Programmiersprache

ALGOL 60

(Revised report on the algorithmic language ALGOL 60)

Überarbeiteter Skript der Vorlesungen

an der

Humboldt-Universität zu Berlin

Frühjahr- und Herbstsemester 1970 bis 1973

Eine Erinnerung an die

problemorientierte Programmiersprache
ALGOL 60

The Programming Language ALGOL- 60 Datum: 18.September 2015

Bearb.: Bernd Hartwich
Diplom Mathematik- und Physiklehrer

ALGOL 60, die Kurzform für „**Algo**rithmic **L**anguage und 60 steht für das Jahr des ersten Entwurfes, ist der Name einer Familie von Programmiersprachen, die ab Ende der fünfziger Jahre ca. 30 Jahre Verwendung fanden. Alle ALGOL-Varianten hatten einen erheblichen Einfluss auf die Entwicklung von weiteren Programmiersprachen. Die Sprache Pascal etwa ist eine Weiterentwicklung von ALGOL 60, letztlich auch ANSI-C.

Die Programmiersprache ALGOL 60 ist eine der ersten **prozeduralen Programmiersprachen**, die 1958-1963 unter der Führung der Association for Computer Machinery (ACM) und der Gesellschaft für Angewandte Mathematik und Mechanik (GAMM), später dann der International Federation for Information Processing (IFIP), entwickelt wurde. Die endgültige Fassung wurde dann von der International Federation for Information Processing (IFIP) veröffentlicht, als » Revised report on the algorithmic language Algol 60« bekannt. Beteiligt waren unter anderem John W. Backus, Friedrich L. Bauer, John McCarthy, Peter Naur, Alan J. Perlis, Heinz Rutishauser und Klaus Samelson.

ALGOL 60 war als international entwickelte, von kommerziellen Interessen unabhängige, portable, prozedurale Programmiersprache in erster Linie für wissenschaftliche Zwecke gedacht. Es gab aber auch wichtige nichtnumerische Anwendungen. Ihre Einfachheit, und die für damalige Verhältnisse weitgehende Freiheit von Restriktionen, machte sie einem großen Anwenderkreis nutzbar.

ALGOL 60 war ein Meilenstein in der Entwicklung weiterer Programmiersprachen:

- Maschinenunabhängigkeit und akademische Fundierung haben ALGOL zu einer internationalen Norm gemacht
- mit ALGOL begann die exakte Definition von Spracheigenschaften
- mit ALGOL wurde die formale Definition der Syntax mit Hilfe der Backus-Naur-Form eingeführt.
- ALGOL zeigte den Wert eines klaren und einfachen Sprachkerns auf;
- ALGOL lieferte wesentliche Beiträge zur strukturierten Programmierung, einer der wichtigsten Beiträge war die Einführung von Blockstrukturen
  die meisten nach ALGOL entwickelten Programmiersprachen wurden in ihrem Sprachaufbau von ALGOL stark beeinflusst
- ALGOL schrieb als erste Programmsprache Laufzeitprüfungen vor, auftretende Laufzeitfehler während der Laufzeit eines Computerprogramms führen zum Absturz des ausgeführten Programms, zu falschen Ergebnissen oder zu nicht vorhersehbarem Verhalten des Programms. Nicht als Laufzeitfehler gelten Programmfehler, die bereits während der Entwicklung des Programmcodes vom Compiler festgestellt werden. Sind dies Syntaxfehler, so wird in der Regel kein ausführbarer Code erzeugt, das Programm kann deshalb nicht ausgeführt werden
- ALGOL sah die Möglichkeit vor, rekursive Verfahren zu programmieren
- ALGOL erlaubte allgemeine Ausdrücke innerhalb von Bereichsgrenzen.

Als großes Problem erwies sich, dass die Ein-/Ausgabe im ALGOL 60 Bericht nicht geregelt worden war, so dass deren Implementierungen stark zwischen den Compilern und Betriebssystemen variierten.

Die Methode der sequentiellen Formelübersetzung ermöglichte es, auf viele Restriktionen zu verzichten. Die darin enthaltene Idee des Stapelspeichers ermöglichte dann auch die Nutzung rekursiver Prozeduren. Es bestand die Möglichkeit innerer Prozedurdefinitionen, die Blockstruktur; die begin-Blöcke wurden vor allem dazu benutzt, Felder mit variablen Grenzen im Stapelspeicher anzulegen.

Inhaltsverzeichnis Seite

1. Einleitung

1.1. Zusammenfassung über den Bericht über die Algorithmische Sprache ALGOL 60 5
1.2. Die Struktur der Sprache 6
1.3. Formalismus für syntaktische Beschreibung 7

2. Grundsymbole, Bezeichner, Zahlen und Ketten

2.1. Grundsymbole 9
2.1.1. Buchstaben 9
2.1.2. Ziffern 9
2.1.3. Logische Werte 10
2.1.4. Begrenzer 10
2.1.4.1. Operatoren 11
2.1.4.1.1. Arithmetische Operatoren 11
2.1.4.1.2. Vergleichsoperatoren 11
2.1.4.1.3. Logische Operatoren 12
2.1.4.1.4. Folgeoperatoren 12
2.1.4.2. Trennzeichen 13
2.1.4.3. Klammern 13
2.1.4.4. Vereinbarungszeichen 14
2.1.4.5. Spezifikationen 15
2.2. Bezeichner 15
2.3. Zahlen 15
2.4. Ketten 16

3. Variable und Ausdrücke

3.1. Variable 18
3.1.1. Einfache Variable 18
3.1.2. Indizierte Variable 19
3.1.3. Typvereinbarungen für einfache Variable 20
3.1.4. Feldvereinbarungen 21
3.2. Standardfunktionen 22
3.3. Arithmetische Ausdrücke 23
3.3.1. Einfache arithmetische Ausdrücke 23
3.3.2. Bedingte arithmetische Ausdrücke 24
3.3.3. Regeln zur Auswertung arithmetischer Ausdrücke 24
3.4. Logische Ausdrücke 26
3.4.1. Einfache logische Ausdrücke 26
3.4.2. Bedingte logische Ausdrücke 28
3.4.3. Regeln bei der Abarbeitung arithmetischer und logischer Ausdrücke 29
3.5. Zielausdrücke 30

4. Anweisungen

4.1.	Anweisungsformen	31
4.2.	Ergibtanweisungen	31
4.3.	Marken, markierte Anweisungen, Sprunganweisungen	33
4.3.1.	Marken, markierte Anweisungen	33
4.3.2.	Leere Anweisungen	34
4.3.3.	Sprunganweisungen	34
4.4.	Bedingte Anweisungen	35
4.5.	Laufanweisungen	37
4.5.1.	Allgemeine Betrachtungen	37
4.5.2.	Formen der Lauflisten	38
4.5.3.	Abschließende Beispiele	39
4.6.	Verbundanweisungen	42
4.6.1.	Allgemeine Betrachtungen	42
4.6.2.	Abschließende Beispiele	42
4.7.	Blöcke	45
4.7.1.	Allgemeine Betrachtungen	45
4.7.2.	Gültigkeit von Bezeichner	46

5. Unterprogrammtechnik

5.1.	Prozedurvereinbarung und Prozeduranweisung	49
5.2.	Dynamische Prozeduren	50
5.3.	Funktionsprozeduren	51
5.4.	Parameterfreie Prozeduren	52
5.5.	Rekursive Prozeduren	53
5.5.1.	Begriffserklärung	53
5.5.2.	Beispiel – Die Berechnung der Fakultät eine Zahl	54
5.5.4.	Kellerprinzip bei rekursiven Prozeduren	54

6. ALGOL-Programm

6.1.	Allgemeine Betrachtungen	56
6.2.	Abschließende Beispiele	56

7. ALGOL-Beschreibung mit der erweiterten Backus-Naur-Form 60

8. Quellenangaben 65

1. Einleitung

1.1 Zusammenfassung über den Bericht über die Algorithmische Sprache ALGOL 60

(ALGOL Bulletin Supplement no 3 - Übersetzung des Report on the Algorithmic Language ALGOL 60 - ALGOL Bulletin Supplement no 2)

Der Bericht enthält eine vollständige, definierte Beschreibung der internationalen algorithmischen Sprache ALGOL 60. Es handelt sich dabei um eine Sprache, in der man eine große Klasse von numerischen Prozessen in einer Form ausdrücken kann, die für die unmittelbare, automatische Übersetzung in die Sprache programmgesteuerter automatischer Rechenanlagen hinreichend und notwendig ist. Darüber hinaus werden die Begriffe

- **Bezugssprache** - die definierte Sprache von ALGOL entsprechend den Festlegungen des Komitees, die Richtschnur für alle Maschinensprachen und für Compilerschöpfungen
- **Veröffentlichungssprache** - erlaubt Abweichungen von der Bezugssprache, wie sie beim Drucken oder beim Schreiben mathematischer oder kommerzieller Texte in erläuterten Zwischentexten in ALGOL, den comments, üblich sind – z.B. Summenzeichen, griechische Buchstaben, andere ländertypische Zeichen, usw.
- **Maschinensprache** - Anpassung an die Eigenschaften und Möglichkeiten der jeweiligen Rechenanlagen erläutert.

Im ersten Kapitel wird eine Übersicht über die grundlegenden Bestandteile und Eigenschaften der Sprache gegeben. Die formale Schreibweise, die durch die syntaktische Struktur definiert wird, wird mit Hilfe der **Backus-Naur-Form** dargelegt. Durch die Backus-Naur-Form im Algol-60-Report wurde es erstmals möglich, die Syntax einer Programmiersprache formal exakt, also ohne die Ungenauigkeiten natürlicher Sprachen, darzustellen.

Im zweiten Kapitel werden alle Grundsymbole aufgeführt und die syntaktischen Einheiten und Begriffe

- Bezeichnungen
- Zahlen
- Ketten
- Größe
- Wert

definiert.

Im dritten Kapitel werden die Regeln, nach denen die Ausdrücke gebildet werden, sowie deren Bedeutung erklärt. Man unterscheidet

- arithmetische Ausdrücke
- logische Ausdrücke
- Zielausdrücke.

Im vierten Kapitel werden die sog. Anweisungen, die operativen Einheiten beschrieben. Zwei Gruppen von Anweisungen bilden die operativen Einheiten - die Grundanweisungen und komplexe Anweisungen.

Grundanweisungen:

- Ergibtanweisungen zur Berechnung von Formeln
- Sprunganweisungen für Unterbrechung der Ausführungsfolge der Anweisungen
- Leeranweisungen zur Gestaltung logischer Abläufe eines Prozesses
- Prozeduranweisungen zum Aufruf geschlossener, in der Regel sich häufig wiederholender Prozesse

Komplexe Anweisungen:

- Bedingte Anweisungen, die Abarbeitung hängt von Bedingungen ab
- Laufanweisungen sind Iterationen, die über Bedingungen beendet werden
- Verbundanweisungen werden über bestimmte syntaktische Einheiten zu einer Anweisung verbunden
- Blöcke werden über bestimmte syntaktische Einheiten zu einem Prozess verbunden

Im fünften Kapitel werden die als Vereinbarungen bekannten Einheiten definiert, die Eigenschaften von Objekte definieren, die während eines bestimmten Prozesses gültig sind.

Hinweis:
Ich hatte damals meine Vorlesung auf Basis dieses Berichtes aufgebaut und mich in der Regel an die aufgezeichnete Gliederung gehalten. Allerdings habe ich aus methodischen und didaktischen Gründen den Aufbau der Vorlesung sowie die Beschreibung der Syntax etwas vereinfacht dargestellt und viele Beispiele eingefügt; den studentischen Belangen angepasst. Ebenso reduziert sich in diesem Skript die Ein- und Ausgabe auf die beiden symbolischen Funktionen **read**( ) und **print**( ). Sie dienen nur als sinnvolle Ergänzungen in den angeführten Beispielen.

1.2. Die Struktur der Sprache

Wie in der Zusammenfassung über den Bericht über die Algorithmische Sprache ALGOL 60 erwähnt, hat die algorithmische Sprache ALGOL drei verschiedene Arten von Darstellungen – **Bezugs-, Maschinen- und Veröffentlichungssprache**. Der Aufbau der Sprache, die im Folgenden beschrieben wird, wird mit Elementen der Bezugssprache durchgeführt. Das bedeutet, dass alle in der Sprache definierten Objekte aus einer gegebenen Menge von Symbolen dargestellt werden und es liegt nur in der Wahl der Symbole, durch die sich die beiden anderen Darstellungsformen unterscheiden. Struktur und der Inhalt müssen für alle Darstellungen die gleichen sein.
Zweck der algorithmischen Sprache ist es, Rechenprozesse beschreiben. Das Grundkonzept für die Beschreibung der Berechnungsvorschriften ist der bekannte **arithmetische Ausdruck**, der als Bestandteile Zahlen, Variablen und Funktionen verwendet. Aus diesen arithmetischen Ausdrücken entstehen durch Anwendung von Regeln der Arithmetik mit Hilfe eines gerichteten Gleichheitszeichens – dem **Ergibtzeichen** – und der Angabe einer Zielvariablen auf der linken Seite des Ergibtzeichens Formeln, die in sich abgeschlossenen Einheiten der Sprache darstellen. Sie werden als **Ergibtanweisungen** bezeichnet. Ergibtanweisungen entsprechen den Anweisungskästchen in der Flussdiagrammtechnik. In der Regel werden Ergibtanweisungen in der Reihenfolge ihres Auftretens abgearbeitet. Um den sequentiellen Ablauf des Programmes zu beeinflussen, werden gewisse nichtarithmetische Anweisungen und Anweisungsklauseln hinzugefügt. Diese können Entscheidungen oder iterative Wiederholungen hervorrufen. Sie sind die eigentlichen dynamischen Anweisungen, um logische Abläufe entsprechend der zu verarbeitenden Daten zu generieren.

Eine Unterbrechung der sequentiellen Abarbeitungsfolge auf Grund von Datenauswertungen durch eine **bedingte Anweisung** kann mittels einer **Sprunganweisung** zu einer mit einer **Marke** versehenen Anweisung oder Anweisungsfolge verzweigt werden. Eine solche Folge von Anweisungen kann zwischen Anweisungsklammeren **begin** und **end** gesetzt werden und bildet in dieser Form eine **Verbundanweisung**.

Die Wirkungsweise von Anweisungen wird von **Vereinbarungen** unterstützt, die selbst keine Recheninstruktionen sind; sie erzeugen keinen Objektcode. Sie informiert den Übersetzer über die Existenz und über bestimmte Eigenschaften der in den Anweisungen auftretenden Objekte. Diese **Typvereinbarungen** und **Feldvereinbarungen** geben z.B. den Typ der Zahlen an, die für die Variable als Wert möglich ist, sie geben die Dimension einer Matrix von Zahlen an, oder auch den Satz von Regeln, der eine Funktion definiert. Eine Folge von Vereinbarungen, gefolgt von einer Folge von Anweisungen und zwischen den Anweisungsklammern **begin** und **end** eingeschlossen bildet einen **Block**. Dadurch ist jede Vereinbarung einem Block zugeordnet und gilt nur für diesen Block. Ein **Programm** ist ein in sich abgeschlossener Block oder eine Verbundanweisung. Gültigkeitsregeln der vereinbarten Objekte innerhalb und außerhalb der Blöcke wird später erörtert.

1.3. Formalismus für syntaktische Beschreibung

Zur Beschreibung der Syntax einer Sprache begibt man sich außerhalb dieser zu beschreibenden Sprache. Eine sinnvolle Variante zur Beschreibung der Syntax von ALGOL ergibt sich mit der **Meta-Sprache von Backus-Naur**, die sog. **Backus-Naur-Form**. Die Backus-Naur-Form ist eine kompakte formale Metasprache zur Darstellung kontextfreier Grammatiken. Erstmalig wurde die Syntax der Sprache ALGOL 60 vollständig mit dieser Methode beschrieben. Durch diese Backus-Naur-Form im ALGOL-60-Report wurde es möglich, die Syntax einer Programmiersprache formal exakt, also ohne die Ungenauigkeiten natürlicher Sprachen, darzustellen.

Ursprünglich war sie nach John W. **Backus**, später wurde sie auch nach Peter **Naur** benannt. Beide waren Pioniere der Informatik, die sich mit der Erstellung der ALGOL 60 Regeln und insbesondere mit der Kunst des Compilerbaus beschäftigten.

Es gibt viele Varianten der Backus-Naur-Form. Die **erweiterte Backus-Naur-Form** – sie wurde ursprünglich von Niklaus Wirth zur Darstellung der Syntax der Programmiersprache Pascal eingeführt - ist eine nützliche Variante, die unter anderem eine kompakte Notation von sich wiederholenden Elementen erlaubt. In diesem Skript wurde die gesamte Syntax fast vollständig mit der erweiterte Backus-Naur-Form dargestellt, allerdings treten hinsichtlich der Klammern [ ] Redundanzen bei der Verwendung von indizierten Variablen auf. Es sei darauf hingewiesen, dass im Punkt 6. die gesamte Syntax von ALGOL 60 in der erweiterten Backus-Naur-Form aufgezeichnet ist.

Ein Programm besteht zunächst aus sichtbaren, auf der Tastatur vorhandenen Zeichen. Daneben treten noch Leerzeichen und Zeilentrenner auf. Die sichtbaren Zeichen werden zu den Terminalsymbolen, den terminals, gerechnet. Die Backus-Naur-Form verwendet Ableitungsregeln, in denen Nichtterminalsymbole, die nonterminals, definiert werden.

Die Backus-Naur-Form definiert Produktionsregeln, in denen Symbolfolgen jeweils einem Nichtterminalsymbol zugeordnet werden.

Fünf sog. metalinguistische Konnektoren dienen der Beschreibung der Syntax:

::=	Definitionszeichenzeichen
|	Oder-Zeichen
< >	Beschreibungskontext
{ }	Auswahlkontext
[ ]	eingefügter Kontext findet wahlweise Verwendung

Beispiel:

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

Lesart:
Eine Ziffer in ALGOL ist **eins** der Symbole von 0 bis 9. (siehe Hinweis 2.1.2.)

Hinweis:
Der Name in der Klammersetzung wird aus der ALGOL-Sprache herausgenommen. Dieser Beschreibungskontext wird grundsätzlich in kursiver und kleiner Schreibweise und im Singular benutzt.

Wiederholungen müssen über Rekursionen definiert werden.

Beispiel für die erweiterte Backus-Naur-Form

< *ganze zahl* > ::= *[{±}]* < *ziffer* > | < *ganze zahl* > < *ziffer* >

Lesart:
Man bedient sich der rekursiven Schreibweise. Eine ganze Zahl kann eine vorzeichenbehaftete Ziffer **oder** eine vorzeichenbehaftete Ziffer **und** eine weitere Ziffer, somit eine Folge von Ziffern mit wahlweisem Vorzeichen sein.

2. Grundsymbole, Bezeichner, Zahlen und Ketten

2.1 Grundsymbole

Die Bezugssprache ALGOL besteht aus 116 Grundsymbolen. Diese gliedern sich in

- 52 Buchstaben
- 10 Ziffern
- 2 logische Werte
- 52 Begrenzer

Syntax Grundsymbol:

< *grundsymbol* > ::= < *buchstabe* > | < *ziffer* > | < *logischer wert* > | < *begrenzer* >

2.1.1. Buchstaben

Buchstaben haben in ALGOL keine individuelle Bedeutung, sondern werden nur zur Bildung von Bezeichnungen und Zeichenketten benutzt. So hat z.B. der Bezeichner e keinen numerischen Bezug zur Eulerschen Zahl e, allerdings kann man aus programmtechnischen Gründen ihr den Wert 2,7182... zuweisen, um dann den Bezeichner e als Basis der natürlichen Logarithmen in Rechenoperationen innerhalb von arithmetischen Anweisungen einzubeziehen. (Hinweis Bezeichner beachten)

Syntax Buchstabe:

< *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

Nach dem ALGOL-Bericht kann dieses Alphabet beliebig eingeschränkt werden, oder durch Hinzunahme von Zeichen, die sich von allen anderen Grundsymbolen unterscheiden, erweitert werden. So waren z.B. bei dem bereits erwähnten ALGOL 60-R300 nur Kleinbuchstaben erlaubt. Dagegen sind beispielsweise im dänischen ALGOL-Alphabet oder im russischen ALGOL-Alphabet dänische bzw. kyrillische Buchstaben vorhanden. Deutsche Umlaute werden mit den definierten ALGOL-Buchstaben entsprechend ausformuliert.

Erweiterungen bzw. Einschränkungen wirken sich nicht nachteilig auf ALGOL als Sprache zur Formulierung von Prozessen aus, allerdings auf ALGOL als Programmiersprache, weil dadurch ein allumfassender Programmaustausch eingeschränkt wird.

2.1.2. Ziffern

Ziffern werden zur Bildung von Zahlen, Bezeichner und Zeichenketten benutzt.

Syntax Ziffer:

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

Erklärung :

Nach den fünf Peanoschen Axiomen, die für eine Menge von Elementen, den natürlichen Zahlen, gelten, gilt nach dem Axiom 1 und Axiom 2, dass es ein ausgezeichnetes Element 1 gibt

und dass es zu jeder natürlichen Zahl a aus den Menge der natürlichen Zahlen genau eine als Nachfolger bezeichnete natürliche Zahl a' gibt. Unter Benutzung der üblichen Bezeichnungsweise wird der Nachfolger von 1 mit 2, der Nachfolger von 2 mit 3 usw., insgesamt neun Symbole, bezeichnet. Das Symbol 0 dient der Darstellung der Zahl 0. Die Symbole 1,2,3,4,5,6,7,8,9,0 werden **Ziffern** genannt.

2.1.3. Logische Werte

Ein logischer Wert in ALGOL ist eines der beiden Wortsymbole **true** und **false**. Sie entsprechen den Wahrheitswerten der zweiwertigen semantischen Aussagenlogik wahr und falsch. Die in anderen Sprachen benutzten Symbole 1 und 0 bzw. L und 0 zur Darstellung eines logischen Wertes sind bereits als Ziffer bzw. Buchstabe in ALGOL vergeben.

Syntax logischer Wert:

< *logischer wert* > ::= **true** | **false**

Die logischen Werte haben eine feste offensichtliche Bedeutung.

Hinweis:
Es soll darauf hingewiesen werden, dass die Unterstreichungen und fett überall in der Bezugssprache zur Definition und Darstellung unabhängiger Grundsymbole benutzt wird. Das bedeutet, dass die Grundsymbole keine Beziehung zu den einzelnen Buchstaben und Ziffern haben, aus denen sie zusammengesetzt sind. Innerhalb dieses Skriptes wird die Unterstreichung zu keinem anderen Zweck verwendet.
In ALGOL 60-R300 wurden zur Definition der unabhängigen Grundsymbole diese Symbole in Hochkommata eingeschlossen.

Beispiel:
Die Aussage 34 > 33 besitzt den internen logischen Wert **true**, während die Aussage 34 < 33 den internen logischen Wert **false** enthält. Diese internen logischen Werte können u.a. in Bedingungen ausgewertet und zur Steuerung des Programmablaufes herangezogen werden.

2.1.4. Begrenzer

Alle Grundsymbole von ALGOL außer Buchstaben, Ziffern und logischen Werten werden Begrenzer genannt. Fünf verschiedene Arten sind definiert:

- Operatoren
- Trennzeichen
- Klammern
- Vereinbarungszeichen
- Spezifikationen

Syntax Begrenzer:

< *begrenzer* > ::= < *operator* > | < *trennzeichen* > | < *klammer* >|
< *vereinbarungszeichen* > | < *spezifikation* >

2.1.4.1. Operatoren

Ein Operator ist eine mathematische Vorschrift (ein Kalkül), durch die man aus mathematischen Objekten neue Objekte bilden kann.
In ALGOL wurden vier Arten von Operatoren definiert:

- arithmetische Operatoren
- Vergleichsoperatoren
- Logische Operatoren
- Folgeoperatoren

Syntax Operator:

< *operator* > ::= < *arithmetische operator* > | < *vergleichsoperator* > | < *logischer operator* > |
< *folgeoperator* >

Arithmetische Operatoren, Vergleichsoperatoren und logische Operatoren dienen der numerischen und logischen Verknüpfung von Werten, während der Folgeoperator zur Bildung von Programmschleifen, den Laufanweisungen, benutzt wird.

2.1.4.1.1. Arithmetische Operatoren

Syntax arithmetischer Operator:

< *arithmetischer operator* > ::= + | - | * | / | ÷ | ↑

Die arithmetischen Operatoren der vier Grundrechenarten sind in der Regel durch den Gebrauch in der Mathematik bekannt, sie werden deshalb nicht näher erläutert.

Erklärung:

+	Addition,
-	Subtraktion,
*	Multiplikation,
/	Division,
÷	ganzzahlige Division ist nur anwendbar, wenn Zähler und Nenner ganzzahlig sind, das Ergebnis ist der ganzzahlige Anteil des Quotienten. Z.B. 7 ÷ 2 = 3
↑	Potenzieren, z.B. a↑2 bedeutet a hoch 2

Regeln zur Nutzung der Operatoren wird im Punkt Arithmetische Ausdrücke erläutert.

2.1.4.1.2. Vergleichsoperatoren

Syntax Vergleichsoperator:

< *vergleichsoperator* > ::= < | ≤ | = | ≥ | > | ≠

Die Vergleichsoperatoren können in arithmetische Ausdrücke eingebunden werden, um Programmabläufe über Vergleiche zu steuern. Das Ergebnis eines Vergleichs ist ein logischer Wert.

Erklärung:

<	kleiner als
≤	kleiner oder gleich
=	gleich im Sinne der Identität, siehe Erklärung zum Gleichheitsbegriff
≥	größer oder gleich
>	größer
≠	ungleich

Die Nutzung dieser Vergleichsoperatoren wird im Kapitel „ Ausdrücke“ dargestellt.

Erklärung zum Gleichheitsbegriff:

Sind X und Y gleich, so schreibt man

X = Y (X ist gleich Y),

sind sie nicht gleich, so schreibt man

X ≠ Y (X ist ungleich Y).

Dann gilt aus rein logischen Gründen:

1. X = X für jedes X (**Reflexivität**)
2. Aus X = Y folgt Y = X (**Symmetrie**)
3. Aus X = Y und Y = Z folgt X = Z (**Transitivität**).

Nach den Leibnitzschen Ersetzbarkeitstheorem kann unter der Voraussetzung X = Y in jeder Aussage nach Belieben X durch Y ersetzt werden. Der Vergleichsoperator = ist also im Sinne der **Identität** zu verstehen.

2.1.4.1.3. Logische Operatoren

Syntax logischer Operator:

< *logischer operator* > ::= ≡ | ⊃ | ∨ | ∧ | ¬

Die logischen Operatoren können in logische Ausdrücke eingebunden werden, um Programmabläufe über logische Operationen zu steuern. Das Ergebnis einer solchen logischen Operation ist ein logischer Wert.

Erklärung:

≡	Äquivalenz
⊃	Implikation
∨	Disjunktion
∧	Konjunktion
¬	Negation

Die aufgeführten logischen Funktionen werden im Kapitel „Logische Ausdrücke“ detailliert besprochen.

2.1.4.1.4. Folgeoperatoren

Folgeoperatoren werden zur Bildung von Programmverzweigungen und Programmschleifen benutzt.

Syntax Folgeoperator:

< *folgeoperator* > ::= **goto | if | then | else | for | do**

Erklärung:

goto	unbedingter Sprung
if	wenn; Bedingung in bedingter Anweisung
then	dann; Bedingung in bedingter Anweisung
else	sonst; Bedingung in bedingter Anweisung
for	für; Anfangswert in Laufanweisung
do	führe aus; Anweisungsteil in einer Lausanweisung

2.1.4.2. Trennzeichen

Als Trennzeichen werden in ALGOL diejenigen Grundsymbole bezeichnet, die andere Grundsymbole voneinander trennen. Typografische Merkmale wie Leerzeichen oder Wechsel in eine neue Zeile haben in der Referenzsprache keine Bedeutung; sie werden nur für die Erleichterung Lesen der Quelltexte verwendet.

Syntax Trennzeichen:

< *trennzeichen* > ::= , | . | $_{10}$ | : | ; | : = | ␣ | **step** | **until** | **while** | **comment**

Erklärung:

.	Dezimalpunkt zur Darstellung einer reellen Zahl
,	Listentrennzeichen
$_{10}$	Basiszehn zur Darstellung des Exponenten
:	Markierungszeichen, zur Darstellung einer Marke in markierten Anweisungen
;	Anweisungstrennzeichen am Ende einer Anweisung
:=	Ergibtzeichen
␣	Zwischenraumelement
step	Lauflistenelement, Schrittweite in einer Laufanweisung
until	Lauflistenelement, Endebedingung in einer Laufanweisung
while	Lauflistenelement, Endebedingung in einer Laufanweisung
comment	Kommentar, dient nur der Kommentierung der Quelltexte, der mit Semikolon abgeschlossen wird

2.1.4.3. Klammern

Klammern werden immer paarweise benutzt. Zu jeder öffnenden Klammer gehört eine schließende.

Syntax Klammer:

< *klammer* > ::= (|) | [|] | ` | ' | **begin** | **end**

Erklärung:

( )	runde Klammern, algebraische Klammern
[ ]	eckige Klammern, Indexklammern
` '	Anführungszeichen für Kettenanführungszeichen oder Kettenklammern
begin **end**	Anweisungsklammern, Zusammenfassung von Anweisungen

2.1.4.4. Vereinbarungszeichen

Jede im Programm benutzte Variable wird vom Anwender hinsichtlich der Nutzung festgelegt. Das muss entsprechend der benutzten Rechenanlagen nach festgelegten Regeln in den **ALGOL Vereinbarungen** unter Nutzung definierter Vereinbarungszeichen geschehen.

Syntax Vereinbarungszeichen:

< *vereinbarungszeichen* > ::= **boolean** | **integer** |**real** | **array** | **switch** | **procedure** | **own**

Erklärung:

boolean	die zu benutzende Variable kann nur logische Werte true und false aufnehmen
integer	**die** zu benutzende Variable kann nur ganze Zahlen aufnehmen
real	die zu benutzende Variable kann nur reelle Zahlen aufnehmen
array	die zu benutzenden Variablen werden in indizierten Feldern gespeichert, z.B. Vektoren und Matrizen
switch	Verteiler werden für Mehrfachverzweigungen benötigt
procedure	dient zur Realisierung von ALGOL Unterprogrammen
own	dient zur Erweiterung der Gültigkeitsdauer von Variablen bei Wiedereintritt in Blöcke und Unterprogramme

2.1.4.5. Spezifikationszeichen

Spezifikationszeichen dienen der näheren Kennzeichnung der Parameter in der Unterprogramtechnik.

Syntax Spezifikationszeichen:

< *spezifikationszeichen* > ::= **string** | **label** | **value**

Erklärung:

string	der zu benutzende Parameter ist eine Zeichenkette
label	der zu benutzende Parameter ist eine Marke
value	definiert den Werteteil einer formalen Parameterliste

Alle Trennzeichen, Klammern, Vereinbarungszeichen und Spezifikationszeichen werden später im Zusammenhang mit den Anwendungen in den speziellen Anweisungen und Unterprogrammen ausführlich besprochen.

2.2. Bezeichner

Im wissenschaftlichen und kommerziellen Bereichen ist es sinnvoll und hilfreich, alle benutzte Variable mit Bezeichnungen zu versehen, die in etwa zu einem inhaltlichen Bezug zum Verwendungszweck und Eigenschaften der benutzten Objekte herstellen. Man spricht von kontextbezogener Vergabe von Bezeichnungen, von **mnemotechnischen** Bezeichnungen. Auch in der Bezugssprache ALGOL werden alle Elemente der Sprache mit Namen versehen, die natürlich nicht mit den vom Nutzer vergebenen Namen übereinstimmen dürfen. Man spricht von **reservierten Namen** seitens ALGOL. Im Wesentlichen sind alle reservierten Namen in dem Kapitel und im Punkt 3.2.- Standardfunktionen - vorgestellt.

Eine vom Nutzer vergebene Bezeichnung nennt man in ALGOL **Bezeichner**, der aus einer beliebigen Folge von Buchstaben und Ziffern, beginnend mit einem Buchstaben, bestehen kann. Im ALGOL-Bericht sind max. 25 Zeichen vorgesehen. Sie haben keine selbständige Bedeutung, sondern dienen zur Kennzeichnung von einfachen Variablen, Feldern, Marken, Verteilern und Prozeduren. Sie müssen innerhalb eines Blockes eindeutig sein, doppelte Vergabe des gleichen Namens führt zu Übersetzungsfehlern.

Syntax Bezeichner:

< *bezeichner* > ::= < *buchstabe* > | < *buchstabe* > < *buchstabe* > | < *buchstabe* > < *ziffer* >

Im ALGOL-Bericht ist festgelegt, dass der Name mindestens einen Buchstaben haben muss, nach oben sind aber keine Grenzen gesetzt. Allerdings ist durch die benutzte Rechenanlage Grenzen hinsichtlich Länge des Namens sowohl auch der zu verwendenden Groß- und Kleinbuchstaben vorgegeben. In ALGOL 60-R300 waren sechs Zeichen für den Bezeichner erlaubt, bestehend aus Ziffern und Kleinbuchstaben.

Beispiele:

Wert
wert1
Summe
Kette

Falsche Bezeichner:

1Wert
wert 1
Summe_1
string

2.3. Zahlen

Wegen der fundamentalen Bedeutung der Zahlen und des Rechnens mit Ihnen ist es notwendig, über den strengen Aufbau der verschiedenen Zahlenbereiche in der Mathematik zumindest informiert zu sein. Für unsere Zwecke ist es natürlich nicht notwendig, den gesamten genetischen Aufbau der Zahlensysteme zu erörtern. Selbstverständlich müssen wir die von ALGOL festgelegten Zahlensysteme und die Regeln zum Rechnen mit diesen vorgegebenen Zahlensystemen kennenlernen.
Im ALGOL-Bericht sind nur Zahlen des Dezimalsystems zugelassen, Rechnen mit anderen Zahlensystemen, wie z.B. Dualsystemen oder Hexadezimalsystemen, ist nicht möglich. Ebenso ist das Rechnen mit komplexen Zahlen nicht möglich, nur reelle Zahlen sind erlaubt. Die Zahlen selbst dürfen nur nach bestimmten Regeln gebildet werden. Wertebereiche werden im ALGOL-Bericht nicht festgelegt. Auch hier gilt die gleiche Aussage, die schon bei der Bildung von Bezeichner gemacht wurde, dass durch die benutzte Rechenanlage der Wertebereich entsprechend ihrer Konfiguration vorgegeben wird.
Zum Aufbau der Zahlen stehen dem Anwender 14 Grundsymbole zur Verfügung, die bereits definierte Ziffer, die Vorzeichen + und - , der Dezimalpunkt und die Basiszehn.

Syntax Zahl:

< *zahl* > ::= < *ganze zahl* > | < *dezimalbruch* > | < *exponententeil* > | < *dezimalzahl* >
< *ganze zahl* > ::= *[{±}]* < *ziffer* > | < *ganze zahl* > < *ziffer* >
< *dezimalbruch* > ::= . < *ganze zahl ohne vorzeichen* >
< *exponententeil* > ::= $_{10}$ < *ganze zahl* >
< *dezimalzahl* > ::= < *ganze zahl ohne vorzeichen* > | < *dezimalbruch* > |
< *ganze zahl ohne vorzeichen* > < *dezimalbruch* >

Erklärung:

Ganze Zahl; Eine ganze Zahl kann eine vorzeichenbehaftete Ziffer **oder** eine vorzeichenbehaftete Ziffer **und** eine weitere Ziffer, somit eine Folge von Ziffern mit wahlweisem Vorzeichen sein.

Dezimalzahl; Eine Dezimalzahl ist entweder eine ganze Zahl oder eine nicht ganze Zahl mit wahlweisem Vorzeichen, einer Folge von Ziffern mit vorrangehendem Dezimalpunkt, der sog. **Festkommadarstellung**.

Exponententeil; Die Zehnerpotenz kann mit und ohne Vorzeichen allein Zahl stehen. Zur Darstellung sehr kleiner und großer Zahlen ist die halblogarithmische Darstellung mit Hilfe des Exponententeils in ALGOL möglich, bestehend aus Mantisse, Basis und Exponententeil. Man spricht hier von der **Gleitkommadarstellung**.

Jede ALGOL-Zahl hat eine der 3 Formen:

1. Dezimalzahl
2. Vorzeichen $_{10}$ ganze Zahl
3. Dezimalzahl $_{10}$ ganze Zahl

Beispiele zu 1.: 25 +25 -123 .001 0.001
Beispiele zu 2.: $+_{10}10$ $_{10}10$ $+_{10}2$ $-_{10}-2$ $_{10}0$ $_{10}-0$
Beispiele zu 3.: $25_{10}10$ $-25_{10}-10$ $-123_{10}3$ $.001_{10}2$ $100_{10}-2$ $-23.1_{10}-02$

2.4. Ketten, Zeichenketten

Bis jetzt wurden Konstrukte vorgestellt, deren Grundsymbole nur nach ganz bestimmten Regeln aneinandergereiht werden dürfen, sofern sinnvoller ALGOL-Text erzeugt werden soll. Nun ist es sicher notwendig, dass in einem ALGOL-Programm Text formuliert werden kann, der nicht den syntaktischen Regeln gehorchen muss, somit eigentlich nicht zu ALGOL gehört. Um die Sprache in die Lage zu versetzen, willkürliche Folgen von Grundsymbolen zu programmieren, die z.B. ausgedruckt werden sollen, ist es notwendig, über festgelegte Symbole derartige Ketten zu formulieren. Über sog. string quotes, den Zeichenkettenklammern, werden Ketten formuliert, die den o.g. Forderungen gerecht werden. Diese paarweise zu benutzenden Zeichenkettenklammern ` ' schließen die Kette ein, die im Gegensatz zum **comment** in die Verarbeitung einbezogen wird.

Syntax Kette:

< kette > ::= ` < offene kette > '
< offene kette > ::= < echte kette > | ` < offene kette > ' | < offene kette > < offene kette >
< echte kette > ::= < jede beliebige folge von grundsymbolen außer ` ' > | < leere kette >
< leere kette > ::=

Erklärung:

Eine Kette ist eine beliebige Folge von Grundsymbolen, die mit einer öffnenden Kettenklammer ` beginnt und der schließenden Kettenklammer ' endet. Im Inneren einer Kette können weitere Kettenklammern auftreten, die dann paarweise geordnet sein müssen. Innerhalb der Kettenklammern können auch Leerzeichen auftreten, die leere Kette. Das darin enthaltende Trennzeichen ␣ hat als Zwischenraumelement nur in Ketten Bedeutung, als Trennzeichen außerhalb ist es syntaktisch falsch. Freigelassener Platz zwischen ALGOL-Zeichen wird ignoriert.

Beispiele:

`Das␣ist␣eine␣Zeichenkette′
`Kettenklammern␣in␣der␣Kette``Ausgabetext␣in␣einen␣Klammer"␣␣␣'
`Preis = 25,23 EUR'

Anmerkung:

Die klassische Textverarbeitung, wie sie heute in allen modernen Programmiersprachen üblich ist, war im ALGOL-Bericht noch nicht vorgesehen. Es ist nicht möglich, Zeichenketten zu spezifizieren, ein Vereinbarungszeichen für Ketten gibt es in ALGOL nicht, somit sind interne Zeichenkettenverarbeitungen wie vergleichen, sortieren, Bildung von Teilketten, Verkettungen, usw. nicht durchführbar.

3. Variable und Ausdrücke

Hauptbestandteile der Programme in der Sprache ALGOL zur Beschreibung von Prozessen sind Anweisungen, die bis auf leere Anweisungen Ausdrücke enthalten. Anweisungen werden im nächsten Kapitel eingehend besprochen.

Ausdrücke lassen sich in drei Gruppen aufteilen:

- die arithmetischen Ausdrücke
- die logischen Ausdrücke
- die Zielausdrücke.

Syntax Ausdruck:

< *ausdruck* > :: = < *arithmetischer ausdruck* > | < *logischer ausdruck* > | < *zielausdruck* >

Bestandteile dieser Ausdrücke, mit Ausnahme bestimmter Begrenzer, sind Variable, logische Werte, Zahlen, Funktionen und elementare arithmetische, logische und Vergleichsoperationen sowie Folgeoperationen.

3.1. Variable

Damit Ausdrücke in ALGOL programmiert werden können, benötigt man zur Aufnahme der Daten symbolische Speicherplätze, die Variablen. Diese aufzunehmenden Daten können numerische oder logische Werte sein. Diese symbolischen Speicherplätze müssen vor der Datenaufnahme bereitgestellt werden, sie müssen durch eine gültige Typ- oder Feldvereinbarungen eindeutig definiert werden.

Syntax der Variable:

< *variable* > ::= < *einfache variable* > | < *indizierte variable* >

3.1.1. Einfache Variable

Eine einfache Variable ist eine Bezeichnung, die einen einzigen Wert aufnehmen kann. Dieser Wert kann in Ausdrücken für die Bildung andere Werte verwendet werden und kann nach Belieben durch Zuweisungen geändert werden. Der Typ des Wertes einer bestimmten Variablen wird in der Vereinbarung für die Variable selbst festgelegt. Diese verschiedenen Typen kennzeichnen permanent die Eigenschaften der zugewiesenen Werte.
Der Compiler, der aus dem Quellprogramm den sog. Objektcode erzeugt, weist jeder Variablen einen konkreten Speicherplatz zu. Der Variablenname stellt somit die symbolische Adresse der Variablen dar. Die auf diesem Speicherplatz stehende Zahl ist der zugewiesene Wert der Variablen.

Syntax einfache Variable:

< *einfache variable* > :: = < *bezeichner* >

Einführendes Beispiel:

Es soll der Kreisumfang u = 2 Π r für den Radius r = 25 cm berechnet und ausgedruckt werden. Das Drucken wird in dem folgenden ALGOL-Programm nur durch eine symbolische Anweisung dargestellt:

```
begin
    real umfang, radius, pi;
        radius := 25;
        pi := 3.1415;
        umfang := 2 * pi * radius;
        print (umfang);      comment Die Funktion print wird hier nur aus didaktischen Gründen
                                     benutzt; sie ist kein Element der Sprache ALGOL 60
end
```

Erklärung:
Das Programm wird durch die Klammern **begin** und **end** definiert. Die einfachen Variablen umfang und radius werden in der Typvereinbarung als Größen vereinbart, die reelle Dezimalzahlen speichern können. Der Variablen umfang wird durch eine Ergibtanweisung die dezimale konstante Zahl 25 zugewiesen, das Ergibtzeichen := definiert die Zuweisung (siehe 4.2.). Das Ergebnis der Auswertung des einfachen arithmetischen Ausdrucks wird der Variablen umfang zugewiesen und mittels der symbolischen Funktion **print** ausgedruckt.

3.1.2. Indizierte Variable

Felder mit indizierten Variablen bezeichnen ein ganzes Feld von Werten. Jede einzelne Variable dieses Feldes, die indizierte Variable, stellt eine Komponente dieses Feldes dar, eingeschlossen in Indexklammern. Die in den Indexklammern eingeschlossene Liste heißt **Indexliste**. Jeder Wert innerhalb dieser Indexliste nimmt eine Indexposition der indizierten Variablen ein und wird als **Indexausdruck** bezeichnet. Die Anzahl der Indexpositionen ist gleichbedeutend mit der Dimension des Feldes.

Syntax indizierte Variable:

< *indizierte variable* > :: = < *feldbezeichner* > [< *indexliste* >]
< *feldbezeichner* > ::= < *bezeichner* >
< *indexliste* > ::= < *indexausdruck* > | < *indexliste* > , < *indexausdruck* >
< *indexausdruck* > ::= < *arithmetischer ausdruck* > (siehe 3.3.)

Beispiele:

Vek [5]	die 5. Komponente des Vektors Vek, die 5. Komponente des eindimensionalen Feldes Vek
Mat [3,4]	das Element der 3.Zeile und der 4.Spalte der Matrix Mat
Feld [m,n]	das Element der m.Zeile und der n.Spalte des zweidimensionalen Feldes Feld
Ten [3,4,5]	das Element(3,4,5) eines dreidimensionalen Feldes, eines Tensors
E [3,3]	das Element der 3.Zeile und der 3.Spalte der Hauptdiagonalen der Einheitsmatrix E
Tab [m – n, a * b]	die Indizes sind arithmetische Ausdrücke
A [m[i + 1], k+1]	das Element in der m. (i+1). Zeile und der (k+1). Spalte der Matrix A

Bemerkung:
Die in den Beispielen benutzen Begriffe Vektor, Matrix und Tensor sind lediglich mathematische Bezeichnungen für ein-, zwei- und dreidimensionale Felder. Die Namensvergabe schließt nicht verbindlich auf seinen Anwendungsbereich, sollte aber einen Bezug zum Verwendungszweck herstellen.
Mehrdimensionale Felder werden grundsätzlich zeilenweise verarbeitet. Um die Indizes einer indizierten Variablen mit arithmetischen Ausdrücken zu ermitteln, müssen zunächst ihre Indexausdrücke mit den aktuellen Werten berechnet werden. Der errechnete Wert des arithmetischen Ausdrucks wird anschließend auf die nächste ganze Zahl gerundet und kann dann in dieser Form als Index zum Adressieren der indizierten Variablen benutzt werden.

3.1.3. Typvereinbarungen für einfache Variable

Für jede in einem ALGOL-Programm oder in einem Unterprogramm vorkommende Variable muss ein symbolischer Speicherplatz angefordert werden, der sich entsprechend des vereinbarten Typs in seiner Länge voneinander unterscheidet. Der Typ des Wertes kann entsprechend seiner Anwendung ein numerischer oder ein logischer Wert sein, also vom Typ **real, integer** oder **boolean**. Im ALGOL-Bericht sind nur ganzzahlige Zahlen und nicht ganzzahlige Zahlen definiert, die mit **integer** und **real** vereinbart werden. Die Grundrechenarten Addition, Subtraktion und Multiplikation können mit **integer** Zahlen durchgeführt **werden**, während die ganzzahlige Division nur für **integer** Zahlen definiert ist. Alle anderen Rechenarten werden mit **real** Zahlen durchgeführt.
Im ALGOL-Bericht werden über interne Zahlendarstellungen, Speicherungsformen und Genauigkeiten bestimmter Typen absichtlich keine Festlegungen getroffen; sie sind Gegenstand maschinengebundener Darstellungsformen von ALGOL.

Syntax Vereinbarung:

< *vereinbarung* > *::=* < *typvereinbarung* > | < *feldvereinbarung* > | < *verteilervereinbarung* > |
< *prozedurvereinbarung* >

Verteilervereinbarung und Prozedurvereinbarung wird in späteren Kapiteln besprochen.

Syntax Typvereinbarung für einfache Variable:

< *typvereinbarung* > *::=* < *typ* >< *typenliste* > | **own**< *typ* >< *typenliste* >
< *typ* > *::=* **integer** | **real** | **boolean**
< *typenliste* > *::=* < *einfache variable* >|< *typenliste* > *,*< *einfache variable* >

Hinweis:
Zur Darstellung der Vereinbarung von Variablen gelten bestimmte Regeln:

- jede verwendete Variable muss in einer Typvereinbarung vereinbart werden
- jede Typvereinbarung steht vor seiner Benutzung als Variable am Programmanfang
- jede einzelne Typvereinbarung wird mit einem Trennzeichen ; abgeschlossen
- in einer Liste aufgeführten Variablennamen werden durch das Trennzeichen , voneinander getrennt
- die vergebenen Variablennamen müssen eindeutig sein, dürfen keine reservierten Namen sein und sich an die Namenskonventionen für Bezeichner halten.

Beispiele Typvereinbarung einfache Variable:

real Pi;
own real Pi;
real wert, Ergebnis, Summe, Wurzel;
integer i, j, k;
integer Laenge, Breite, Hoehe;
boolean Vergleich

Erklärung:
Integer-vereinbarte Variable können nur ganzzahlige positive und negative Dezimalzahlen und die Zahl 0 aufnehmen, die sog. Festkommadarstellung (siehe Punkt 2.3.). Der Wertebereich wird durch den benutzten Rechner festgelegt. Real-vereinbarte Variable können positive und negative Dezimalzahlen und die Zahl 0 entsprechend den Festlegungen im Punkt 2.3. aufnehmen, die sog. Gleitkommadarstellung. Auch hier gilt der vorgegebene Wertebereich der benutzten Rechenanlage.

Bei Angabe des Wortsymbols **own** in der Typvereinbarung behält die vereinbarte Variable bei Verlassen eines Blockes ihren Wert, mit dem dann bei Wiedereintritt in diesen Block weitergerechnet werden kann. Bei Weglassen der **own**-Angabe geht der aktuelle Wert der Variablen bei Verlassen eines Blockes verloren und wird bei Wiedereintritt in diesen Block neu initialisiert.

3.1.4. Feldvereinbarungen für indizierte Variable

Feldvereinbarungen bedeuten für indizierte Variable dasselbe wie die Typvereinbarung für einfache Variable. Während in der Typvereinbarung jede Variable einzeln vereinbart werden muss, wird mit der Feldvereinbarung die Menge der Variablen vereinbart, die in ihr angegeben sind. Über die Anzahl der Indizes und die Indexgrenzen ist die Anzahl der indizierten Variablen definiert. Zur Kennzeichnung der Feldvereinbarung wird das Vereinbarungszeichen **array** herangezogen.

Syntax Feldvereinbarung für indizierte Variable:

< *feldvereinbarung* > ::= **array**< *feldliste* > |< *typ* > **array**< *feldliste* > |
own< *typ* > **array**< *feldliste* >
< *typ* > ::= **integer** | **real** | **boolean**
< *feldliste* > ::=< *feldsegment* > |< *feldliste* > ,< *feldsegment* >
< *feldsegment* > ::=< *feldbezeichnung* > [< *grenzenliste* >] |
< *feldbezeichnung* > ,< *feldsement* >
< *grenzenliste* > ::=< *grenzenpaar* > |< *grenzenliste* > ,< *grenzenpaar* >
< *grenzenpaar* > ::=< *untere grenze* > :< *obere grenze* >
< *untere grenze* > ::=< *arithmetischer ausdruck* >
< *obere grenze* > ::=< *arithmetischer ausdruck* > (siehe 3.3.)

Beispiele Feldvereinbarung:

integer array E[1:3,1:3];	Einheitsmatrix 3 Zeilen, 3 Spalten
array Feld [n:m, k:l], Ber [1:5];	ohne Typ ist real Standard für beide Felder
integer array F1, F2, F3 [1:m, 0:n];	drei Felder mit den Feldangaben
own integer array Mat [1:10, 1:25] ;	Matrix mit 250 indizierten statischen Variablen
array Bereich [-10 : 0, 15 : 30];	
own integer array Mat1 [**if** i< 0 **then** 0 **else** 1 : 20];	untere Grenze ist 0 oder 1 in Abhängigkeit von i
array EntscheidTab [**if** a – b< 0 **then** b **else** a : **if** x ≤ 0 **then** 1 **else** x, -10 : 0];	
boolean array Selektor [0 : 10];	Feld für indizierte logische Variable zur Aufnahme von logischen Werten **true** und **false**.

Bemerkung:
Da sich die in den arithmetischen Ausdrücken der Indexgrenzen eingehende Werte während der Ausführung des Programmes ändern können, können sich auch die Größe des Feldes und damit die Größe des zu reservierenden Speicherplatzes bei Eintritt in den Block ändern. Diese dynamischen Felder werden bei der Blockverarbeitung später näher erläutert. Am Programmanfang dürfen selbstverständlich keine variablen Indexgrenzen angegeben werden, da ja vorher keine Wertzuweisung an diese Variablen erfolgen kann.

3.2. Standardfunktionen

In arithmetischen und logischen Ausdrücken werden mathematische Aufgabenstellungen dadurch vereinfacht, indem man für bestimmte mathematische Standardlösungen die von ALGOL angebotenen **Standardfunktionen** benutzt. Standardfunktionen brauchen nicht vereinbart zu werden. Sie sind fester Bestandteil der Sprache ALGOL. Im ALGOL-Bericht sind neun Standardfunktionen festgelegt, es betrifft die Funktionen **abs, arctan, cos, entier, exp, ln, sign, sin, sqrt**.
Für die Nutzung auf bestimmten Rechenanlagen wurden spezielle Ein- und Ausgabefunktionen hinzugefügt. Für ALGOL 60-R300 wurden beispielsweise fünf Ein- und Ausgabefunktionen festgelegt. Diese Funktionsbezeichnungen dürfen nicht zur Bezeichnung anderer Größen verwendet werden, man spricht von den bereits erwähnten reservierten Namen.

Die Standardfunktionen enthalten entsprechend ihrer Namensgebung den vollständigen Algorithmus der Aufgabenstellung. Der Anwender ruft diese Funktion über die Funktionsbezeichnung auf und übergibt in der Parameterliste den aktuellen Parameter. Als Ergebnis dieser internen Funktionsberechnung erhält der Anwender den entsprechenden Funktionswert. Ausführlich werden Funktionsaufrufe sowie die Parameterübergabe später besprochen; hier wird nur eine Sonderform der Funktionen vorgestellt, die bei der Nutzung der arithmetischen Ausdrücke sehr hilfreich sein kann. Arithmetische Ausdrücke werden im Punkt 3.3. ausführlich behandelt.

Standardfunktionen haben die Form f(x), wobei f eine der neun Standardfunktionen und x den aktuellen Parameter beschreibt. Als aktueller Parameter darf ein beliebiger arithmetischer Ausdruck innerhalb des Definitionsbereichs der benutzten Funktion eingesetzt werden. Der Funktionswert ist mit Ausnahme der Funktionen **entier**(x) und **sign**(x) vom Typ **real**, der Funktionswert der beiden genannten Funktionen ist vom Typ **integer**. Boolsche Standardfunktionen sind in ALGOL-Bericht nicht vorgesehen.

Standardfunktionen:

- **abs** (x) liefert den absoluten Betrag des Wertes x ,
- **sign** (x) liefert das Signum für x (+1 für x > 0, 0 für x = 0, -1 für x< 0)
- **sqrt** (x) liefert für die positive Quadratwurzel des Wertes von x >= 0
- **sin** (x) liefert den Sinus des Wertes von x
- **cos** (x) liefert den Kosinus des Wertes von x
- **arctan** (x) liefert den Hauptwert des Arcustangens des Wertes von x
- **ln** (x) liefert den natürlichen Logarithmus des Wertes x > 0
- **exp** (x) liefert die Exponentialfunktion des Wertes von x ($e^{\uparrow}x$)
- **entier**(x) liefert die größte ganze Zahl, die kleiner oder gleich x ist

Beispiele werden im Zusammenhang mit den arithmetischen Ausdrücken aufgezeigt.

Hinweis zur Funktion entier(x):
Es ist oft nützlich bzw. zwingend, einen in einer arithmetischen Operation errechneten Wert vom Typ **real** intern nach der mathematischen Rundung einer mit **integer** vereinbarten Variablen zuzuweisen. Um Rundungsfehler zu vermeiden, hat der ALGOL-Bericht die **Rundungsvorschrift** festgelegt. Das Ergebnis der Rundung ist äquivalent mit

entier(x + 0.5),

wobei x der Wert des Ausdrucks ist, dessen Typ umgewandelt werden soll.

Beispiele für Rundungen:

entier(4.8+0.5) = 5, aufgerundet auf 5
entier(4.5+0.5) = 5, aufgerundet auf 5
entier(4.3+0.5) = 4, abgerundet auf 4

3.3. Arithmetische Ausdrücke

Ein arithmetischer Ausdruck ist eine Regel zur Berechnung eines numerischen Wertes, bei der die Gesetzmäßigkeiten der Mathematik gelten, allerdings besteht keine strenge Analogie zu den herkömmlichen arithmetischen Formeln. Bei der Umsetzung arithmetischer Formeln in arithmetische Ausdrücke in ALGOL muss die Schreibweise dahingehend präzisiert werden, dass Mehrdeutigkeiten bei der maschinellen Abarbeitung völlig ausgeschlossen sind. Folgerichtig wurden im ALGOL-Bericht Regeln zur Erstellung und Auswertung arithmetischer Ausdrücke festgelegt. Darüber hinaus lassen sich bei der Erstellung arithmetischer Ausdrücke Bedingungen formulieren, durch deren Auswertung einer der verschiedenen arithmetischen Ausdrücke zur Abarbeitung ausgewählt wird.

Syntax arithmetischer Ausdruck:

< arithmetischer ausdruck > ::= < einfacher arithmetischer ausdruck > |
< bedingter arithmetischer ausdruck >

3.3.1. Einfache arithmetische Ausdrücke

Ein arithmetischer Ausdruck jeden Umfangs setzt sich aus Operanden, Operatoren und algebraische Klammern zusammen.

Syntax einfacher arithmetischer Ausdruck:

< einfacher arithmetischer ausdruck > ::=
< arithmetischer operand > < arithmetischer operator > < arithmetischer operand >

Syntax arithmetischer Operand:

< arithmetischer operand > ::=< zahl > |*< variable >* |*< funktion >*

Syntax arithmetischer Operator:

< arithmetischer operator > ::= + | - | * | / | ÷ | ↑

Syntax algebraische Klammern:

< algebraische klammer > ::= (|)

Die Definitionen der arithmetischen vier Grundrechenarten und der arithmetischen Klammern sind offensichtlich. Der Operator ÷ bedeutet die ganzzahlige Division. Sie ist nur anwendbar, wenn Zähler und Nenner ganzzahlig und vom Typ **integer** sind. Der Operator ↑ bedeutet Potenzierung. Dieses Zeichen ersetzt die Hochstellung, da aufgrund der Zeilenkonvention eine Hoch- oder Tiefstellung von Zeichen nicht erlaubt ist.

Beispiele:

siehe Punkt 3.3.3., Regeln zur Auswertung arithmetischer Ausdrücke

3.3.2. Bedingte arithmetische Ausdrücke

Ein bedingter arithmetischer Ausdruck erlaubt die Auswahl über eine Bedingung zwischen zwei arithmetischen Ausdrücken, wobei verschachtelte Bedingungen erlaubt sind. Diese Konstrukte werden später ausführlich besprochen.

Syntax bedingter arithmetischer Ausdruck:

< *bedingter arithmetischer ausdruck* > ::= **if** < *logischer ausdruck* > **then** < *arithmetischer aus druck* > **else** < *arithmetischer ausdruck* >

Erklärung:
Die logischen Werte hinter if werden entsprechend ihrer aktuellen Zuweisungen von links nach rechts durchgehend nacheinander ausgewertet, bis einer mit dem logischen Wert **true** gefunden wird. Der Wert des gesamten arithmetischen Ausdrucks ist dann der Wert, der hinter **then** aufgelistet wurde. Ist dann der logische Wert nach der Auswertung nicht **true** sondern **false**, so ist dann der Wert des gesamten arithmetischen Ausdrucks der Wert, der hinter **else** steht.
Die logischen Ausdrücke beschränken sich zunächst nur auf Operationen mit dem bereits definierten Vergleichsoperatoren; die logischen Ausdrücke in ihrer Gesamtheit werden im Punkt 3.4. besprochen.

Beispiele:

1.) **if** i = j **then** 1 **else** 0 Wenn z.B. die Zeilen- und Spaltenzahl einer Matrix identisch sind, wird der Wert 1, ansonsten 0 angewiesen.

2.) **if** q ≠ 0 **then** p/q **else** p Wenn der Nenner 0 ist, dann wird die Division nicht ausgeführt, da Division durch Null mathematisch nicht definiert ist

oder

if q = 0 **then** p **else** p / q

3.) **if** d ≥ 0 **then** **sqrt**(d) **else** **sqrt**(-d)
Wenn der aktuelle Wert d größer gleich 0 ist, wird aus diesem Wert die positive Quadratwurzel gezogen, ansonsten wird der aktuelle Wert mit -1 multipliziert und aus diesem Wert die positive Quadratwurzel gezogen

4.) a + (**if** a< x **then** b **else** x)
Fall 1: a = 5, b= 1, x = 6 ist der arithmetische Wert gleich 6
Fall 2: a = 5, b= 1, x = 4 ist der arithmetische Wert gleich 9
Fall 3: a = 5, b= 1, x = 5 ist der arithmetische Wert gleich 10

5.) **if** a < 0 **then** b - a **else** if b< 0 **then** a - b **else** 0
Hinter **else** steht ein weiterer bedingter Ausdruck. Die Abarbeitung erfolgt von links nach rechts. Ist a< 0, wird dieser bedingte Ausdruck nicht ausgewertet.

Die sinnvolle Anwendung der arithmetischen Ausdrücke wird bei der Besprechung der Anweisungen im Punkt 4. dargestellt.

3.3.3. Regeln zur Auswertung arithmetischer Ausdrücke

Bei der Umsetzung mathematischer Formeln in eine ALGOL-Notation müssen Regeln beachtet werden, um Mehrdeutigkeiten bei der maschinellen Abarbeitung auszuschließen. Bei der Bildung arithmetischer Ausdrücke sind folgende **Grundregeln** einzuhalten:

1. Arithmetische Operatoren dürfen nicht unmittelbar aufeinander folgen. Im Bedarfsfall sind Klammern zu setzen.
2. Arithmetische Operanden und arithmetische Klammerausdrücke dürfen nicht unmittelbar aufeinander folgen. Insbesondere darf das Multiplikationszeichen nicht wegelassen werden, wie es in der mathematischen Umgangsschreibweise üblich ist.

Bei der Berechnung mathematischer Ausdrücke gilt die allgemeine Regel: Punktrechnen geht vor Strichrechnen. Diese Regel gilt bei der Bildung arithmetischer Ausdrücke ebenso, da die Auflösung dieser Algorithmen in Maschinensprache durch den entsprechenden Compiler sich streng nach dieser Abarbeitungslogik erfolgt. Somit gelten bei der Erstellung arithmetischer Ausdrücke folgende **Prioritätsregeln**:

1. Klammerrechnung
2. Potenzierung
3. Multiplikation oder Division
4. Addition oder Subtraktion

Im Bedarfsfall sind Klammern zu setzen. Darüber hinaus gilt die für die Erstellung aller Ausdrücke die Regel, dass der Ausdruck von **links nach rechts** aufgelöst wird.

Beispiele für arithmetische Ausdrücke:

Gewöhnliche Schreibweise	ALGOL-Schreibweise	Inkorrekte, falsche Schreibweise	Begründung
$\frac{a}{-b}$	a/(-b)	a/-b	Grundregel 1
$\frac{-a}{b}$	-a/b		
a(-b)	a*(-b)	a*-b	Grundregel 2
(a+b)(a-b)	(a+b)*(a-b)	(a+b) (a-b)	Grundregel 2
(a+b)(a-b)	(a+b)*(a-b)	a + b * a - b	Prioritätsregel 1
(a+bc)d + 2(a+b)e	(a+b*c) * d + 2 * (a+b) * e	(a+b*c) d + 2(a+b) e	Grundregel 1
ax↑2 + bx + c	a * x↑2 + b * x + c		
$\sqrt{(r^2 + h^2)}$	**sqrt**(r * r + h * h)		
½ + ⅔	(1/2) + (2/3) oder 1/2 + 2/3		Im Bedarfsfall Klammern setzen
⅓ h (G+$\sqrt{GD + D}$)	1/3 * h * (G + **sqrt**(G * D) + D)	1/3 * h * G + **sqrt**(G * D) + D	Prioritätsregel 1
Π r^2 h	3.14 * r * r * h	3,14 * r * r * h	3,14 keine real-Zahl
2Πr	2 * 3.14 * r	2r * 3.14	Grundregel 1
⅓ Π x ↑ 3	1/3 * 3.14 * x ↑ 3	0.3 * 3.14 * x ↑ 3	Ungenaues Ergebnis
Π r^2 h	3.14 * r* r *h	Π * r * r * h	Π ungültiges Zeichen
a / (b +1) für (b + 1) ≠ 0	a / **if** (b +1)≠ 0 **then** a / (b+1) **else** 0	a / **if** (b +1)≠ 0 **then** a / b+1 **else** 0	Prioritätsregel 1

3.4. Logische Ausdrücke

Ein logischer, auch boolscher Ausdruck ist eine Formel zur Berechnung eines logischen Wertes, der im Gegensatz zu den arithmetischen Ausdrücken nur zwei logische Werte annehmen kann, **true** und **false**. Logische Ausdrücke dienen zur Formulierung von Bedingungen, die der wesentliche Bestandteil bedingter Anweisungen und bedingter Ausdrücke sind. Neben den im Aussagenkalkül der mathematischen Logik allgemeingültigen Operatoren zählen auch in ALGOL die arithmetischen Relationen bzw. Vergleiche zu den logischen Ausdrücken, da für Vergleichsergebnisse ebenfalls nur zwei logische Werte möglich sind; nämlich wenn die Relation durch die aktuellen Werte erfüllt ist, hat sie den logischen Wert **true**, in allen anderen Aussagen den Wert **false**.

Syntax logischer Ausdruck:

< *logischer ausdruck* > *::=*< *einfacher logischer ausdruck* > |< *bedingter logischer ausdruck* >

3.4.1. Einfache logische Ausdrücke

Ein logischer Ausdruck jeden Umfangs setzt sich aus Operanden, Operatoren und algebraische Klammern zusammen und wird nach ähnlichen Regeln gebildet wie der arithmetische Ausdruck.

Syntax einfacher logischer Ausdruck:

< *einfacher logischer ausdruck* > *::=*< *logischer operand* > < *logischer operator* >
< *logischer operand* >

Syntax logischer Operand:

< *logischer operand* > *::=* {**true** | **false**} |< *variable vom typ boolean* > |
< *vergleich* > |< *funktion vom typ boolean* >

Die Typvereinbarung **boolean** wurde im Punkt Typvereinbarungen erläutert, Funktionen werden später besprochen..

Syntax Vergleich:

< *vergleich* > *::=*< *einfacher arithmetischer ausdruck* >< *vergleichsoperator* >
< *einfacher arithmetischer ausdruck* >

Syntax Vergleichsoperator:

< *vergleichsoperator* > ::= < | ≤ | = | ≥ | > | ≠

Hinweis:
Es ist darauf hinzuweisen, dass Vergleichsoperatoren nicht zu den logischen Operatoren zählen, da sie ja keine logischen Werte verknüpfen, sondern lediglich Größen vom Typ **real** oder **integer**. Das Ergebnis ist immer vom Typ **boolean**. Mehrere Vergleiche lassen sich mit logischen Operatoren zusammenfassen. Die Operatoren selbst wurden in 2.1.4.1.2. , sowie der Gleichheitsbegriff, ausführlich dargestellt.

Beispiele für Vergleiche:

Vergleich	Aktuelle Werte	Ergebnis: logischer Wert
5.6 = 36		**false**
x< -y	x = 3, y = 2	**false**
x< 3.5	x = 3	**true**
abs(x) ≥ x	für alle x Beträge	**true**
abs(x + y) ≥ **abs** (x) + **abs** (y)	für alle x und y Beträge	**false**
x = -3.25	x = 3.25	**false**
x = (x + 1)	für alle x Beträge	**false**
x = x + 1	für alle x Beträge	**false**

Hinweis:
Der Vergleich X = X + 1 hat nichts mit einer iterativen Anweisung gemeinsam. Diese Schreibweise wird in vielen Sprachen benutzt, unterscheidet sich aber grundsätzlich von ALGOL 60 (siehe nächstes Kapitel).

Syntax logischer Operator:
< *logischer operator* > ::= ≡ | ⊃ | ∨ | ∧ | ¬

Die logischen Operatoren wurden schon im Punkt 2.1.4.1. vorgestellt, die nähere Bedeutung erfolgt nachstehend.

Die Definition der logischen Operatoren wird üblicher Weise in Form einer Tabelle eindrucksvoll dargestellt. A und B repräsentieren logische Werte oder logische Ausdrücke.

Logische Werte		Negation	Konjunktion	Disjunktion	Implikation	Äquivalenz
A	B	¬A	A ∧ B	A ∨ B	A ⊃ B	A ≡ B
true	**true**	**false**	**true**	**true**	**true**	**true**
true	**false**	**false**	**false**	**true**	**false**	**false**
false	**true**	**true**	**false**	**true**	**true**	**false**
false	**false**	**true**	**false**	**false**	**true**	**true**

Hinweis:
Mit den logischen Operatoren Negation, Konjunktion und Disjunktion ist man in der Lage, jedes beliebige aussagenlogische Konstrukt darzustellen. Die Implikation A ⊃ B lässt sich durch die Beziehung (¬ A) ∨ B bequem darstellen, während die Äquivalenz A ≡ B durch die kompliziertere Beziehung (A ∧ B) ∨((¬A) ∧ (¬B)) dargestellt werden muss. Wie später gezeigt wird, können in diesen beiden Darstellungen die Klammern völlig entfallen. Aus diesem Grund wurde bei der Entwicklung weiterer Sprachen größtenteils auf Implikation und Äquivalenz verzichtet. Stattdessen wurden in Erweiterung der Anwendung logischer Sprachelemente im Hinblick der kommerziellen Nutzung in einigen Sprachen logische Ketten kreiert, die es gestatten, diese Bit für Bit mit logischen Operatoren zu verknüpfen.

Bei der Bildung logischer Ausdrücke müssen wie bei der Bildung arithmetischer Ausdrücke Regeln beachtet werden.

1. Logische Operatoren dürfen nicht unmittelbar aufeinander folgen. Im Bedarfsfall sind Klammern zu setzen. Eine Ausnahme gilt bei der Negation; vor dem unären, einem einstelligen Operator muss immer ein zweistelliger, ein binärer Operator, eine öffnende Klammer oder kein Zeichen stehen.

2. Logische Operanden und logische Klammerausdrücke dürfen nicht unmittelbar aufeinander folgen.

Beispiele:

¬ (A ∧ B) Negation der Konjunktion
¬ (A ∨ B) Negation der Disjunktion
A< B ≡ C ∧D
A > B + X ≡ C ∨**true**
A + B > -1 ∨ X – Y< 1
P ∧ Q ∨ X ≠ Y
¬ ¬ A Fehler, logische Operatoren dürfen nicht aufeinander folgen, richtig ist ¬ (¬A)
A ⊃ B **true** Fehler, logische Operanden dürfen nicht aufeinander folgen

3.4.2. Bedingte logische Ausdrücke

Eine Erweiterung des einfachen logischen Ausdrucks ist der bedingte logische Ausdruck. Er enthält genau wie der bedingte arithmetische Ausdruck eine oder mehrere Bedingungen, durch die je nach Aussagewert einer von verschiedenen einfachen logischen Ausdrücken ausgewählt wird, der an die Stelle des gesamten logischen Ausdrucks tritt.

Syntax bedingter logischer Ausdruck:

< *bedingter logischer ausdruck* > ::=
if < *logischer ausdruck* > **then** < *einfacher logischer ausdruck* >
else < *logischer ausdruck* >

Erklärung:
Die logischen Werte hinter **if** werden entsprechend ihrer aktuellen Zuweisungen von links nach rechts durchgehend nacheinander ausgewertet, bis einer mit dem logischen Wert true gefunden wird. Der Wert des gesamten logischen Ausdrucks ist dann der Wert, der hinter then aufgelistet wurde. Ist dann der logische Wert nach der Auswertung nicht true sondern false, so ist dann der Wert des gesamten logischen Ausdrucks der Wert, der hinter else steht.

Beispiele:

if a > b **then true else** c ≥ d
if k ≤ 100 **then true else false**
if if A **then** B **else** C **then** D **else** G

Weitere Beispiele werden im Kapitel „Bedingte Anweisungen“ aufgezeigt.

3.4.3. Regeln bei der Abarbeitung arithmetischer und logischer Ausdrücke

Die Operationen in einem Ausdruck werden im Allgemeinen in einer festgelegten Reihenfolge **von links nach rechts** abgearbeitet. Die Eindeutigkeit bei der Bestimmung eines logischen Wertes ist wie bei den arithmetischen Ausdrücken durch Vorrangregeln festgelegt. Diese festgelegte Reihenfolge der Abarbeitung von Operationen innerhalb eines Ausdrucks kann immer durch passendes Setzen von Klammern beeinflusst werden.

Somit gelten bei der Erstellung arithmetischer und logischer Ausdrücke folgende **Prioritätsregeln**:

1. Klammerrechnung
2. Potenzierung ↑
3. Multiplikation oder Division * | / | ÷
4. Addition oder Subtraktion + | -
5. Vergleichsoperatoren < | ≤ | = | ≥ | > | ≠
6. Negation ¬
7. Konjunktion ∧
8. Disjunktion ∨
9. Implikation ⊃
10. Äquivalenz ≡

Zunächst werden also die in den Vergleichen vorkommenden arithmetischen Ausdrücke berechnet. Danach wird den Vergleichen je nach Aussagewert der logische Wert true oder false intern zugeordnet. Anschließend werden diese logischen Werte in der angegebenen Reihenfolge der logischen Operatoren ausgeführt. Bei Operatoren gleicher Priorität haben die weiter links stehenden in der Abarbeitung Vorrang.

Beispiele:

1. Bool1 ≡ wert1 > wert2 ⊃ (**if** wert3 > wert4 **then** Bool2 **else** wert5< wert6)
2. Bool1 ≡ Bool2 ⊃ Bool3 ∨ Bool4 ∧¬ wert1< wert2 ↑ wert3

Hinweis:
Bool... sind logische, wert... sind arithmetische Variable

Auswertung 2. Beispiel für drei Beispielwertesätze:
Die Auswertung wird entsprechend der Prioritätsregeln durch Klammern veranschaulicht:
Bool1 ≡ (Bool2 ⊃ (Bool3 ∨ (Bool4 ∧(¬ (wert1 < (wert2 ↑ wert3))))))

Bool1	Bool2	Bool3	Bool4	wert1	wert2	wert3	**Lösung**
true	**true**	**true**	**true**	5	2	3	**true**
false	**true**	**false**	**true**	36	6	2	**false**
false	**false**	**false**	**false**	-5	0	1	**false**

3.5. Zielausdrücke

Ein Zielausdruck ist eine Regel zur Ermittlung eines Sprungzieles. Ein Sprungziel einer Sprunganweisung ist entweder eine Marke bei Einfachverzweigungen oder ein Zielverteiler bei Mehrfachverzweigungen. Diese neuen ALGOL-Begriffe werden im Punkt Sprunganweisungen ausführlich dargestellt; hier wird lediglich auf das notwendige Grundwissen zum Verständnis der äußerst wichtigen Programmverzweigungen hingewiesen.

Das Rechenprinzip ist demjenigen bei arithmetische Ausdrücken völlig analog. Im allgemeinen Fall werden die booleschen Ausdrücke der **if** -Klauseln einen einfachen Zielausdruck auswählen. Wenn dieser Zielausdruck eine Marke ist, dann ist das gewünschte Resultat bereits gefunden. Ein Zielverteiler bezieht sich auf die zugehörige Verteilervereinbarung **switch**, durch den aktuellen numerischen Wert seiner Indexausdruckes wird einer der Zielausdrücke ausgewählt, die in der zugehörenden Liste der Verteilervereinbarung stehen. Der Zielausdruck wird in dieser Liste entsprechend des aktuellen numerischen Wertes durch Auszählen von links nach rechts ausgewählt. Da der so ausgewählte Zielausdruck wieder ein Zielverteiler sein kann, ist diese Auswertung offensichtlich rekursiv.

Syntax Zielausdruck:

< *marke* > :: =< *bezeichner* > |< *ganze zahl ohne vorzeichen* >
< *verteilerbezeichnung* > :: =< *bezeichner* >
< *zielverteiler* > :: =< *verteilerbezeichnung* > *[< indexausdruck >]*
< *einfacher zielausdruck* > :: = < *marke* > | < *zielverteiler* > | (< *zielausdruck* >)
< *zielausdruck* > :: = < *einfacher zielausdruck* > | **if** < *logischer Ausdruck* > **then**
< *einfacher zielausdruck* > **else** < *zielausdruck* >

Beispiele:

25	Marke
0025	Marke, Vornullen sind ohne Bedeutung, 25 und 0025 haben die gleiche Sprungadresse
marke1	Marke
ziel [n-1]	Zielverteiler

verteiler [**if** x < 0 **then** N **else** N +1]
Maximum [**if** a > b ∧ a > c **then** 3 **else if** a > b ∧ c > a **then** 2 **else if** b > c **then** 1 **else** 2]

Erklärung:
Von den drei eingelesenen Variablen a ≠ b ≠ c wird jeweils die größte gesucht und ausgedruckt. Die Sprungadresse 1 druckt die Variable b aus, die Sprungadresse 2 die Variable c und 3 druckt die Variable a aus. Das vollständige Beispiel wird in vielen Programmvarianten im Kapitel bedingte Anweisungen ausführlich dargestellt.

4. Anweisungen
4.1. Anweisungsformen

Die Einheiten einer Operation werden in ALGOL Anweisungen genannt. Sie werden entsprechend ihrer Niederschrift in ihrer Reihenfolge abgearbeitet. Der ALGOL-Compiler erstellt aus diesem Quellcode einen Objektcode, der dann nach weiterer Bearbeitung Befehl für Befehl umgesetzt wird. Diese sequentielle Ablauffolge kann von entsprechenden Anweisungen verändert werden; wir sprechen von dynamischen Einheiten, charakteristisch für algorithmische Sprachen.
Gesamtheit der Anweisungen:

- Ergibtanweisung
- leere Anweisung
- Sprunganweisung
- bedingte Anweisung
- Laufanweisung
- Verbundanweisung
- Blöcke
- Proceduranweisung

Syntax Anweisung:

< *anweisung* > ::= < *unbedingte anweisung* > | < *bedingte anweisung* > | < *laufanweisung* >
< *unbedingte anweisung* > ::= < *ergibtanweisung* > | < *sprunganweisung* > | < *leere anweisung* > |
< *prozeduranweisung* >

Alle Anweisungsformen werden im Folgenden besprochen.

4.2. Ergibtanweisungen

Ergibtanweisungen dienen dazu, den Wert eines Ausdrucks einer oder mehreren Variablen oder Prozedurbezeichnungen zuzuordnen. Die Ergibtanweisung wird durch das Ergibtzeichen := und abschließendem Semikolon hinreichend charakterisiert. Das Ergibtzeichen, zu lesen als „ergibt sich aus“ stammt von K.Zuse und ist als ein gerichtetes Gleichheitszeichen aufzufassen, das den zu errechneten linken Ausdruck mit dem rechts stehenden Resultat verbindet. Die dadurch nach beendeter Operation entstehende „Gleichung“ wird nach Zuse **Plangleichung** genannt.
Während das Gleichheitszeichen in der Mathematik den Gleichungen einen statischen Charakter verleiht, bringt das Ergibtzeichen den dynamischen Charakter der Plangleichungen zum Ausdruck. Durch Gleichungen werden mathematische Zusammenhänge beschrieben, durch Plangleichungen jedoch Abläufe und Vorgänge.

Beispiele:

a = b + c stellt eine symmetrische Relation dar (Vergleiche Gleichheitsbegriff in 2.1.4.1.2.)
a := b + c; stellt die analoge Ergibtanweisung dar; das Ergebnis b + c wird der Variablen a zugewiesen

dagegen ist

k = k + 1 mathematisch für k im Bereich der reellen Zahlen nicht definiert

während

k := k + 1; der aktuelle Wert von k wird in jedem Durchlauf um 1 erhöht. Diese iterativen Verfahren finden vor allem Einsatz in den Laufanweisungen

Neben diesen einfachen Ergibtanweisungen gibt es die erweiterte Form der mehrfachen Ergibtanweisungen, wie z.B.

```
x := y := z:= 0;
```

Syntax Ergibtanweisung:

< linke seite > :: = < variable > := | < prozedurbezeichnung > :=
< liste linke seite > ::= < linke seite > | < liste linke seite > < linke seite >
< ergibtanweisung > ::= < liste linke seite > < arithmetischer ausdruck > |
< liste linke seite > < logischer ausdruck >

Beispiele - Zuweisung einer Konstanten:

```
pi := 3.14159265;
e :=  2.71828182;
null := 0;
selektor := true;
i := i + 1;
erg := Mat [ 2,4];
```

Beispiele - Zuweisung einer arithmetischen Ergibtanweisung:

```
wert := (w1 + w2) * w3;
umfang := 2 * pi * radius;
c := Mat1 [1,1] + Mat2 [1,1];
max := if a  > b ∧ a  > c then a else if a  > b ∧ c  > a  then c else if b  > c  then b else c;
u := v := w := x + y * z;          Abarbeitung:  pseudovar := x + y * z;
                                                 w := pseudovar;
                                                 v := pseudovar;
                                                 u := pseudovar;
```

Beispiele - Zuweisung einer logischen Ergibtanweisung:

```
flag := if a < b then  true  else  false;
erg := Bool1  ≡ Bool2  ⊃  Bool3  ∨  Bool4  ∧¬ wert1 < wert2  ↑ wert3;
```

Wertzuweisung an erg für drei Beispielwertesätze:

Bool1	Bool2	Bool3	Bool4	wert1	wert2	wert3	erg
true	**true**	**true**	**true**	5	2	3	**true**
false	**true**	**false**	**true**	36	6	2	**false**
false	**false**	**false**	**false**	-5	0	1	**false**

Beispiele - Ergibtanweisung an eine indizierte Variable:

```
Mat [ i, k] := 1;
Mat [1,1] := Mat1 [1,1] + Mat2 [1,1];
Tab [ i + 1, k + 1] := 25;          Abarbeitung:  ps1 := i + 1;
                                                  ps2 := k + 1;
                                                  Tab [ps1 , ps2 ] := 25;
```

Interne Abarbeitung einer Ergibtanweisung:

1.Schritt: - Vorhandene Indexausdrücke in links stehenden Variablen werden in der Rei- - henfolge von links nach rechts berechnet und intern mit **real** vereinbarten Pseudovariablen zugewiesen
2.Schritt: - Der auf der rechten Seite stehende Ausdruck wird berechnet
3.Schritt: - und allen links stehenden Variablen zugewiesen.

4.3. Marken, markierte Anweisungen, Sprunganweisungen

Die Niederschrift in Maschinencode geschieht auf besonderen Programmierformularen, bei der jede niedergeschriebene Zeile des Programmcodes einer relativen Speicherzelle des zu programmierenden Computers entspricht. Um den sequentiellen Ablauf entsprechend der Niederschrift des Programmtextes zu verlassen, sind bedingte oder unbedingte Sprünge zu Nummern der Programmzeilen notwendig, bei der die Abarbeitung weiter erfolgen soll. Diese relativen Adressen werden bei der Übersetzung durch entsprechende Programme in die tatsächlichen absoluten Adressen zur Laufzeit umgewandelt.
In der Niederschrift von problemorientierten Programmiersprachen sind keine besonderen Programmformulare notwendig. Bei der Festlegung von Sprungadressen ist man somit in ALGOL an keine Zeilennummerierung gebunden. Bei maschinenorientierten Programmiersprachen entspricht jede niedergeschriebene Zeile einem Elementarbefehl, bei problemorientierten Programmiersprachen ist eine derartige Niederschrift nicht möglich. Für jede ALGOL-Instruktion wird von dem Compiler eine Folge von Elementarbefehlen generiert, so dass man bei der Angabe von Sprungzielen die Möglichkeit haben muss, über sog. Marken die herkömmlichen Befehlsadressen zu erreichen.

4.3.1. Marken, markierte Anweisungen

Marken dienen zur Kennzeichnung von Anweisungen und können als beliebige Bezeichnung gewählt werden. Marken werden nicht explizit vereinbart, sie sind durch den nachfolgenden Doppelpunkt als Marke hinreichend charakterisiert. Marken sind mit symbolischen Befehlsadressen vergleichbar.

Syntax Marke:

< marke > :: = < bezeichner > : | < ganze zahl ohne vorzeichen > :

Marken haben zwei wichtige Aufgaben:

- die markierten Anweisungen können abweichend von dem sequentiellem Ablauf der Niederschrift der Anweisungen erreicht werden, z.B. durch Sprunganweisungen goto
- durch Setzen von Marken mit einem semantischen Bezug kann die Lesbarkeit und die Übersichtlichkeit von Quellprogrammen stark verbessert werden

Eine Anweisung gilt als markiert, wenn vor die Anweisung eine Marke gesetzt wird. Jede Anweisung kann markiert werden. Aus ablauftechnischen Gründen kann eine Anweisung auch mehrfach markiert werden.

Syntax markierte Anweisung:

< markierte anweisung > :: < marke > : < anweisung > | < marke > : < markierte anweisung >

Hinweis:
Diese Definition gilt für bereits beschriebene als auch für alle noch folgenden Anweisungstypen.

Beispiele – markierte Anweisungen:

```
mark: wert := (w1 + w2) * w3;
Umfang: umfang := 2 * pi * radius;
25: Mat [ i, k] := 1;
mark1: mark2: mark3: wert := 0;
```

4.3.2. Leere Anweisungen

Aus ablauftechnischen Gründen ist es mitunter notwendig, an das Ende eines Programmes oder Unterprogrammes zu springen, ohne das noch Anweisungen ausgeführt werden sollen. Sie dienen im weitesten Sinn der Programmsteuerung. Diese Notwendigkeit hatte die Einführung der leeren Anweisung zur Folge. Eine leere Anweisung besteht aus einer leeren Zeichenkette, die natürlich markiert werden kann. Eine leere Anweisung erzeugt keinen Objektcode.

Syntax leere Anweisung:

< *leere anweisung* > ::= < *leere zeichenkette* >
< *leere zeichenkette* > ::=

Wie jede Anweisung muss auch die leere Anweisung mit einem Semikolon abgeschlossen werden. Ausnahmen werden später besprochen.

Beispiel – leere Anweisungen:

ende: ;

Sinnvolle Beispiele zeigen sich bei später zu besprechenden Anweisungsformen, wie bedingte Anweisungen und Laufanweisungen.

4.3.3. Die Sprunganweisung

Eine Sprunganweisung unterbricht den sequentiellen Programmablauf, allerdings geht die Sprunganweisung in ihren vielfältigen Möglichkeiten über die Leistungsfähigkeit von einfachen Sprungbefehlen hinaus.
Die einfachste Form der Sprunganweisung ist die unbedingte Sprunganweisung, vergleichbar mit einem Sprungbefehl zu einer bestimmten symbolischen Adresse. Der Folgeoperator goto mit nachfolgender Marke bewirkt den unbedingten Sprung zum Adressteil, der durch die Marke definiert wird. Die Marke kann durch einen Zielausdruck ersetzt werden, die Mehrfachverzweigungen gestatten. Ein Zielausdruck ist eine Regel zur Ermittlung einer Marke. Der Zielausdruck kann eine Marke (siehe 4.3.1.)., ein Zielverteiler oder ein bedingter Zielausdruck sein (siehe 3.5. Zielverteiler).

Syntax Sprunganweisung:

< *sprunganweisung* > ::= **goto** < *zielausdruck* >
< *zielausdruck* > ::= < *einfacher zielausdruck* > | < *wenn-klausel* > < *einfacher zielausdruck* >
else < *zielausdruck* >
< *wenn-klausel* > ::= **if** < *logischer ausdruck* > **then**
< *einfacher zielausdruck* > ::= < *marke* > | < *zielverteiler* > | < *zielausdruck* >

Beispiele Sprunganweisungen:

goto marke:
goto zielverteiler;
goto if a < b **then** ziel1 **else** ziel2;

ziel1 und ziel2 sind Zielausdrücke. Wenn a < b wahr ist, wird zur Marke ziel1 gesprungen, in allen anderen Fällen zur Marke ziel2.
Bevor ein Zielverteiler als Zielausdruck in einer Sprunganweisung benutzt werden kann, muss eine Verteilervereinbarung vorausgegangen sein, die diesen Zielverteiler definiert.

Eine Verteilervereinbarung hat die Form

switch vt := za1, za2,..., zan ;

wobei der Bezeichner vt die Verteilerbezeichnung und die Bezeichner zan die Zielausdrücke sind. Die nach dem Ergibtzeichen angegebenen Zielausdrücke nennt man in ihrer Gesamtheit die Verteilerliste. Jedem Zielausdruck in dieser Verteilerliste wird eine natürliche Zahl von links nach rechts in ihrer Niederschrift zugeordnet, beginnend mit der natürlichen Zahl 1. Demzufolge wird in der Anweisung

goto vt [i] für i=2

ein unbedingter Sprung zu dem Zielausdruck za2 ausgeführt.

Syntax Verteilervereinbarung:

< *verteilerbezeichnung* > ::= **switch** < *verteilerbezeichnung* > := < *verteilerliste* >
< *verteilerbezeichnung* > ::= < *bezeichner* >

Beispiele Verteilervereinbarung:

switch vert := mark, vert[1], vert[2];
goto vert[3];

Bevor der Sprung in dem geschachtelten Zielverteiler zur Marke mark ausgeführt werden kann, werden zuerst die beiden letzten Zielausdrücke in der Verteilerliste abgearbeitet.

Ein vollständiges Beispiel wird im Beispiel 5 im Punkt 4.4. aufgezeigt.

4.4. Bedingte Anweisungen

Diese Anweisungen werden benutzt, wenn die Abarbeitung der Anweisung von Bedingungen abhängig gemacht werden soll. Sie werden auch zur Programmierung von zyklischen Abläufen herangezogen. Man unterscheidet die unvollständige und die vollständige Form der bedingten Anweisung, die auch verschachtelt sein können (siehe Beispiel).

Syntax bedingte Anweisung:

< *bedingte anweisung* > ::= **if** < *logischer ausdruck* > **then** < *unbedingte anweisung* >|
if < *logischer ausdruck* > **then** < *unbedingte anweisung* >
else < *anweisung* >

Bedingte Anweisungen bewirken, dass bestimmte Aussagen, die ausgeführt oder übersprungen werden, je nach aktuellem Wert der angegebenen logischen Ausdrücke. Die unbedingte Anweisung einer **if** -Anweisung wird ausgeführt, wenn der logische Ausdruck der **if** -Klausel wahr ist. Sonst wird sie übersprungen und die Operation wird mit der nächsten Anweisung fortgesetzt.

Die Regel zur Bildung und Bearbeitung bedingter Ausdrücke sind analog den Regeln und Bearbeitung bedingter Ausdrücke (siehe ab 3.3.3.) und werden deshalb nicht weiter ausgeführt.

Beispiele – unvollständige Form der bedingten Anweisung:

if w3 ≠ 0 **then** w1 := w2 / w3; ausschließen Division durch Null
if a > b **then** c := a – b;
if n = 100 **then goto** ende;

Beispiele – vollständige Form der bedingten Anweisung:

```
if n < 100 then n := n + 1 else goto drucke;
if a ≥ 0 then erg := a else erg := -a;    absoluter Wert von a für erg
if a > 0 then erg := 1 else if a = 0 then erg := 0 else erg := -1;    Signum für erg (siehe 3.2)
```

1. **Komplexes Beispiel:** Drei Zahlen werden eingelesen und die größte ermittelt und ausgedruckt mit Sprunganweisungen:

```
begin real a, b, c, max;
      read (a, b, c);      comment symbolische Leseanweisung mit a ≠ b ≠ c;
      if a > b then goto m1;
      if b > c then max := b else max := c; goto m2;
m1: if a > c then max := a else max := c;
m2: print(max);          comment symbolische Druckanweisung
end
```

2. **Komplexes Beispiel**: Drei Zahlen werden eingelesen und die größte ermittelt und ausgedruckt mit logischen Operatoren und ohne Sprunganweisungen:

```
begin real a, b, c, max;
      read (a, b, c);      comment symbolische Leseanweisung mit a ≠ b ≠ c;
      if a > b ∧ a > c then max := a else if a > b ∧ c > a then max := c else
                                                  if b > c then := b else max := c;
      print(max);      comment symbolische Druckanweisung
end
```

3. **Komplexes Beispiel**: Drei Zahlen werden eingelesen und die größte ermittelt und ausgedruckt mit bedingten Ausdrücken und logischen Operatoren:

```
begin real a, b, c, max;
      read (a, b, c);      comment symbolische Leseanweisung mit a ≠ b ≠ c;
      max := if a > b ∧ a > c then a else if a > b ∧ c > a then c else if b > c then b else c;
      print(max);      comment symbolische Druckanweisung
end
```

4. **Komplexes Beispiel**: Drei Zahlen werden eingelesen und die größte ermittelt und ausgedruckt mit bedingtem Zielausdruck und logischen Operatoren:

```
begin real a, b, c, max;
      read (a, b, c);      comment symbolische Leseanweisung mit a ≠ b ≠ c;
      goto if a > b ∧ a > c then m3 else if a > b ∧ c > a then m2 else if b > c then m1 else m2;
   m1: max := b; goto m4;
   m2: max := c; goto m4;
m3: max := a;
   m4: print(max);      comment symbolische Druckanweisung
end
```

5. **Komplexes Beispiel**: Drei Zahlen werden eingelesen und die größte ermittelt und ausgedruckt mit Verteiler und logischen Operatoren:

```
begin real a, b, c, max;
      switch vert := m1, m2, m3;
      read (a, b, c);      comment symbolische Leseanweisung mit a ≠ b ≠ c;
      goto vert [ if a > b ∧ a > c then 3 else if a > b ∧ c > a then 2 else if b > c then 1 else 2 ];
   m1: max := b; goto m4;
   m2: max := c; goto m4;
   m3: max := a;
   m4: print(max);      comment symbolische Druckanweisung
end
```

Bemerkung: Die sicher elegantesten Lösungen sind Beispiele 2 und 3, da hier der ungeschriebenen Regel „goto freies Programmieren“ am ehesten entsprochen wird.

4.5. Laufanweisungen

4.5.1. Allgemeine Ausführungen

Soll die Folge von Anweisungen mehrmals durchlaufen werden, bietet sich die sog. Laufanweisung an. Diese Folge von Anweisungen, auch Programmschleifen genannt, stellen sicher in jedem Programm einen wesentlichen Bestandteil dar. Man unterscheidet in der Charakteristik der Schleifensteuerung die Induktionsschleifen und die Iterationsschleifen. Die Induktionsschleifen sind dadurch gekennzeichnet, dass bei jedem Durchlauf eine neue Gruppe von Variablen in die Anweisungsfolge eingeht, während bei den Iterationsschleifen immer die gleiche Gruppe von Variablen in die Anweisungsfolge eingeht.

Syntax Laufanweisung:

< laufanweisung >::= [< marke >] **for** *< laufvariable > := < folge von lauflistenelementen >*
do *< anweisung >;*
< laufvariable > ::= < einfache variable > | < indizierte variable >
< lauflistenelement > ::= < arithmetischer ausdruck >
| *< aritmetischer ausdruck >* **step** *< arithmetischer ausdruck >* **until** *< arithmetischer ausdruck >*
| *< aritmetischer ausdruck >* **while** *< logischer ausdruck >*

Die Lauflistenelemente, die untereinander kombiniert werden, können folgendermaßen dargestellt werden:

for laufvar := L1,L2,...,Ln **do** anweis1; anweisn;

Der Anweisung „anweis1“ werden alle Werte, die die Laufvariable „laufvar“ bei der Auswertung der Lauflistenelemente „L1“ bis „Ln“ erhält, der Reihe nach zugewiesen, bis die Laufliste erschöpft ist. Jedes Lauflistenelement kann einen der folgenden drei Formen annehmen:

1. der arithmetischen Ausdruck
2. das step-until Element
3. das while Element

Die interne Abarbeitung einer Laufanweisung lässt sich wie folgt darstellen:

Anfangswert: Der Laufvariablen wird ihr erster bzw. nächster Wert aus der Laufliste zugewiesen.

Ausführung: Es wird geprüft, ob eine Abarbeitung des Anweisungsteils hinter **do** möglich ist

Fallunterscheidung: 1. die Laufliste ist erschöpft, d.h. „L1“ bis „Ln“ sind abgearbeitet
2. der numerische Wert hinter **until** ist über- bzw. unterschritten
3. der logische Wert hinter **while** ist **false**

Ist eine der drei Bedingungen erfüllt, so wird die der Anweisung „anweis1“ folgende Anweisung „anweisn“ abgearbeitet. In allen anderen Fällen wird die Anweisung „anweis1“ ausgeführt und die Laufvariable wird um den Wert modifiziert, die in der Laufliste angegeben ist

Die Ausführung einer Laufanweisung kann auf zweierlei Weise beendet werden:

1. dadurch, dass die Laufliste erschöpft ist, das letzte Element der Laufliste der verschiedenen drei Formen ist abgearbeitet
2. dadurch, dass die Laufanweisung vorzeitig mit Hilfe einer im Anweisungsteil der Laufanweisung vorkommenden Sprunganweisung **goto** verlassen wird

Man kann also aus einer Laufliste heraus springen, aber mit einer Sprunganweisung, die außerhalb einer Laufanweisung steht, sich aber auf eine Marke innerhalb der Laufanweisung bezieht, ist verboten. Die Laufvariable behält beim vorzeitigen Verlassen mit einer Sprunganweisung ihren aktuellen Wert, während sie bei Erschöpfung der Laufliste nach dem ALGOL-Bericht nicht definiert ist.

4.5.2. Formen der Lauflisten

1. Form des Laufliste – arithmetischer Ausdruck als Lauflistenelement
Beispiel:

```
begin integer array vek [ 1 : 10 ];
    integer i;
    for i := 1, 2, 3, 4, 5, 6 ,7, 8, 9, 10 do vek[ i ]  := 1;
    comment  Alle Elemente des Vektors werden auf 1 gesetzt;
    print (vek)
end
```

Die Laufvariable i nimmt der Reihe nach die durch Komma getrennten Werte an, die durch die Ausdrücke bestimmt sind. Ist die Laufliste erschöpft, wird die der Laufanweisung folgende Anweisung hinter **do** abgearbeitet; der Vektor wird ausgedruckt.

2. Form des Laufliste – step-until als Lauflistenelement
Beispiel:

```
begin integer array vek [ 1 : 10 ];
        integer i;
        for i := 1 step 1 until 10 do vek[ i ]  := 1;
        comment  Alle Elemente des Vektors werden auf 1 gesetzt;
        print (vek)
end
```

Die Laufvariable i nimmt der Reihe nach die Werte an, die in der Laufliste spezifiziert sind.
for i := 1 ist der Anfangswert der Laufvariablen,
step 1 ist die Schrittweite, d.h. Erhöhung des Wertes der Laufvariablen um den Wert 1,
until 10 ist der Endwert der Laufvariablen
do vek[i] := 1 ist der Anweisungsteil
Ist die Laufliste erschöpft, wird die der Laufanweisung folgende Anweisung abgearbeitet. Die Laufvariable hat nach Verlassen der Laufanweisung einen um die Schrittweite erhöhten Endwert.

3. Form des Laufliste – while-Element als Lauflistenelement
Beispiel:

```
begin integer array vek [ 1 : 10 ];
        integer i;
        i := 0;
        for i := i +1 while i ≤ 10 do vek [ i ]  := 1;
        comment  Alle Elemente des Vektors werden auf 1 gesetzt;
        print (vek)
end
```

Die Laufvariable i nimmt der Reihe nach die Werte an, die in der Laufliste spezifiziert sind.
for i := i +1 ist der Anfangswert der Laufvariablen, durch Iteration i + 1 wird der Wert von i um 1 erhöht (Schrittweite)
while i ≤ 10 die Anweisung wird solange ausgeführt, solange der logische Ausdruck wahr ist

Ist die Laufliste erschöpft, wird die der Laufanweisung folgende Anweisung abgearbeitet. Die Bedingung wird **vor** jedem Schleifendurchgang neu berechnet. Auch hier hat die Laufvariable nach Verlassen der Laufanweisung einen um die Schrittweite erhöhten Endwert, da ja erst jetzt die Bedingung i ≤ 10 falsch ist, die das Verlassen der Laufanweisung veranlasst, aber nach dem ALGOL-Bericht ist Endwert nicht definiert.

4. Form des Laufliste – Kombinationen der Lauflistenelemente
Beispiel:

```
begin integer array vek [ 1 : 10 ];
        integer i;
        for i := 1 , 2  step 1 while i ≤ 10 do vek [ i ]  := 1;
        comment  Alle Elemente des Vektors werden auf 1 gesetzt;
        print (vek)
end
```

Selbstverständlich sind Kombinationen aller Spezifikationen möglich. Ebenso ist eine Verschachtelung für Bereiche mit mehreren Indexgrenzenpaaren möglich. Grundsätzlich werden verschachtelte Anweisungen von Innen nach Außen abgearbeitet.

4.5.3. Abschließende Beispiele

(1) Bestimmte Elemente des Vektors sollen auf 0 gesetzt werden

```
integer i;
        for i := 1, 10, 50, 100 do vek[ i ]  := 0;
        comment  Bestimmte Elemente des Vektors werden auf 0 setzen;
        print (vek)
end
```

(2) Die Elemente eines Feldes sollen mit den Werten ihrer Indizes gefüllt werden.

```
begin integer array vek [ 1 : 1000 ];
    integer i ;
    for i := 1 step 1 until 1000 do vek[ i ]  := i;
    comment  Alle Elemente des Vektors werden auf aktuelles i gesetzt
    print (vek)
end
```

(3) Negative Zahlen eines Bereichs sollen ermittelt werden, untereinander ausgedruckt und durch ihre positiven Werte ersetzt werden:

```
begin real array vek [ -500 : 500 ];
      integer i ;
      for i := -500 step 1 until 500 do
                if vek [ i ] < 0 then begin
                                          print (i);
                                          print (vek [ i ]);
                                          vek [ i ] := vek [ i ] * (-1);
                                       end
end
```

Bemerkung:
Die Anweisungen **begin** und **end** hinter **then** nennt man Verbundanweisung; sie wird hier aus programmtechnischen Gründen benötigt. Da nach Definition hinter **do** genau eine Anweisung abgearbeitet wird, ist das in diesem Fall hier die Verbundanweisung. Sie wird im nächsten Kapitel besprochen.

(4) Die folgende Summe soll gelöst werden:

$$\text{Sum} = \sum_{\substack{k=-r \\ k \neq \pm 1}}^{r} \frac{1}{k^2 - 1}$$

```
begin real summe;
      integer k; r;
      summe :=0;          comment  Anfangswert setzen;
      lesen: read (r);    comment  Eingabe der Anzahl der Summanden;
                          comment  In der Laufanweisung Division durch 0 verhindern;
            for k := -r , 2  step 1 until -2, 0, 2 step 1 until r do  summe := summe + 1 / ( k ↑ 2 -1) ;
      print (summe);
      goto lesen
end
```

Laufanweisungen können beliebig verschachtelt werden. Die Berechnung von mehrdimensionalen Bereichen, wie z.B. Matrizen, wird in den folgenden Beispielen dargestellt. Die Matrizen werden zeilenweise abgearbeitet, sie werden von Innen nach Außen heraus aufgelöst. Bereiche werden grundsätzlich lexikografisch abgearbeitet.

(5) Es soll die Summe aller Werte eines zweidimensionalen Bereichs - einer Matrix - berechnet werden.

```
begin real array mat [ 1 : 3, 1 : 4 ];
      integer i, k;
      real summe;
      summe := 0;
      read (mat);       comment 3 Zeilen und 4 Spalten der Matrix  werden eingelesen;
      for i := 1 step 1 until 3 do
          for k := 1 step 1 until 4 do  summe := summe + mat [ i, k ];
      print (summe);      comment Druck der Summe aller 12 Werte
end
```

Die Abarbeitung der Indizes erfolgt lexikografisch; d.h.

```
MAT(1,1)  MAT(1,2)  MAT(1,3)  MAT(1,4)
MAT(2,1)  MAT(2,2)  MAT(2,3)  MAT(2,4)
MAT(3,1)  MAT(3,2)  MAT(3,3)  MAT(3,4)
```

(6) In einer Matrix sind alle Spaltensummen zu bilden und zu speichern.

```
begin real array mat [ 1 : 3, 1 : 4 ];
      real array spaltsum [ 1 : 4 ];
      integer i, k;
      real spaltsum, tempsum;
      tempsum := 0;
      read (mat);              comment 3 Zeilen und 4 Spalten der Matrix  werden eingelesen;
      for i := 1 step 1 until 3 do
         for k := 1 step 1 until 4 do  tempsum := tempsume + mat [ i , k ];
          spaltsum [ k ] := tempsum;        comment Abarbeitung von innen nach aussen;
      print (spaltsum);                     comment Druck der vier Spaltensummen
end
```

(7) Matrizenmultiplikation - Zwei Matrizen sollen nach mathematischen Regeln multiplikativ verknüpft werden, d.h. die Zeilen der 1. Matrix werden mit den Spalten der 2. Matrix multipliziert, d.h. wiederum, die Zeilenzahl der 1. Matrix muss mit der Spaltenzahl der 2. Matrix übereinstimmen; sie müssen verkettet sein.

$$C_{i,k} = \sum_{j=1}^{n} a_{i,j} * b_{j,k}$$

```
begin real array A[ 1 : 3, 1 : 4 ], B [ 1 : 4, 1 : 3], C [ 1 : 3, 1 : 4];
      integer i, k, j;
      real summe;
      summe := 0;
      read (A, B);                          comment Beide Matrizen werden eingelesen;
      for i := 1 step 1 until 3 do
         for k := 1 step 1 until 4 do
                  begin  summe := 0;
                         for j := 1 step 1 until 4 do summe := summe + A [ i, j ] * B [ j , k ];
                         C [ i, k ] := summe;
                  end
     print ( C )                            comment Ausgabe der Ergebnismatrix C
end
```

Bemerkung:
Die sog. Verbundanweisung mit **begin** und **end** verbindet mehrere Anweisungen zu einer syntaktischen Einheit. Der Begriff Verbundanweisung wird nachfolgend erklärt.

4.6. Verbundanweisungen

4.6.1. Allgemeine Ausführungen

Die Syntax einer Programmiersprache setzt gewisse Regeln voraus, die unbedingt einzuhalten sind. So ist z.B. festgelegt, dass hinter der **do** Klausel in einer Laufanweisung genau eine Anweisung zyklisch abgearbeitet wird, solange die Auswertung der Endebedingung oder die Erschöpfung der Laufliste einen weiteren Durchlauf generiert. Müssen nun aus Gründen der Organisation des zugehörigen Algorithmus mehrere Anweisungen hinter der **do** Klausel ausgeführt werden, so werden diese Anweisungen durch Einschließen in die Anweisungsklammern **begin** und **end** zu einer Verbundanweisung verbunden. Auch aus Gründen der Übersichtlichkeit eines Quelltextes können mehrere Anweisungen in einer Verbundanweisung „verpackt" werden, eine sinnvolle Variante, die strukturierte Programmierung anzuwenden. Das Beispiel **(7)** im Punkt 4.5.3. Matrizenmultiplikation zeigt sehr deutlich die Notwendigkeit des Einsatzes einer „verbundenen Anweisung".

Syntax Verbundanweisung:

< *verbundanweisung* >::= [< *marke* >] **begin** < *folge von anweisungen* > **end**

Verbundanweisungen können also markiert sein und sind demnach Ziele von Sprunganweisungen. Da die Klausel **end** das Ende einer Verbundanweisung anzeigt, ist sie ebenso wie das Semikolon und die Klausel **else** Abschluss einer Anweisung. Folgerichtig wird eine Anweisung vor einer **end** oder **else** Klausel nicht durch ein weiteres Semikolon abgeschlossen. In den vorangehenden Beispielen wurde von dieser Regel ständig Gebrauch gemacht.

4.6.2. Beispiele – Verbundanweisungen

(1) Veranschauliches triviales Beispiel

```
if a < b  then  begin   anweisung_1;                    comment  Verbundanweisung _1_2;
                        anweisung_2
                end
        else  if  d < e  then  begin   anweisung_3;     comment  Verbundanweisung _3_4;
                                       anweisung_4
                               end
                      else   for i := 1 step 1 until 100 do anweisung_5;
anweisung_6;
```

Abarbeitung:

1. Wenn a < b, dann wird die Verbundanweisung_1_2 abgearbeitet; der zugehörige **else** Zweig wird nicht abgearbeitet und anschließend die Anweisung anweisung_6 ausgeführt
2. Sonst (a >= b) wird die Verbundanweisung_1_2 übersprungen, der **else** Zweig wird abgearbeitet

2a. Wenn d < e, dann wird die Verbundanweisung_3_4 abgearbeitet; anschließend die Anweisung anweisung_6 ausgeführt

2b. Sonst (d >= e) wird die Verbundanweisung_3_4 übersprungen, der **else** Zweig mit der Laufanweisung anweisung_5 abgearbeitet; anschließend die Anweisung anweisung_6 ausgeführt

(2) In einer Matrix ist der betragsgrößte Wert zu suchen und mit seinem Index auszudrucken.

```
begin real array bereich [ 1 : 100, 1 : 200 ];
      integer i, k, imax, kmax;
      real max;
      imax := kmax := 1;
      max := 0;
      read (bereich);      comment 100 Zeilen und 200 Spalten der Matrix  werden eingelesen;
      for i := 1 step 1 until 100 do
         for k := 1 step 1 until 200 do
                  if  abs (bereich [ i, k ] )  > max then begin;
                                                      max := abs (bereich [ i, k ] );
                                                      kmax := i; kmax := k;
                                                      end
      print (max, i, k);         comment Druck des betraggroessten Wertes des Bereichs
                                          Druck des Zeilenindexes;
                                          Druck des Spaltenindexes
         end
```

(3) Von 5000 reellen Werten, die in einen Bereich eingelesen werden, ist der Wert der größten zu suchen und mit ihrem Index auszudrucken.

```
begin real array bereich [ 1 : 5000 ];
      integer ind, imax;
      real max;
      read (bereich);          comment 5000 reelle Zahlen werden eingelesen;
      max := bereich [ 1 ];
      for ind := 1 step 1 until 5000 do
                  if bereich [ ind ]   > max then begin;
                                                max := bereich [ ind ];
                                                imax := ind;
                                             end
      print  (max);              comment Druck des groessten Wertes des Bereichs;
      print  (imax);           comment Druck des Indexes des groessten Wertes des Bereichs
end
```

(4) Die erste negative Zahl eines Bereichs soll ermittelt und ausgedruckt werden:

```
begin real array bereich [ -500 : 500 ];
      integer i ;
      boolean stopbit;
      stopbit := true;
      for i := -500 step 1 while stopbit do if bereich [ i ] < 0 then begin
                                                                        print (vek [ i ]);
                                                                        stopbit := false
                                                                     end
end
```

Die logische Variable stopbit, mit dem anfänglichen Wert **true** sorgt für den Fall des Zutreffens einer negativen Zahl in dem Bereich mit dem Wert **false** für das Verlassen der Laufanweisung.

(5) Die erste negative Zahl eines Bereichs soll ermittelt und ausgedruckt werden. Danach ist die Laufanweisung sofort zu verlassen.

```
begin real array bereich [ -500 : 500 ];
        integer i ;
        for i := -500 step 1 until 500  do if bereich [ i ] < 0 then begin
                                                                        print (vek [ i ]);
                                                                        goto ende;
                                                                end
ende:
end
```

(6) Hornerschema - Es ist der Wert eines Polynoms n-ten Grades zu berechnen

$$P(x) = \sum_{i=0}^{n} a_i \, X^i$$

an einer Stelle $x = x_i$. Die Werte für n, x_0 und die Koeffizienten a_0, $a_{1,...,} a_n$ sind über das Eingabemedium einzulesen. Das Programm soll für wechselnde a_i und x_i für verschiedene n benutzt werden, ohne dass das Programm neu übersetzt werden muss.

```
begin integer n;
   lies: read (n);          comment  n wird eingelesen, um einen dynamischen Bereich a zu definieren;
    begin real P, x0;
            integer i;
            real array  a [ 0 : n ];        comment  Der numerische Wert der oberen Grenze n wird im
                                                     übergeordneten Block  eingelesen;
            read ( x0, a);
            P := a [ n ];
            for i := n - 1 step -1 until 0 do   P := P * x0 + a [ i ];
            print (P)
    end
goto lies
end
```

Bemerkung:
Im Unterschied zu den Verbundanweisungen erlaubt die sog. Blockstruktur mit **begin** und **end** ein dynamisches Verarbeiten von Bereichen, die Blöcke werden vor allem dazu benutzt, Felder mit variablen Grenzen im Stapelspeicher anzulegen. Die Blockstruktur wird im nächsten Kapitel besprochen.

4.7. Blöcke

4.7.1. Allgemeine Ausführungen

Ein Block besteht aus einer von einer **begin** - und einer **end -** Klausel begrenzten Folge von Vereinbarungen und Anweisungen. Streng genommen ist jedes einfaches ALGOL-Programm ein Block, bestehend aus Vereinbarungen und Anweisungen.

Syntax Block:

< block >::= [< marke >] **begin**
[< folge von vereinbarungen >]
< folge von anweisungen >
end *[< markenpräfix >];*

Allgemeines Beispiel:

```
Block: begin
          folge von vereinbarungen;
          folge von anweisungen;
          blöcke;
       end Block
```

Im Unterschied zu den Verbundanweisungen können Blöcke Vereinbarungen enthalten, die ineinander geschachtelt die Programmierung der wichtigen dynamischen Speicherplatzverwaltung mit variablen Grenzen in Bereichen ermöglicht. Im übergeordneten Block können in den untergeordneten Block Variable übergeben werden, die die Variabilität der Bereichsgrenzen ermöglicht. Der Speicherbedarf wird also während der Abarbeitung festgelegt und die Daten werden im sog. Stapelspeicher abgelegt. Diese Prozedur kann sich bei jedem Eintritt in den Block wieder verändern. Dies ist der Sinn und der Inhalt der **dynamischen Speicherverwaltung**, ein Grundprinzip moderner Compiler. Beim Aktivieren und Beenden eines Blockes müssen also bestimmte organisatorische Maßnahmen getroffen werden. Sie werden beim Aktivieren als Prolog und beim Verlassen als Epilog bezeichnet.

Die wichtigsten Funktionen des Prologs:

- Festlegung von Dimensionsgrenzen und deren Initialisierung
- Berechnung von Anfangswerten der Bereichsgrenzen
- Zuordnung von Speicherplatz für alle in diesem Block vereinbarte Variablen
- Zuordnung von Speicherplatz für Zwischenergebnisse, für Pseudovariable

Die wichtigsten Funktionen des Epilogs:

- Wiederherstellung des Zustandes, der vor der Blockaktivierung existierte
- Freigabe des Speicherplatzes, der durch den Prolog zugeordnet wurde.

Veranschauliches triviales Beispiel:

```
if a < b  then  begin  anweisung_1;                    comment  Verbundanweisung _1_2;
                       anweisung_2
                end
          else  if  d < e  then  begin   vereinbarung_1:
                                         anweisung_3;          comment  Block;
                                         anweisung_4
                                 end
                        else   for i := 1 step 1 until 100 do anweisung_5;
anweisung_6;
```

Abarbeitung:

1. Wenn a < b, dann wird die Verbundanweisung_1_2 abgearbeitet; der zugehörige **else** Zweig wird nicht abgearbeitet und anschließend die Anweisung anweisung_6 ausgeführt
2. Sonst (a >= b) wird die Verbundanweisung_1_2 übersprungen, der **else** Zweig wird abgearbeitet

2a. Wenn d < e, dann wird der Block abgearbeitet; die Vereinbarungen werden initialisiert, die Anweisungen anweisung_3 und anweisung_4 werden ausgeführt. Beim Verlassen des Blockes wird der durch die Vereinbarung vereinbarung_1 reservierte Speicherplatz wieder freigegeben und die Anweisung anweisung_6 wird ausgeführt

2b. Sonst (d >= e) wird der Block übersprungen, der **else** Zweig mit der Laufanweisung anweisung_5 abgearbeitet; anschließend wird die Anweisung anweisung_6 ausgeführt

4.7.2. Gültigkeit von Bezeichner

Bei Blöcken gilt grundsätzlich das Prinzip, das die Gültigkeit einer Variablen oder eines Bereiches nicht nach außen erweitert werden kann; d.h., dass Bezeichner innerhalb eines Blockes **lokal** in diesem Block als auch in allen weiteren inneren Blöcken sind, während diese Bezeichner in den übergeordneten Blöcken nicht definiert sind. Bezeichner, die in einem Block auftreten, aber nicht in ihm vereinbart wurden, sind bezüglich dieses Blockes **nichtlokal**, d.h. sie sind außerhalb dieses Blockes erreichbar. Beim Austritt aus dem Block gehen alle Werte der darin vereinbarten Variablen verloren.

Beispiel (6) Block Hornerschema im Punkt 4.6.2.

```
...
   begin real P, x0;
            integer i;
            real array  a [ 0 : n ];        comment  Der numerische Wert der oberen Grenze n wird im
                                                     übergeordneten Block  eingelesen;
            read ( x0, a);
            P := a [ n ];
            for i := n - 1 step -1 until 0 do   P := P * x0 + a [ i ];
            print (P)
      end
...
```

Die Bezeichner P, x0, i und der Bereich a sind lokale Bezeichner, der Bezeichner n ist nichtlokal bezüglich dieses Blockes.

Das Vereinbarungszeichen **own** veranlasst, dass die Wirkung der Vereinbarungen beim Verlassen eines Blockes nur teilweise aufgehoben wird. Die **own real** Variable z.B. geht beim Austritt aus dem Block nur namensmäßig verloren. Das Vereinbarungszeichen **own** bewirkt, dass der Wert der **own** Variablen wiederauffindbar gespeichert wird. Allerdings ist **own** bei den meisten ALGOL-Compiler Herstellern sehr umstritten und es wurde auf deren Anwendung verzichtet.

Ein Block wird aktiviert, wenn seine **begin** Klausel im Programmablauf erreicht wird. Das kann durch den sequentiellen Ablauf als auch durch eine **goto** Anweisung erreicht werden.

Ein Block wird normal beendet, wenn die Programmablaufsteuerung auf die **end** Klausel trifft. Auch Möglichkeiten, Blöcke bei der Benutzung von internen Blöcken durch die Aufführungen von **goto** Anweisungen anormal zu beenden, bestehen.

Man darf niemals in einen Block hineinspringen. Da der Gültigkeitsbereich der Marke, also das Sprungziel der **goto** Anweisung, der Block ist , in dem sie definiert ist, kann sie also von „Au-

ßen“ nicht angesprungen werden, da sie bezüglich des Blockes lokal ist. Es darf von außen nur die markierte **begin** Klausel angesprungen werden.

Beispiel - Transponierung einer quadratischen Matrix:

Die Transponierte einer m x n Matrix A = $(a_{i,k})$ ist die Matrix A_{trans} = $(a_{i,k})$. Die Matrix wird sozusagen um die Hauptdiagonale $a_{11}, a_{22},...,a_{nn}$ gespiegelt.

```
begin
    integer m;
    Lies: read ( m );                          comment Eingabe Indexgrenze;
    if m < 2 v m  > 100 then goto Lies;
    begin                                      comment auesserer Block;
       real array A [ 1 : m, 1 : m ];          comment Vereinbarung dyn. Bereich;
       read ( A );                             comment Eingabe quadratische Matrix;
       print ( A );                            comment Druck der eingelesenen Matrix;
   Trans: begin                                comment innerer Block;
             integer i,k;
             real hv;                          comment Die Variablen i, k, hv sind lokal;
             for i := 2 step 1 until m  do     comment Indexgrenze m nichtlokal;
                for k := 1 step 1 until i – 1 do
                   begin                       comment Verbundanweisung;
                      hv := A[ i, k ];
                      A[ i, k ]  := A[ k, i ]; comment transponieren;
                      A[ k, i ]  := hv
                    end                        comment Ende Verbundanweisung;
             print ( A );                      comment Druck der transponierten Matrix;
       end Trans                               comment Ende interner Block;
    end                                        comment Ende auesserer Block;
end                                            comment Ende ALGOL Programm
```

Ablaufbeispiel:

Eingabe: m = 3

$$A = \begin{matrix} 1.5 & 2.3 & 3.6 \\ 4.0 & 5.1 & 6.8 \\ 7.4 & 8.3 & 9.2 \end{matrix}$$

Wertzuweisungen nach jedem Durchlauf des internen Blockes:

Durchläufe	i	k	hv := A[i, k]	A[i, k] := A[k, i]	A[k, i] := hv
1. Durchlauf	2	1	4.0	2.3	4.0
2. Durchlauf	3	1	7.4	3.6	7.4
3. Durchlauf	3	2	8.3	6.8	8.3

Ergebnis:

$$A_{trans} = \begin{matrix} 1.5 & 4.0 & 7.4 \\ 2.3 & 5.1 & 8.3 \\ 3.6 & 6.8 & 9.2 \end{matrix}$$

5. Unterprogrammtechnik

Die Unterprogrammtechnik nimmt in der Programmierung eine äußerst wichtige Stellung ein. Sollen Formeln oder Algorithmen ohne Kenntnis der Variablen programmiert werden, auf die sie später angewendet werden, so bietet ALGOL mit seiner eleganten Prozedurkonzeption eine äußerst moderne Lösung an. Die Prozeduren stellen den wichtigsten Bestandteil der Sprache ALGOL dar. Dem Unterprogramm selbst entspricht dabei die **Prozedurvereinbarung** und dem Unterprogrammaufruf, die **Prozeduranweisung**. Prozeduren sind im Gegensatz zu den Blöcken nicht sequentiell erreichbar, sie werden über die Prozeduranweisung aktiviert und springen nach Abarbeitung hinter die Aufrufstelle zurück.

Syntax Prozeduranweisung:

< *aktueller parameter* > ::= < *kette* > | < *ausdruck* > | < *feldbezeichner* > | < *verteilerbezeichner* | < *prozedurbezeichnung* >
< *buchstabenkette* > ::= < *buchstabe* > | < *buchstabenkette* > < *buchstabe* >
< *parameterbegrenzer* > ::= , |) < *buchstabenkette* > : (
< *aktuelle parameterliste* > :: = < *aktueller parameter* > | < *aktuelle parameterliste* > < *parameterbegrenzer* > < *aktueller parameter* >
< *aktueller parameterteil* > :: = < *leere kette* > | (< *aktuelle parameterliste* >)
< *prozeduranweisung* > ::= < *prozedurbezeichnung* > < *aktueller parameterteil* >

Syntax Prozedurvereinbarung:

< *formaler parameter* > :: = < *bezeichner* >
< *formale parameterliste* > :: = < *formaler parameter* > | < *formale parameterliste* > < *parameterbegrenzer* > < *formaler parameter* >
< *formaler parameterteil* > :: = < *leere kette* > | (< *formale parameterliste* >)
< *bezeichnungenliste* > :: = < *bezeichnung* > | < *bezeichnungenliste* >, < *bezeichnung* >
< *werteteil* > :: = **value** < *bezeichnungenliste* > ; | < *leere kette* >
< *vereinbarungszeichen* > :: = **string** | < *type* > | **array** | < *type* > **array** | **label** | **switch** | **procedure** | < *type* > **procedure**
< type > ::= **real** | **integer** | **boolean**
< *vereinbarungsteil* > :: = < *leere kette* > | < *vereinbarungszeichen* > < *bezeichnungenliste* > ; | < *vereinbarungsteil* > < *vereinbarungszeichen* > < *bezeichnungenliste* >
< *prozedurkopf* > :: = < *prozedurbezeichnung* > < *formaler parameterteil* > ; < *werteteil* > < *vereinbarungsteil* >
< *prozedurhauptteil* > :: = < *anweisung* > | < **code** >
< *prozedurvereinbarung* > :: = **procedure** < *prozedurkopf* > < *prozedurhauptteil* > | < *type* > **procedure** < *prozedurkopf* > < *prozedurhauptteil* >

5.1. Prozedurvereinbarung und Prozeduranweisung

Die Prozedurvereinbarung – das Unterprogramm – enthält eine Kennzeichnung, die Prozedurbezeichnung und den formalen Parameterteil; den **Prozedurkopf** der Prozedurvereinbarung. Parameter werden durch Komma getrennt. Um die Parameterliste besser dokumentieren zu können, kann an Stelle eines Kommas als **Parameterbegrenzer** die Zeichenfolge

) < *buchstabenkette* > : (

eingesetzt werden. Neben dem Kopf existiert der **Prozedurhauptteil**; er beschreibt den Algorithmus des Verfahrens. Der Hauptteil wird in Form einer Anweisung oder eines Blockes niedergeschrieben. Der Prozedurhauptteil kann auch in Maschinencode niedergeschrieben werden, durch das besondere Codierungszeichen **code** wird dem Compiler mitgeteilt, das die folgenden Codezeichen durch den Compiler zu ignorieren sind. Dadurch ist fast jede beliebige Erweiterung der Sprache möglich. Die Prozedurvereinbarung ist durch das Vereinbarungszei-

chen **procedure** hinreichend gekennzeichnet.

Beispiel Prozedurvereinbarung:
Es soll eine quadratische Gleichung der Form
$Ax^2 + Bx + C = 0$ mit den Lösungen $X_{1,2} = -p/2 \pm \sqrt{(p*p/4 - q)}$
programmiert werden, wobei die Beziehungen p = B/A und q = C/A gelten.

```
procedure  Wurzel1 (x, y) positive Wurzel: ( wp) :  wp  := - x/2 + sqrt ( x *x /4 – y);
procedure  Wurzel2 (x, y) negative Wurzel: (wn) :  wn  := - x/2 -  sqrt ( x *x /4 – y);
```

Der Kopf umfasst Wurzel1 bzw. Wurzel2 als die Prozedurbezeichnungen und (x,y,w..) sind der **formale Parameterteil**. Der Prozedurhauptteil besteht aus einer Ergibtanweisung, die die Formel beschreibt. Formale Parameter werden in einer Prozedur nicht vereinbart, ihre Niederschrift in der formalen Parameterliste weist sie als formale Parameter aus. Eine explizite Vereinbarung dieser Parameter in einer Prozedur definiert eine neue lokale Variable; das Ergebnis der Verarbeitung wäre undefiniert.
Die Prozedurvereinbarung wird durch die Prozeduranweisung aktiviert. Im Aufruf werden die aktuellen Parameter übergeben, die zur Berechnung der Formel notwendig sind.

Beispiel Prozeduranweisung:

```
Wurzel1 (u, v, w);
```

Die Zuordnung der **aktuellen Parameter** in der Parameterliste der Prozeduranweisung zu den **formalen Parametern** in der Parameterliste der Prozeduranweisung erfolgt ausschließlich durch die Reihenfolge ihres Auftretens. Die aktuellen Parameter werden natürlich in dem Block vereinbart, in dem die Prozeduranweisung programmiert wurde.
Die Prozeduranweisung Wurzel1 (u, v, w); wirkt so, als ob der Prozedurhauptteil mit den eingesetzten aktuellen Parametern (u, v, w) an ihrer Stelle stünde; dieses Verfahren nennt man **Einsatzregel**. Durch dieses Verfahren, das Ersetzen der formalen durch die aktuellen Parameter, wird die Vereinbarung der formalen Parameter hinfällig.
Die Zuordnung bei Aktivierung der Prozedurvereinbarung zwischen formalen und aktuellen Parametern entspricht der in der Mathematik üblichen Funktionsschreibweise Wurzel1 (u, v, w) nach Wurzel1 (x, y, wp), lexikographisch von links nach rechts. Die Anzahl der Parameter in beiden Listen müssen numerisch übereinstimmen. Darüber hinaus muss die Art und der Typ jedes aktuellen Parameters mit der Art und dem Typ des entsprechenden formalen Parameters übereinstimmen.

Gesamtbeispiel:

```
begin
  procedure  Wurzel1 (x, y , wp) :  wp  := - x/2 + sqrt ( x *x /4 – y);
  procedure  Wurzel2 (x, y , wn) :  wn  := - x/2  - sqrt ( x *x /4 – y);
  begin real u, v, wp, wn, m, n, pp, pn;
      lies: read (u, v);
      Wurzel1 (u, v, wp);     comment erster Aufruf der Prozedur positive Wurzel;
      Wurzel2 (u, v, wn);     comment erster Aufruf der Prozedur negative Wurzel;
      m := u / 10;
      n := v;
      Wurzel1 (m, n, pp);     comment zweiter Aufruf der Prozedur positive Wurzel;
      Wurzel2 (m, n, pn);     comment zweiter Aufruf der Prozedur negative Wurzel;
      print (wp, wn, pp, pn);
      goto lies
  end
end
```

5.2. Dynamische Prozeduren

Die dynamische Speicherverwaltung ist bei der Nutzung von Blöcken bereits bekannt. Für Prozeduren gelten die gleichen Regeln wie für Blöcke, nur dass bei Prozeduren die Grenzen der Bereiche über Parameter durch den Prozeduraufruf übergeben werden. Für die Vereinbarung von Bereichen mit variablen Grenzen muss immer genügend Arbeitsspeicher im Unterprogramm zur Verfügung gestellt werden. Bei jedem Aufruf des Unterprogramms kann eine andere Anzahl von Speicherstellen bereitgestellt werden.

Beispiel:

Ein Unterprogramm soll das Quadrat einer Matrix vom Typ (n,n) berechnen. Dabei genügt es nicht, die Quadrate der einzelnen Matrixelemente zu bilden, sondern salopp gesagt, die Matrizenmultiplikation ist auf sich selbst auszuführen. Damit das möglich ist, muss die Zeilen- und die Spaltenanzahl übereinstimmen, also vom Typ (n,n) sein.

$$A_{i,k} := \sum_{j=1}^{n} a_{i,j} * a_{j,k}$$

Das übergeordnete Programm soll zunächst den Speicherplatz für eine quadratische Matrix vom Typ (10,10) vereinbaren und die Matrix mit Werten füllen. In der Prozedurvereinbarung wird allgemein das Feld einer Matrix vom Typ (n,n) mit dem formalen Parameter n sowie Hilfsspeicher zum Zwischenspeichern bereitgestellt.

```
begin real array Mat[ 1 : 10, 1 : 10 ];
  procedure  MatrixQuad (QMat , n );                    comment Prozedurkopf;
                    begin integer  i, k, j;             comment Prozedurhauptteil;
                       real summe;
                       array  HMat [ 1 : n, 1 : n] ;
                       for i := 1 step 1 until n do
                         for k := 1 step 1 until n do
                           begin  summe := 0;
                                  for  j := 1 step 1 until n do
                                      summe := summe + QMat [ i, j ] * QMat [ j , k ];
                                  HMat [ i, k ] := summe
                           end
                           for i := 1 step 1 until n do
                             for k := 1 step 1 until n do
                                  QMat [ i, k ] := HMat [ i, k ];
        end MatrixQuad;                                 comment Ende Prozedurhauptteil;
  for i := 1 step 1 until n do
    for k := 1 step 1 until n do  Mat [ i, k ] :=  i + k;   comment Wertezuordnung fuer Mat;
  MatrixQuad ( Mat, 10);                                comment Prozeduranweisung;
  print (Mat);
  MatrixQuad ( Mat, 3);                                 comment Ablaufprotokoll s.u.;
  print (Mat);
end
```

Ablaufbeispiel für Aufruf: MatrixQuad (Mat, 3);

Ausgangsmatrix Mat =

2	3	4
3	4	5
4	5	6

Wertzuweisungen nach jedem Durchlauf im Prozedurhauptteil:

Durchläufe	i	k	j	QMat [i, j]	QMat [j, k]	summe	HMat[i, k]
Nach 1. Durchlauf	1	1	1	2	2	4	
Nach 2. Durchlauf	1	1	2	3	3	13	
Nach 3. Durchlauf	1	1	3	4	4	29	**29**
Nach 4. Durchlauf	1	2	1	2	3	6	
Nach 5. Durchlauf	1	2	2	3	4	18	
Nach 6. Durchlauf	1	2	3	4	5	38	**38**
Nach 7. Durchlauf	1	3	1	2	4	8	
Nach 8. Durchlauf	1	3	2	3	5	23	
Nach 9. Durchlauf	1	3	3	4	6	47	**47**
Nach 10. Durchlauf	2	1	1	3	2	6	
Nach 11. Durchlauf	2	1	2	4	3	18	
Nach 12. Durchlauf	2	1	3	5	4	38	**38**
Nach 13. Durchlauf	2	2	1	3	3	9	
Nach 14. Durchlauf	2	2	2	4	4	25	
Nach 15. Durchlauf	2	2	3	5	5	50	**50**
Nach 16. Durchlauf	2	3	1	3	4	12	
Nach 17. Durchlauf	2	3	2	4	5	32	
Nach 18. Durchlauf	2	3	3	5	6	62	**62**
Nach 19. Durchlauf	3	1	1	4	2	8	
Nach 20. Durchlauf	3	1	2	5	3	23	
Nach 21. Durchlauf	3	1	3	6	4	47	**47**
Nach 22. Durchlauf	3	2	1	4	3	12	
Nach 23. Durchlauf	3	2	2	5	4	32	
Nach 24. Durchlauf	3	2	3	6	5	62	**62**
Nach 25. Durchlauf	3	3	1	4	4	16	
Nach 26. Durchlauf	3	3	2	5	5	41	
Nach 27. Durchlauf	3	3	3	6	6	77	**77**

Quadrierte Matrix Mat =

29	38	47
38	50	62
47	62	77

5.3. Funktionsprozeduren

In arithmetischen und logischen Ausdrücken werden mathematische Aufgabenstellungen bekanntlich dadurch vereinfacht, indem man für bestimmte mathematische Standardlösungen die von ALGOL angebotenen **Standardfunktionen** benutzt (siehe 3.2). Standardfunktionen brauchen nicht vereinbart zu werden. Sie sind fester Bestandteil der Sprache ALGOL. Nun besteht die Möglichkeit, eigene Funktionen zu programmieren, die allerdings eine entsprechende Prozedurvereinbarung benötigen. Zwei wichtige Vorschriften sind zu beachten:

1. Es muss sich um eine einwertige Prozedur handeln, es gibt genau einen Funktionswert
2. Der Typ des Funktionswertes muss explizit vereinbart werden; vor der Prozedurvereinbarung wird eines der Typvereinbarungszeichen **real**, **integer** oder **boolean** gesetzt.

Der errechnete Funktionswert wird an die Aufrufstelle zurückgegeben und falls erforderlich konvertiert. Die Einsetzungsregel reduziert sich auf das Einsetzen des errechneten Wertes.

Beispiel:
Es soll eine quadratische Gleichung der Form

$Ax^2 + Bx + C = 0$ mit den Lösungen $X_{1,2} = -p/2 \pm \sqrt{(p^2/4 - q)}$

programmiert werden, wobei die Beziehungen p = B/A und q = C/A gelten. Die Lösungen der Gleichungen sollen in Form einer Funktionsprozedur programmiert werden.

```
begin real A, B, C, P, Q, x1, x2;
real procedure Wurzel1 (x, y);                    comment Funktionswert ist vom Typ real;
               Wurzel1 := - x / 2 + sqrt (x * x / 4 - y);
real procedure Wurzel2 (x, y);
               Wurzel2 := - x / 2 -  sqrt (x * x / 4 - y)

lies: read (A,B,C);
      if A = 0 then begin;   print  (' Koeffizient A = 0') ;  comment Vermeidung Div. durch  0;
                             goto lies
                    end;
   P = B / A ;
   Q = C / A;
   x1 = Wurzel1 (P,Q);                            comment Funktionsprozeduraufruf; Wurzel1;
   x2 = Wurzel2 (P,Q);                            comment Funktionsprozeduraufruf; Wurzel2;
   print ('x1 = ', x1, 'x2 = ', x2) ;
   goto lies;
end
```

5.4. Parameterfreie Prozeduren

Bei immer wiederkehrenden Algorithmen ist es mitunter lästig oder zu kompliziert, diese Vorgänge immer wieder neu niederzuschreiben. Es ist hilfreich, für derartige Algorithmen Abkürzungen einzuführen, was durch parameterfreie Prozeduren ermöglicht wird. Parameterfreie Prozeduren können ein Spezialfall der Funktionsprozeduren sein, eine weitere Möglichkeit ist die Nutzung von eigentlichen Prozeduren..

Beispiel:
Beim Übergang von kartesischen Koordinaten auf Polarkoordinaten ist oft der Radius

$$r = \sqrt{(x^2 + y^2)}$$

zu berechnen. Eine Abkürzung für diese Wurzel soll über eine parameterfreie Prozedur eingeführt werden.
Für der Bestimmung des Winkels φ ist eine der beiden Formeln zu programmieren:
cosφ = x / r , cosφ = y / r.

```
real procedure Rad; Rad := sqrt ( x ↑2 + y↑2);
```

Der Prozeduraufruf könnte in der Form

```
KosinPhi := x / Rad;
```

erfolgen. Die im Prozedurhauptteil auftretenden Variablen sind natürlich globale Größen.

Etwas umständlicher ist die Nutzung eigentlicher Prozeduren. Das vorangegangene Beispiel müsste durch einen Aufruf mittels einer Prozeduranweisung erweitert werden.

```
procedure Rad; Radius := sqrt ( x ↑2 + y↑2);
.....
Rad ; KosinPhi := x / Radius;
```

Die eigentliche Prozedur Rad kann nur durch die Prozeduranweisung aufgerufen werden. Neben x und y ist auch noch Radius eine globale Größe.

5.5. Rekursive Prozeduren

5.5.1. Begriffsklärung

Eine Funktion oder Prozedur, die sich selbst aufruft, wird als rekursive Funktion bzw. als rekursive Prozedur bezeichnet. Den Aufruf selbst nennt man **Rekursion**.

Die **rekursive Schreibweise** bei der Definition der ganzen Zahl mittels der erweiterten Backus-Naur-Form wurde im Punkt 2.3. angewendet.

< ganze zahl > ::= [{±}] < ziffer > | < ganze zahl > < ziffer >

Die Lesart: eine ganze Zahl kann eine vorzeichenbehaftete Ziffer **oder** eine vorzeichenbehaftete Ziffer **und** eine weitere Ziffer, somit eine Folge von Ziffern mit wahlweisem Vorzeichen sein.
Bei der **rekursiven Programmierung** ruft sich die Prozedur oder Funktion selbst wieder auf. Rekursive Programmierung kann in prozeduralen und in objektorientierten Programmiersprachen angewendet werden. Obwohl diese Sprachen in ihrem Sprachstandard die Rekursion ausdrücklich zulassen, stellen Selbstaufrufe bzw. gegenseitige Aufrufe auf Grund der verwendeten Programmierparadigmen eher die Ausnahme dar. Auch wenn zur Verbesserung der Lesbarkeit und des Programmierstils häufig auf die Rekursion zurückgegriffen wird, sind die meisten Programmeinheiten iterativ. Bei der **iterativen Programmierung** werden im Gegensatz zur rekursiven Programmierung keine Selbstaufrufe, sondern Schleifen, wie in ALGOL z.B. die Laufanweisungen verwendet.

5.5.2. Beispiel – Berechnung der Fakultät einer Zahl

Ein einfaches und sinnvolles Beispiel für die Verwendung der rekursiven Programmierung ist die Berechnung der Fakultät einer Zahl. Die Fakultät n! ist das Produkt aller ganzen Zahlen von 1 bis zur n.ten Zahl. Für alle natürlichen Zahlen ist

$$n! = 1 * 2 * 3 * ... * n = \prod_{k=1}^{n} k$$

als das Produkt der natürlichen Zahlen von 1 bis n definiert. Da das leere Produkt gleich 1 ist, gilt außerdem 0! = 1.

Die Fakultät lässt sich auch rekursiv definieren:

$n! = n\,(n-1)!$ für ganzes $n > 0$
$0! = 1$

Für eine zu programmierende Prozedur würde die Problemstellung folgendermaßen zu beschreiben sein:

Man definiert eine Prozedur Fakultaet, die eine ganze Zahl n als Eingabewert erhält. Diese Prozedur multipliziert diese Zahl n mit dem Rückgabewert von Fakultaet (n – 1). Für n = 0 liefert die Prozedur das Ergebnis 1, die Abbruchbedingung der Rekursion.

Die rekursive Prozedur kann mit einem bedingten Ausdruck dargestellt werden:

```
integer procedure Fakultaet ( n ); value n; integer n;
        Fakultaet := if n = 0 then 1 else n * Fakultaet (n – 1)
```

Die rekursive Prozedur kann auch ohne bedingten Ausdruck aber mit bedingter Anweisung programmiert werden:

```
integer procedure Fakultaet ( n ); value n; integer n;
        if n = 0 then Fakultaet := 1 else Fakultaet := n * Fakultaet (n – 1)
```

Beide Prozedurvereinbarungen sind in der Wirkungsweise völlig identisch. Der Werteteil des Prozedurkopfes, durch das Grundsymbol **value** hinreichend charakterisiert, besteht aus der ganzen Zahl, deren Fakultät ermittelt werden soll. Die Rekursivität wird nicht nur durch das Auftreten der gleichen Prozedurbezeichnung im Hauptteil erzeugt, sondern auch durch Prozeduranweisungen, in denen die Prozedurbezeichnung als aktueller Parameter vorkommt.

Alle rekursiven Algorithmen lassen sich auch durch iterative Programmierung implementieren. Es gilt der Grundsatz, dass für einfache Probleme eine iterative Programmierung häufig effizienter ist, da durch wiederholte Funktionsaufrufe zusätzlicher Programmcode und erhöhter Arbeitsspeicherverbrauch zu verzeichnen ist. Bei komplizierten Problemstellungen lohnt sich oft der Einsatz rekursiver Lösungen, da eine iterative Lösung schnell sehr unübersichtlich und ineffizient werden kann. Die Ermittlung der Fakultät einer Zahl kann damit auch mit einer Funktionsprozedur mit iterativer Vorgehensweise programmiert werden:

```
integer procedure Fakultaet ( n ); integer n;
        begin integer i, Ergebnis;
              Ergebnis := 1;
              for i := 1 step 1 until n do  Ergebnis := Ergebnis * i;
              Fakultaet := Ergebnis;
        end
```

5.5.3. Kellerprinzip bei rekursiven Prozeduren

Bei der Programmierung der Berechnung der Fakultät mittels einer rekursiven Prozedur (siehe Beispiel oben) würde z.B. mit der Startzahl n = 4 der Computer ausführen:

4* (3* (2* (1* Fakultaet (0)))) mit dem Ergebnis = 24.

Damit eine solche Vorgehensweise möglich ist, sind Compiler notwendig, die die Objektcodes so erstellen, dass rekursive Prozeduren und auch entsprechende Blöcke ihre Daten nach dem sog. Keller-Prinzip ablegen können. Die Verwendung eines Stapelspeichers zur Übersetzung von Programmiersprachen wurde 1957 von Friedrich L. Bauer und Klaus Samelson unter dem Namen „Kellerprinzip“ patentiert. Ein Stapelspeicher, auch Keller, Kellerspeicher, Stapel oder Stack genannt, ist eine der einfachsten abstrakten Datenstrukturen mit beschränktem Zugriff auf gespeicherte Elemente. Im Stapelspeicher können Datenobjekte nur „von oben“ hinzugefügt (eingekellert) und von oben entnommen (ausgekellert) werden. Es gilt also das LIFO-

Prinzip (**Last-in-First-out-Prinzip**). Im Compilerbau ist dieser rekursiver Abstieg eine Technik, bei der die entsprechende Sprache geparst wird, d.h. die Sprache wird zerlegt und umgewandelt in ein brauchbares Format für die weitere Verarbeitung, in eine sog. Metasprache.
Betrachtet man nun das Kellerprinzip bei der Ermittlung der Fakultät mittels der oben programmierten rekursiven Prozedur, kann man jeden Aufruf derselben Prozedur aus ihrem Hauptteil heraus als eine Kellerstufe abwärts annehmen, auf der alle diejenigen Größen abgelegt werden, die beim neuen Aufruf auch einen neuen Wert erhalten. Auf der Kellerstufe werden der Reihe nach die Werte

n, n - 1, ..., 3, 2, 1

eingekellert. Ist der aktuelle Parameter n = 0, wird im nächsten Aufruf der **then** – Zweig der bedingten Anweisung wirksam. Die unterste Stufe der Kellerung ist erreicht, denn jetzt wird erstmalig der Prozedurbezeichnung Fakultaet ein Wert zugewiesen. Beim Aufstieg aus dem Keller, beim Auskellern, kommt man immer zu den **else** – Zweig zurück, also werden die jeweils aktuellen Werte von n mit Fakultaet (n – 1) verknüpft, so dass beim Aufstieg das Produkt

1 * 1 * 2 * ... * (n – 1) * n

der Reihe nach gebildet werden kann und somit die Fakultät der Größe n vorliegt.

6. ALGOL – Programm

6.1. Allgemeine Betrachtungen

Nach dem „Revised Report on the Algorithmic Language ALGOL 60“ hat jedes ALGOL-Programm die Form eines Blockes oder einer Verbundanweisung. Dieser Block oder diese Verbundanweisung darf nicht in einer anderen Anweisung enthalten sein und nicht andere Anweisungen abarbeiten, die nicht in ihnen enthalten sind. Genau dann kann man den Block oder die Verbundanweisung als ALGOL-Programm bezeichnen.

Syntax Programm:
< *programm* > ::= < *block* > | < *verbundanweisung* >

Zur Bildung eines ALGOL-Programmes gelten folgende Regeln:

1. Es dürfen in einem Programm keine Bezeichner auftreten, die nicht bzgl. dieses Programmes lokal sind
2 Alle Vereinbarungen und Anweisungen müssen innerhalb dieses Programmes erfolgen
3 Aus einem Programm darf nicht herausgesprungen werden
4 Ein Programm darf nicht markiert werden
5 Ein Programm muss alle Vereinbarungen und Anweisungen sich selbst beschaffen, die zur Abarbeitung des Programmes notwendig sind.

6.2. Abschließende Beispiele

Es ist ein Algol-Programm zu schreiben, das aus einer vorgegebenen Mengen von reellen Zahlen die positive Quadratwurzel mittels der Standardfunktion **sqrt** () zieht und die Ergebnisse mittlels **print** () ausgibt . Diese Programme sollen einige Möglichkeiten der Programmierung hinsichtlich dieser Aufgabenstellung aufzeigen.

1. Quadratwurzelberechnung als Verbundanweisung:

```
begin print (10);                                   comment Vom Eingabegeraet sind 10
                                                            Zahlen zur Verfuegung zu
                                                            stellen;
    begin real array Feld [ 1 : 10 ] ;
          integer i;
          for i := 1 step 1 until 10 do read ( Feld [ i ] ) ;   comment Vom Eingabegeraet werden10
                                                            Zahlen zur Verfuegung  gestellt;
             for i := 1 step 1 until 10 do ;
                begin print ( i ) ;                     comment Verbundanweisung;
                      print ( sqrt (Feld [ i ] )) ;     comment Druck Wurzel;
                end
    end
end
```

2. Quadratwurzelberechnung als Verbundanweisung mit variablen Feld:

```
begin integer n;
      read ( n );                                   comment Eingabe der Indexgrenze;
      begin real array Feld [ 1 : n ] ;
            integer i;
            for i := 1 step 1 until n do read ( Feld [ i ] ) ; comment Vom Eingabegeraet wer den
                                                              n Zahlen eingelesen;
            for i := 1 step 1 until n do ;
               begin print (Feld [ i ] ) ;              comment Druck Zahl;
                     print ( sqrt (Feld [ i ] ) );      comment Druck Wurzel;
               end
      end
end
```

3. Quadratwurzelberechnung als Block:

```
begin integer n;
      read (n);
      begin real array Feld [ 1 : n ] ;
            integer i;
            for i := 1 step 1 until n do read ( Feld [ i ] ) ; comment Vom Eingabegeraet wer den
                                                              n Zahlen eingelesen;
            for i := 1 step 1 until 10 do ;
               begin print (Feld [ i ] ) ;              comment Druck Zahl;
                     print ( sqrt (Feld [ i ] ) );      comment Druck Wurzel;
               end
      end
end
```

Der Unterschied bei diesen beiden Beispielen hinsichtlich des Einsatzes einer Verbundanweisung und eines Blockes ist nicht ersichtlich.
Im nächsten Beispiel werden die Wurzeln in einer Prozedur ermittelt. Durch die Prozeduraufrufe können jeweils n verschiedene Wurzeln berechnet werden.

4. Quadratwurzelberechnung als parameterfreie Prozedur

```
begin
 procedure  Wurzel;
            begin integer n;
               read ( n );                              comment Eingabe der Indexgrenze;
               begin real array Feld [ 1 : n ] ;
                  integer i;
                  for i := 1 step 1 until n do read ( Feld [ i ] ) ;
                  for i := 1 step 1 until n do ;
                     begin print (Feld [ i ] ) ;        comment Druck Zahl;
                           print ( sqrt (Feld [ i ] ) ); comment Druck Wurzel;
                     end
               end
            end
   Wurzel ;                                             comment 1. Aufruf der Prozedur;
   Wurzel ;                                             comment 2. Aufruf der Prozedur;
   Wurzel ;                                             comment 3. Aufruf der Prozedur;
end
```

5. Quadratwurzelberechnung als Prozedur mit Parameterübergabe

```
begin integer n, m, k ;
  procedure  Wurzel ( p ) ;
             begin real array Feld [ 1 : p ] ;
                   integer i;
                   for i := 1 step 1 until p do read ( Feld [ i ] ) ;
                   for i := 1 step 1 until p do ;
                      begin print (Feld [ i ] ) ;          comment Druck Zahl;
                            print ( sqrt (Feld [ i ] ) );  comment Druck Wurzel;
                      end
                end
  read ( n );                  comment Eingabe der Indexgrenze für 1. Prozeduraufruf;
  Wurzel ( n );                comment 1. Aufruf der Prozedur;
  read ( m );                  comment Eingabe der Indexgrenze für 2. Prozeduraufruf;
  Wurzel ( m ), ;              comment 2. Aufruf der Prozedur;
  read ( k );                  comment Eingabe der Indexgrenze für 3. Prozeduraufruf;
  Wurzel ( k ) ;               comment 3. Aufruf der Prozedur;
end
```

6. Berechnung von Primzahlen

Das **Sieb des Eratosthenes** ist ein Algorithmus zur Bestimmung einer Liste oder Tabelle aller Primzahlen kleiner oder gleich einer vorgegebenen Zahl. Er ist nach dem griechischen Mathematiker Eratosthenes von Kyrene benannt.
Zunächst werden alle Zahlen 2, 3, 4,… bis zu einem frei wählbaren Maximalwert k aufgeschrieben. Die zunächst unmarkierten Zahlen sind potentielle Primzahlen. Die kleinste unmarkierte Zahl ist immer eine Primzahl. Nachdem eine Primzahl gefunden wurde, werden alle Vielfachen dieser Primzahl als zusammengesetzt markiert. Man bestimmt die nächstgrößere nicht markierte Zahl. Da sie kein Vielfaches von Zahlen kleiner als sie selbst ist, kann sie nur durch eins und sich selbst teilbar sein. Folglich muss es sich um eine Primzahl handeln. Diese wird dementsprechend als Primzahl ausgegeben. Man streicht wieder alle Vielfachen und führt das Verfahren fort, bis man am Ende der Liste angekommen ist. Im Verlauf des Verfahren werden alle Primzahlen ausgegeben.
Da ein Primfaktor einer zusammengesetzten Zahl immer kleiner gleich der Wurzel der Zahl sein muss, ist es ausreichend, nur die Vielfachen von Zahlen zu streichen, die kleiner oder gleich der Wurzel der Schranke k sind.
Ebenso genügt es beim Streichen der Vielfachen, mit dem Quadrat der Primzahl zu beginnen, da alle kleineren Vielfachen bereits markiert sind.
Das Verfahren beginnt also damit, die Vielfachen 4, 6, 8,… der kleinsten Primzahl 2 durchzustreichen. Die nächste unmarkierte Zahl ist die nächstgrößere Primzahl, die 3. Anschließend werden deren Vielfache 9, 12, 15,… durchgestrichen. Dieses Verfahren wird bis zur vorgegebenen Zahl durchgeführt.

```
begin integer n;
      procedure Primzahlen ( k );                  comment Prozedurvereinbarung;
           value k; integer k;
           begin  integer i, j ;
                real r, s;                         comment Hilfsvariable;
                boolean Mark;                      comment Markierung ein - aus;
                boolean  array  Hilfsfeld [ 1 : k / 2 ] ;
                print ( 2 );                       comment Druck erste Primzahl;
                Mark := false;
```

```
                r := entier ( k – 1 ) / 2 );
                s :=  sqrt (( k ) – 1 ) / 2 ;
                for i := 1 step 1 until r do Feld [ i ] := true;
                for i := 1 step 1 until s do
                    for j := 1 step 1 until  r / ( 2 * i + 1)  do  Hilfsfeld [ i + j * ( 2 * i + 1 ] := false;
                for j := 1 step 1 until r do
                        begin  if  Hilfsfeld [ i ]  then  if ¬ Mark then  print ( 2 * i + 1 );
                                Mark := Hilfsfeld [ i ]
                        end
            end
            read ( n) ;
        Primzahlen ( n );                          comment Aufruf der Prozedur Primzahlen;
end
```

7. EULER-Transformation

Dieses Beispiel wurde dem „Revised Report on the Algorithmic Language ALGOL 60“ entnommen. Es zeigt viele der vorgestellten Vereinbarungs- und Anweisungsmöglichkeiten auf.

Euler berechnet die Summe von fct(i) für i von Null bis unendlich durch eine passend verfeinerte EULER-Transformation. Die Summation wird beendet, sobald der Absolutwert der Terme der transformierten Reihen tim mal aufeinanderfolgend kleiner als eps ist. Deshalb muss man eine Funktion fct mit einem ganzzahligen Argument, eine obere Grenze eps und eine ganze Zahl tim vorgeben. Das Ergebnis ist die Summe sum. Euler ist besonders leistungsfähig im Fall von langsamen konvergenten oder divergenten alternierenden Reihen.

```
procedure euler (fct, sum, eps, tim); value eps, tim; integer tim;
  real procedure fct;
     real sum, eps;
     comment euler computes the sum of fct(i) for i from zero up to infinity by means of a suitably
     refined euler transformation. The summation is stopped as soon as tim times in succession the
     absolute value of the terms of the transformed series are found to be less than eps. Hence, one
     should provide a function fct with one integer argument, an upper bound eps, and an integer tim.
     The output is the sum sum. euler is particularly efficient in the case of a slowly convergent or
     divergent alternating series;
begin integer i, k, n, t;
       array m [0: 15]; real mn, mp, ds;
        i := n; t := 0; m [0] := fct (0); sum := m [0] /2;
nextterm: i := i + 1; mn := fct(i);
        for k := 0 step 1 until n do
            begin mp := (mn + m[k]) /2; m [k] := mn;
                  mn := mp
            end means;
         if (abs (mn) < abs (m[n])) ∧ (n < 15)  then
              begin ds := mn /2; n := n+1; m[n] := mn end accept
                                                  else ds := mn;
         sum := sum + ds;
         if abs (ds) < eps then t := t + 1 else t := 0;
        if t < tim then goto nextterm
  end euler
```

7. Syntax der Sprache ALGOL - Beschreibung mit der erweiterten Backus-Naur-Form

Syntax Grundsymbol:

< *grundsymbol* > ::= < *buchstabe* > | < *ziffer* > | < *logischer wert* > | < *begrenzer* >

Syntax Buchstabe:

< *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

Syntax Ziffer:

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

Syntax logischer Wert:

< *logischer wert* > ::= **true** | **false**

Syntax Begrenzer:

< *begrenzer* >::= < *operator* > | < *trennzeichen* > | < *klammer* > | < *vereinbarungszeichen* > |
< *spezifikation* >

Syntax Operator:

< *operator* > ::= < *arithmetische operator* > | < *vergleichsoperator* > | < *logischer operator* > |
< *folgeoperator* >

Syntax arithmetischer Operator:

< *arithmetischer operator* > ::= + | - | * | / | ÷ | ↑

Syntax Vergleichsoperator:

< *vergleichsoperator* > ::= < | ≤ | = | ≥ | > | ≠

Syntax logischer Operator:

< *logischer operator* > ::= ≡ | ⊃ | ∨ | ∧ | ¬

Syntax Folgeoperator:

< *folgeoperator* > ::= **goto** | **if** | **then** | **else** | **for** | **do**

Syntax Trennzeichen:

< *trennzeichen* > ::= , | . | 10 | : | ; | : = | ␣ | **step** | **until** | **while** | **comment**

Syntax Klammer:

< *klammer* > ::= (|) | [|] | ` | ' | **begin** | **end**

Syntax Vereinbarungszeichen:

< *vereinbarungszeichen* >::= **boolean** | **integer** |**real** | **array** | **switch** | **procedure** | **own**

Syntax Spezifikationszeichen:

< *spezifikationszeichen* > ::= **string** | **label** | **value**

Syntax Bezeichner:

< *bezeichner* > ::= < *buchstabe* > | < *buchstabe* >< *buchstabe* > | < *buchstabe* >< *ziffer* >

Syntax Zahl:

< zahl > ::= < ganze zahl > | < dezimalbruch > | < exponententeil > | < dezimalzahl >
< ganze zahl > ::= [{±}] < ziffer > | < ganze zahl > < ziffer >
< dezimalbruch > ::= . < ganze zahl ohne vorzeichen >
< exponententeil > ::= 10 < ganze zahl >
< dezimalzahl > ::= < ganze zahl ohne vorzeichen > | < dezimalbruch > |
< ganze zahl ohne vorzeichen > < dezimalbruch >

Syntax Kette:

< kette > ::= ` < offene kette > '
< offene kette > ::= < echte kette > | ` < offene kette > ' | < offene kette > < offene kette >
< echte kette > ::= < jede beliebige folge von grundsymbolen außer ` ' > | < leere kette >
< leere kette > ::=

Syntax Ausdruck:

< ausdruck > ::= < arithmetischer ausdruck > | < logischer ausdruck > | < zielausdruck >

Syntax Variable:

< variable > ::= < einfache variable > | < indizierte variable >

Syntax einfache Variable:

< einfache variable > :: = < bezeichner >

Syntax indizierte Variable:

< indizierte variable > :: = < feldbezeichner > [*< indexliste >*]
< feldbezeichner > ::= < bezeichner >
< indexliste > ::= < indexausdruck > | < indexliste > , < indexausdruck >
< indexausdruck > ::= < arithmetischer ausdruck >

Syntax Vereinbarung:

< vereinbarung > ::= < typvereinbarung > | < feldvereinbarung > | < verteilervereinbarung > |
< prozedurvereinbarung >

Syntax Typvereinbarung einfache Variable:

< typvereinbarung > ::= < typ > < typenliste > | **own** *< typ > < typenliste >*
< typ > ::= **integer** | **real** | **boolean**
< typenliste > ::= < einfache variable >| < typenliste > , < einfache variable >

Syntax Feldvereinbarung für indizierte Variable:

< feldvereinbarung > ::= **array** *< feldliste >* | *< typ >* **array** *< feldliste >* |
own *< typ >* **array** *< feldliste >*
< typ > ::= **integer** | **real** | **boolean**
< feldliste > ::= < feldsegment > | < feldliste > , < feldsegment >
< feldsegment > ::= < feldbezeichnung > [*< grenzenliste >*] |
< feldbezeichnung > , < feldsegment >
< grenzenliste > ::= < grenzenpaar > | < grenzenliste > , < grenzenpaar >
< grenzenpaar > ::= < untere grenze > : *< obere grenze >*
< untere grenze > ::= < arithmetischer ausdruck >
< obere grenze > ::= < arithmetischer ausdruck >

Syntax arithmetischer Ausdruck:

< arithmetischer ausdruck > ::= < einfacher arithmetischer ausdruck > |
< bedingter arithmetischer ausdruck >

Syntax einfacher arithmetischer Ausdruck:

< einfacher arithmetischer ausdruck > ::= < arithmetischer operand > < arithmetischer operator >
< arithmetischer operand >

Syntax arithmetischer Operand:

< arithmetischer operand > ::= < zahl > | < variable > | < funktion >

Syntax algebraische Klammern:

< algebraische klammer > ::= (|)

Syntax bedingter arithmetischer Ausdruck:

< bedingter arithmetischer ausdruck > ::= **if** *< logischer ausdruck >*
then *< arithmetischer ausdruck >*
else *< arithmetischer ausdruck >*

Syntax logischer Ausdruck:

< logischer ausdruck > ::= < einfacher logischer ausdruck > | < bedingter logischer ausdruck >

Syntax einfacher logischer Ausdruck:

< einfacher logischer ausdruck > ::= < logischer operand > < logischer operator >
< logischer operand >

Syntax logischer Operand:

< logischer operand > ::= {**true** | **false**} | *< variable vom typ boolean > |*
< funktion vom typ boolean > | < vergleich >

Syntax Vergleich:

< vergleich > ::= < einfacher arithmetischer ausdruck > < vergleichsoperator >
< einfacher arithmetischer ausdruck >

Syntax bedingter logischer Ausdruck:

< bedingter logischer ausdruck > ::= **if** *< logischer ausdruck >*
then *< einfacher logischer ausdruck >*
else *< logischer ausdruck >*

Syntax Zielausdruck:

< marke > :: = < bezeichner > | < ganze zahl ohne vorzeichen >
< verteilerbezeichnung > :: = < bezeichner >
< zielverteiler > :: = < verteilerbezeichnung > [*< indexausdruck >*]
< einfacher zielausdruck > :: = < marke > | < zielverteiler > | (< zielausdruck >)
< zielausdruck > :: = < einfacher zielausdruck > |
if *< logischer Ausdruck >* **then** *< einfacher zielausdruck >* **else** *< zielausdruck >*

Syntax Anweisung:

< anweisung > ::= < unbedingte anweisung > | < bedingte anweisung > | < laufanweisung >
< unbedingte anweisung > ::= < ergibtanweisung > | < sprunganweisung > | < leere anweisung > |
< prozeduranweisung >

Syntax Ergibtanweisung:

< linke seite > :: = < variable > := | < prozedurbezeichnung > :=
< liste linke seite > ::= < linke seite > | < liste linke seite > < linke seite >
< ergibtanweisung > ::= < liste linke seite > < arithmetischer ausdruck > |
< liste linke seite > < logischer ausdruck >

Syntax Marke:

< *marke* > :: = < *bezeichner* > := | < *ganze zahl ohne vorzeichen* >

Syntax markierte Anweisung:

< *markierte anweisung* > :: < *marke* > : < *anweisung* > | < *marke* > : < *markierte anweisung* >

Syntax leere Anweisung:

< *leere anweisung* > ::= < *leere zeichenkette* >
< *leere zeichenkette* > ::=

Syntax Sprunganweisung:

< *sprunganweisung* > ::= **goto** < *zielausdruck* >
< *zielausdruck* > ::= < *einfacher zielausdruck* > | < *wenn-klausel* > < *einfacher zielausdruck* >
else < *zielausdruck* >
< *wenn-klausel* > ::= **if** < *logischer ausdruck* > **then**
< *einfacher zielausdruck* > ::= < *marke* > | < *zielverteiler* > | < *zielausdruck* >

Syntax Verteilervereinbarung:

< *verteilerbezeichnung* > ::= **switch** < *verteilerbezeichnung* > := < *verteilerliste* >
< *verteilerbezeichnung* > ::= < *bezeichner* >

Syntax bedingte Anweisung:

< *bedingte anweisung* > ::= **if** < *logischer ausdruck* > **then** < *unbedingte anweisung* > |

if < *logischer ausdruck* > **then** < *unbedingte anweisung* > **else** < *anweisung* >

Syntax der Laufanweisung:

< *laufanweisung* >::= [< *marke* >] **for** < *laufvariable* > := < *folge von lauflistenelementen* >
do < *anweisung* >;
< *laufvariable* > ::= < *einfache variable* > | < *indizierte variable* >
< *lauflistenelement* > ::= < *arithmetischer ausdruck* >
| < *aritmetischer ausdruck* > **step** < *arithmetischer ausdruck* > **until** < *arithmetischer ausdruck* >
| < *aritmetischer audruck* > **while** < *logischer ausdruck* >

Syntax Verbundanweisung:

< *verbundanweisung* >::= [< *marke* >] **begin** < *folge von anweisungen* > **end**

Syntax Block:

< *block* >::= [< *marke* >] **begin**
[< *folge von vereinbarungen* >]
< *folge von anweisungen* >
end [< *markenpräfix* >];

Syntax Prozeduranweisung:

< *aktueller parameter* > ::= < *kette* > | < *ausdruck* > | < *feldbezeichner* > | < *verteilerbezeichner* | < *prozedurbezeichnung* >

< *buchstabenkette* > ::= < *buchstabe* > | < *buchstabenkette* > < *buchstabe* >

< *parameterbegrenzer* > ::= , |) < *buchstabenkette* > : (

< *aktuelle parameterliste* > :: = < *aktueller parameter* > |
< *aktuelle parameterliste* > < *parameterbegrenzer* >
< *aktueller parameter* >

< *aktueller parameterteil* > :: = < *leere kette* > | (< *aktuelle parameterliste* >)

< *prozeduranweisung* > ::= < *prozedurbezeichnung* > < *aktueller parameterteil* >

Syntax Prozedurvereinbarung:

< *formaler parameter* > :: = < *bezeichner* >

< *formale parameterliste* > :: = < *formaler parameter* > |
< *formale parameterliste* > < *parameterbegrenzer* > < *formaler parameter* >

< *formaler parameterteil* > :: = < *leere kette* > | (< *formale parameterliste* >)

< *bezeichnungenliste* > :: = < *bezeichnung* > | < *bezeichnungenliste* >, < *bezeichnung* >

< *werteteil* > :: = **value** < *bezeichnungenliste* > ; | < *leere kette* >

< *vereinbarungszeichen* > :: = **string** | < *type* > | **array** | < *type* > **array** | **label** | **switch** | **procedure** | < *type* > **procedure**

< type > ::= **real** | **integer** | **boolean**

< *vereinbarungsteil* > :: = < *leere kette* > | < *vereinbarungszeichen* > < *bezeichnungenliste* > ; |
< *vereinbarungsteil* > < *vereinbarungszeichen* > < *bezeichnungenliste* >

< *prozedurkopf* > :: = < *prozedurbezeichnung* > < *formaler parameterteil* > ; < *werteteil* >
< *vereinbarungsteil* >

< *prozedurhauptteil* > :: = < *anweisung* > | < **code** >

< *prozedurvereinbarung* > :: = **procedure** < *prozedurkopf* > < *prozedurhauptteil* > |
< *type* > **procedure** < *prozedurkopf* > < *prozedurhauptteil* >

Syntax Programm:

< *programm* > ::= < *block* > | < *verbundanweisung* >

8. Quellenangaben

1. Peter Naur
   Resived Report on the Algorithmic Language ALGOL 60
   ALGOL Bulletin Supplement no 2

2. Peter Naur
   Bericht über die Algorithmische Sprache ALGOL 60
   ALGOL Bulletin Supplement no 3

3. Autorenkollektiv
   Revidierter Bericht über die algorithmische Sprache ALGOL 60 und Bericht über Eingabe- und Ausgabeprozeduren
   Akademie-Verlag, Berlin 1966

4. B.Higman
   Programiersprachen – eine vergleichende Studie
   BSB B.G. Teubner Verlagsgesellschaft

5. Kerner - Zielke
   Einführung in die algorithmische Sprache ALGOL
   BSB B.G.Teubner Verlagsgesellschaft 1969

6. Günter Riedewald
   Formale Beschreibung von Programmiersprachen
   Eine Einführung in die Semantik

7. Meine persönlichen Schulungsunterlagen

8. Bernd Hartwich
   Einführung in die algorithmische Sprache ALGOL 60 – R300
   Skript der Vorlesung an der Humboldt-Universität zu Berlin
   Frühjahr - und Herbstsemester 1970 - 1974