir abbrechen, da dann nur noch Nullen folgen können.
Allerdings sollte die Schleife nicht unendlich oft durchgeführt werden.

Programm (für 0 < x < 1):

Dieses Programm liefert die Ziffern der Nachkommastellen in der richtigen Reihenfolge.

```
program  Darst_zur_Basis_reell_0_1  is
declare  real x;  natural b;  integer y;
begin
    read x; read b;
    if b < 2 then  write "Basis ist zu klein"
    else
        if (0 < x) and (x < 1) then
            while x > 0 do y := trunc (x * b); write y; x := (x * b) - y od
        else  write "Falsche Eingabe für die reelle Zahl"
        fi
    fi
end
```

Hinweise: y ist stets eine natürliche Zahl von 0 bis b-1; ausgedruckt werden soll die zugehörige Ziffer. Die while-Schleife endet in der Regel nicht. In der Praxis sind x * b und (x * b) - y manchmal nicht zulässig, da x, y und b verschiedene Datentypen haben.

Fall 3: Allgemeiner Fall:
Zifferndarstellung für eine beliebige reelle Zahl x.

Falls x negativ ist, gib ein Minuszeichen '-' aus.
Gehe zum Absolutbetrag a := abs x über.
Es ist g := trunc(a) der ganzzahlige Anteil von a.
Weiterhin ist f := a - g der Nachkommaanteil von a.
Bearbeite nun g nach Fall 1 und f nach Fall 2.

Noch vorhandene Schwächen:
- Es müssen Ziffern (statt Zahldarstellungen) ausgegeben werden.
- Die zweite while-Schleife endet in der Regel nicht.
- Fall 1 gibt die Ziffern in der falschen Reihenfolge aus.

Aber wir schreiben den bisherigen Stand erst einmal auf.

Programm (für beliebiges reelles x):

Noch fehlerhaft

```
program  Darst_zur_Basis_reell  is
declare  real x, a, f;  natural b;  integer y, g;
begin
    read x; read b;
    if b < 2 then  write "Basis ist zu klein"
    else
        if (x < 0) then  write '-'  fi;
        a := abs x; g := trunc(a); f := a - g;
        if g = 0 then  write '0'
        else  while g > 0 do  write  g mod b; g := g div b  od
        fi;
        if f > 0 then  write '.';
            while f > 0 do y := trunc(f * b); write y; f := (f * b) - y od
        fi
    fi
end
```

Lösung der noch offenen Fragen.

Frage 1: Wie gibt man Ziffern (statt Zahlen) aus?

Statt write *g mod b*; und write *y;* müssen die Zeichen (vom Typ character) ausgegeben werden. Wie üblich soll für eine Basis b die Zahldarstellung mit Ziffern für 0, 1, 2, ..., b-1 erfolgen. Für die ersten zehn Ziffern nehmen wir die Zeichen '0', '1', '2', '3', '4', '5', '6', '7', '8' und '9'; die Ziffer zehn stellen wir durch 'A', die Ziffer elf durch 'B' usw. bis zur Ziffer fünfunddreißig durch 'Z' dar. Hiermit können wir alle Zahlen bis zur Basis b = 36 darstellen.

Andere Darstellungen sind auch üblich, insbesondere die Darstellung einer Ziffer durch die Dezimaldarstellung ihrer zugehörigen Zahl eingeschlossen in Apostrophe, also '10' für die Ziffer zehn, '11' für elf, ..., '51' für einundfünfzig usw.

Wir entscheiden uns hier für die erstgenannte Darstellung (mit $2 \leq b \leq 36$) und beschränken zugleich die Basis auf 36. Dann können wir die Frage 1 leicht lösen mit Hilfe der Operation Val, die auf character definiert ist und zu jeder Zahl n das n-te Zeichen (beginnend mit 0) in der Aufzählung ermittelt. Aus der Tabelle für Latin-1 entnehmen wir:
Val(48) = '0', ..., Val(57) = '9', Val(65) = 'A', ..., Val(90) = 'Z'.
Man rechnet leicht um für jedes k mit $0 \leq k \leq 35$:
if k < 10 then write Val(48+k) else write Val(55+k) fi;
Wir ersetzen also write *g mod b*; und write *y;* durch

```
k := g mod b;
if k < 10 then write Val(48+k) else write Val(55+k) fi;
```

bzw.

```
if y < 10 then write Val(48+y) else write Val(55+y) fi;
```

Lösung der noch offenen Fragen.

Frage 2: Wie beendet man die zweite while-Schleife stets?

Hier fällt uns nichts besseres ein, als die Zahl der Schleifendurchläufe mitzuzählen und die Schleife spätestens dann zu beenden, wenn diese Zahl eine vorgegebene Größe "max" überschreitet. Wir ersetzen

```
while f > 0 do y := trunc (f * b); write y; f := (f * b) - y od
```

also durch

```
max := ...; durchläufe := 0;
while (f > 0) and (durchläufe < max) do
    y := trunc (f * b); write y; f := (f * b) - y;
    durchläufe := durchläufe +1
od
```

Anfangs muss "max" auf einen geeigneten Wert, den der Benutzer des Programms festlegt (z.B. 200), gesetzt werden.

Lösung der noch offenen Fragen.

Frage 3: Wie bringt man die Ausgabe der Ziffern des ganzzahligen Anteils g in die richtige Reihenfolge?

Einfachste Lösung: Speichere die Ziffern in neuen Variablen H_0, H_1, H_2, ..., H_m und drucke, nachdem die while-Schleife beendet ist, diese in der Reihenfolge H_m, H_{m-1}, H_{m-2}, ..., H_1, H_0 aus. Die Variablen H_i müssen alle vom Typ character sein. Der Wert für m ergibt sich im Laufe der Rechnung. m-1 ist genau die Anzahl der Durchläufe durch die while-Schleife. Wir ersetzen also die bisherige erste while-Schleife (sie wurde hier bereits modifiziert entsprechend Frage 1)

```
while g > 0 do  k := g mod b;
    if  k < 10  then write Val(48+k) else write Val(55+k) fi;
    g := g div b
od
```

Erste while-Schleife ersetzen durch:

```
natural k, m;
character H0, H1, H2, ..., Hm ;
...
...

m := 0;
while g > 0 do
    k := g mod b;
    if  k < b  then Hm := Val(48+k) else Hm := Val(55+k) fi;
    g := g div b;
    m := m+1
od ;
while m > 0 do  m := m-1; write Hm  od;
```

zu den Deklarationen hinzufügen

Nun haben wir aber eine Unklarheit in unsere Programmiersprache gebracht: Was bedeutet genau

character $H_0, H_1, H_2, ..., H_m$?

Als Mensch interpretiert man die Punkte "..." hoffentlich richtig, eine Maschine braucht jedoch eine klare Festlegung. Da eine Zusammenfassung von r gleichartigen Variablen ständig auftritt (Vektoren, Matrizen, Auflistungen), führen wir hierfür eine eigene Deklarationsmöglichkeit ein, die sog. Feldvereinbarung:

array [0..r] of character H

Bedeutung: Es werden r+1 Variablen vom Typ character vereinbart. Diese werden insgesamt mit H bezeichnet. Auf die einzelnen Komponenten kann man mittels H[i] für i = 0, 1, ..., r zugreifen.
Wir schreiben nun also:

```
natural k, m;
array [0..m] of character H;
...
...

m := 0;
while g > 0 do
    k := g mod b;
    if  k < 10  then H[m] := Val(48+k) else H[m] := Val(55+k) fi;
    g := g div b;
    m := m+1
od;
while m > 0 do  m := m-1; write H[m]  od;
```

zu den Deklarationen hinzufügen

Nun bleibt aber noch ein Problem bestehen: Wenn jemand dieses Programm liest und ausführen möchte, so muss er, sobald er die Deklaration array [0..m] of character H; erreicht, genau m+1 Behälter H[0], H[1], ..., H[m] anlegen. Die Variable m besitzt jedoch hier noch keinen Wert, denn sie ist hier noch ein leerer Behälter bzw. ein Behälter mit undefiniertem Inhalt.

Dieses Problem lösen wir zunächst dadurch, dass wir eine Konstante (z.B. 500) wählen und ein Feld von Variablen

array [0..500] of character H;

deklarieren. Sollten dann mehr als 501 Variablen H[i] benötigt werden, so muss man das Programm leider abbrechen (und es ggf. mit einer größeren Konstanten erneut durchrechnen).
(Später lösen wir dieses Problem mit Hilfe von Blöcken.)

Nun fügen wir alle Modifikationen, die in den drei Fragen auftraten, hinzu und beachten noch, dass b höchstens 36 sein und m höchstens 500 werden darf. Dann ergibt sich folgendes Programm zur Darstellung einer reellen höchstens 500-stelligen Zahl zur Basis b, die zwischen 2 und 36 liegen darf:

```
program  Darst_zur_Basis_reell  is
declare   real x, a, f;  natural b, k, m, max, durchläufe;
          integer y, g;  array [0..500] of character H;
begin
    read x;  read b;
    if b < 2 then write "Basis ist zu klein"
    else
    if b > 36 then write "Basis ist zu groß"
    else  -- hier steht das eigentliche Programm, síehe nächste Folie
    fi fi
end
```

Füge oben zwischen else und fi ein:

Programm zur Lösung der Zahlendarstellungen reeller Zahlen.

```
a := abs x;  g := trunc(a);  f := a - g;
if  g = 0  then  write '0'
else   m := 0;
       while (g > 0) and (m < 501) do
            k := g mod b;
            if  k < 10  then H[m] := Val(48+k) else H[m] := Val(55+k) fi;
            g := g div b;  m := m+1
       od ;
       if g = 0  then
                  if (x < 0) then  write '-'  fi;
                  while m > 0 do  m := m-1; write H[m]  od
       else  f:=-1.0;  write "Abbruch, da die Zahl mehr als 500-stellig ist"  fi
fi;
if f > 0 then    write '.';  max := 200;  durchläufe := 0;
                 while (f > 0) and (durchläufe < max) do
                      y := trunc (f * b);
                      if  y < 10  then write Val(48+y) else write Val(55+y) fi;
                      f := (f * b) - y;  durchläufe := durchläufe +1
                 od;
                 if  durchläufe ≥ max  then
                      write "  Abbruch nach 200 Nachkommastellen"  fi
fi;
```

Was wir nun als Ergebnis erhalten haben, ist vermutlich eine korrekte Lösung, aber ein schlecht lesbares Programm.
Schon nach zwei Wochen werden wir selbst nicht mehr verstehen, was wir da eigentlich geschrieben haben und was die einzelnen Programmteile bewirken.
Wir verlangen daher:

- Erläuterungen, Kommentare müssen im Programm stehen!
- Logisch zusammengehörige Teile müssen auch als solche identifizierbar sein (zum Beispiel: Man ziehe solche Teile als Unterprogramme heraus, siehe später).
- Konstanten (hier: 200, 500 und 501) dürfen nicht irgendwo im Programm versteckt sein, sondern müssen klar erkennbar und auswechselbar sein.
- Die Namen müssen aussagekräftig sein.
- Verschachtelungen müssen sichtbar gemacht werden (dies haben wir aber durch die Einrückungen gut realisiert!)

Beachten Sie daher:

1. Programme müssen für alle denkbaren Eingaben korrekt arbeiten und genau das realisieren, was "spezifiziert" wurde.
2. Programme müssen gut lesbar und klar verständlich sein; sie müssen den Lösungsansatz und die Einzelheiten der Lösungsteile deutlich erkennen lassen.

Zu einem Programm gehören daher stets auch folgende Dokumente: die Problembeschreibung und die Spezifikation der erwarteten Leistungen des fertigen Systems, eine Ist- und Soll-Analyse, die Beschreibung der künftigen Einsatz-Umgebung, die Auflistung von Eigenschaften des Problems, der Lösungs-ansatz, die Aufteilung in Teilprobleme, die "Architektur" der Lösung und deren Lösungen, Aussagen zur Implementierung usw. Das fertige Programm ist eigentlich nur die "Spitze dieses Eisbergs".

All diese Erfordernisse hält man für unwichtig, wenn man sich nur mit kleinen überschaubaren Problemen (ggT, Primzahlen, Teiltext suchen, Sortieralgorithmus u.ä.) befasst. Es ist daher zu Beginn der Ausbildung und in der Schule nur schwer zu vermitteln, eine gewisse Sorgfalt und Disziplin einzuhalten.

Später kommen weitere Punkte hinzu, zum Beispiel:

3. Einmal erzeugte Lösungen müssen erweiterbar, anpassbar, übertragbar und wiederverwendbar sein und sie müssen sich in beliebige Software-Umgebungen leicht einbetten lassen.
4. Lösungen müssen sich aus meist hochkomplexen Bausteinen zusammensetzen und leicht zuschneidern lassen.
5. Der Einsatz der Software wird frühzeitig simuliert und die Erfordernisse werden entsprechend ständig angepasst. Im späteren Einsatz sollte die Übereinstimmung von Zielen und Realität laufend überwacht werden.

2. Einfache Datenstrukturen

Sind Datentypen gegeben, so kann man aus ihnen neue Datentypen erzeugen. Man orientiert sich hierbei an den mathematischen Operationen auf Mengen. Wichtige Operationen:

- Bildung von Unterbereichen (Intervalle, Projektion), → 2.1
- kartesisches Produkt (Datensätze, "record"), → 2.3
- kartesisches Produkt mit sich (Vektoren, Matrizen, "array"), ↘ 2.2
- disjunkte Vereinigung ("varianter record"), → 2.4
- Folgen-Bildung (Wörter-Bildung, "Listen"),
- Menge der Teilmengen (Potenzmenge, "set of"),
- Menge von Abbildungen (Verallgemeinerung der "function")
- Menge der Terme, Hierarchiebildung (Bäume)
- Menge der zweistelligen Beziehungen (Graphen)
- Menge der Namen oder Adressen (reference, access, Zeiger).

Die ersten vier zählen wir zu den "einfachen Datenstrukturen".

Unter einer "Datenstruktur" verstehen wir hier einen Datentyp, der aus einfacheren Datentypen nach einem bestimmten Schema entsteht und der bis auf Zugriffsoperationen keine weiteren Operatoren neu benötigt oder einführt.

Dieses Schema ist im einfachsten Fall ein "Konstruktor".

Zum Beispiel wirkt ein Konstruktor "Paarbildung" auf die beiden Datentypen integer und natural wie folgt: Aus den Wertebereichen $\mathbb{Z}$ und $\mathbb{N}_0$ wird der Wertebereich $\mathbb{Z} \times \mathbb{N}_0$.

Formal schreiben wir

type paar is record integer Z; natural N end record;

und eine Variable X vom Typ paar, also

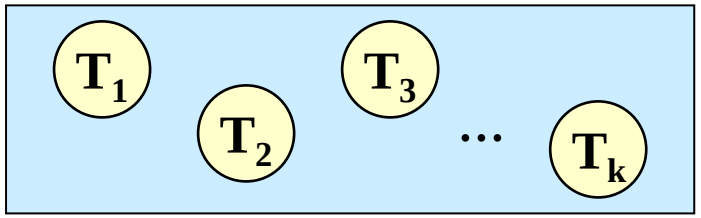
paar X

ist ein Behälter, der die zwei Komponenten X.Z und X.N (= Teil-Variablen = Unter-Behälter) besitzt.

Grundsätzliches Vorgehen

Zu jedem Konstruktor K, der auf k Datentypen T_i wirkt,

$$K\,(T_1, T_2, ..., T_k)$$



wird angegeben:

(i) Wie sieht die Wertemenge aus (bezogen auf die Wertemengen der Datentypen T_i)?

(ii) Wie kann man auf die einzelnen Komponenten T_i zugreifen?

(iii) Welche Operationen der Datentypen T_i werden übernommen und in welcher Form?

(iv) Werden neue Operationen hierdurch eingeführt? (Vor allem Umwandlungen in einen anderen Datentyp.) ((iv) sollte bei einfachen Datenstrukturen nicht geschehen.)

2.1 Unterbereiche

Unterbereich oder Intervall $a..b = \{x \in M \mid a \leq x \leq b\}$.

Wir können dem Unterbereich auch einen Namen geben:

type <Name> is <Datentyp> <Einschränkung>

z.B.: type Kohl_Aera is integer 1982..1998;

<Datentyp> darf fehlen, wenn der Wertebereich klar ist.

Alle Operationen von <Datentyp> gelten auch für den durch <Name> bezeichneten Unterbereich, sofern der Unterbereich hierbei nicht verlassen wird (man kann auch festlegen, dass bei den Zwischenrechnungen der Unterbereich nicht verlassen werden darf).

Variante: Der Unterbereich darf unbestimmt ("flex", Abk. von flexible) bleiben; die Grenzen werden bei weiteren Deklarationen festgelegt oder die zugehörige "≤"-Beziehung entfällt. Z.B.: flex..b = $\{x \in M \mid x \leq b\}$, die Relation $a \leq x$ entfällt also.

```
type natural is integer 0 .. maxinteger;
   (hierbei sei maxinteger die größte darstellbare ganze Zahl)
type Kleiner8000 is natural 0..7999;
type Bis_Heute is integer flex..2004;
type Wochentage is (Mo, Di, Mi, Do, Fr, Sa, So);
type Arbeitstage is Wochentage Mo..Fr;
 ...
natural M;      -- die Werte für M sind aus IN0 = {z∈ Z | z≥0}
Kleiner8000 K;  -- der Wertebereich von K ist {0, 1, ..., 7999}
Bis_heute (1900) H;  -- hier wird 1900 für flex eingesetzt und
                     -- der Wertebereich von H ist {1900, 1901, ..., 2004}
Bis_heute J;  -- hier ist der Wertebereich { ..., 2002, 2003, 2004}
```

Beantwortung der vier Fragen für Unterbereiche

Gegeben: ein Datentyp T mit dem Wertebereich M. Der Daten-Konstruktor ist T unten..oben .
V sei eine Variable dieses neuen Datentyps.

(i) Wertemenge: unten..oben = $\{x \in M \mid unten \leq x \leq oben\}$.

(ii) Zugriff auf die Komponenten: entfällt, V enthält den Wert.

(iii) Alle Operationen des Datentyps T sind auch für V zulässig. Wird V ein Wert zugewiesen, so muss geprüft werden, ob er im Intervall unten..oben liegt.

(iv) Werden neue Operationen hierdurch eingeführt? Nein.

2.2 Felder

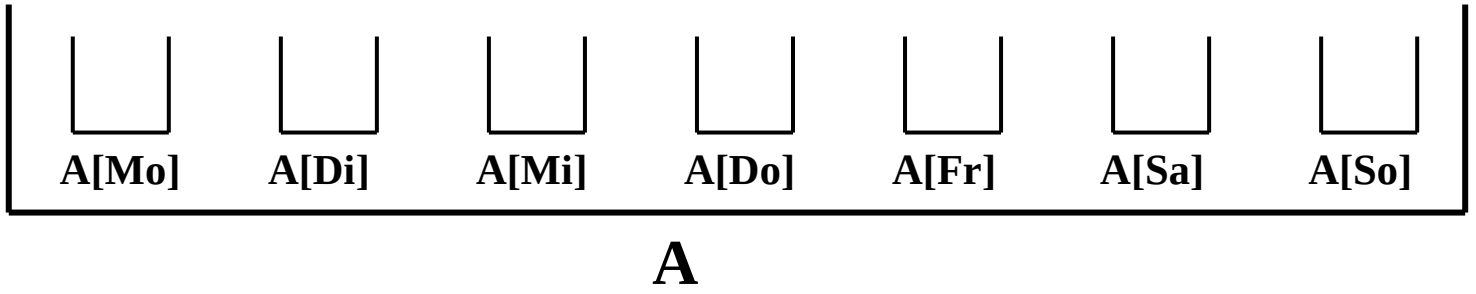
Gegeben: ein Datentyp T_2 mit Wertemenge M und ein Datentyp T_1 mit einem endlichen Wertebereich D, |D| = r. Datentyp T für das r-fache kartesische Produkt M^r mit sich:

type T is array [T_1] of T_2 .

T_1 heißt der *Indextyp*, T_2 heißt der *Komponententyp* von T.

Beispiel:

type Arbeitsbeginn is array [Wochentag] of 0..23;
Arbeitsbeginn A;
...
A[Mo] := 9; A[Di] := 8; A[Mi] := A[Di] - 1; ...



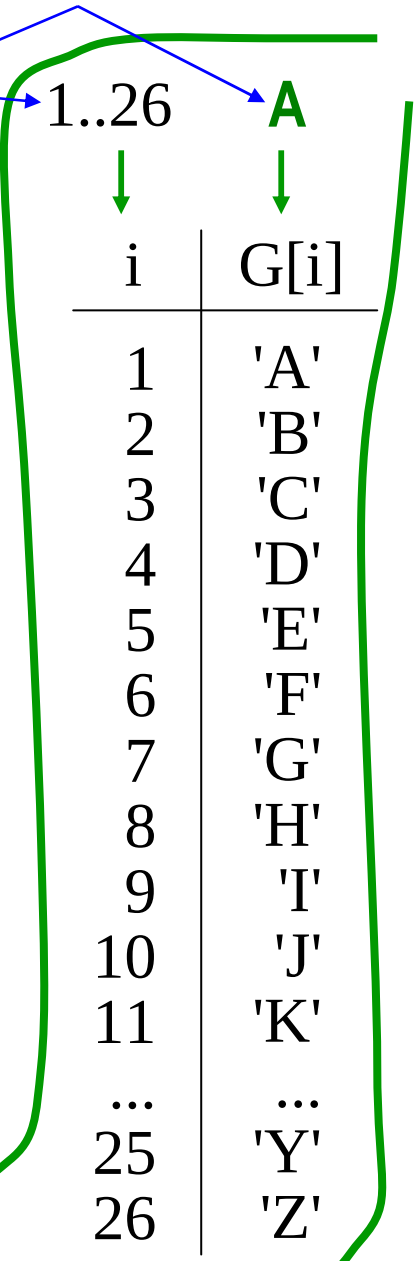
Genauer: Der Konstruktor array [ ] of wirkt auf zwei Datentypen:

array [T_1] of T_2

Besitzen T_1 die Wertemenge D mit r Elementen und T_2 die Wertemenge M, so ist genau genommen die Menge der Abbildungen M^D die zu array [T_1] of T_2 gehörende Wertemenge.

Für ein beliebiges Element i ∈ D greift man auf den zu i gehörenden Wert in M zu mittels [i] .

Ein Element aus array [T_1] of T_2 ist somit eine als Tabelle dargestellte Abbildung von D nach M (dies kann auch als r-stelliger Vektor geschrieben werden).



Großbuchstaben: G: array [1..26] of character

Beantwortung der vier Fragen

Gegeben: zwei Datentypen T_1 und T_2 mit den Wertebereichen D und M. Der Daten-Konstruktor ist array $[T_1]$ of T_2 .
V sei eine Variable von diesem neuen Datentyp.

(i) Wertemenge: M^D (= Menge der Abbildungen von D nach M).

(ii) Zugriff auf die Komponenten: V[i] für $i \in D$. Meist ist der Zugriff auf die ganze Struktur erlaubt (Aggregatbildung). Bei n Dimensionen: $V[i_1, i_2, ..., i_n]$, siehe unten.

(iii) Alle Operationen des Datentyps T_2 sind auch für V[i] zulässig. Statt i kann ein Ausdruck vom Typ T_1 stehen, also ein Ausdruck, der einen Wert aus D als Ergebnis hat.

(iv) Werden neue Operationen hierdurch eingeführt? Nein. (Außer der Verwendung von Aggregaten für komplette Zuweisungen und Abfragen.)

Betrachte nochmals array [1..r] of T.

Hierbei darf T erneut ein Feldtyp sein usw.:

array [1..r] of array [1..s] of U
array [1..r] of array [1..s] of array [1..t] of V

Dies kürzt man wie folgt ab:
array [1..r, 1..s] of U und array [1..r, 1..s, 1..t] of V.
Statt 1..r kann jeder andere Indextyp verwendet werden.

Die Anzahl dieser Indexbereiche nennt man die *Dimension* des Feldes. In der Praxis sind d-dimensionale Felder meist für d = 1, 2 und 3 üblich (z.B. Vektoren, Matrizen, Bildpunkte).

Ein Feld heißt *statisch*, wenn zur Übersetzungszeit (also ohne Kenntnis von Eingabewerten) alle Feldgrenzen bekannt sind. Wird mindestens eine Feldgrenze erst zur Laufzeit des Programms berechnet, so heißt das Feld *dynamisch*.

Unendliche Indexbereiche: In der Programmierung sind Wertebereiche der Form M^{∞} ungebräuchlich, auch wenn sie in der Mathematik eine bedeutende Rolle spielen.

Man könnte einen Datentyp array [1..flex] of ... einführen, wobei das Symbol "flex" (=flexibel) wie in 2.1 bedeutet, dass die obere Grenze offen bleibt und sich während der Berechnungen ändern kann. Natürlich sind dann auch array [1..flex, flex..350, flex..flex] of ... möglich.

Diese Möglichkeit gibt es in manchen Sprachen, z.B. in Algol 68 und in gewisser Form auch in Ada (range <>).

Beispiel: Intervallschachtelung (oder binäre Suche)

Ein Feld A: array [1..n] of integer sei gegeben. Das Feld sei sortiert, d.h.: $A[i] \leq A[i+1]$ für $i = 1, 2, ..., n-1$.

Aufgabe: Man schreibe einen Algorithmus, der zu einer Zahl S in möglichst kurzer Zeit feststellt, ob S im Feld A liegt oder nicht. Im Falle, dass S im Feld A enthalten ist, soll ein Index m mit A[m] = S ausgegeben werden, anderenfalls sei m = 0.

Geht man das Feld von links nach rechts durch, so dauert es bis zu n Schritte, um das Ergebnis zu ermitteln.

Ein schnelleres Verfahren ist die *Intervallschachtelung:* Teste, ob S genau in der Mitte A[Mitte] von A liegt, falls nein und es ist A[Mitte] < S, suche rechts von der Mitte weiter, sonst links. Dies führt zu folgendem Programm:

Programm 1: Wir setzen hier "maximales n" = 100000.

```
program Search1 is
integer n, Mitte, Links, Rechts, S; Boolean Gefunden;
array [1..100000] of integer  A;
begin  ...;  -- "lies n, das Feld A und die zu suchende Zahl S ein"
     Links:=1; Rechts := n; Gefunden := false;
     while (Links ≤ Rechts) and (not Gefunden) do
          Mitte := (Rechts+Links) div 2;
          if A[Mitte] = S then  Gefunden := true
          else  if A[Mitte] < S  then Links := Mitte+1
                                 else Rechts := Mitte-1 fi  fi
     od;
     if not Gefunden then Mitte := 0 fi;
     ...                        -- "drucke das Ergebnis Mitte aus"
end
```

Wie lange dauert es, bis dieser Algorithmus spätestens endet?

Hierzu zählen wir die Wertzuweisungen und Bedingungen, die im ungünstigsten Fall ausgerechnet werden müssen.

In diesem Sinne dauert die Durchführung der 3 Anweisungen

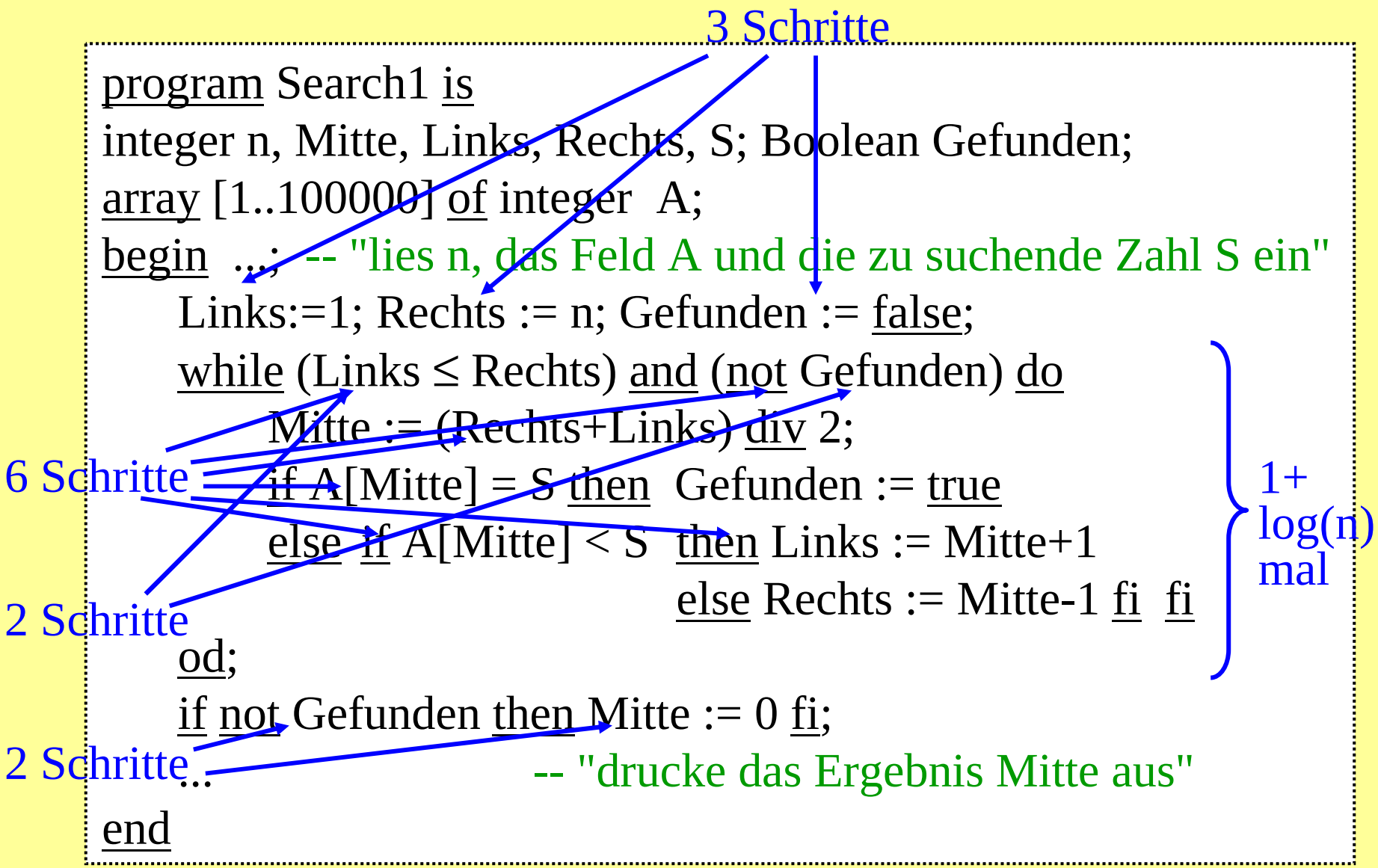
Links:=1; Rechts := n; Gefunden := false;

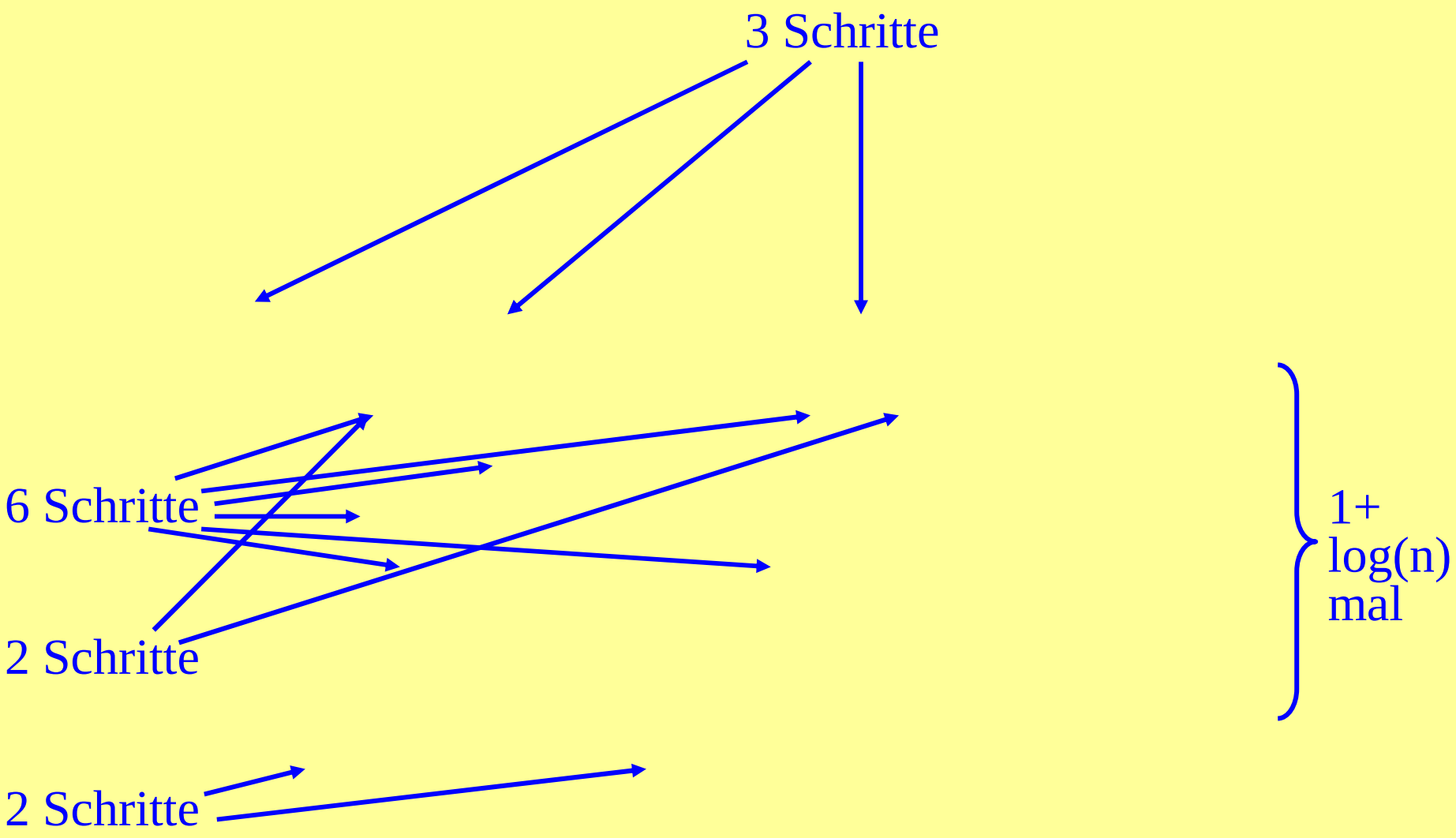
genau 3 Schritte.

In der Schleife werden maximal 6 Schritte benötigt, nämlich je einer für die vier Bedingungen (Links <= Rechts), (not Gefunden), A[Mitte] = S und A[Mitte] < S und je einer für zum Beispiel die Wertzuweisungen

Mitte := (Rechts+Links) div 2; und Links := Mitte+1;

Wie oft wird die Schleife durchlaufen? Das Intervall von Links bis Rechts halbiert sich mindestens in jedem Schritt, folglich muss nach log(n) Schleifendurchläufen Schluss sein.


3 Schritte
program Search1 is
integer n, Mitte, Links, Rechts, S; Boolean Gefunden;
array [1..100000] of integer A;
begin ...; -- "lies n, das Feld A und die zu suchende Zahl S ein"
Links:=1; Rechts := n; Gefunden := false;
while (Links ≤ Rechts) and (not Gefunden) do
Mitte := (Rechts+Links) div 2;
6 Schritte
if A[Mitte] = S then Gefunden := true
else if A[Mitte] < S then Links := Mitte+1
else Rechts := Mitte-1 fi fi
2 Schritte
od;
if not Gefunden then Mitte := 0 fi;
2 Schritte
... -- "drucke das Ergebnis Mitte aus"
end
1+
log(n)
mal



Gesamt: 6·log(n) + 13 Schritte ∈ O(log(n))

Einschub zum Symbol O( ), Vorgriff auf Kapitel 6.1:

O(...) ist die "maximale Größenordnung", gesprochen "groß Oh".

f sei eine Funktion. O(f) ist eine Menge von Funktionen. Und zwar liegt die Funktion g in O(f), wenn g höchstens so stark wächst wie f, wobei Konstanten nicht zählen. Statt *g ist höchstens von der Größenordnung f*, sagen wir, *g ist groß-O von f,* und meinen damit, dass $g \in O(f)$ ist.

Formale Definition: Es sei $f: \mathbb{R}^+ \to \mathbb{R}^+$ eine Funktion über den positiven reellen Zahlen $\mathbb{R}^+ \subset \mathbb{R}$.

$$\mathbf{O(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{R}^+ \; \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}.$$

(Oft wird diese Definition auf die natürlichen Zahlen beschränkt, also auf Funktionen $f: \mathbb{N}_0 \to \mathbb{N}_0$; in der Definition müssen dann nur $g: \mathbb{R}^+ \to \mathbb{R}^+$ durch $g: \mathbb{N}_0 \to \mathbb{N}_0$ sowie $n_0 \in \mathbb{N}_0$ ersetzt werden.)

Der ungünstigste oder schlechteste Fall (der "worst case") kann auch tatsächlich eintreten, wenn nämlich das gesuchte Element S nicht im Feld A enthalten ist. Wir sagen: Die *uniforme worst case Zeitkomplexität t(n)* des Programms Search1 lautet

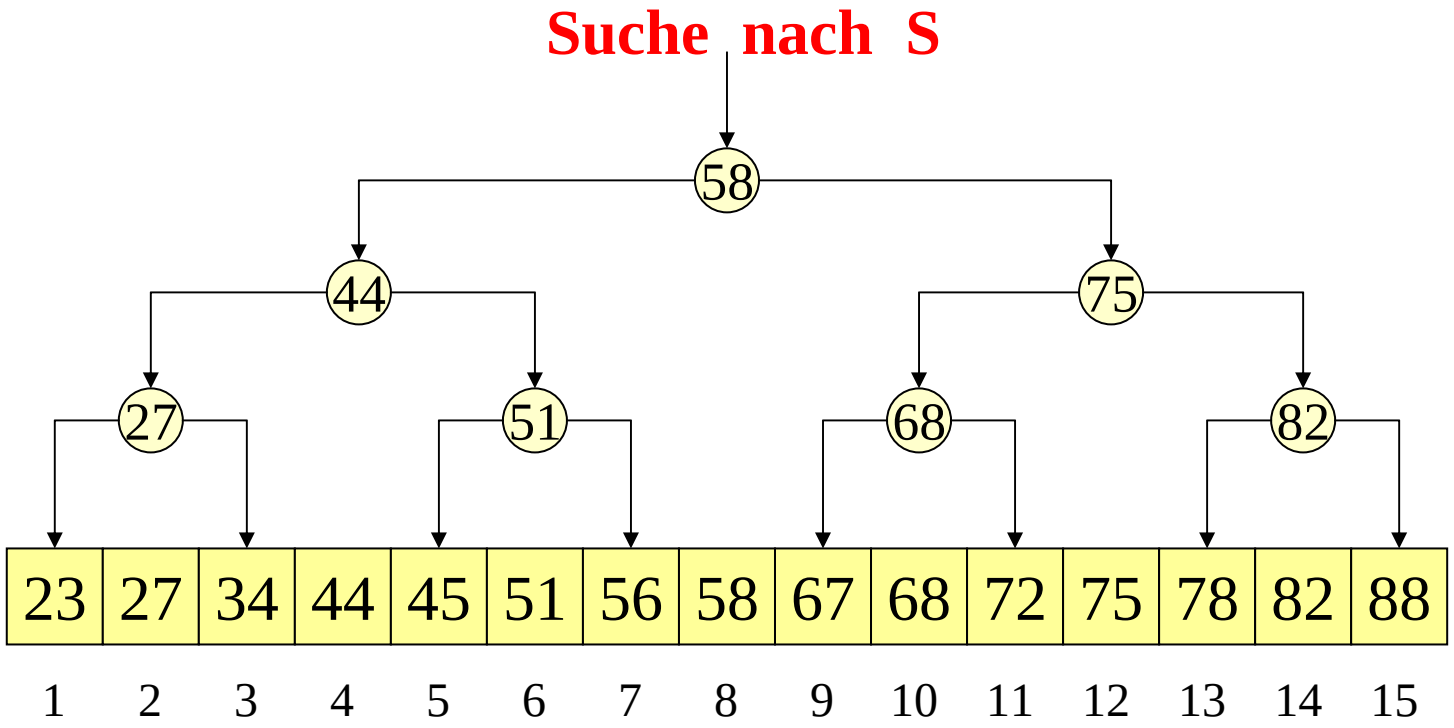
$$t(n) = 6 \cdot \log(n) + 13.$$

Beachten Sie: n ist hier die Anzahl der Elemente im Feld A. "Uniform", weil wir annehmen, alle Wertzuweisungen und Bedingungen würden die gleiche Zeit kosten.

Was ist der beste Fall? In diesem Fall wird S im ersten Durchgang der while-Schleife gefunden, d.h., dann ist das Element S nach 13 Schritten gefunden.

Mit wie vielen Schritten muss man im Mittel rechnen? Hierzu nehmen wir an, dass sich das gesuchte Element S tatsächlich im Feld A befindet (sonst kann man nur die obige worst case Abschätzung verwenden).

Wir skizzieren die Verhältnisse, wobei wir hier $n=15=2^4-1$ wählen:



In $2^3 = 8$ Fällen braucht man 4 Schleifendurchläufe,
in $2^2 = 4$ Fällen braucht man 3 Schleifendurchläufe,
in $2^1 = 2$ Fällen braucht man 2 Schleifendurchläufe,
in $2^0 = 1$ Fall braucht man 1 Schleifendurchlauf.

Allgemein gilt also, wenn $n = 2^k - 1$ ist:

In 2^{k-1} Fällen braucht man k Schleifendurchläufe,
in 2^{k-2} Fällen braucht man k-1 Schleifendurchläufe,
in 2^{k-3} Fällen braucht man k-2 Schleifendurchläufe,
....
in $2^0 = 1$ Fall braucht man 1 Schleifendurchlauf.

Daher braucht man im Mittel:

$$\frac{1}{n}\,(k\cdot 2^{k-1} + (k-1)\cdot 2^{k-2} + (k-2)\cdot 2^{k-3} + \ldots + 2\cdot 2^1 + 1) \text{ Durchläufe.}$$

Berechne also die Summe $\displaystyle\sum_{j=1}^{k} j\cdot 2^{j-1} = \frac{1}{2}\sum_{j=1}^{k} j\cdot 2^j$

$$\sum_{j=1}^{k} j\cdot 2^{j-1} = \frac{1}{2}\sum_{j=1}^{k} j\cdot 2^{j} = \frac{1}{2}\sum_{j=1}^{k} (j-1)\cdot 2^{j} + \frac{1}{2}\sum_{j=1}^{k} 2^{j}$$

$$= \sum_{j=1}^{k} (j-1)\cdot 2^{j-1} + \frac{1}{2}(2^{k+1} - 2)$$

$$= \sum_{j=0}^{k-1} j\cdot 2^{j} + (2^{k} - 1) = \sum_{j=1}^{k} j\cdot 2^{j} - k\cdot 2^{k} + (2^{k} - 1), \quad \text{d.h.:}$$

$\frac{1}{2}\sum_{j=1}^{k} j\cdot 2^{j} = k\cdot 2^{k} - (2^{k} - 1)$. Folglich erhalten wir

$$\frac{1}{n}(k\cdot 2^{k-1}+(k-1)\cdot 2^{k-2}+\ldots+2\cdot 2^{1}+1) = \frac{k\cdot 2^{k} - (2^{k} - 1)}{2^{k} - 1} \approx k - 1$$

Somit beträgt die *average case Zeitkomplexität* der Intervallschachtelung ziemlich genau $6\cdot\log(n) + 7$ Schritte, also nur einen Schleifendurchlauf weniger als im schlechtesten Fall.

Erkenntnis: Im Mittel spart man nur eine konstante Zahl an Operationen gegenüber dem schlechtesten Fall. Folglich lohnt sich zum Beispiel die Abfrage " A[Mitte] = S " nicht; kann man sie weglassen, so würde man log(n) viele Schritte sparen. Dies führt auf folgende bessere Version des Algorithmus für die Suche mittels Intervallschachtelung:

Man entscheide erst ganz am Ende, ob A[Mitte] = S ist; hierzu muss man im Falle A[Mitte] < S im rechten Teil des Feldes weitersuchen (Links:=Mitte+1), anderenfalls im linken Teil einschließlich des gerade betrachteten Feldes Mitte (Rechts:=Mitte, statt "Mitte+1", da A[Mitte] gleich S sein könnte).

Programm 2: erst am Ende testen, ob S im Feld vorhanden ist

```
program Search2 is
integer n, Mitte, Links, Rechts, S; Boolean Gefunden;
array [1..100000] of integer  A;
begin  ...;  -- "lies n, das Feld A und die zu suchende Zahl S ein"
      Links:=1; Rechts := n;
      while (Links < Rechts) do
            Mitte := (Rechts+Links) div 2;
            if A[Mitte] < S   then Links := Mitte+1
                              else Rechts := Mitte fi;
      od;
      Gefunden := A[Mitte] = S;
      if not Gefunden then Mitte := 0 fi;
      ...                          -- "drucke das Ergebnis Mitte aus"
end
```

Hinweis: Natürlich braucht man jetzt die Variable "Gefunden" nicht mehr und man kann die beiden Anweisungen

Gefunden := A[Mitte] = S; if not Gefunden then Mitte := 0 fi;

ersetzen durch

if A[Mitte] ≠ S then Mitte := 0 fi;

Weisen Sie nun nach, dass für diese Version Search2 gilt:

Die *uniforme worst case Zeitkomplexität* beträgt $9 + 4 \cdot \log(n)$ und die *uniforme average case Zeitkomplexität* besitzt genau den gleichen Wert.

Hierbei ist n die Zahl der Elemente im array.

Das schlechter erscheinende Programm Search2 ist also im Mittel um rund 33% schneller.

2.3 Records

Gegeben seien die Mengen M_1, M_2, ..., M_n, die zu den Datentypen T_1, T_2, ..., T_n gehören. Die Zusammenfassung zum Datentyp T mit dem kartesichen Produkt $M = M_1 \times M_2 \times ... \times M_n$ als Wertemenge ist:

type T is record S_1: T_1; S_2: T_2; ...; S_n: T_n end record;

Hierbei sind S_1, S_2, ..., S_n Namen, die so genannten "Selektoren", mit deren Hilfe man auf die einzelnen Komponenten des Datentyps T zugreifen kann. Man "selektiert" die i-te Komponente, indem man S_i durch einen Punkt getrennt hinter den Namen der Variable vom Typ T hängt (Punkt-Schreibweise, *dot-notation*).

Beantwortung der vier Fragen für kartesische Produkte

n Datentypen $T_1, T_2, ..., T_n$ mit den Wertebereichen $M_1, M_2, ..., M_n$.
Datentyp type T is record S_1: T_1; S_2: T_2; ...; S_n: T_n end record.
V sei eine Variable dieses neuen Datentyps T.

(i) Wertemenge: $M = M_1 \times M_2 \times ... \times M_n$

(ii) Zugriff: $V.S_i$ für die i-te Komponente. (Meist ist auch der Zugriff auf die ganze Struktur erlaubt, Aggregatbildung.)

(iii) Auf jeder Komponente bleiben die Operationen des zugehörigen Typs T_i erhalten.

(iv) Werden neue Operationen hierdurch eingeführt? Nein. (Außer der Verwendung von Aggregaten für komplette Zuweisungen und Abfragen.)

Beispiele

```
type Monatsname is (Januar, Februar, März, April, Mai, Juni,
        Juli, August, September, Oktober, November, Dezember);
type Datum is record
        integer Jahr;
        Monatsname Monat;
        1..31  Tag
end record;
```

In einer Variablen-Deklaration darf eine Initialisierung stehen!

```
Datum  Start_Fusssball_Europameisterschaft :=(2004,Juni,12);
Datum  Heute, Ende, W, U;      ...

Heute.Jahr := 2004; Heute.Monat := Dezember; Heute.Tag := 15;
```

Beispieldarstellungen aus Ada 95 für Aggregate:

```
if  Heute.Jahr = 2004  then  Ende := Datum'(2004,Juni,19)
else  Ende := Heute  fi;
W := (Tag => 25, Jahr => 2003, Monat => Dezember);
U := Datum'(2004, Monat => Mai, Tag => 1);
```

Beispiele (Fortsetzung)

```
--  Komplexe Zahlen:
type Komplex is record
        real  Realteil;
        real  Imaginärteil
end record;
--  Rationale Zahlen:
type Rational is record
        integer  Zähler;
        positive  Nenner
end record;
Rational P, R;  Boolean  Gleich, Eins, Wurzel2;  ...
Gleich := P.Zähler * R.Nenner = R.Zähler * P.Nenner;
Eins := P.Zähler = P.Nenner;
Wurzel2 := P.Zähler * P.Zähler = 2 * P.Nenner * P.Nenner;
```

Ist dies überhaupt erlaubt? Verschiedene Datentypen! Siehe später.

Beispiele (Fortsetzung)

Records spielen im täglichen Leben eine zentrale Rolle, weil sie die programmiersprachliche Beschreibung von Formularen sind. Wer ein Formular ausfüllt, füllt einen Verbundtyp mit Werten.

```
type Hotelformular is record
        Datum  Ankunftstag, Abreisetag;
        array [1..30] of character  Name;
        1880..2004  Geburtsjahr;
        array [1..30] of character  Wohnort;
        array [1..5] of character '0'..'9'  PLZ;
        array [1..30] of character  Strasse;
        positive  Hausnummer;
        Boolean  Allein, Raucherzimmer
end record
```

2.3 Variante Records

Gegeben: M_1, M_2, ..., M_n, die zu den Datentypen T_1, T_2, ..., T_n gehören. Zusammenfassung zum Datentyp T mit der disjunkten Vereinigung $M = M_1 \cup M_2 \cup ... \cup M_n$ als Wertemenge:

```
type T (1..n Index) is record
        case  Index  is
                when 1 => S1: T1;
                when 2 => S2: T2;   ...

                when n => Sn: Tn;
        end case
end record
```

Index gibt den aktuellen Wertebereich innerhalb von M an. Mit dem Selektor S_{Index} kann man auf den aktuellen Wert zugreifen. Die Auswahlvariable Index nennt man Diskriminante. An Stelle von 1..n kann ein anderer Datentyp "choice" stehen.

Beantwortung der vier Fragen für disjunkte Vereinigungen

Datentyp choice mit n-elementigem Wertebereich $\{a_1, a_2, ..., a_n\}$ und n Datentypen $T_{a_1}, T_{a_2}, ..., T_{a_n}$ mit den Wertebereichen $M_{a_1}, M_{a_2}, ..., M_{a_n}$. Obiger Datentyp T für $M_{a_1} \cup M_{a_2} \cup ... \cup M_{a_n}$. V sei eine Variable dieses neuen Datentyps T.

(i) Wertemenge: die disjunkte Vereinigung $M = M_{a_1} \cup M_{a_2} \cup ... \cup M_{a_n}$ zuzüglich des Wertebereichs $\{a_1, a_2, ..., a_n\}$, um die aktuelle Menge M_{a_i} festzulegen.

(ii) Zugriff: $V.S_{a_i}$, sofern die Diskriminante von V den Wert a_i besitzt. Der Wert a_i lässt durch V.Index ändern.

(iii) Auf jeder "Komponente" bleiben die Operationen des zugehörigen Typs T_{a_i} erhalten.

(iv) Werden neue Operationen hierdurch eingeführt? Nein.

Beispiele: Bei Studierenden wird meist nach Inländern, ausländischen EU-Bürgern und Ausländern, die nicht zur EU gehören, unterschieden.

```
type SB is (D, EU, sonst);
type Student (SB Herkunft) is record
    array [1..30] of character  Name;
    positive  Matrikel_Nummer;
    case  Herkunft  is
        when D => array [1..40] of character  Geburtsort;
        when EU | sonst => array [1..25] of character  Land
    end case
end record
```

Hinweis: Es gibt auch die entsprechende "mehrfache Fallunterscheidung" bei den Kontrollstrukturen, die wir hier aber nicht behandelt haben; diese ist wie das obige case aufgebaut.

```
type Kategorie is (PKW, LKW, Bus, Karren);
type Fahrzeug (Kategorie Art) is record
      real  Länge, Breite, Höhe;
      case  Art  is
            when Bus => 8..60  Sitzplätze;  0..80  Stehplätze;
            when LKW => positive  Ladefläche;
            when PKW => 0..10  Airbagzahl;
            when Karren => null record
      end case
end record;
...
Fahrzeug  A (PKW); Fahrzeug B (Bus);
Fahrzeug C := (Bus, 15.856, 2.95, 3.13, 55, 12);
A := (LKW, 12.4, 2.75, 3.542, 21.66);   -- Aggregat-Zuweisung
A.Ladefläche := 14;  -- korrekt, da die Diskriminante hier LKW lautet
A.Airbagzahl := 4;    -- Fehler, da Diskriminante LKW (und nicht PKW) ist
B.Höhe := A.Höhe;   -- korrekt, da reelle Zahl zugewiesen wird
A.Art := LKW;         -- verboten, da Diskriminante nicht allein änderbar
A := (Karren, 2.9, 1.85, 1.02);      -- ebenfalls Aggregat-Zuweisung
```

null record soll "keine Komponente" bedeuten.

In einer Variablen-Deklaration darf eine Initialisierung stehen!

Weiteres Beispiel: Mit varianten Records lassen sich Variablen deklarieren, die (gesteuert durch die Diskriminante) Werte von verschiedenen Datentypen besitzen können, zum Beispiel eine Variable X, die Zeichen oder Zahlen enthält:

```
type Zwei (Boolean Wahl) is record
   case  Wahl  is
         when true => integer  Zahl;
         when false => character  Z  end case
end record;
Zwei  X;  integer  L := 17;
...
X := (false, 'R');  write Succ(X.Z);
X := (true, L+28);
X.Zahl := L*L;  write X.Zahl;  ...
```

In einer Variablen-Deklaration darf eine Initialisierung stehen!

Abschließender Hinweis: Gleichheit von Datentypen

Prinzipiell sollten alle verwendeten Datentypen einen Namen besitzen und untereinander verschieden sein. Variablen verschiedener Datentypen können ihre Werte nicht unmittelbar einander zuweisen. In den meisten Programmiersprachen sind jedoch Ausnahmen zugelassen.

Zum Beispiel bei der Zuweisung ganzzahliger Werte an reellwertige Variablen und in entsprechenden Ausdrücken.

Zum Beispiel beim Verschieben eines Teils innerhalb der eigenen Struktur. Beispiel: Man möchte die 20 Elemente von Nr. 40 bis 59 eines Feldes V auf die Positionen von 1 bis 20 verschieben (sofern so etwas zugelassen ist; in manchen Sprachen geht das, "Aggregatbildung" bei Feldern):
V[1..20] := V[40..59];

Konversion

In jeder Programmiersprache muss man Regeln festlegen, wie man von einem zum anderen Datentyp (speziell auch: von einem zum anderen Unterbereich) umschalten kann. Diese Konversionsregeln können eine "automatische" Anpassung oder eine "explizite" Anpassung bewirken.

In vielen Sprachen sind fast nur "explizite" Umwandlungen erlaubt, d.h., man muss stets genau angeben, wie man den Wert eines Datentyps zu einem Wert eines anderen Datentyps macht, also z.B. die ganze Zahl 2 in die reelle Zahl 2.0 umwandelt.

Bei objektorientierten Sprachen ist die Richtung "vom vererbten Typ zum Obertyp" automatisch zulässig.

In Java gibt es Umwandlungs-Operatoren.

3. (Künstliche) Sprachen

Gliederung:

Von diesem Kapitel 3 behandeln wir im Kurs nur die Teile

3.1 Definitionsmöglichkeiten
3.2 Sprachen und ihre Operationen
3.3 Kontextfreie Grammatiken
3.6 EBNF
3.7 Syntaxdiagramme
3.10 Zwei Aufgaben und Syntax von Java 1.1

Der übrige Text bietet vor allem Hintergrundwissen und Formalisierungen, die für den Schulbereich (noch?) nicht relevant sind, jedoch ein Licht darauf werfen, wie viel mathematische Denkweise in diesem Teil der Informatik steckt.

Am Ende dieses Kapitels sollten Sie die Syntax der Folien 160/161, 182 und 246/247 lesen und verstehen können.

Vorbemerkung

Wir haben Algorithmen mit Hilfe von Programmen beschrieben. Jetzt wollen wir präzise festlegen, wie Programme aufgebaut sein müssen.

Programme bestehen aus einer Folge von Zeichen. Diese Zeichen sind "character" und "key-words":

(1) Die Elemente von **A**, die auf der Tastatur zu finden sind.

(2) Besondere Wörter ("Schlüsselwörter" der Programmier-sprache) und zwar bisher die Elemente der folgenden Menge

SW = {program, procedure, function, return, is, declare, if, then, else, fi, begin, end, while, do, od, for, to, repeat, until, read, write, skip, halt, true, false, not, and, or, xor, equiv, abs, div, mod, rem, type, array, of, null, case, when}.

Wir unterstreichen die Schlüsselwörter stets. Jedes Schlüsselwort ist wie ein einzelnes Zeichen aufzufassen. (Weitere werden folgen.)

Ein Programm ist eine Zeichenfolge, wobei die Zeichen aus dem "Alphabet" $\mathbf{A} \cup SW$ stammen.

Wenn M eine Menge ist, so bezeichnet M* die Menge der endlichen Folgen von Elementen aus M einschl. der leeren Folge ε. Man nennt M* auch die Menge der Wörter über M. Konkatenation (=Hintereinanderschreiben) von Wörtern: $u, v \in M^*$, dann ist auch $uv \in M^*$. $u\varepsilon = u = \varepsilon u$ für alle Wörter u.

Ein Programm ist also ein Wort über $(\mathbf{A} \cup SW)$, d. h. ein Element aus der Menge $(\mathbf{A} \cup SW)^*$.

Eine Programmiersprache L ist daher eine Teilmenge dieser Menge von Wörtern, also $L \subseteq (\mathbf{A} \cup SW)^*$.

Man muss nun genau festlegen, welche Zeichenfolge ein Programm ist und welche nicht. Wir erläutern zunächst an dem Beispiel "Bezeichner" mehrere Möglichkeiten einer präzisen Definition.

3.1 Definitionsmöglichkeiten für Bezeichner und Ausdrücke

Als einführendes Beispiel untersuchen wir einen sehr einfachen Fall, nämlich den Aufbau von "Bezeichnern". Bezeichner sind Zeichenfolgen über Buchstaben, Ziffern und dem Unterstrich "_", die mit einem Buchstaben beginnen. Seien also

$\Phi_B = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$

die Menge der Buchstaben und

$\Phi_Z = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ die Menge der Ziffern und

$\Phi = \Phi_B \cup \Phi_Z \cup \{ \_ \}$ die Menge der 63 Zeichen, die für einen Bezeichner verwendet werden dürfen.

Definition (umgangssprachlich): Ein Bezeichner ist eine Folge von Elementen aus Φ, die mit einem Buchstaben (d.h. mit einem Element aus Φ_B) beginnt und danach endlich viele Zeichen aus Φ besitzen darf.

Formale "induktive" Definition der Bezeichner: 1. Definition

(1) Jedes Element aus Φ_B ist ein Bezeichner.

(2) Wenn w ein Bezeicher ist und $a \in \Phi$, dann ist auch wa ein Bezeichner.

(3) Nur Zeichenfolgen, die ausschließlich mit den Regeln (1) und (2) aufgebaut wurden, sind Bezeichner.

Zum Beispiel ist Hz4 ein Bezeichner.

■

2. Definition

Die Menge der Bezeichner Bez lässt sich auch *direkt* angeben:

$\underline{\text{Bez}} = \{w \in \Phi^* \mid w = bv,\ b \in \Phi_B \text{ und } v \in \Phi^*\} = \Phi_B \Phi^*.$ ■

Man kann die Menge der Bezeichner auch erzeugen lassen:

Es sei β ein neues Zeichen. Betrachte folgende *Regeln:*

$\beta \to \eta$ für jedes $\eta \in \Phi_B$ (dies sind also 52 Regeln),

$\beta \to \beta\lambda$ für jedes $\lambda \in \Phi$ (dies sind also 63 Regeln).

Eine Herleitung (oder "Ableitung") beginnt mit dem Zeichen β In jedem Schritt darf man ein Zeichen, das links von "$\to$" in einer Regel steht, durch die davon rechts stehende Zeichenfolge ersetzen. (In unserem Fall haben nur das eine Zeichen β, das ersetzt werden darf.) Man leitet solange ab, bis das Zeichen β nicht mehr auftritt.

Bez ist dann die Menge der Zeichenfolgen, die man aus β in endlich vielen Schritten herleiten (oder "ableiten") kann.

Zum Beispiel kann man $Hz4$ wie folgt ableiten:

Beginne mit β und wende die Regel $\beta \rightarrow \beta\lambda$ speziell für $\lambda = 4$ an:

$\beta \Rightarrow \beta 4$ Wende die Regel erneut für $\lambda = z$ an:

$\beta \Rightarrow \beta 4 \Rightarrow \beta z 4$ Wende nun die Regel $\beta \rightarrow \eta$ mit $\eta = H$ an:

$\beta \Rightarrow \beta 4 \Rightarrow \beta z 4 \Rightarrow Hz4$

Da man $Hz4$ auf diese Weise ableiten konnte, ist diese Zeichenfolge also ein Bezeichner.

Allgemein definieren wir:

3. Definition

Bez = $\{w \in \Phi^* \mid$ Es gibt eine Ableitung von β nach $w\}$.

■

$\beta \Rightarrow^* w$

Mit folgenden Regeln kann man ganze Zahlen herleiten:

$\gamma \rightarrow \gamma\alpha$ für jedes $\alpha \in \Phi_Z$, also $\gamma \rightarrow \gamma 0$, $\gamma \rightarrow \gamma 1$, ..., $\gamma \rightarrow \gamma 9$.

Dies liefert Ziffernfolgen. Hier gibt es führende Nullen, die wir nicht zulassen möchten. Ganze Zahlen erzeugen wir daher durch:

$\zeta \rightarrow 0$,

$\zeta \rightarrow +\gamma$, $\zeta \rightarrow \gamma$, $\zeta \rightarrow -\gamma$,

$\gamma \rightarrow \gamma\alpha$ für jedes $\alpha \in \Phi_Z$,

$\gamma \rightarrow \delta$ für jedes $\delta \in \Phi_Z - \{0\}$, also $\gamma \rightarrow 1$, ..., $\gamma \rightarrow 9$.

Überprüfen Sie, dass 0, +1, -13, 3004 usw. herleitbar sind, die drei Wörter -0, 012, +084 dagegen nicht.

Genauso kann man korrekt geklammerte Ausdrücke herleiten:

$S \to (S),\ \ S \to S+S,\ \ S \to S-S,$

$S \to S*S,\ \ S \to S\ \underline{\text{div}}\ S,\ \ S \to S\ \underline{\text{mod}}\ S,$

$S \to \zeta,\ \ S \to \beta,$

$\zeta \to 0,$

$\zeta \to +\gamma,\ \ \zeta \to \gamma,\ \ \zeta \to -\gamma,$

$\gamma \to \gamma\alpha \quad$ für jedes $\alpha \in \Phi_Z$,

$\gamma \to \delta \quad$ für jedes $\delta \in \Phi_Z - \{0\},$

Ableitung von Zahlen

$\beta \to \beta\lambda \quad$ für jedes $\lambda \in \Phi,$

$\beta \to \eta \quad$ für jedes $\eta \in \Phi_B$.

Ableitung von Bezeichnern

Zum Beispiel leiten wir X1 + (-3*X2) ab (wir schreiben bei Regeln das Zeichen $\rightarrow$ und bei Ableitungen das Zeichen $\Rightarrow$):

$$S \Rightarrow S + S \Rightarrow S + (S) \Rightarrow S + (S*S)$$
$$\Rightarrow \beta + (S*S) \Rightarrow \beta + (\zeta*S) \Rightarrow \beta + (\zeta*\beta)$$
$$\Rightarrow \beta 1 + (\zeta*\beta) \Rightarrow X1 + (\zeta*\beta)$$
$$\Rightarrow X1 + (-\gamma*\beta) \Rightarrow X1 + (-3*\beta)$$
$$\Rightarrow X1 + (-3*\beta 2) \Rightarrow X1 + (-3*X2)$$

Aus S lassen sich genau die korrekt geklammerten Ausdrücke mit Zahlen und Bezeichnern als Operanden herleiten.
(Warum? Wie kann man das beweisen? Wie leitet man genau ab? Wie lassen sich alle abgeleiteten Wörter beschreiben? ...)

3.2 Sprachen und ihre Operationen

Auf die Regeln und die Herleitung von Zeichenfolgen aus speziellen Zeichen, die im Laufe einer Herleitung wieder verschwinden müssen, gehen wir im nächsten Abschnitt ein.

Wir halten fest: Die uns interessierenden Mengen ("Sprachen") bestehen aus Zeichenfolgen, die über einer festen Menge (z.B. Φ oder $\mathbf{A} \cup$ SW) gebildet werden. Wir definieren daher:

Definition: Eine endliche Menge $\Sigma = \{a_1, a_2, ..., a_n\}$, deren Elemente linear angeordnet sind ($a_1 < a_2 < ... < a_n$), heißt ein (endliches) Alphabet.
Σ^* sei die Menge der Wörter (=Zeichenfolgen) über Σ.

Jede Menge von Zeichenfolgen über Σ bezeichnen wir als Sprache über dem Alphabet Σ (engl.: "language").
D.h.: L ist eine Sprache über Σ genau dann, wenn $L \subseteq \Sigma^*$.

Operationen auf Sprachen: Obige Definition hat den Vorteil, dass wir alle Operationen, die wir für Mengen und für Wortmengen kennen, auch für Sprachen nutzen können. Zum Beispiel:

Vereinigung von Sprachen: Wenn L_1 und L_2 Sprachen über dem gleichen Alphabet Σ sind, dann ist auch $L_1 \cup L_2$ Sprache über Σ.

Durchschnitt von Sprachen: Wenn L_1 und L_2 Sprachen über dem gleichen Alphabet Σ sind, dann ist auch $L_1 \cap L_2$ Sprache über Σ.

Komplement von Sprachen: Wenn L eine Sprache über Σ ist, dann ist auch das Komplement $\Sigma^* \backslash L = \overline{L}$ eine Sprache über Σ.

Konkatenation ("Verkettung") von Sprachen: Sind L_1 und L_2 Sprachen über Σ, dann ist auch ihre Konkatenation
$L_1 \circ L_2 = \{u\,v \mid u \in L_1, v \in L_2\}$ eine Sprache über Σ.
Man schreibt wie bei Wörtern kurz L_1L_2 anstelle von $L_1 \circ L_2$.

Operationen (Fortsetzung)

Iteration von Sprachen: Wenn L eine Sprache über Σ ist, dann ist auch $L \circ L = LL$, also die Konkatenation von L mit sich, eine Sprache über Σ. Seien $L^0 = \{\varepsilon\}$ und L^i die i-fache Konkatenation von L mit sich, dann sei L^* die Vereinigung aller dieser Mengen L^i für $i \geq 0$ (bzw. L^+ für $i > 0$). *Formale Definition:*

$L^0 = \{\varepsilon\}$

Beachte: $L\{\varepsilon\} = L$, also $L\,L^0 = L^1 = L$.

$L^{i+1} = L\,L^i$ für alle $i \geq 0$

$$L^* = \bigcup_{i \geq 0} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cup L^4 \ldots.$$

Die Iterierte von L
oder
das von L erzeugte Untermonoid in Σ^*

$$L^+ = \bigcup_{i \geq 1} L^i = L^1 \cup L^2 \cup L^3 \cup L^4 \ldots.$$

Die von L erzeugte Unterhalbgruppe in Σ^*

Es gilt: $L^* = L^+ \cup \{\varepsilon\}$.

Beispiele: Sei $\Sigma = \{0, 1\}$ das zugrunde liegende Alphabet.

Sei $K = \{0\}$, dann ist $K^2 = KK = \{00\}$, $K^3 = \{000\}$ usw.,

$K^* = K^0 \cup K^1 \cup K^2 \cup K^3 \cup \ldots = \{\varepsilon, 0, 00, 000, \ldots\}$

$= \{0^i \mid i \geq 0\}$ und

$K^+ = \{0, 00, 000, \ldots\} = \{0^i \mid i > 0\} \subset \Sigma^*$.

Sei $L = \{\varepsilon, 0, 01\}$, dann ist $L^2 = \{\varepsilon, 0, 00, 01, 001, 010, 0101\}$,

$L^3 = \{\varepsilon, 0, 00, 01, 000, 001, 010, 0001, 0010, 0100, 0101,$
$00101, 01001, 01010, 010101\}$ usw.,

$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots =$?

Das lässt sich nicht mehr einfach hinschreiben! Jedes Wort aus L* muss man als Folge von Wörtern aus L darstellen können.

Gehört zum Beispiel 0100101000010 zu L*?

Ja, wegen der Zerlegung 01 0 01 01 0 0 0 01 0 $\in$ L*

Beispiel: Alphabet $\Sigma = \{0, 1, 2\}$. Sei L die Menge aller Wörter über Σ, in denen kein von 0 verschiedenes Zeichen vor einer 0 und kein von 2 verschiedenes Zeichen nach einer 2 im Wort stehen darf.

Skizze, wie solche Wörter aussehen:

$a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_9\ a_{10}\ a_{11}\ a_{12}\ a_{13}\ a_{14}\ a_{15}\ a_{16}\ a_{17}$ ..... a_n =

.....0.....2.....

Hier dürfen nur Nullen stehen

Hier dürfen nur Zweien stehen

Beispiel: Alphabet $\Sigma = \{0, 1, 2\}$. Sei L die Menge aller Wörter über Σ, in denen kein von 0 verschiedenes Zeichen vor einer 0 und kein von 2 verschiedenes Zeichen nach einer 2 im Wort stehen darf.

Skizze, wie solche Wörter aussehen:

$a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_9\ a_{10}\ a_{11}\ a_{12}\ a_{13}\ a_{14}\ a_{15}\ a_{16}\ a_{17}$ ..... a_n =

.....0.....0.....2.....

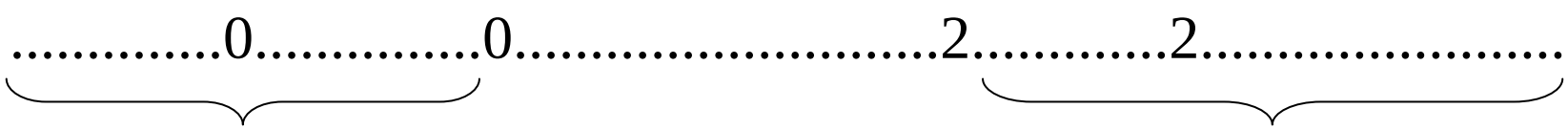
Hier dürfen nur Nullen stehen

Hier dürfen nur Zweien stehen

Beispiel: Alphabet $\Sigma = \{0, 1, 2\}$. Sei L die Menge aller Wörter über Σ, in denen kein von 0 verschiedenes Zeichen vor einer 0 und kein von 2 verschiedenes Zeichen nach einer 2 im Wort stehen darf.

Skizze, wie solche Wörter aussehen:

$a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_9\ a_{10}\ a_{11}\ a_{12}\ a_{13}\ a_{14}\ a_{15}\ a_{16}\ a_{17}$ ..... a_n =



Hier dürfen nur Nullen stehen

Hier dürfen nur Zweien stehen

Folglich können diese Wörter nur die Form
000...00111....11222.....2 = $0^i1^j2^k$ mit i, j, k ≥ 0 besitzen.

Beispiel: Alphabet $\Sigma = \{0, 1, 2\}$. Sei L die Menge aller Wörter über Σ, in denen kein von 0 verschiedenes Zeichen vor einer 0 und kein von 2 verschiedenes Zeichen nach einer 2 im Wort stehen darf.

Es gilt also $L = \{0^i 1^j 2^k \mid i, j, k \geq 0\} \subset \Sigma^*$.

Mit Hilfe unserer Operationen kann man L auch wie folgt beschreiben:

$L = \{0\}^* \{1\}^* \{2\}^*$.

Wie sehen Regeln zur Ableitung der Wörter, die zu L gehören, aus?

Dies ist einfach. Finden Sie die Lösung selbst!
Wir geben eine Lösung auf der folgenden Folie an.

Erzeugt werden soll:

$L = \{0^i 1^j 2^k \mid i, j, k \geq 0\} \subset \Sigma^*$

d.h.: $L = \{0\}^* \{1\}^* \{2\}^*$

Alphabet $\Sigma = \{0, 1, 2\}$

Wir verwenden vier neue Zeichen N, E, Z und S (für Nullen, Einsen, Zweien und Start). Die Regeln lauten (ε ist das leere Wort):

$S \rightarrow NEZ$

$N \rightarrow \varepsilon \qquad N \rightarrow 0N$

$E \rightarrow \varepsilon \qquad E \rightarrow 1E$

$Z \rightarrow \varepsilon \qquad Z \rightarrow 2Z$

Beispiel für die Ableitung des Wortes 0112:

$S \Rightarrow NEZ \Rightarrow 0NEZ \Rightarrow 0EZ \Rightarrow 01EZ \Rightarrow 011EZ$
$\Rightarrow 011Z \Rightarrow 0112Z \Rightarrow 0112$

Erzeugt werden soll:

$L = \{0^i 1^j 2^k \mid i, j, k \geq 0\} \subset \Sigma^*$

d.h.: $L = \{0\}^* \{1\}^* \{2\}^*$

Alphabet $\Sigma = \{0, 1, 2\}$

Wir hätten auch folgende Regeln verwenden können, um genau die gleiche Menge L abzuleiten (N entfällt hierbei, da S jetzt dessen Rolle übernimmt):

$S \rightarrow 0S$ $\quad S \rightarrow E$ $\quad E \rightarrow 1E$

$E \rightarrow Z$ $\quad Z \rightarrow 2Z$ $\quad Z \rightarrow \varepsilon$

Beispiel für die Ableitung des Wortes 0112 mit diesen Regeln:

$S \Rightarrow 0S \Rightarrow 0E \Rightarrow 01E \Rightarrow 011E \Rightarrow 011Z \Rightarrow 0112Z \Rightarrow 0112$

Wir erzeugen mit diesen Regeln also wiederum die Sprache $L = \{0^i 1^j 2^k \mid i, j, k \geq 0\} \subset \Sigma^*$.

Möchte man zusätzlich, dass die Zahl der Nullen und der Zweien gleich sein soll, so muss man eine andere Sprache über dem Alphabet $\Sigma = \{0, 1, 2\}$ definieren, nämlich die folgende Sprache

$L' = \{0^i 1^j 2^k \mid i, j, k \geq 0 \text{ mit } i = k\} \subset \Sigma^*$.

Aufgabe: Versuchen Sie, Regeln für die Ableitung genau der Wörter aus L' zu finden.
(Eine Lösung steht auf der nächsten Folie, aber zunächst selbst probieren!)

Lösung:
Man muss die Nullen und Zweien gleichzeitig erzeugen. Hierzu eignet sich eine Regel der Form $S \rightarrow 0S2$. Es gilt:

$$S \Rightarrow 0S2 \Rightarrow 00S22 \Rightarrow 000S222 \Rightarrow 0000S2222 \Rightarrow ... \Rightarrow 0^iS2^i$$

für jede natürliche Zahl i. Hat man genügend viele Nullen und Einsen erzeugt, dann ersetzt man S durch E und erzeugt hiermit die gewünschten Einsen. Die Regeln zur Erzeugung von L' lauten also:

$S \rightarrow 0S2$ $\quad S \rightarrow E$ $\quad E \rightarrow 1E$ $\quad E \rightarrow \varepsilon$

3.3 Kontextfreie Grammatiken

Definition: Eine kontextfreie Grammatik ist ein Viertupel $G = (V, \Sigma, P, S)$ mit

(1) V ist eine nicht-leere endliche Menge (die Menge der Nichtterminalzeichen oder Variablen),

(2) Σ ist eine nicht-leere endliche Menge (die Menge der Terminalzeichen) mit $V \cap \Sigma = \emptyset$,

(3) $S \in V$ ist ein Nichtterminalzeichen (das Startsymbol),

(4) $P \subset V \times (V \cup \Sigma)^*$ ist eine endliche Menge (die Menge der Regeln oder Produktionen).

(Englisch: contextfree grammar.)

Wir wiederholen nun einiges, was durch die bisherigen Beispiele schon klar ist. Vor allem definieren wir exakt die Begriffe "Ableitung" und "erzeugte Sprache".

Beispiel: Betrachte die Grammatik $G_1 = (V_1, \Sigma_1, P_1, S_1)$ mit $V_1 = \{S_1\}$, $\Sigma_1 = \{0, 1\}$, $P_1 = \{(S_1, 1), (S_1, S_1 0), (S_1, S_1 1)\}$.

Die Idee ist, ausgehend von dem Startsymbol S_1 schrittweise alle Nichtterminalzeichen (hier ist dies nur das Zeichen S_1) durch ihre rechten Seiten in P_1 zu ersetzen. Alle Wörter, die man auf diese Weise erhält und die kein Nichtterminalzeichen mehr enthalten, bilden die Sprache, die von dieser Grammatik erzeugt wird. Bei der Ersetzung ("Ableitung" oder "Herleitung") hat man Freiheiten, da man sich willkürlich für eine Regel entscheiden kann, sofern P_1 mehrere Regeln mit der gleichen linken Seite besitzt.

Betrachte unser Beispiel: Da S_1 die linke Seite von drei Regeln ist, hat man in jedem Schritt mehrere Auswahlmöglichkeiten.

Grammatik $G_1 = (V_1, \Sigma_1, P_1, S_1)$ mit

$V_1 = \{S_1\}$, $\Sigma_1 = \{0, 1\}$, $P_1 = \{(S_1, 1), (S_1, S_1 0), (S_1, S_1 1)\}$.

S_1 ersetzen durch 1 (fertig, denn das Wort 1 enthält kein Nichtterminalzeichen mehr). Andere Möglichkeiten:

S_1 ersetzen durch $S_1 0$, hierin S_1 wieder ersetzen durch $S_1 0$, so dass $S_1 00$ entsteht; hierin S_1 ersetzen durch 1; Ergebnis ist das Wort 100 (fertig, da 100 kein Nichtterminalzeichen enthält).

S_1 ersetzen durch $S_1 1$, hierin S_1 ersetzen durch $S_1 0$, so dass $S_1 01$ entsteht; hierin S_1 ersetzen durch $S_1 1$, wodurch $S_1 101$ entsteht; hierin S_1 ersetzen durch 1; Ergebnis ist das Wort 1101 (fertig, da dieses Wort kein Nichtterminalzeichen mehr enthält).

Welche Wörter entstehen auf diese Weise aus S_1?
(Schreiben Sie diese Menge hin.) ■

Um die Ersetzung der linken Seite A durch die rechte Seite w eines Paares $(A,w) \in P$ deutlicher hervor zu heben, schreibt man statt (A,w) im Allgemeinen $A \rightarrow w$.

$P_1 = \{(S_1, 1), (S_1, S_1 0), (S_1, S_1 1)\}$ schreibt man also folgendermaßen:

$$P_1 = \{S_1 \rightarrow 1,\ S_1 \rightarrow S_1 0,\ S_1 \rightarrow S_1 1\}.$$

Wir müssen nun den Ableitungsprozess genau definieren.
Ein Wort xAy darf man durch das Wort xwy in einem Schritt ersetzen, sofern $A \rightarrow w$ eine Regel ist. Wiederholt man dies endlich oft (auch "keinmal"), so erhält man eine Ableitung.

Definition: Gegeben sei eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$. Die Regelmenge P definiert auf der Menge $(V \cup \Sigma)^*$ die "Ableitungsrelationen" $\Rightarrow$ und $\Rightarrow^*$:

(1) Es gilt $u \Rightarrow v$ genau dann, wenn man die Wörter u und v in der Form $u = xAy$, $v = xwy$ mit $x, y \in (V \cup \Sigma)^*$ und $(A,w) \in P$ schreiben kann. Man sagt: v ist aus u in einem Schritt ableitbar oder v lässt sich aus u in einem Schritt ableiten.

(2) Es gilt $u \Rightarrow^* v$ genau dann, wenn entweder $u = v$ ist oder wenn es Wörter $z_0, z_1, ..., z_k \in (V \cup \Sigma)^*$ für ein $k \geq 1$ gibt mit $u = z_0$, $v = z_k$, $z_i \Rightarrow z_{i+1}$ für alle $i = 0, 1, ..., k-1$. Man sagt dann, v ist aus u herleitbar oder ableitbar. (Die Zahl k heißt Länge der Ableitung.)

Hinweis: $\Rightarrow^*$ ist der so genannte "reflexive und transitive Abschluss" von $\Rightarrow$. (Englisch: Ableitung = derivation.)

Definition:

Die von einer kontextfreien Grammatik $G = (V, \Sigma, P, S)$ erzeugte Sprache (engl.: generated language) ist die Menge $L(G) = \{ w \in \Sigma^* \mid S \Rightarrow^* w \} \subseteq \Sigma^*$.

Eine Sprache $L \subseteq \Sigma^*$ heißt kontextfreie Sprache, wenn es eine kontextfreie Grammatik G mit $L = L(G)$ gibt.

Die Nichtterminalzeichen dienen also dazu, dass der Ableitungsprozess durchgeführt werden kann; zur erzeugten Sprache zählen dagegen nur die Wörter, die ausschließlich aus Terminalzeichen bestehen.

Wir werden nun zeigen, wie man Programmiersprachen mit Hilfe von kontextfreien Grammatiken erzeugt. Hier betrachten wir zunächst "Bezeichner" und "Ausdrücke".

Beispiel: Betrachte erneut die Grammatik

$G_1 = (V_1, \Sigma_1, P_1, S_1)$ mit $V_1 = \{S_1\}$, $\Sigma_1 = \{0, 1\}$ und

$P_1 = \{S_1 \to 1,\ S_1 \to S_1 0,\ S_1 \to S_1 1\}$.

Aus dem Startsymbol S_1 kann man mit der zweiten und der dritten Regel $S_1 0$ und $S_1 1$ ableiten, hieraus mit den gleichen Regeln $S_1 00$, $S_1 10$, $S_1 01$ und $S_1 11$, hieraus mit den gleichen Regeln $S_1 000$, $S_1 100$, $S_1 010$, $S_1 110$, $S_1 001$, $S_1 101$, $S_1 011$ und $S_1 111$ usw., also alle Wörter der Form $S_1 x$ für ein beliebiges Wort $x \in \Sigma^*$. Um das Nichtterminalzeichen zu entfernen, muss irgendwann die erste Regel verwendet werden, welche hieraus das Wort $1x$ für ein beliebiges Wort $x \in \Sigma^*$ herleitet.

Da andere Wörter nicht herleitbar sind, gilt:

$L(G_1) = \{1x \mid x \in \Sigma^*\} = \{1\}\Sigma^*$. ■

Beispiel: Nochmals: Die Menge der Bezeichner.

Erinnerung: Seien $\Phi_Z = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$,

$\Phi_B = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U,$
$V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u,$
$v, w, x, y, z\}$,

$\Phi = \Phi_B \cup \Phi_Z \cup \{ \_ \}$, dann ist die Menge der Bezeichner

$\underline{Bez} = \{w \in \Phi^* \mid w = bv, b \in \Phi_B \text{ und } v \in \Phi^*\} = \Phi_B \Phi^*$.

Definiere die kontextfreie Grammatik $G_2 = (V_2, \Sigma_2, P_2, S_2)$ mit
$V_2 = \{S_2, S_2', S_2''\}$ und $\Sigma_2 = \Phi$.
P_2 bestehe aus folgenden 66 Regeln:

P_2 =

$\{ S_2' \to A, S_2' \to B, S_2' \to C, S_2' \to D, S_2' \to E, S_2' \to F, S_2' \to G,$
$S_2' \to H, S_2' \to I, S_2' \to J, S_2' \to K, S_2' \to L, S_2' \to M, S_2' \to N,$
$S_2' \to O, S_2' \to P, S_2' \to Q, S_2' \to R, S_2' \to S, S_2' \to T, S_2' \to U,$
$S_2' \to V, S_2' \to W, S_2' \to X, S_2' \to Y, S_2' \to Z, S_2' \to a, S_2' \to b,$
$S_2' \to c, S_2' \to d, S_2' \to e, S_2' \to f, S_2' \to g, S_2' \to h, S_2' \to i,$
$S_2' \to j, S_2' \to k, S_2' \to l, S_2' \to m, S_2' \to n, S_2' \to o, S_2' \to p,$
$S_2' \to q, S_2' \to r, S_2' \to s, S_2' \to t, S_2' \to u, S_2' \to v, S_2' \to w,$
$S_2' \to x, S_2' \to y, S_2' \to z,$
$S_2'' \to 0, S_2'' \to 1, S_2'' \to 2, S_2'' \to 3, S_2'' \to 4, S_2'' \to 5,$
$S_2'' \to 6, S_2'' \to 7, S_2'' \to 8, S_2'' \to 9, S_2'' \to \_$
$S_2 \to S_2', \quad S_2 \to S_2S_2', \quad S_2 \to S_2S_2'' \}$

Es gilt dann:

Aus S_2' sind in einem Schritt genau alle Buchstaben ableitbar, d.h.: $S_2' \Rightarrow y$ genau dann, wenn $y \in \Phi_B$.
Aus S_2'' sind in einem Schritt genau alle Ziffern und der Unterstrich ableitbar, d.h.: $S_2'' \Rightarrow y$ genau dann, wenn $y \in \Phi_Z \cup \{\_\}$.

Mit den letzten beiden Regeln sind aus S_2 genau alle Wörter der Form $S_2 x$ mit $x \in \{S_2', S_2''\}^*$ ableitbar.

Aus S_2 sind mit den letzten drei Regeln in mindestens einem Schritt genau alle Wörter der Form $S_2' x$ mit $x \in \{S_2', S_2''\}^*$ ableitbar.

Da aus S_2' nur Buchstaben und aus S_2'' nur die Ziffern und der Unterstrich ableitbar sind, so sind folglich aus S_2 genau alle Wörter der Form zx mit $z \in \Phi_B$ und $x \in \Phi^*$ ableitbar, d.h.:
$L(G_2) = \{zx \mid z \in \Phi_B \text{ und } x \in \Phi^*\} = \Phi_B \Phi^* =$ Bez. ■

In der Praxis geht man folgendermaßen vor (siehe 3.5 und 3.6):

Nichtterminalzeichen schreibt man in der Form
<...> mit einem aussagekräftigen Namen zwischn < und >.
Terminalzeichen sind die Zeichen, aus denen Programme bestehen.

Regeln: Statt $A \rightarrow w$ schreibt man meist A ::= w.
Gleiche linke Seiten fasst man zusammen, getrennt durch "|".
Beispiel:
<Farbe> ::= weiß , <Farbe> ::= rot , <Farbe> ::= gelb ,
<Farbe> ::= grün , <Farbe> ::= blau , <Farbe> ::= schwarz
ersetzt man durch
<Farbe> ::= weiß | rot | gelb | grün | blau | schwarz .

Beispiel: Grammatik für Bezeichner.

<Buchstabe> ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
p | q | r | s | t | u | v | w | x | y | z

<Ziffer> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Zeichen für Bezeichner> ::= _ | <Buchstabe> | <Ziffer>

<Bezeichner> ::= <Buchstabe> |
<Bezeichner> <Zeichen für Bezeichner>

V = {<Bezeichner>, <Zeichen für Bezeichner>, <Buchstabe>, <Ziffer>}
Σ = {A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,
a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t u,v,w,x,y,z,0,1,2,3,4,5,6,7,8,9,_}
Das Startsymbol ist hier <Bezeichner>.
Regelmenge: siehe oben.

Grammatik für die ganzen Zahlen:

<Ziffer_ohne_Null> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<positive_Zahl> ::= <Ziffer_ohne_Null> |
<positive_Zahl> <Ziffer>

<Zahl> ::= 0 | + <positive Zahl> | - <positive_Zahl>

Die Grammatik lautet also:
V = {<Ziffer_ohne_Null>, <positive Zahl>, <Zahl>}
Σ = {0,1,2,3,4,5,6,7,8,9,+,-}
Das Startsymbol ist hier <Zahl>.
Regelmenge: siehe oben.

Aufbau der Anweisungen (vgl. (A1) bis (A9))

<Anweisung> ::= skip | <Wertzuweisung> | <Alternative> |
<while-Schleife> | <for-Schleife> |
<Anweisung> ; <Anweisung>

<Wertzuweisung> ::= <Variable> := <Ausdruck>

<Variable> ::= <Bezeichner>

<Ausdruck> ::= <arithmetischer Ausdruck> | <Boolescher Ausdruck>

<Alternative> ::=
if <Boolescher Ausdruck> then <Anweisung> fi |
if <Boolescher Ausdruck> then <Anweisung> else <Anweisung> fi

<while-Schleife> ::= while <Boolescher Ausdruck> do <Anweisung> od

<for-Schleife> ::=
for <Laufvariable> := <arithmetischer Ausdruck> to
<arithmetischer Ausdruck> do <Anweisung> od

<Laufvariable> ::= <Bezeichner>

<arithmetischer Ausdruck> ::=

<Bezeichner> | <Zahl> |

(<arithmetischer Ausdruck>) |

<arithmetischer Ausdruck> + <arithmetischer Ausdruck> |

<arithmetischer Ausdruck> - <arithmetischer Ausdruck> |

<arithmetischer Ausdruck> * <arithmetischer Ausdruck> |

<arithmetischer Ausdruck> div <arithmetischer Ausdruck> |

<arithmetischer Ausdruck> mod <arithmetischer Ausdruck>

<Boolescher Ausdruck> ::=

<Bezeichner> | true | false |

(<Boolescher Ausdruck>) |

not <Boolescher Ausdruck> |

<Boolescher Ausdruck> or <Boolescher Ausdruck> |

<Boolescher Ausdruck> and <Boolescher Ausdruck> |

<Boolescher Ausdruck> exor <Boolescher Ausdruck> |

<Boolescher Ausdruck> impl <Boolescher Ausdruck> |

<arithmetischer Ausdruck> <VO> <arithmetischer Ausdruck>

<VO> ::= > | >= | = | < | <= | ≠

(Hinweis: VO steht hier für "Vergleichsoperator".)

<Programm> ::= <Kopf> is <Deklarationsteil> ; <Anweisungsteil>

<Kopf> ::= program <Bezeichner>

<Deklarationsteil > ::= <Deklaration> |
<Deklaration> ; <Deklarationsteil >

<Deklaration> ::= <Datentyp> <Variablenliste>

<Variablenliste> ::= <Variable> | <Variable>, <Variablenliste>

<Datentyp> ::= integer | Boolean

<Anweisungsteil> ::= begin <Anweisung> end

Nun fassen wir alles zusammen. Der besseren Unterscheidung wegen schreiben wir alle Terminalzeichen in blauer Schrift.

Zusammenfassung: Grammatik für unsere einfache Programmiersprache:

V = {<Programm>, <Kopf>, <Deklarationsteil>, <Deklaration>,
<Anweisungsteil>, <Anweisung>, <Datentyp>, <Variable>,
<Variablenliste>, <Ausdruck>, <arithmetischer Ausdruck>,
<Boolescher Ausdruck>, <arithmetischer Operator>,
< Boolescher Operator>, Wertzuweisung>, <Alternative>,
<while-Schleife>, <for-Schleife>, <Laufvariable>, <Bezeichner>,
<Zeichen für Bezeichner>, <Buchstabe>, <Ziffer>,
<Ziffer_ohne_Null>, <positive_Zahl>, <Zahl> }

Σ = {A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,
a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,
0,1,2,3,4,5,6,7,8,9, _ , ; , : , , , =, <, >, + , - , * , (,), }

Regelmenge: siehe nächste Folie.

Das Startsymbol ist hier <Programm>.

(Schlüsselwörter wurden hier nicht gesondert aufgeführt;
sie werden wie Bezeichner mit Buchstaben des Alphabets formuliert.)

<Programm> ::= <Kopf> is <Deklarationsteil> **;** <Anweisungsteil>
<Kopf> ::= program <Bezeichner>
<Deklarationsteil > ::= <Deklaration> | <Deklaration> **;** <Deklarationsteil >
<Deklaration> ::= <Datentyp> <Variablenliste>
<Datentyp> ::= integer | Boolean
<Variablenliste> ::= <Variable> | <Variable>**,** <Variablenliste>
<Anweisungsteil> ::= begin <Anweisung> end
<Anweisung> ::= skip | <Wertzuweisung> | <Alternative> | <while-Schleife> |
<for-Schleife> | <Anweisung> **;** <Anweisung>
<Wertzuweisung> ::= <Variable> := <Ausdruck>
<Variable> ::= <Bezeichner>
<Ausdruck> ::= <arithmetischer Ausdruck> | <Boolescher Ausdruck>
<Alternative> ::= if <Boolescher Ausdruck> then <Anweisung> fi |
if <Boolescher Ausdruck> then <Anweisung> else <Anweisung> fi
<while-Schleife> ::= while <Boolescher Ausdruck> do <Anweisung> od
<for-Schleife> ::= for <Laufvariable> := <arithmetischer Ausdruck> to
<arithmetischer Ausdruck> do <Anweisung> od
<Laufvariable> ::= <Bezeichner>
<arithmetischer Ausdruck> ::= <Bezeichner> | <Zahl> | (<arithmetischer Ausdruck>) | - <arithmetischer Ausdruck> |
<arithmetischer Ausdruck> <arithmetischer Operator> <arithmetischer Ausdruck> |
<arithmetischer Operator> ::= + | - | * | div | mod
<Boolescher Ausdruck> ::= <Bezeichner> | true | false | (<Boolescher Ausdruck>) | not <Boolescher Ausdruck> |
<Boolescher Ausdruck> <Boolescher Operator> <Boolescher Ausdruck> |
<arithmetischer Ausdruck> <Vergleichsoperator> <arithmetischer Ausdruck>
<Boolescher Operator> ::= and | or | exor | impl
<Vergleichsoperator> ::= > | >= | = | < | <= | ≠
<Buchstabe> ::= A | B | C | D | E | F | G | H | I | J | K | L | M |N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
<Ziffer> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Ziffer_ohne_Null> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Zeichen für Bezeichner> ::= _ | <Buchstabe> | <Ziffer>
<Bezeichner> ::= <Buchstabe> | <Bezeichner> <Zeichen für Bezeichner>
<positive_Zahl> ::= <Ziffer_ohne_Null> | <positive_Zahl> <Ziffer>
<Zahl> ::= 0 | + <positive_Zahl> | - <positive_Zahl>

Beispiel zur Ableitungsvielfalt: Einfache arithmet. Ausdrücke

Definiere die kontextfreie Grammatik $G_3 = (V_3, \Sigma_3, P_3, S_3)$ mit
$V_3 = \{S_3\}$ und $\Sigma_3 = \{ \mathbf{(}, \mathbf{)}, \mathbf{a}, \mathbf{+}, \mathbf{-}, \mathbf{*} \}$.
P_3 bestehe aus folgenden vier Regeln:

$S_3 \rightarrow \mathbf{a}$, $S_3 \rightarrow \mathbf{(}S_3 \mathbf{+} S_3\mathbf{)}$, $S_3 \rightarrow \mathbf{(}S_3 \mathbf{-} S_3\mathbf{)}$, $S_3 \rightarrow \mathbf{(}S_3 \mathbf{*} S_3\mathbf{)}$

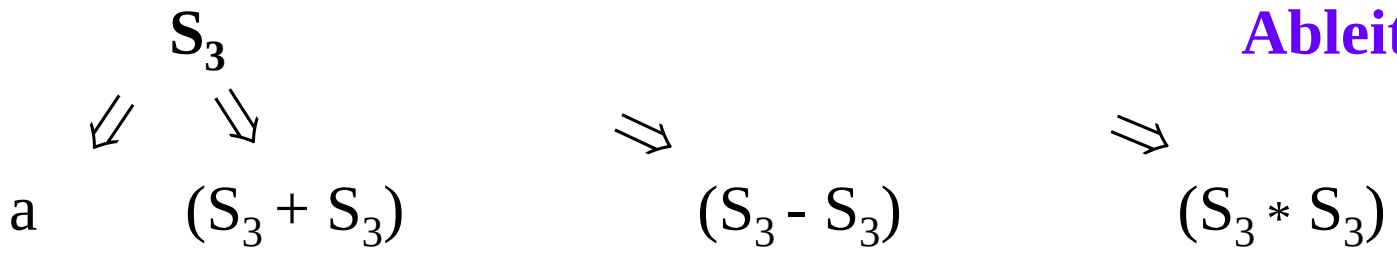
Die Sprache $L(G_3)$ bilden alle *vollständig geklammerten arithmetischen Ausdrücke, die nur den Operanden "***a***" enthalten*, z.B.:

$S_3 \Rightarrow \mathbf{a}$, $S_3 \Rightarrow \mathbf{(}S_3 + S_3\mathbf{)} \Rightarrow \mathbf{(a +} S_3\mathbf{)} \Rightarrow \mathbf{(a + a)}$,

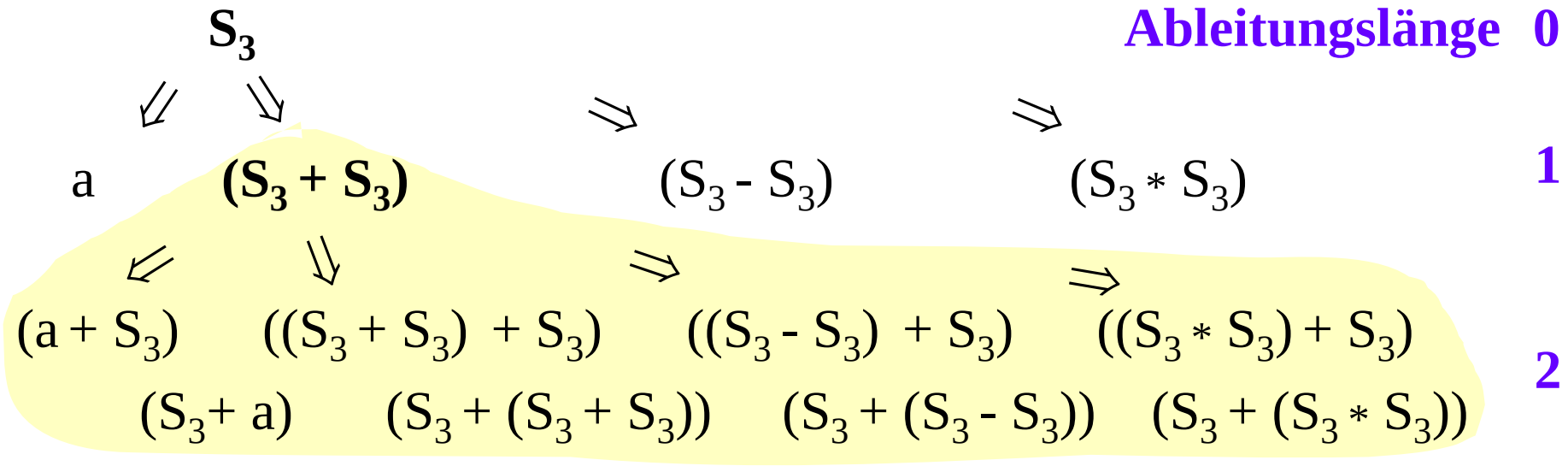
$S_3 \Rightarrow \mathbf{(}S_3 + S_3\mathbf{)} \Rightarrow \mathbf{(a +} S_3\mathbf{)} \Rightarrow \mathbf{(a +} \mathbf{(}S_3 * S_3\mathbf{))} \Rightarrow \mathbf{(a +} \mathbf{(}S_3 * \mathbf{a))}$

$\Rightarrow \mathbf{(a +} \mathbf{((}S_3 - S_3\mathbf{)} * \mathbf{a))} \Rightarrow \mathbf{(a +} \mathbf{((a -} S_3\mathbf{)} * \mathbf{a))} \Rightarrow \mathbf{(a +} \mathbf{((a - a)} * \mathbf{a))}$.

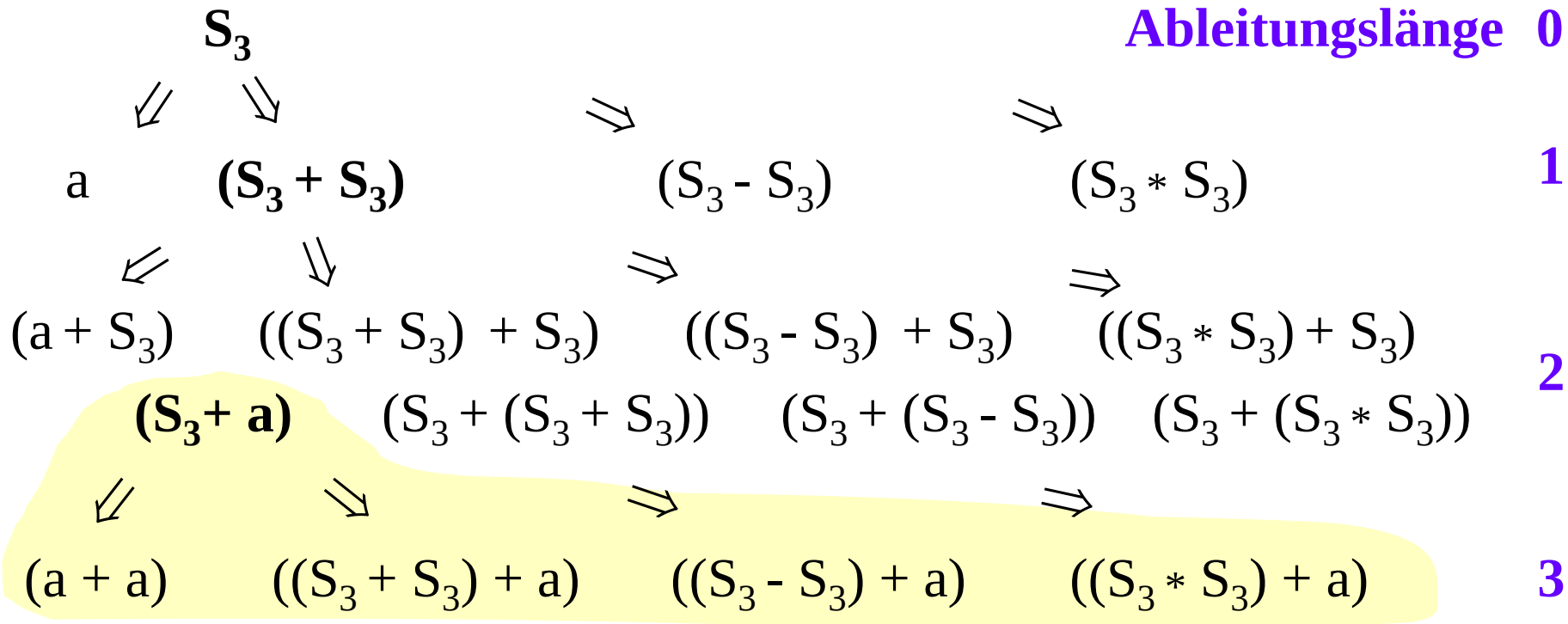
Leiten Sie weitere Wörter ab, z.B. $\mathbf{(((a - a)} * \mathbf{a) +} \mathbf{(a + a))}$, und überzeugen Sie sich über die in G_3 ableitbaren Wörter.



Regeln: $S_3 \rightarrow a$, $S_3 \rightarrow (S_3 + S_3)$, $S_3 \rightarrow (S_3 - S_3)$, $S_3 \rightarrow (S_3 * S_3)$



Regeln: $S_3 \rightarrow a$, $S_3 \rightarrow (S_3 + S_3)$, $S_3 \rightarrow (S_3 - S_3)$, $S_3 \rightarrow (S_3 * S_3)$



Überprüfen Sie:

4 verschiedene Wörter werden aus S_3 in genau einem Schritt abgleitet,

24 verschiedene Wörter werden aus S_3 in genau 2 Schritten abgleitet,

230 verschiedene Wörter werden aus S_3 in genau 3 Schritten abgleitet,

mehr als 1000 verschiedene Wörter werden aus S_3 in 4 Schritten abgeleitet.

Viele Wörter lassen sich auf verschiedene Art herleiten (z.B. ?).

Betrachten Sie zum Beispiel das Wort **(a + (a - a))**.

Ableitung 1:

$S_3 \Rightarrow \mathbf{(}S_3 + S_3\mathbf{)} \Rightarrow \mathbf{(a +} S_3\mathbf{)} \Rightarrow \mathbf{(a +} (S_3 - S_3)\mathbf{)} \Rightarrow \mathbf{(a +} (a - S_3)\mathbf{)}$

$\Rightarrow \mathbf{(a +} (a - a)\mathbf{)}$.

Ableitung 2:

$S_3 \Rightarrow \mathbf{(}S_3 + S_3\mathbf{)} \Rightarrow \mathbf{(}S_3 + (S_3 - S_3)\mathbf{)} \Rightarrow \mathbf{(}S_3 + (S_3 - a)\mathbf{)} \Rightarrow \mathbf{(}S_3 + (a - a)\mathbf{)}$

$\Rightarrow \mathbf{(a +} (a - a)\mathbf{)}$.

In Ableitung 1 wurde immer das am weitesten links im Wort stehende Nichtterminalzeichen ersetzt; in Ableitung 2 ist es das am weitesten rechts stehende. Bei Ableitungen hat man also Wahlfreiheiten. (Würde man aber die beiden Ableitungen wirklich als "wesentlich verschieden" auffassen? Wir werden dies präzisieren.)

■

Aus diesem Beispiel erkennen wir:

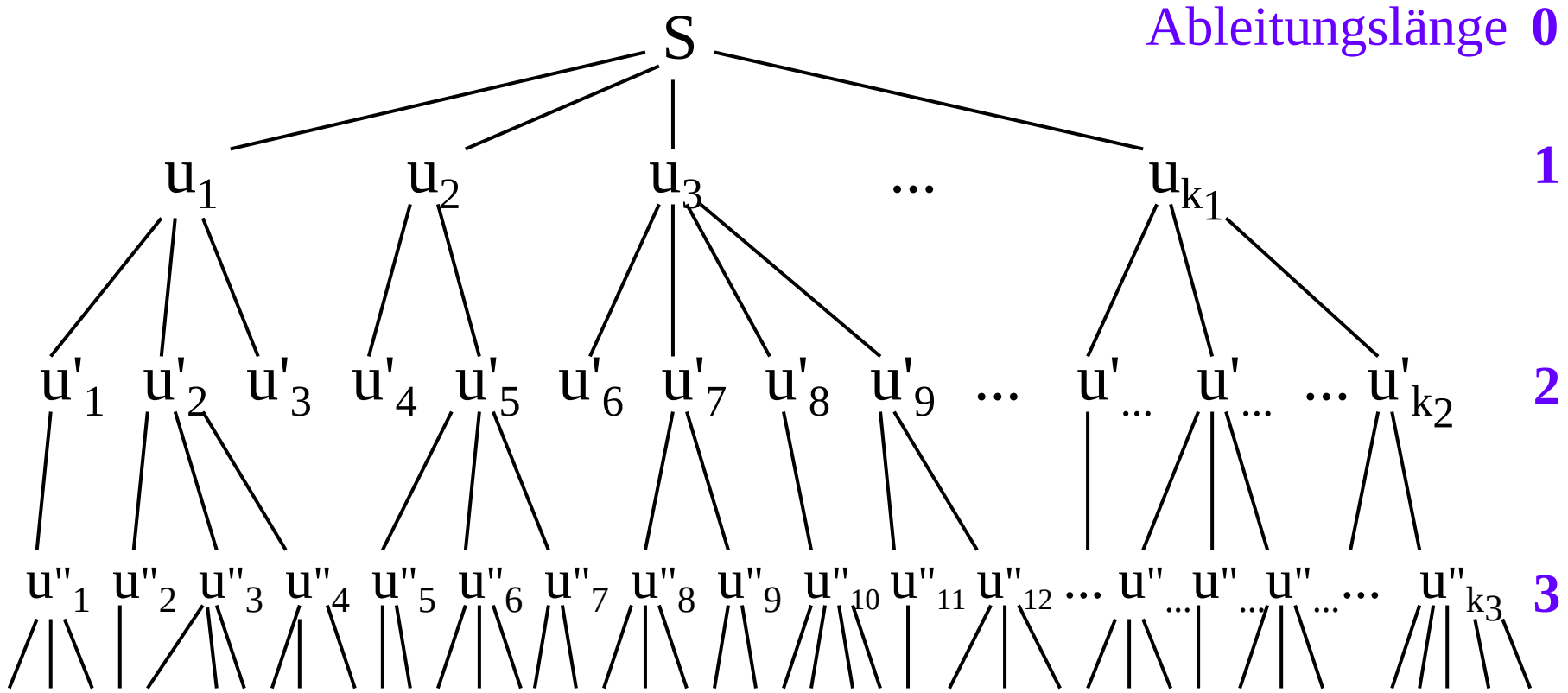
(1.) Es lassen sich alle Ableitungen aus dem Startsymbol S einer kontextfreien Grammatik entsprechend ihrer Länge grafisch gut als sog. "Baum" veranschaulichen.

(2.) Hierin ist jede einzelne Ableitung eines Wortes als ein Pfad enthalten, bei dem sich in jedem Schritt die Länge der Ableitung um eins erhöht. Zum Beispiel $S_3 \Rightarrow \mathbf{(}S_3 + S_3\mathbf{)} \Rightarrow \mathbf{(}S_3 + \mathbf{a)} \Rightarrow$ ... (mit der Ableitungslängen 0, 1, 2, ...). Auch jede einzelne Ableitung lässt sich als ein Baum veranschaulichen.

Beide Aspekte werden wir nun genauer betrachten und insbesondere die Darstellung als Baum herausarbeiten.

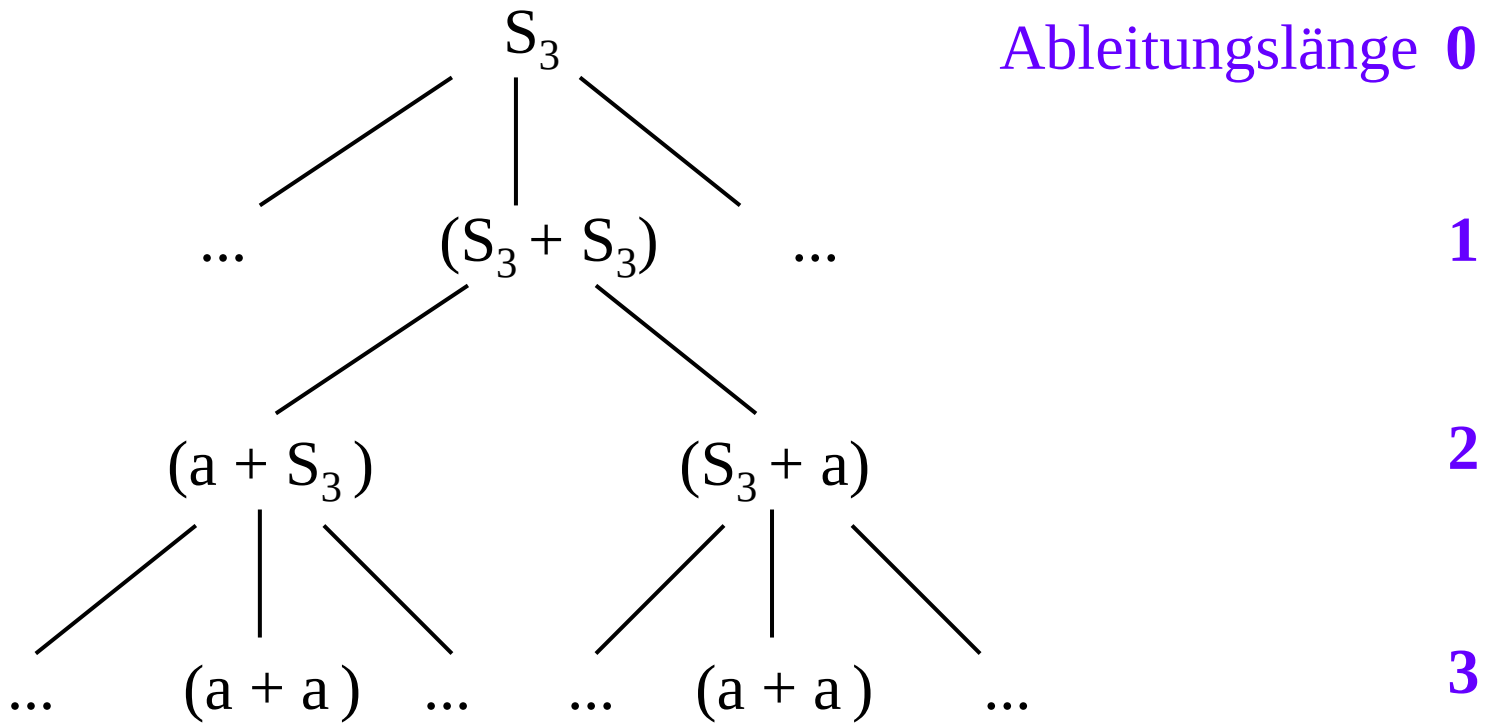
Aspekt (1.): Aus obigen Beispielen erkennen wir folgendes.

Alle Ableitungen aus dem Startsymbol S einer kontextfreien Grammatik, d.h., die Herleitung aller möglicher Wörter, lassen sich gleichzeitig in folgender Form schreiben:

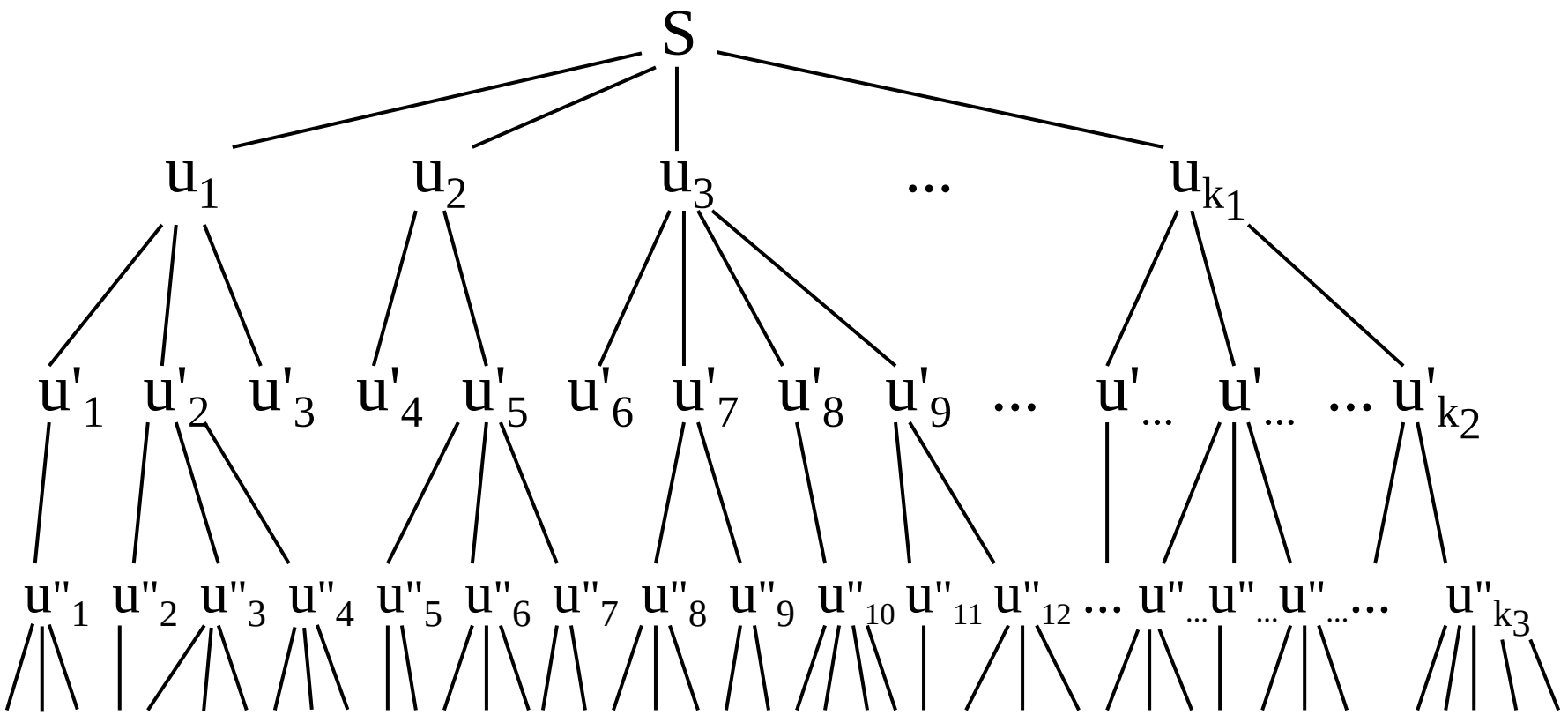


Die Menge der hergeleiteten Wörter, die nur Terminalzeichen enthalten, bilden hierbei die erzeugte Sprache $L(G_3)$.

In dieser Menge aller Ableitungen können Wörter mehrfach (sogar unendlich oft) vorkommen. Zum Beispiel in obiger Grammatik G_3 das Wort (a+a):

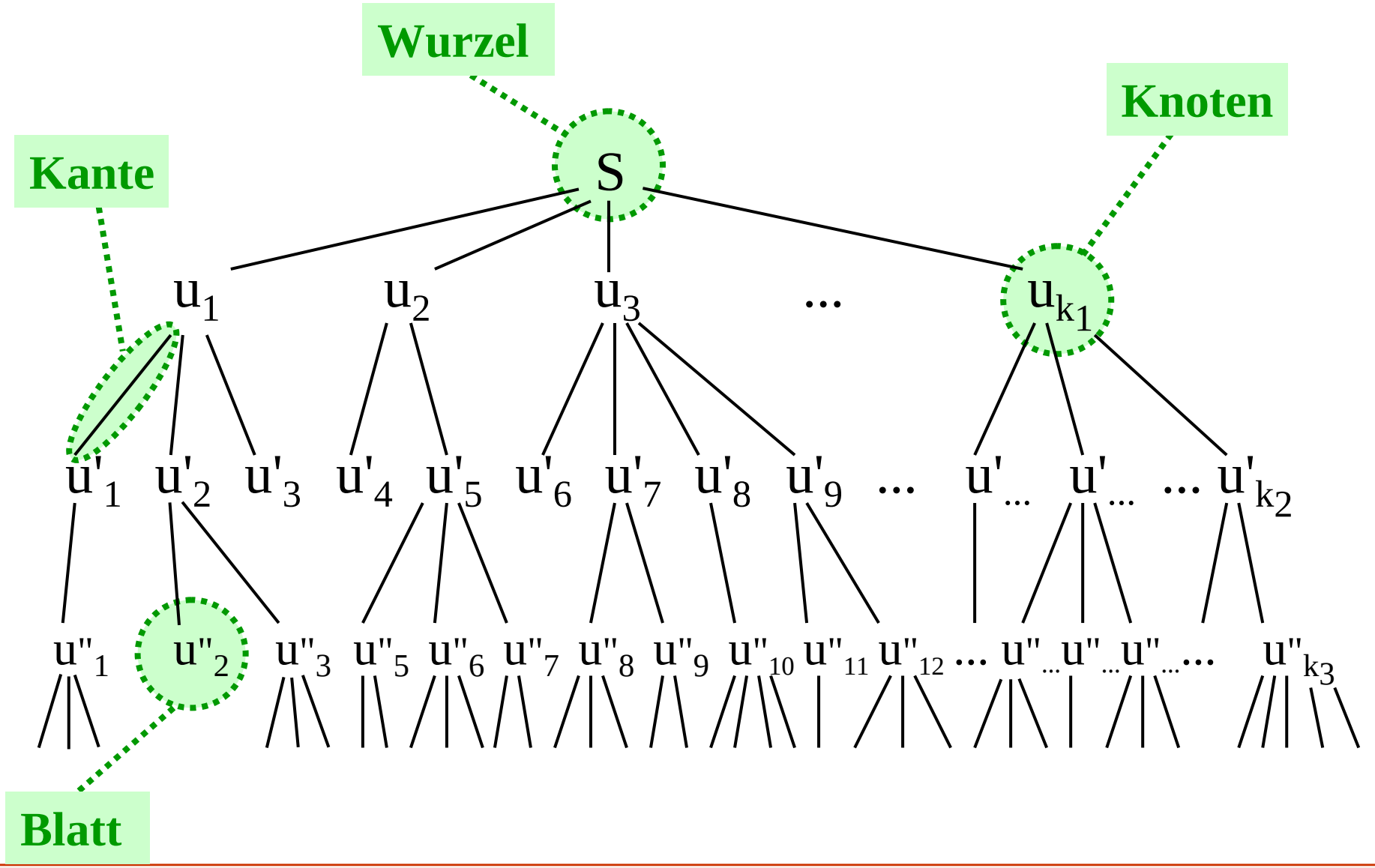


Bemerkung: Solch ein Gebilde, bei dem von jedem Punkt aus mehrere Linien abgehen, nirgends aber Linien zusammenlaufen können, nennt man einen Baum (engl.: tree). Das Gebilde sieht ja auch baumartig aus:



(Allerdings wächst dieser Baum nach unten statt nach oben.)

Einige Bezeichnungen bei Bäumen (anschaulich):



Bezeichnungen bei "Bäumen":

Bei den Linien spricht auch von Zweigen, meist aber von Kanten (engl.: edges);

eine Folge aneinander nach unten sich anschließender Kanten nennt man einen Pfad (oder einen Weg, engl.: path),

die Stellen, an denen die Wörter stehen, heißen Knoten (engl.: nodes); die Knoten können durch Kanten verbunden sein;

ein Knoten, von dem keine Kante mehr weiterführt, heißt Blatt (engl.: leaf);

der Knoten, von dem aus alle Knoten durch einen Pfad erreicht werden können, heißt die Wurzel des Baums (engl.: root).

Feststellung: Alle Ableitungen einer kontextfreien Grammatik zusammen bilden einen (i.A. unendlich großen) Baum.

In der Wurzel des Baumes steht das Startsymbol S.

Jeder Ableitung

$$u = z_0 \Rightarrow z_1 \Rightarrow z_2 \Rightarrow z_{3i} \Rightarrow \ldots \Rightarrow z_{k-1} \Rightarrow z_k = v$$

entspricht in diesem Baum ein Pfad (= eine Folge von Kanten) beginnend an einem Knoten, in dem u steht, und endend an einem Knoten, in dem v steht.

Knoten, in denen Wörter aus Terminalzeichen stehen, bilden stets Blätter; alle diese Wörter bilden die erzeugte Sprache L(G).

Die Ableitungen von Wörtern der erzeugten Sprache beginnen (als Pfad) stets in der Wurzel S und enden bei dem jeweiligen Wort in einem Blatt.

Aspekt (2.): Die Ableitung eines speziellen Wortes aus dem Startsymbol einer Grammatik haben wir wie folgt dargestellt (wir verwenden weiterhin die Grammatik G_3):

$S_3 \Rightarrow (S_3 + S_3) \Rightarrow (a + S_3) \Rightarrow \mathbf{(a + a)}$

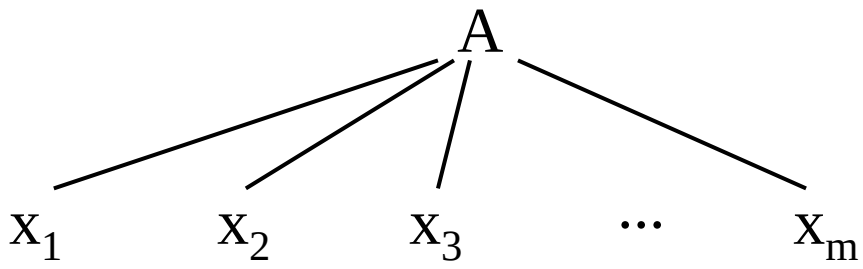
Diese Ableitung kann man umstellen, indem man zuerst das zweite S_3 durch a ersetzt und danach erst das erste S_3:

$S_3 \Rightarrow (S_3 + S_3) \Rightarrow (S_3 + a) \Rightarrow \mathbf{(a + a)}$

Dies ist im Wesentlichen *die gleiche Ableitung*.

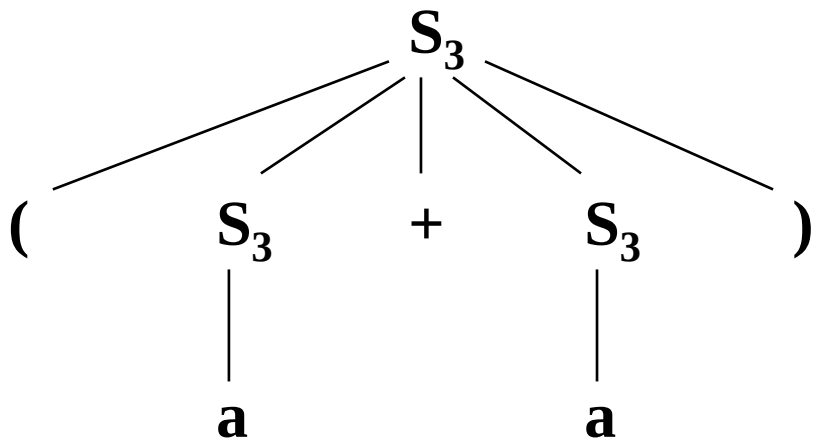
Um dies präzise zu definieren, schreiben wir nun auch die einzelnen Ableitungen baumartig auf, wobei in jedem Knoten genau ein Terminal- bzw. Nichtterminalzeichen steht.

Genauer: Eine Regel $A \rightarrow x_1 x_2 x_3 \ldots x_m$ mit $x_i \in (V \cup \Sigma)$ notieren wir in der Form



Die Regel $A \rightarrow \varepsilon$ (mit dem leeren Wort ε auf der rechten Seite) schreiben wir in der Form:

Die Ableitung $S_3 \Rightarrow (S_3 + S_3) \Rightarrow (S_3 + a) \Rightarrow (a + a)$ in der Grammatik G_3 wird dann durch folgenden Baum dargestellt:



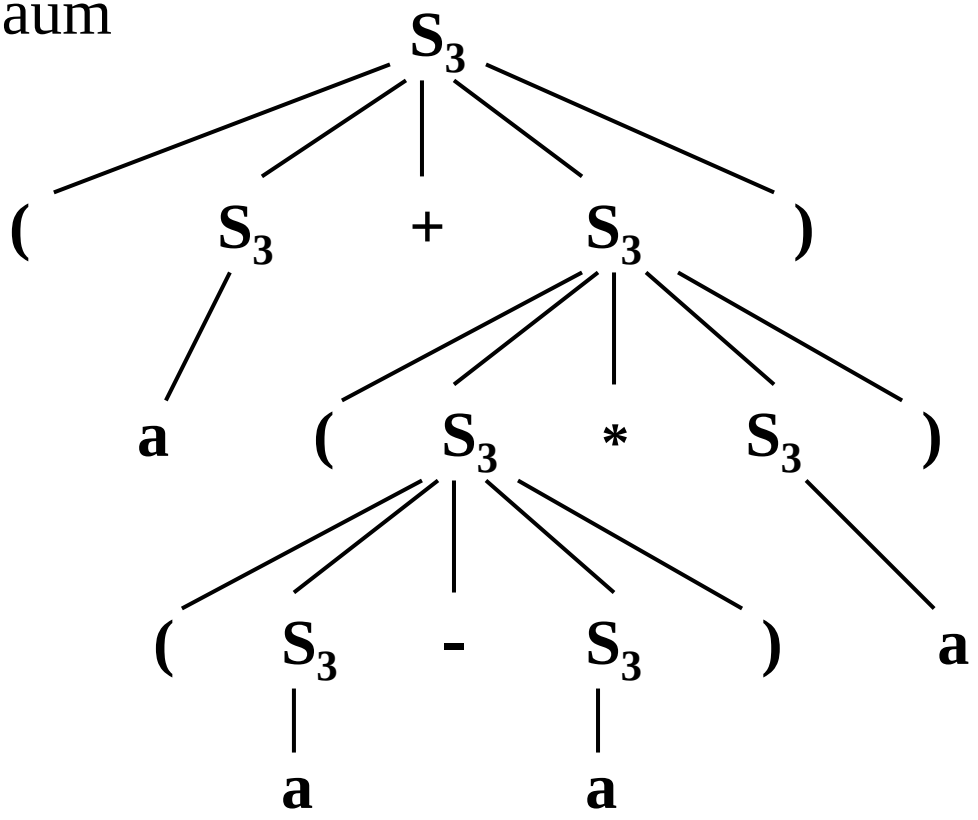
Definition:
Dieses Gebilde heißt Ableitungsbaum (engl.: derivation tree) zu obiger Ableitung für das Wort $(a + a)$ in der Grammatik G_3.

Obiger Baum ist zugleich der Ableitungsbaum zur Ableitung
$S_3 \Rightarrow (S_3 + S_3) \Rightarrow (a + S_3) \Rightarrow (a + a)$

Als weiteres Beispiel betrachten wir in G_3 die Ableitung

$$S_3 \Rightarrow (S_3 + S_3) \Rightarrow (a + S_3) \Rightarrow (a + (S_3 * S_3)) \Rightarrow (a + (S_3 * a))$$
$$\Rightarrow (a + ((S_3 - S_3) * a)) \Rightarrow (a + ((a - S_3) * a)) \Rightarrow (a + ((a - a) * a)) .$$

Ableitungsbaum hierzu:



Läuft man von links nach rechts durch die Blätter, so ergibt sich das abgeleitete Wort.

Diesen Ableitungsbaum besitzen auch andere Ableitungen, z.B.:

$$S_3 \Rightarrow (S_3 + S_3) \Rightarrow (S_3 + (S_3 * S_3)) \Rightarrow (S_3 + (S_3 * a))$$
$$\Rightarrow (S_3 + ((S_3 - S_3) * a)) \Rightarrow (S_3 + ((a - S_3) * a))$$
$$\Rightarrow (S_3 + ((a - a) * a)) \Rightarrow (a + ((a - a) * a))$$

oder:

$$S_3 \Rightarrow (S_3 + S_3) \Rightarrow (S_3 + (S_3 * S_3)) \Rightarrow (a + (S_3 * S_3))$$
$$\Rightarrow (a + ((S_3 - S_3) * S_3)) \Rightarrow (a + ((a - S_3) * S_3))$$
$$\Rightarrow (a + ((a - S_3) * a)) \Rightarrow (a + ((a - a) * a)).$$

Vereinbarung: *Ableitungen, die den gleichen Ableitungsbaum besitzen, sehen wir als gleich an.* Sie unterscheiden sich nur in der Reihenfolge, in der Regeln auf Nichtterminalzeichen angewendet werden, die an voneinander unabhängigen Stellen im Wort stehen.

Das Wort, das sich aus einem Ableitungsbaum ergibt, wenn man die Blätter von links nach rechts durchläuft, heißt das mit diesem Ableitungsbaum abgeleitete Wort.

Definition: Es sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik und $w \in L(G)$ ein Wort der erzeugten Sprache.

(1) w heißt eindeutig (bzgl. G), wenn es nur genau einen Ableitungsbaum gibt, dessen abgeleitetes Wort w ist.

(2) Gibt es mindestens zwei verschiedene Ableitungsbäume für w, so heißt w mehrdeutig (bzgl. G).

(3) G heißt eindeutig, wenn alle Wörter $w \in L(G)$ eindeutig sind.

(4) G heißt mehrdeutig, wenn mindestens ein Wort $w \in L(G)$ mehrdeutig ist.

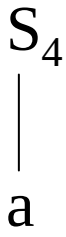
(Englisch: mehrdeutig = ambiguous, eindeutig = unambiguous.)

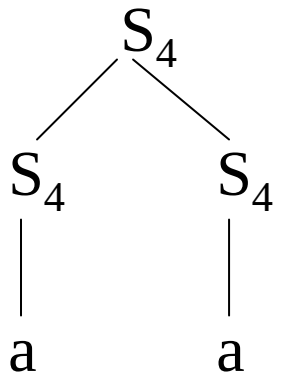
Beispiel: Wir untersuchen die folgende kontextfreie Grammatik $G_4 = (V_4, \Sigma_4, P_4, S_4)$ mit $V_4 = \{S_4\}$, $\Sigma_4 = \{a\}$ und $P_4 = \{S_4 \rightarrow S_4S_4,\ S_4 \rightarrow a\}$.

Offenbar kann man mit der ersten Regel aus S_4 jede nicht-leere Folge des Zeichens S_4 erzeugen; mit der zweiten Regel lässt sich hieraus dann jedes nicht-leere Wort über Σ_4 ableiten, d.h.,

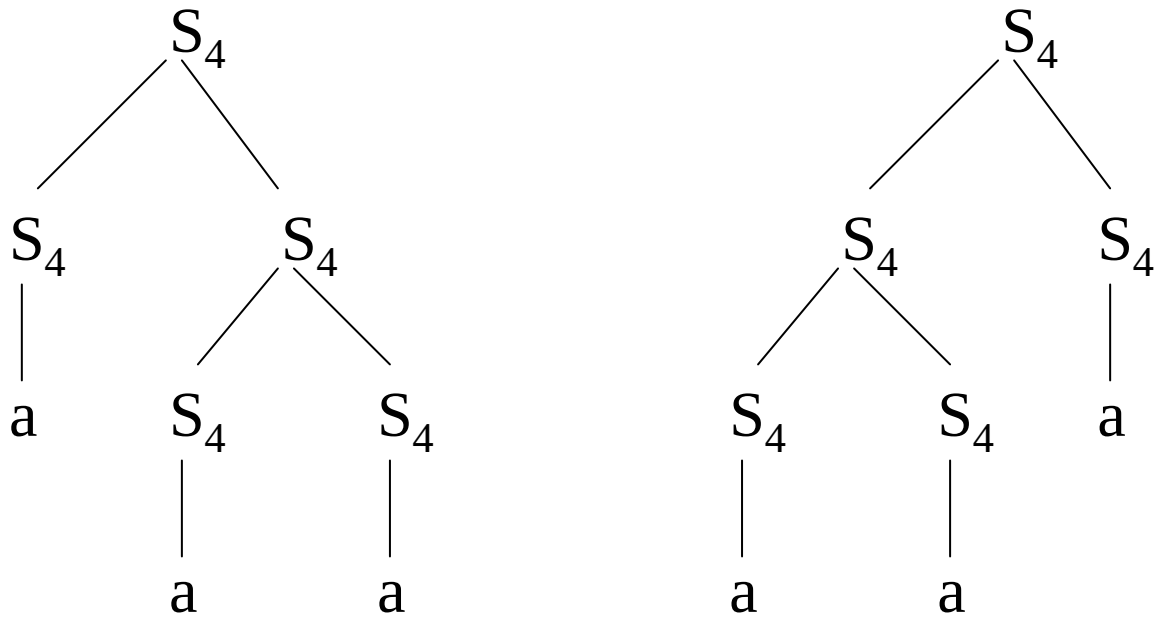
$L(G) = \Sigma_4^+ = \{a^n \mid n \geq 1\}$.

Die Wörter a und aa sind eindeutig bzgl. G_4. Sie besitzen nur die Ableitungsbäume:





Jedes Wort über Σ_4, das mindestens die Länge 3 besitzt, ist jedoch mehrdeutig. Zum Beispiel besitzt aaa die beiden verschiedenen (!) Ableitungsbäume



Beachten Sie: Damit zwei Bäume *gleich* sind, muss sich der eine Baum in der Ebene deckungsgleich auf den andern schieben lassen. Spiegelungen sind nicht erlaubt!

■

Hinweis zu natürlichen Sprachen: Dort sind Grammatiken gut bekannt. Ein Ausschnitt aus der deutschen Grammatik (Nichtterminalzeichen sind die in < ... > eingeschlossenen Zeichenfolgen) könnte lauten:

<Satz> → <Subjekt> <Prädikat> <Objekt>
<Subjekt> → <Artikel im Nominativ> <Substantiv im Nominativ>
<Objekt> → <Artikel im Dativ> <Substantiv im Dativ>
<Objekt> → <Artikel im Akkusativ> <Substantiv im Akkusativ>

<Artikel im Nominativ> → das	<Artikel im Nominativ> → die
<Artikel im Dativ> → dem	<Artikel im Dativ> → der
<Artikel im Akkusativ> → den	<Artikel im Akkusativ> → die
<Substantiv im Nominativ> → Kind	<Substantiv im Nominativ> → Luft
<Substantiv im Dativ> → Fernrohr	<Substantiv im Dativ> → Fernrohr
<Substantiv im Akkusativ> → Mann	
<Prädikat> → <Verb, 3.Person,Präsenz>	
<Verb, 3.Person, Präsenz> → sieht	<Verb, 3.Person, Präsenz> → lernt
<Verb, 3.Person, Präsenz> → hört	<Verb, 3.Person, Präsenz> → rennt

.....

Aus dem Startsymbol <Satz> kann man z.B. folgendes "Wort" ableiten:

<Satz> ⇒ <Subjekt> <Prädikat> <Objekt>
⇒ <Artikel im Nominativ> <Substantiv im Nominativ> <Prädikat> <Objekt>
⇒ das <Substantiv im Nominativ> <Prädikat> <Objekt>
⇒ das Kind <Prädikat> <Objekt> ⇒ ... ⇒ das Kind sieht den Mann

Leiten Sie weitere Wörter ab. Stellen Sie weitere Regeln auf. Weisen Sie Mehrdeutigkeiten nach wie "Das Kind sieht den Mann mit dem Fernrohr." ... usw. usw.

Fazit: Die Syntax einer Sprache lässt sich also mit kontextfreien Grammatiken beschreiben, wobei "Baumstrukturen" entstehen.

Der Vollständigkeit halber fügen wir hier einen Abschnitt über allgemeine Grammatiken (3.4) und die formale Definition der BNF (3.5) ein, die nicht im Kurs behandelt werden. *Der Kurs wird nun direkt zu 3.6 springen.*

Die bisher eingeführten Grammatiken heißen "kontextfrei", weil die Ersetzung eines Nichtterminalzeichens *ohne Beachtung des Kontexts* erfolgt, d.h., eine Regel $A \rightarrow w$ kann auf das Nichtterminalzeichen A in jedem Wort angewendet werden, ohne die links und rechts von A stehenden Zeichen zu beachten.

Darf man eine Regel $A \rightarrow w$ aber nur anwenden, wenn links von A das (Teil-) Wort x und rechts das (Teil-) Wort y stehen, dann würde man eine Regel $xAy \rightarrow xwy$ verwenden müssen. Dies führt zu den "kontextsensitiven" Grammatiken. Wir verallgemeinern diesen Gedanken weiter und stellen in 3.4. die Grammatiken allgemein vor.

3.4 Allgemeine Grammatiken

Definition (allgemein):
$G = (V, \Sigma, P, S)$ heißt (Chomsky-) Grammatik genau dann, wenn:

(1) V ist eine nicht-leere endliche Menge (die Menge der Nichtterminalzeichen oder Variablen),

(2) Σ ist eine nicht-leere endliche Menge (die Menge der Terminalzeichen),

(3) $S \in V$ ist ein Nichtterminalzeichen (das Startsymbol),

(4) $P \subset V^+ \times (V \cup \Sigma)^*$ ist eine endliche Menge (die Menge der Regeln oder Produktionen).

Wir haben an der Definition "kontextfreie Grammatik" (siehe Folie 142 zu Beginn von 3.3) nur eine Kleinigkeit geändert, indem wir in (4) die Menge V durch V^+ (siehe Folie 132) ersetzt haben. Die Begriffe "Ableitung" und "erzeugte Sprache" können nun unverändert übernommen werden (bei der Ableitung ist nur $p \rightarrow q$ statt $A \rightarrow w$ zu schreiben).

Definition: Gegeben sei eine Grammatik
$G = (V, \Sigma, P, S)$. Die Regelmenge P definiert auf der Menge $(V \cup \Sigma)^*$ die "Ableitungsrelationen" $\Rightarrow$ und $\Rightarrow^*$:

(1) Es gilt $u \Rightarrow v$ genau dann, wenn man die Wörter u und v in der Form $u = xpy$, $v = xqy$ mit $x, y \in (V \cup \Sigma)^*$ und $p \rightarrow q \in P$ schreiben kann. Man sagt: v ist aus u in einem Schritt ableitbar oder v lässt sich aus u in einem Schritt ableiten.

(2) Es gilt $u \Rightarrow^* v$ genau dann, wenn entweder $u = v$ ist oder wenn es Wörter $z_0, z_1, ..., z_k \in (V \cup \Sigma)^*$ für ein $k \geq 1$ gibt mit $u = z_0$, $v = z_k$, $z_i \Rightarrow z_{i+1}$ für alle $i = 0, 1, ..., k-1$.
Man sagt dann, v ist aus u herleitbar oder ableitbar.
(Die Zahl k heißt Länge der Ableitung.)

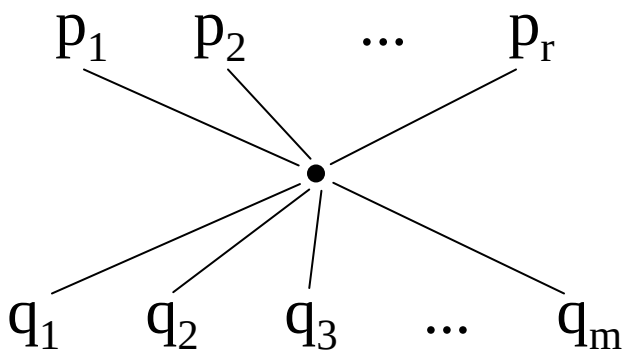
Hinweis: $\Rightarrow^*$ ist der so genannte "reflexive und transitive Abschluss" von $\Rightarrow$. (Englisch: Ableitung = derivation.)

Definition:

Die von einer Grammatik $G = (V, \Sigma, P, S)$ erzeugte Sprache (engl.: generated language) ist die Menge

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

Ableitungen kann man nun nicht mehr als Baum darstellen, vielmehr bilden sie ein "Netz", das aus Gebilden der Form



für eine Regel $p_1p_2p_3 \ldots p_r \rightarrow q_1q_2q_3 \ldots q_m$ mit $p_i \in V$, $q_j \in (V \cup \Sigma)$, $r \geq 1$, $m \geq 0$, aufgebaut wird. Wir betrachten ein Beispiel.

Beispiel: Gegeben sei folgende Grammatik

$G_5 = (V_5, \Sigma_5, P_5, S_5)$ mit $V_5 = \{S_5, A, B, C\}$, $\Sigma_5 = \{a, b\}$ und

$P_5 = \{S_5 \rightarrow AS_5B,\ S_5 \rightarrow CC,\ AC \rightarrow BA,\ CB \rightarrow BC,$
$BAB \rightarrow C,\ AB \rightarrow a,\ CC \rightarrow b\}.$

Um die "Arbeitsweise" der Grammatik zu verstehen, versucht man zunächst, einige Wörter herzuleiten:

$S_5 \Rightarrow CC \Rightarrow b$

$S_5 \Rightarrow AS_5B \Rightarrow ACCB \Rightarrow BACB \Rightarrow BABC \Rightarrow CC \Rightarrow b$

Hinweis: Dies sind offensichtlich zwei wesentlich verschiedene Ableitungen, d.h., die Grammatik G_5 ist mehrdeutig. Man kann jetzt aber nicht mehr mit Bäumen argumentieren, sondern man muss "Ableitungsnetze" betrachten.

Zunächst stellen wir die einzelnen Regeln dar:

$S_5 \rightarrow AS_5B$

$S_5 \rightarrow CC$

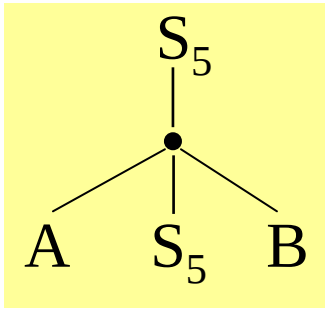
$AC \rightarrow BA$

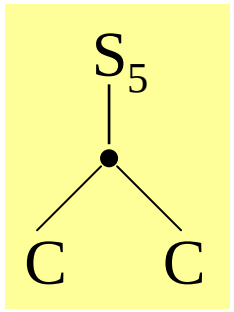
$CB \rightarrow BC$

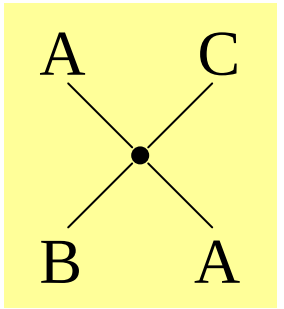
$BAB \rightarrow C$

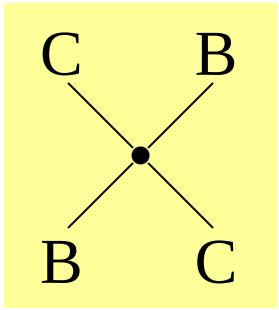
$AB \rightarrow a$

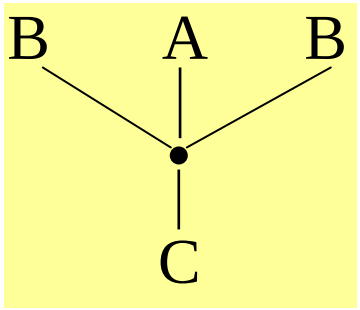
$CC \rightarrow b$

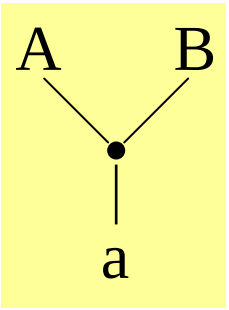


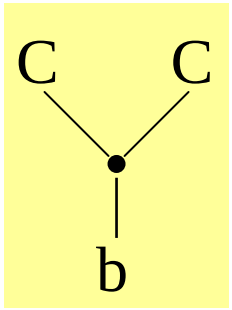












Nun kleben wir die einzelnen Regeln zu Ableitungen zusammen:

$S_5 \rightarrow A S_5 B$

$S_5 \rightarrow CC$

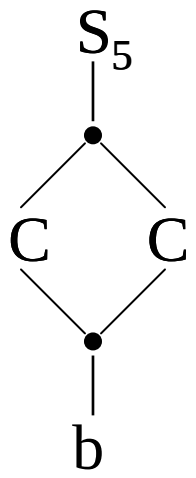
$AC \rightarrow BA$

$CB \rightarrow BC$

$BAB \rightarrow C$

$AB \rightarrow a$

$CC \rightarrow b$



Nun kleben wir die einzelnen Regeln zu Ableitungen zusammen:

$S_5 \rightarrow AS_5B$

$S_5 \rightarrow CC$

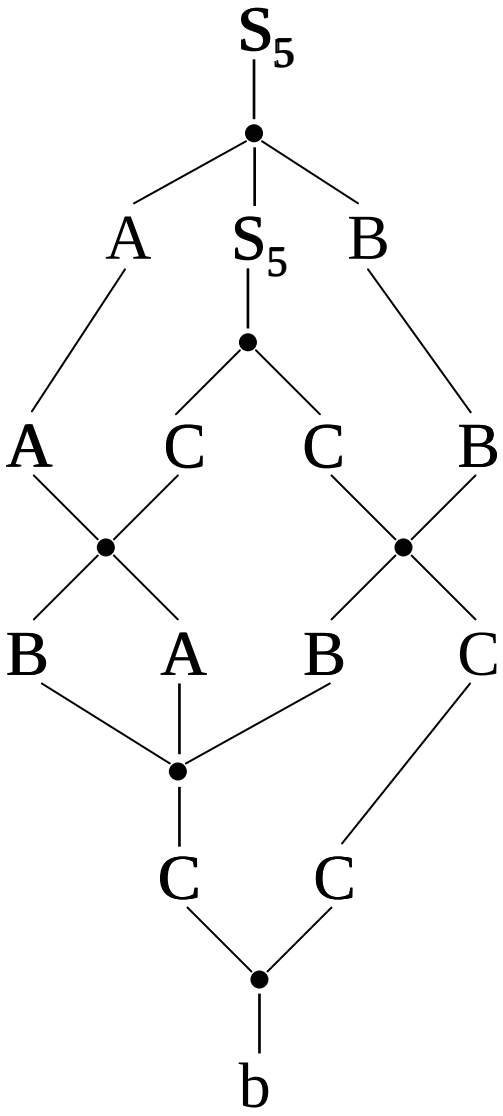
$AC \rightarrow BA$

$CB \rightarrow BC$

$BAB \rightarrow C$

$AB \rightarrow a$

$CC \rightarrow b$



S_5

$\Rightarrow AS_5B$

$\Rightarrow ACCB$

$\Rightarrow BACB$

$\Rightarrow BABC$

$\Rightarrow CC$

$\Rightarrow b$

Suche nach weiteren Ableitungen:

$$S_5 \Rightarrow AS_5B \Rightarrow ACCB \Rightarrow ACBC \Rightarrow ABCC \Rightarrow aCC \Rightarrow ab$$

$$S_5 \Rightarrow AS_5B \Rightarrow AAS_5BB \Rightarrow AACCBB \Rightarrow ABACBB$$
$$\Rightarrow ABABCB \Rightarrow ACCB \Rightarrow ACBC \Rightarrow ABCC \Rightarrow aCC \Rightarrow ab$$

Es gibt noch viele weitere Ableitungen, aber sie alle erzeugen eines der beiden Wörter b oder ab, d.h., $L(G_5) = \{b, ab\}$.

Stimmt das wirklich? Wie kann man so etwas beweisen? Wie kann man die Menge L(G) für eine Grammatik bestimmen bzw. wie kann man zu einem Wort w und einer Grammatik feststellen, ob diese Grammatik das Wort erzeugt oder nicht (sog. Wortproblem)? *Siehe Informatikstudium oder Bücher ...*

■

Definition: Gegeben sei eine Grammatik $G = (V, \Sigma, P, S)$.
Die Grammatik G heißt

(1) vom Typ 1 oder kontextsensitiv genau dann, wenn alle Regeln von der Form $xAy \rightarrow xwy$ sind mit $A \in V$, $x, y \in V^*$ und $w \in (V \cup \Sigma)^+$ (beachte: $w \neq \varepsilon$).

(2) vom Typ 2 oder kontextfrei genau dann, wenn alle Regeln von der Form $A \rightarrow w$ sind mit $A \in V$ und $w \in (V \cup \Sigma)^*$ ($w = \varepsilon$ ist hier erlaubt).

(3) vom Typ 3 oder rechtslinear genau dann, wenn alle Regeln von der Form $A \rightarrow uB$ oder $A \rightarrow u$ sind mit $A, B \in V$ und $u \in \Sigma^*$.

(4) linkslinear genau dann, wenn alle Regeln von der Form $A \rightarrow Bu$ oder $A \rightarrow u$ sind mit $A, B \in V$ und $u \in \Sigma^*$.

Definition: Eine Sprache heißt "xxx", wenn es eine "xxx" Grammatik G mit L = L(G) gibt für
"xxx" ∈ {kontextsensitiv, kontextfrei, rechtslinear, linkslinear}.

Die Einteilung in Grammatiken vom Typ 0 (= keine Einschränkung), Typ 1 (kontextsensitiv), Typ 2 (kontextfrei) und Typ 3 (rechtslinear) stammt aus der Originalarbeit von Noam Chomsky 1959 und hat sich bis heute gehalten. Es gibt aber mittlerweile viele weitere Grammatikklassen.

Alle in der Praxis verwendeten Programmiersprachen sind kontextsensitive Sprachen; sie lassen sich als kontextfreie Sprachen mit zusätzlichen Einschränkungen beschreiben.

Hinweise: Die Sprache der Bezeichner Bez und die Sprache der korrekten Zahldarstellungen sind rechtslinear. Die Sprache der korrekt geklammerten arithmetischen oder Booleschen Ausdrücke ist kontextfrei, aber nicht rechtslinear.

Einige Aussagen (ohne Beweis):

Fügt man zu einer "xxx" Sprache eine endliche Sprache hinzu oder zieht man eine endliche Sprache ab, so bleibt das Ergebnis eine "xxx" Sprache.

Eine Sprache genau dann rechtslinear, wenn sie linkslinear ist.

Die Sprache
$\{a^n b^n \mid n \geq 0\}$ ist kontextfrei, aber nicht rechtslinear.

Die Sprache
$\{a^n b^n a^n \mid n \geq 0\}$ ist kontextsensitiv, aber nicht kontextfrei.

Es gibt Sprachen vom Typ 0, die nicht kontextsensitiv sind (leider lassen sie sich nicht mit einer kurzen Formel beschreiben).

3.5 BNF

Definition: Backus-Naur-Form

Eine BNF ist ein Viertupel (V, Σ, P, S) mit

(1) V ist eine nicht-leere endliche Menge (die Menge der Nichtterminalzeichen); alle Elemente sind von der Form < Zeichenkette > (in "Zeichenkette" treten '<' und '>' nicht auf),

(2) Σ ist eine nicht-leere endliche Menge (die Menge der Terminalzeichen) mit $V \cap \Sigma = \emptyset$ und $| \notin \Sigma$,

(3) $S \in V$ ist ein Nichtterminalzeichen (das Startsymbol),

(4) $P \subset V \{::=\} (V \cup \Sigma)^* (\{|\} (V \cup \Sigma)^*)^*$ ist eine endliche Menge (die Menge der Regeln oder Produktionen).

Eine BNF ist eine kontextfreie Grammatik, bei der die Nichtterminalzeichen durch Namen der Form "< Zeichenkette >" genauer bezeichnet werden können und bei der alle kontextfreien Regeln, die die gleiche linke Seite besitzen, zu einer Regel zusammengefasst werden, wobei die verschiedenen rechten Seiten durch einen senkrechten Strich getrennt werden.

Statt des Ersetzungspfeils "$\rightarrow$" schreibt man "::=" (gesprochen: "Doppel-Doppelpunkt-Gleich"; die Bedeutung ist: "kann ersetzt werden durch").

Bei Programmiersprachen verwendet man gerne das Nichtterminalzeichen <program> als Startsymbol, aus dem man die zulässigen Programme herleitet.

Zum Formalismus: Welche Elemente sind in der Menge

$$V\,\{::=\}\,(V \cup \Sigma)^*\,(\{|\}\,(V \cup \Sigma)^*\,)^*$$

Dies ist die Konkatenation von vier Mengen, wobei die dritte und die vierte Menge beliebige Konkatenationen enthalten können. Betrachten wir diese Menge genauer:

$$\underbrace{V}\;\underbrace{\{::=\}}\;\underbrace{(V \cup \Sigma)^*}\;\underbrace{(\{|\}\,(V \cup \Sigma)^*\,)^*}$$

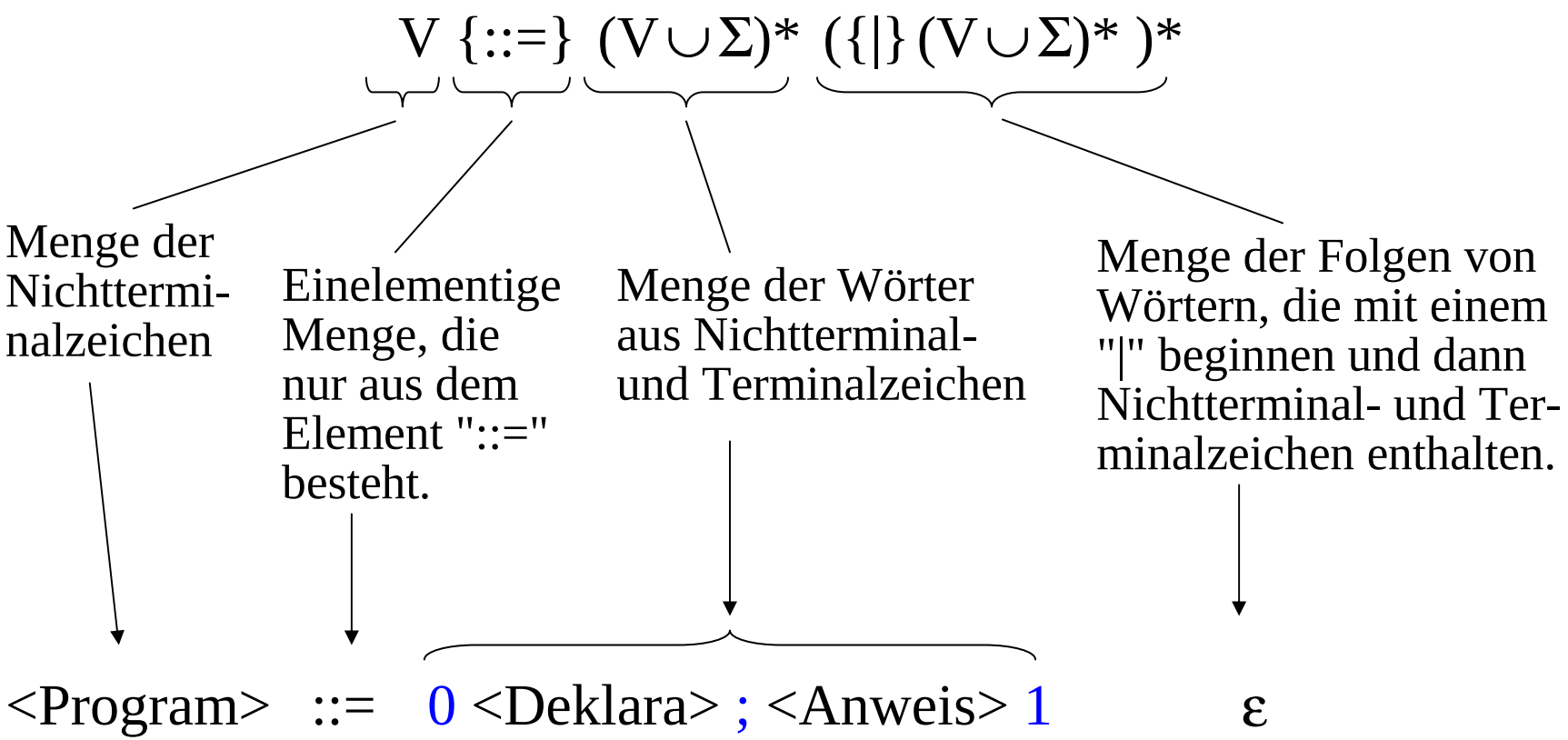
Menge der Nichtterminalzeichen

Einelementige Menge, die nur aus dem Element "::=" besteht.

Menge der Wörter aus Nichtterminal- und Terminalzeichen

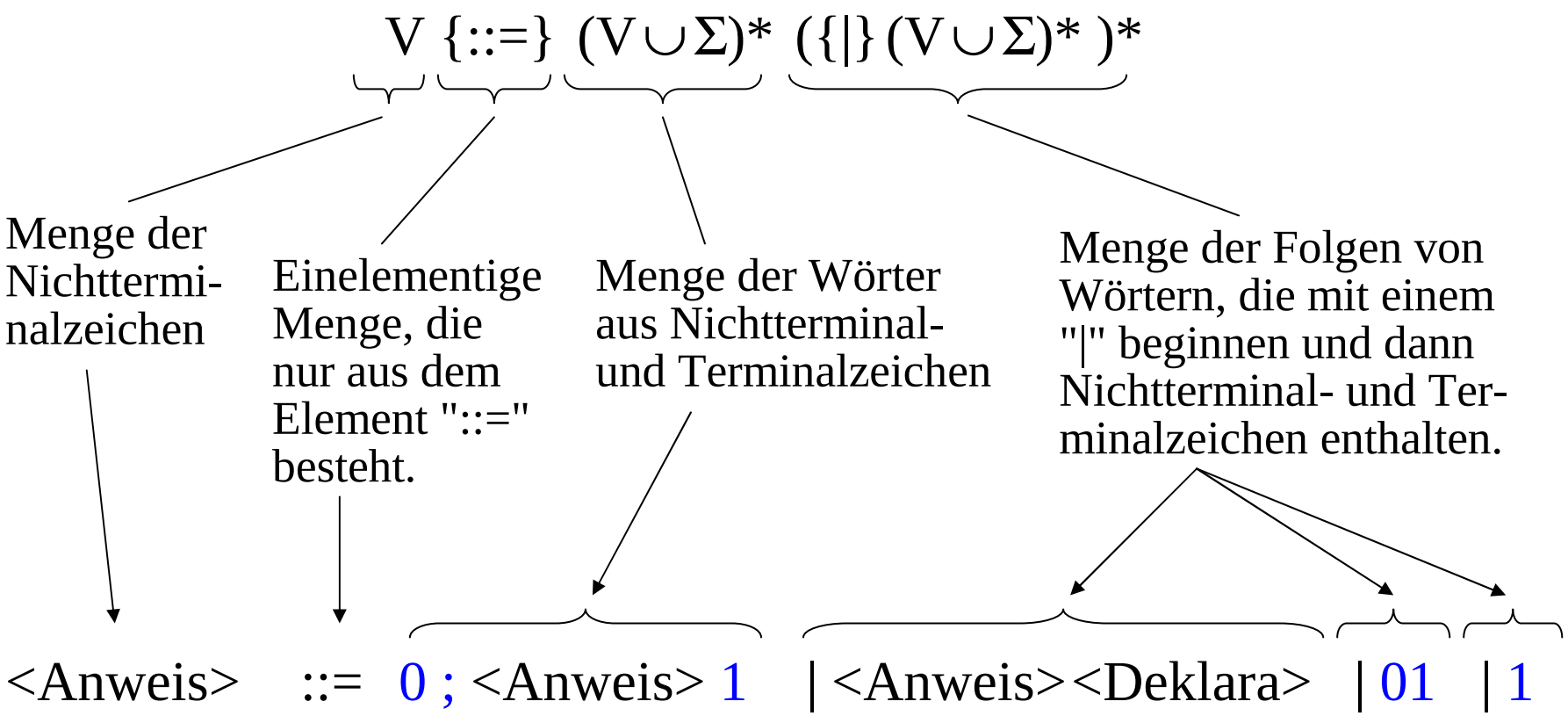
Menge der Folgen von Wörtern, die mit einem "|" beginnen und dann Nichtterminal- und Terminalzeichen enthalten.

Beispielwort, das in dieser Menge liegt, für
V = {<Program>, <Deklara>, <Anweis>} und Σ = {0, 1, ;} :



ein Zeichen *::=* *Wort der Länge 5 über V ∪ Σ* *leeres Wort*

Noch ein Beispielwort, das in dieser Menge liegt, für
V = {<Program>, <Deklara>, <Anweis>} und Σ = {0, 1, ;} :



ein Zeichen ::= Wort der Länge 4 3 Wörter, die jeweils mit "|" beginnen

Folglich gilt mit den obigen Mengen V und Σ:

<Program>::=0<Deklara>;<Anweis>1

$\in$ V {::=} $(V \cup \Sigma)^*$ ({|}$(V \cup \Sigma)^*$)*

und

<Anweis>::=0;<Anweis>1|<Anweis><Deklara>| 01| 1

$\in$ V {::=} $(V \cup \Sigma)^*$ ({|}$(V \cup \Sigma)^*$)*

Dieses Beispiel demonstriert nur die Wirkungsweise; es hat nichts mit einer konkreten Anwendung zu tun. Anwendungen hatten wir bereits in 3.3. kennen gelernt: Dort hatten wir unsere Programmiersprache in BNF formuliert (Folie 160, siehe auch Folie 181).

Umwandlung einer kontextfreien Grammatik in die BNF:
Gegeben sei eine kontextfreie Grammatik (V', Σ, P', S').

1. Ersetze jedes Nichtterminalzeichen $A' \in V'$ durch einen aussagekräftigen Namen A der Form <Zeichenkette> , wobei die Zeichen '<' und '>' in dieser "Zeichenkette" nicht auftreten dürfen. Dies ergibt die neue Menge der Nichtterminalzeichen V mit dem Startsymbol S (anstelle S').

2. Ersetze in P' jedes Nichtterminalzeichen durch seinen neuen Namen der Form <Zeichenkette>, d.h.: Wenn $A' \rightarrow u'$ eine Regel in P' ist, so ersetze sie durch $A \rightarrow u$, wobei A das zu A' gehörende neue Nichtterminalzeichen aus V ist und wobei man u aus u' erhält, indem man alle in u' auftretenden Nichtterminalzeichen entsprechend durch ihre neuen Bezeichnungen der Form "< ... >" ersetzt.

3. Wenn $A \rightarrow u_1$, $A \rightarrow u_2$, ..., $A \rightarrow u_k$ alle Regeln im neuen P' mit A als linker Seite sind, dann ersetze diese k Regeln durch
   $A ::= u_1 \mid u_2 \mid ... \mid u_k$.

So entsteht aus P' die neue Regelmenge P.

Umgekehrt kann man mit dieser Anleitung auch eine BNF in eine "normale" kontextfreie Grammatik zurück verwandeln.

Der Ableitungsbegriff der BNF ist daher der gleiche wie der für kontextfreie Grammatiken. Somit sind die Ableitungsrelationen $\Rightarrow$ und $\Rightarrow^*$ sowie die erzeugte Sprache L(G) auch für die BNF definiert.

Die BNF wird nun um einige hilfreiche Abkürzungen zur EBNF erweitert.

3.6 EBNF

Verwenden von Schlüsselwörtern:
Schlüsselwörter der Sprache werden als Zeichenketten dargestellt, die in Apostrophe eingeschlossen werden.
(Tritt im Schlüsselwort ein Apostroph auf, so wird dies durch zwei Apostrophe dargestellt.)
Beispiel:
<Boolesche Operatoren> ::= 'and' | 'or'

Hierdurch wird ausgedrückt, dass die Operatoren and und or eigentlich einzelne Terminalzeichen sind, jedoch als Zeichenkette durch Konkatenation anderer Zeichen dargestellt werden. Man könnte ansonsten in BNF auch schreiben:

<And-Operator> ::= and
<Or-Operator> ::= or
<Boolesche Operatoren> ::= <And-Operator> | <Or-Operator>

Einführen von eckigen Klammern ("ein- oder keinmal"): Symbole oder Folgen von Symbolen, die auch wegfallen dürfen, werden in eckige Klammern eingeschlossen.

Beispiele:

<Ziffernfolge> ::= <Ziffer> [<Ziffernfolge>]

ist die Abkürzung für

<Ziffernfolge> ::= <Ziffer> | <Ziffer> <Ziffernfolge>

<Alternative> ::= 'if' <Boolescher Ausdruck>
'then' <Anweisung>
['else' <Anweisung>] 'fi'

Einführen von geschweiften Klammern ("Iteration"):
Symbole oder Folgen von Symbolen, die beliebig oft (auch keinmal) wiederholt werden dürfen, werden in geschweifte Klammern eingeschlossen.

Beispiele:

<Ziffernfolge> ::= <Ziffer> {<Ziffer>}
ist die Abkürzung für
<Ziffern*> ::= ε | <Ziffer> <Ziffern*>
<Ziffernfolge> ::= <Ziffer> <Ziffern*>

Die Definition von Bezeichnern lässt sich mit geschweiften Klammern sehr kurz schreiben:
<Bezeichner> ::= <Buchstabe> { _ | <Buchstabe> | <Ziffer> }

Definition: Erweiterte Backus-Naur-Form

Die EBNF ist die Erweiterung der BNF um die Möglichkeiten:

Apostrophe für Schlüsselwörter als Terminalzeichen.

Eckige Klammern mit der Bedeutung: Die eingeschlossenen Teile können hier ein Mal auftreten, müssen aber nicht.

Geschweifte Klammern mit der Bedeutung: Die hierdurch eingeschlossenen Teile können beliebig oft nacheinander auftreten. (Dies entspricht dem Operator * auf Wortmengen.)

Beachten Sie, dass "|" (mit der Bedeutung "oder") nun auch innerhalb der eckigen und geschweiften Klammern auftreten darf (siehe Beispiel auf voriger Folie, letzte Zeile). Man muss dann oft zusätzlich runde Klammern verwenden.

Ergänzender Hinweis: Falls eckige oder geschweifte Klammern als Terminalzeichen in den Regeln auftreten können, so bleibt es dem Ersteller der Regeln überlassen, für die Eindeutigkeit zu sorgen.

Hinweis: Es gibt Varianten der EBNF.

Oft schreibt man (...)* oder auch < ...> statt {...}.
Man führt auch das Konstrukt (...)+ ein, wenn man die 0-Iteration verbieten möchte.

Statt [...] ("einmal oder keinmal") findet man auch (...)? .

Statt Apostrophen werden oft auch Anführungsstriche verwendet.

Statt ::= wird manchmal auch einfach = geschrieben.

Terminalzeichen (und auch Schlüsselwörter) müssen oftmals in Anführungsstriche eingeschlossen werden; die Schlüsselwörter werden oft fett gedruckt. Die Nichtterminalzeichen werden dann nicht mehr in spitze Klammern '<' und '>' eingeschlossen.

Regeln werden mit einem besonderen Zeichen, meist mit einem Punkt, abgeschlossen.

Beispiel: Anweisung (statement) in Java 1.1:

```
statement  =  variable_declaration  |  ( expression  ";"  )  |  statement_block  |
              if_statement  |  do_statement  | while_statement |
              for_statement  |  try_statement  |  switch_statement  |
              ( "synchronized"  "(" expression  ")" statement  )  |
              ( "return"  [ expression ]  ";"  )  |  (  "throw" expression ";"  )  |
              ( identifier  ":" statement )  |  ( "break"  [ identifier ]  ";"  )  |
              ( "continue"  [ identifier ]  ";"  )  |  ";"   .
statement_block  =  "{"  { statement }  "}"  .
if_statement = "if"  "(" expression  ")" statement  [ "else" statement ]  .
do_statement = "do" statement  "while"  "(" expression  ")"  ";"   .
while_statement = "while"  "(" expression  ")" statement  .
for_statement = "for"  "("  ( variable_declaration  |  ( expression  ";"  )  |  ";"  )
              [ expression ]  ";"  [ expression ]  ";"  ")" statement  .
try_statement = "try" statement  { "catch"  "(" parameter  ")" statement }
              [ "finally" statement ]  .
switch_statement = "switch"  "(" expression  ")"
              "{"  { ( "case" expression  ":" ) |  ( "default"  ":" ) | statement }  "}"  .
```

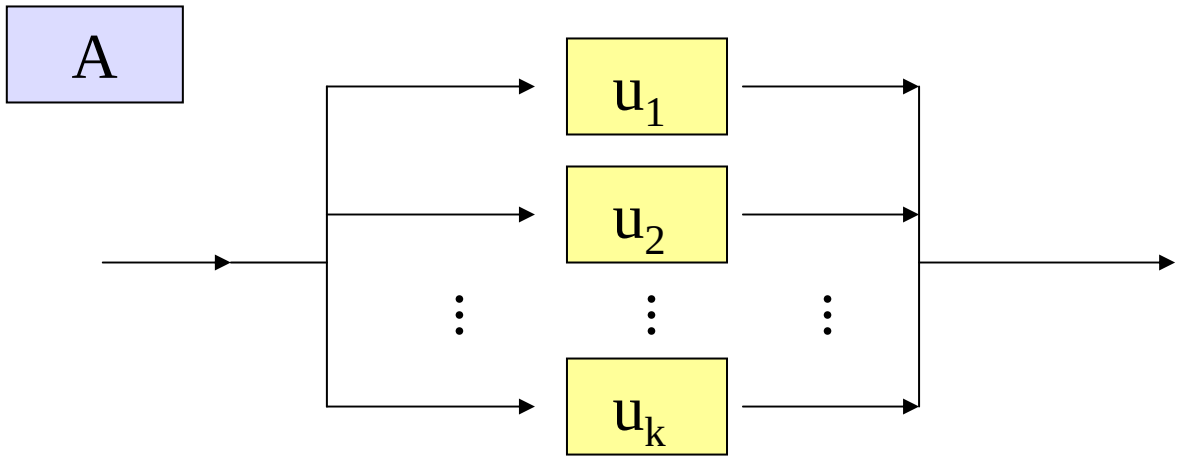
3.7 Syntaxdiagramme

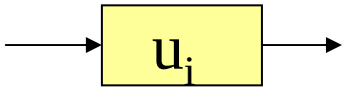
Grafische Darstellung von EBNF-Regeln. Für jede linke Seite (also für jedes Nichtterminalzeichen) legen wir ein eigenes Diagramm an, das mit dem Nichtterminal bezeichnet wird.

Betrachte eine solche Regel mit dem Nichtterminal A:

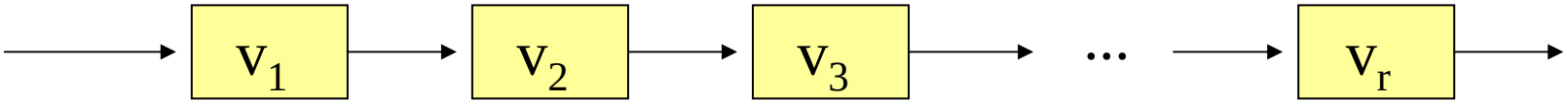
$$A ::= u_1 \mid u_2 \mid ... \mid u_k$$

Dann zeichne hierzu das Diagramm

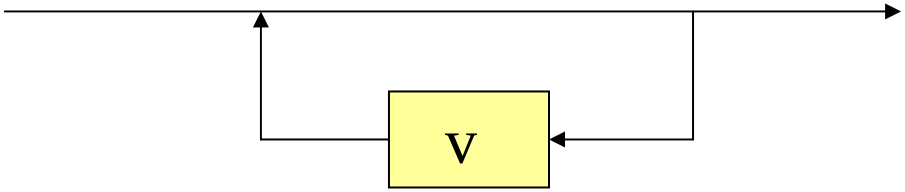


Ist ein u_i von der Form $v_1 v_2 \ldots v_r$ (Konkatenation von Zeichen),
so ersetze 

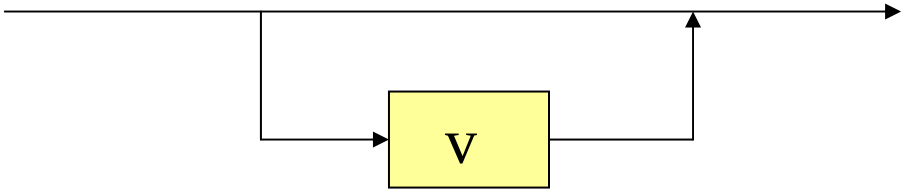
durch:



Ist ein u_i oder v_j von der Form { v } , so ersetze es durch



Ist ein u_i oder v_j von der Form [v] , so ersetze es durch



Zum Schluss ersetzt man bei allen Terminalzeichen und Schlüsselwörtern die rechteckigen Umrandungen durch Kreise.

Dieses Vorgehen liefert schließlich für jede Grammatik bzw. EBNF eine grafische Darstellung, das sog. Syntaxdiagramm.

Um Wörter aus einem Nichtterminalzeichen A zu erzeugen, durchläuft man das zu A gehörende Syntaxdiagramm vom Eingangspfeil bis zum Ausgangspfeil. Hier gibt es i.A. viele Wege. Für den gewählten Weg sammelt man die hierbei besuchten Terminalzeichen (= die in Kreisen stehenden Zeichen) in der Durchlaufreihenfolge auf. Alle diese Zeichenfolgen bilden dann die Menge der von A erzeugten Wörter. Genauere Formulierung:

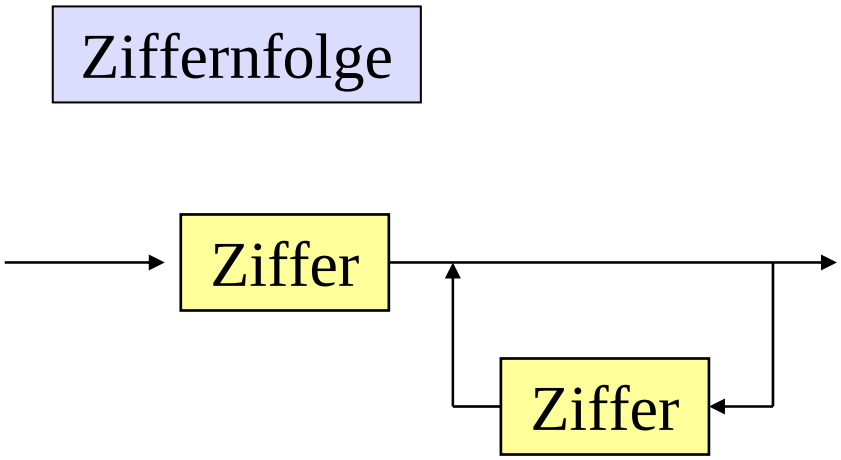
Präzisiere Beschreibung der Bedeutung von Syntaxdiagrammen: Gegeben sind Syntaxdiagramme und eine anfangs leere Ausgabe.

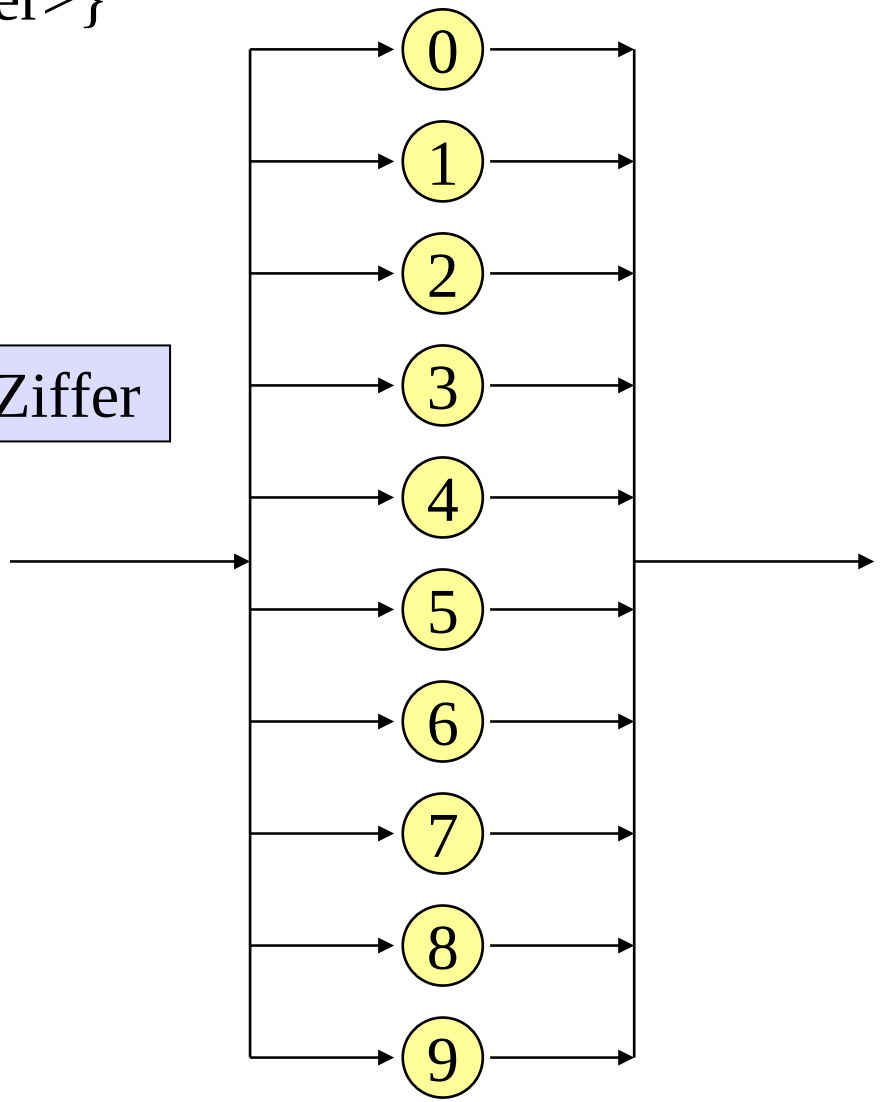
Ziel ist es, das Diagramm, das zum Startsymbol gehört, in Richtung der Pfeile vom Eingangspfeil bis zum Ausgangspfeil zu durchlaufen. Trifft man hierbei auf Zeichen in Kreisen, so schreibt man diese in der Durchlaufreihenfolge hinter die bereits vorhandene Ausgabe. Trifft man auf ein Rechteck mit dem Nichtterminalzeichen A, so klebt man das zu A gehörende Diagramm an dieser Stelle ein und durchläuft das neu entstandene Diagramm weiter. (Gibt es kein zu A gehörendes Diagramm, so bricht man ergebnislos ab.)

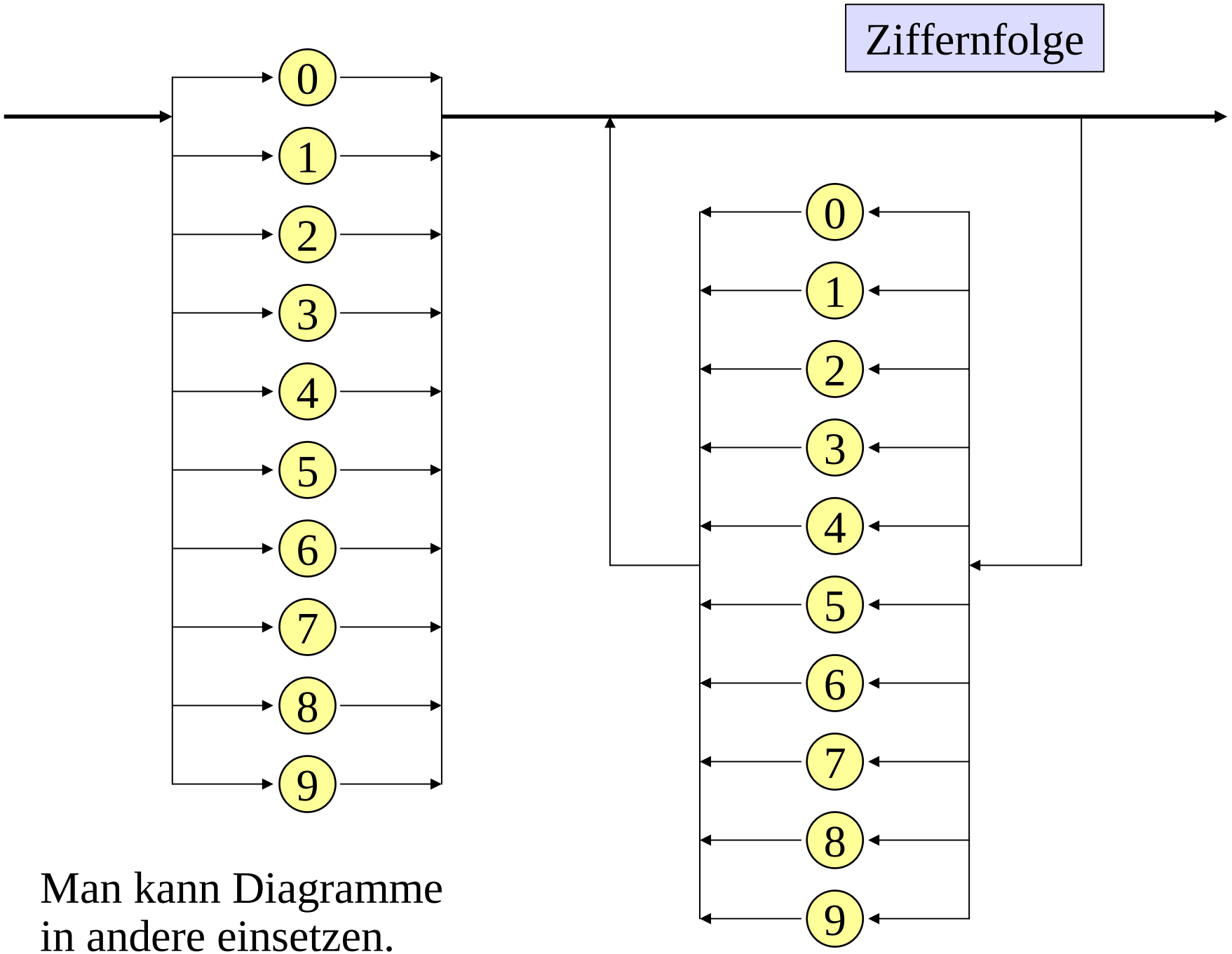
Alle möglichen Ausgaben, die zu einem vollständigen Durchlauf vom Eingangspfeil bis zum Ausgangspfeil des Startsymbol-Diagramms gehören, bilden die erzeugte Sprache.

Beispiel : <Ziffer> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Ziffernfolge> ::= <Ziffer> {<Ziffer>}

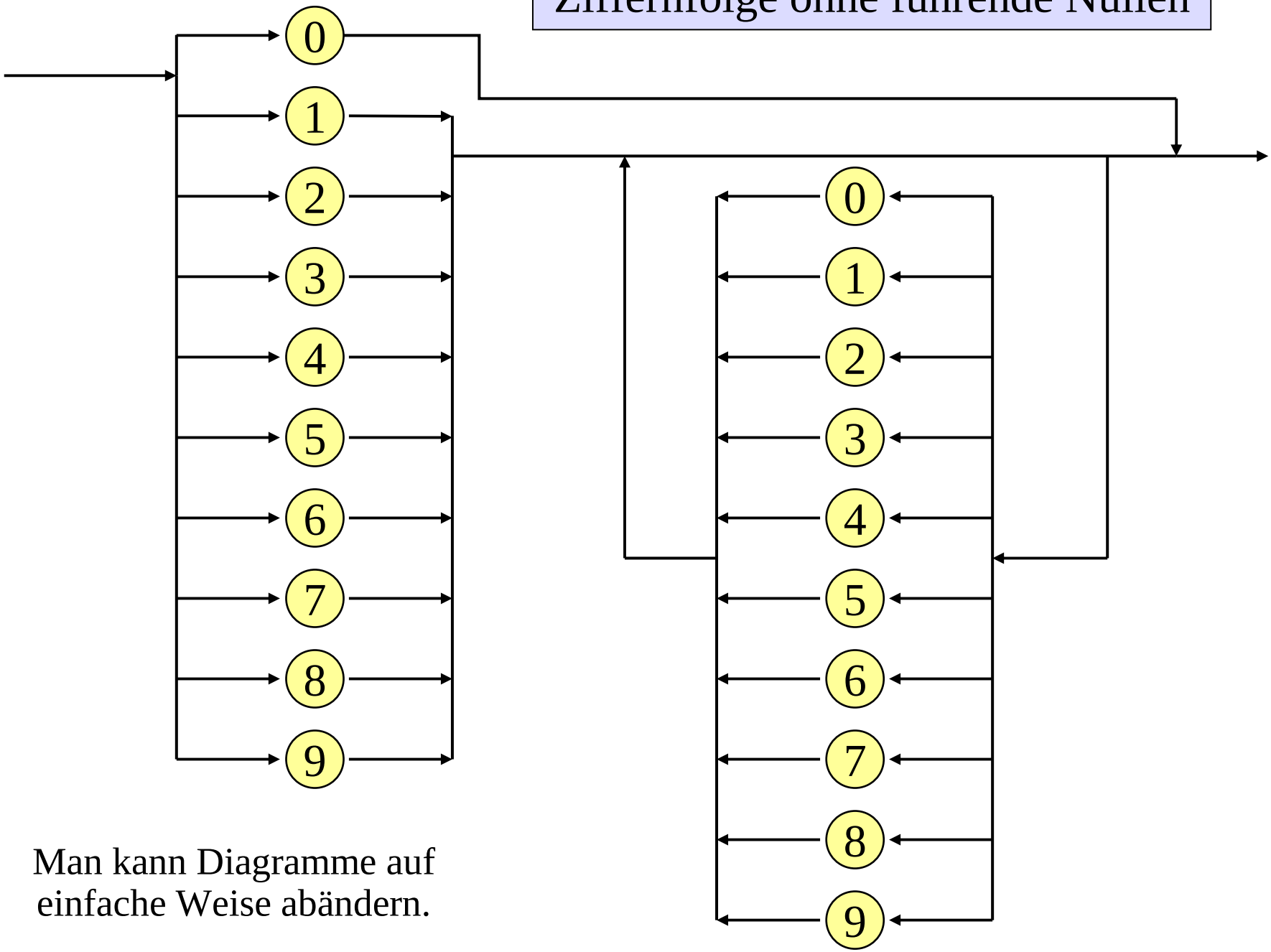






Man kann Diagramme
in andere einsetzen.

Ziffernfolge ohne führende Nullen

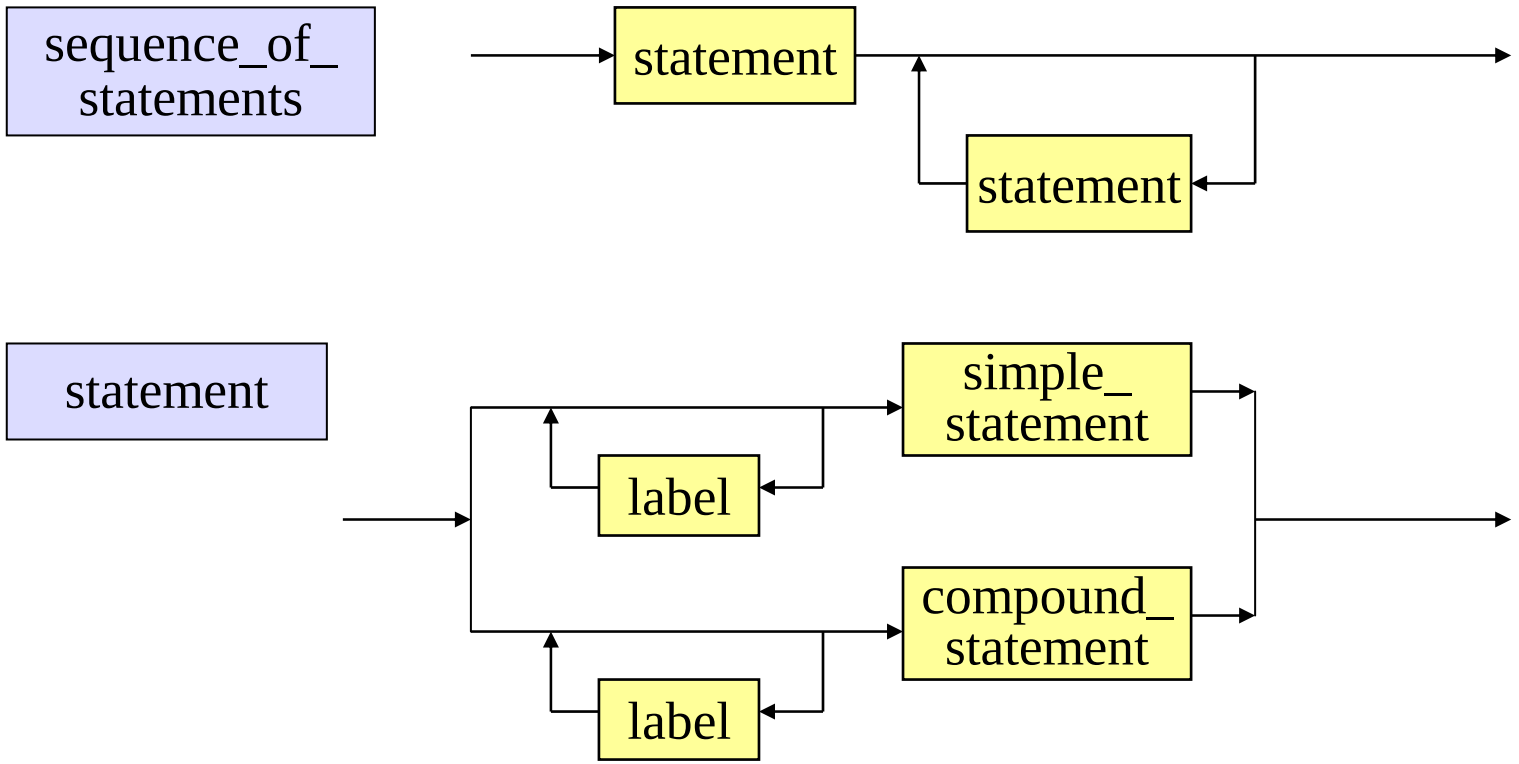


Man kann Diagramme auf einfache Weise abändern.

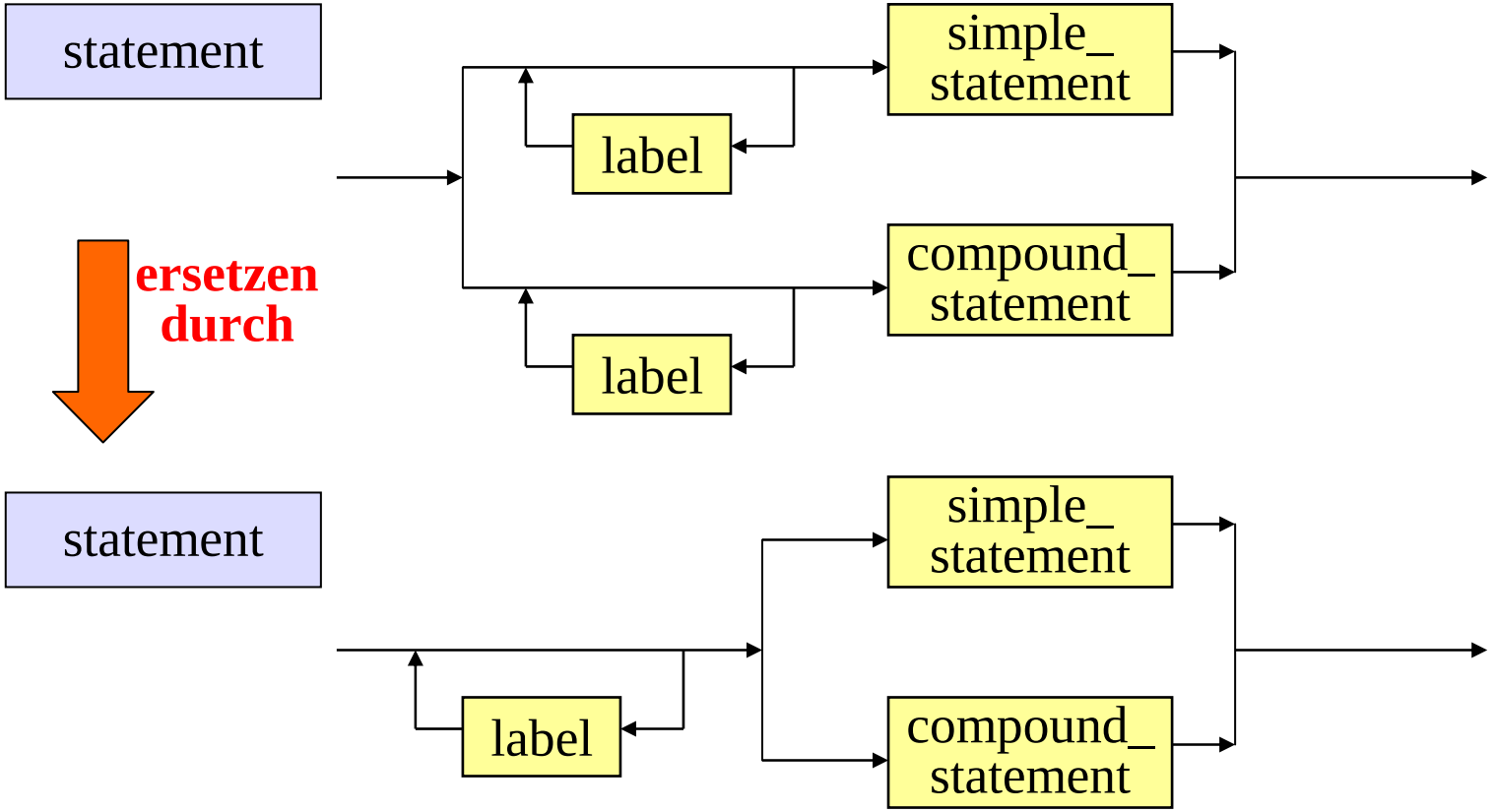
Beispiel: Konkrete Programmiersprache: Anweisungen in Ada

<sequence_of_statements> ::= <statement> {<statement>}

<statement> ::= {<label>} <simple_statement> |
{<label>} <compound_statement>



Es gibt oft Vereinfachungen bei der grafischen Darstellung:

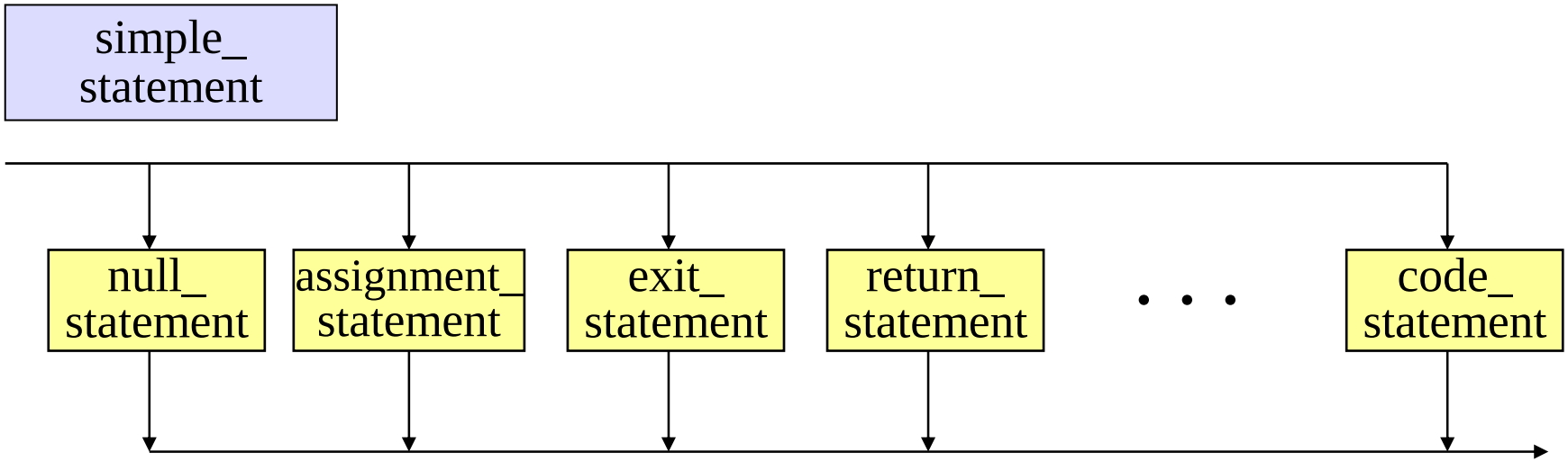


Dies lässt sich auch in der EBNF beschreiben:

<statement> ::= {<label>} (<simple_statement> | <compound_statement>)

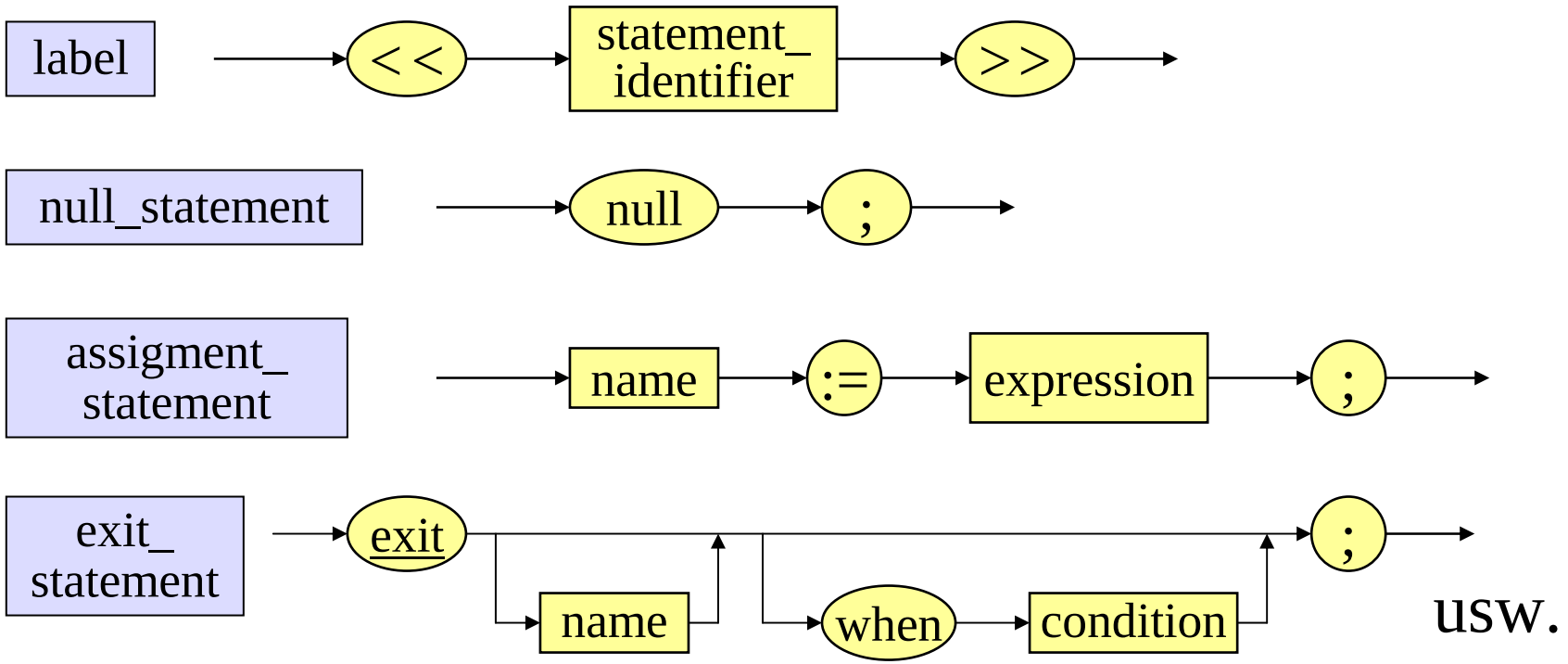
Fortsetzung des Beispiels Ada-Anweisungen

<simple_statement> ::= <null_statement> |
<assignment_statement> | <exit_statement> |
<return_statement> | <procedure_call_statement> |
<entry_call_statement> | <goto_statement> |
<requeue_statement > | <delay_statement> |
<abort_statement> | <raise_statement> | <code_statement>



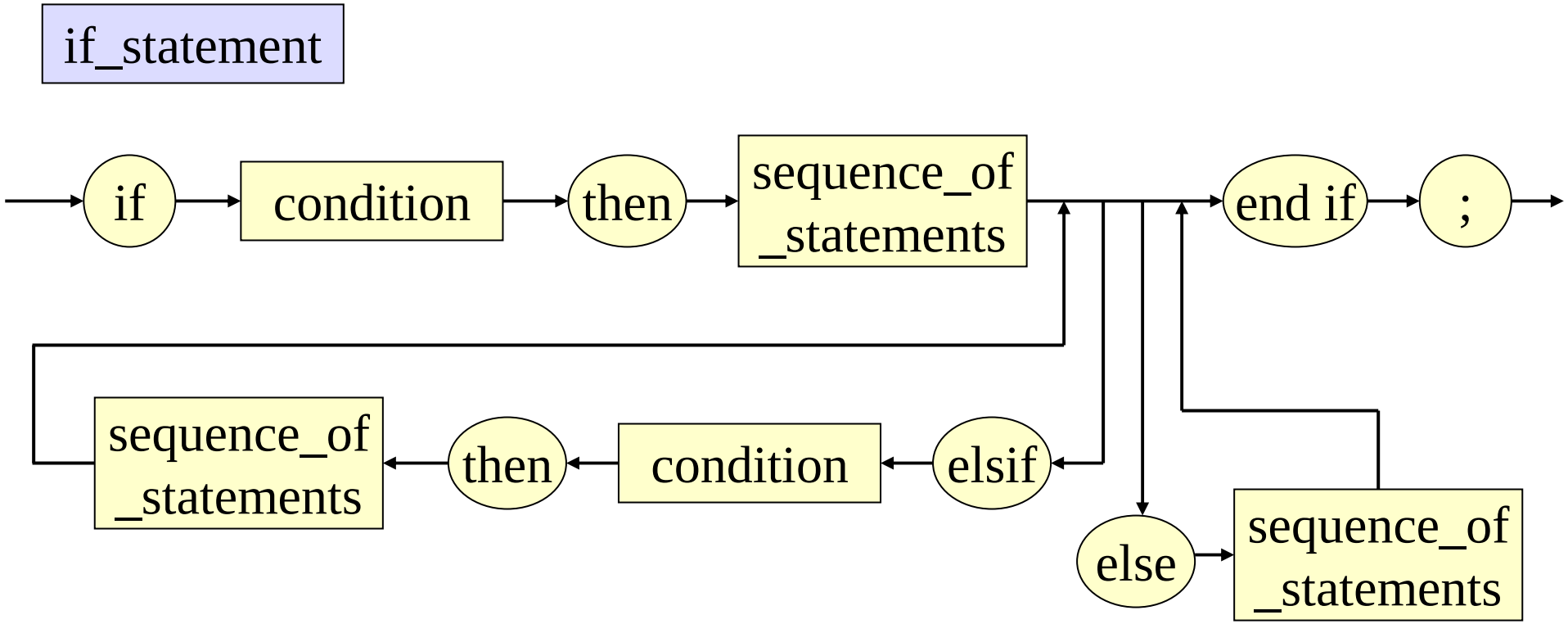
Fortsetzung des Beispiels Ada-Anweisungen

label ::= "<<" statement_identifier ">>"
statement_identifier ::= direct_name
null_statement ::= "null" ";"
assignment_statement ::= name ":=" expression ";"
exit_statement ::= "exit" [name] ["when" condition] ";"
goto_statement ::= "goto" name ";"



Fortsetzung des Beispiels Ada-Anweisungen

if_statement ::= "if" condition "then" sequence_of_statements
{"elsif" condition "then" sequence_of_statements}
["else" sequence_of_statements] "end if" ";"



3.8 Sprachen zur Beschreibung von Sprachen

Eine Sprache, mit der man eine andere Sprache beschreiben kann, nennt man Metasprache. Dabei muss man zwischen mehreren "Ebenen" unterscheiden:

Ebene "Syntax": Hier geht um den korrekten Aufbau der Wörter bzw. Sätze einer Sprache. Der oft zitierte Satz

"Der Tisch ist ein Säugetier."

ist syntaktisch (oder grammatikalisch) korrekt, völlig unabhängig davon, ob er inhaltlich zutrifft.

Kritischer ist dies bei einem Satz wie

"Nachts ist es kälter als draußen."

da es möglicherweise von der deutschen Grammatik her nicht zulässig ist, eine Temperaturangabe mit einer Ortsangabe zu vergleichen. Dennoch klingt der Satz syntaktisch korrekt.

Die formal richtige Anordnung der Sprachelemente lässt sich insbesondere durch Grammatiken oder gleichwertige Kalküle festlegen. Dies wurde in der Informatik in den letzten vierzig Jahren gut untersucht und hier liegen zugleich umfangreiche Erfahrungen aus der Praxis vor.

Will man die Syntax von Programmiersprachen oder anderen Sprachen im mathematisch-technisch-naturwissenschaftlichen Bereich beschreiben, so verwendet man spezielle kontextfreie Grammatiken, die folgende Eigenschaft besitzen: Ist der Aufbau der Wörter bzw. Sätze einer Sprache hierin formuliert, so kann man automatisch einen Algorithmus (einen sog. "Parser") erzeugen lassen, der das Wortproblem in relativ kurzer Zeit löst, d.h., der zu *jedem* Wort über dem Terminalalphabet feststellt, ob es zur Sprache gehört oder nicht, und dies in einer Zeitspanne, die proportional zur Länge des Wortes ist.

Ebene "Semantik": Die Semantik ordnet jedem Wort bzw. Satz einer Sprache seine Bedeutung zu. Die Bedeutung kann recht unterschiedlich sein: Die Bedeutung von Sätzen der natürlichen Sprachen ist in der Regel eine Handlung oder eine Information; bei mathematisch-naturwissenschaftlichen Aussagen geht es meist um deren inhaltliche Korrektheit, "logische Gültigkeit" und Widerspruchsfreiheit; bei Programmiersprachen weist die Semantik jedem Programm seine realisierte Abbildung zu.

Eine genaue Semantik ist unverzichtbar für die Sicherheit von Software. Zum einen muss das Programm korrekt sein, also genau das durchführen, was (in einer sog. "Spezifikation") vorgegeben ist, zum anderen muss es zuverlässig arbeiten, also z.B. gegenüber Veränderungen der Semantik, bedingt durch eine neue Umgebung, stabil sein.

Ebene "Pragmatik": Die Pragmatik untersucht und beschreibt die Beziehungen zwischen der Sprache und deren Benutzern (Menschen, Maschinen) bzw. der jeweiligen Umwelt. Dies können einfache Fragen der Auswirkungen von Wörtern bzw. Sätzen sein, aber auch verwickelte Zusammenhänge zwischen Interessen, die man mit der Benutzung einer Sprache verfolgt.

Die Pragmatik führt aus dem Bereich der über Alphabeten aufgebauten Sprachen hinaus ("welche Wirkung soll ein Satz erzielen?"), wirkt aber auch in sie von außen hinein, indem sie z.B. die Begründung für Sprachelemente oder Darstellungen von Daten ist ("wir führen Felder *(arrays)* ein, weil man hiermit die in der Technik verwendeten Vektoren darstellen kann").

Der Pragmatik wurde in der Informatik bisher wenig Aufmerksamkeit gewidmet; dies ändert sich aber zurzeit.

Fast jede natürliche Sprache (Deutsch, Chinesisch, Latein, Englisch usw.) kann als Metasprache verwendet werden, z.B., indem wir mit dieser Sprache eine andere natürliche Sprache beschreiben und erlernen (Fremdsprachenunterricht).

Natürliche Sprachen besitzen die Eigenschaft, *sich selbst* beschreiben zu können (Muttersprachunterricht einschl. Grammatikkunde). Diese Eigenschaft finden wir bei vielen künstlichen Sprachen auch: Oft lässt sich in einer Sprache die Sprache selbst beschreiben; insbesondere kann man hiermit die Sprache erweitern, indem man die Syntax und die Bedeutung der zusätzlichen Sprachelemente in der Sprache selbst formuliert. Natürliche Sprachen und Programmiersprachen können sich auf diese Weise ständig weiterentwickeln und neue Gebiete der Beschreibung und Bearbeitung erschließen.

Wir demonstrieren diese "Selbstbeschreibungsfähigkeit" an einem Beispiel, indem wir kontextfreie Grammatiken mit Hilfe der EBNF erzeugen (die EBNF ist selbst wieder eine kontextfreie Grammatik, siehe Abschnitte 3.3 und 3.6).

Hierzu müssen wir Grammatiken als Wörter einer Sprache aufschreiben. Die Grammatik G_1 aus Abschnitt 3.3

$$G_1 = (V_1, \Sigma_1, P_1, S_1) \text{ mit } V_1 = \{S_1\},$$
$$\Sigma_1 = \{0, 1\}, \; P_1 = \{(S_1, 1), (S_1, S_1 0), (S_1, S_1 1)\}$$

kann man als folgendes Wort der Länge 25

$$\mathbf{S_1;0,1;(S_1,1)(S_1,S_1 0)(S_1,S_1 1);S_1} \in \Sigma^*$$

über dem Alphabet Σ = { '**(**' , '**)**' , '**,**' , '**;**' , '**0**' , '**1**' , '$\mathbf{S_1}$' } auffassen.

Auf diese Weise erzeugen wir nun (kontextfreie) Grammatiken aus einer EBNF.

Beispiel: Die folgende EBNF (V_G, Σ_G, P_G, <Grammatik>) umfasst die Sprache aller kontextfreien Grammatiken, deren Terminalzeichen alle von der Form Tz und deren Nichtterminalzeichen alle von der Form Nz sind mit z∈{1}{0,1}*:

V_G = {<Ntz>, <Tz>, <P>, <rSeite>, <Grammatik>},

Σ_G = { '**(**' , '**)**' , '**,**' , '**;**' , '**0**' , '**1**' , '**N**' , '**T**' },

Regeln:

<Grammatik> ::=
<Ntz> {',' <Ntz>} ';' <Tz> {',' <Tz>} ';' <P> ';' <Ntz>

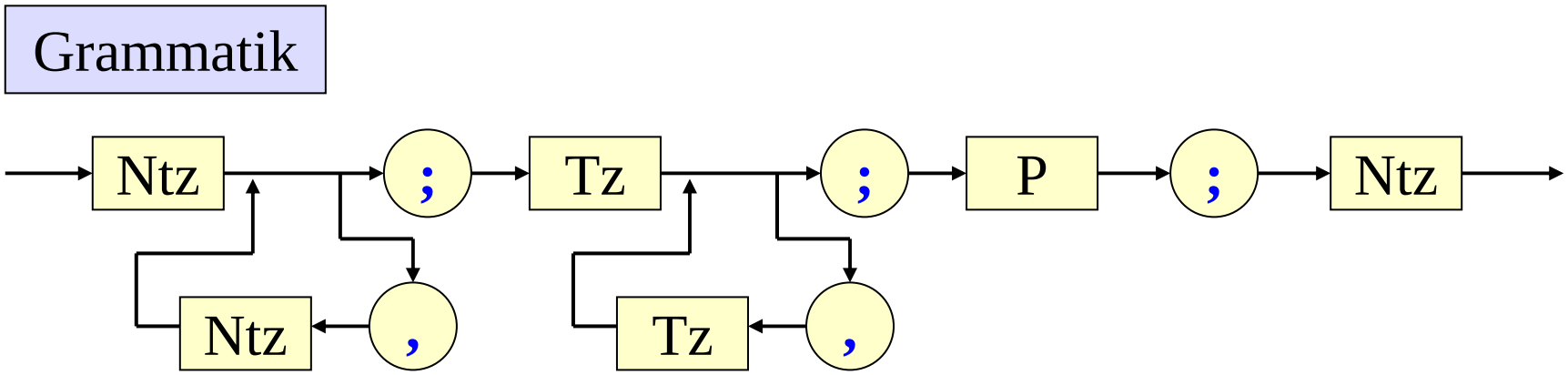
<Ntz> ::= '**N**' '**1**' {'**0**' | '**1**'}

<Tz> ::= '**T**' '**1**' {'**0**' | '**1**'}

<P> ::= { '**(**' <Ntz> ',' <rSeite> '**)**' }

<rSeite> ::= { <Ntz> | <Tz> }

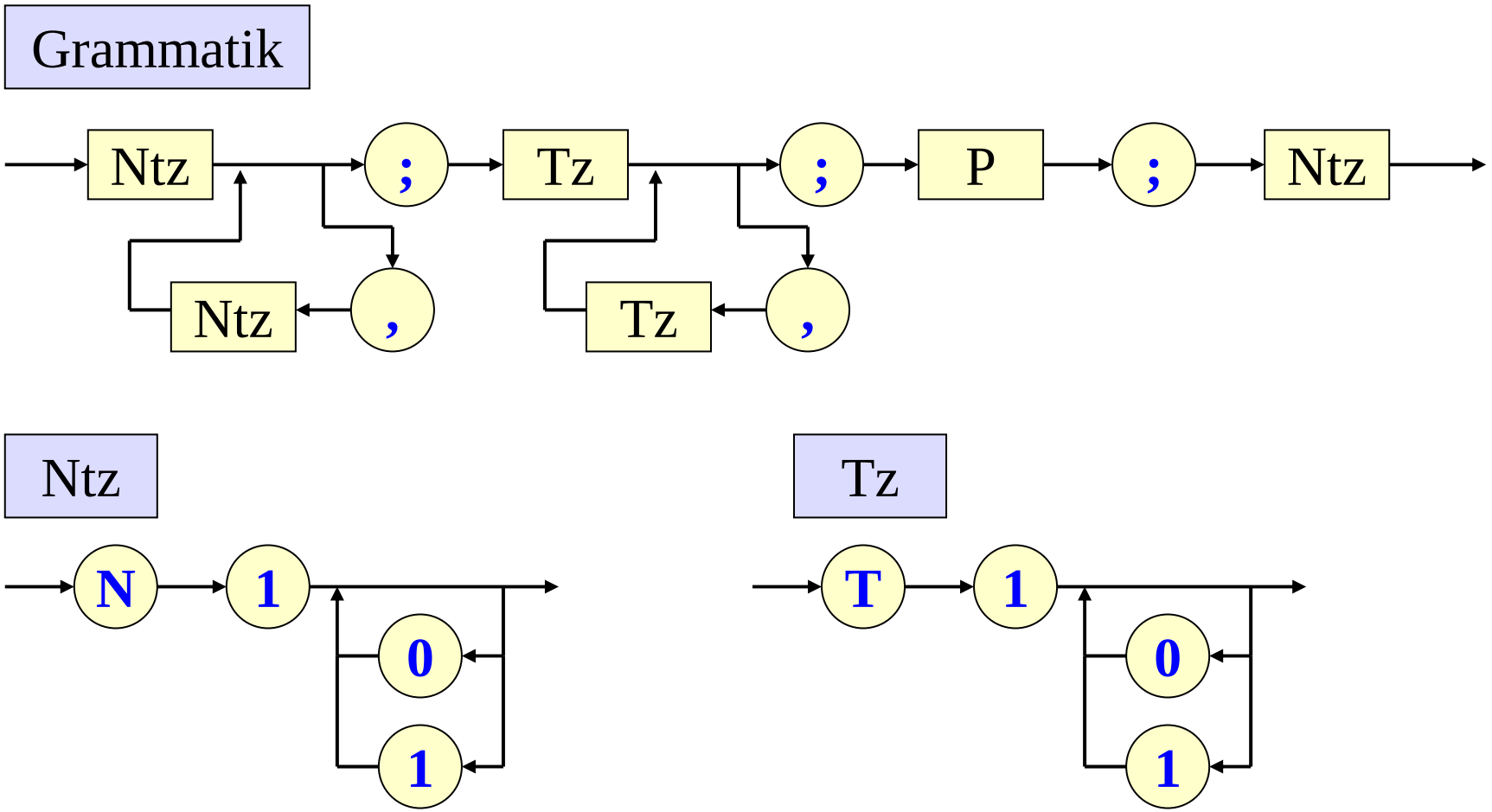
Syntaxdiagramme hierzu zeichnen:



<Grammatik> ::=

<Ntz> {',' <Ntz>} ';' <Tz> {',' <Tz>} ';' <P> ';' <Ntz>

Syntaxdiagramme hierzu zeichnen:

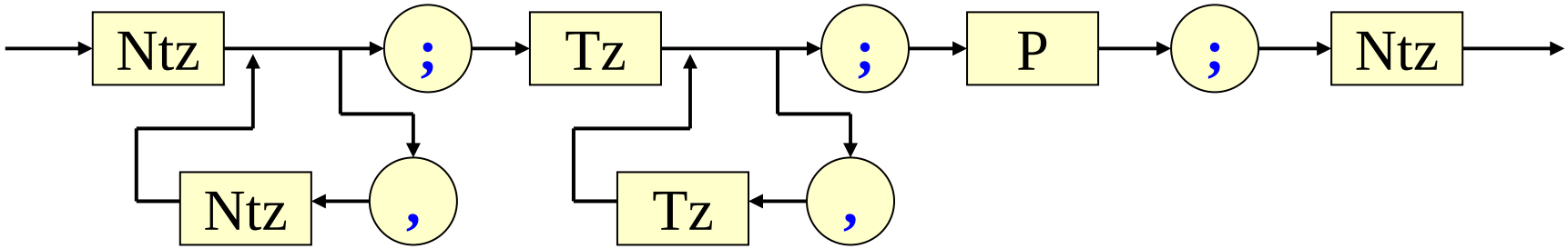


<Ntz> ::= '**N**' '**1**' {'**0**' | '**1**'}

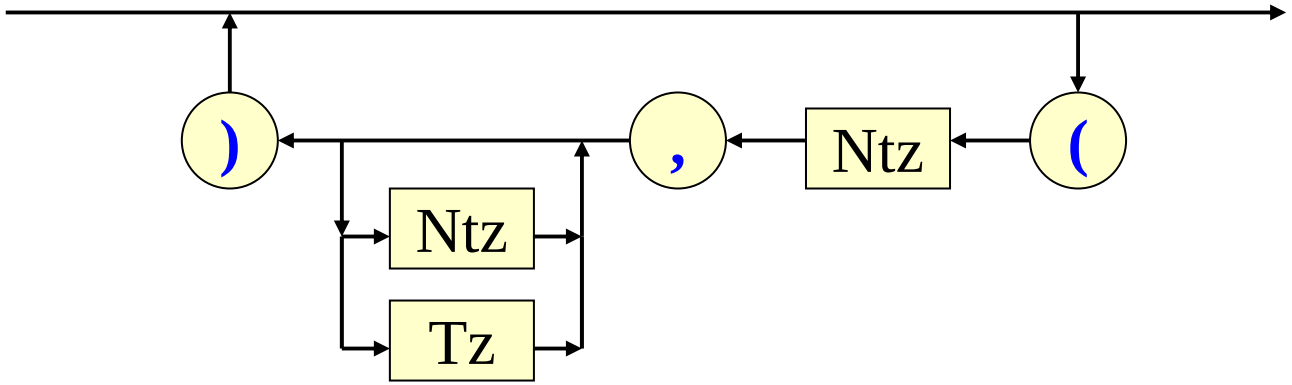
<Tz> ::= '**T**' '**1**' {'**0**' | '**1**'}

Syntaxdiagramme hierzu: (<rSeite> kann man offenbar sparen)

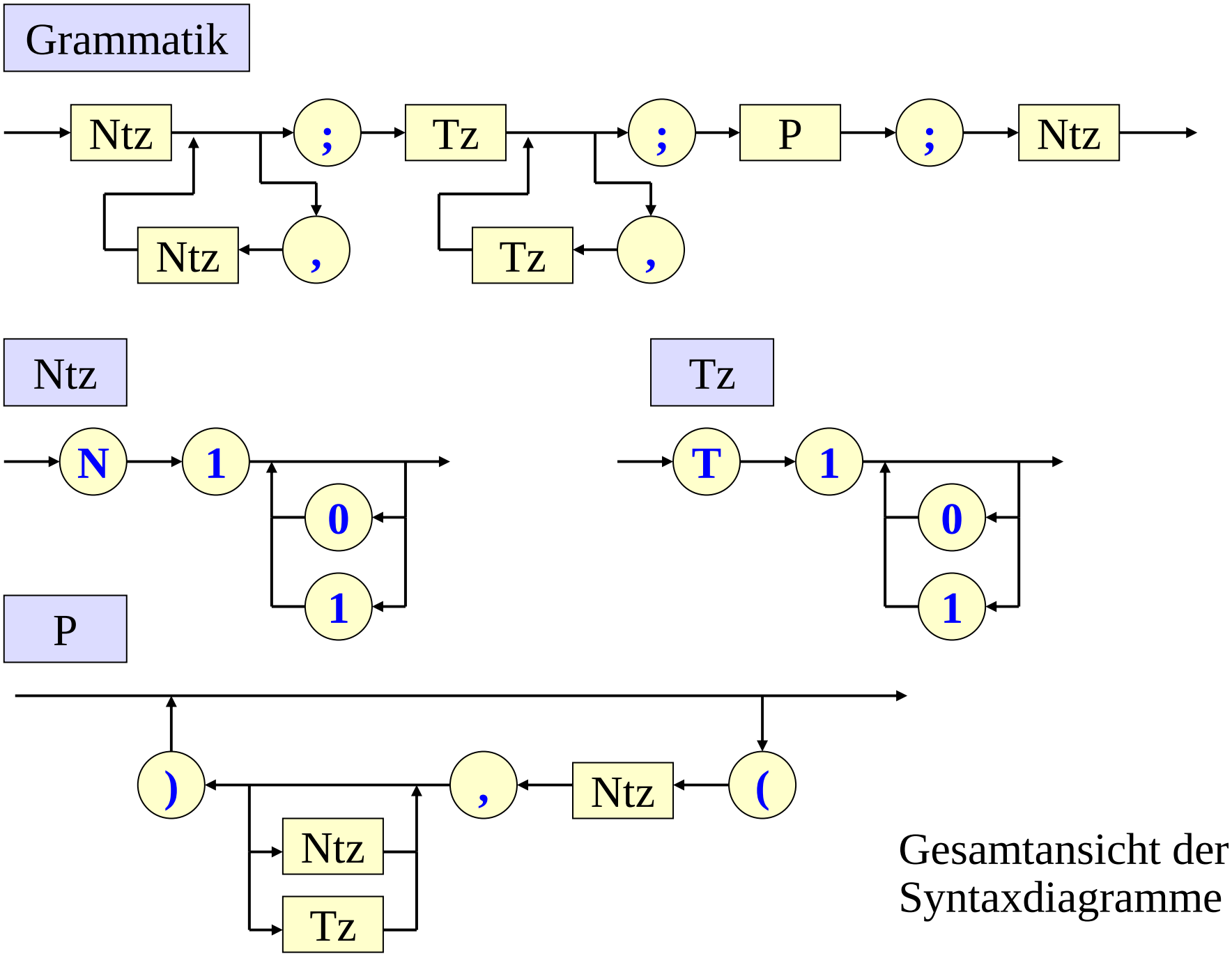
Grammatik



P



<P> ::= { '(' <Ntz> ',' <rSeite> ')' } <rSeite> ::= { <Ntz> | <Tz> }



Gesamtansicht der Syntaxdiagramme

Nun erzeugen wir ein Wort u mit dieser EBNF, z.B.:

N1,N11;T10,T101;(N1,N1N11)(N1,T10)(N11,T101);N1 $\in \Sigma_G^*$

Zugehörige Grammatik: $G_u = (V_u, \Sigma_u, P_u, S_u)$ mit $V_u = \{$**N1**, **N11**$\}$, $\Sigma_u = \{$**T10**, **T101**$\}$ und $P_u = \{$**N1**$\rightarrow$**N1N11**, **N1**$\rightarrow$**T10**, **N11**$\rightarrow$**T101**$\}$.
Die erzeugte Sprache lautet $L(G_u) = \{$**T10**$\}\{$**T101**$\}^*$.

Indem man die Zeichen nach irgendeiner Vorschrift umcodiert (z.B.: **N1** $\leftrightarrow$ S, **N11** $\leftrightarrow$ Y, **T10** $\leftrightarrow$ a, **T101** $\leftrightarrow$ b), gewinnt man aus G_u eine gleichwertige, besser lesbare Grammatik G, die bis auf Umbenennung genau dem oben abgeleiteten Wort u entspricht:

$G = (V, \Sigma, P, S)$ mit $V = \{S, Y\}$, $\Sigma = \{a, b\}$ und
$P = \{ S \rightarrow SY,\ S \rightarrow a,\ Y \rightarrow b \}$. Die erzeugte Sprache lautet
$L(G) = \{a\}\{b\}^* = \{a\, b^k \mid k \geq 0\}$.

■

Hinweis 1: Beliebige Grammatiken lassen sich auf die gleiche Weise beschreiben; man muss nur als "linke Seite" der Regeln eine nicht-leere Folge von Nichtterminalzeichen zulassen.

Hinweis 2: Manches lässt sich nicht mit einer EBNF darstellen, nämlich alle "Kontext bezogenen" Bedingungen. Insbesondere:

- Alle Nichtterminalzeichen müssen paarweise verschieden sein.
- Alle Terminalzeichen müssen paarweise verschieden sein.
- Alle Zeichen, die in den Regeln vorkommen, müssen in den Auflistungen der Terminal- bzw. Nichtterminalzeichen vorkommen.

Hinweis 3: Über die Eindeutigkeit der dargestellten Grammatik kann man hier keine Aussage machen. Sie muss i. A. für jede Grammatik einzeln bewiesen werden.

Hinweis 4: Bei der Definition von Programmiersprachen setzt man bei der Semantik gerne ein schrittweises Vorgehen ein: Zuerst definiert man eine sehr kleine Sprache, wie wir es z.B. mit den Forderungen (A1) bis (A9) in Kapitel 1 getan haben. Ist die Bedeutung dieser "Kern"-Sprache bekannt, so erweitert man sie und führt die neuen Sprachelemente auf die der Kernsprache zurück. Z.B. kann man auf diese Weise die for- und die repeat-Schleife, die zweiseitige Fallunterscheidung usw. schrittweise hinzunehmen und durch äquivalente Programmstücke beschreiben.

Im Übersetzerbau ist dieses Vorgehen als Bootstrapping bekannt, wobei man immer mächtigere Übersetzer einer Sprache in eine andere erhält. Am Ende kann man sogar einen optimierenden Übersetzer in der Sprache selbst schreiben und von einem "schlechten" Übersetzer realisieren lassen.

Bemerkungen zur Definition von Programmiersprachen

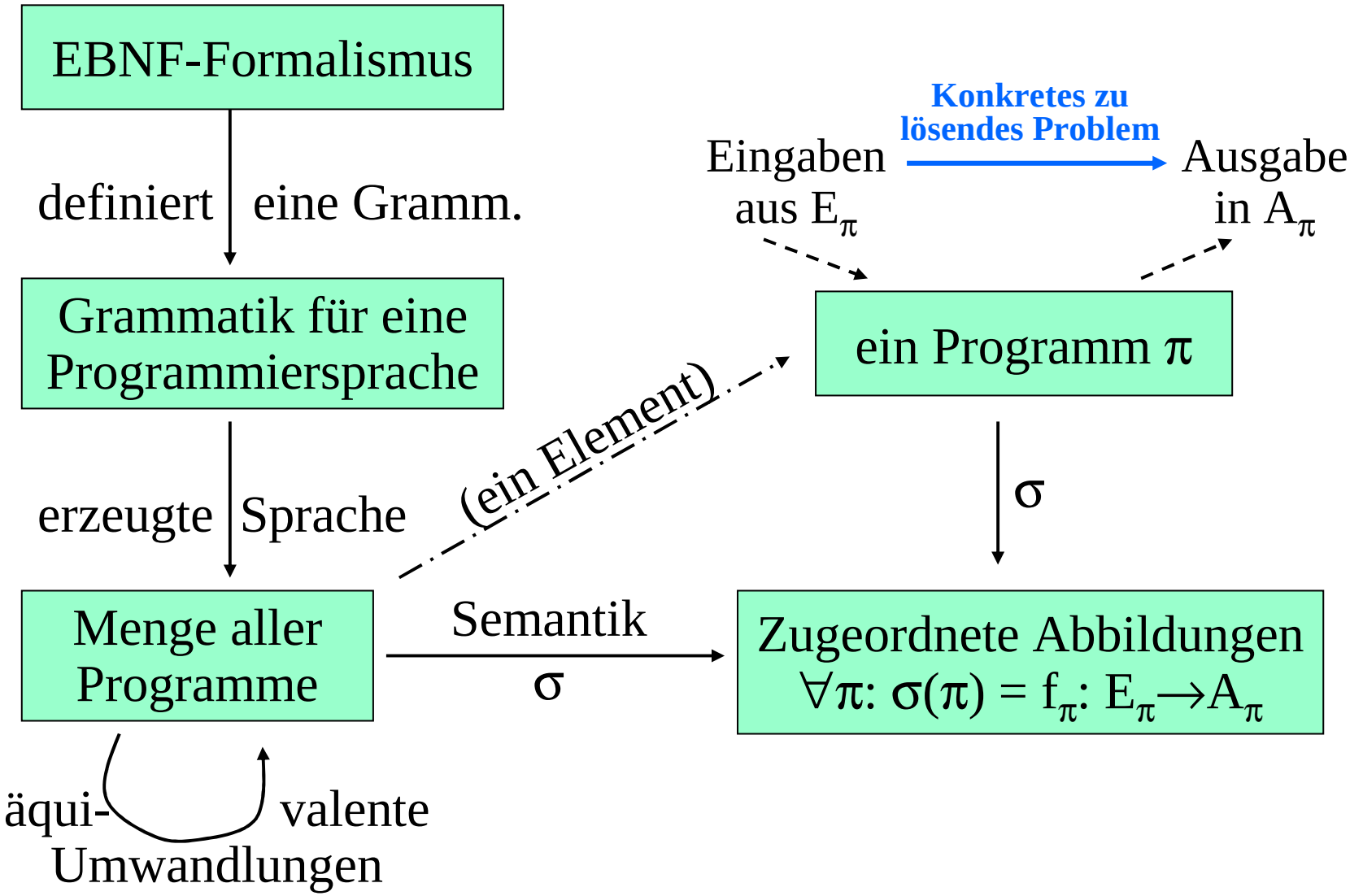
Die Syntax wird durch eine kontextfreie Grammatik bzw. eine EBNF definiert, der man Zusatzbedingungen umgangssprachlich hinzufügt (vgl. Hinweis 2). In der Regel lässt sich aus der EBNF ein Parser *automatisch* erzeugen.

Die Semantik wird meist anhand vieler Beispiele erläutert. Es gibt jedoch auch formale Methoden, um einem Programm die realisierte Abbildung zuzuordnen (Stichwörter: denotationale Semantik, axiomatische Semantik).

Bei der Definition geht man meist schrittweise vor, wobei man die Sprache ständig um neue Sprachelemente anreichert (Hinweis 4 "Bootstrapping").

Die Eindeutigkeit der EBNF einer Programmiersprache muss getrennt nachgewiesen werden (diese Eigenschaft ist "unentscheidbar", siehe später).

Skizze zu Syntax und Semantik:



3.9 Historische Anmerkungen zu Kapitel 1 bis 3

Zum Begriff Algorithmus: Hiermit verbindet man das griechische Wort arithmós (Zahl), vor allem aber den arabisch-persischen Mathematiker Mohamad Ibn Musa Al-Chwarismi; dieser lebte von 780 bis ca. 850 n.Chr., stammte aus der Region südöstlich des Kaspischen Meeres, arbeitete am Hof des Kalifen von Bagdad und hat neben anderen das "Kitab al-jabr w'al-muqabala" (das Buch über die "Regeln der Wiedereinsetzung und Reduktion") geschrieben; es behandelt die Lösungen von linearen und quadratischen Gleichungen; im Titel tritt das hier mit "Wiedereinsetzung" übersetzte Wort "algebra" erstmals in der Mathematik auf. In der lateinischen Fassung wird der Autor mit "Algorithmi" bezeichnet. Hieraus entstand das Wort Algorithmus als Begriff für "exaktes Rechenverfahren".

Algorithmen sind als mathematische Lösungsverfahren recht alt. Der Euklidische Algorithmus zur Berechnung des größten gemeinsamen Teilers natürlicher Zahlen

while b≠0 do r := a mod b, a := b; b := r od

stammt aus der Zeit um 300 v. Chr. Mit dem Sieb des Eratosthenes kann man seit ca. 230 v. Chr. die Primzahlen bis zu einer Zahl n ermitteln. Das Newtonsche Verfahren zur Berechnung einer einfachen Nullstelle von stetig differenzierbaren Funktionen f (wähle einen Anfangswert für x und die Genauigkeit δ geeignet)

while |f(x)| > δ do x := x - f(x) / f '(x) od

wird seit ca. 1670 verwendet. Das Gaußsche Eliminationsverfahren zur Lösung linearer Gleichungssysteme wird seit Anfang des 19. Jahrhunderts eingesetzt.

Die formale Definition für "Algorithmus " erfolgte 1936 unabhängig voneinander in drei Arbeiten, und zwar über den Lambda-Kalkül (dieser λ-Kalkül bildet die Grundlage der funktionalen Programmierung) von Alanzo *Church* (1903-1995), über μ-rekursive Funktionen (dies sind einfache Programme mit den Konstruktoren ";", "if-then" und "while") von Steven Cole *Kleene* (1909-1994) und über eine auf Zeichen arbeitende Maschine (die Turing-maschine) von Alan Mathison *Turing* (1912-1954).

Ab nun wurde es möglich, Aussagen über Algorithmen herzuleiten und "Unmöglichkeitsbeweise" zu führen. In der Folgezeit wurden weitere Kalküle als gleichwertig zu den drei oben genannten Darstellungen für Algorithmen nachgewiesen.

Zur Darstellung der Grundbereiche: Für Boolean und character sind sie recht alt (vor dem griechischen Alphabet plus Satzzeichen gab es bereits bei den Phöniziern und bei den Ägyptern vor 1300 v. Chr. ein buchstabenorientiertes Alphabet). Der ASCII-Code wurde Anfang der 1950er Jahre fixiert.

Das dezimale Stellenwertsystem für natürliche Zahlen hat sich ab dem 7. Jahrhundert in Indien entwickelt und gelangte durch die Araber bis Ende des 12. Jahrhunderts nach Europa. Das mathematische Standardwerk "liber abbaci", das "Buch der Rechenkunst" von Leornardo Pisano (bekannt als "Fibonacci") aus dem Jahre 1202, verwendete und verbreitete dieses System im Westen. Hier traten erstmals negative Zahlen auf. Somit gibt es bereits seit 700 Jahren die uns geläufige Darstellung für den Datentyp integer.

Erweitert man das Dezimalsystem auf Brüche, so erhält man die Darstellung des Datentyps real. Diese Erweiterung nahm erstmals Al-Kasi, Direktor der Sternwarte von Samarkant, 1427 vor.

Gottfried Wilhelm Leibniz beschreibt 1697 das Binärsystem, das vieles vereinfacht. Aber erst nach 1930 werden diese Ideen technisch für den Bau von Rechenmaschinen umgesetzt.

Die Darstellung im Zweierkomplement (vgl. Java-Kurs) erfolgte mit der Entwicklung digitaler Rechenautomaten in den 1940er Jahren.

Zur Programmierung: Der englische Mathematiker *Charles Babbage* (1792-1871) entwirft ab 1838 die "Analytical Engine", eine modern anmutende Rechenmaschine mit Steuerwerk, Rechenwerk und Programmspeicher. Seine Assistentin ist *Ada Augusta countess of Lovelace* (1815-1852, Tochter von Lord Byron), die in Ergänzung eines von ihr übersetzten Artikels die Verwendung von Programmen anregt und hierfür erste Programme schreibt. So wurde sie zur "ersten Programmiererin". Alle diese Arbeiten geraten jedoch in Vergessenheit und werden erst nach dem Bau der ersten Computer wieder entdeckt.

Die Programmierung beginnt ab 1940 mit Abfolgen von Maschinenbefehlen, zwischen 1956 und 1961 mit Programmen in den Programmiersprachen FORTRAN, ALGOL, LISP, APL und COBOL. Das Programmieren wird ein Kerngebiet für die neue Wissenschaft "Informatik".

Zu Grammatiken und BNF: Die erste Arbeit über formale Grammatiken stammt 1959 von Noam Chomsky. Hierin sind die Typ 0, 1, 2 und 3 Grammatiken, deren Sprachen und diverse Eigenschaften beschrieben. Es folgen viele Arbeiten, vor allem über kontextfreie Grammatiken. Die Backus-Naur-Form entstand im Rahmen der Entwicklung der Programmiersprache ALGOL um 1958; sie ist nach ihren wesentlichen Erfindern, dem Amerikaner J. Backus und dem Dänen P. Naur, benannt. Die Syntax der meisten heutigen Programmiersprachen wird in EBNF formuliert. Die Syntax-diagramme wurden als Veranschaulichung von BNF und kontextfreien Grammatiken in den 1960er Jahren vorgeschlagen.

3.10 Zwei Aufgaben und eine Syntax von Java 1.1

Aufgabe: *Eindeutige kontextfreie Grammatiken?*

Betrachten Sie folgende Grammatiken $G_i = (V, \Sigma, P_i, S)$ mit $V = \{S, A, B\}$ und $\Sigma = \{0, 1, 2, 3\}$. Welche sind eindeutig? Begründen Sie Ihr "ja" oder geben Sie ein mehrdeutiges Wort an.

$P_1 = \{ S \to 0, S \to A, A \to 0 \}$

$P_2 = \{ S \to 1, S \to S \}$

$P_3 = \{ S \to 0SS, S \to 1 \}$

$P_4 = \{ S \to S1, S \to 0, A \to AA, A \to 1, A \to AS \}$

$P_5 = \{ S \to 00B01S11, S \to 00B01S10S11, S \to 2, B \to 3 \}$

$P_6 = \{ S \to A0S, S \to A, A \to B1A, A \to B, B \to 2S2, B \to 3 \}$

$P_7 = \{ S \to AB, A \to 0A1, A \to \varepsilon, B \to 0B, B \to \varepsilon \}$

$P_8 = \{ S \to A1B, S \to B1A, A \to 0A0, A \to 1, B \to 0B, B \to 0 \}$

$P_9 = \{ S \to AS, A \to 1B, B \to 0S, B \to 1AS, S \to 0B, S \to 2 \}$

Anmerkung: P_5 hat etwas mit der ein- und zweiseitigen Fallunterscheidung, P_6 etwas mit arithmetischen Ausdrücken zu tun.

Aufgabe: *Umwandeln, Gleichwertigkeit*

Geben Sie zu folgenden kontextfreien Grammatiken $G_i = (V, \Sigma, P_i, S)$ mit $V = \{S, A, B\}$ und $\Sigma = \{0, 1, 2, 3\}$ möglichst einfache Syntaxdiagramme an:

$P_1 = \{ S \rightarrow 0SS,\ S \rightarrow 1 \}$

$P_2 = \{ S \rightarrow AA,\ S \rightarrow 0,\ A \rightarrow AA,\ A \rightarrow 1,\ A \rightarrow AS \}$

$P_3 = \{ S \rightarrow 0B1S3,\ S \rightarrow 0B1S2S3,\ S \rightarrow 00,\ B \rightarrow 11 \}$

$P_4 = \{ S \rightarrow A0S,\ S \rightarrow A,\ A \rightarrow B1A,\ A \rightarrow B,\ B \rightarrow 2S2,\ B \rightarrow 3 \}$

$P_5 = \{ S \rightarrow ABS,\ A \rightarrow 0A1,\ A \rightarrow \varepsilon,\ B \rightarrow 0B,\ B \rightarrow \varepsilon,\ S \rightarrow 2 \}$

$P_6 = \{ S \rightarrow A1B,\ S \rightarrow B1A,\ A \rightarrow 0A0,\ A \rightarrow 1,\ B \rightarrow 0B,\ B \rightarrow 0 \}$

Anwendung des 3. Kapitels: Syntax für Java 1.1

Beachten Sie: Es gibt hier keine Garantie für Fehlerfreiheit!

Quelle unserer Regeln im Netz: http://cui.unige.ch/db-research/Enseignement/analyseinfo/JAVA/AJAVA.html (vermutlich von 1996).
Eckige Klammern = ein- oder keinmal. Geschweifte Klammern = beliebig oft (inkl. keinmal). Runde Klammern dienen zum Zusammenfassen. Anführungszeichen klammern Terminalzeichen ein. Terminalzeichen sind zusätzlich in blau geschrieben. Kursives ist selbsterklärend (?). Nichtterminalzeichen stehen hier nicht in spitzen Klammern. Den Abschluss jeder Regel bildet ein Punkt.

Eine andere Quelle "The syntax of Java in Backus-Naur form" im Netz siehe: http://www.math.grin.edu/~stone/courses/languages/Java-syntax.xhtml
Jene Grammatik ist anders aufgebaut und orientiert sich am Buch "The Java language specification" (J.Gosling, B. Joy, G. Steele; Addison-Wesley, 1996, 433 - 453).
Jede Quelle liefert eine unterschiedliche Syntax, bedingt auch durch die Java-Versionen. Aktuell ist derzeit Java 1.4. Die folgenden Regeln gehören noch zu Java 1.1.
Es geht hier auch nur um das Prinzip. Die von Jahr zu Jahr sich ändernde aktuelle Syntaxdefinition lesen Sie bitte in einem Lehrbuch nach.

compilation_unit ::= [package_statement] { import_statement } { type_declaration } .
package_statement ::= "package" package_name ";" .
import_statement ::= "import" ((package_name "." "*" ";") | (class_name | interface_name)) ";" .
type_declaration ::= [d_comment] (class_declaration | interface_declaration) ";" .
d_comment ::= "/**" *Folge von Zeichen ohne */* "*/" .
class_declaration ::= {modifier} "class" identifier ["extends" class_name]
["implements" interface_name { "," interface_name }] "{" {field_declaration} "}" .
interface_declaration ::= { modifier } "interface" identifier
["extends" interface_name { "," interface_name }] "{" { field_declaration } "}" .
field_declaration ::= ([d_comment] (method_declaration | constructor_declaration | variable_declaration)) | static_initializer | ";" .
method_declaration ::= { modifier } type identifier "(" [parameter_list] ")" { "[" "]" } (statement_block | ";") .
constructor_declaration ::= { modifier } identifier "(" [parameter_list] ")" statement_block .
statement_block ::= "{" { statement } "}" .
variable_declaration ::= { modifier } type variable_declarator { "," variable_declarator } ";" .
variable_declarator ::= identifier { "[" "]" } ["=" variable_initializer] .
variable_initializer ::= expression | ("{" [variable_initializer { "," variable_initializer } [","]] "}") .
static_initializer ::= "static" statement_block .
parameter_list ::= parameter { "," parameter } .
parameter ::= type identifier { "[" "]" } .
statement ::= variable_declaration | (expression ";") | statement_block | if_statement | do_statement | while_statement
| for_statement | try_statement | switch_statement | ("synchronized" "(" expression ")" statement) | ("return" [expression] ";")
| ("throw" expression ";") | (identifier ":" statement) | ("break" [identifier] ";") | ("continue" [identifier] ";") | ";" .
if_statement ::= "if" "(" expression ")" statement ["else" statement] .
do_statement ::= "do" statement "while" "(" expression ")" ";" .

```
while_statement ::= "while"  "(" expression  ")" statement  .
for_statement ::= "for"  "("  ( variable_declaration  | ( expression  ";"  ) | ";"  ) [ expression ]  ";"  [ expression ]  ";"  ")" statement  .
try_statement ::= "try" statement  { "catch"  "(" parameter  ")" statement } [ "finally" statement ]  .
switch_statement ::= "switch"  "(" expression  ")"  "{"  { (  "case" expression  ":"  ) | (  "default"  ":"  ) | statement }  "}"  .
expression  ::= numeric_expression  | testing_expression  | logical_expression  | string_expression  | bit_expression  | casting_expression
                | creating_expression  | literal_expression  | "null"  | "super"  | "this"  | identifier  | (  "(" expression  ")"  )
                | ( expression  (  (  "("  [ arglist ]  ")"  ) | (  "[" expression  "]"  )
                                | (  "." expression  ) | (  "," expression  ) | (  "instanceof"  ( class_name | interface_name ) ) ) ) .
numeric_expression  ::= (  (  "-"  | "++"  | "--"  )  expression )  |  ( expression  (  "++"  | "--"  ) )
                | ( expression  (  "+"  | "+="  | "-"  | "-="  | "*"  | "*="  | "/"  | "/="  | "%"  | "%="  )  expression  ) .
testing_expression  ::= expression  (  ">"  | "<"  | ">="  | "<="  | "=="  | "!="  )  expression  .
logical_expression  ::= ( expression  (  "ampersand"  | "ampersand="  | "|"  | "|="  | "^"  | "^="
                                                | ( "ampersand"  "ampersand" ) | "||="  | "%"  | "%="  ) expression )
                | (  "!" expression ) | ( expression  "?" expression  ":" expression ) | "true"  | "false"  .
string_expression  ::= ( expression  (  "+"  | "+="  )  expression  ) .
bit_expression  ::= (  "~" expression ) | ( expression  (  ">>="  | "<<"  | ">>"  | ">>>"  )  expression  ) .
casting_expression  ::= "(" type  ")" expression  .
creating_expression ::= "new" ( ( classe_name "(" [ arglist ] ")" ) | ( type_specifier [ "[" expression "]" ] { "[" "]" } ) | ( "(" expression ")" ) ) .
literal_expression  ::= integer_literal | float_literal | string | character  .
arglist  ::= expression  { "," expression } .
type  ::= type_specifier  { "["  "]" } .
type_specifier ::= "boolean"  | "byte"  | "char"  | "short"  | "int"  | "float"  | "long"  | "double"  | class_name  | interface_name  .
modifier ::= "public" | "private" | "protected" | "static" | "final" | "native" | "synchronized" | "abstract" | "threadsafe" | "transient"  .
package_name ::= identifier  | ( package_name  "." identifier )  .
class_name  ::= identifier  | ( package_name  "." identifier )  .
interface_name ::= identifier  | ( package_name  "." identifier )  .
integer_literal ::= (  (  "1..9" {"0..9"} )  | {"0..7"} | (  "0"  "x"  "0..9a..f"  {"0..9a..f"} )  )  [  "l"  ]  .
float_literal  ::= ( decimal_digits  "."  [ decimal_digits ]  [ exponent_part ]  [ float_type_suffix ]  )
                | ( "." decimal_digits [ exponent_part ] [ float_type_suffix ] ) | ( decimal_digits [ exponent_part ] [ float_type_suffix ] ) .
decimal_digits  ::= "0..9"  { "0..9" }  .
exponent_part  ::= "e"  [  "+"  | "-"  ] decimal_digits  .
float_type_suffix  ::= "f"  | "d"  .
character  ::= " based on the unicode character set "  .
string  ::= """  {character}  """  .
identifier  ::= "a..z,$,_"  { "a..z,$,_,0..9, unicode character over 00C0" }  .
```

(Als Vereinigungsmenge von Intervallen zu lesen.)

4. Datentypen

4.1 Konkrete Datentypen

Erinnerung (1.4): Eine oder mehrere Mengen zusammen mit hierauf definierten Operationen nennt man einen (konkreten) Datentyp.

Mengen mit den auf ihnen definierten Operationen nennt man in der Mathematik eine (konkrete) Algebra. Statt einer konkreten Menge untersucht man meist alle Algebren, die gegebene Axiome erfüllen (Gruppen, Körper, ...).

Die Mathematik untersucht oft Eigenschaften, die aus Axiomen folgen, z.B. aus den Axiomen, dass die Addition kommutativ [für alle a und b gilt a+b=b+a] und dass die Multiplikation assoziativ [$a*(b*c)=(a*b)*c$] sind. Zugleich sucht sie Mengen und Strukturen, die diese Axiome erfüllen.
Beispiele: endliche Gruppen (diese sind mittlerweile kategorisiert) oder endliche nicht-kommutative Körper (es konnte gezeigt werden, dass es solche Körper nicht gibt).

Beispiele für (konkrete) Algebren sind:

Die natürlichen Zahlen $\mathbb{N}_0$ mit der Addition: $(\mathbb{N}_0;+)$.
Die natürlichen Zahlen $\mathbb{N}_0$ mit der Addition und dem neutralen Element "0" bzgl. der Addition: $(\mathbb{N}_0;+,0)$.
Die natürlichen Zahlen $\mathbb{N}_0$ mit der Multiplikation "$*$" und dem neutralen Element "1" bzgl. der Multiplikation: $(\mathbb{N}_0;*,1)$.
Die reellen Zahlen mit den Operationen +, -, $*$ und / und den neutralen Elementen "0.0" (bzgl. +) und "1.0" (bzgl. $*$): $(\mathbb{R};+,-,0.0,*,/,1.0)$.
Die Wahrheitswerte $\mathbb{B}$ = {false, true} mit den Operationen "and", "or" und "not": $(\mathbb{B};\wedge,\vee,\neg)$.
Die reellen Zahlen mit den Operationen +, -, $*$ und / und den neutralen Elementen "0.0" (bzgl. der Addition) und "1.0" (bzgl. der Multiplikation) und den Vergleichsoperationen =, < und >, wobei diese von $\mathbb{R}^2$ nach $\mathbb{B}$ führen: $(\mathbb{R},\mathbb{B};+,-,0.0,*,/,1.0,=,<,>)$.
Die Menge der reellwertigen (n×n)-Matrizen mit der Addition, der Subtraktion, der Multiplikation und den neutralen Elementen "Nullmatrix" 0 und "Einheitsmatrix" E_n: $(\mathbb{R}^{n,n};+,-,0,*,E_n)$.
Die komplexen Zahlen $\mathbb{C}$, dargestellt als zweidimensionale Ebene $\mathbb{R}^2$ mit der Addition $(a_1,b_1)+(a_2,b_2)=(a_1+a_2,b_1+b_2)$ und der Multiplikation $(a_1,b_1)*(a_2,b_2)=(a_1*a_2-b_1*b_2,a_1*b_2+b_1*a_2)$, der Wurzel i aus (-1.0,0.0), dem Betrag $|\cdot|:\mathbb{C}\rightarrow\mathbb{R}$, der Gleichheit usw.: $(\mathbb{R}^2,\mathbb{B};+,-,(0.0,0.0),*,/,(1.0,0.0),i,=,|\cdot|)$.

"Abstrakte" Algebren werden durch Symbole für Mengen und darauf definerten Operationen und durch vorgegebene Gesetzmäßigkeiten (Axiome) definiert.

Z.B.: (R;+,-,0,*,1) heißt ein "Ring", wenn + und * zweistellige Operationen auf R sind, (R;+.-,0) eine kommutative Gruppe und (R;*,1) ein Monoid bilden und die Distributivgesetze gelten.

Jede konkrete Algebra, die die Axiome einer abstrakten Algebra erfüllt, nennen wir ein Modell dieser (abstrakten) Algebra.

Z.B.: Die konkrete Algebra "ganze Zahlen" ist ein Modell der Algebra "Ring", aber sie ist kein Modell der Algebra "Körper". Die rationalen Zahlen sind ein Modell für einen Körper.

Will man allgemeine Konstruktionsprinzipien und Lösungsverfahren für möglichst viele Strukturen verwenden, so muss man sie für eine abstrakte Algebra formulieren können. Solche Beschreibungen müssen in der Informatik möglich sein.

Zur Beschreibung eines Datentyps kann man folgendes Schema verwenden (die letzte Zeile unten lässt man in der Regel weg).

Datentyp xyz:
Zugrunde liegende Mengen: ...
Nullstellige Operationen (=besondere Konstanten): ...
Einstellige Operationen: ...
Zweistellige Operationen: ...
höherstellige Operationen (sofern vorhanden): ...
Definition gewisser Operationen: ...
Beziehungen, Gesetzmäßigkeiten: ...
Dieser Datentyp lautet in der Sprache Java: ....

Datentyp integer:

Zugrunde liegende Mengen: {..., -2, -1, 0, 1, 2, 3, ...}, {false, true}.

Nullstellige Operationen: 0, 1, -1.

Einstellige Operationen: - (Negation), abs (Absolutbetrag), odd (ist die Zahl ungerade?), even (gerade?), prim (Primzahl?), sign (sign(x) $_{\text{def.}}$= if x>0 then 1 else if x<0 then -1 else 0 fi fi)

Zweistellige Operationen: +, -, $*$, div, mod, exp, =, ≠, <, ≤, >, ≥
(bei exp(a,b) = a^b muss b ≥ 0, bei a mod b muss b ≠ 0 sein, ...)

Definition von div und mod: Für alle a, b mit b≠0 gilt:
a = (a div b) $*$ b + (a mod b) und 0 ≤ a mod b < abs(b).
Zum Beispiel: 5 mod (-3) = 2 und 5 div (-3) = -1.
(-5) mod (-3) = 1 und (-5) div (-3) = 2.

Beziehungen, Gesetzmäßigkeiten:
exp(a,b_1+b_2) = exp(a,b_1) $*$ exp(a,b_2) für b_1≥0 und b_2≥0 .

Dieser Datentyp lautet in Java: int (mit folgenden Abänderungen: ...)

Einschub: Darstellung der Operationen

Hat man zwei Operationen $f, g: M \times M \rightarrow M$ auf einer Menge M, so kann man zusammengesetzte Ausdrücke ("Terme") bilden:

f(g(x, y), x) oder g(g(f(x,y),z), g(z, x))

Diese Darstellung, dass der Operator *vor* seinen Operanden steht, bezeichnet man als Prefixnotation. Man kann dann die Klammern auch weglassen:

f g x y x oder g g f x y z g z x

und die Auswertung der Ausdrücke bleibt eindeutig.

Einschub: Darstellung der Operationen

Statt dessen kann man die Operatoren auch *hinter* ihre Operanden schreiben:

x y g x f oder x y f z g z x g g

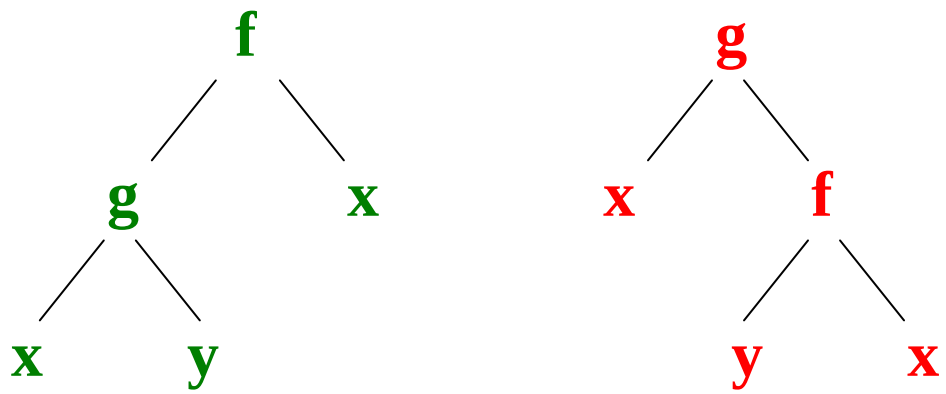
Dies nennt man Postfix-Notation (oder polnische Notation). Auch hierbei bleibt die Auswertung der Ausdrücke eindeutig.

Im täglichen Leben verwendet man jedoch in der Regel die Infix-Notation, bei der man das Operationszeichen *zwischen* die Operanden stellt; obige Beispiele werden dann zu

x g y f x oder x f y g z g z g x

Dies ist nicht mehr eindeutig, da nun z. B. f(g(x, y), x) und g(x, f(y, x)) die gleiche Infix-Notation x g y f x besitzen. Man erkennt dies, wenn man die Baumstruktur betrachtet:

Einschub: Darstellung der Operationen



Beide Bäume gehören zu
x g y f x

Um bei der Infixnotation Eindeutigkeit zu sichern, verwendet man zum einen Klammern und zum anderen Prioritätsregeln.

Daher müssen bei den Operationen fast immer auch solche Prioritätsregeln für die Operatoren angegeben werden.

Ist nichts angegeben oder haben Operatoren die gleiche Priorität, so wird ein Ausdruck stets von links nach rechts ausgewertet.

Einschub: Darstellung der Operationen

Wenden Sie diese Überlegungen auf die Eindeutigkeit von Wörtern oder Grammatiken an (3.3, Folie 180):

Stellt man die Ableitungsbäume mit der Prefixnotation dar, so sind zwei Ableitungen genau dann gleich (im Sinne von Folie 179), wenn sie die gleiche Prefixnotation besitzen.

Die Prefixnotation erhält man, indem man den (Ableitungs-) Baum von der Wurzel immer zunächst nach links zu den Blättern hin durchläuft und genau dann, wenn man einen Knoten erstmals besucht, dessen Inhalt herausschreibt.

Machen Sie sich dies an Beispielen klar. Dadurch kann man die Gleichheit von Ableitungen auch "algorithmisch" relativ leicht nachprüfen.

4.2 Abstrakte Datentypen

Wer Probleme zu lösen hat, interessiert sich erst in zweiter Linie um die Typen und deren detaillierte Implementierung. Zunächst stellt man Forderungen an die Lösungen, meist in Form von Eigenschaften. Von einem *Keller* (Stapel, Stack, Pushdown) erwartet man, dass er die Eigenschaft
"Was als letztes hingelegt wird, kommt als erstes wieder heraus"
d.h. die Gesetzmäßigkeit $Pop(Push(K, A)) = K$ erfüllt.
Von einer *Warteschlange* W erwarten wir, dass ihr erstes Element nicht durch Hinzufügen weiterer Elemente A verändert wird, d.h., es gilt:
$First(W) = First(Enter(W, A))$.
Die "Abstraktion" besteht darin, dass die Implementierung oder die Darstellung in einer Sprache nicht interessiert.

Dies führt zum "abstrakten Datentyp":

Ein abstrakter Datentyp wird durch Bezeichnungen für Mengen, durch Bezeichnungen von Abbildungen (einschl. ihrer Stelligkeiten) und durch die zu erfüllenden Gesetze charakterisiert.

Er abstrahiert somit von den konkreten Darstellungen (Datentypen, Implementierung von Unterprogrammen und anderen Programmeinheiten) und gibt nur die Gesetzmäßigkeiten an, denen die beteiligten Mengen genügen müssen. Man kann ihn mit dem Schema aus 4.1 beschreiben; wir wählen allerdings im Folgenden eine eigene Formulierung.

Mathematisch gesehen ist ein abstrakter Datentyp somit eine (abstrakte) Algebra.

Schema zur Beschreibung eines abstrakten Datentyps:

structure < Name > is
(< Liste von Parametern und benötigten Einheiten >)
modes < nicht-leere Liste von Bezeichnern >
functions < Liste von Bezeichnern mit "Stelligkeiten" >
laws < Liste von logischen Aussagen (= Gesetzmäßigkeiten) >
end structure

Man kann auch andere Schlüsselwörter verwenden, z.B.:
sorts oder sets oder types anstelle von modes,
mappings, operators oder procedures anstelle von functions,
axioms, equations oder rules anstelle von laws.
Die Auflistungen können durch Komma oder Semikolon getrennt sein und die Abbildungen post- oder preorder dargestellt werden.

In diesem Schema nennt man den Teil

modes <nicht-leere Liste von Bezeichnern>;
functions < Liste von Bezeichnern mit "Stelligkeiten" >

die Signatur des Datentyps.

Manchmal bezeichnet man auch die Signatur alleine schon als abstrakten Datentyp.

Man beachte, dass die Konstanten, die verwendet werden sollen, als nullstellige Funktionen dargestellt werden.

Zu einem abstrakten Datentyp sucht man nun konkrete Datentypen ("Modelle"), die die Signatur und die Gesetze erfüllen.

Wir betrachten als Beispiel die Wahrheitswerte "WHW1":

4.2.3: Beispiel Wahrheitswerte 1: Wir nehmen, wir benötigen von den Wahrheitswerten nur folgende Eigenschaften:

```
structure WHW1 is
modes  B                          -- dies steht für Menge der Wahrheitswerte
functions  wahr () B;
           not (B) B;
           and, or, impl (B, B) B;
           ifthenelsefi (B, B, B) B
laws       ASS: and (and (x,y), z) = and (x, and (y,z));
           KOMM: and (x, y) = and (y, x);
           NEG: not (not (x)) = x;
           IMP: impl (x, y) = or (not (x), y);
           IF1: ifthenelsefi (wahr, x, y) = x;
           IF2: ifthenelsefi (not (wahr), x, y) = y
end structure
```

Modell 1: Betrachte den abstrakten Datentyp WHW1.
Für die Sorte B wählen wir die Menge **IB** = {false, true},
wahr werde durch true und not(wahr) durch false dargestellt
und die Funktionen not, and, or, impl und ifthenelsefi
entsprechen den Funktionen Negation, Konjunktion, Disjunktion, Implikation und Alternative auf **IB** (vgl. Folie 55).
Dann sind alle Gesetze erfüllt, wie man leicht nachprüft.

Doch dieses Modell ist nicht das einzige. Vielmehr ist der abstrakte Datentyp WHW1 "vielgestaltig" oder polymorph, d.h., er hat viele Modelle. Ein weiteres Modell ist zum Beispiel:

Modell 2: Betrachte erneut den abstrakten Datentyp WHW1. Für die Sorte B wählen wir die einelementige Menge $E = \{0\}$, wahr werde durch 0 dargestellt und die Funktionen not, and, or, impl und ifthenelsefi liefern für alle Argumente den Wert 0. Dann sind ebenfalls alle Gesetze erfüllt, z.B.:

Für das Gesetz

impl (x, y) = or (not (x), y);

setzen wir alle möglichen Werte ein (hier ist nur der Wert 0 möglich):

0 = impl (0,0) = or (not (x), y) = or (0,0) = 0.

Es gibt zu WHW1 sogar unendlich viele verschiedene Modelle. Ein weiteres ist:

Modell 3: Betrachte erneut den abstrakten Datentyp WHW1.
Für die Sorte B wählen wir die ganzen Zahlen **Z**,
wahr werde durch die Konstante 1 dargestellt und die
Funktionen werden repräsentiert durch:

not(z) = - z (das Negative einer ganzen Zahl),
and(a,b) = a + b (Addition),
or(a,b) = a + b (Addition)
impl(a,b) = b - a (Subtraktion der ersten Zahl von der zweiten)
ifthenelsefi(1,b,c) = b und ifthenelsefi(a,b,c) = c für $a \neq 1$.

Dann sind ebenfalls alle Gesetze erfüllt, z.B.:
impl(x, y) = y - x = (-x) + y = or(not(x), y).

Ordnet man den Sorten Mengen und den Funktionen Abbildungen entsprechend der vorgegebenen Stelligkeiten zwischen diesen Mengen zu und sind dann alle Gesetze erfüllt, so nennt man diese Mengen mit ihren Abbildungen (also diese "konkrete Algebra")
ein Modell des abstrakten Datentyps oder einen zugehörigen konkreten Datentyp oder eine Konkretisierung oder eine Ausprägung oder eine Instanz.

Besitzt ein abstrakter Datentyp (bis auf Isomorphie) nur ein Modell, so heißt er monomorph, anderenfalls polymorph.

Kann man den abstrakten Datentyp "Wahrheitswerte" auch monomorph machen, also so formulieren, dass es im Wesentlichen nur noch *ein* Modell zu ihm gibt? Versuchen wir es mit dem Beispiel Wahrheitswerte 2 (WHW2):

```
structure WHW2 is
modes  B
functions  wahr () B;      falsch () B;          -- 5 Funktionen
           not (B) B;      and (B, B) B;
           or (B, B) B
laws       ZWEI: wahr ≠ falsch;                  -- ≥ 2 Elemente
           A1: and (wahr,x) = x;                 -- A1 bis A3
           A2: and (x,wahr) = x;                 -- legen and fest
           A3: and (falsch,falsch) = falsch;
           OR1: or (falsch,x) = x;               -- OR1 bis OR3
           OR2: or (x,falsch) = x;               -- legen or fest
           OR3: or (wahr,wahr) = wahr;
           N1: not(falsch) = wahr;               -- N1 und N2
           N2: not(wahr) = falsch                -- legen not fest
end structure
```

Setze nun die Menge **IB** = {false, true} für die Sorte B;
wahr werde durch true und falsch durch false dargestellt.
Die Funktionen not, and und or entsprechen den Funktionen Negation, Konjunktion und Disjunktion auf **IB**.
Dann sind alle Gesetze erfüllt.

Da die Funktionen not, and und or in den Gesetzen komplett wie in einer Funktionstabelle festgelegt sind, gibt es im Wesentlichen auch kein zweites hiervon verschiedenes Modell.

Aber es gibt natürlich Erweiterungen (oder Einbettungen in größere Bereiche).

Setze hierfür die Menge der ganzen Zahlen $\mathbb{Z}$ für die Sorte B; wahr werde durch 1 und falsch durch 0 dargestellt. Für die Funktionen wähle man:

not (x) = 1 - x

and (x, y) = $x \cdot y$

or (x,y) = 1 - ((1-x) $\cdot$ (1-y))

Hier wurde $\mathbb{B}$ in die Menge $\mathbb{Z}$ geeignet eingebettet: Mit den Operationen and, or und not kann man die Teilmenge $\{0,1\}$ nicht verlassen; die Operationen sind allerdings auf ganz $\mathbb{Z}$ definiert und somit haben wir ein weiteres Modell für WHW2.

Um solche erweiterten Modelle auszuschließen, kann man zwei Wege beschreiten:

a) Man fügt ein Gesetz der Form "Es gibt höchstens zwei Elemente" hinzu. In unserem Beispiel:
$x \in B \Rightarrow (x = \text{wahr} \vee x = \text{falsch})$.

b) Oder man erlaubt generell nur "erzeugbare Modelle", d.h., in den Modellen sind nur solche Mengen erlaubt, die sich aus den angegebenen Konstanten mit Hilfe der angegebenen Abbildungen erzeugen lassen.
In unserem Beispiel sind dies alle Werte, die man aus "wahr" und "falsch" mittels not, and und or erzeugen kann; man sieht, dass hierdurch keine neuen Werte hinzukommen, so dass damit nur das zweielementige Modell zugelassen ist.

4.2.4: Beispiel "längenbeschränkter Keller" (vgl. Duden Inf.)

Unter einem Keller (Stack, Stapel, Pushdown) versteht man eine Struktur, in die man Elemente nacheinander hineinlegen kann, die später in umgekehrter Reihenfolge wieder heraus genommen werden müssen. Beispiele sind der Ablagekorb, die Groschenbox oder das als Sackgasse gestaltete Rangiergleis, aber auch die Auswertung arithmetischer Ausdrücke und die Realisierung der Rekursion.

Wir wollen diesen Datentyp nun abstrakt beschreiben, ohne Hinweise zur konkreten Realisierung zu geben. Man benötigt die Menge der Elemente Δ und die Struktur "Keller" lbsΔ (= längenbeschränkter Stack mit Elementen aus Δ) sowie einige Operationen wie

push(d,x) lege d in den Keller x hinein,
pop(x) entferne das zuletzt hineingelegte Element aus x.

Bei einem Keller interessiert also nicht die Implementierung, sondern nur dieses genannte Prinzip "LIFO" = last-in-first-out, d.h., die kellerartige Speicherungstechnik. Diese lässt sich durch Gleichungen beschreiben (d ist ein Element, x ist ein Keller):

$$\text{top}(\text{push}(d, x)) = d,$$
$$\text{pop}(\text{push}(d, x)) = x.$$

Wir erweitern diesen Ansatz und fühhren den "längenbeschränkten Keller" ein, der nur bis zu "max" Elemente aufnehmen kann. "max" und die Sorte Δ der einzufügenden Elemente übergeben wir als Parameter; der Keller wird durch die Sorte lbsΔ repräsentiert. (Weiterhin mixen wir hier die Postorder-Notation bei den "functions" mit der Preorder-Notation in den Gesetzen.)

structure LBK is (based on boolean, based on natural,
mode Δ, natural max: max > 0)

modes lbsΔ

functions ( ) lbsΔ empty;
(lbsΔ) boolean isempty, isfull;
(lbsΔ) Δ top;
(lbsΔ) lbsΔ pop;
(Δ, lbsΔ) lbsΔ push;
(lbsΔ) natural length

laws LEER: (x = empty) ⇔ isempty(x);
VOLL: (length(x) = max) ⇔ isfull(x);
LIFO: not isfull(x) ⇒ pop(push(d,x)) = x;
KON: not isempty(x) ⇒ push(top(x), pop(x)) = x;
OBEN: not isfull(x) ⇒ top(push(d,x)) = d;
ANZ: not isfull(x) ⇒ length(push(d,x)) = length(x)+1;
ANZ0: length(empty) = 0

end structure

4.2.5 Beispiel: Hiermit können wir nun einen kleinen Algorithmus formulieren, nämlich das Spiegeln eines Feldes.

```
A, B: array [1..n] of character;
...
with LBK (Δ => character, max=>4); use LBK;
integer i;
lbsΔ K;                -- wobei Δ hier character sei!
begin  K := empty;
   for i := 1 to n do push(A[i],K) od;
   for i := 1 to n do B[i] := top(K); pop (K) od
end
```

4.2.6 Beispiel arithmetische Ausdrücke, die nur aus den Operatoren "+" und "*", aus den Klammern "(" und ")", aus ganzen Zahlen und Zwischenräumen bestehen.

Darstellung: Inorder. Die Priorität "mal vor plus" soll beachtet werden.

In welcher Reihenfolge werden die Operationen des Ausdrucks ausgewertet? Hierzu verwendet man zwei Keller (Stacks), einen für die Operatoren und einen für die Operanden.

Idee: Es seien zwei Operanden und der dazwischen stehende Operator gelesen. Hat der folgende Operator gleiche oder kleinere Priorität, so führe den ersten Operator aus, anderenfalls kellere ("push") auf den Stacks.

Man kann dies auch mit *einem* Stack realisiseren, auf den man die Operatoren und Zahlen ablegt (die Elemente des Stacks sind dann ein union-Datentyp, varianter Record, siehe Abschnitt 2.3, z.B. Folie 118).

Dann muss man den links stehenden (zuvor gelesenen) Operator in einer Variablen "links" notieren, um die Priorität beachten zu können.

Ein Programmstück hierfür folgt auf der nächsten Folie. *Es wird allerdings vorausgesetzt, dass der arithmetische Ausdruck korrekt aufgebaut ist.* Mit einer Erweiterung (z.B. mit einer Booleschen Variable, in der man sich merkt, ob das letzte Objekt ein Operator oder ein Operand war), lässt sich dies zusätzlich überprüfen.

```
character C, links; integer X, Y, Z; lbsΔ K; ...      -- wobei Δ hier character sei!
K := empty; push('⊥',K); links := '⊥'; read (C);
while not eof do                          -- eof bedeutet "end of file" = Ende der Eingabe
      while C = ' ' do read(C) od;
      if C = '*' then
            if links = '(' or links = '+' or links = '⊥' then Push(C,K); links := C
            else Z := top(K); pop(K); Op := top(K); pop(K); Y := top(K); pop(K);
                  links := top(K); X := Y Op Z; Push(X,K); read(C) fi
      elsif C = '+' then
            if links = '(' or links = '⊥' then Push(C,K); links := C
            else Z := top(K); pop(K); Op := top(K); pop(K); Y := top(K); pop(K);
                  links := top(K); X := Y Op Z; Push(X,K); read(C) fi
      elsif C = '(' then Push(C,K); links := C; read(C)
      elsif C = ')' then
            if links = '(' then X:=top(K); pop(K); pop(K); links:=top(K); push(X,K); read(C)
            else Z := top(K); pop(K); Op := top(K); pop(K); Y := top(K); pop(K);
                  links := top(K); X := Y Op Z; Push(X,K) fi   -- kein read(C) !
      elsif "C ist nicht eof" then
            "C ist der Beginn einer ganzen Zahl; ermittle diesen Wert als Variable X
            und speichere ihn mittels Push(X,K); read(C)" fi
od
```

X:=*Y Op Z* bedeutet: Führe den Operator Op mit den Variablen Y und Z aus:
if Op = '+' then X:=Y+Z else if Op='*' then X:=Y*Z else "Fehlerfall" fi fi;
elsif ist die Abkürzung für "else if", wobei ein fi am Ende entfällt.

4.2.7: Natürlich muss man den abstrakten Datentyp später in einem Programm realisieren (genau ausprogrammieren = implementieren). Dazu gibt es in einer Programmiersprache neben dem obigen Spezifikationsteil einen Implementierungsteil. Zum Beispiel in der Form

structure implementation <Name> is ...

Auf der folgenden Folie ist eine mögliche Darstellungsform genannt.

Wichtig ist, dass der Implementierungsteil unabhängig von der Spezifikation aufgeschrieben wird und daher später leicht gegen eine "bessere" Implementierung ausgetauscht werden kann.

Hinweis hier haben wir den Keller S nicht als Parameter, sondern als Feldvariable mit der aktuellen Länge t implementiert:

```
structure implementation LBK is
type index is 0..max;
array [1..max] of Δ  S;  index t;
procedure empty is begin t:= 0 end;
function isempty return Boolean is  begin return (t = 0) end;
function isfull return Boolean is  begin return (t = max) end;
function top return Δ is
        begin if isempty then „Fehlerabbruch“
        else top:= S[t] fi end;
procedure pop is
        begin if isempty then „Fehlerabbruch“
        else t:= t-1 fi end;
procedure push (d: Δ) is
        begin if isfull then „Fehlerabbruch“
        else t:= t+1; S[t]:= d fi end;
function length return integer is  begin return t end;
begin empty end;                                  -- Initialisierung
```

Hinweise: Hier wurde die Sorte lbsΔ zwar in der Signatur aufgeführt, aber in der Implementierung stillschweigend durch

array [1 .. max] of Δ

ersetzt. Bei der Umsetzung in eine konkrete Programmiersprache kann dieser Zusammenhang deutlich gemacht oder verschwiegen ("private") werden.

Statt mit einem Feld hätte man den Datentyp LBK auch durch eine Liste realisieren können (siehe später Kapitel 11).

In der Praxis muss auch der Zugriff auf die Sorten Δ und lbsΔ geklärt werden. In der Praxis sind diverse Varianten und weitere Möglichkeiten üblich.

4.2.8 Empfohlenes Vorgehen, um in der Praxis Programme zur Realisierung von Algorithmen zu schreiben.

Top-down-Phase:

(1) Zerlege den Algorithmus in kleinere Einheiten (in der Regel nicht mehr als sieben) und gib an, wie diese im Gesamtalgorithmus zusammenwirken. Gib für jede Einheit ihre Signatur an.

(2) Ermittle die Eigenschaften, die die Funktionen der Einheiten erfüllen müssen. Formuliere diese als Gesetze eines abstrakten Datentyps aus.

(3) Wiederhole (1) und (2) mit den Algorithmen, die von den einzelnen Einheiten realisiert werden, bis so kleine Einheiten entstanden sind, dass ihre Implementierung in der Programmiersprache leicht möglich ist.

Bottom-Up-Phase:

(4) Lege nun die Datenbereiche (möglichst in Moduln) als Typen in der gegebenen Programmiersprache fest und implementiere die zuletzt erhaltenen Algorithmen-Teile (als Prozeduren, Funktionen, Moduln, kommunizierende Einheiten oder Klassen).

(5) Wiederhole (4) schrittweise für jeden Unter-Algorithmus, dessen gesamte Daten und Operationen bereits implementiert sind, bis schließlich der Gesamtalgorithmus implementiert ist.

5. Iteration und Rekursion. Funktionen und Prozeduren.

5.1 Iteration = Wiederhole etwas k mal.

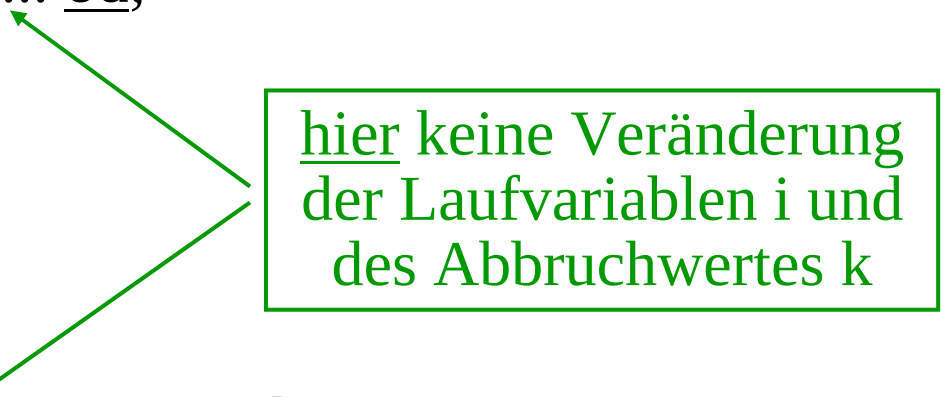
Hierbei ist k konstant oder der Wert einer Variablen.

Realisierung der Iteration:

for i:=1 to k do .... od;

oder

i := 1;
while i ≤ k do ...; i :=i+1 od;



Vgl. (A8c), Folie 35.

Solche for-Schleifen sind in der Praxis häufig. Sie erlauben meist eine gute Abschätzung der zu erwartenden Laufzeit.

Beispiel: n sei eine globale Variable

```
declare integer i, j, H; array [1..n] of integer A;
begin
  for i := 1 to n-1 do
     for j := i+1 to n do
        if A[j-1] > A[j] then H:=A[j]; A[j]:=A[j-1]; A[j-1]:=H fi
     od
  od
end;
```

Dies ist eine Variante von "Bubblesort" zur Sortierung eines Feldes. Es werden genau $(n-1) + (n-2) + ... + 2 + 1 = n \cdot (n-1)/2$ Vergleiche durchgeführt; das Verfahren benötigt also quadratischen Aufwand bezüglich der Anzahl der zu sortierenden Elemente.

5.2 Blöcke

Ein Block besteht aus einem Deklarationsteil und einem darauf folgenden Anweisungsteil.

declare <Deklarationsteil> ;
begin <Folge von Anweisungen>
end;

Der Block, in dem eine Variable (oder ein anderes Objekt) deklariert ist, ist dessen "Lebensdauer". Die Sichtbarkeit (oder der Gültigkeitsbereich) eines Namens ist der Teilbereich der Lebensdauer, in dem der Name nicht neu deklariert ist.

Eine Variable ist in dem Block, in dem er deklariert wird, lokal, in allen Unter-Blöcken ist sie global. Sie kann in einem Unter-Block neu deklariert werden, wobei dort dann eine neue Variable unter diesem Namen eingeführt wird. Das Gleiche gilt für alle anderen deklarierten Objekte.

Beispiel: Maximale Häufigkeit

Gegeben ist eine Folge $a_1, a_2, ..., a_n$ von n ganzen Zahlen ($n \geq 1$). Hierunter können mehrere Zahlen mehrfach vorkommen.

Gesucht ist eine Zahl, die am häufigsten in der Folge vorkommt, sowie ihre Häufigkeit.

Hinweis: Wir verwenden hier eine Folge von Zahlen; das Problem und seine Lösung gelten auch für jede Folge von anderen Objekten anstelle von Zahlen.

Naive Lösung: Prüfe für die erste Zahl, wie oft sie vorkommt, prüfe dies für die zweite Zahl, die dritte Zahl usw. Teste jedes Mal, ob die soeben ermittelte Häufigkeit das bisherige Maximum darstellt und notiere gegebenenfalls die neue Zahl und ihre Häufigkeit.

145 Zahlen zur Illustration: Wie lautet die maximale Häufigkeit?

2301, 4892, 8197, 7823, 6541, 2639, 7891, 6883, 9211, 6738,
3371, 10892, 4394, 13823, 11741, 2663, 4852, 3197, 7623,
7841, 6383, 10512, 6938, 4092, 8144, 7823, 6741, 2639, 7391,
6884, 9291, 6735, 5171, 10892, 4994, 13623, 12742, 2662,
4432, 3857, 5623, 10395, 2394, 1823, 1751, 2263, 4152, 3647,
7635, 7741, 6383, 1022, 6938, 4992, 8744, 4823, 6641, 7739,
5191, 6294, 4971, 7035, 6631, 11542, 4794, 1373, 15542,
2362, 4412, 3707, 5323, 5371, 4892, 4294, 1373, 11940, 2664,
4252, 3737, 7913, 7221, 6373, 11512, 6928, 4492, 2144, 7433,
6641, 12799, 7341, 6284, 9201, 4735, 5441, 10852, 4984,
12223, 11741, 2632, 2432, 3657, 5629, 10355, 4394, 1823,
1751, 7263, 4452, 6647, 8645, 7641, 6383, 1322, 3938, 4022,
8441, 4323, 6941, 7832, 5121, 6354, 4931, 7235, 6431, 9542,
1794, 3273, 4542, 2662, 4812, 2707, 8323, 6484, 9251, 3795,
5071, 6362, 4812, 2747, 5422, 5371, 1592, 4294, 2723, 6242.

Programmiersprachliche Umsetzung:

Variablen: Man benötigt eine Variable N für die Zahl n und n Variablen A[1], ..., A[n] für die Folge der Zahlen. Sodann muss man sich die aktuell zu prüfende Zahl und ihre zu ermittelnde Häufigkeit in zwei Variablen Aktuell und Häufig merken, und man braucht zwei Variablen MaxZahl und MaxHäufig, in denen man die bisher am häufigsten vorkommende Zahl und deren Häufigkeit notiert.

Vorbereiten der Variablen: Als erstes muss n eingelesen werden. Danach muss man die Variablen A[1], ..., A[n] deklarieren und ihre Werte einlesen. Dann sind die restlichen Variablen anzulegen und eventuell zu initialisieren (= mit Anfangswerten zu belegen). Dann kann der Lösungsalgorithmus beginnen. Wir realisieren dieses Vorgehen mit Hilfe von drei Blöcken.

Datenbereiche und Blockstruktur:

```
program MaxHäufigkeit is
declare positive N;                                    --  n > 0
begin
   read(N);
   declare  array [1..N] of integer  A;
   begin
      for i := 1 to N do read(A[i]) od;
      declare integer Aktuell, MaxZahl;
         natural Häufig, MaxHäufig;
      begin
         MaxZahl := A[1]; MaxHäufig := 1;
            -- Hier folgen die Anweisungen des Lösungsalgorithmus
      end
   end;  ...          -- Arbeite hier mit den gewonnenen Werten
end;
```

Datenbereiche und Blockstruktur:

```
program MaxHäufigkeit is
declare positive N;                                   --  n > 0
begin
   read(N);
   declare  array [1..N] of integer  A;
   begin
      for i := 1 to N do read(A[i]) od;
      declare integer Aktuell, MaxZahl;
         natural Häufig, MaxHäufig;
      begin
         MaxZahl := A[1]; MaxHäufig := 1;
            -- Hier folgen die Anweisungen des Lösungsalgorithmus
      end
   end;  ...          -- Arbeite hier mit den gewonnenen Werten
end;
```

Datenbereiche und Blockstruktur:

3 ineinander geschachtelte Programmteile: "Blöcke"

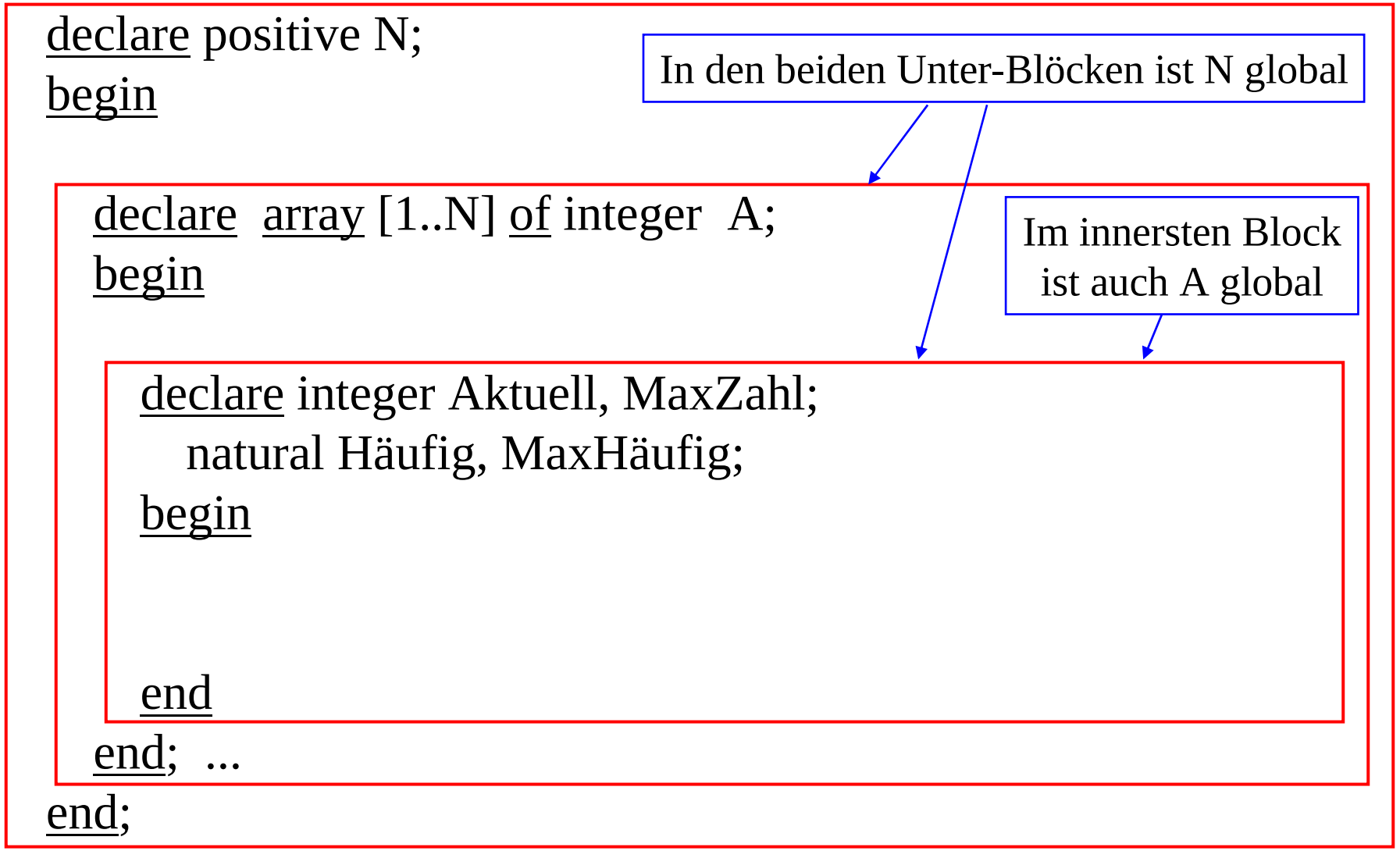
```
declare positive N;
begin

   declare  array [1..N] of integer  A;
   begin

      declare integer Aktuell, MaxZahl;
         natural Häufig, MaxHäufig;
      begin


      end
   end;  ...
end;
```

Globale und lokale Variable:



Kontrollstruktur zur Lösung:

```
-- Die Anweisungen im innersten Block lauten
    begin      -- Beginn des Anweisungsteils des innersten Blocks.
               -- Initialisiere zunächst:
      MaxZahl := A[1]; MaxHäufig := 1;
      for i := 1 to N do                    -- Prüfe für jede Zahl
        Aktuell := A[i]; Häufig := 0;       -- Initialisiere
        for j := 1 to N do                  -- Vergleiche mit jeder Zahl
          if A[j] = Aktuell then Häufig := Häufig + 1 fi
        od;
        if Häufig > MaxHäufig then          -- Neues Maximum gefunden?
          MaxZahl := Aktuell; MaxHäufig := Häufig fi
      od;
      write(MaxZahl); write(MaxHäufig);
    end;       -- Ende des innersten Blocks.
               -- Dieses Programmstück oben einfügen! Dies ergibt:
```

Gesamtprogramm zur Ausgabe der maximalen Häufigkeit:

```
program MaxHäufigkeit is
declare positive N;
begin
    read(N);
    declare  array [1..N] of integer A;
    begin
        for i := 1 to N do read(A[i]) od;
        declare integer Aktuell, MaxZahl;
            natural  Häufig, MaxHäufig;
        begin
            MaxZahl := A[1]; MaxHäufig := 1;
            for i := 1 to N do
                Aktuell := A[i]; Häufig := 0;
                for j := 1 to N do
                    if A[j] = Aktuell then Häufig := Häufig + 1 fi
                od;
                if Häufig > MaxHäufig then
                    MaxZahl := Aktuell; MaxHäufig := Häufig fi
            od;
            write(MaxZahl); write(MaxHäufig);
        end
    end
end MaxHäufigkeit
```

Gesamtprogramm mit Struktur:

```
program MaxHäufigkeit is
declare positive N;
begin
    read(N);
    declare  array [1..N] of integer A;
    begin
        for i := 1 to N do read(A[i]) od;
        declare integer Aktuell, MaxZahl;
            natural  Häufig, MaxHäufig;
        begin
            MaxZahl := A[1]; MaxHäufig := 1;
            for i := 1 to N do
                Aktuell := A[i]; Häufig := 0;
                for j := 1 to N do
                    if A[j] = Aktuell then Häufig := Häufig + 1 fi
                od;
                if Häufig > MaxHäufig then
                    MaxZahl := Aktuell; MaxHäufig := Häufig fi
            od;
            write(MaxZahl); write(MaxHäufig);
        end
    end
end MaxHäufigkeit
```

Blöcke haben mindestens die beiden folgenden Vorteile:

(1) Sie fassen in sich abeschlossene Einheiten zusammen und tragen damit zur Strukturierung und zur besseren Verständlichkeit des Programms bei.
(2) Sie erlauben es dem Programmierer, Einfluss auf die Belegung des Speicherplatzes zu nehmen. (Dieser wird nämlich kellerartig gemäß der Blockstruktur verwaltet.)

Viele Sprachen besitzen ein durchgängiges Blockkonzept, wobei jeder Rumpf der Programmeinheiten (Funktionen, Prozeduren, Module usw., siehe im Folgenden) diesem Aufbau folgt.
Oft wird allerdings das "declare" nicht geschrieben, meist dann nicht, wenn es direkt auf ein "is" folgen würde.

Beispiel: Transitive Hülle eines gerichteten Graphen

Ein gerichteter Graph einschließlich seiner Wege und seine transitiver Hülle sind anschaulich rasch erläutert. Die Darstellung erfolgt oft durch Adjazenzmatrizen.

G = (V, E) mit $V = \{x_1, ..., x_n\}$ heißt gerichteter Graph (mit n Knoten), wenn $E \subseteq V \times V$ gilt.
Ein gerichteter Graph ist also eine Menge von m = |E| Knotenpaaren (x_i, x_j). Diese Paare heißen Kanten. Die Kante (x_i,x_j) verläuft vom Knoten x_i zum Knoten x_j.
Ein Weg $(y_0, y_1, ..., y_r)$ der Länge r von y_0 nach y_r ist eine Folge Knoten, die durch Kanten $(y_{i-1},y_i) \in E$ (für i = 1, ..., r) verbunden sind. Die transitive Hülle eines gerichteten Graphens (V,E) ist der gerichtete Graph (V,E') mit
E' = {(u,v) | es gibt einen Weg von u nach v}.

Oft beschreibt man einen gerichteten Graphen (mit den n Knoten $x_1, x_2, ..., x_n$) durch seine Adjazenzmatrix $A = (a_{i,j})_{i,j=1,2,...,n}$, die definiert ist durch

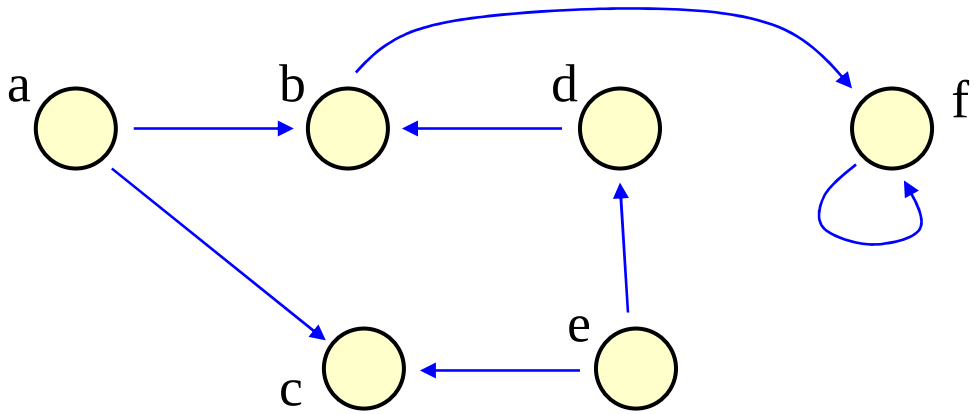
$$a_{i,j} = 1 \text{ für } (x_i, x_j) \in E \text{ und } a_{i,j} = 0 \text{ sonst } (i, j = 1, ..., n).$$

Die Aufgabe lautet also, zur Adjazenzmatrix A eines Graphens G die Adjazenzmatrix D der zu G gehörenden transitiven Hülle zu berechnen.

Beispiel: G = (V, E) mit V = {a, b, c, d, e, f} und
E = {(a, b), (a, c), (b, f), (d, b), (e, c), (e, d), (f, f)}

Adjazenzmatrix (hier ist n = |V| = 6): $A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$

Anschauliche Skizze:

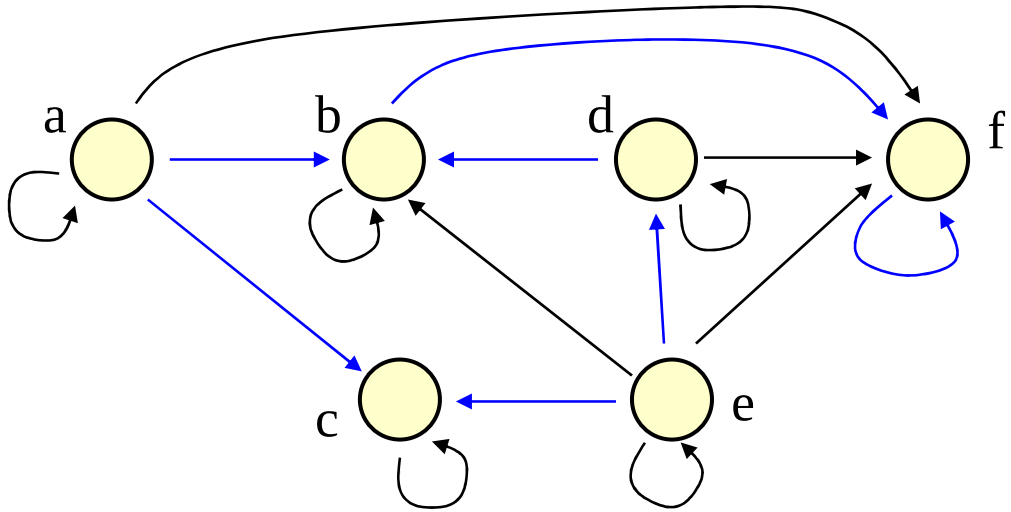


Zugehörige transitive Hülle: $G' = (V, E')$ mit $V = \{a, b, c, d, e, f\}$
und $E' = \{(a, a), (a, b), (a, c), (a, f), (b, b), (b, f), (c, c), (d, b),$
$(d, d), (d, f), (e, b), (e, c), (e, d), (e, e), (e, f), (f, f)\}$

Adjazenzmatrix (hier ist $n = |V| = 6$): $A' =$

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Anschauliche Skizze (die schwarzen Kanten kommen neu hinzu):



Allgemein ist die Adjazenzmatrix D der transitiven Hülle

$D = (d_{i,j})$ also definiert durch (i, j = 1, 2, ..., n):

$d_{i,j} = 1 \Leftrightarrow$ es gibt einen gerichteten Weg von x_i nach x_j

$d_{i,j} = 0 \Leftrightarrow$ sonst.

Lösungsidee:

Prüfe für alle i, j = 1, ..., n, ob es ein k gibt, so dass ein Weg von Weg von x_i nach x_j über x_k existiert.

Beginne damit, dass für jeden Knoten x stets ein Weg (der Länge 0) von x nach x führt und dass die Wege der Länge 1 durch die Kantenmenge E, also durch die Adjazenzmatrix A gegeben sind.

Diese Idee liefert den Warshall-Algorithmus zur Berechnung der transitiven Hülle (n ist global zu diesem Block):

```
declare  type adj is array [1..n, 1..n] of 0..1;
          adj A, D;  integer i, j, k;  ...
begin   ... ;  D := A;
        for i := 1 to n do  D[i,i] := 1 od;
        for k := 1 to n do
            for i := 1 to n do
                for j := 1 to n do
                    if D[i,k]=1 and D[k,j]=1 then D[i,j] := 1 fi
                od
            od
         od;
        ...
end;
```

Hinweis: k muss in der äußersten Schleife stehen!

Wie beweist man, dass dieses Verfahren tatsächlich die transitive Hülle berechnet?

Durch Induktion. Die Induktionsannahme lautet: Besitzt k am Ende der beiden inneren Schleifen den Wert **t**, so gilt

$D[i,j] = 1 \Leftrightarrow$ Es gibt $r \geq 0$ und einen Weg $(x_i, u_1, u_2, ..., u_r, x_j)$, so dass für alle $s = 1, .., r$ gilt: $u_s \in \{x_1, x_2, ..., x_{\mathbf{t}}\}$.

Man sieht dann leicht ein, dass diese Aussage nach Durchlauf der beiden inneren Schleifen auch für **t**+1 gilt. Die transitive Hülle erfüllt genau diese Aussage für **t** = n, d.h., am Ende ist daher D die Adjazenzmatrix für die transitive Hülle.

Hinweise (siehe später):

Zeitkomplexität: $O(n^3)$, da drei ineinander geschachtelte Schleifen und in der innersten Schleife konstanter Aufwand.

Zusätzlicher Platzbedarf: Für die Matrix D: $O(n^2)$ Plätze.

5.3 Funktionen

Ziel: Wenn ein Verfahren, das einen Wert als Ergebnis liefert, ausformuliert ist, möchte man es als "elementare Handlung" überall verwenden können. Hierzu benötigt man einen Namen und die genaue Angabe zur Übergabe von Werten ("Parameter").

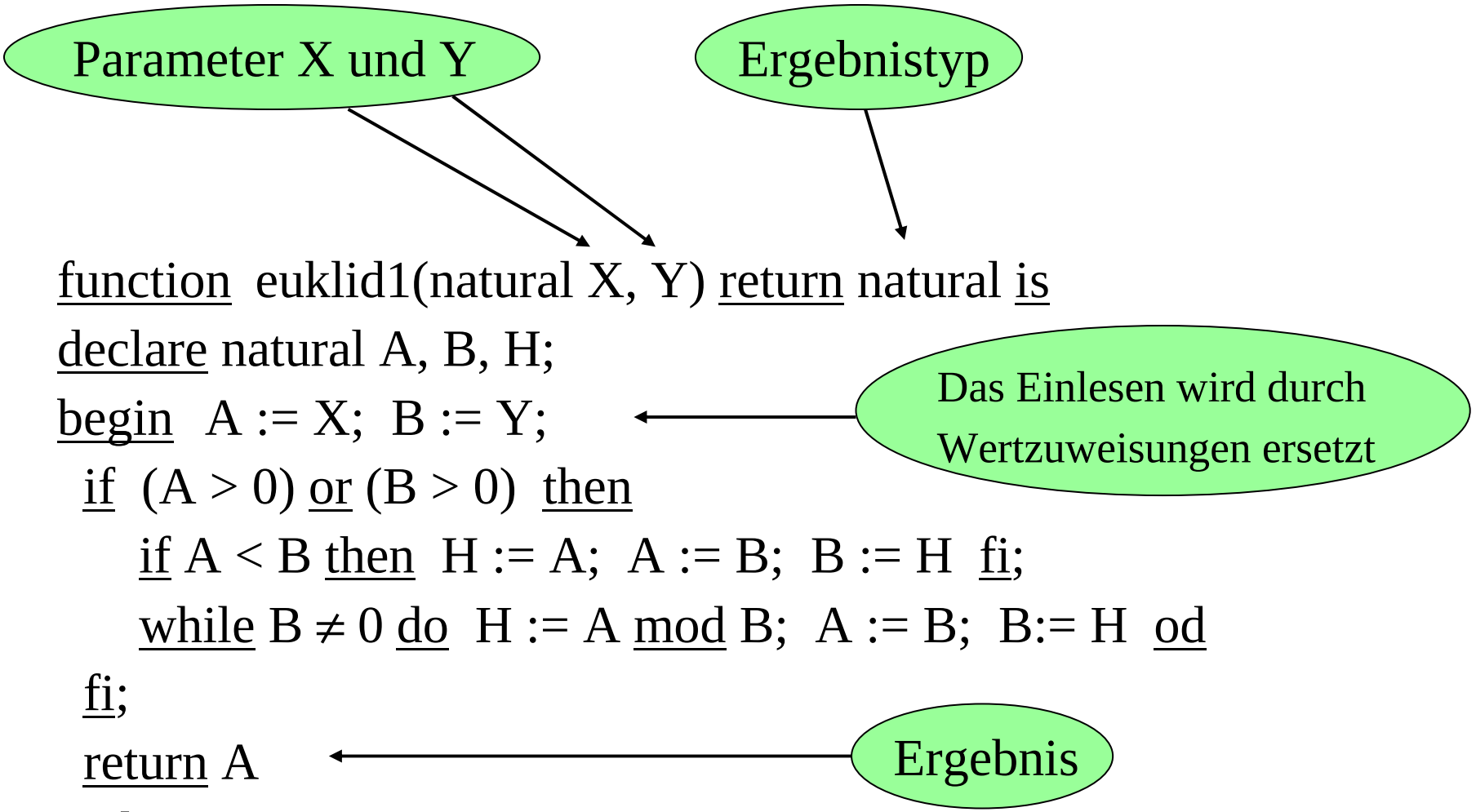
Der Name muss als Funktion deklariert werden.

Die Funktion muss die Quell- und Zielbereiche der zugehörigen realisierten Abbildung genau bezeichnen. Dies geschieht durch eine Liste von formalen Parametern mit ihren Typen und durch die Angabe des Ergebnistyps. Die formalen Parameter werden später durch aktuelle Parameter(werte) ersetzt.

Standardbeispiel ggT. Euklidischer Algorithmus:

```
program euklid1 is
declare natural A, B, H;
begin  read (A); read (B);
   if  (A > 0) or (B > 0)  then
      if A < B then  H := A;  A := B;  B := H  fi;
      while B ≠ 0 do  H := A mod B;  A := B;  B:= H  od
   fi;
   write (A)
end
```

Die Veränderlichen ("Parameter") sind die einzulesenden Variablen A und B. Das Ergebnis ist eine natürliche Zahl (write (A)). Wir schreiben diesen Algorithmus wie folgt in eine Funktion von $\mathbb{N}_0 \times \mathbb{N}_0$ nach $\mathbb{N}_0$ um:


Parameter X und Y
Ergebnistyp
function euklid1(natural X, Y) return natural is
declare natural A, B, H;
begin A := X; B := Y;
Das Einlesen wird durch Wertzuweisungen ersetzt
if (A > 0) or (B > 0) then
if A < B then H := A; A := B; B := H fi;
while B ≠ 0 do H := A mod B; A := B; B:= H od
fi;
return A
Ergebnis
end;

Man definiert die Funktion euklid1 im Deklarationsteil. Damit ist festgelegt, wo der Name bekannt ist (Sichtbarkeitsbereich). Innerhalb dieses Sichtbarkeitsbereichs kann euklid1 in allen ganzzahligen Ausdrücken verwendet werden, wobei die aktuellen Werte natürliche Zahlen sein müssen, z.B. (K sei vom Typ Natural und I, J, M vom Typ integer):

...

M := (7 + euklid1(K, 720)) * (I + J);

...

Eine Funktion der Form

function ... return T is ....

kann also wie jeder Operand vom Typ T in Ausdrücken verwendet werden.

5.4 Rekursion

Im Inneren einer Funktion ist der Name der Funktion bekannt (man sagt auch "sichtbar"). Man kann daher dort die Funktion selbst verwenden. Die (direkte oder indirekte) Verwendung einer Funktion in ihrem eigenen Rumpf nennt man Rekursion.

Standardbeispiel: Fakultätsfunktion

$$n! = \begin{cases} 1 & \text{für } n = 0; \\ n \cdot (n-1)! & \text{für } n > 0 \end{cases}$$

```
function Fak(natural n) return natural is
begin  if n=0 then return 1 else return n*Fak(n-1) fi
end Fak;
```

Vorteil: Diese rekursive Formulierung übernimmt direkt die Definition.

Rekursive Funktion für die Fakultät in Java

```
public int Fakultaet (int n) {
if (n<=0) {return 1}
else {return n*Fakultaet(n-1)}
}
```

Aufruf mittels (für int X, A)

```
if (A>=0) {X = Fakultaet (A); ...} else { ... };
```

Funktion für den ggT in Java (siehe auch unter 6.3)

```
public int ggT (int i, j) {
if (i==0) {return j}
else if (j==0) {return i}
      else {return ggT(j, i mod j)}
}
```

Beachte: Hier wird ggT(0,0) = 0 zurückgegeben.

Aufruf mittels:

```
if (a>=0 and b>=0) {c = ggT(a,b); ... }
else { ... };
```

Beachte: Jede while-Schleife lässt sich leicht in eine rekursive Funktion (oder Prozedur) umschreiben.
In Java sieht das folgendermaßen aus:

i := a;
while i ≤ b do C; i:=i+1 od;

```
public void WHILE(int i) {
if (i<=b)
   {C; Fakultaet(i+1)}
}
```

Aufruf durch: WHILE(a);

Da die Rekursion in WHILE stets nur am Ende erfolgt, nennt man diese Art der Rekursion "tail recursion".

5.5 Prozeduren

Man darf eine Folge von Deklarationen und Anweisungen zu einer Programmeinheit, genannt "Unterprogramm" oder "Prozedur" (engl.: procedure, subprogram, subroutine) unter einem Namen einschließlich der formalen Parameter zusammenfassen. Diesen Namen mit aktuellen Parametern kann man dann wie eine (elementare) Anweisung im Sichtbarkeitsbereich des Namens benutzen (Prozeduraufruf, "call").

Spezifikation für Unterprogramme, wobei der Teil "(<Parameterteil>)" auch fehlen darf:
procedure <Name> (<Parameterteil>);

Die *Prozedurdeklaration* beginnt mit dieser Spezifikation. Danach folgt der Rumpf (is ...) als Block bestehend aus dem (eventuell leeren) Deklarationsteil und den Anweisungen.

Beispiel: Austauschen zweier Inhalte

```
program P is
declare real A, B, C, D, H;
begin   ...  if A<B then  H:=A; A:=B; B:=H  fi;  ...
        ...  H:=C; C:=D; D:=H;  ...
end;
```

Herausziehen des Vertauschens als Unterprogramm:

```
procedure Vertausche (pppp real X, Y) is
real Z;
begin Z:=X; X:=Y; Y:=Z end;
```

pppp gibt an, in welcher Weise die Variableninhalte übergeben werden sollen. Das abgewandelte Programm lautet dann:

Beispiel:

```
program PP is
declare real A, B, C, D, H;

procedure Vertausche (pppp real X, Y) is
declare real Z;
begin Z:=X; X:=Y; Y:=Z end;

begin   ... if A<B then Vertausche(A, B) fi; ...
        ... Vertausche(C, D); ...
end;
```

?

Ob dies korrekt ist, hängt von der *Übergabe der Parameter "pppp"* ab! Falls X und Y als Konstanten aufgefasst werden, dann ist PP kein äquivalentes Programm zu P.
Als *pppp* betrachten wir *value*, *ref*(erence) und *name*.

Drei Beispiele für die Parameterübergabe (bei Funktionen, analog für Prozeduren). Wir betrachten den leicht abgeänderten ggT:

```
function  euklid1(value natural A, B) return natural is
declare natural H;
begin
  if  (A > 0) or (B > 0)  then
      if A < B then  H := A;  A := B;  B := H  fi;
      while B ≠ 0 do H := A mod B;  A := B;  B:= H  od  fi;
  return A
end;
```

Hier werden A und B als lokale Variablen der Funktion angesehen, denen beim Aufruf die Werte der aktuellen Parameter zugewiesen werden; sie dürfen im Rumpf der Funktion auch neue Werte erhalten. Allerdings werden am Ende die Werte von A und B *nicht* an die Aufrufstelle übermittelt.
Solche Parameter heißen "call by value"-Parameter.

Statt Werte zu übergeben, könnte man auch mit den Variablen des aufrufenden Programms arbeiten. Dies lässt sich elegant mit Verweisen ("reference" oder "access") realisieren.

```
function  euklid1(ref natural A, B) return natural is
declare natural H;
begin
  if  (A > 0) or (B > 0)  then
     if A < B then  H := A;  A := B;  B := H  fi;
     while B ≠ 0 do  H := A mod B;  A := B;  B:= H  od  fi;
  return A
end;
```

A und B sind nun *Zeiger* auf Variablen vom Typ natural und erhalten die (Speicher-) Adresse des jeweiligen aktuellen Parameters, der dann natürlich eine Variable sein muss, zugewiesen. Beachte: Die Funktion benutzt statt A dann stets die Variable, auf die A zeigt (genannt deref A oder A↑ oder A.).

Der Aufruf Z := euklid1(X,Y); bewirkt also, dass an der Aufrufstelle folgender Block ausgeführt wird:

```
declare ref natural A, B;  -- Zeiger auf Variablen vom Typ natural
begin
    A zeigt auf X;  B zeigt auf Y;
    declare natural H;
    begin
        if  (deref A > 0) or (deref B > 0)  then
            if  deref A < deref B  then  H := deref A;
                                 deref A := deref B;  deref B := H  fi;
            while deref B ≠ 0 do  H := A mod B;
                                 deref A := deref B;  deref B:= H  od   fi;
        Z := deref A      -- dies entspricht dem return A
    end
end;
```

Dieser Aufruf übergibt Zeiger (Referenzen, Adressen) an die formalen Parameter. Die formalen Parameter sind nun lokale Variable vom Typ ref ... . Die Funktion arbeitet über diese Zeiger mit den Variablen des aufrufenden Programmstücks!

Solche Parameter heißen "call by reference"-Parameter.

Man beachte, dass diese Zeiger auch in Strukturen hineinführen können. Z.B. kann man auch Z := euklid1(D[1], D[2]); schreiben, sofern D vom Typ array [1..N] of natural ist. Dann werden die (Adress-) Zuweisungen A zeigt auf D[1]; B zeigt auf D[2]; ausgeführt. Das Gleiche gilt, wenn die aktuellen Parameter Komponenten eines Records sind.

Als dritte Möglichkeit betrachten wir, dass die formalen Parameter A und B textuell durch die aktuellen Parameter X und Y ersetzt werden ("call by name"-Parameter).

```
function  euklid1(name natural A, B) return natural is
declare natural H;
begin
  if  (A > 0) or (B > 0)  then
      if A < B then  H := A;  A := B;  B := H  fi;
      while B ≠ 0 do  H := A mod B;  A := B;  B:= H  od  fi;
  return A
end;
```

Der Aufruf Z := euklid1(X,Y); bewirkt dann an der Aufrufstelle die Ausführung des Blocks

```
declare natural H;
begin
  if  (X > 0) or (Y > 0)  then
      if X < Y then  H := X;  X := Y;  Y := H  fi;
      while Y ≠ 0 do  H := X mod Y;  X := Y;  Y:= H  od  fi;
  Z := X
end;
```

Parameterübergaben für Funktionen und Prozeduren

Die Zuordnung der aktuellen Parameter an die formalen Parameter beim Prozeduraufruf ("call") bezeichnet man als Parameterübergabe. Dieser ist bei Funktionen und Prozeduren im Wesentlichen gleich. Die drei wichtigsten Mechanismen lauten nochmals zusammengefasst:

call by value: Nur die Werte werden übergeben; die formalen Parameter sind lokale Variablen der Prozedur.

call by reference: Ein Verweis auf die aktuelle Variable wird übergeben; die formalen Parameter sind Zeiger auf Variablen.

call by name: Der formale Parameter wird textuell durch den aktuellen Parameter ersetzt (wobei keine Namen im aktuellen Parameter hierdurch lokal werden dürfen - sonst umbenennen).

Etwas präziser: Bedeutung eines Funktionsaufrufs $f(\alpha_1, ...,\alpha_k)$

Wenn f eine (zuvor deklarierte) Funktion mit k formalen Parametern ist und f in der Wertzuweisung für eine Variable X

$X := ... f(\alpha_1, ...,\alpha_k) ...$

steht, so wird die Berechnung des Ausdrucks "... $f(\alpha_1,...,\alpha_k)$ ..." unterbrochen, sobald man auf den Namen f stößt. Zuerst wird geprüft, ob f hier ein sichtbarer Name ist, dann werden die Ausdrücke $\alpha_1, ..., \alpha_k$ (dies sind die k aktuellen Parameter) in irgendeiner Reihenfolge ausgewertet; diese Werte werden je nach Parameterübergabemechnismus den zugehörigen formalen Parametern von f zugewiesen. Dann wird der Funktionsrumpf von f in diese Stelle des Programms hineinkopiert und ausgerechnet, wobei man ein Resultat b erhält. Danach wird diese Kopie wieder entfernt. Wenn b den Ergebnistyp der Funktion besitzt, so wird $f(\alpha_1, ...,\alpha_k)$ durch diesen Wert ersetzt und der Ausdruck "... $f(\alpha_1, ...,\alpha_k)$ ..." wird weiter ausgewertet.

In dieser präziseren Beschreibung des Funktions- bzw. Prozeduraufrufs wird der Rumpf an die Aufrufstelle kopiert (man bezeichnet dies als "Kopierregel"). Man führt also keinen "Sprung in die Prozedur" aus, und zwar aus dem Grunde, weil im Rumpf der Prozedur erneut Prozeduraufrufe und insbesondere rekursive Aufrufe vorkommen können. Durch die Kopierregel wird der Programmtext entsrechend ausgeweitet und im Inneren des Rumpfes folgende weitere Prozeduraufrufe sind automatisch miterfasst.

Eine solche (entsprechend der Parameterübergabe modifizierte) Kopie des Rumpfes nennt man auch eine "Inkarnation" oder "Instanz" der Prozedur. Vgl. Duden Informatik für ein einleuchtendes Beispiel.

Prozeduren verändern in der Regel Inhalte von Variablen (oder sie erzeugen Ausdrucke). Diese Veränderungen können nur geschehen, wenn auf die aktuellen Parameter explizit zugegriffen werden kann oder die Variablen "global" zu der Prozedur sind. Alle Variablen, die vorher im Block, in dem die Prozedur deklariert ist, oder in einem übergeordneten Block deklariert sind, sind global zu der Prozedur und können von ihr verändert werden.

Hinweis: Die Veränderung von globalen Variablen in einer Funktion oder Prozedur bezeichnet man allgemein als "Seiteneffekt". Solche Effekte sind oft Anlass für Fehler, die nur schwer aufzuspüren sind. Das Programmieren mit Seiteneffekten sollte daher vermieden werden (schlechter Programmierstil)!

6. Komplexität von Algorithmen

6.1 O-Notation

Wer Programme einsetzt, muss die Zeitspanne kennen, innerhalb derer die Ergebnisse ausgegeben werden. Man spricht von der "Zeitkomplexität" des Programms.

Diese "Rechendauer" ist abhängig von der Eingabe. In der Regel definiert man den Zeitaufwand $t_\pi : \mathbb{N}_0 \to \mathbb{N}_0$ als Funktion der Länge der Eingabe des Programms π:

$t_\pi(n)$ = maximale Zeit, die das Programm π für irgendeine Eingabe w der Länge n bis zum Anhalten benötigt.

Auf gleiche Weise kann man eine Speicherplatzfunktion s_π definieren, die die Anzahl der vom Programm benötigten Speicherplätze (z.B. gemessen in Byte) angibt.

Die Rechendauer t_π ist natürlich nur dann eine totale Funktion, wenn das Programm π stets terminiert. Dies kann man immer erreichen, indem man einen Zähler mitlaufen lässt und das Programm abbricht, wenn ein vorgegebener Wert überschritten wird. Daher kann man verlangen, dass das Programm π für alle Eingabewerte anhält.
Wie misst man die Zeit? In Sekunden oder Millisekunden wäre nicht sinnvoll, da dies vom verwendeten Rechner abhängig wäre. Statt dessen zählt man die Zahl der elementaren Schritte (meist: Auswertung von Bedingungen und Ausführung von Wertzuweisungen).

Wir betrachten zunächst ein Beispiel und führen Funktionsklassen ein, bevor wir in 6.3 die Komplexität präzisieren.

Beispiel

```
program was is
declare  natural A, B;
begin read (A);
        B := 1;
        while A > 1 do A := A div 2; B := B+1 od;
        write (B)
end
```

Mit Ablaufprotokollen ermittelt man einige Werte der realisierten Funktion g: $\mathbb{N}_0 \rightarrow \mathbb{N}_0$:

a	g(a)
0	1
1	1
2	2
3	2

a	g(a)
4	3
5	3
6	3
7	3

a	g(a)
8	4
16	5
80	7
1024	11

```
program was is
declare  natural A, B;
begin read (A);
        B := 1;
        while A > 1 do A := A div 2; B := B+1 od;
        write (B)
end
```

Man vermutet nun, dass die realisierte Funktion g: $\mathbb{N}_0 \rightarrow \mathbb{N}_0$ lautet: g(a) = Länge der binären Darstellung der Zahl a.

Dies trifft zu, weil in jedem Schritt die Länge der Zahl in A durch die Wertzuweisung A := A div 2 genau um eins verringert wird, bis eine Darstellung der Länge 1 erreicht ist. Die Schleife wird einmal weniger, als die Länge angibt, durchlaufen. Da B anfangs auf 1 gesetzt wurde, wird daher genau die Länge der Binärdarstellung der Eingabe berechnet.

```
program was is
declare  natural A, B;
begin read (A);
      B := 1;
      while A > 1 do A := A div 2; B := B+1 od;
      write (B)
end
```

Wie lange dauert nun die Berechnung? Wir nehmen an: Die Auswertung jeder Bedingung und die Durchführung jeder Wertzuweisung dauern gleich lange. Dann erhält man (es sei n die Länge der Binärdarstellung der Eingabezahl a):

$1 + 1 + (n-1) * (1 + 1 + 1) + 1 + 1 = 3n + 1$ Zeiteinheiten.

Die Größenordnung ist also proportional zu n *(man sagt, sie sei O(n))*, und meist interessiert nur diese Abschätzung, bei der multiplikative und additive Konstanten ignoriert werden. ■

Um die Größenordnung einer reellwertigen oder ganzzahligen Funktion zu beschreiben, verwenden wir die so genannten *Landau-Symbole* (nach dem deutschen Mathematiker Edmund Landau, 1877-1938). Hierbei werden multiplikative und additive Konstanten vernachlässigt; es wird nur *der* Term in Abhängigkeit von n, der für $n \to \infty$ alles andere überwiegt, berücksichtigt.

Formal gesehen handelt es sich bei O(f) um die Definition einer Funktionenklasse in Abhängigkeit von einer Funktion f. In O(f) sind alle Funktionen über den reellen Zahlen enthalten, die "schließlich von f dominiert" werden.

Insgesamt verwendet man folgende 5 Klassen O, o, Ω, ω und Θ, wobei am häufigsten "O" verwendet wird.

Landau-Symbole (vgl. Folie 101)

Definition: "groß O", "klein O", "groß Omega", klein Omega", "Theta"

Es sei $f: \mathbb{R}^+ \to \mathbb{R}^+$ eine Funktion über den positiven reellen Zahlen $\mathbb{R}^+ \subset \mathbb{R}$ (oft wird diese Definition auf die natürlichen Zahlen eingeschränkt, also auf Funktionen $f: \mathbb{N} \to \mathbb{N}$; unten muss dann nur $g: \mathbb{R}^+ \to \mathbb{R}^+$ durch $g: \mathbb{N} \to \mathbb{N}$ ersetzt werden).

$$\mathbf{O(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \exists\, n_0 \in \mathbb{N}\ \forall n \geq n_0: g(n) \leq c \cdot f(n)\},$$

$$\mathbf{o(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \forall c \in \mathbb{R}^+ \exists\, n_0 \in \mathbb{N}\ \forall n \geq n_0: c \cdot g(n) \leq f(n)\},$$

$$\mathbf{\Omega(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \exists\, n_0 \in \mathbb{N}\ \forall n \geq n_0: f(n) \leq c \cdot g(n)\},$$

$$\mathbf{\omega(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \forall c \in \mathbb{R}^+ \exists\, n_0 \in \mathbb{N}\ \forall n \geq n_0: c \cdot f(n) \leq g(n)\},$$

$$\mathbf{\Theta(f)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+ \exists\, n_0 \in \mathbb{N}\ \forall n \geq n_0: c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}.$$

Erläuterungen: Es sei f: $\mathbb{R}^+ \rightarrow \mathbb{R}^+$

g liegt in **O(f)**, wenn g höchstens so stark wächst wie f, wobei Konstanten nicht zählen. Statt *g ist höchstens von der Größenordnung f*, sagen wir, *g ist groß-O von f,* und meinen damit, dass $g \in O(f)$ ist.

Wenn eine Funktion g zusätzlich echt schwächer als f wächst, wenn also zusätzlich gilt:

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

dann sagen wir, *g ist von echt kleinerer Größenordnung als f* oder *g ist klein-o von f,* und meinen damit, dass $g \in$ **o(f)** ist.

Erläuterungen (Fortsetzung): Es sei f: $\mathbb{R}^+ \to \mathbb{R}^+$

Die Klassen Ω und ω (groß Omega und klein Omega) bilden die "Umkehrungen" der Klasse O und o. Eine Funktion g liegt genau dann in **Ω(f)** bzw. in **ω(f)**, wenn f in O(g) bzw. in o(g) liegt.

In Ω(f) liegen also die Funktionen, die mindestens so stark wachsen wie f, und in ω(f) liegen die Funktionen, die zusätzlich bzgl. n echt stärker wachsen.

In der Klasse **Θ(f)** liegen die Funktionen, die sich bis auf Konstanten im Wachstum wie f verhalten. Wenn $g \in \Theta(f)$ ist, dann sagen wir, *g ist von der gleichen Größenordnung wie f* oder *g ist Theta von f*. Da in diesem Fall g in O(f) und f in O(g) liegen müssen, folgt unmittelbar die Gleichheit $\Theta(f) = O(f) \cap \Omega(f)$.

Schreibweisen: Es sei f: $\mathbb{R}^+ \to \mathbb{R}^+$

Statt $g \in O(f)$ schreibt man manchmal auch $g = O(f)$, um auszudrücken, dass g höchstens von der Größenordnung f ist. Das Gleiche gilt für die anderen vier Klassen.

Anstelle der Funktionen gibt man meist nur deren formelmäßige Darstellung an. Beispiel: Statt

O(f) für die Funktion f mit $f(n) = n^2$ *für alle* $n \in \mathbb{N}$

schreibt man einfach $O(n^2)$.

Man schreibt auch "Ordnungs-Gleichungen", die aber nur von links nach rechts gelesen werden dürfen, z.B.:

$3 \cdot n^3 + 12 \cdot n^2 + 8 \cdot n \cdot \log(n) + 6/n = 3 \cdot n^3 + O(n^2) = O(n^3)$.

Korrekt müsste man hierfür eigentlich schreiben:

$3 \cdot n^3 + 12 \cdot n^2 + 8 \cdot n \cdot \log(n) + 6/n \in O(3 \cdot n^3) \cup O(n^2 + n \cdot \log(n)) = O(n^3)$.

6.2 Einige Funktionsklassen

O(1) ist die Klasse der Funktionen, die höchstens wie ein Vielfaches der konstanten Funktion *f(n) = 1 für alle n∈ lN* wachsen. Somit gehören alle konstanten Funktionen, aber auch Funktionen wie sin(n), cos(n), 1/n, $1/n^2$ oder 1/log(n) zu O(1).

Gibt es eine Klasse von Funktionen, die nicht in O(1) liegen und nur sehr schwach wachsen, also deutlich langsamer als f(n)=n? Aus der Schule kennen Sie den Logarithmus log(n). Noch wesentlich schwächer wächst der "iterierte Logarithmus" log*:

log*(n) = 0, für n=0 und 1,

log*(n) = Min{k | log(log(log(...log(n)...))) < 2} für n>1.

k ineinander geschachtelte Logarithmen

log* ∉ O(1). [Untersuchen Sie diese Funktion log*(n) oder vgl. Folie 410.]

O(n) = Klasse der höchstens linear wachsenden Funktionen:
O(n) := $\{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0: g(n) \leq c \cdot n\}$.
Man beachte, dass hierin auch alle Funktionen der Form $g(n) = c_1 \cdot n + c_2$ (für zwei positive Konstanten c_1 und c_2) enthalten sind, weil für $n \geq 1$ gilt: $g(n) = c_1 \cdot n + c_2 \leq (c_1 + c_2) \cdot n$.
Wenn g in O(n) liegt, so sagt man auch, g sei *höchstens linear*.

Zu den höchstens linear wachsenden Funktionen gehört (wegen $\log(x) < x$ für alle $x > 0$) auch der Logarithmus. Es gilt daher: $\log(n) \in O(n)$. Aber auch für die Potenzen des Logarithmus gilt $\log^m(n) \in O(n)$ für alle natürlichen Zahlen m. Hierfür beachte:

Der Logarithmus wächst schwächer als jede noch so kleine positive Potenz, d.h.: Für jedes $m \geq 1$ gilt ab einem hinreichend großen n: $\log(n) < n^{\frac{1}{m}}$ und folglich $\log^m(n) < n$.

$\Omega(n)$ = Klasse der mindestens linear wachsenden Funktionen:
$\mathbf{\Omega(n)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \; \exists\, n_0 \in \mathbb{N} \; \forall\, n \geq n_0 : n \leq c \cdot g(n)\}$.
Auch hierin sind alle Funktionen der Form $g(n) = c_1 \cdot n + c_2$ (für zwei positive Konstanten c_1 und c_2) enthalten, weil für $n \geq 1$ gilt:
$n \leq (1/c_1) \cdot g(n) = n + (c_2/c_1)$.
Wenn g in $\Omega(n)$ liegt, so sagt man auch, g sei *mindestens linear*.

$\Theta(n)$ = Klasse der linear wachsenden Funktionen:
$\mathbf{\Theta(n)} := \{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+ \; \exists\, n_0 \in \mathbb{N} \; \forall\, n \geq n_0 : c_1 \cdot n \leq g(n) \leq c_2 \cdot n\}$.
Wegen $\Theta(f) = O(f) \cap \Omega(f)$ für alle Funktionen f gehören zu $\Theta(n)$ insbesondere alle Funktionen der Form $g(n) = c_1 \cdot n + c_2$ (für zwei positive Konstanten c_1 und c_2), aber auch Funktionen wie $g(n) = n + \log^m(n) + 1/n \in \Theta(n)$ usw.

$O(n^2)$ = Klasse der höchstens quadratisch wachsenden Funktionen.
$O(n^2)$:= $\{g: \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \; \exists\, n_0 \in \mathbb{N} \; \forall n \geq n_0: g(n) \leq c \cdot n^2\}$.
Hierin sind alle Funktionen der Form $g(n) = c_1 \cdot n^2 + c_2 \cdot n + c_3$ (für Konstanten c_1, c_2 und c_3) enthalten, weil ab einem gewissen $n \geq 1$ dann gilt: $g(n) = c_1 \cdot n^2 + c_2 \cdot n + c_3 \leq (c_1+1) \cdot n^2$.
Wenn g in O(n) liegt, so sagt man, g wächst *höchstens quadratisch*.
$\Omega(n^2)$ ist die Klasse der mindestens quadratisch wachsenden Funktionen.

Für jede natürliche Zahl k ist **$O(n^k)$** die Klasse der höchstens wie n^k wachsenden Funktionen; $O(n^k)$ umfasst insbesondere alle Polynome vom Grad k.

Für jede natürliche Zahl k ist **$o(n^k)$** die Klasse der echt schwächer als n^k wachsenden Funktionen. Hierin liegen z.B. Funktionen wie n^{k-1} oder $n^k/\log(n)$ oder n^{k-d} für jede reelle Zahl $d > 0$.

Für die Praxis wichtig sind vor allem folgende Funktionsklassen:

O(1): konstante Funktionen.

O(log n): höchstens logarithmisch wachsende Funktionen; wenn die Länge einer Darstellung wichtig ist, kommt oft der Logarithmus ins Spiel.

$O(n^{1/k})$: höchstens wie die k-te Wurzel wachsende Funktionen.

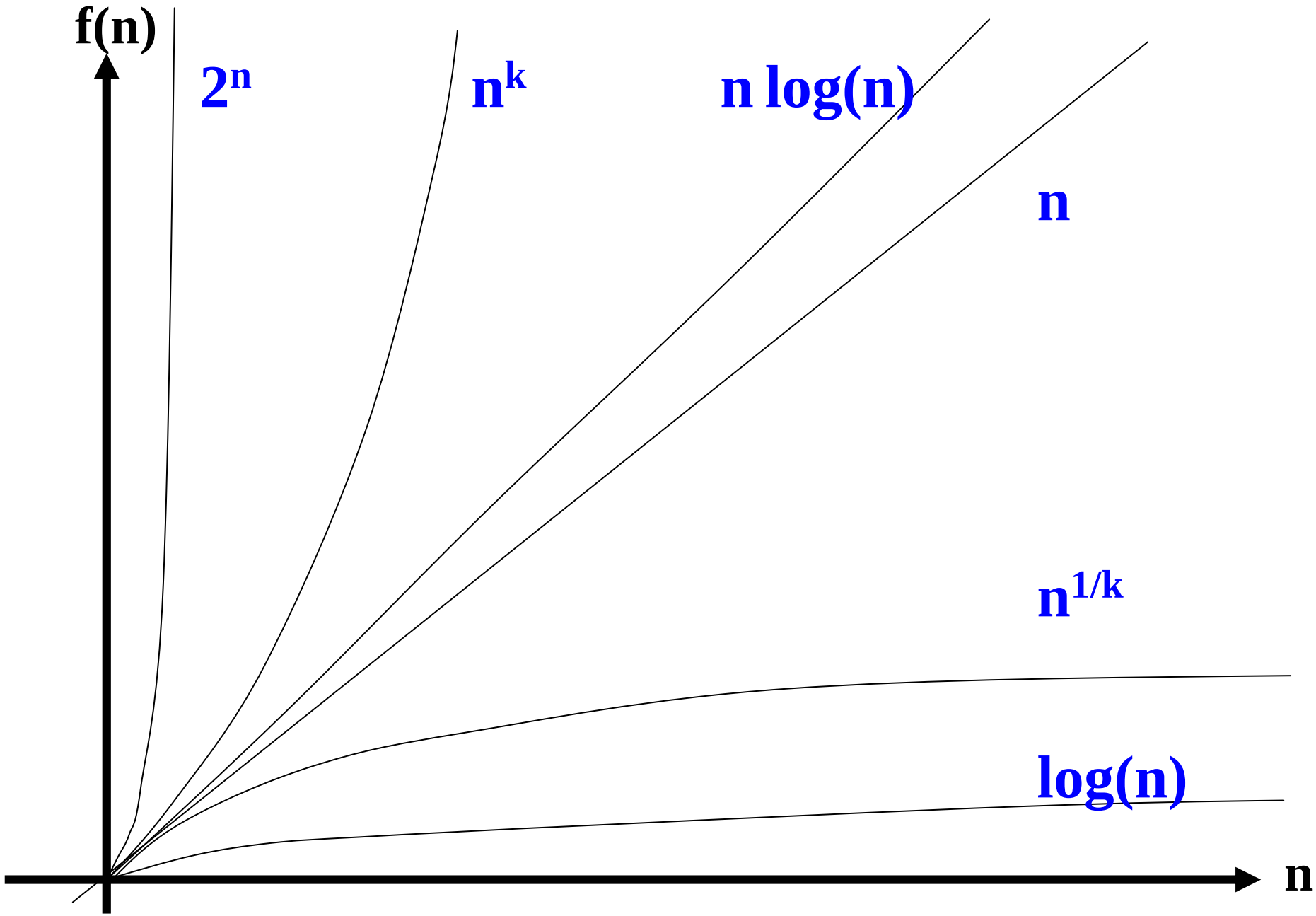
O(n): lineare Funktionen.

O(n·log(n)): Das sind Funktionen, die "ein wenig" stärker als linear wachsen.

$O(n^2)$: höchstens quadratisch wachsende Funktionen.

$O(n^k)$: höchstens polynomiell vom Grad k wachsende Funktionen.

$O(2^n)$: höchstens exponentiell (zur Basis 2) wachsende Funktionen.


f(n)
2^n
n^k
n log(n)
n
n^1/k
log(n)
n

Hilfssatz:
Es gelten folgende Aussagen (die Beweise sind einfach):

$o(f) \subset O(f), \quad \omega(f) \subset \Omega(f), \quad \Theta(f) = O(f) \cap \Omega(f).$

Für alle $g \in O(f)$ gelten $O(f+g) = O(f)$ und $o(f+g) = o(f)$.

Für alle $g \in \Omega(f)$ gelten $\Omega(f+g) = \Omega(g)$ und $\omega(f+g) = \omega(g)$.

Für alle $g \in \Theta(f)$ gilt $\Theta(f+g) = \Theta(f) = \Theta(g)$.

Zur Übung: Untersuchen Sie, ob folgende Formeln gelten:

$\omega(f) \cup O(f) = \Omega(f)$?

$O(f) - o(f) = \Theta(f)$?

$o(f) \cap \omega(f) = \varnothing$?

In der Regel sind Funktionen durch solche "Landau"-Klassen nicht vergleichbar. Zum Beispiel gilt für die beiden Funktionen

$$f(n) = \begin{cases} 1, \text{ für gerades } n \\ n, \text{ für ungerades } n \end{cases} \qquad g(n) = \begin{cases} n, \text{ für gerades } n \\ 1, \text{ für ungerades } n \end{cases}$$

weder $f \in O(g)$ noch $g \in O(f)$.

Man verwendet vor allem die Klassen O und Θ bei der Untersuchung des Zeit- und Platzaufwands. Meist interessiert nur die Größenordnung der Komplexität und nicht der genaue Wert von Konstanten (der implementierungsabhängig ist und den man in der Praxis durch Messung ermitteln kann).

6.3 Uniforme Komplexität von Algorithmen

Wir betrachten hier nur Algorithmen und Programme, die für alle Eingaben anhalten. Die Komplexität für die Eingabe w ist die Anzahl der Schritte $t_A(w)$ oder die Zahl an Speicherplätzen $s_A(w)$, die ein Programm oder ein Algorithmus A für die Eingabe w benötigt, bis seine Berechnung beendet ist.

Genauer: Sei A ein Algorithmus (oder Programm). Eine *Berechnung* für die Eingabe w ist eine Folge von elementaren Anweisungen und Ausdrücken. Hiermit definieren wir:

$t_A(w)$ = Anzahl der elementaren Anweisungen und Bedingungen, die der Algorithmus A während seiner Berechnung bei Eingabe von w durchläuft, bis er anhält.

$s_A(w)$ = Anzahl der verschiedenen Speicherplätze, auf die A während seiner Berechnung bei Eingabe von w zugreift, bis er anhält.

Man fasst die Eingaben gleicher Länge zusammen und betrachtet als Komplexität in der Regel den schlechtesten Fall ("worst case" complexity), manchmal den durchschnittlichen Fall ("average case") oder selten den besten Fall ("best case"). Daher definiert man die **Zeit- und Platz-Komplexität** für Programme/Algorithmen A wie folgt (Σ ist die Menge der Eingabezeichen, z.B. ASCII-Zeichen):

$t_A(n) = \text{Max}\ \{t_A(w) \mid \text{die Länge von w ist n}\},$
$s_A(n) = \text{Max}\ \{s_A(w) \mid \text{die Länge von w ist n}\}.$

$$t_A^{av}(n) = \frac{1}{|\Sigma^n|} \sum_{w \in \Sigma^n} t_A(w), \qquad s_A^{av}(n) = \frac{1}{|\Sigma^n|} \sum_{w \in \Sigma^n} s_A(w).$$

$t_A^{best}(n) = \text{Min}\ \{t_A(w) \mid \text{die Länge von w ist n}\},$
$s_A^{best}(n) = \text{Min}\ \{s_A(w) \mid \text{die Länge von w ist n}\}.$

Die Komplexitätsmaße t und s behandeln alle Bedingungen und alle Wertzuweisungen in gleicher Weise. Jede solche Elementareinheit erfordert genau einen Schritt. Auch der Zugriff auf eine (indizierte) Variable erfolgt in einem Schritt.

Man nennt die oben definierte Komplexität bei Algorithmen und Programmen daher das "uniforme Komplexitätsmaß".

Hinweis: Oft genügt diese uniforme Abschätzung. Allerdings muss man ein anderes Komplexitätsmaß definieren, wenn man Rechnungen *zeichenweise* wie auf einem Stück Papier durchzuführen hat. Dann dauert eine Addition X := Y+Z nicht einen Schritt, sondern so viele Schritte, wie die Inhalte von X und Y lang sind. Da die Länge bei Zahlen ungefähr gleich dem Logarithmus ist, spricht man dann vom "logarithmischen Komplexitätsmaß".

Einschub: Die Folien 344-350 sind nur als Hinweise für Fortgeschrittene gedacht und für andere möglicherweise unverständlich.

Man kann nun "Komplexitätsklassen" definieren: Ein Problem liegt in der (Zeit-) Komplexitätsklasse K(f), wenn es einen Algorithmus A gibt, der dieses Problem löst und dessen Zeitdauer t_A in O(f) ist. (Analog für "Platz".)

Was ist ein "Problem"? Im Wesentlichen die Menge der Werte, die eine Lösung des Problems bilden. Somit kann man ein Problem stets als eine Sprache darstellen.

Sprachen und Funktionen sind durch die semicharakteristische Funktion ψ miteinander verbunden. Will man das Komplement ebenfalls beschreiben (und das will man meist), so verwendet man die charakteristische Funktion χ.

Einschub:

Für jede Sprache $L \subseteq \Sigma^*$ heißt die Abbildung $\chi_L : \Sigma^* \to \{0,1\}$

mit $\chi_L(w)$ = if $w \in L$ then 1 else 0 fi

die charakteristische Funktion von L.

Diese Funktion ist total, d.h., für alle Eingabewerte definiert.

Für jede Sprache $L \subseteq \Sigma^*$ heißt die partielle Abbildung

$\psi_L : \Sigma^* \to \{1\}$

mit $\psi_L(w)$ = if $w \in L$ then 1 else undefiniert fi

die semi-charakteristische Funktion von L.

Diese Funktion ist nur für die Werte aus L definiert, also (außer im Falle $L = \Sigma^*$) immer eine partielle Abbildung.

Einschub:

Hiermit lassen sich die **Komplexitätsklassen** für Probleme (also für Sprachen und Funktionen) definieren. Es seien T, S: $\mathbb{N}_0 \to \mathbb{N}_0$ zwei gegebene Funktionen, dann setze:

$\text{DTimeSpace}(T,S) = \{ L \subseteq \Sigma^* \mid$ Es gibt einen deterministischen Algorithmus A, der χ_L berechnet, mit $t_A(n) \in O(T(n))$ und $s_A(n) \in O(S(n)).\}$

$\text{NTimeSpace}(T,S) = \{ L \subseteq \Sigma^* \mid$ Es gibt einen nichtdeterministischen Algorithmus A, der ψ_L berechnet, mit $t_A(n) \in O(T(n))$ und $s_A(n) \in O(S(n)).\}$

(Zu "nichtdeterministisch" siehe übernächste Folie und 6.4.7.)

Einschub:

$DTime(T) = \{ L \subseteq \Sigma^* \mid$ Es gibt einen deterministischen Algorithmus A, der χ_L berechnet, mit $t_A(n) \in O(T(n)).\}$

$NTime(T) = \{ L \subseteq \Sigma^* \mid$ Es gibt einen nichtdeterministischen Algorithmus A, der ψ_L berechnet, mit $t_A(n) \in O(T(n)).\}$

$DSpace(S) = \{ L \subseteq \Sigma^* \mid$ Es gibt einen deterministischen Algorithmus A, der χ_L berechnet, mit $s_A(n) \in O(S(n)).\}$

$NSpace(S) = \{ L \subseteq \Sigma^* \mid$ Es gibt einen nichtdeterministischen Algorithmus A, der ψ_L berechnet, mit $s_A(n) \in O(S(n)).\}$

Einschub:

Statt Sprachklassen kann man auf die gleiche Art auch Funktionsklassen definieren. Diese bezeichnet man ebenfalls mit DTime (T), NTime (T), DSpace (S) und NSpace (S).

Nichtdeterministische Berechnungen haben wir bei Algorithmen noch nicht kennen gelernt. Solche Berechnungen entstehen, wenn man "nicht-deterministische" Sprachelemente in die Programmiersprache einführt. Ein Beispiel kann ein Sprachelement "or" sein, das eine willkürliche Auswahl zwischen Anweisungen bezeichnet. Beispiel: (X:=X+1 or X:=X+2);
Hierbei kann man willkürlich eine der beiden Anweisungen auswählen. Führt irgendeine Folge dieser willkürlichen Auswahlen zu einer endenden Berechnung, so liegt ein mögliches Ergebnis vor (es kann mehrere geben, wenn man sich anders entscheidet).

Einschub:

Auch hier kümmern wir uns generell nicht um Konstanten, sondern nur um die Abhängigkeit von der Länge der Eingabe.

Man kann dann leicht folgern:

DTime (T) $\subseteq$ NTime (T), DSpace (S) $\subseteq$ NSpace (S).

DTime (T) $\subseteq$ DSpace (T), NTime (T) $\subseteq$ NSpace (T).

Die Komplexitätstheorie befasst sich mit den Zusammenhängen zwischen solchen Klassen und mit der Einordnung konkreter Probleme in die Hierachie von Komplexitätsklassen.

Einschub: Deterministisch polynomielle Probleme **P** und nichtdeterministisch polynomielle Probleme **NP**.

Zwei spezielle für die Praxis wichtige Komplexitätsklassen:

DTime (n) = DTime (id) mit id(n) = n für alle $n \in \mathbb{N}_0$.

Für ein Polynom p(n) vom Grad k ist:

DTime (n^k) = DTime (p(n)), klar wegen der O-Klassen.

$$\mathbf{P} = \bigcup_{k \geq 0} \text{DTime}(n^k) \qquad \mathbf{NP} = \bigcup_{k \geq 0} \text{NTime}(n^k)$$

Berühmtestes Problem der Informatik derzeit:

Ist **P** = **NP** oder nicht?

(Man vermutet **P** ≠ **NP**, es gibt aber bisher keinen Beweis.)

6.4 Beispiele

6.4.1 Beispiel Vollständige Behandlung des ggT (oder gcd), größter gemeinsamer Teiler zweier natürlicher Zahlen
(im Englischen: gcd = greatest common divisor)

Sei $n \in \mathbb{N}_0$ eine natürliche Zahl. Sei
$T(n) = \{d \in \mathbb{N} \mid d \text{ teilt } n$, d.h., es gibt ein $c \in \mathbb{N}_0$ mit $c \cdot d = n\}$
die Menge der Teiler von n (nur positive Zahlen).

Wegen $1 \in T(n)$ ist T(n) niemals die leere Menge.
Für $n>0$ gilt: Wenn $d \in T(n)$ ist, so ist $d \leq n$.
Speziell gilt: $T(0) = \mathbb{N}_0$.

Bilde zu zwei Zahlen $a, b \in \mathbb{N}_0$ den Durchschnitt $T(a) \cap T(b)$. Diese Menge ist nicht leer, da sie mindestens die Zahl 1 enthält. Das maximale Element in diesem Durchschnitt heißt der größte gemeinsame Teiler von a und b, bezeichnet als ggT(a,b). Außer für a=b=0 ist er stets eindeutig bestimmt.

Für $a, b, d \in \mathbb{N}_0$ mit $a \neq 0$ oder $b \neq 0$ gilt also ggT(a,b) = d genau dann, wenn d sowohl a als auch b teilt und wenn für alle natürlichen Zahlen d', die sowohl a als auch b teilen, stets $d' \leq d$ gilt.

Zu a und b kann man ggT(a,b) daher berechnen, indem man alle Teiler von a und alle Teiler von b berechnet und hieraus die maximale Zahl ermittelt, die in beiden Teilermengen vorkommt.

Um die Teilbarkeit zu testen, verwendet man die Operation mod (= "modulo" = Rest bei der ganzzahligen Division). Für zwei natürliche Zahlen x und y mit $y \neq 0$ gilt:
y ist genau dann ein Teiler von x, wenn (x mod y) = 0 ist.

Aus der Definition des ggT erhält man also folgenden Algorithmus:

Setze eine Variable ggt anfangs auf den Wert 1.
Sei "min" das Minimum der Zahlen a und b,
dann: prüfe für i = 2, 3, 4, ..., min,
ob i ein Teiler von b und ein Teiler von a ist;
trifft dies zu, dann setze jeweils ggt := i.

Am Ende besitzt ggt den Wert ggT(a,b), da bei dem aufsteigenden Testen das größte i, das beide Zahlen teilt, in der Variable ggt gespeichert wird.

Wir schreiben diesen Algorithmus "ggT_elementar" mit unseren Sprachelementen formal korrekt auf. Wir verwenden für die Variablen hier große Anfangsbuchstaben.

```
program ggT_elementar is
declare natural A, B, I, ggt, Min_a_b;
begin read (A); read (B);
        if B ≤ A then Min_a_b := B else Min_a_b := A fi;
        ggt := 1;
        for I := 2 to Min_a_b do
              if (A mod I = 0) and (B mod I = 0) then ggt:=I fi
        od;
        write (ggt)
end
```

[Im Falle a=b=0 gibt dieser Algorithmus jedoch den Wert 1 aus. Schreiben Sie den Algorithmus so um, dass er genau in diesem Fall den Wert 0 ausgibt.]

Wie lange läuft dieser Algorithmus?
Offenbar wird die for-Schleife genau (Min_a_b - 1) Mal durchlaufen. Für große Zahlen a und b kann man diesen "elementaren ggT-Algorithmus" daher nicht verwenden.

Man erhält ein schneller arbeitendes Verfahren, *wenn man die Eigenschaften ausnutzt*, die der ggT(a,b) besitzt.

Eigenschaften des ggT:

(1) $ggT(a,b) = ggT(b,a)$ für alle $a,b \in \mathbb{N}_0$, "Symmetrie",

(2) $ggT(a,0) = ggT(a,a) = a$ für alle $a \in \mathbb{N}$,

(3) $ggT(a,b) = ggT(a-b, b)$ für alle $a,b \in \mathbb{N}_0$ mit $a \geq b$,
$ggT(a,b) = ggT(a+b, b)$ für alle $a,b \in \mathbb{N}_0$,

(4) $ggT(a,b) = ggT(b, a \bmod b)$ für alle $a,b \in \mathbb{N}_0$ mit $a \geq b$.

Beweise: zu (1) und (2): Diese ersten beiden Zeilen folgen direkt aus der Definition und der Tatsache $T(0) = \mathbb{N}_0$.

zu (3): Wenn eine Zahl t sowohl a als auch b teilt, dann gibt es zwei Zahlen c_1 und c_2 mit $c_1 \cdot t = a$ und $c_2 \cdot t = b$. Hieraus folgt $a-b = c_1 \cdot t - c_2 \cdot t = (c_1 - c_2) \cdot t$, d.h., t teilt auch a-b. Folglich teilt der ggT(a,b) auch die Zahl a-b (für $c_1 - c_2 \geq 0$, d.h., für $a \geq b$). Offenbar teilt t stets auch a+b (ersetze einfach "-" durch "+").

Eigenschaften des ggT:

(1) $ggT(a,b) = ggT(b,a)$ für alle $a,b \in \mathbb{N}_0$, "Symmetrie",

(2) $ggT(a,0) = ggT(a,a) = a$ für alle $a \in \mathbb{N}$,

(3) $ggT(a,b) = ggT(a-b,b)$ für alle $a,b \in \mathbb{N}_0$ mit $a \geq b$,
$ggT(a,b) = ggT(a+b,b)$ für alle $a,b \in \mathbb{N}_0$,

(4) $ggT(a,b) = ggT(b, a \bmod b)$ für alle $a,b \in \mathbb{N}_0$ mit $a \geq b$.

Beweis zu (3), Fortsetzung:
Es sei $d = ggT(a,b)$. Wenn $d' = ggT(a-b,b)$ ist, so teilt d' die Zahl b und nach dem soeben Bewiesenen ist d' auch ein Teiler von $(a-b)+b = a$. Weil d der ggT(a,b) ist, muss daher $d' \leq d$ sein. Andererseits ist d (wie jeder Teiler von a und b) auch Teiler von b und a-b, woraus $d \leq d'$ folgt, da d' der ggT(a,a-b) ist. Aus beiden Ungleichungen folgt $d'=d$, d.h., $ggT(a,b) = ggT(a-b,b)$. Analog folgert man $ggT(a,b) = ggT(a+b,b)$.

Eigenschaften des ggT:

(1) $ggT(a,b) = ggT(b,a)$ für alle $a,b \in \mathbb{N}_0$, "Symmetrie",

(2) $ggT(a,0) = ggT(a,a) = a$ für alle $a \in \mathbb{N}$,

(3) $ggT(a,b) = ggT(a-b, b)$ für alle $a,b \in \mathbb{N}_0$ mit $a \geq b$,
$ggT(a,b) = ggT(a+b, b)$ für alle $a,b \in \mathbb{N}_0$,

(4) $ggT(a,b) = ggT(b, a \bmod b)$ für alle $a,b \in \mathbb{N}_0$ mit $a \geq b$.

Beweis zu (4): (hier: die Aussagen gelten auch für b = 0)
Iteriere die Gleichung (3), woraus (4) mit (1) unmittelbar folgt:

$ggT(a,b) = ggT(a-b, b)$ für alle $a,b \in \mathbb{N}_0$ mit $a \geq b$,
$ggT(a-b,b) = ggT(a-2 \cdot b, b)$ für alle $a,b \in \mathbb{N}_0$ mit $a-b \geq b$,
$ggT(a-2 \cdot b, b) = ggT(a-3 \cdot b, b)$ für alle $a,b \in \mathbb{N}_0$ mit $a-2 \cdot b \geq b$

usw., bis man $a-(k-1) \cdot b \geq b > a-k \cdot b$ für $k = a \operatorname{div} b$ erreicht, also:

$ggT(a-(a \operatorname{div} b - 1) \cdot b, b) = ggT(a-(a \operatorname{div} b) \cdot b, b) = ggT(a \bmod b, b)$
für alle $a,b \in \mathbb{N}_0$ mit $a \geq b > 0$.

Aus Eigenschaft (4) erhalten wir sofort einen Algorithmus zur Berechnung des ggT, wobei Eigenschaft (2) als Terminierungs-bedingung dient. Halb umgangssprachliche Formulierung:

```
read(A); read(B);     -- falls A=B=0 ist, das Verfahren abbrechen
if A < B then "vertausche die Werte von A und B" fi;
while B ≠ 0 do
        nächster Wert von B wird A mod B;
        nächster Wert von A wird der Wert des alten B
od;
write(A)
```

Dieser Algorithmus endet, da wegen B > A mod B der Wert von B in jedem Schleifendurchlauf kleiner wird. Irgendwann wird daher B gleich 0 und die while-Schleife bricht ab.

Um zwei Werte auszutauschen oder einen "alten Wert" zuzuweisen, benötigt man Variablen zur Zwischenspeicherung.

Man kann "vertausche die Werte von A und B" nicht einfach durch A:=B; B:=A realisieren, da dann zwar A den Wert von B erhält, aber anschließend auch B diesen Wert bekommt, da der alte Wert von A ja bereits überschrieben wurde.

Wir verwenden also eine weitere Variable H, die im Programm sonst nicht benötigt wird, und setzen:

H := A; A := B; B := H

So gehen wir auch im Schleifenrumpf (= in der Anweisung zwischen do und od) vor.

```
program euklid1 is
declare natural A, B, H;
begin read (A); read (B);
  if (A > 0) or (B > 0) then
     if A < B then H := A; A := B; B := H fi;
     while B ≠ 0 do
          H := A mod B; A := B; B:= H od
  fi;
  write (A)
end
```

Hinweis: Genau dann, wenn beide Eingabewerte 0 sind, wird der Wert 0 ausgegeben.

Nun zur Komplexität. Zusätzlichen Platz braucht der Algorithmus kaum, nur den Speicherplatz für H.

Frage zur Zeitkomplexität: Wie lange arbeitet der Euklidische Algorithmus, bis er anhält?

Beispielrechnung mit a = 144, b = 89:

$$\begin{aligned} &ggT(144,89) = ggT(89,55) = ggT(55,34) = ggT(34,21) \\ &= ggT(21,13) = ggT(13,8) = ggT(8,5) = ggT(5,3) \\ &= ggT(3,2) = ggT(2,1) = ggT(1,1) = ggT(1,0) = 1. \end{aligned}$$

Dieses Beispiel benötigt 11 Schleifendurchläufe.

Algorithmus: -- es ist hier ggT(a,0) = a, auch für a=0.

```
declare natural A, B, R;
begin  read (A); read (B);
       while B≠0 do  R:=A mod B; A:=B; B:=R  od;
       write (A)
end
```

Werteverlauf für die Variablen A und B bei Eingabe von a und b:

A	B
a	b
b	a mod b
a mod b	b mod (a mod b)
b mod (a mod b)	(a mod b) mod (b mod (a mod b))

Kann man etwas über die Größe dieser Werte sagen?

Hilfssatz:

b mod (a mod b) < b/2, für alle a ≥ b > 0, a mod b ≠ 0, d.h.,

nach spätestens zwei Schleifendurchläufen hat sich der Wert von B mehr als halbiert.

Beweis: Es ist immer 0 ≤ a mod b < b.

Fall 1: 0 ≤ a mod b ≤ b/2.
Dann gilt: b mod (a mod b) < a mod b ≤ b/2,
da für y > 0 stets x mod y < y ist.

Fall 2: b/2 < a mod b < b.
Dann gilt: b mod (a mod b) = b - (a mod b) < b - b/2 = b/2,
da für x/2 < y < x stets x mod y = x - y ist.

Damit ist der Hilfssatz bewiesen.

Folgerung:

Der Euklidische Algorithmus durchläuft höchstens $2 \cdot \log(b)$ mal seine Schleife.

Die uniforme Zeitkomplexität des Euklidischen Algorithmus

```
begin  read (A); read (B);
       while B≠0 do  R:=A mod B; A:=B; B:=R  od;
       write (A)
end
```

beträgt somit höchstens $8 \cdot \log(b) + 4$ Zeiteinheiten.
Da $\log(a) + \log(b) = n$ die Länge der Eingabe ist und $\log(b)$ somit in der Größenordnung von n liegt, so folgt:
Der Euklidische Algorithmus besitzt eine lineare uniforme Zeitkomplexität (d.h., er liegt in DTime (n)). Diese günstige Komplexitätsklasse liegt aber an der Uniformität.

Schwäche der uniformen Zeitkomplexität:

Sie misst, wie viele elementare Anweisungen ausgeführt und wie viele Ausdrücke ausgewertet werden, aber sie gibt keine Auskunft darüber, wie aufwändig die Durchführung einer elementaren Anweisung oder die Ausrechnung eines Ausdrucks ist.

Wie lange dauert denn die Wertzuweisung R := A mod B ? Der ungünstigste Fall liegt vor, wenn A k Stellen und B ungefähr k/2 Stellen lang ist. Im Binärsystem führt man dann k/2 Subtraktionen der Länge k/2 aus, d.h., die Berechnung des Restes A mod B erfordert zeichenweise $O(k^2)$ Schritte. Da k beim Euklidischen Algorithmus in der Größenordnung von n liegt (k = log(a) ist meist größer als n/2), erfordert der obige Algorithmus also in Wahrheit $O(n^3)$ Schritte.

Dennoch hat die uniforme Komplexität gewisse Vorteile. In den meisten praktischen Fällen wird die mod-Funktion durch einen Coprozessor für natürliche Zahlen, die in einem "normalen" Bereich liegen, oder mit Hilfe einer Software-simulation berechnet. In solchen Fällen scheint die Modulo-Bildung in konstanter Zeit abzulaufen, so dass sich der Euklidische Algorithmus in der Praxis meist wie ein Linear-zeitalgorithmus verhält.

Hinweis: Man kann zeigen, dass das Problem, den ggT zeichenweise zu berechnen, bereits in $O(n^2)$ Schritten möglich ist. Versuchen Sie sich einmal an diesem Problem. Oder suchen Sie in der Literatur /im Internet nach dem "binären euklidischen Algorithmus".

■

6.4.2 Beispiel "Quersumme"

Aufgabe: Bilde die Quersumme einer binären Folge, d.h.:
$quer(z_{n-1}z_{n-2}...z_1z_0)$ = Binärdarstellung der Anzahl der Einsen unter den z_i.

Erneut ist das Eingabealphabet $\Sigma = \{0,1\}$. Anfangs steht eine Folge aus Nullen und Einsen auf der Eingabe und es wird stets das nächste Zeichen gelesen (Datentyp character).

Idee: Man verwende eine Variable Z für die Anzahl der Einsen. Man lese die Eingabe zeichenweise ein und addiere bei jeder gefundenen Eins eine 1 auf Z. Wenn diese Addition in konstanter Zeit erfolgt (uniforme Komplexität!), dann liegt die Berechnung von "quer" in O(n).

Stimmt dies auch zeichenweise??

(Überraschenderweise *ja*, bitte genau selbst durchdenken!)

6.4.3 Beispiel "Palindrome"

Aufgabe: Stelle fest, ob die Eingabe ein Palindrom ist, d.h., ob das Eingabewort w gleich seinem gespiegelten Wort w^R ist. Falls ja, drucke eine "**1**", anderenfalls eine "**0**" am Ende aus.

Hinweis: w^R ist das rückwärts gelesene Wort w.
Formal: Gegeben sei ein Alphabet Σ. Die Spiegelung ^{R} ist eine bijektive Abbildung von Σ^* nach Σ^*, die induktiv definiert ist durch:

$$\varepsilon^R = \varepsilon \quad \text{und} \quad (aw)^R = w^R a \quad \text{für alle } a \in \Sigma \text{ und alle } w \in \Sigma^*.$$

Bezeichnung: Ein Wort, das vorwärts und rückwärts gelesen gleich ist, nennt man ein Palindrom.

$$\text{Pal} = \{ w \in \Sigma^* \mid w = w^R \} \subset \Sigma^*$$

ist die Menge der Palindrome über dem Alphabet Σ.

Palindrome kann man folgendermaßen charakterisieren:

(1) Das leere Wort ε ist ein Palindrom.

(2) Jedes Zeichen $a \in \Sigma$ ist ein Palindrom.

(3) Wenn a ein Zeichen und w ein Palindrom sind, dann ist auch awa ein Palindrom.

(4) Alle Palindrome werden nach den Regeln (1) bis (3) gebildet.

Entsprechend dieser Charakterisierung kann man Palindrome wie folgt erkennen: Falls das Wort höchstens die Länge 1 hat, so ist es ein Palindrom. Anderenfalls vergleiche das erste mit dem letzten Zeichen. Falls diese gleich sind, streiche sie weg und wende das Verfahren auf das restliche Wort an, anderenfalls lag kein Palindrom vor.
Dies dauert uniform O(n) Schritte, wenn man die Eingabe in einem Feld array [1..n] of character W speichert.

6.4.4 Beispiel "wiederholungsfreie Wörter"

Es sei Σ ein Alphabet. Ein Wort $w \in \Sigma^*$ heißt wiederholungsfrei, wenn es keine Wörter x, y und z gibt mit $w=xyyz$ (mit $y \neq \varepsilon$). Kein echtes Teilwort kommt also in w zweimal hintereinander vor. Insbesondere darf kein Buchstabe auf sich selbst folgen.

Ist Σ ein- oder zweielementig, so gibt es nur endlich viele wiederholungsfreie Wörter (bitte ausprobieren!).

Ist Σ dagegen mindestens dreielementig, so gibt es unendlich viele wiederholungsfreie Wörter. Über $\Sigma = \{a, b, c\}$ kann man beliebig lange wiederholungsfreie Wörter konstruieren:

a b a c b a b c a b a c b c a b c a c b a b c ...

Es sei $WF = \{w \in \Sigma^* \mid w \text{ ist wiederholungsfrei}\}$.
Aufgabe: Entscheide zu $w \in \Sigma^*$, ob w in WF liegt oder nicht, d.h.: Berechne die charakteristische Funktion χ_{WF} zu WF.

Lösungsansatz, deterministisch:

Sei Σ mindestens dreielementig. Sei $n = |w|$ und $w = w_1 w_2 \ldots w_n$ mit $w_i \in \Sigma$ für i =1, 2, ..n. Falls $n \leq 1$, dann ist $w \in$ WF.

Anderenfalls prüfe für jede Position i ($1 \leq i \leq n-1$) und für jede Länge k (mit $1 \leq k \leq (n-i+1)$ div 2 bzw. $i+2k-1 \leq n$), ob $w_i w_{i+1} .. w_{i+k-1} = w_{i+k} w_{i+k+1} \ldots w_{i+2k-1}$ gilt.

Trifft dies für kein i und k zu, so ist w wiederholungsfrei.

Wir formulieren diese Überprüfung, indem wir ein neues Sprachelement exit verwenden. Dieses hat die Syntax:
exit_statement ::= exit [*loop*-name] [when condition];

Bedeutung: Verlasse die umgebende Schleife (oder die Schleife mit dem Namen *loop*-name), sofern die Bedingung hinter when zutrifft. (Man darf jetzt Schleifen einen Namen geben, durch Doppelpunkt abgetrennt, siehe Programm.)

Mit exit kann man Schleifen verlassen. Gibt man nichts an, so wird die innerste Schleife, die die exit-Anweisung umfasst, verlassen. Will man mehrere Schleifen gleichzeitig verlassen, wie in unserem Fall, so muss die Schleife, die zu verlassen ist, einen Namen erhalten, den man in der exit-Anweisung angibt.

```
Schleife_i:
  for i := 1 to n-1 do
    for k := 1 to (n-i+1) div 2 do
      wh := true;
      for j := i to i+k-1 do wh := wh and (W[j]=W[j+k]) od;
      exit Schleife_i when wh
    od
  od;
if wh then "das Eingabewort besitzt eine Wiederholung"
else "das Eingabewort ist wiederholungsfrei" fi;
```

Zeitaufwand dieses Algorithmus?

Der am längsten dauernde Fall liegt vor, wenn das Wort wiederholungsfrei ist. Dann werden alle Kombinationen durchprobiert.
Hierbei können i und k in der Größenordnung von n/2 liegen, d.h., die beiden äußeren Schleifen benötigen bereits $O(n^2)$ Schritte.
Die innerste Schleife erfordert nochmals bis zu n/2 Schritte. Insgesamt erhalten wir somit einen Zeitaufwand von $O(n^3)$.

Groß-Oh gibt nur einen obere Schranke an. Könnte es sein, dass das Verfahren in Wahrheit schneller als in kubischer Zeit arbeitet? Dies kann man beweisen, indem man genau ausrechnet, wie oft die Anweisungen in der innersten Schleife "for j ..." durchlaufen werden:

$$\sum_{i=1}^{n-1} \sum_{k=1}^{\frac{n-i+1}{2}} \sum_{j=i}^{i+k-1} 1 = \sum_{i=1}^{n-1} \sum_{k=1}^{\frac{n-i+1}{2}} k = \sum_{i=1}^{n-1} \frac{1}{2} \frac{n-i+1}{2} \frac{n-i+3}{2}$$

$$= \frac{1}{8} \sum_{i=2}^{n} i\,(i+2) = \ldots = \frac{n^3}{24} + O(n^2) = \Theta(n^3).$$

Das Verfahren braucht also im schlechtesten Fall (= die Eingabe ist wiederholungsfrei) kubisch viele Schritte!

Ein zusätzlicher *Speicheraufwand*, außer konstant viel Platz für die Hilfsvariablen i, j, k, wh und Zwischenergebnisse wie (n+i-1) div 2, entsteht nicht, da der Algorithmus nur auf dem Wort W arbeitet. Die uniforme zusätzliche Platzkomplexität beträgt also O(1).

Einspruch: Genau genommen stimmt dies aber nicht. Es fällt nämlich zusätzlicher Speicherplatz in Abhängigkeit von der Länge der Eingabe an.

Denn es müssen die natürlichen Zahlenwerte von i, j, k und n dargestellt werden.

Um die Zahl n darzustellen, braucht man log(n) Ziffern in der Binärdarstellung und in jeder anderen Darstellung ebenfalls O(log(n)) Ziffern. Für wh braucht man dagegen nur eine Ziffer.

Also erfordert der Algorithmus insgesamt zusätzlichen Speicherplatz in der Größenordnung 4log(n), also O(log(n)). In der Praxis ist dies eine geringe Größe.

(Hier sieht man die "logarithmische Platzkomplexität".)

6.4.5 Multiplikation natürlicher Zahlen

Wie lange dauert die zeichenweise Multiplikation, die wir in der Schule kennen gelernt haben? Dort wird für die Multiplikation a·b ein Schema aufgeschrieben, das an ein Parallelogramm erinnert; dessen Fläche ist log(a)·log(b). Wenn beide Zahlen a und b ungefähr n/2 Stellen besitzen, so dauert die Multiplikation zeichenweise also $O(n^2)$ Schritte.

Als Programm nimmt man gewisse Modifikationen vor, um statt des Parallelogramms immer nur eine Zeile bzw. eine Variable mitzuführen. Eine einfache Technik lautet: Sei Z (anfangs 0) das angestrebte Ergebnis, so gilt

$Z + A \cdot B = Z + (2A) \cdot (B/2)$, sofern B gerade ist,

$Z + A \cdot B = (Z + A) + A \cdot (B-1)$, sofern B ungerade ist.

Dies ergibt folgendes Verfahren:

Algorithmus:

```
declare natural A, B, Z;
begin  read(A); read(B);
       Z:=0;
       while B > 0 do
           if (B mod 2 = 0) then B := B div 2; A := A+A
           else B := B-1; Z := Z+A fi  od;
       write(Z)
end
```

Wenn B>0 ist, so beginnt die Binärdarstellung von B mit einer 1 und somit ist B im letzten Schleifendurchlauf 1; dort wird also spätestens der Wert von Z verändert.

[*Spezialhinweis zur Semantik:* Die Schleifeninvariante lautet {Z+A·B = a·b}, wenn a und b die eingelesenen Anfangswerte sind. D.h.: Obiger Algorithmus berechnet korrekt das Produkt.]

Wie lange dauert dieser Algorithmus für die Eingaben a, b?
Die uniforme Zeitkomplexität liefert:

```
declare natural A, B, Z;
begin  read(A); read(B);        2 Schritte
       Z:=0;                    1 Schritt
       while B > 0 do           1 Schritt                    2k+1 mal
           if (B mod 2 = 0) then    1 Schritt
                B := B div 2; A := A+A      2 Schritte
           else
                B := B-1; Z := Z+A  fi  od;  2 Schritte      2k mal
       write(Z)                 1 Schritt
end
```

Insgesamt: 8k + 5 Schritte, mit k = log(b) und wegen $k \approx n/2$ liegt dies uniform in DTime(n).

Betrachte die elementaren Anweisungen im Detail. Alle Zahlen seien binär dargestellt. Die Eingabelänge sei n = log(a) + log(b). Damit ist auch die Ausgabe durch n Stellen beschränkt.

Die Anweisungen read(A), read(B) und write(Z) dauern zeichenweise so lange, wie die Zahlen lang sind, also O(n). Z:=0 dauert einen Schritt. Ebenso kann man "B>0" in einem Schritt entscheiden. Der Ausdruck (B mod 2 = 0) kostet einen Schritt, da man nur das letzte Zeichen von B auf Null testen muss. B := B div 2 und A := A+A kosten ebenfalls jeweils einen Schritt, da nur die letzte Null gestrichen bzw. eine Null angehängt werden muss. Auch B := B-1 kostet nur einen Schritt, da B ja ungerade ist und daher nur die letzte 1 in eine 0 umgewandelt werden muss. Z := Z+A kann dagegen bis zu n Schritte dauern. *Es ist also der else-Zweig, der die Laufzeit im Wesentlichen bestimmt.*

In den else-Zweig gelangt man immer, wenn B ungerade ist. Dies ist genau dann der Fall, wenn in der Binärdarstellung von B am Ende eine 1 steht. Da im then-Zweig die letzte Ziffer von B jeweils gestrichen wird, so wird der else-Zweig also genau so oft durchlaufen, wie Einsen in der Binärdarstellung von b auftreten. Der then-Zweig dagegen wird genau so oft durchlaufen, wie b lang ist, also log(b) mal.

Es sei n die Länge der Eingabe und es sei eins(b)

$$1 \leq \text{eins}(b) \leq \log(b) \leq n \in O(n)$$

die Anzahl der Einsen in der Binärdarstellung der Eingabezahl b, dann wird der else-Zweig insgesamt eins(b) Mal durchlaufen (genau: 3·eins(b) Schritte für "B>0", "B $\underline{\text{mod}}$ 2 = 0" und "B:=B-1" sowie höchstens eins(b)·n Schritte für die Addition "Z:=Z+A" in der while-Schleife) und der then-Zweig höchstens n Mal; es ergeben sich insgesamt höchstens 4·n Schritte.

Somit erfordert die zeichenweise Zeitkomplexität des obigen Algorithmus zur Multiplikation höchstens

$$\text{eins}(b)\cdot(n+3) + 4\cdot n + 5 \in O(n^2) \text{ Schritte}$$

da ja eins(b) im schlechtesten Fall n sein kann.

Der Algorithmus sieht zwar gegenüber der Schulmethode des Multiplizierens geschickter aus, hat aber die gleiche quadratische Zeitabhängigkeit.

■

Nochmals der Hinweis:
Berechnet man die Komplexität $t_A(w)$ bzw. $s_A(w)$ so, dass jede elementare Anweisung und jede Abfrage genau einen Schritt benötigen bzw. dass jede Zahl genau einen Speicherplatz belegt, so spricht man von der **uniformen Komplexität**.
Berechnet man die Komplexitäten so, dass jede Operation so viel Zeit kostet, wie die zeichenweise Ausführung erfordert, bzw. dass Werte so viel Platz belegen, wie sie Zeichen haben, so spricht man von der **logarithmischen Komplexität**.

In der Praxis berechnet man zunächst die uniforme Komplexität: Sie gibt meist eine hinreichend genaue Orientierung über den zu erwartenden Aufwand. Erst für eine genaue Abschätzung bzw. bei der Programmierung betrachtet man die logarithmische Komplexität, die den tatsächlichen Aufwand genauer wiedergibt.

6.4.6 Gibt es einen schnelleren Algorithmus für die Multiplikation? *Oder kann man beweisen, dass es kein schnelleres Verfahren geben kann?*

Wenn man zwei natürliche Zahlen (ungleich Null), die jeweils k bzw. m Ziffern lang sind, miteinander multipliziert, so entsteht eine Zahl, die (k+m-1) oder (k+m) Ziffern besitzt.
Da also das Ergebnis der Multiplikation nur so lang sein kann, wie die beiden Multiplikanten zusammen, könnte es eventuell einen Algorithmus geben, der die Multiplikation proportional zur Eingabelänge n (=k+m+2) durchführt.
Dies würde bedeuten, dass man die Multiplikation bis auf einen konstanten Faktor genau so schnell ausführen könnte wie die Addition.
Niemand glaubt, dass dies mögilch ist. Dennoch ist es bisher nicht gelungen zu *beweisen*, dass die Multiplikation deutlich mehr Zeit als die Addition erfordert.

Nun wollen wir ein Verfahren vorstellen, welches zeichen-weise schneller als mit der Zeitkomplexität $O(n^2)$ multipliziert.

Gegeben seien zwei k-stellige natürliche Zahlen x und y (mit $k \approx n/2$); die Länge k sei eine gerade Zahl. Setze $p=k/2$. Dann lassen sich x und y schreiben in der Form:

$$x = A \cdot 2^p + B \quad \text{und} \quad y = C \cdot 2^p + D,$$

wobei A, B, C, D mindestens 0 und echt kleiner als 2^p sind. Dann gilt

$$x \cdot y = (A \cdot 2^p + B) \cdot (C \cdot 2^p + D) = A \cdot B \cdot 2^{2p} + (A \cdot D + B \cdot C) \cdot 2^p + B \cdot D.$$

Man kann also die Multiplikation zweier k-stelliger Zahlen durchführen, indem man sie auf 4 Multiplikationen von halber Länge k/2 und drei Additionen zurückführt. Dies ergibt erneut eine quadratische Zeitkomplexität. Kommt man vielleicht mit weniger als vier Multiplikationen halber Länge aus?

Es gilt: $(A \cdot D + B \cdot C) = (A-B) \cdot (D-C) + A \cdot C + B \cdot D$,
wie man durch Ausrechnen leicht nachprüft.

Somit erhalten wir aus obiger Gleichung:

$$x \cdot y = A \cdot C \cdot 2^{2p} + (A \cdot D + B \cdot C) \cdot 2^{p} + B \cdot D$$
$$= A \cdot C \cdot 2^{2p} + ((A-B) \cdot (D-C) + A \cdot C + B \cdot D) \cdot 2^{p} + B \cdot D$$

wobei A, B, C, D mindestens 0 und echt kleiner als 2^p sind.

Nun haben wir es schon geschafft: Auf der rechten Seite stehen nur noch drei verschiedene Multiplikationen:

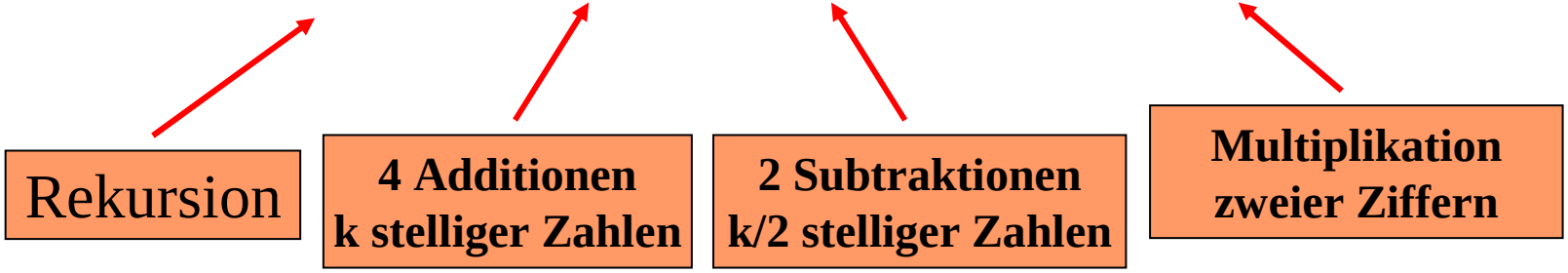
$A \cdot C$, $B \cdot D$ und $(A-B) \cdot (D-C)$

Beachte: Die Multiplikation mit 2^p bzw. 2^{2p} ist keine echte Multiplikation, sondern nur ein Anhängen von p bzw. 2p Nullen, wenn man zur Basis 2 rechnet.

Somit haben wir die Multiplikation zweier k-stelliger Zahlen auf drei Multiplikationen von k/2-stelligen Zahlen zurückgeführt. Der Preis, den wir dafür bezahlen müssen, sind zwei zusätzliche Subtraktionen zweier k/2-stelliger Zahlen und eine zusätzlich Additionen zweier k-stelliger Zahlen. Da aber die Zeit für die Addition und die Subtraktion proportional zur Länge der Zahlen ist, müssten wir dennoch schneller fertig werden. Wir prüfen dies nach.

Wenn t(k) die Anzahl der durchzuführenden Operationen für die Multiplikation zweier k-stelliger Zahlen nach diesem Verfahren ist, dann erhalten wir also folgende Gleichung:

$$t(k) = 3 \cdot t(k/2) + 4 \cdot k + 2 \cdot (k/2) \quad \text{mit} \quad t(1)=1$$



Wie lautet die Lösung dieser Gleichung?

Probieren wir es aus. Ersetzen von t(k/2), t(k/4) usw. entsprechend der Rekursionsformel t(k) = 3·t(k/2) + 5·k ergibt:

$$
\begin{aligned}
t(k) &= 3\cdot t(k/2) + 5\cdot k \\
&= 3\cdot(3\cdot t(k/4) + 5\cdot k/2) + 5\cdot k \\
&= 3\cdot 3\cdot t(k/4) + 3\cdot 5\cdot k/2 + 5\cdot k \\
&= 3\cdot 3\cdot t(k/4) + 5\cdot k\cdot(1 + 3/2) \\
&= 3\cdot 3\cdot(3\cdot t(k/8) + 5\cdot k/4) + 5\cdot k\cdot(1 + 3/2) \\
&= 3\cdot 3\cdot 3\cdot t(k/8) + 5\cdot k\cdot(1 + 3/2 + 9/4) \\
&= 3\cdot 3\cdot 3\cdot(3\cdot t(k/16) + 5\cdot k/8) + 5\cdot k\cdot(1 + 3/2 + 9/4) \\
&= 3\cdot 3\cdot 3\cdot 3\cdot t(k/16) + 5\cdot k\cdot(1 + 3/2 + 9/4 + 27/8) \\
&= \ldots
\end{aligned}
$$

Die allgemeine Formel nach i Schritten lautet daher:

$$t(k) = 3^i \cdot t(k/2^i) + 5 \cdot k \cdot (1 + 3/2 + 9/4 + \ldots + 3^{i-1}/2^{i-1})$$
$$= 3^i \cdot t(k/2^i) + 10 \cdot k \cdot ((3/2)^i - 1)$$

Beachte die geometrische Reihe

$$1 + a + a^2 + a^3 + \ldots + a^m = \frac{a^{m+1} - 1}{a - 1}$$

In unserem Fall ist a = 3/2.

Diese Ersetzungen kann man vornehmen, bis $k = 2^i$ geworden ist, also bis $i = \log(k)$. Dann ist $t(k/2^{\log(k)}) = t(1) = 1$. Wir setzen dies ein und erhalten:

$$\begin{aligned} t(k) &= 3^{\log(k)} \cdot t(k/2^{\log(k)}) + 10 \cdot k \cdot ((3/2)^{\log(k)} - 1) \\ &= k^{\log(3)} + 10 \cdot k \cdot (k^{\log(1,5)} - 1) \\ &= 11 \cdot k^{\log(3)} - 10 \cdot k \approx 11 \cdot k^{1,585} - 10 \cdot k \in O(k^{1,585}) \end{aligned}$$

Beachte hierbei die Formeln:
log ist der Logarithmus zur Basis 2,

$a^{\log(b)} = b^{\log(a)}$ und

$k \cdot k^{\log(1,5)} = k^{1+\log(1,5)} = k^{\log(2)+\log(1,5)} = k^{\log(2 \cdot 1,5)} = k^{\log(3)}$

Die Multiplikation zweier k-stelliger Zahlen lässt sich also zeichenweise in proportional zu $k^{1,585}$ Schritten durchführen.

(Die Schulmethode benötigt k^2 Schritte, siehe früher.)

Satz: Die Multiplikation natürlicher Zahlen liegt in $O(n^{1,585})$.

Geht es noch schneller? Ja. Der beste derzeit bekannte Algorithmus, der Algorithmus nach Schönhage und Strassen (1971), der sich allerdings nur für riesige Zahlen eignet, benötigt

$$O(n \cdot \log(n) \cdot \log(\log(n)))$$ Schritte.

Dies ist schon relativ dicht an der Größenordnung O(n), so dass man vielleicht eines Tages doch einen Algorithmus finden wird, der die Multiplikation in O(n) Schritten - und damit ungefähr so schnell wie eine Addition - durchführt?

6.4.7 Mit dem *Nichtdeterminismus* kann man "schwierige" Probleme leicht beschreiben, z.B. das *Binpacking-Problem:* Es sollen n Gegenstände, die die Gewichte g_1, g_2, ..., g_n besitzen, so in m Behälter gepackt werden, dass in jedem Behälter das Maximalgewicht Max nicht überschritten wird.

Anwendungen:

1. Es sollen n Kisten, die jeweils g_1, g_2, ..., g_n Tonnen wiegen, mit m LKWs transportiert werden, wobei jeder LKW die maximale Zuladung Max Tonnen besitzt. Geht dies?
2. Kann man n Produkte mit m Maschinen in insgesamt Max Zeiteinheiten herstellen, wobei das i-te Produkt g_i Zeiteinheiten zu seiner Herstellung benötigt?
3. Optimierungsproblem:Verteile möglichst viele von n Goldstücken der Gewichte g_1, g_2, ..., g_n so auf m Personen, die jede höchstens Max Gewichtseinheiten tragen können, dass das Gewicht der zurückbleibenden Goldstücke minimal ist.

Formalisierung des BPP als Entscheidungsproblem:

Definition: Binpacking Problem (BPP)

Gegeben: natürliche Zahlen n, m, Max, g_1, g_2, ..., g_n.
Gesucht: eine Abbildung f: {1, 2, ..., n} $\rightarrow$ {1, 2, ..., m},
so dass für jedes i ($1 \leq i \leq m$) gilt:

$$\sum_{f(j)=i} g_j \leq \text{Max}.$$

f(j) = i bedeutet also: Der j-te Gegenstand mit dem Gewicht g_j wird in den i-ten Behälter gelegt.

Um das Problem, ob es zu n, m, Max, g_1, g_2, ..., g_n ein solches f gibt, zu lösen, kann man deterministisch alle m^n Möglichkeiten durchprobieren. Nichtdeterministisch lässt sich eine Lösung dagegen leicht beschreiben.

Nichtdeterministisches Programmstück nach der Methode "guess and check" (raten und dann nachprüfen):

```
declare  natural  n, m, Max;
begin  read(n, m, Max); ...
declare  array [1..n] of natural  g;  array [1..n] of natural  f;
     array [1..m] of natural  B;  Boolean Fehler;  natural i, j;
begin   < Lies die Gewichte g ein >;
     Fehler := false;  for i:=1 to n do B[i] := 0 od;
     for j := 1 to n do                                   } raten (guess)
          ( f[j] := 1 or  f[j] := 2  or ...
               or  f[j] := m-1 or  f[j] := m )  od;
     for j := 1 to n do B[f[j]] := B[f[j]] + g[j];        } prüfen (check)
          if B[f[j]] > Max then Fehler := true fi od;
     if not Fehler then < gib die Lösung f aus > fi
end  end;
```

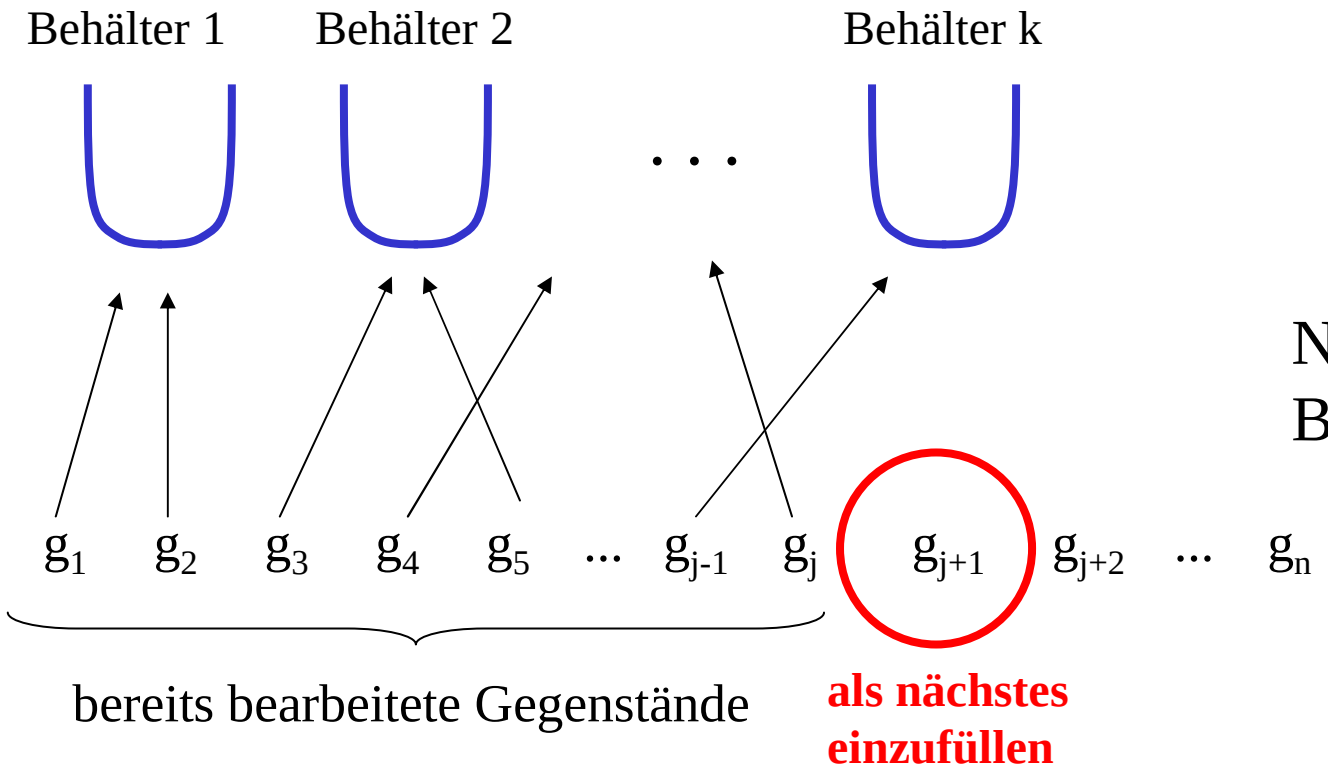
6.4.8 Backtracking

Beachten Sie, dass bei einem nichtdeterministischen Programm der Misserfolg nicht beachtet wird. Nur im Erfolgsfall (also der Fall "Es existiert eine Lösung") wird eine Lösung ausgegeben.

Ein nichtdeterministisches Programm dient der Klärung des Problems; für die Ermittlung einer Lösung ist es in der Praxis kaum hilfreich. Um nun eine Lösung zu finden, geht man alle Möglichkeiten durch: Die verschiedenen Möglichkeiten werden baumartig aufgeschrieben und systematisch durchlaufen und abgeprüft.

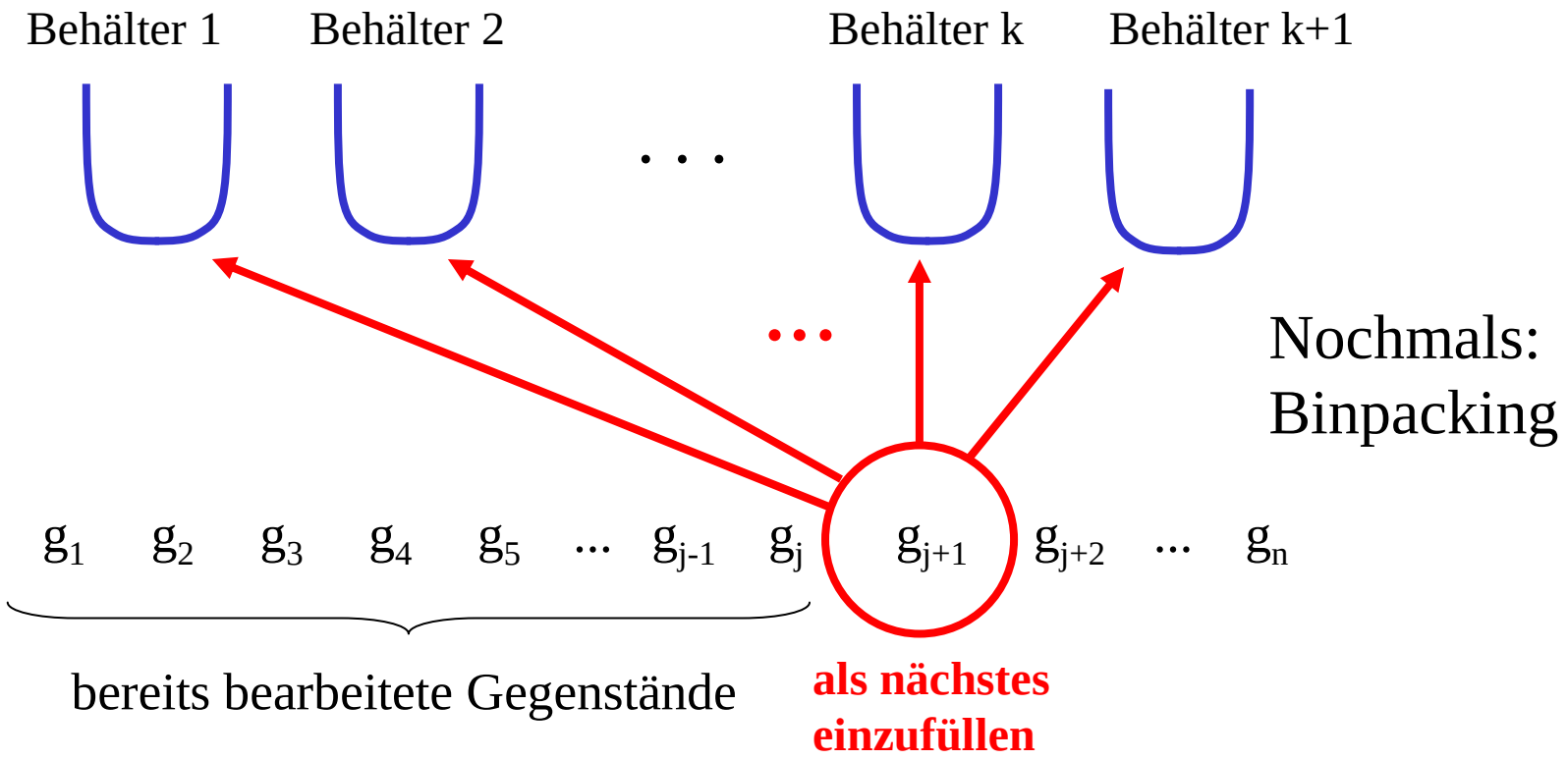
Ein solches systematisches Durchprobieren aller Möglichkeiten bezeichnet man als Backtracking = "Rückverfolgen durch den Lösungsbaum".

Backtracking geht stets von einer vorhandenen Situation aus und reduziert das Problem auf einfachere Probleme. Eine typische Situation beim Füllen der m Behälter ist: Man hat die ersten j Gegenstände bereits konfliktfrei in die ersten k Behälter gefüllt und muss nun den (j+1)-ten Gegenstand einfüllen.



Nochmals: Binpacking

Wir können den (j+1)-ten Gegenstand nun einem der bereits vorhandenen Behälter 1, 2, ..., k zuordnen oder einen neuen Behälter (k+1) verwenden, sofern $k+1 \leq m$, d.h. $k < m$ ist.



Alle diese k+1 Fälle probieren wir im Backtracking durch. Die folgende Prozedur vollzieht genau die obige Rekursion nach.

Entwurf der rekursiven Prozedur (es fehlen noch Details):

```
procedure BTBPP (natural j, k);
declare natural i;
begin
        if  j = n  then  ..... < Lösung gefunden > .....
        else  for i:=1 to k do
                  if  Gegenstand [j+1] passt noch in Behälter i
                        then  BTBPP (j+1, k)  fi   od;
              if k < m then BTBPP (j+1, k+1) fi
        fi
end;
```

Um zu überprüfen, ob der Gegenstand [j+1] noch in den Behälter i passt, verwenden wir wieder das globale Feld array [1..m] of natural B, in welchem wir die Summe der Gewichte aller Gegenstände, die aktuell im jeweiligen Behälter liegen, notieren. Der Gegenstand [j+1] darf in den Behälter i nur gelegt werden, wenn dadurch das Maximalgewicht nicht überschritten wird, also nur, wenn $B[i]+g[j+1] \leq Max$ ist.

So erhalten wir die Verfeinerung des Prozedurschemas (man beachte, dass vor und nach jedem rekursiven Aufruf das Gewicht des jeweiligen Behälters B[i] aktualisiert werden muss):

```
procedure BTBPP (natural j, k);
declare natural i;
begin
    if  j = n  then  < Lösung gefunden >
    else  for i:=1 to k do
            if  B[i] + g[j+1] ≤ Max  then
                B[i]:=B[i]+g[j+1];  BTBPP (j+1, k);
                B[i]:=B[i]-g[j+1]  fi  od;
          if k < m then B[k+1]:=g[j+1];  BTBPP (j+1, k+1);
                B[k+1]:=0  fi
    fi
end;
```

Diese Prozedur stellt bisher nur fest, ob eine Lösung existiert, aber sie gibt keine mögliche Zuordnung zu den Behältern aus. Hierfür müssen wir uns noch zu jedem Gegenstand merken, in welchen Behälter er gelegt wurde. Dies geschieht in dem globalen Feld array [1..n] of natural f.

Wir müssen nicht alle m^n möglichen Abbildungen f durchprobieren. Wir können annehmen, dass der Gegenstand mit der Nummer 1 stets im Behälter 1 liegt, der Gegenstand mit der Nummer 2 stets in einem der Behälter 1 oder 2 usw. Durch Umnummerieren der Behälter lässt sich also stets erreichen, dass $f[j] \leq j$ für alle j=1,2,...,n gilt. Speziell ist dann f[1]=1. In der Prozedur stellen wir automatisch sicher, dass $f[j] \leq j$ für alle weiteren j gilt, indem für den Gegenstand j+1 neben den bereits betrachteten Behältern nur der Behälter k+1 (und nicht k+2, k+3 usw.) ausprobiert wird. Weil m die höchste Nummer eines Behälters ist, brauchen wir also nur Abbildungen f zu betrachten mit:

$f[j] \leq Min(j,m)$, wobei Min(j,m) das Minimum der beiden Zahlen j und m ist.

So erhalten wir BTBPP (Backtrackingprozedur für das Binpackingproblem):

```
procedure BTBPP (natural j, k);
< Beachte: Die Gegenstände 1 bis j sind bereits eingefügt, wobei k Behälter verwendet wurden >
declare natural i;
begin
    if  j = n  then  < Lösung gefunden, drucke das Feld f aus >
    else  for i:=1 to k do
              if  B[i] + g[j+1] ≤ Max  then
                  f[j+1]:=i; B[i]:=B[i]+g[j+1]; BTBPP (j+1, k);
                  B[i]:=B[i]-g[j+1]; f[j+1]:=0  fi  od;
          if k < m then f[j+1]:=k+1; B[k+1]:=g[j+1];
                  BTBPP (j+1, k+1); B[k+1]:=0; f[j+1]:=0  fi
    fi
end;
```

Globale Variablen für die Prozedur BTBPP sind wiederum:

```
natural  Max, m, n;
array[1..n] of natural  g, f;
array[1..m] of natural  B;
```

Der Aufruf der Prozedur BTBPP lautet, sofern bereits alle Daten in die globalen Variablen Max, m, n und g eingelesen wurden:

```
if (n<1) or (m<1) then
        < Abbruch, da keine Lösung zu suchen ist > fi; ...
for j:=1 to n do if g[i]>Max then
        < Abbruch, da keine Lösung möglich > fi od; ...
for i:=1 to m do f[i]:=0 od;
for j:=1 to n do B[j]:=0 od;
B[1]:=g[1]; f[1]:=1; BTBPP(1,1);
```

Manche der obigen Anweisungen erscheinen überflüssig (z.B. f[j+1]:=0 oder for i:=1 to n do f[i]:=0 od;). Wir haben sie dennoch aufgeführt, da sie eventuell sehr hilfreich werden können, wenn Fehler auftreten oder wenn die Prozedur noch von anderen Programmen aufgerufen wird.

Die Übertragung in eine konkrete Programmiersprache ist nun einfach.

Aufgabe: Schreiben Sie das Programm für die Lösung des Binpacking-Problems fertig und testen Sie es an einigen Daten. Messen Sie Zeit und Speicherplatz. Machen Sie sich das systematische Durchprobieren mittels Backtracking klar.

6.5 Entwurfsmethoden für Algorithmen

Für Algorithmen gibt es einige Entwurfsprinzipien. Die vier bekanntesten lauten:

Greedy-Technik (= gieriges Vorgehen): Man arbeitet sich schrittweise zur Lösung vor, indem man in jedem Schritt die Maßnahme, die den größten lokalen Nutzen verspricht, durchführt.
(Beispiele: Dijkstra-Verfahren für kürzeste Wege, Prim-Algorithmus für spannende Bäume, Hillclimbing Techniken)

Divide and Conquer (= teile-und-herrsche): Zerlege das Problem in Teilprobleme, löse diese nach der gleichen Technik und setze dann aus den Teilen eine Lösung zusammen.
(Beispiele: Quicksort, Sortieren durch Mischen, schnellere Multiplikation in $O(n^{1,585})$.)

Dynamisches Programmieren: Setze die Lösung aus allen kleineren Lösungen zusammen.
(Beispiele: Warshall-Algorithmus, optimale Suchbäume, Syntaxanalyse kontextfreier Sprachen.)

Backtracking: Systematisches baumartiges Durchmustern sämtlicher Lösungsmöglichkeiten.
(Beispiele: Binpacking, Erbschaftsproblem (siehe 6.6), Syntaxanalyse kontextsensitiver Sprachen,)

Hinweis für die, die doch noch einmal studieren wollen:
Diese und andere Entwurfsmethoden können Sie im Studium in Lehrveranstaltungen wie "Theorie III für SwT" oder "Entwurf und Analyse effizienter Algorithmen" vertiefen.

6.6 Historisches

Dass Algorithmen lange Laufzeiten haben können, ist seit altersher bekannt. Der Euklidische Algorithmus oder das Newtonverfahren zur Berechnung von Nullstellen sind relativ schnell arbeitende Verfahren. Die Gaußsche Elimination zur Lösung linearer Gleichungssysteme benötigt $O(n^3)$ Schritte, wobei n die Zahl der Variablen ist. Dies ist für die Berechnung per Hand meist schon zu aufwändig.

Die Erfindung von Rechenmaschinen wurde sicher durch die zeitraubenden Rechnungen, vor allem im technischen Bereich, beflügelt. Systematische Untersuchungen zur Komplexität begannen in den 1950er Jahren. Das exponentielle Wachstum mancher Algorithmen führte zu der Frage, ob man die Komplexitätsklasse **P** charakterisieren und ihr Verhältnis zur Klasse **NP**, in der viele praktische Probleme liegen, klären könne.

1971 zeigte S. Cook, dass es in **NP** Probleme mit folgender Eigenschaft gibt: Liegt nur eines dieser Probleme in **P**, so fallen die Klassen **P** und **NP** zusammen. Diese Probleme sind unter dem Begriff "NP-vollständige Probleme" bekannt geworden. Man kennt mittlerweile rund 10.000 solcher Probleme.

Hierzu zählen Zuordnungs- und Optimierungsprobleme wie die Erstellung von Stundenplänen oder optimale Standorte oder Rundreisen mit vorgegebenen Eigenschaften. Als besonders einfach gilt das "Erbschaftsproblem" (engl.: "partition problem"): Gegeben seien n natürliche Zahlen $g_1, g_2, ..., g_n$, gibt es hierzu eine Menge von Indizes $I=\{i_1, i_2, ..., i_r\} \subseteq \{1, 2, ..., n\}$ mit

$$\sum_{j=1}^{r} g_{i_j} = G, \quad \text{wobei } G = \left(\sum_{i=1}^{n} g_i\right) / 2 \quad ?$$

Kann man also die n Zahlen in zwei Teilmengen zerlegen, deren Summen gleich sind?

Falls Sie einmal an diesem einfach aussehenden Problem üben wollen, so stellen Sie für folgende 20 Zahlen, deren Summe 33.480.070 ist (d.h. G = 16.740.035), fest, ob es eine Teilmenge gibt, deren Summe gleich G ist:

1.976.834, 1.864.558, 1.755.621, 1.575.931, 2.169.504,
1.567.429, 2.001.571, 1.682.544, 1.289.337, 1.223.752,
1.884.283, 1.671.449, 1.400.530, 1.547.733, 1.338.626,
1.438.792, 2.010.563, 1.422.589, 1.863.866, 1.794.558

Mittlerweile ist die "Komplexitätstheorie", die eng mit der "Algorithmik" zusammenhängt, ein etablierter Zweig der Informatik. Auch wenn hier viele abstrakte Erkenntnisse gewonnen wurden, so hat man bisher noch keine Technik entdeckt, um für die NP-vollständigen Probleme nachzuweisen, dass es zu ihrer Lösung keine deterministischen Programme geben kann, die in polynomieller Zeit arbeiten.

Immer wieder stieß man auf das Problem, nichtdeterministische Verfahren deterministisch simulieren zu müssen. Es entstanden hunderte von Klassen und Hierarchien und zugleich gute und nützliche Verfahren.

Heute weiß man von sehr vielen Problemen ziemlich genau, in welcher Komplexitätsklasse sie liegen; z.B. lässt sich die Frage, ob eine Zahl eine Primzahl ist, in polynomieller Zeit (bzgl. der Länge der eingegebenen Zahl) lösen. Doch wir wissen noch viel zu wenig über die (Nicht-) Existenz mathematischer Räume, in denen man sehr effizient rechnen und zwischen denen man relativ schnell (mittels Codierungen) hin- und herschalten kann.

Alle diese Fragen werden durch parallele und verteilte Algorithmen noch verschärft. Hier kommen neue Komplexitätsklassen ins Spiel, auf die wir hier nicht eingehen können.

Zur Aufgabe aus 6.2: Untersuchen Sie die Funktion log*.

Hinweise: Man kann die Funktion log* auch induktiv definieren:

$$\log^*(x) = \begin{cases} 0, & \text{für } x < 2, \\ \log^*(\log(x)) + 1, & \text{für } x \geq 2. \end{cases}$$

So lassen sich einige Werte leicht bestimmen:

$\log^*(2) = \log^*(\log(2))+1 = \log^*(1)+1 = 1$

$\log^*(3) = \log^*(\log(3))+1 = \log^*(1.58496...)+1 = 1$

$\log^*(4) = \log^*(\log(4))+1 = \log^*(2)+1 = 2$

$\log^*(40) = \log^*(5.32193...)+1 = \log^*(2.40...)+1+1$

$= \log^*(1.5...)+1+1+1 = 3$

$\log^*(65536) = \log^*(16)+1 = \log^*(4)+2 = 4$

$\log^*(2^{65536}-1) = \log^*(65535.99...)+1 = \log^*(15.99...)+2$

$= \log^*(3.99...)+3 = \log^*(1.99...)+4 = 4$

$\log^*(2^{65536}) = \log^*(65536)+1 = \log^*(16)+2 = \log^*(4)+3 = 5$

Zur Aufgabe aus 6.2 (Fortsetzung): Die Funktion log*.

Die Funktion log* besitzt eine "Umkehrfunktion"
(eingeschränkt auf die natürlichen Zahlen $\mathbb{N}$),
nennen wir sie h: $\mathbb{N} \to \mathbb{N}$, mit den Eigenschaften
$\log^*(h(n)) = n$ und $h(\log^*(n)) = n$ für alle $n \in \mathbb{N}$.
Offfenbar gilt
$h(1) = 2$, $h(2) = 4$, $h(3) = 16$, $h(4) = 65536$, $h(5) = 2^{65536}$.
Die Funktion h erfüllt für alle $n \in \mathbb{N}$ die Eigenschaft

$$h(n+1) = 2^{h(n)}.$$

Diese Funktion h wächst also viel viel schneller als
beispielsweise die Exponentialfunktion 2^n.
h lässt sich folgendermaßen beschreiben:

$$h(n) = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}$$

n Zweien stehen hier insgesamt.

Zur Erinnerung: Ehemals geplanter Aufbau der Veranstaltung "Grundbegriffe der Informatik"

Es werden mehrere Einzelthemen behandelt. *Geplant sind derzeit:*

0. Objekte und Interaktion. Programme.
1. Aufbau einfacher Algorithmen (Kontrollstruktur, elementare Datentypen).
2. Einfache Datenstrukturen (Feld, Verbund, Vereinigung).
3. Begriff der Sprache, Grammatik/BNF und formale Sprache.
4. Datentypen, Beispiele (Boolean, Keller, Prioritätswarteschlange).
5. Iteration und Rekursion. Prozeduren und Funktionen.
6. Zeit- und Platzkomplexität. Groß-O.
7. Unentscheidbare Probleme, Terminierung.
8. Korrektheit und Beweiskalkül.
9. Speicherstrukturen, Speicherbereiche, Halde.
10. Eigene Definition von Programmiersprachen.
11. Listen, Bäume, Graphen.
12. Suchen und Sortieren.
13. Gültigkeitsbereiche, Polymorphie, Vererbung, Klassen.
14. Graphalgorithmen.

Dies schaffen wir nicht mehr. Aber immerhin: über 50% Erfüllungsgrad.

Gliederung des Kapitels 7

7. Unentscheidbare Probleme, Terminierung

7.1 Charakteristika von Algorithmen

Wie lauten notwendige Eigenschaften, die ein Algorithmus erfüllen muss, "damit er ein Algorithmus sein kann"?

Beispielsweise darf ein Algorithmus nicht in einem Schritt eine unendliche Menge von Möglichkeiten abprüfen oder unendlich viele Kopien von sich erzeugen können.

Auch darf ein Algorithmus nicht über unendlich viele verschiedene elementare Handlungen verfügen können; dies dürfen sogar nur beschränkt viele sein, da man anderenfalls keine mechanische Maschine zur Bearbeitung bauen könnte.

Wir listen einige Eigenschaften auf, die aus der Forderung an Algorithmen "realisierbar mit einer mechanischen Maschine" abgeleitet werden können.

7.1.1: Ein Algorithmus ist eine Vorschrift, die die Reihenfolge von durchzuführenden Handlungen (Operationen) auf Daten (Operanden) genau beschreibt. Hierbei muss gelten:

a) Die Daten sind "diskret" aufgebaut und es gibt beschränkt viele ("digitale") Zeichen $\{a_1, ...,a_k\}$, so dass jedes Datum eine endliche Folge dieser Zeichen ist.
b) Die Operationen sind "diskret" aufgebaut. Genauer: Es gibt beschränkt viele Zeichen $\{b_1, ...,b_m\}$, so dass jede Operation einschließlich ihrer Operanden hiermit beschrieben werden kann.
c) Die Vorschrift ist eine endliche Folge von Operationen. Die Vorschrift wird schrittweise abgearbeitet (diskrete Zeitskala).

Hinweis: Eine Menge heißt "diskret", wenn ihre Elemente gut unterscheidbar und mit endlicher Länge darstellbar sind. Erfolgt die Darstellung mit beschränkt vielen Zeichen, so spricht man von einer digitalen Darstellung.

d) Eine der Operationen ist als Startoperation ausgezeichnet.

e) Für jede Operation ist unmittelbar nach ihrer Ausführung bekannt, welches die möglichen (endlich vielen) Folgeoperationen sind oder ob der Algorithmus in nächsten Schritt abbricht/endet.

f) Die Eingabe für die Vorschrift ist eine (eventuell abzählbar unendliche oder auch leere) Folge von Daten (vgl. a).

g) In jedem Schritt (d.h. zu jedem Zeitpunkt) gilt: Die bis dahin bearbeitete oder betrachtete Menge an Daten und durchgeführten Operationen ist endlich.

Hinweis: Ein Algorithmus verfügt über eigene Speicherbereiche für die Daten und für die Vorschrift. Beide Bereiche kann der Algorithmus während seiner Abarbeitung verändern, aber in jedem Schritt nur einen endlichen Bereich. Beide Bereiche sind prinzipiell unbeschränkt, auch wenn zu jedem Zeitpunkt nur ein endlicher Teil betrachtet worden sein kann.

Hinweis: Obige Ausführungen sind keine richtige Definitionen, sondern nur eine Liste von umgangssprachlich formulierten Forderungen.

"Mathematische Maschinen" erfüllen diese Forderungen, aber auch andere "Rechenmaschinen" und die bereits vorgestellten Grammatiken, siehe Kapitel 3.4.

Auch Programme beschreiben Algorithmen. Prüfen Sie die Forderungen an einer Programmiersprache und ihren Programmen nach.

In der Praxis wünscht man sich oft *noch zwei weitere Forderungen, die aber für Algorithmen nicht notwendig sind:*
- die Terminierung und
- den Determinismus.

Dies erläutern wir auf den folgenden Folien.

7.1.2: Determinismus, Nichtdeterminismus

Oft weiß man nicht, welches die nachfolgende Operation sein soll. Dann gibt man nicht nur eine, sondern *mehrere mögliche Folgeoperationen* an. Beispiele sind Spiele: Bei Schach, Skat, Siedler von Catan usw. kann man in jedem Zug einen unter vielen auswählen. Ein Verfahren, welches in mindestens einer Situation für den nächsten Schritt mehrere Operationen alternativ zulässt, bezeichnet man als nichtdeterministisch.

Solange diese Menge der zulässigen Operationen endlich ist, lässt sich das Problem durch einen "normalen" deterministischen Algorithmus simulieren (siehe Backtracking in 6.4.8). Ist diese Menge jedoch unendlich, so liegt kein Algorithmus mehr vor.

In der Praxis verlangt man meist, dass Algorithmen deterministisch sein müssen, d.h., nach Abarbeitung jeder Operation steht eindeutig fest, welches die nachfolgende Operation ist oder ob die gesamte Berechnung hiermit beendet ist.

Diese Forderung lässt sich aber bei vielen Anwendungen nicht einhalten, etwa wenn Programme miteinander kommunizieren und die Reihenfolge der Nachrichtenübermittlungen von der Umwelt oder von Zufällen abhängt.

Beispiel 7.1.3: NIM-Spiel.
Gegeben sei ein Haufen von n Hölzchen. Zwei Spieler A und B ziehen abwechselnd (A beginnt). In jedem Zug darf man 1, 2 oder 3 Hölzchen wegnehmen. Wer das letzte Hölzchen wegnimmt, hat verloren.

In jedem Zug werden nichtdeterministisch 1, 2 oder 3 Hölzchen weggenommen. Ein nichtdeterministisches Programm (mit dem Sprachelement or) für n=17 Hölzchen würde dann lauten:

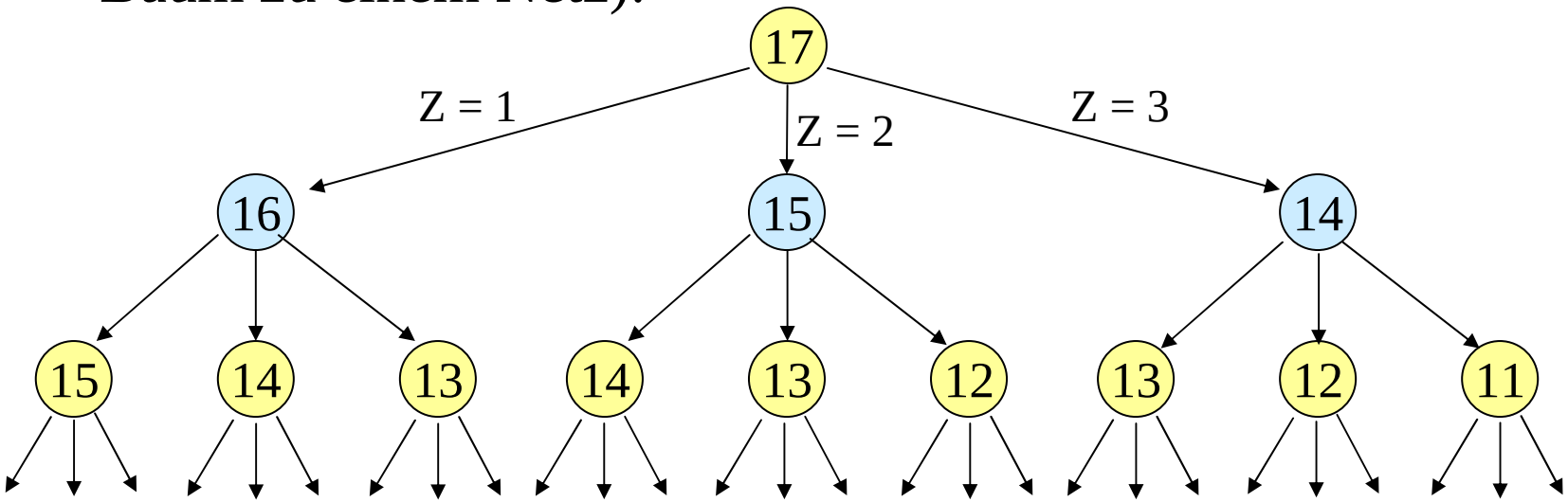
```
program NIM is
declare integer Anzahl, Z;  Boolean A_gewinnt;
begin  Anzahl := 17; A_gewinnt := true;
   while Anzahl > 0 do
      ( Z := 1  or  Z:= 2  or  Z:= 3 );
      Anzahl := Anzahl - Z;  A_gewinnt := not A_gewinnt;
      if A_gewinnt then write ("B nimmt", Z," Hölzchen")
                   else write ("A nimmt", Z," Hölzchen") fi
   od;
   if A_gewinnt then write ("A hat gewonnen")
                else write ("B hat gewonnen") fi
end
```

Aufgabe für Sie:

Vollziehen Sie dieses Beispiel nach und machen Sie sich klar, welche verschiedenen Abläufe möglich sind.

Obiges Programm enthält eine Unkorrektheit, die aber keinen Einfluss auf das Ergebnis hat. Welche ist dies?

Suchen Sie nach einer Darstellung für solche nichtdeterministischen Abläufe, z.B. baumartig (gelb entspricht "A ist am Zug", blau entspricht "B ist am Zug", vereinfachen Sie den Baum zu einem Netz):



Beachten Sie:
Wir haben nur die Abläufe des NIM-Spiels beschrieben. Dies gibt noch keinen Hinweis darauf, wie man das Spiel spielen muss, um zu gewinnen.

Hierzu muss man die Darstellung analysieren und nach Situationen suchen, die den Gewinn garantieren.

Wenn z.B. noch 8 Hölzchen vorhanden sind und B am Zug ist, so kann B den Sieg erzwingen, indem B 3 Hölzchen wegnimmt (Situation "5 Hölzchen") und nach dem Zug von A so viele Hölzchen wegnimmt, dass noch genau ein Hölzchen übrig bleibt.
Dies wäre ein "Zusatzalgorithmus", der zu jeder Situation den zurzeit verheißungsvollsten Zug ermittelt.

■

7.1.4 Terminierung

Ein Algorithmus *terminiert für eine Eingabe u*, wenn der Algorithmus bei Eingabe von u nach endlich vielen Schritten anhält. Ein Algorithmus "terminiert stets", wenn er für alle Eingaben terminiert.

Ein Programm realisiert (oder berechnet) eine Abbildung, nämlich die Zuordnung der Eingabewerte zu den zugehörigen Ausgabewerten. Von terminierenden Algorithmen werden also genau die totalen Funktionen realisiert.

In der Praxis wird die Terminierung oft gefordert, insbesondere von Benutzern, die auf jede Eingabe eine Antwort erwarten, und dies möglichst schon nach kurzer Zeit. Die Bedingung der Terminierung ist für Algorithmen aber *nicht notwendig* und ist auch nicht für alle Algorithmen erwünscht. Z.B. sollte ein Betriebssystem prinzipiell unendlich lange arbeiten.

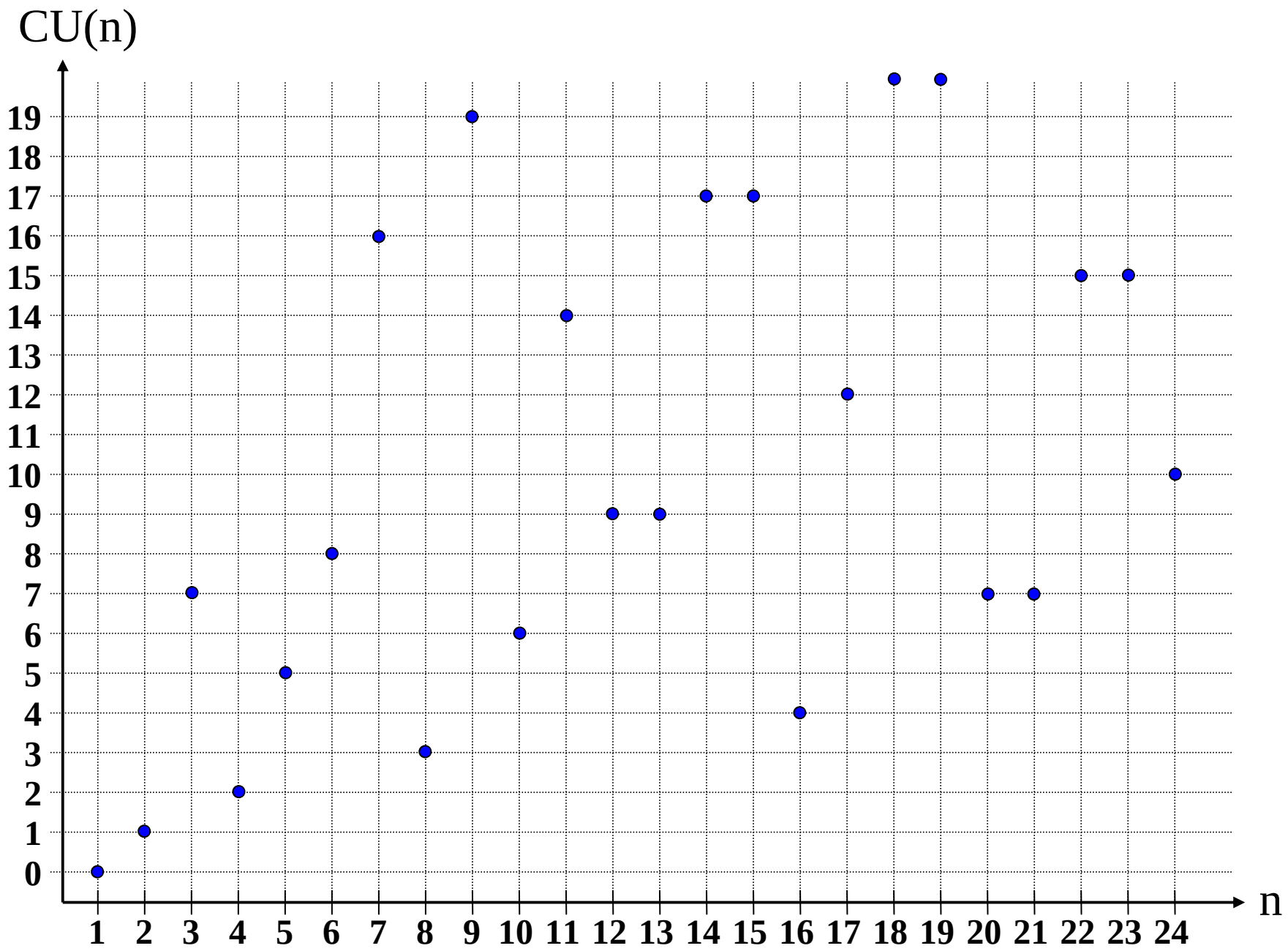
Einschub zweier Beispiele CU und MC91:

Die Terminierung eines Programms ist in der Praxis oft extrem schwierig nachzuweisen.

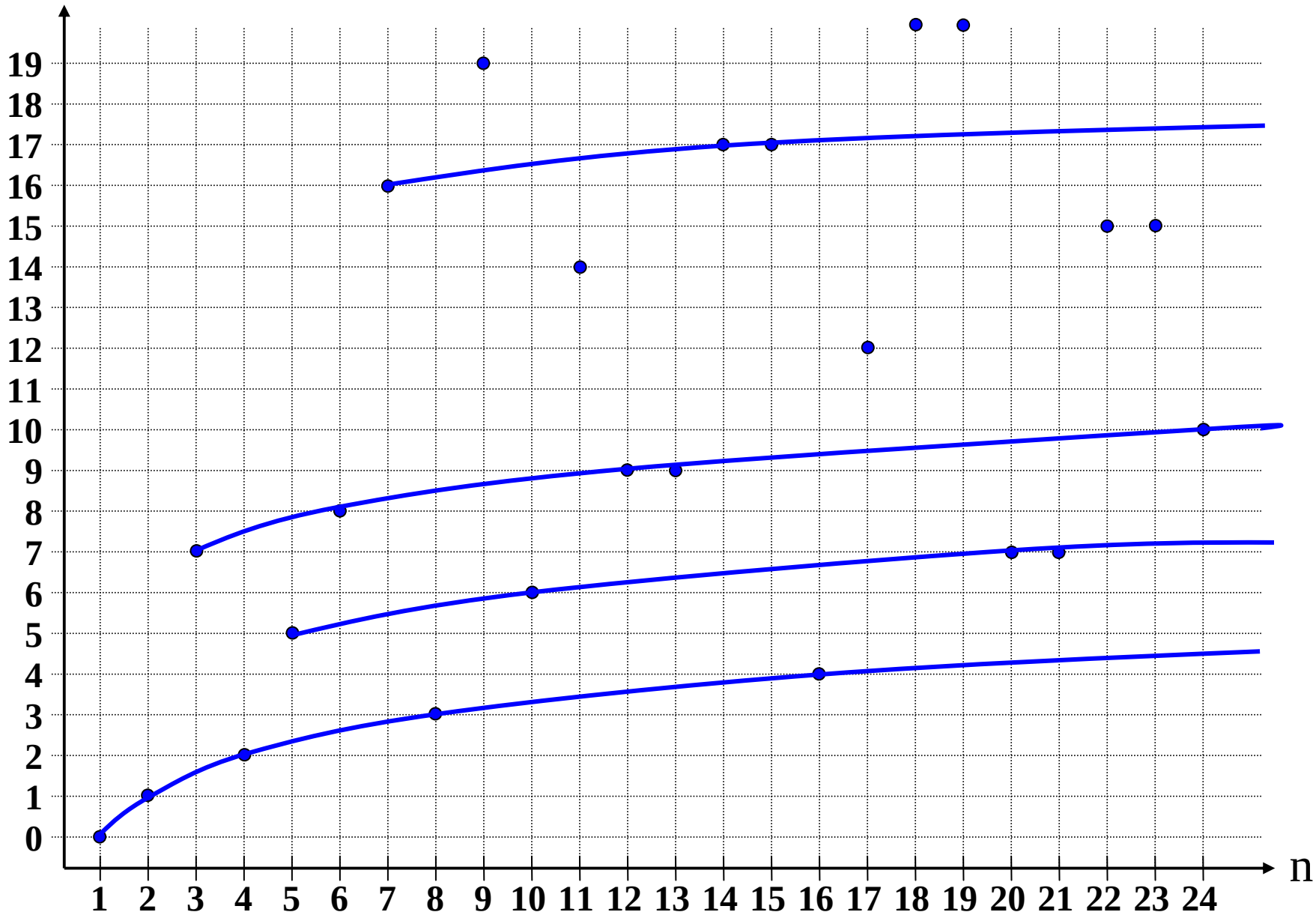
7.1.5 Standardbeispiel: Collatz- oder Ulam-Funktion:

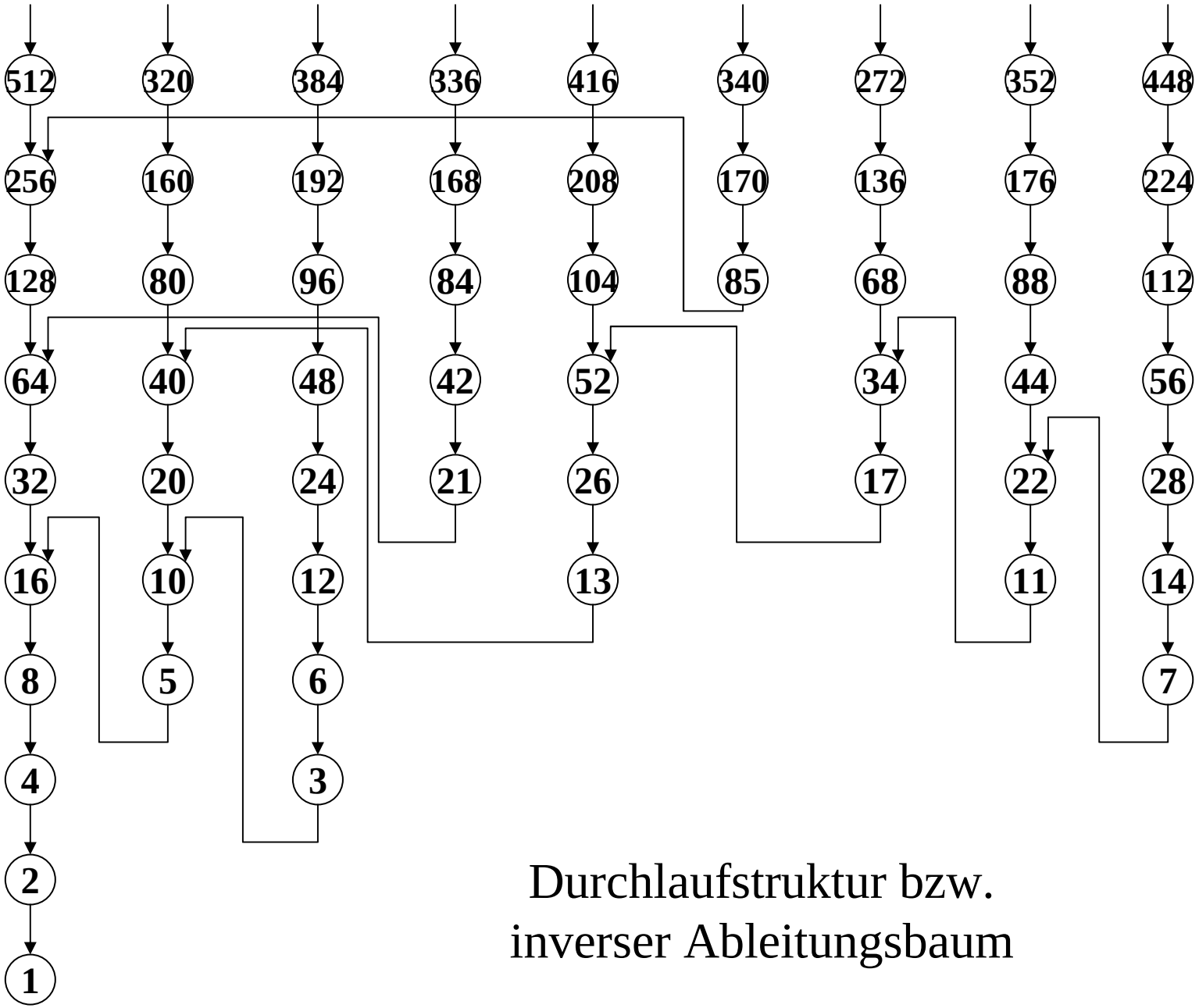
```
function CU(positive X) return natural is
begin
    if X=1 then return 0;
    else if X mod 2 = 0 then return CU(X/2) + 1
        else return CU(3*X+1) + 1 fi
    fi
end CU;
```

Veranschaulichung auf den nächsten Folien. Auffällig sind die häufigen Pärchen im Funktionsgraphen.


CU(n)
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
n

CU(n) Überlagerung logarithmischer Kurven, jeweils beginnend bei einer ungeraden Zahl





Durchlaufstruktur bzw.
inverser Ableitungsbaum

Es ist bis heute unbekannt, ob diese Funktion für alle Eingaben terminiert. (Man weiß dies zwar für unendlich viele Werte, nur "einige Zahlen" entziehen sich bisher hartnäckig einem Beweis.)

Für alle Zahlen, für die man mit heutigen Rechnern diese Funktion testen kann (insbesondere für alle Zahlen bis 10^{15}), liefert die Funktion CU ein Ergebnis, sodass man vermutet, die von CU berechnete Funktion sei total.

Diese Funktion wird meist mit Stanislaw Ulam, 1909-1984, Prof. an der Univ. Florida, in Verbindung gebracht, der sich erst in den 1950er Jahren mit ihr beschäftigte. Aber bereits in den 1930er Jahren untersuchte der Hamburger Mathematik-professor Lothar Collatz diese Funktion, weshalb sie oft als Collatz-Funktion in der Literatur erscheint; manchmal auch als Hasse's Problem oder Thwaite's Problem oder "the syracuse algorithm" (nach der Syracuse University, wo sie genauer erforscht wurde).

7.1.6: Die Terminierung ist bei rekursiven Definitionen nicht leicht nachzuweisen. Relativ einfach geht dies noch bei folgender doppelt-rekursiv dargestellten, genannt McCarthys 91-Function:

```
function MC91 (natural X) return natural is
begin if X > 100 then return X-10
      else return MC91(MC91(X+11)) fi
end
```

Übung: Beweisen Sie, dass die hiervon realisierte Funktion lautet:
f(n) = if n > 100 then n-10 else 91 fi .

7.1.7 Eine Besonderheit von Algorithmen:

Algorithmen können Algorithmen als Daten einlesen. Weil:

Ein Algorithmus lässt sich als Programm formulieren und ist dann eine Folge von Zeichen über dem Terminalalphabet der Programmiersprache, d.h., jeder Algorithmus lässt sich als ein Wort $w \in \Sigma^*$ auffassen. Es gibt Algorithmen, die solche Zeichenfolgen einlesen und verarbeiten können, folglich kann man Algorithmen als Eingabe für Algorithmen verwenden. Dies ist nicht überraschend; denn ein Compiler ist ein Algorithmus, der Algorithmen in Form von Programmen einliest und sie in Programme der Maschinensprache übersetzt.

(Wie üblich verwenden wir im Folgenden das Zeichen $\forall$ für "für alle" und das Zeichen $\exists$ für "es existiert".)

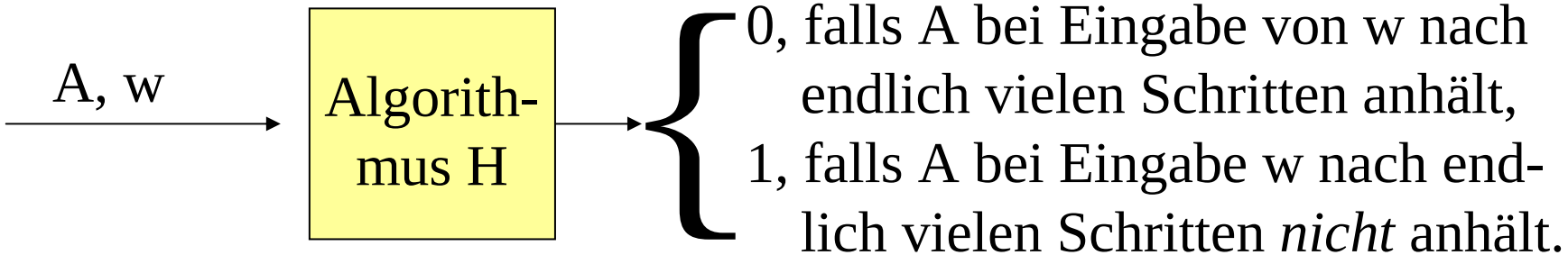
7.2 Grenzen der Algorithmen, Unentscheidbarkeit

Was kann man mit Algorithmen ***nicht*** beschreiben / lösen?

Betrachte folgende Aufgabe:
Konstruiere einen Algorithmus H, der beliebige Algorithmen und zugehörige Eingabedaten einlesen kann und der zu jedem beliebigen Algorithmus A und zu jeder Folge von Daten w in endlich vielen Schritten feststellt, ob der Algorithmus A für die Eingabedaten w nach endlich vielen Schritten anhält oder nicht.

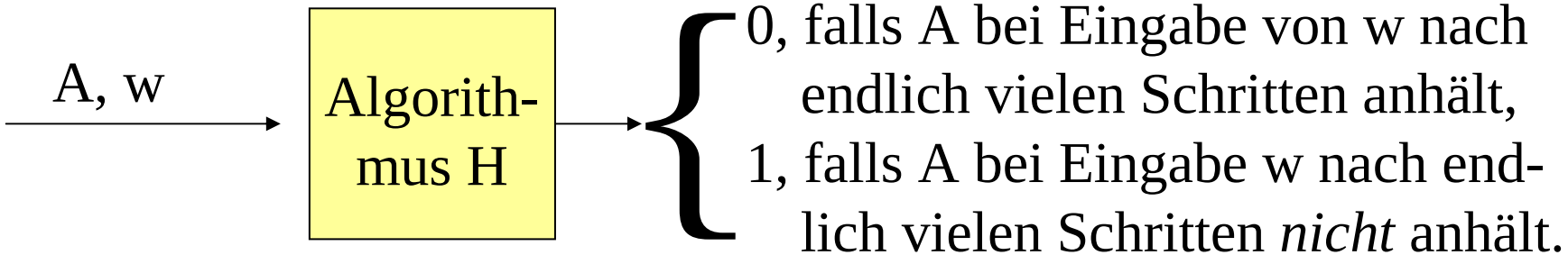
Bezeichnung 7.2.1: Die Aufgabe, einen solchen stets terminierenden Algorithmus H zu finden, bezeichnet man als das Halteproblem für Algorithmen.

Gesucht wird also ein Algorithmus H

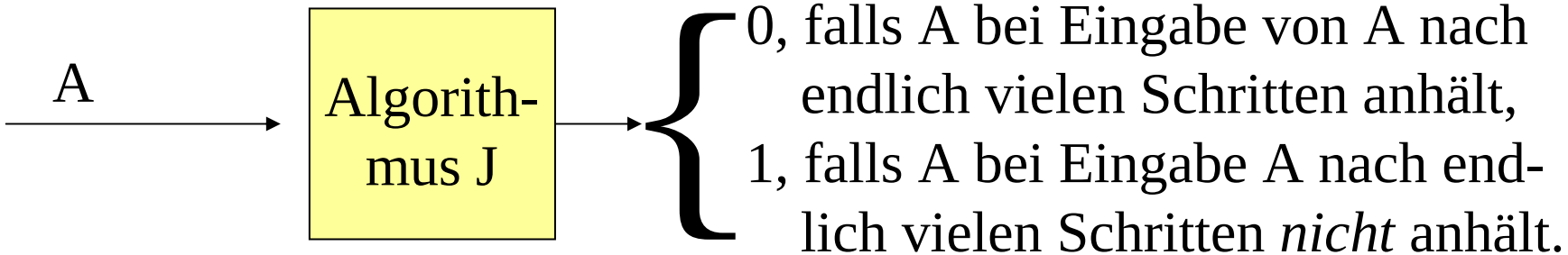


für alle Algorithmen A und für alle Eingabefolgen w
(formelartig geschrieben als: $\forall A \; \forall w$).

Angenommen, es gibt solch einen Algorithmus H mit $\forall A\ \forall w$

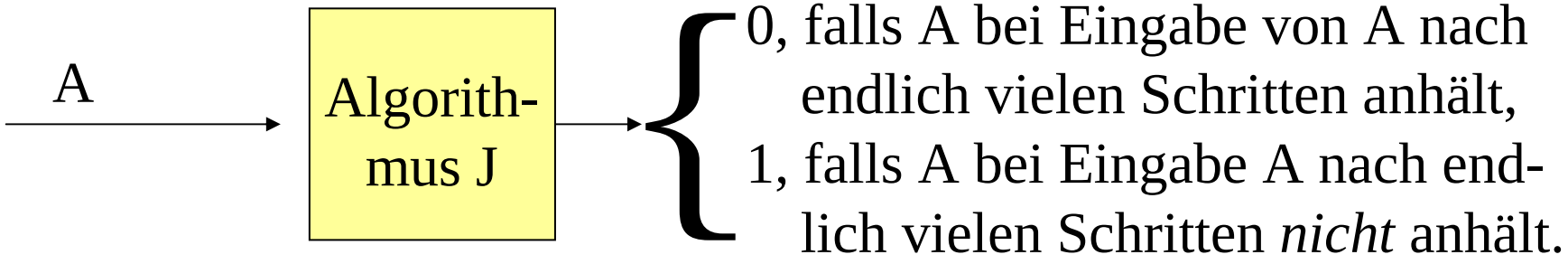


Speziell gibt es dann auch einen Algorithmus J mit $\forall A$

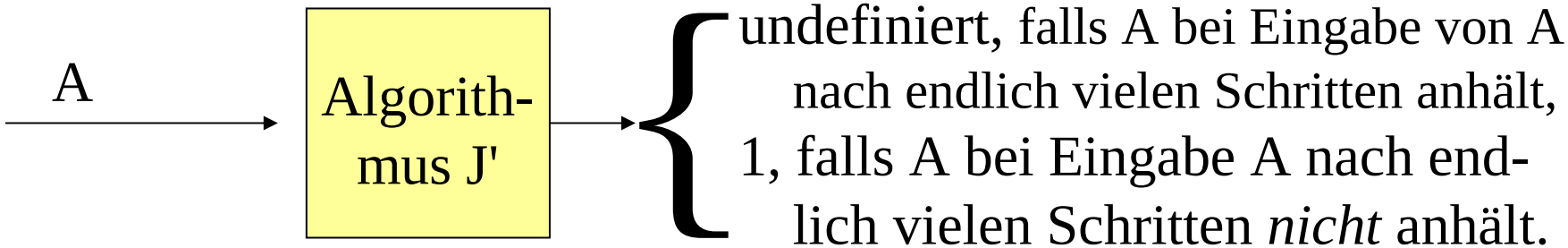


für alle Algorithmen A, indem man nur den Algorithmus A eingibt und dann H mit der Eingabe A und A (=w) ablaufen lässt.

Zu diesem Algorithmus J mit $\forall A$

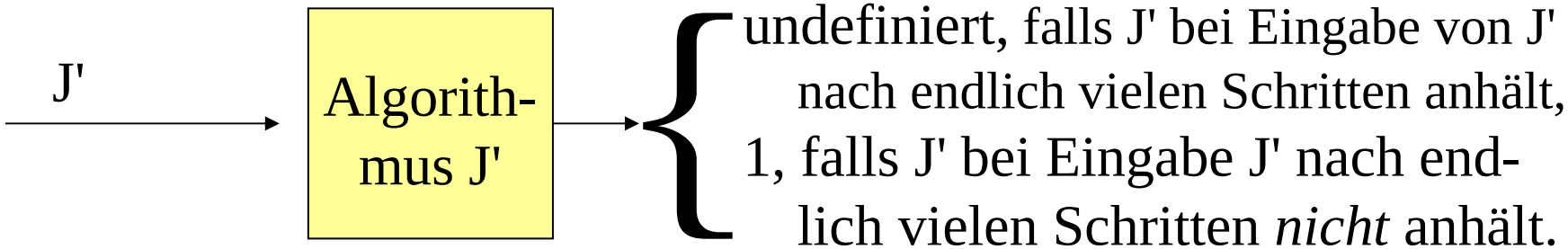


gibt es dann auch einen Algorithmus J' mit $\forall A$



Klar wegen: Modifiziere J so, dass der Algorithmus J anstelle der Ausgabe 0 in eine unendliche Schleife geht; so erhält man J'.

Was macht J' bei Eingabe von J'?



Fall 1: Bei Eingabe von J' hält J' an.
Dann liefert J' bei Eingabe von J' den Wert 1.
Nach Definition von J' hält dann J' bei Eingabe J' *nicht* an.
Widerspruch!

Fall 2: Bei Eingabe von J' hält J' *nicht* an.
Dann läuft J' bei Eingabe von J' in eine unendliche Schleife.
Nach Definition von J' hält dann J' bei Eingabe J' an und liefert 1.
Widerspruch!

Beide möglichen Fälle führen also auf einen Widerspruch.
Mehr Möglichkeiten gibt es nicht.
Folglich muss die Annahme, dass es den Algorithmus H gibt, falsch gewesen sein. Es gilt daher der

Satz 7.2.2 (Unlösbarkeit des Halteproblems)
Es gibt keinen Algorithmus H, der zu jedem beliebigen Algorithmus A und zu jeder Folge von Daten w in endlich vielen Schritten feststellt, ob der Algorithmus A für die Eingabedaten w nach endlich vielen Schritten anhält oder nicht.

Dieses Resultat formuliert man kurz in der Form:

Das Halteproblem ist algorithmisch nicht lösbar.

Anmerkung 7.2.3: Eine Menge $L \subseteq \Sigma^*$ heißt entscheidbar, wenn ihre charakteristische Funktion (Folie 345) berechnet werden kann, d.h., wenn es einen stets terminierenden Algorithmus A mit der Eingabemenge Σ^* und der realisierten Funktion $f_A : \Sigma^* \rightarrow \{0,1\}$ gibt mit $f_A = \chi_L$, d.h., für alle $w \in \Sigma^*$ muss gelten:

$$f_A(w) = \begin{cases} 1, & \text{falls } w \in L \\ 0, & \text{falls } w \notin L \end{cases}$$

Anderenfalls heißt die Menge L unentscheidbar.

Der obige Satz besagt also:
Das Halteproblem für Algorithmen ist unentscheidbar.

Bezeichnung 7.2.4: Die Aufgabe, einen stets terminierenden Algorithmus H_ε zu finden, der zu jedem beliebigen Algorithmus A nach endlich vielen Schritten feststellt, ob der Algorithmus A für die leere Eingabe ε nach endlich vielen Schritten anhält oder nicht, bezeichnet man als das spezielle oder als das ε-Halteproblem für Algorithmen.

Satz 7.2.5: Das ε-Halteproblem ist unentscheidbar.

Beweis durch Rückführung auf das Halteproblem. Wir konstruieren zu jedem Algorithmus A und zu jeder Eingabe w einen Algorithmus A_w, so dass gilt: A hält bei Eingabe von w genau dann an, wenn A_w mit der leeren Eingabe anhält. Wäre das ε-Halteproblem also entscheidbar, dann wäre auch das Halteproblem entscheidbar, im Widerspruch zu Satz 7.2.2. Folglich muss das ε-Halteproblem unentscheidbar sein.

Konstruktion von A_w aus A und w:
Es seien A ein Algorithmus und $w \in \Sigma^*$ eine Eingabefolge. Dann wandele A in folgenden Algorithmus A_w um: A_w ist wie A aufgebaut, besitzt jedoch eine weitere Zeichenketten-Variable X (implementiert z.B. als array [1..n] of character) und an den Anfang des Anweisungsteils wird die Wertzuweisung X := "w"; gesetzt. Ein read-Befehl wird jetzt dadurch ersetzt, dass immer das nächste Zeichen von X (anstelle: "von der Eingabe") der jeweiligen Variablen zugewiesen wird. Der so erhaltene Algorithmus A_w liest also nichts ein und arbeitet genau so, wie A bei der Eingabe w arbeitet. Folglich gilt:

A hält bei Eingabe von w $\Leftrightarrow$ A_w hält bei Eingabe von ε.

Damit ist Satz 7.2.5 bewiesen. ■

7.2.6 Anmerkungen zum Halteproblem

Aus dem Unentscheidbarkeitssatz 7.2.2 lassen sich viele weitere Probleme ableiten, die alle algorithmisch nicht lösbar sind, z.B.

- entscheide, ob Programme für *alle* Eingaben anhalten,
- entscheide, ob Programme für *alle* Eingaben nicht anhalten,
- entscheide, ob zwei beliebige Programme dasselbe berechnen, also die gleiche realisierte Abbildung besitzen,
- entscheide, ob eine kontextfreie Grammatik alle Wörter erzeugt

usw.

Weitere Aussagen hierzu finden Sie in Theorie-Vorlesungen und in Büchern über Grundlagen der Informatik oder der Philosophie.

7.3 Andere Überlegungen zur Unentscheidbarkeit

Man kann das Halteproblem auch anschaulich erläutern. Hierzu ist es nützlich, eine Aufzählung der Wörter über einem Alphabet festzulegen. Diese basiert auf der längenlexikografischen Ordnung. Dies ist die Anordnung, wie wir sie in einem Lexikon finden, jedoch werden die Wörter zunächst nach ihrer Länge und dann erst nach der Lexikon-Anordnung aufgelistet.

7.3.1 Hinweis: Eine Ordnung "$\leq$" ist eine zweistellige Relation auf einer Menge Σ: $\leq \subseteq \Sigma \times \Sigma$, die reflexiv, antisymmetrisch und transitiv ist, d.h.: (statt $(a,b) \in \leq$ schreibt man $a \leq b$)
$\forall a \in \Sigma$ gilt: $a \leq a$ (Reflexivität),
$\forall a, b \in \Sigma$ gilt: aus $a \leq b$ und $b \leq a$ folgt $a = b$ (Antisymmetrie),
$\forall a, b, c \in \Sigma$ gilt: aus $a \leq b$ und $b \leq c$ folgt $a \leq c$ (Transitivität).

Meist betrachtet man die Relation "<", die sich von "≤" nur dadurch unterscheidet, dass sie nirgends reflexiv ist, d.h.
Für kein $a \in \Sigma$ gilt: $a < a$ (Irreflexivität)
und für $a \neq b$ gilt $a < b$ genau dann, wenn $a \leq b$ ist.

Eine Ordnung heißt total (oder linear), wenn je zwei Elemente in der Ordnungsbeziehung stehen, d.h.,
$\forall a, b \in \Sigma$ gilt $a \leq b$ oder $b \leq a$ (Totalität oder Linearität)

Ordnungen begegnen uns ständig. Sie sind meist die Basis für das Ablegen und das schnelle Auffinden von Daten. Im Rechner sind wegen der unten in 7.3.2 genauer beschriebenen Ordnung auf $\{0,1\}^*$ faktisch alle Daten angeordnet und lassen sich daher sortieren und (z.B. mit Intervallschachtelung) rasch wiederfinden.

7.3.2 Definition

Gegeben sei ein Alphabet $\Sigma = \{a_1, a_2, ..., a_m\}$ mit der Ordnung $a_1 < a_2 < ... < a_m$. Dann wird hierdurch folgende längenlexikografische Ordnung "$\leq$" auf Σ^* definiert:

$\forall u, v \in \Sigma^*$ gilt $u \leq v$ genau dann, wenn

entweder $|u| < |v|$ gilt

oder wenn $u = v$ gilt

oder wenn $|u| = |v|$ und $u \neq v$ ist und u und v die Form

$u = xa_iy$, $v = xa_jz$ mit $a_i < a_j$ (mit $x, y, z \in \Sigma^*$) haben.

Standardbeispiel: $\Sigma = \{0, 1\}$ mit $0 < 1$. Dann gilt auf $\{0, 1\}^*$:

$\varepsilon < 0 < 1 < 00 < 01 < 10 < 11 < 000 < 001 < 010 < ...$

In dieser Ordnung kann man alle Wörter auflisten ("aufzählen"):

$\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110,$
$111, 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, ...\}.$

Folgerung: Für jedes Alphabet $\Sigma = \{a_1, a_2, ..., a_m\}$ mit der Ordnung $a_1 < a_2 < ... < a_m$ kann man die Menge der Wörter Σ^* nacheinander hinschreiben. Dies lässt sich mit einem nicht endenden Algorithmus effektiv durchführen: Beginnend mit dem leeren Wort ε gehe man das als letztes hingeschriebene Wort u von hinten nach vorne durch; trifft man dabei auf ein a_i mit $i < m$, dann ersetze man dieses a_i durch a_{i+1} und ist fertig; falls man auf a_m trifft, ersetzt man dieses Zeichen durch a_1 und wiederholt das Verfahren mit dem Zeichen davor; falls man auf diese Weise alle Zeichen umgewandelt hat (d.h., das Wort war leer oder enthielt ausschließlich das Zeichen a_m), so setze man ein weiteres a_1 vor das Wort und ist fertig. Das so erhaltene Wort ist das nächste Wort in der längenlexikografischen Aufzählung und dient als neues Wort u für die Berechnung des folgenden Wortes. (Man sagt, die Menge der Wörter Σ^* ist aufzählbar, da diese Erzeugung eine "Aufzählung" bedeutet.)

7.3.3 Die widersprüchliche Funktion d: $\mathbb{N}_0 \rightarrow \mathbb{N}_0$

Nach dieser Vorbereitung kommen wir zur Definition einer Funktion d durch Aufzählung. Wir gehen vom Alphabet A einer Programmiersprache aus, dann können wir alle möglichen Zeichenfolgen (also Wörter aus A^*) nacheinander hinschreiben. In dieser Folge wählen wir genau diejenigen Zeichenfolgen aus, die

- ein Programm der Programmiersprache sind (das können wir mit Hilfe der Grammatik feststellen) und die zugleich
- genau eine natürliche Zahl einlesen und genau eine natürliche Zahl ausgeben (also nur eine read- und eine write-Anweisung für einen Wert "natural" enthalten, die ohne Beschränkung der Allgemeinheit als erste und als letzte Anweisung im Programm ausgeführt werden).

So erhalten wir konstruktiv (!) eine Folge von Programmen $p_0, p_1, p_2, ...$, die jeweils eine Funktion $f_0, f_1, f_2, ...$ mit $f_i: \mathbb{N}_0 \rightarrow \mathbb{N}_0$ realisieren.

Nun können wir d definieren:
Für alle natürlichen Zahlen $n \in \mathbb{N}_0$ setze: $d(n) = f_n(n) + 1$.

Offensichtlich kann man d(n) für jedes n berechnen:
Hierzu lese man die Zahl n ein, zähle die Programme mit obiger Eigenschaft auf, stoppe diese Aufzählung beim n-ten Programm, rechne dieses Programm mit der Zahl n durch, addiere zum Ergebnis 1 und gib diese Zahl aus.

Dies ist zugleich ein Algorithmus, um d zu berechnen. Folglich gibt es ein Programm p', das diesen Algorithmus nachvollzieht, das also die Funktion d berechnet. Dieses Programm p' besitzt die obige Eigenschaft und muss daher unter den Porgrammen p_0, p_1, p_2, ... vorkommen, sagen wir $p' = p_k$. Das Programm p_k berechnet sowohl die Funktion d als auch die Funktion f_k. Berechnen wir den Funktionswert für die Eingabe k, dann gilt: $f_k(k) = d(k) = f_k(k)+1$.

Dies ist ein Widerspruch! *Warum? Die Argumentation war doch logisch richtig, oder? Durchdenken Sie sie nochmals genau!*

7.3.4 Auflösung 1:

Die Argumentation ist in der Tat korrekt, sie verschweigt nur eine Tatsache: Algorithmen und Programme berechnen in der Regel keine totalen Funktionen, d.h., es gibt Eingabewerte, für die die Berechnung nicht endet (unendliche Schleifen z.B.).

Das bedeutet: Die Funktion d kann man - wie angegeben - definieren, man kann auch - wie angegeben - ein Programm für ihre Berechnung schreiben und dieses Programm muss in der Aufzählung der Programme p_0, p_1, p_2, ... als ein Programm p_k vorkommen, aber an der Stelle k ist d dann zwangsläufig nicht definiert.

Wir haben auf den beiden obigen Folien also nur die Aussage

d(k) ist undefiniert

bewiesen und nichts sonst.

Schön und gut, werden Sie sagen, aber man kann sich doch bei der Aufzählung der Programme auf die beschränken, die stets terminieren, man fügt also noch die Eigenschaft "und das Programm hält für alle Eingaben an" hinzu.

Dann erhält man eine Folge von stets terminierenden Programmen p_0, p_1, p_2, ..., die jeweils eine totale Funktion f_0, f_1, f_2, ... mit f_i: $\mathbb{N}_0 \rightarrow \mathbb{N}_0$ realisieren. Hierzu kann man ganz genau so die Funktion $d(n) = f_n(n) + 1$ definieren und man erhält erneut den Widerspruch $f_k(k) = d(k) = f_k(k)+1$.

Dies ist schon wieder ein Widerspruch! Warum? Die Argumentation ist nun doch logisch unanfechtbar, oder? Durchdenken Sie sie unter dem Aspekt, dass wir uns auf die stets terminierenden Programme beschränken, genau!

7.3.5 Auflösung 2:

Die Argumentation ist in der Tat korrekt, sie verschweigt nur wiederum eine Tatsache, aber nun eine andere: Wir haben stillschweigend angenommen, wir könnten zu jedem Programm entscheiden, ob es stets terminiert oder nicht. Denn nur wenn wir dies von jedem Programm eindeutig feststellen können, können wir die Aufzählung p_0, p_1, p_2, ... bilden.

Da alles andere tatsächlich richtig ist, muss hier also der Fehler liegen. Das heißt: Wir haben bewiesen, dass es keinen Algorithmus geben kann, der zu jedem Programm nach endlich vielen Schritten feststellt, ob dieses Programm stets terminiert oder nicht. (Dies ist genau die erste Aussage in 7.2.6!)

Schön und gut, werden Sie wieder sagen, dann beschränken wir uns bei der Aufzählung der Programme auf solche Programme, die, wenn als nächstes der Index j zu vergeben ist, für die Eingabe j nach endlich vielen Schritten anhalten (alle anderen Eingabewerte interessieren uns nicht). Ist dies der Fall, so wird das gerade untersuchte Programm als p_j in die Aufzählung aufgenommen, ansonsten wird es ignoriert.

Dann erhält man wieder eine Folge von Programmen p_0, p_1, p_2, ..., die jeweils eine Funktion f_0, f_1, f_2, ... realisieren mit $f_i: \mathbb{N}_0 \rightarrow \mathbb{N}_0$ und $f_i(i)$ ist definiert. Hierzu kann man erneut die Funktion $d(n) = f_n(n) + 1$ definieren und man erhält den gleichen Widerspruch $f_k(k) = d(k) = f_k(k)+1$.

Jetzt ist der Widerspruch aber hieb- und stichfest?! Die Argumentation ist wohl unanfechtbar, oder? Wieder sollten Sie sie genau durchdenken.

7.3.6 Auflösung 3:

Die Argumentation enthält den gleichen Fehler wie er in der Auflösung 2 genannt wurde: Wir haben hier stillschweigend angenommen, wir könnten zu jedem Programm und zu jeder einzelnen Eingabe w entscheiden, ob das Programm bei Eingabe von w anhält.

Da alles andere tatsächlich richtig ist, muss hier also der Denkfehler liegen. Das heißt: Wir haben bewiesen, dass es keinen Algorithmus geben kann, der zu jedem Programm und zu jeder einzelnen Eingabe w nach endlich vielen Schritten feststellt, ob das Programm für die Eingabe w terminiert oder nicht. Dies ist die Aussage des Halteproblems 7.2.2, die wir hier sogar in einer etwas verschärfteren Form beweisen haben, da wir uns ja auf die speziellen Eingabewerte j beschränkten.

7.3.7 Hinweis

Die von einem Programm berechnete Funktion bezeichnet man als die Semantik des Programms. Die obigen Aussagen besagen, dass es keine Algorithmen gibt, um von der Syntax (= dem syntaktisch korrekten Programm) auf die Semantik (= das Ergebnis von Berechnungen) zu schließen.

Dies gilt in einem sehr weiten Sinne. In der Theorie ist diese Erkenntnis als "Satz von Rice" seit 1953 bekannt.
Sie gilt analog auch für Grammatiken (man kann von einer allgemeinen Grammatik nicht auf die Menge der abgeleiteten Wörter schließen) und für alle anderen Mechanismen zur Beschreibung von Algorithmen. Dies leitet nun über zu Kapitel 8 über Semantik / Korrektheit - doch leider ist der Kurs des Frühjahrs 2004 genau hier zu **Ende**.