Ada-Kurs, Teil 1

Universität Stuttgart, April 2003

"Übungsteilnahme" oder "Klausur Ende Juli 2003"

Volker Claus, Fakultät 5 "I, E und I"
Institut für Formale Methoden der Informatik (FMI)

Hinweis: Dies ist ein einmaliger Spezialkurs für die Studierenden der Wirtschaftsinformatik, die bisher nicht an dem obligatorischen Programmierkurs 1 haben teilnehmen können.

Da diese Studierenden den Kurs zusätzlich zu ihrem normalen Pflichtprogramm besuchen müssen, sind die Bedingungen anders als in den üblichen Kursen.

Es wird erwartet, dass alle Teilnehmer(innen) die im Wintersemester 02/03 verteilten Übungsaufgaben bearbeiten, um den obligatorischen zweiten Teil des Programmierkurs 2, der im SS 03 zu belegen ist, erfolgreich bestehen zu können.

Teil 1 des Ada-Kurses

1. Grundbegriffe 1: EBNF, einfache Programme, Bezeichner, Literale, Deklarationen, Ein-/Ausgabe, if, while, for.
2. Grundbegriffe 2: Aufzählungstypen, Standarddatentypen, Variable, Konstanten, Unterbereiche, Typanpassung, case, exit, Sprung
3. Grundbegriffe 3: Blöcke, Sichtbarkeit, Untertyp, Records, Felder
4. Datenstrukturen: Feldgrenzen, Zeiger, Listen
5. Funktionen und Unterprogramme, Überladen
6. Diskriminanten, Generizität, Vererbung
7. Pakete, Spezifikations- und Implementierungsteil
8. Ausnahmebehandlung

Abschnitt 1

Grundbegriffe 1:

Algorithmen,
EBNF, einfache Programme,
Bezeichner, Literale,
Deklarationen,
Ein-/Ausgabe,
Boolean, Integer,
if, while, for.

Aus der Grundvorlesung Informatik kennen wir einige wichtige Sprachelemente zur Beschreibung von (sequentiellen) Algorithmen und den von ihnen manipulierten Daten:

(A1) Ein Algorithmus ist eine Folge von Anweisungen. Eine Anweisung ist entweder eine elementare Anweisung oder sie setzt sich aus anderen Anweisungen zusammen. Der Algorithmus erhält einen Bezeichner (einen Namen).

(A2) Ein Algorithmus arbeitet auf Daten. Daten werden in "Behältern", genannt Variablen, abgelegt. Variablen werden zu Beginn des Algorithmus vereinbart oder deklariert. Dadurch wird festgelegt, welche Daten in den Behälter gelegt werden dürfen und welche nicht.

(A3) Elementare Anweisungen sind von der Form:

null	< leere Anweisung > *Bedeutung:* Tue nichts.
X := α	"Wertzuweisung". α ist ein Ausdruck. *Bedeutung:* Rechne den Ausdruck α aus und lege den erhaltenen Wert in X ab.
get (X)	Leseanweisung. *Bedeutung:* Lies den nächsten Wert ein und lege ihn in der Variablen X ab.
put (α)	Schreibanweisung. *Bedeutung:* Drucke den Wert, den der Ausdruck α besitzt, aus.
$F(X_1, ..., X_n)$	*Bedeutung:* Führe den Algorithmus F mit den Werten der Variablen $X_1, ..., X_n$ aus.

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

(A5) Jede elementare Anweisung ist auch eine Anweisung.

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

(A7) Alternative oder Fallunterscheidung:

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

if β then C else D end if

eine Anweisung.

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

Spezialfälle: Falls D die leere Anweisung ist, so schreibt man kurz if β then C end if (einseitige Falluntersch.).

Falls D erneut eine Fallunterschiedung ist, so schreibt man if β then C elsif β' then D1 else D2 end if

Hinweis: end if ist wie die "Klammer Zu" zum Symbol if.

(A8a) while-Schleife:

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

(A8b) unbegrenzte-Schleife:

Wenn C eine Anweisung ist, dann auch loop C end loop .
Bedeutung: Die Anweisung C wird beliebig oft wiederholt. Man kann sie verlassen durch eine Anweisung in C
exit when β; (oder: if β then exit; end if;)
wobei β ein Boolescher Ausdruck ist (trifft er zu, wenn man auf diese exit-Anweisung stößt, so endet die Schleife sofort.

(A8c) for-Schleife oder Zählschleife:

Wenn C eine Anweisung, i eine Variable für "Bereich" (die in Ada nicht gesondert deklariert wird!), und "Bereich" eine endliche geordnete Menge sind, dann ist auch

```
for i in Bereich loop C end loop
```

eine Anweisung. (i darf in C nicht verändert werden!)

Bedeutung: Setze i auf den ersten Wert von "Bereich". Falls i nicht größer als der letzte Wert von "Bereich" ist, führe C aus. Setze anschließend i auf den nächsten Wert von "Bereich" und wiederhole diesen Vorgang, bis i größer als der letzteWert von "Bereich" ist.

Die for-Schleife besitzt also die gleiche Bedeutung wie

```
i := erster Wert von "Bereich";
while i ≤ letzter Wert von "Bereich" loop
    C; i := nächster Wert von i in "Bereich";  end loop
```

Anmerkungen zur for-Schleife (A8c):

Anmerkung 1: Man kann auch "herunterzählen":

```
for i in reverse Bereich loop C end loop
```

Hierbei wird am Ende der Schleife i nicht erhöht, sondern durch den vorhergehenden Wert in "Bereich" ersetzt:

```
i := letzter Wert von "Bereich";
while i ≥ erster Wert von "Bereich" loop
    C;  i := vorhergehender Wert von i in "Bereich; end loop
```

Anmerkung 2: In der Praxis verwendet man für "Bereich" meist ein Intervall der ganzen Zahlen. Zulässig ist aber jede beliebige, total angeordnete, endliche Menge, also zum Beispiel ein Intervall der Buchstaben (z.B. von 'F' bis 'K', dann wird die Anweisung C genau für die Buchstaben F, G, H, I, J und K durchgeführt).

(A9) Ergänzende Vorschriften

Aus Gründen der Übersichtlichkeit und Lesbarkeit macht man weitere Vorschriften, z.B.:
Die Deklarationen müssen stets am Anfang des Algorithmus oder eines zusammenhängenden Teil-Algorithmus (in Ada sind dies "Blöcke", Funktionen, Prozeduren oder Pakete) angegeben werden (in Blöcken eingeleitet durch declare).
Die auf die Deklarationen folgende Anweisung wird in begin ... end eingeklammert.
Kommentare trennt man bis zum Zeilenende durch das Zeichen -- vom Algorithmus ab.
Jeder vorkommende Bezeichner muss vor seiner ersten Verwendung in einer Deklaration vereinbart worden sein.
Algorithmen beginnen in Ada in der Regel mit
procedure <Name des Algorithmus> is ....

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

Unsere Programme sind also folgendermaßen aufgebaut:

```
procedure <Name des Algorithmus> is
<Deklarationen>;
begin <Anweisung> end
```

Später werden wir dies erweitern. Insbesondere werden wir den Deklarationsteil ausgestalten und wir werden Parameter und Importe ("with") hinzufügen, um mehr Flexibilität zu erreichen und um bereits geschriebene Algorithmen nutzen zu können.

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

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

```
procedure quadratzahl1 is
x, ergebnis: natural;
begin  get (x);                    -- Es wird a eingelesen und in x abgelegt.
       ergebnis := 0;
       for i in 0.. x loop         -- prüfe für i von 0 bis a, ob i^2 = a ist
            if i*i = x then ergebnis := 1; end if; end loop;
       put (ergebnis);
end;
```

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

```
procedure quadratzahl1 is
x, ergebnis: natural;
begin  get (x);                -- Es wird a eingelesen und in x abgelegt.
       ergebnis := 0;
       for i in 0 .. x loop    -- prüfe für i von 0 bis a, ob i² = a ist
           if i*i = x then ergebnis := 1; end if; end loop;
       put (ergebnis);
end;
```

Also: korrekt gebildetes Programm

Fortsetzung Beispiel 1:
Rechnet man das Programm für eine Zahl, z.B. für a = 33 durch, so quadriert man alle Zahlen von 0 bis 33, wobei man jedes Mal feststellt, dass i^2 ungleich 33 ist. Man hätte bereits bei i=6 aufhören können, da ab dann $i^2 > 33$ ist. Dies führt zu folgendem "effizienter" arbeitenden Programm:

```
procedure quadratzahl2 is
var x, i, ergebnis: natural;
begin  get (x);                -- Es wird a eingelesen und in x abgelegt.
       ergebnis := 0;  i := 0;
       while i*i ≤ x loop      -- prüfe nur für i von 0 bis wurzel(a)
           if i*i = x then ergebnis := 1; end if; end loop;
       put (ergebnis);
end;
```

Fortsetzung Beispiel 1:
Das Programm quadratzahl2 führt nicht mehr a, sondern nur noch 2·wurzel(a) Multiplikationen durch. Frage: Kann man die lästigen Multiplikationen sparen?

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

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

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

Fortsetzung Beispiel 1:
Dies führt auf das Programm

```
procedure quadratzahl3 is
var x, u, q, ergebnis: natural;
begin  get (x);                    -- Es wird a eingelesen und in x abgelegt.
       q := 0;  u := 1; ergebnis := 0;
       while q ≤ x loop            -- prüfe für i von 0 bis a, ob i² = a ist
            if q = x then ergebnis := 1; end if;
            q := q+u;  u := u+2; end loop;
       put (ergebnis);
end;
```

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

Frage: Wie prüft man nach, was ein Programm macht?

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

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

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

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

Fortsetzung Beispiel 1

```
procedure quadratzahl3 is
var x, u, q, ergebnis: natural;
begin  get (x);
        q := 0;  u := 1; ergebnis := 0;
        while q ≤ x loop
            if q = x then ergebnis := 1; end if;
            q := q+u;  u := u+2; end loop;
        put (ergebnis);
end;
```

Eingabe sei 14, ⊥ bedeutet "undefiniert"

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


```
procedure quadratzahl3 is
var x, u, q, ergebnis: natural;
begin  get (x);
        q := 0;  u := 1; ergebnis := 0;
        while q ≤ x loop
            if q = x then ergebnis := 1; end if;
            q := q+u;  u := u+2; end loop;
        put (ergebnis);
end;
```

Eingabe sei 14, ⊥ bedeutet "undefiniert"

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

```
procedure quadratzahl3 is
var x, u, q, ergebnis: natural;
begin  get (x);
        q := 0;  u := 1; ergebnis := 0;
        while q ≤ x loop
            if q = x then ergebnis := 1; end if;
            q := q+u;  u := u+2; end loop;
        put (ergebnis);
end;
```

Eingabe sei 14, ⊥ bedeutet "undefiniert"

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

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

Folgende Begriffe wurden vor allem an der Tafel vorgestellt:

BNF, EBNF, Syntaxdiagramm,

Begrenzer, Literale, Zeichensatz,
die 69 reservierten Wörter von Ada,

Variablendeklaration,
Konstantendeklaration

Beispiel: Beschreibung einiger Ada-Anweisungen

sequence_of_statements ::= statement {statement}

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

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

Bemerkung: Bisher haben wir hiervon nur das null_statement, das assignment_statement (= Wertzuweisung) und das exit_statement (zum Verlassen einer Schleife) kennen gelernt.

Beispiel Ada-Anweisungen (Fortsetzung)

label ::= "<<" *label*_statement_identifier ">>"

statement_identifier ::= direct_name

null_statement ::= 'null' ";"

assignment_statement ::= *variable*_name ":=" expression ";"

exit_statement ::= 'exit' [*loop*_name] ['when' condition] ";"

goto_statement ::= 'goto' *label*_name ";"

procedure_call_statement ::= *procedure*_name ";" |
*procedure*_prefix actual_parameter_part ";"

return_statement ::= 'return' [expression] ";"

Bemerkung: Die verbleibenden sechs <simple_statement> behandeln wir später bzw. im Programmierkurs 2.

Beispiel: Ada-Anweisungen (Fortsetzung)

Erneuter Hinweis zur EBNF-Schreibweise:
Alle Nichtterminalzeichen, die in Normalschrift geschrieben sind, stehen irgendwo auf der linken Seite einer EBNF-Regel. Manche Nichtterminalzeichen enthalten kursiv geschriebene Teile. Diese sind bedeutungstragende Hinweise und nur das restliche Nichtterminalzeichen (ohne die kursiven Teile) tritt als linke Seite einer EBNF-Regel auf. Zwei Beispiele:
(1) *variable*_name ist ein name
Der Zusatz *variable* besagt, dass an dieser Stelle eine Variable stehen muss und nicht etwa ein Prozedurname oder eine Marke.
(2) *label*_statement_identifier ist ein statement_identifier
Der Zusatz *label* besagt, dass hier der Bezeichner für eine Marke stehen muss, die irgendwo im Programm vorhanden ist.

Beispiel: Ada-Anweisungen (Fortsetzung)

```
loop_statement ::= [loop_statement_identifier ":"]
                   [iteration_scheme] 'loop'
                   sequence_of_statements
                   'end loop' [loop_identifier] ";"

iteration_scheme ::= 'while' condition |
                     'for' loop_parameter_specification

loop_parameter_specification ::=
    defining_identifier 'in' ['reverse'] discrete_subtype_definition

block_statement ::= [block_statement_identifier ":"]
                    ['declare' declarative_part]
                    'begin' handled_sequence_of_statements
                    'end' [block_identifier] ";"
```

Dies waren 20 Regeln von etwa 400 EBNF-Regeln zur Beschreibung der Syntax von Ada 95.

Üben Sie das Lesen und Verstehen der EBNF-Regeln und damit der Syntax von Ada 95 und anderer Programmiersprachen .

Die komplette Syntax von Ada 95 finden Sie in Büchern, aber auch im Internet an mehreren Stellen, z.B. unter

http://www.adahome.com/rm95/rm9x-P.html

oder (einschl. Syntaxdiagrammen) unter

http://cui.unige.ch/db-research/Enseignement/analyseinfo/Ada95/BNFindex.html.

Die Ada-Syntax ist im "Ada Reference Manual" festgelegt; die Auflistungen der Syntax erfolgt nach dem dort benutzten Schema in 13 Abschnitten.

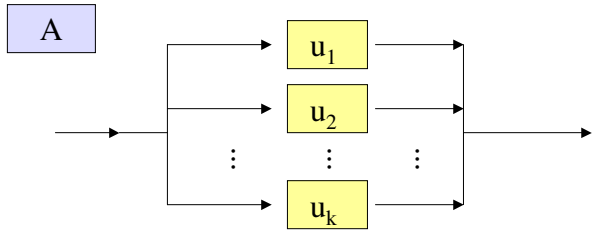
Ende der Syntax von Anweisungen in Ada ■

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

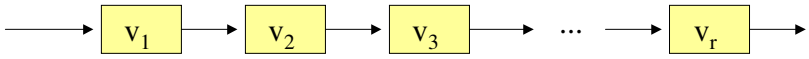
Betrachte eine solche Regel mit dem Nichtterminal A:

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

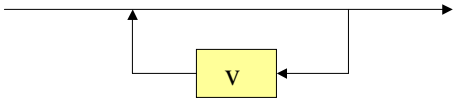
Dann zeichne hierzu das Diagramm



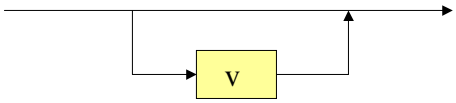
Ist ein u_i von der Form $v_1 v_2 \ldots v_r$ (Konkatenation von Zeichen), so ersetze → u_i → durch:



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



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



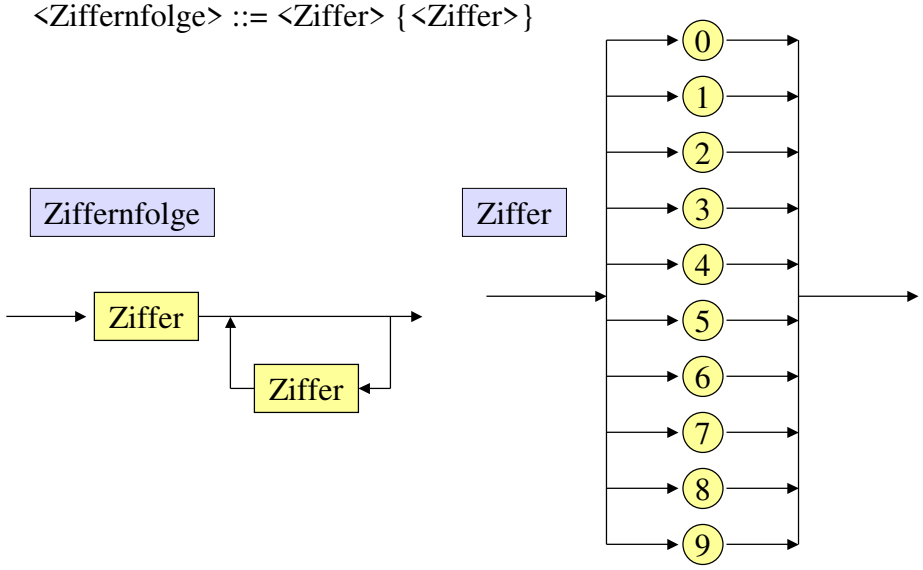
Zum Schluss ersetzt man bei allen Terminalzeichen und Schlüsselwörtern die rechteckigen Umrandungen durch Kreise.

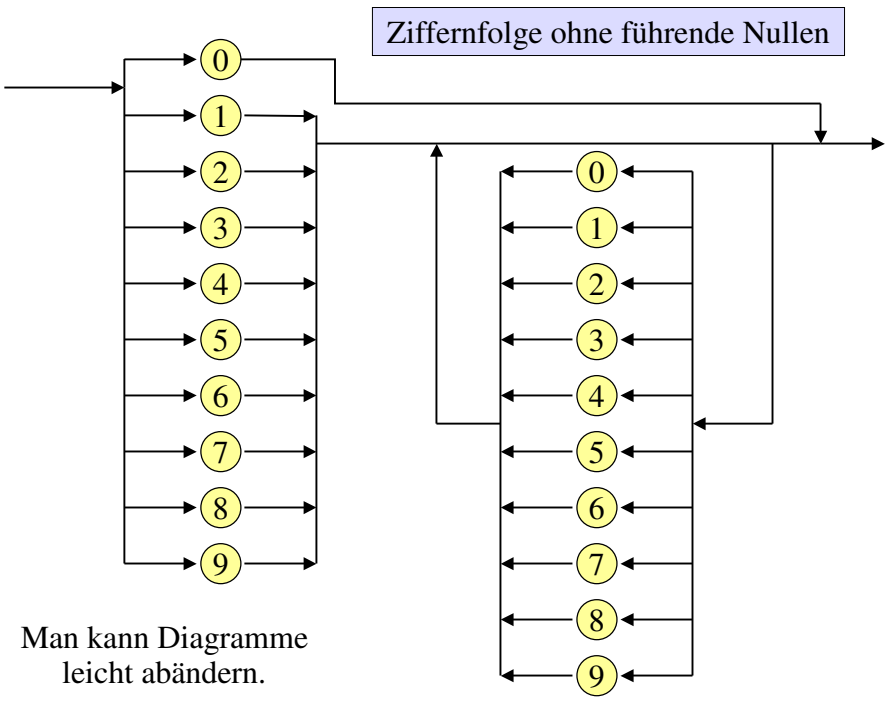
Dieses Vorgehen wird hierarchisch fortgesetzt, wo-durch sich schließlich für jede Grammatik, bzw. EBNF eine grafische Darstellung, das sog. Syntaxdiagramm, ergibt.

Um Wörter aus einem Nichtterminalzeichen A zu erzeugen, durchläuft man das zu A gehörende Syntaxdiagramm vom Eingangspfeil bis zum Ausgangspfeil. Hier gibt es i.A. viele Wege. Für jeden Weg sammelt man die Folge aller hierbei besuchten Terminalzeichen in der Durchlaufreihenfolge auf. Alle diese Zeichenfolgen bildet dann die Menge der von A erzeugten Wörter. Beispiele und eine genauere Formulierung folgen.

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

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





Man kann Diagramme leicht abändern.

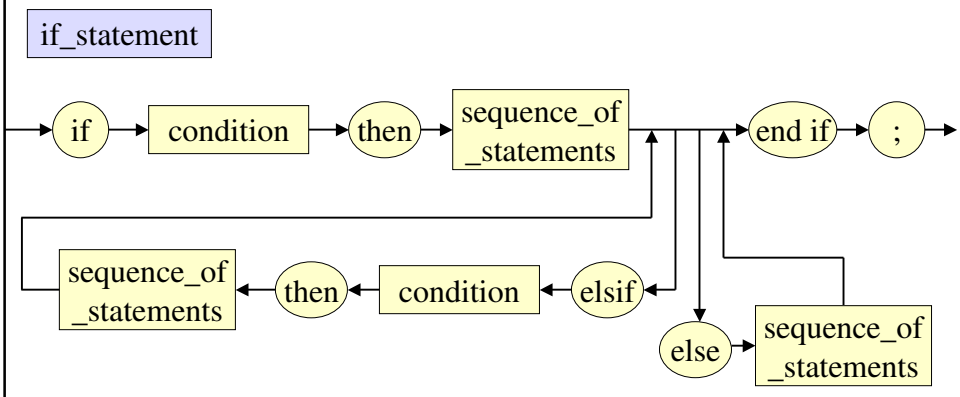
Etwas präzisiere Beschreibung der Bedeutung von Syntaxdiagrammen: Gegeben sind Syntaxdiagramme und eine anfangs leere Ausgabe.

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

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

Beispiel

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



ASCII-Code, erste Hälfte (nach ISO/IEC 646, vgl. DIN 66003)

Nr.	binär	Zeichen	Nr.	binär	Zeichen	Nr.	binär	Zeichen	Nr.	binär	Zeichen
0	0000000	NUL	16	0010000	DLE	32	0100000	Zwi.r	48	0110000	0
1	0000001	SOH	17	0010001	DC1	33	0100001	!	49	0110001	1
2	0000010	STX	18	0010010	DC2	34	0100010	"	50	0110010	2
3	0000011	ETX	19	0010011	DC3	35	0100011	#	51	0110011	3
4	0000100	EOT	20	0010100	DC4	36	0100100	$	52	0110100	4
5	0000101	ENQ	21	0010101	NAK	37	0100101	%	53	0110101	5
6	0000110	ACK	22	0010110	SYN	38	0100110	&	54	0110110	6
7	0000111	BEL	23	0010111	ETB	39	0100111	'	55	0110111	7
8	0001000	BS	24	0011000	CAN	40	0101000	(	56	0111000	8
9	0001001	HT	25	0011001	EM	41	0101001	)	57	0111001	9
10	0001010	LF	26	0011010	SUB	42	0101010	*	58	0111010	:
11	0001011	VT	27	0011011	ESC	43	0101011	+	59	0111011	;
12	0001100	FF	28	0011100	FS	44	0101100	,	60	0111100	<
13	0001101	CR	29	0011101	GS	45	0101101	-	61	0111101	=
14	0001110	SO	30	0011110	RS	46	0101110	.	62	0111110	>
15	0001111	SI	31	0011111	US	47	0101111	/	63	0111111	?

ASCII-Code, zweite Hälfte (nach ISO/IEC 646, vgl. DIN 66003)

Nr.	binär	Zeichen	Nr.	binär	Zeichen	Nr.	binär	Zeichen	Nr.	binär	Zeichen
64	1000000	@	80	1010000	P	96	1100000	`	112	1110000	p
65	1000001	A	81	1010001	Q	97	1100001	a	113	1110001	q
66	1000010	B	82	1010010	R	98	1100010	b	114	1110010	r
67	1000011	C	83	1010011	S	99	1100011	c	115	1110011	s
68	1000100	D	84	1010100	T	100	1100100	d	116	1110100	t
69	1000101	E	85	1010101	U	101	1100101	e	117	1110101	u
70	1000110	F	86	1010110	V	102	1100110	f	118	1110110	v
71	1000111	G	87	1010111	W	103	1100111	g	119	1110111	w
72	1001000	H	88	1011000	X	104	1101000	h	120	1111000	x
73	1001001	I	89	1011001	Y	105	1101001	i	121	1111001	y
74	1001010	J	90	1011010	Z	106	1101010	j	122	1111010	z
75	1001011	K	91	1011011	[	107	1101011	k	123	1111011	{
76	1001100	L	92	1011100	\	108	1101100	l	124	1111100	|
77	1001101	M	93	1011101	]	109	1101101	m	125	1111101	}
78	1001110	N	94	1011110	^	110	1101110	n	126	1111110	~
79	1001111	O	95	1011111	_	111	1101111	o	127	1111111	DEL

Meist schreibt man diesen Code als zweidimensionale Tabelle, siehe Literatur.

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

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

Latin-1 enthält ASCII als die ersten 128 Zeichen. Die Zeichen mit den Nummern 128 bis 159 wurden in Latin-1 frei gelassen; sie werden von verschiedenen Softwareherstellern unterschiedlich verwendet. Oben sind die Zeichen von Microsoft Windows eingetragen (die Nummern 128, 129, 141-144, 157, 158 werden dabei nicht festgelegt; 128 wird meist für das Eurozeichen, 142 und 158 für das modifizierte Z benutzt), in Ada kann man auf diese Zeichen mit symbolischen Bezeichnungen zugreifen, siehe nächste Folie. **vgl. Folie 66**

Die Werte mit den Nummern 0 bis 31 und 127 bis 159 können zwar verwendet, aber oft nicht angezeigt werden. Sie werden durch zwei- oder dreibuchstabige Kürzel bezeichnet und zwar:

Nr.	Kürzel	Nr.	Kürzel			Nr.	Kürzel
				127	DEL		
0	NUL	16	DLE	128	128	144	DCS
1	SOH	17	DC1	129	129	145	PU1
2	STX	18	DC2	130	BPH	146	PU2
3	ETX	19	DC3	131	NBH	147	STS
4	EOT	20	DC4	132	132	148	CCH
5	ENQ	21	NAK	133	NEL	149	MW
6	ACK	22	SYN	134	SSA	150	SPA
7	BEL	23	ETB	135	ESA	151	EPA
8	BS	24	CAN	136	HTS	152	SOS
9	HT	25	EM	137	HTJ	153	153
10	LF	26	SUB	138	VTS	154	SCI
11	VT	27	ESC	139	PLD	155	CSI
12	FF	28	FS	140	PLU	156	ST
13	CR	29	GS	141	RI	157	OSC
14	SO	30	RS	142	SS2	158	PM
15	SI	31	US	143	SS3	159	APC

Reservierte Wörter (Schlüsselwörter) in Ada

In Ada 95 gibt es 69 reservierte Wörter, die nicht als Bezeichner verwendet werden dürfen. Mindestens die folgenden 50 sollten Sie am Ende des Teil 1 des Adakurses kennen:
abs, access, and, array, begin, body, case, constant, declare, delta, digits, do, else, elsif, end, exception, exit, for, function, generic, goto, if, in, is, loop, mod, new, not, null, of, or, others, out, package, private, procedure, raise, range, record, rem, return, reverse, subtype, then, type, use, when, while, with, xor.

Folgende 19 reservierte Wörter kommen in Ada 95 noch hinzu (siehe Programmierkurs 2 im Sommersemester) sowie gewisse Mehrfachbedeutungen der bereits bekannten Schlüsselwörter:
abort, abstract, accept, aliased, all, at, delay, entry, limited, pragma, protected, renames, requeue, select, separate, tagged, task, terminate, until.

Abschnitt 2

Grundbegriffe 2:

Aufzählungstypen,
Float,
Character,
Unterbereiche,
Typanpassung,
case, exit, Sprung

Elementare Datentypen

Die meisten Probleme kann man mit Hilfe von Mengen und auf ihnen definierten Operationen und Relationen beschreiben.

"Elementare" Mengen oder Daten sind: *Wahrheitswerte, Zeichen, natürliche Zahlen, ganze Zahlen, reelle Zahlen.*

Es gibt sie in fast allen Programmiersprachen.

Hinzu kommen selbstdefinierte Mengen (Aufzählungstyp).

In manchen Programmiersprachen gelten auch die Zeichenketten (= Menge der Wörter über dem Alphabet der Zeichen) als elementare Daten mit speziellen Operationen.

Kürzel	Menge	bekannt als	Datentypname
$\mathbb{B}$	{false, true} oder {0, 1}	Boolesche Werte Wahrheitswerte	Boolean
Â	ASCII-Code Latin_1-Code	Tastatur-Alphabet Alphabetszeichen	character
$\mathbb{N}$	{1, 2, 3, 4, ...}	natürliche Zahlen	positive
$\mathbb{N}_0$	{0, 1, 2, 3, ...}	natürliche Zahlen mit der 0	natural
$\mathbb{Z}$	{ ..., -2, -1, 0, 1, 2, 3, ... }	ganze Zahlen	integer
$\mathbb{R}$		reelle Zahlen	float

Definition: *Datentyp*
Eine Menge (oder mehrere Mengen) zusammen mit den hierauf definierten Operationen nennt man einen (konkreten) Datentyp.
Einen Datentyp, den man nicht auf andere Datentypen zurückführt, nennt man einen elementaren Datentyp.

"Unsere" elementaren Datentypen sind die folgenden: Boolean, character, integer und float.

Um sie genau festzulegen, müssen wir zu jeder Menge die zulässigen Operationen hinzufügen. (Mathematisch ist ein Datentyp daher eine Algebra.)

Jede Programmiersprache mag hiervon abweichen. Wir führen die Darstellungen in Ada auf.

Bei der genauen Beschreibung der Datentypen verwenden wir folgendes Schema; dieses Schema bildet die Signatur des Datentyps.

Der Datentyp xyz:
Zugrunde liegende Menge: ...
Nullstellige Operationen (=besondere Konstanten): ...
Einstellige Operationen: ...
Zweistellige Operationen: ...
höherstellige Operationen (sofern vorhanden): ...
Definition gewisser Operationen: ...
Beziehungen, Gesetzmäßigkeiten: ...
Dieser Datentyp in der Sprache Ada 95: ....

Darstellung der Operationen:

Hat man zwei Operationen f, g: $M \times M \rightarrow M$ auf einer Menge M, so kann man zusammengesetzte Ausdrücke ("Terme") bilden:

f(g(x, y), x) oder g(g(f(x,y),z), g(z, x))

Diese Darstellung, dass der Operator vor seinen Operanden steht, bezeichnet man als Prefixnotation. Man kann dann die Klammern auch weglassen:

f g x y x oder g g f x y z g z x

und die Auswertung der Ausdrücke bleibt eindeutig.

Statt dessen kann man die Operatoren auch hinter ihre Operanden schreiben:

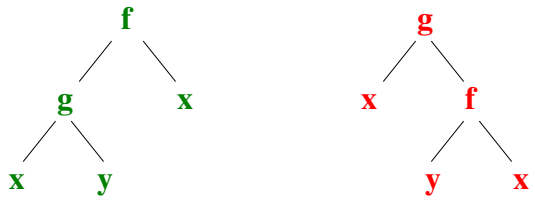
x y g x f oder x y f z g z x g g

Dies nennt man Postfix-Notation (oder polnische Notation). Auch hierbei bleibt die Auswertung der Ausdrücke eindeutig.

Im täglichen Leben verwendet man jedoch in der Regel die Infix-Notation, bei der man das Operationszeichen zwischen die Operanden stellt; obige Beispiele werden dann zu

x g y f x oder x f y g z g z g x

Dies ist nicht mehr eindeutig, da nun z. B. f(g(x, y), x) und g (x, f(y, x)) die gleiche Infix-Notation x g y f x besitzen. Man erkennt dies, wenn man die Baumstruktur betrachtet:



Beide Bäume gehören zu x g y f x

Um bei der Infixnotation Eindeutigkeit zu sichern, verwendet man zum einen Klammern und zum anderen Prioritätsregeln.

Daher müssen bei den Operationen fast immer auch solche Prioritätsregeln für die Operatoren angegeben werden.

Ist nichts angegeben oder haben Operatoren die gleiche Priorität, so wird ein Ausdruck stets von links nach rechts ausgewertet!

Aufzählungstypen ("enumeration types")

Zugrunde liegende Menge: Eine endliche Menge M, die man selbst definieren muss. Die Menge M wird eingeführt durch
type <Name des Datentyps> is (<Liste der Elemente>)
Die Reihenfolge in der Liste gibt zugleich die *Anordnung* der Elemente in der Menge M = $\{m_1, m_2, ..., m_n\}$ an.

Es kommt hierbei per Definition nicht zu Namenskonflikten! Das heißt: Wenn ein Element in mehreren Mengen liegt, so muss der Programmierer selbst darauf achten, dass bei jeder Verwendung eines Elementes eindeutig klar ist, zu welchem Datentyp es gehört.

Nullstellige Operationen (= besondere Konstanten) sind alle Elemente m_i der Menge M. Weiterhin seien First das erste und Last das letzte Element der Menge. (Man muss hierbei den Datentyp angeben, d.h., wenn T der Name des Datentyps ist, so schreibt man T'First bzw. T'Last.)

Einstellige Operationen: (Vorgänger, Nachfolger, Position, Wert)

"predecessor" und "successor" pred, succ: M → M mit
pred(X) = das Zeichen vor X in der Auflistungs-Reihenfolge,
succ(X) = das Zeichen nach X in der Auflistungs-Reihenfolge.
$pred(m_i) = m_{i-1}$ für i>1 und $pred(m_1)$ = undefiniert,
$succ(m_i) = m_{i+1}$ für i<n und $succ(m_n)$ = undefiniert.
Position in der Anordnung der Zeichen: pos: M → $\mathbb{N}_0$
pos(X) = n bedeutet: X ist das n-te Zeichen in der Auflistungsreihenfolge (beginnend mit 0).
n-tes Element in der Anordnung: val: $\mathbb{N}_0$ → M
val(n) = X bedeutet: X ist das n-te Zeichen in der Auflistungsreihenfolge (beginnend mit 0).

Zweistellige Operationen (Vergleichsoperatoren):
Alle sechs Vergleichoperationen "=", "/=", "<", "<=", ">" und ">=" bezüglich der Anordnung der Menge sind zugelassen. Das Ergebnis ist vom Typ Boolean (siehe später).

Hinweis: Tritt ein Element in mehreren Datentypdefinitionen auf

type stuhlteil is (bein, sitzfläche, lehne);
type körperteil is (arm, bein, rücken, sitzfläche, kopf)

so kann man die Elemente eindeutig charakterisieren, indem man den Datentyp abgetrennt durch ' davor schreibt und das Element in Klammern setzt, also:

körperteil'(bein) oder stuhlteil'(bein) oder
körperteil'pos(bein) *(dies ist 1),* stuhlteil'pos(bein) *(dies ist 0).*

Zweistellige Operationen (Fortsetzung):

Für alle Aufzählungstypen T sind die zweistelligen Abbildungen Min und Max definiert: Min, Max: $M \times M \to M$ mit

$Min(X,Y) = X \Leftrightarrow T'pos(X) \leq T'pos(Y) \quad (\Leftrightarrow T'(X) \leq T'(Y))$

$Max(X,Y) = X \Leftrightarrow T'pos(X) > T'pos(Y)$

Im Falle $X \neq Y$ gilt stets $T'Min(X,Y) \neq T'Max(X,Y)$.

Dreistellige Operationen:

if then else fi: $\mathbb{B} \times M \times M \to M$

Diese Operation ist wie folgt definiert:

$$\text{if b then m1 else m2 fi} = \begin{cases} \text{m1, falls b den Wert true besitzt} \\ \text{m2, falls b den Wert false besitzt} \end{cases}$$

Prioritäten werden auf diesen Operationen nicht festgelegt.

Beziehungen, Gesetzmäßigkeiten:

Es gilt stets:

$pos(val(k)) = k$ für $0 \le k \le n-1$,

$val(pos(X)) = X$ für alle Elemente $X \in M$,

$succ(pred(m_i)) = m_i$ für $1 < i \le n$,

$pred(succ(m_j)) = m_j$ für $1 \le j < n$,

$X < Y$ im Datentyp T $\Leftrightarrow$ $pos(X) < pos(Y)$ in $\mathbb{N}_0$.

Der Aufzählungstyp in der Sprache Ada 95:

In Ada werden wie Aufzählungs-Datentypen und ihre ein- und zweistelligen Operationen wie oben angegeben verwendet. Den dreistelligen Operator gibt es in Ada nicht.

Der Datentyp Boolean:

Zugrunde liegende Menge: $\mathbb{B} = \{false, true\}$ (= {falsch, wahr})
Boolean wird zugleich als Aufzählungstyp aufgefasst.

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

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

Zweistellige Operationen (UND, ODER, EXKLUSIVES ODER):
Konjunktion $\wedge: \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ (statt $\wedge$ schreibt man and)
Disjunktion $\vee: \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ (statt $\vee$ schreibt man or)
Ungleichheit (oder auch "Exklusives Oder" genannt)
$\neq: \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ (statt $\neq$ schreibt man xor)
Man kann auch die Gleichheit auf $\mathbb{B}$ verwenden (wie auf allen Aufzählungstypen):
$=: \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

Definition der Operationen des Datentyps Boolean:

	not
false	true
true	false

and	false	true
false	false	false
true	false	true

or	false	true
false	false	true
true	true	true

=	false	true
false	true	false
true	false	true

xor	false	true
false	false	true
true	true	false

Man kann nun Beziehungen und Gesetzmäßigkeiten zwischen diesen Operationen auflisten.

Einige dieser Beziehungen sind (für alle $a, b \in \mathbb{B}$):

not (not a) = a

exor (a, b) = not (equiv (a, b))

impl (a, b) = (not a) or b

not (a and b) = (not a) or (not b)
not (a or b) = (not a) and (not b)

} deMorganschen Regeln

Man benötigt nur die Operationen not und and, um alle anderen Operationen darzustellen (selbst nachweisen).

Der Datentyp Boolean in der Sprache Ada 95:

Der Datentyp Boolean wird in Ada 95 als Aufzählungsdatentyp mit zusätzlichen Operationen aufgefasst, also:

```
type Boolean is (false, true);
```

Damit sind zugleich alle Operationen, die unter 4. beschrieben wurden, auch auf Boolean zugelassen.
Beispielsweise gilt in Ada false < true, Min(true, false) = false, Boolean'val(1) = true und Boolean'First = false.
Man sollte diese Möglichkeiten aber nicht verwenden, da die Wahrheitswerte aus logischer Sicht keine Ordnung besitzen.

Grundsätzlich wird in Ada jeder Ausdruck *vollständig* ausgewertet. Der Boolesche Ausdruck

```
(A > 3) AND (B/0 = 7)
```

hat daher den Wert "undefiniert" wegen der Division durch 0.

Oft möchte man aber den Ausdruck a AND b sofort mit dem Ergebnis false verlassen, wenn sich a schon als false erwiesen hat; man spart die Ausrechnung von b.

Für diese "logische Abkürzung" oder "Kaskadenoperation" gibt es in Ada 95 den Operator "AND THEN". Analoges gilt für or und die Operation "OR ELSE".

In Ada stehen daher neben and und or auch die beiden „Kaskadenoperationen“ AND THEN und OR ELSE zur Verfügung, die man aber "sehr bewusst" einsetzen sollte, da man hierdurch Fehler verdecken kann (vgl. später die "exceptions"):

a AND THEN b entspricht:
falls a nicht zutrifft, ist das Ergebnis false, anderenfalls b.

a OR ELSE b entspricht:
falls a zutrifft, ist das Ergebnis true, anderenfalls b.

Im Normalfall, dass a und b einen der Werte false oder true besitzen, stimmen AND und AND THEN bzw. OR und OR ELSE überein. Den Unterschied zu AND und OR erkennt man nur, wenn man die Operationen zusätzlich mit dem Wert "undefiniert" aufschreibt:

AND	false	true	undef
false	false	false	undef
true	false	true	undef
undef	undef	undef	undef

AND THEN	false	true	undef
false	false	false	**false**
true	false	true	undef
undef	undef	undef	undef

OR	false	true	undef
false	false	true	undef
true	true	true	undef
undef	undef	undef	undef

OR ELSE	false	true	undef
false	false	true	false
true	true	true	**true**
undef	undef	undef	undef

Prioritäten in Ada für Boolean:

In Ada werden nur zwei Prioritätsstufen für Boolesche Operationen festgelegt:

Der Operator NOT erhält eine höhere Priorität, die anderen Operatoren AND, OR und XOR erhalten eine niedrigere Priorität. Aber:

In Ada müssen alle Booleschen Ausdrücke, in denen mindestens zwei der drei Operatoren AND, OR und XOR vorkommen, geklammert werden!

a AND b OR c ist also verboten; man muss (a AND b) OR c oder a AND (b OR c) schreiben.

Erlaubt ist dagegen a OR b OR c ; dies wird von links nach rechts ausgewertet.

In Booleschen Ausdrücken können auch Vergleiche auftreten.

In Ada erhalten *alle* Vergleichsoperatoren eine höhere Priorität als alle Booleschen Operationen. Statt

(17 < 10) AND (NOT (X=Y))

kann man also auch schreiben:

17 < 10 AND NOT X=Y.

Beachte: Allein schon bzgl. der Prioritätsregeln unterscheidet sich Ada von den meisten anderen Programmiersprachen.

Der Datentyp Character:

Zugrunde liegende Menge: In Ada wird Latin-1 Code für Zeichen verwendet. Man unterscheidet zwischen druckbaren Zeichen (graphic_character), Formatierungszeichen (format_effector) und Kontrollzeichen (other_control_function).

Nullstellige Operationen (die Zeichen sowie First, Last)
Einstellige Operationen (pred, succ, pos, val)
Zweistellige Operationen (Vergleichsoperationen, Min, Max)
Dreistellige Operationen: keine
Beziehungen und Gesetzmäßigkeiten:

Alles wie bei Aufzählungstypen.

Der Datentyp Character in der Sprache Ada 95:

Die zugrunde liegende Menge des Datentyps character ist in Ada 95 der Aufzählungsdatentyp der Zeichen (auch "Literale" genannt) des Alphabets "Latin-1", das von 0 bis 127 mit ASCII überein stimmt:
type character is (*NUL*, ..., *US*, ' ', '!', '"', '#', '$', ..., 'ÿ');

Die einzelnen Zeichen werden in Apostroph eingeschlossen, sofern sie graphisch darstellbar sind (das Apostroph selbst wird hierbei durch zwei Apostrophs dargestellt).

Zur Kenntnisnahme verweisen wir auf den vollständigen Latin-1 Code, definiert in der ISO-Norm 8859-1, auf Folie 39. In der Tabelle wird die hexadezimale und die (mit dem Nummerzeichen '#' versehene) dezimale Nummerierung vorangestellt und dann das zugehörige Zeichen genannt. Die blauen Zeichen können in Ada nicht direkt, sondern nur codiert wie auf Folie 40 angegeben, angesprochen werden. Bei den anderen unterscheidet Ada zwischen den druckbaren Zeichen und den Formatierungszeichen (dies wurde unter "Literale" an der Tafel erläutert, geht aber aus der Tabelle auch hervor).

First, Last, die Funktionen pred, succ, pos, val, Min, Max und die Vergleichsoperationen sind hiermit definiert. Zum Beispiel gilt Character'First = NUL, val(51) = '3', pos(pred(pred(DEL))) = 125, 'C' < '©', val(145) = PU1 und succ(Character'Last) ist undefiniert.

Hinweis: Es gibt verschiedene Zeichensätze. Daher gibt es auch in Ada mehrere vordefinierte Character-Datentypen. Diese werden angesprochen durch

Ada.Characters.<Name des Zeichensatzes>

Beispiel: Der oben beschriebene übliche Zeichensatz ist

Ada.Characters.Latin_1

Man kann über

Ada.Characters.ASCII

Ada.Characters.Wide_Character

andere Zeichensätze nutzen. Details siehe spezielle Literatur.

Der Datentyp integer

Zugrunde liegende Menge:
Z = { ..., -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, ... } mit üblicher Ordnung.

Nullstellige Operationen: Alle Zahlen.

Hinweis: Im Prinzip genügen die beiden Konstanten 0 und 1. Jede Zahl lässt sich dann durch Anwenden der Addition und der Bildung der negativen Zahl beschreiben:

3 durch 1 + 1 + 1 oder - 4 = - (1+1+1+1).

Auch die 0 kann man wegen 0 = 1 - 1 noch weglassen.

Einstellige Operationen:

+	einstelliges Plus (wie Identität, verändert also nichts)
-	einstelliges Minus, Bildung der negativen Zahl, Negation
abs	Absolutbetrag einer Zahl
sgn	"Signum", also das Vorzeichen einer Zahl:

$$\text{sgn}(x) = \begin{cases} 1, & \text{falls } x > 0 \\ 0, & \text{falls } x = 0 \\ -1, & \text{falls } x < 0 \end{cases}$$

square Quadrat einer Zahl ($\text{square}(x) = x^2$)

odd: $\mathbf{Z} \to \mathbb{B}$ mit: $\text{odd}(x) = \text{true} \Leftrightarrow x$ ist ungerade $\Leftrightarrow x \bmod 2 = 1$.

even: $\mathbf{Z} \to \mathbb{B}$ mit: $\text{even}(x) = \text{true} \Leftrightarrow x$ ist gerade $\Leftrightarrow x \bmod 2 = 0$.

sgn, square, odd even gibt es in Ada nicht. Man muss sich diese Operationen selbst definieren.

Zweistellige Operationen:

+	Addition zweier Zahlen
-	Subtraktion zweier Zahlen, Differenz zweier Zahlen
·	Multiplikation zweier Zahlen (in Ada: *)
div	ganzzahlige Division (mit stets nichtnegativem Rest) (in Ada nicht vorhanden)
mod	Modulo-Funktion
/	ganzzahlige Division (wie bei reellen Zahlen, aber dann nächste ganze Zahl in Richtung zur Null nehmen)
rem	Rest bei der Division mit /
exp	Exponentiation ($\exp(x,y) = x^y$, nur für $y \geq 0$ definiert; in Ada verwendet man hierfür das Zeichen **)

Alle sechs Vergleichsoperationen "=", "/=", "<", "<=", ">" und ">=" der Funktionalität $\mathbf{Z} \times \mathbf{Z} \to \mathbb{B}$ kommen hinzu.

Definition gewisser Operationen:

Alle Operationen werden als bekannt vorausgesetzt, nur div, mod, / und rem werden hier nochmals präzisiert. Diese vier Operationen sind nur für $y \neq 0$ definiert.

div und mod sind eindeutig festgelegt durch (für $y \neq 0$):

$x = (x \text{ div } y) \cdot y + (x \text{ mod } y)$ mit $0 \leq x \text{ mod } y < abs(y)$.

Die integer-Division "/" ist definiert durch:

$x / y = sgn(x \cdot y) \cdot (abs(x) \text{ div } abs(y))$

Hieraus ergibt sich dann rem durch (für $y \neq 0$):

$(x \text{ rem } y) = x - (x / y) \cdot y$

Auf den natürlichen Zahlen stimmen div und / sowie mod und rem überein.

Machen Sie sich den Unterschiede genau klar: Beide Funktionspaare div und mod sowie / und rem erfüllen die Gleichung:

$x = f(x,y) \cdot y + g(x,y)$ mit $0 \leq abs(g(x,y)) < abs(y)$.

Während mod stets nicht-negative Werte liefert, erhält man bei rem ein nicht-positives Resultat, falls der erste Operand negativ ist.

Beispiele:

16 div 5 = 3,	16 mod 5 = 1,	16 / 5 = 3,	16 rem 5 = 1,
16 div (-5) = -3,	16 mod (-5) = 1,	16 / (-5) = -3,	16 rem (-5) = 1,
(-16) div 5 = -4,	(-16) mod 5 = 4,	(-16) / 5 = -3,	(-16) rem 5 = -1,
(-16) div (-5) = 4,	(-16) mod (-5) = 4,	(-16) / (-5) = 3,	(-16) rem (-5) = -1.

Beziehungen und Gesetzmäßigkeiten:

Die ganzen Zahlen bilden einen mathematischen Ring. Also gelten die üblichen Gesetze: Die Addition ist kommutativ und assoziativ mit der Einheit 0 und der Inversenbildung "-x", die Multiplikation ist kommutativ und assoziativ mit der Einheit 1, die beiden Operationen sind das Distributivgesetz miteinander verbunden. Weitere Gesetzmäßigkeiten lassen sich herleiten, zum Beispiel:

odd(x) = not even(x)

((-x) div y) = -1 - (x div y)

x mod y = y - ((-x) mod y)

Der Datentyp integer in der Sprache Ada 95

In Ada ist auch der Datentyp integer ein Aufzählungstyp, der von der kleinsten bis zur größten darstellbaren Zahl reicht:

type integer is (Integer'First, Integer'First+1, ..., Integer'Last);

Daher sind auch Min, Max, pred, succ, pos und val für Integer definiert, allerdings mit der Ausnahme, dass die Zahl 0 die Positionsnummer 0 erhält, d.h., für integer gilt für alle Zahlen a:

pred(a) = a-1; succ(a) = a+1; pos(a) = a, val(a) = a.

In der Praxis orientiere man sich stets zuvor, welche Zahlen auf dem jeweiligen Rechner Integer'First und Integer'Last sind. Heute sind es meist die Zahlen -2_147_483_648 = -2^{31} und $2^{31}-1$ (dies entspricht einer 32-Bit-Darstellung im Zweierkomplement).

Gewisse Kombinationen sind in Ausdrücken ohne zusätzliche Klammerung verboten, insbesondere:
Folgt ein einstelliger Operator nach einem höher- oder gleichrangigen zweistelligen Operator, so müssen Klammern verwendet werden.
Da die Exponentiation nicht assoziativ ist, muss bei Mehrfachverwendung von ** geklammert werden.
Kombinationen aus NOT, ABS und ** dürfen nur mit Klammern verwendet werden.

Verboten sind also: x / -y, x**ABS y, x**y**z, ABS X**Y, NOT ABS(X) > 17.

Prioritäten der gängigen Operatoren in Ada 95:

1. Priorität:	ABS, ** , NOT
2. Priorität:	*, MOD, /, REM
3. Priorität:	+, - (als einstellige Operationen)
4. Priorität:	+, - (als zweistellige Operationen)
5. Priorität:	=, /=, <, <=, >, >=
6. Priorität:	AND, OR, XOR

Bei Integer ist (im Gegensatz zu Boolean) die Verwendung gleichrangiger Operatoren ohne Klammern erlaubt (siehe jedoch vorige Folie). Dies gilt auch für FLOAT.

Empfehlung: In Ada möglichst vollständig klammern!

Der Datentyp real (in Ada: float)

Zugrunde liegende Menge:
$\mathbb{R}$ = Menge der reellen Zahlen mit der üblichen Ordnung.
($\mathbb{R}$ ist die "Vervollständigung" von $\mathbf{Q}$, d.h., die Menge aller Grenzwerte konvergierender Folgen rationaler Zahlen, grafisch dargestellt durch den "Zahlenstrahl".)
Man beachte, dass diese Menge überabzählbar ist. Konkret darstellen können wir immer nur eine endliche Teilmenge.

Nullstellige Operationen: Alle reellen Zahlen, die mit endlich vielen Ziffern und weiteren Symbolen (wie z.B. π, e, das Wurzelzeichen, die Potenzbildung) dargestellt werden können; hierzu gehören insbesondere alle rationalen Zahlen.

Einstellige Operationen:

+	einstelliges Plus (wie Identität, verändert also nichts)
-	einstelliges Minus, Bildung der negativen Zahl, Negation
abs	Absolutbetrag einer Zahl
square	Quadrat einer Zahl (square(x) = x^2)
sqrt	Positive Wurzel (square root) einer Zahl $x \geq 0$
sgn	$\mathbb{R} \rightarrow \mathbf{Z}$ "Signum", also das Vorzeichen einer Zahl:

$$\text{sgn}(x) = \begin{cases} 1, & \text{falls } x > 0 \\ 0, & \text{falls } x = 0 \\ -1, & \text{falls } x < 0 \end{cases}$$

Man kann sgn auch als Abbildung $\mathbb{R} \rightarrow \mathbb{R}$ auffassen.

Hinzu kommen weitere Funktionen wie log (Logarithmus zur Basis 10), ln (Logarithmus zur Basis e), sin (Sinus), cos (Cosinus), tan (Tangens), cot (Cotangens), arcsin (Arcussinus), arccos, arctan, arccot, sinh (Sinus hyperbolicus), cosh, tanh, coth, arcsinh, arccosh, arctanh, arccoth usw.

Einstellige Operationen (Fortsetzung):

Eine wichtige Funktion ist das Abschneiden der Nachkommastellen von reellen Zahlen, wobei man die nächst größere oder die nächst kleinere ganze Zahl erhält. Dies wird in der Mathematik durch die obere und die untere Gaußklammer und durch die "Truncate"-Funktion beschrieben:

$\lceil\ \rceil, \lfloor\ \rfloor : \mathbb{R} \to \mathbb{Z}$ (obere und untere Gaußklammer)

trunc: $\mathbb{R} \to \mathbb{Z}$ ("truncate" = abschneiden) mit
trunc(x) = if $x \geq 0$ then $\lfloor x \rfloor$ else $\lceil x \rceil$ fi.

Hinzu kommt das Runden zur nächsten ganzen Zahl:
round: $\mathbb{R} \to \mathbb{Z}$

Zweistellige Operationen:

+	Addition zweier Zahlen
-	Subtraktion zweier Zahlen, Differenz zweier Zahlen
·	Multiplikation zweier Zahlen (in Programmiersprachen: *)
/	Division (die Division durch 0 ist undefiniert)
exp	Exponentiation ($\exp(x,y) = x^y$; in Programmiersprachen verwendet man hierfür das Zeichen **)

Hinzu kommen alle sechs Vergleichsoperationen "=", "/=", "<", "<=", ">" und ">=" der Funktionalität $\mathbb{R} \times \mathbb{R} \to \mathbb{B}$.

Definition gewisser Operationen:

Die meisten Operationen werden als bekannt vorausgesetzt.

Eine wichtige Funktion ist das Abschneiden der Nachkommastellen von reellen Zahlen, wobei man die nächst größere oder die nächst kleinere ganze Zahl erhält. Dies wird in der Mathematik durch die obere und die untere Gaußklammer beschrieben:

$\lceil\ \rceil, \lfloor\ \rfloor : \mathbb{R} \to \mathbf{Z}$, die obere und untere Gaußklammer sind definiert durch

$$\lceil x \rceil = \text{Min}\{\ z \in \mathbf{Z} \mid x \le z\ \},$$

$$\lfloor x \rfloor = \text{Max}\{\ z \in \mathbf{Z} \mid z \le x\ \}.$$

Es gilt stets $\lfloor x \rfloor \le \lceil x \rceil$
und nur im Falle ganzer Zahlen x ist $\lfloor x \rfloor = x = \lceil x \rceil$.

Definition gewisser Operationen (Fortsetzung):

Will man die Nachkommastellen unabhängig vom Vorzeichen abschneiden, so nimmt man die Funktion trunc.

trunc: $\mathbb{R} \to \mathbf{Z}$ ("truncate" = abschneiden) mit

trunc(x) = if $x \ge 0$ then $\lfloor x \rfloor$ else $\lceil x \rceil$ fi.

Oft benötigt man das Runden zur nächsten ganzen Zahl:
round: $\mathbb{R} \to \mathbf{Z}$ ("round" = runden) mit

für $x \ge 0$: $\text{round}(x) = z \iff z \in \mathbf{Z}$ und $\text{abs}(z-x) < 0.5$ oder
$\text{abs}(z-x) = 0.5$ und $z > x$,

für $x < 0$: $\text{round}(x) = -\text{round}(-x)$.

Z.B.: round(2.4) = 2, round(2.5) = 3, round(-2.4) = -2, round(-2.5) = -3.

Der Datentyp float in der Sprache Ada 95

In Ada müssen die reellen Werte durch endlich viele Zeichen dargestellt werden und zwar entweder als Gleitkommazahl oder als Festkommazahl.

Darstellung als Festkommazahl: Man gibt in Ada die kleinste ("unten") und die größte ("oben") darstellbare Zahl an und legt die feste Differenz "d" zwischen je zwei aufeinander folgenden Zahlen fest. Definitionsschema eines solchen reellen Datentyps:

type fkz is delta d range unten .. oben;

Beispiel: type km is delta 0.001 range -1000.0 .. 1000.0;
Dies legt das Intervall von -1000.0 bis 1000.0 als Wertemenge fest, das mit der Schrittweite 0.001 durchlaufen wird. Dies sind insgesamt 2_000_001 Werte.

Darstellung von Festkommazahlen durch Dezimalziffern

Eine andere Möglichkeit, wie sie zum Beispiel im Geldverkehr gefordert wird, ist die Angabe der Nachkommastellen und der Gesamtzahl an Dezimalziffern. In Ada beschreibt man diesen durch Dezimalziffern festgelegten Festkommazahlentyp wie folgt (a ist eine natürliche Zahl, p eine Zehnerpotenz):

type dzz is delta p digits a;

Der Wertebereich sind alle Zahlen, die mit a Dezimalziffern beschrieben werden können, wobei das Komma so zu setzen ist, dass zwei aufeinander folgende Zahlen den Abstand p voneinander haben.

Beispiel: type kontostand is delta 0.01 digits 8;
Dies legt als Wertemenge genau die Zahlen des Intervalls von -999999.99 bis 999999.99 mit der Schrittweite 0.01 fest.

Darstellung von Gleitkommazahlen

Hier muss man zunächst nichts tun, denn in Ada ist der Typ FLOAT vordefiniert, der der Gleitkommadarstellung entspricht und mindestens eine Genauigkeit von 6 Dezimalziffern besitzen muss; dies entspricht mindestens 21 gültigen Binärstellen pro Zahl. Weiterhin gibt es einen Typ LONG_FLOAT mit mindestens 11 Dezimalziffern Genauigkeit.

Man kann die Genauigkeit vorgeben, z.B. durch

type MY_FLOAT is digits 14;

wodurch stets mindestens 48 Binärstellen mitgeführt werden.

(Faustformel: Wegen $2^{10} \approx 10^3$ kann man von "digits a" auf rund $10 \cdot a/3 + 1$ Binärstellen schließen; "+1" wegen des Vorzeichens.)

Operationen auf den reellen Typen

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

Weiterhin gibt es die sechs Vergleichsoperatoren: =, /=, <, >, <=, >=, deren Ergebnis vom Typ Boolean ist. Man sollte wegen der unvermeidlichen Rundungsfehler die Vergleichsoperatoren = und /= bei reellen Zahlen aber nicht verwenden.

Darüber hinaus gibt es eine Vielzahl weiterer Funktionen in speziellen Paketen für die reellen Datentypen, z.B. in Ada.Numerics.Generic_Elementary_Functions .

Zulässige Operanden

Ada ist "streng typisiert", was insbesondere bedeutet, dass die Datentypen der Operanden in Operationen genau festgelegt sind und beachtet werden müssen. Zum Beispiel ist in

```
X, Z: FLOAT;  K, L: INTEGER;  ...
Z := X + K;
```

die Wertzuweisung nicht zulässig, da der Operator "+" entweder für zwei ganze Zahlen oder für zwei reelle Werte definiert ist, aber nicht für einen Mix hieraus. Man muss zunächst die Datentypen der Operanden anpassen:

```
Z := X + FLOAT(K);     oder       L := INTEGER(X) + K;
```

FLOAT bewirkt die Umwandlung einer ganzen Zahl in die entsprechende reelle Zahl; INTEGER entspricht genau der Funktion round.

Zulässige Operanden

Dagegen ist ausnahmsweise die Multiplikation von ganzen mit reellen Zahlen erlaubt, ebenso die Division einer reellen Zahl durch eine ganze Zahl.

Die Prioritäten sind wie auf Folie 76 angegeben.

Auf weitere Details gehen wir hier nicht ein. Sie sind in jedem Buch über Ada aufgelistet und werden in den Übungen und den Programmierübungen "nebenbei" mitbehandelt.

Hinweis: In Ada 95 sind die Funktionen PRED und SUCC auch auf den real-Datentypen zugelassen. Sie liefern die vorherige bzw. nächst größere darstellbare Zahl.

Typanpassung (oder Typkonversion) findet für eine Integer-Variable X und eine Float-Variable Z zwischen Float und Integer statt durch

Float (X) wandelt X in die entsprechende reelle Zahl um,
Integer (Z) rundet Z zur nächsten ganzen Zahl.

In Ada ist diese "explizite" Typanpassung nur selten erlaubt, zum Beispiel bei "abgeleiteten Typen". Dies sind Typen mit gleichem Wertebereich und Operationen (oder auch Unterbereiche hiervon). Wir erläutern dies nur am Beispiel:


```
type Geld is delta 0.01 digits 15;
        -- Fixpunktdarstellung, 15 gültige Ziffern,
        -- zwei nach dem Dezimalpunkt

type Dollar is new Geld;
type Euro is new Geld;  ...

D1, D2: Dollar; E1, E2: Euro;
D1 := D2; E1 := E1*E2;
```

Geld is ein reller Datentyp. Dollar und Euro sind von Geld abgeleitete Typen ("derived types"). Sie übernehmen alle Eigenschaften von Geld. Da es neue Typen sind, sind E1:= D1 oder D2:=E2 sowie der Ausdruck D1+E1 verboten. Man kann jedoch stets eine explizite Umwandlung D1 := Dollar(E1) oder E1 := Euro(1.08*D1) durchführen.

Auswahlanweisung (Syntax: 5.4 und 3.8.1):

```
case_statement ::= 'case' expression 'is'
                     case_statement_alternative
                   {case_statement_alternative}  'end case' ";"
case_statement_alternative ::=
        'when' discrete_choice_list "=>" sequence_of_statements
discrete_choice_list ::= discrete_choice {"|" discrete_choice}
discrete_choice ::= expression | discrete_range | 'others'
```

Die Auswahlmöglichkeiten müssen disjunkt und vollständig sein!

```
case Tag is
        when Mo | Di => Schulbeginn := "7:45";
        when Mi..Fr => Schulbeginn := "8:35";
        when Sa => Schulbeginn := "9:40";
        when others => Schulbeginn := "entfällt";
end case;
```


Hinweis zur Auswertung von case-Anweisungen:

Prinzipiell wertet Ada sämtliche Möglichkeiten aus, d.h., für alle Auswahlen hinter allen when wird geprüft, ob sie zutreffen. Die case-Anweisung wird nur dann ausgeführt, wenn hierbei genau einmal der Wert true auftritt.

Daher können case-Anweisungen in der Praxis deutlich mehr Zeit benötigen als iterierte if-Anweisungen, mit denen man jede case-Anweisung simulieren kann. Wenngleich aus Gründen der Lesbarkeit die case-Anweisungen oft bevorzugt werden sollten, so sollte man sie dennoch in Programmen, die schnell arbeiten müssen, vermeiden.

exit dient dem Verlassen einer strukturierten Anweisung.

exit_statement ::= 'exit' [*loop*_name] ['when' condition] ";"

Beispiel:

```
Schleife1:
while X>Y loop
        for i in .. loop
                while Z > 0 loop
                        ...
                        exit Schleife1 when Y=Z;
                        ...
                end loop;  ...
        end loop; ...
end loop;
```


Mit dem Sprung goto <Marke> kann man direkt ab der Stelle im Programm, die durch <Marke> markiert ist, fortfahren. Sprünge in Strukturen von außen (Schleifen, then- oder else-Zweige, Blöck, Prozeduren usw.) sind nicht erlaubt.

Beispiel:

```
while X>Y loop
        for i in .. loop
                ...
                if Y=Z then goto weiter; end if;
                ...
        end loop;
        ...
        <<weiter>> null;  -- dies zugleich deklarierendes Auftreten!
end loop;
```

Man kann jeden Algorithmus mit der einseitigen Fallunterscheidung und Sprüngen simulieren. Zum Beispiel lässt sich jede while-Schleife

while β loop C; end loop;

wie folgt darstellen:

<<Schleife>> if β then C; goto Schleife; end if;

Sprünge sind in maschinenahen Sprachen eine wichtige Kontrollstruktur. Sie sind aber für Menschen schwer verständlich und führen daher zu kaum auffindbaren Fehlern. Sie sollten daher nicht benutzt werden.

Abschnitt 3

Grundbegriffe 3:

Blöcke, Sichtbarkeit,

Unterbereiche,

Felder,

Records, variante Records

Block:
Programmeinheit, die aus einem Deklarationsteil und einem darauf folgenden Anweisungsteil besteht.

declare <Deklarationsteil> ;
begin <Folge von Anweisungen>
end;

Der Block, in dem eine Variable (oder ein anderes Objekt) deklariert ist, ist dessen "Lebensdauer". Die Sichtbarkeit ist der Teilbereich der Lebensdauer, in dem sein Name nicht umdefiniert ist.

Die in Blöcken deklarierten Variablen werden im lokalen Speicher des Programms abgelegt.
(An der Tafel erläutert. Vergleiche Folie 115 für ein Beispiel.)

Erinnerung: Aufzählungstyp

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

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

Die Reihenfolge in der Liste gibt zugleich die Anordnung der Elemente an.

Beispiele:

type Wochentage is (Mo, Di, Mi, Do, Fr, Sa, So);

type Farbe is (weiß, gelb, grün, rot, blau, schwarz);

type erste_zehn_Primzahlen is (2,3,5,7,11,13,17,19,23,29)

Unterbereiche, Untertypen

Wenn M eine angeordnete Menge ist und a und b zwei Elemente aus M sind, dann ist

$$a \,..\, b = \{x \in M \mid a \leq x \leq b\}$$

der Unterbereich von a bis b der Menge M.

Gilt $M \neq \emptyset$ und $a \leq b$, so ist $[a \,..\, b] \neq \emptyset$.
Im Falle $a > b$ ist $a \,..\, b = \emptyset$.

Unterbereiche von Zahlen sind Intervalle, z.B.
17 .. 35 oder -23 .. -4 oder -23.6875 .. 0.1875

In den meisten Programmiersprachen sind nur Unterbereiche zulässig, die endlich sind; insbesondere erlaubt man meist keine Unterbereiche reeller Zahlen.

Zur Definition verwendet Ada das Schlüsselwort subtype.

Die Unterbereichsbildung in Ada:

Unterbereiche werden in Ada als "subtype" eines anderen Datentyps bezeichnet. Die allgemeine Form lautet:

subtype <Name> is <Datentyp> [<constraint>]

Die Beschränkung <constraint> gibt den Bereich ("range") von unterer bis oberer Grenze an:
<constraint> ::= range <Ausdruck> .. <Ausdruck>

Alle Operationen des Typs <Datentyp> sind auch für den Unterbereichsdatentyp gültig.

Die Beschränkung darf fehlen; dann hat der Unterdatentyp die gleiche Wertemenge wie der <Datentyp>.

Basistyp

In der Definition

subtype U is T <constraint>

ist U der Unterdatentyp von T und T der Oberdatentyp von U. Der Datentyp, von dem U letztlich abgeleitet wurde, heißt Basisdatentyp zu U. Es sei T ein elementarer Datentyp. Im Falle

subtype U is T <constraint>
subtype V is U <constraint>
subtype W is V <constraint>

ist W Unterdatentyp von V, von U und von T. U ist Obertyp von V und von W. Der Basisdatentyp von U, V und W ist T.

Operationen auf dem Untertyp:

"subtype" bezieht sich ausschließlich auf die Wertebereiche. Alle Operationen des Obertyps werden vom Unterdatentyp übernommen.

In Ada wird jede Variable eines Unterdatentyps stets auch als Variable des Basistyps aufgefasst, so dass man also rechnen kann, als ob man im Obertyp wäre; erst bei der Zuweisung des Ergebnisses wird geprüft, ob das Ergebnis die Beschränkung erfüllt.

(Der Unterdatentyp wird in Ada also als der Basistyp mit zusätzlichen Beschränkungen aufgefasst.)

Beispiele

subtype Monatstage is integer range 1..31;

subtype MinMonatstage is Monatstage range 1..28;

Wenn man nun zu einem Tag vom Typ Monatstage den Tag der folgenden Woche (plus 7) berechnen will, so könnte man schreiben:

```
X, Z: Monatstage;   .....
Z := X + 7;
if Z > 31 then Z := Z-31 end if;
```

Dies führt natürlich zu einem Fehler, falls X mindestens den Wert 25 besaß, da dann Z den Unterbereich verlassen würde. Dagegen führt `if X > 24 then Z := X + 7 - 31; end if;` nicht zu einem Fehler, auch wenn zwischenzeitlich der Bereich verlassen wurde, da Ada den Ausdruck X+7-31 im Obertyp integer berechnet.

Zuweisungen an Variablen der Unterdatentypen sind immer erlaubt, wenn der zuzuweisende Wert im Unterbereich liegt:

subtype Monatstage is integer range 1..31;

subtype MinMonatstage is Monatstage range 1..28;

```
X, Y, Z: Monatstage; A: MinMonatstage; H, J: integer;   .....
H := 50; X := 30; Z := 20;
```

Erlaubt sind dann (beachte: (H*Z) mod 30 +1 ist ein Ausdruck in den ganzen Zahlen, der dort berechnet wird, erst das Ergebnis wird der Variablen Y zugewiesen):

```
A := Z; Y := H-20; Y := (H*Z) mod 30 +1; J := X; A := X-Z;
```

Verboten sind dann (d.h., zu Bereichsüberschreitungen führen):

```
A := X; Z := H; A := Z*Z;
```

Wichtige vordefinierte Unterdatentypen in Ada sind:

subtype natural is integer range 0..integer'Last;

subtype positive is integer range 1..integer'Last;

Unterbereiche kann man auch für den Typ float einführen, sofern die Grenzen Ausdrücke sind, deren Ergebnisse reellwertig sind:

subtype einheitsintervall is float range 0.0 .. 1.0;

Records (Datensätze)

Oft muss man die Elemente verschiedener Mengen zu einem Paar, einem Tripel oder einem n-Tupel zusammenfassen. Gegeben seien die Mengen M_1, M_2, ..., M_n, die zu den Datentypen T_1, T_2, ..., T_n gehören mögen. Dann kann man nach folgendem Schema hieraus den Datentyp T konstruieren, dessen Wertemenge die Menge $M = M_1 \times M_2 \times ... \times M_n$ ist (dies ist zugleich die Ada-Darstellung):

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

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

Beispiele

```
type Monatsname is (Januar, Februar, März, April, Mai, Juni,
        Juli, August, September, Oktober, November, Dezember);

type Datum is record
        Jahr: integer;
        Monat: Monatsname;
        Tag: 1..31;
end record;

heute, ende: Datum;

heute.Jahr := 2003; heute.Monat := April; heute.Tag := 4;

if heute.Jahr = 2003 then
        ende.Jahr := 2003; ende.Monat := April; ende.Tag := 11;
else ende := heute; end if;
```


Beispiele (Fortsetzung)

Komplexe Zahlen:

```
type komplex is record
        Realteil: float;
        Imaginärteil: float;
end record;
```

Rationale Zahlen:

```
type rational is record
        Zähler: integer;
        Nenner: positive;
end record;
P, R: rational; gleich, eins, wurzel2: Boolean; ...

gleich := P.Zähler * R.Nenner = R.Zähler * P.Nenner;
eins := P.Zähler = P.Nenner;
wurzel2 := P.Zähler * P.Zähler = 2 * P.Nenner * P.Nenner;
```

Beispiele (Fortsetzung)

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

```
type hotelformular is record
        Ankunftstag, Abreisetag: Datum;
        Name: array (1..30) of character;
        Geburtsjahr: 1880..2002;
        Wohnort: array (1..30) of character;
        PLZ: array (1..5) of '0'..'9';
        Strasse: array (1..30) of character;
        Hausnummer: positive;
        allein: Boolean;
        männlich: Boolean;
        Raucherzimmer: Boolean;
end record;
```


Felder

Sehr häufig benötigt man n Variablen der gleichen Art, auf die mit einem Index zugegriffen werden kann. Diese fasst man unter einem Namen zu einem "Feld" (oder Vektor) zusammen.

Zur Angabe solcher Felder benötigt man den Datentyp, den jede Variable besitzen soll, und einen Bereich für den Index. Man verwendet das Schlüsselwort "array", um ein Feld zu bezeichnen, gibt dann den Indexbereich an (meist als Unterbereich) und fügt daran den gemeinsamen Datentyp der einzelnen Komponenten an, abgetrennt durch "of". Darstellung in einer Deklaration:

array (<Datentyp des Index>) of <Datentyp>.

Beispiel :

type Wochentage is (Mo, Di, Mi, Do, Fr, Sa, So);

Hierzu bilden wir den Unterbereich:

subtype Arbeitstage is Wochentage range Mo .. Fr ;

und dann folgendes Feld aus fünf Elementen

type Arbeitsbeginn is array (Arbeitstage) of 0 .. 23;

Variablen dieses Typs werden deklariert durch

X: Arbeitsbeginn;

In X kann man für jeden Arbeitstag die Uhrzeit des Arbeitsbeginns speichern. Auf die Komponenten wird mittels X(J) zugegriffen. Dabei durchläuft J den Wertebereich des Datentyps Arbeitstage, d.h. J ∈ {Mo, Di, Mi, Do, Fr}.

Skalarprodukt im $\mathbb{R}^n$

Im Vektorraum $\mathbb{R}^n$ werden Punkte durch ihren Koordinaten beschrieben, also durch den Vektor $(x_1,x_2,...,x_n)$. Der Abstand vom Nullpunkt ist dann die Wurzel aus dem Skalarprodukt mit sich selbst $x_1^2 + x_2^2 + ... + x_n^2$.

Allgemein ist das Skalarprodukt zweier Vektoren $(x_1,x_2,...,x_n)$ und $(y_1,y_2,...,y_n)$ definiert als $x_1 \cdot y_1 + x_2 \cdot y_2 + ... + x_n \cdot y_n$. Folgendes Programm berechnet dieses Skalarprodukt.

Wir nehmen an, die Zahl n, der Vektor x und der Vektor y stehen in dieser Reihenfolge in der Eingabe. Wir lesen diese Werte ein, berechnen dann das Skalarprodukt s und geben es aus. Wir wählen zunächst die Länge der Vektoren n = 80.

```
procedure skalarprodukt1 is
n: constant natural := 80; s: float := 0.0;
x, y: array (1..n) of float;
begin
        for i in 1..n loop  get (X(i));  end loop;
        for i in 1..n loop  get (Y(i));  end loop;
        for i in 1..n loop  s := s + X(i) * Y(i);  end loop;
        put (s);
end;
```


Hinweis: Grundsätzlich strebt man in der Programmierung an, dass alle Werte, die an einer Stelle des Programms verwendet werden, bekannt sind, wenn man sich dort befindet.

Wäre n nicht konstant, so würde das Programm skalarprodukt1 diese Forderung nicht erfüllen, weil dann beim Abarbeiten der Deklarationen an der Stelle " x, y: array (1..n) of float; " der Wert von n noch nicht bekannt ist; denn er wird erst später eingelesen.

Mit dem *Blockkonzept* können wir dieses Problem lösen.

```
procedure skalarprodukt1 is
n: natural; s: float := 0.0;
begin  get(n);
    declare  x, y: array (1..n) of float;
    begin        -- n und s sind hier globale Variable
        if n > 0 then
                for i in 1..n loop  get (X(i));  end loop;
                for i in 1..n loop  get (Y(i));  end loop;
                for i in 1..n loop  s := s + X(i) * Y(i);  end loop;
                put (s);
        end if;
    end;
end;
```


In der Feld-Deklaration

array (<Datentyp des Index>) of <Datentyp>

darf der <Datentyp> der Komponenten erneut eine Felddeklaration sein:

array (<Datentyp des 1.Index>) of
array (<Datentyp des 2.Index>) of <Datentyp>

Dies kürzt man meist ab, z.B. durch
array (<Datentyp des 1.Index, Datentyp des 2.Index>) of <Datentyp>

bzw. man schreibt anstelle von
array (3 .. 25) of array (-4 .. 7) of float

die Deklaration:
array (3 .. 25, -4 .. 7) of float
und nennt dies ein zwei-dimensionales Feld. Wiederholt man dies, so lassen sich d-dimensionale Felder deklarieren.

Beispiele:

type Hvektor is array (integer range 1..100) of float;

Da durch 1..100 bereits der Datentyp integer festgelegt ist, darf man in Ada hierfür auch kürzer schreiben:

type Hvektor is array (1..100) of float;

type Hmatrix is array (1..50) of Hvektor;

type Bundesligatabelle is array (1..18) of fussballverein;

type codierung is array (Character) of Character;

Die iterierte array-Bildung kürzt man wie gesagt ab, indem alle Indexbereiche in eine Definition geschrieben werden:

Statt
type Hvektor is array (1..100) of float;
type Hmatrix is array (1..50) of Hvektor;
type Hvolume is array (1..80) of Hmatrix;

schreibt man also kurz:
type Hvolume is array (1..100,1..50,1..80) of float;

Die Zahl der hierbei verwendeten Indexbereiche heißt die *Dimension* des Feldes. Hvolume ist also ein 3-dimensionales Feld.

Statt Konstanten dürfen in den Indexbereichen auch Ausdrücke verwendet werden, sofern jeder Ausdruck in dem Augenblick, in dem die Deklaration erreicht wird, auch tatsächlich ausgerechnet werden kann.

Beispiele (wobei f und g Funktionen vom Ergebnistyp integer sein sollen):
type Hvektor is array (1..N, x..I*J) of Boolean;
type ausschnitt is array (f(unten)..g(oben)) of integer;
type sonstiges is array ((oben-unten) / 2 .. f(g(unten*oben))) of float;

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

Unspezifizierte Feldgrenzen sind zulässig. Man trägt dann nur den Datentyp des Index ein und fügt ein
range <> ("<>" sprich "box")
hinzu. Solche unspezifizierten array-Deklarationen *muss* man bei ihrer Verwendung spezifizieren, z.B. bei der Deklaration von Variablen und beim konkreten Parameteraufruf.

Beispiele:
type text is array (natural range <>) of Character;
type raster is array (integer range <>, integer range <>) of Boolean;
type ganzzahlvektor is array (integer range <>) of integer;

Bei der Deklaration von Variablen gibt man dann die konkreten Grenzen statisch oder dynamisch vor, z.B.:
X: ganzzahlvektor (-10..10) oder Y: ganzzahlvektor (I..J)

In Ada sind weitere Operationen mit arrays verbunden (in Ada werden sie als Attribute bezeichnet):

Die Konstante "Range(i)" gibt zu jeder Feld-Variablen den Indexdatentyp ihrer i-ten Dimension an. Analog bezeichnet "Length(i)" die Anzahl der Elemente des Indexdatentyps in der i-ten Dimension. Mittels First(i) und Last(i) erhält man das erste bzw. letzte Element des Indexdatentyps der i-ten Dimension. Hat man nur eine Dimension, so darf man "(1)" weglassen.

Beispiel:

```
type Codes is (character range 'A'..'Z', 1..32) of
                                    character range 'B'..'z';
W: Codes;
```

Dann gilt:

W'Range(2) = 1..32, W'Length(1) = 26,
W'First(1) = 'A' , W'Last(2) = 32

Weiterhin darf man Feldvariablen, die *vom gleichen Datentyp* sind, einander zuweisen.

```
type vektor is array (1..100) of character;

X, Y: vektor;

....

X := Y;
```

Bedeutung dieser Zuweisung: Der gesamte Inhalt von Y wird komponentenweise in die Variable X kopiert.

In Ada gibt es eine klare restriktive Regel, wann zwei Variablen A und B vom gleichen Datentyp sind: Sie müssen den gleichen "benannten" Datentyp haben, d.h., sie besitzen entweder den gleichen vordefinierten Datentyp (Boolean, character, integer, natural, positive, float usw.) oder es gibt eine Typ- oder Untertyp-Definition mit einem Namen und beide Variablen sind mit diesem Namen deklariert worden.

"Benannt" bedeutet, dass dem Datentyp ein Name per Datentypdefinition zugewiesen ist. In Ada kann man nämlich Feld-Variablen auch unbenannt deklarieren:

X: array (1..100) of character;

Solch eine Deklaration wird stets als neue, noch nicht vorhandene Feld-Datentypdefinition aufgefasst (s.u.).

Gleichen Datentyp haben z.B.:

```
type zehnziffern is array(1..10) of 0..9;
A, B: integer; K, L: positive;
X, Y: zehnziffern;
```

Dagegen haben *nicht* den gleichen Datentyp in Ada:

```
D: zehnziffern; E: array (1..10) of 0..9;
F, G: array(1..10) of 0..9;
H, I: 0..50;
```

denn Ada fasst H, I: 0..50 auf als die Deklarationsfolge

```
H: 0..50;
I: 0..50;
```

und diese beiden Datentypen sind verschieden, da sie keinen vom Benutzer vergebenen Namen besitzen.

Weiteres Beispiel:

```
type Codes is (character range 'A'..'Z', 1..32) of
                    character range 'B'..'z';
V, W: Codes;  R, S: character range W'Range(1);
begin
for I in W'Range(1) loop
    for K in W'Range(2) loop
        W(I,K) := val(pos(I) + K);  end loop;  end loop;
V := W;      -- dies bewirkt, dass das ganze Feld W nach V kopiert wird
R := W('C',17);   -- es wird geprüft, ob das Ergebnis im Unterdatentyp liegt
S := W(R,32);     -- dies wird daher zu einem Fehlerabbruch führen
...
end;
```


Beispiel: Intervallschachtelung (oder binäre Suche)

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

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

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

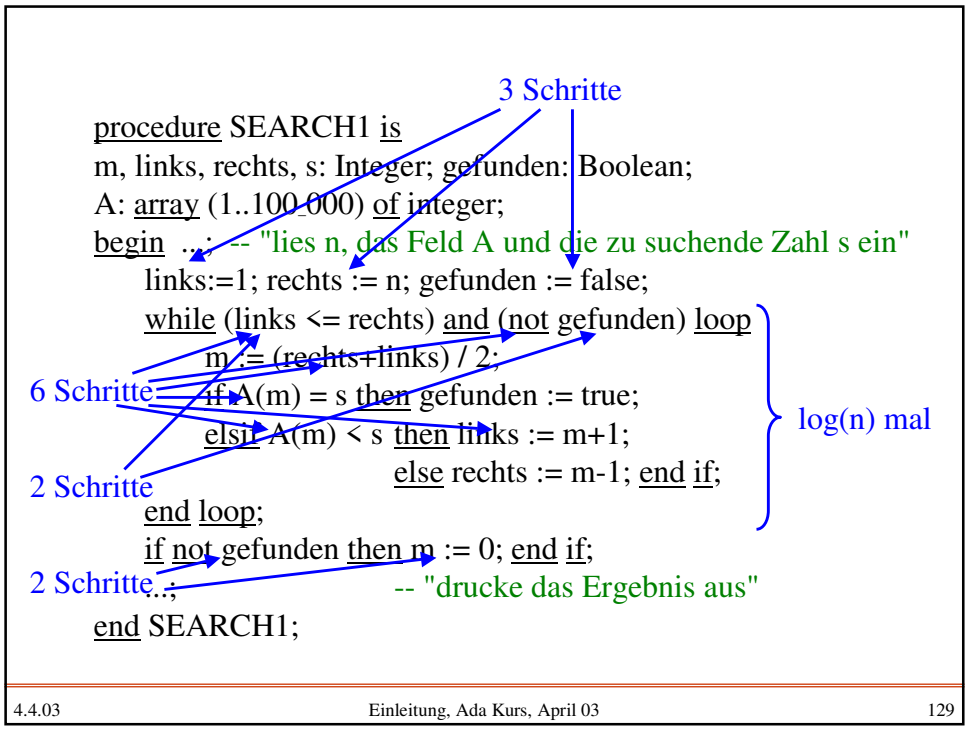
Ein schnelleres Verfahren ist die Intervallschachtelung: Teste, ob s genau in der Mitte A(mitte) von A liegt, falls nein und es ist $A(\text{mitte}) < s$, suche rechts von der Mitte weiter, sonst links.

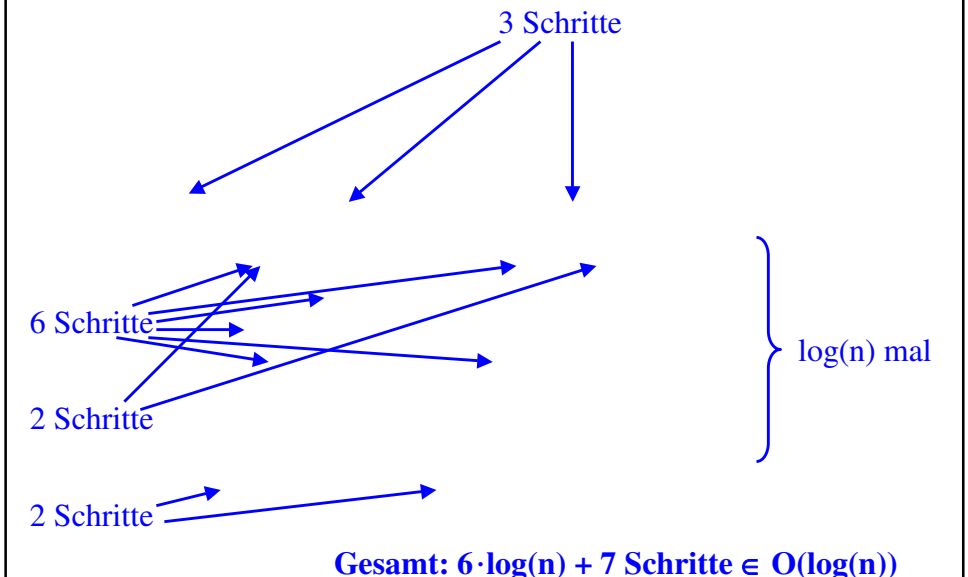
Programm 1 (in Ada): wir setzen hier nmax = 100_000.

```
procedure SEARCH1 is
m, links, rechts, s: Integer; gefunden: Boolean;
A: array (1..100_000) of integer;
begin  ...;  -- "lies n, das Feld A und die zu suchende Zahl s ein"
     links:=1; rechts := n; gefunden := false;
     while (links <= rechts) and (not gefunden) loop
          m := (rechts+links) / 2;
          if A(m) = s then gefunden := true;
          elsif A(m) < s then links := m+1;
                         else rechts := m-1; end if;
     end loop;
     if not gefunden then m := 0; end if;
     ...;                    -- "drucke das Ergebnis aus"
end SEARCH1;
```


Wie lange dauert es, bis dieser Algorithmus spätestens endet?

Hierzu zählen wir die Wertzuweisungen und Bedingungen, die im ungünstigsten Fall ausgerechnet werden müssen.
In diesem Sinne dauert die Durchführung der 3 Anweisungen
links:=1; rechts := n; gefunden := false;
genau 3 Schritte.
In der Schleife werden maximal 6 Schritte benötigt, nämlich je einer für die Bedingungen
(links <= rechts), (not gefunden), A(m) = s und A(m) < s
und je einer für zum Beispiel die Wertzuweisungen
m := (rechts+links) / 2; und links := m+1;
Wie oft wird die Schleife durchlaufen? Das Intervall von links bis rechts halbiert sich mindestens in jedem Schritt, folglich muss nach log(n) Schleifendurchläufen Schluss sein.


3 Schritte
procedure SEARCH1 is
m, links, rechts, s: Integer; gefunden: Boolean;
A: array (1..100_000) of integer;
begin ...; -- "lies n, das Feld A und die zu suchende Zahl s ein"
links:=1; rechts := n; gefunden := false;
while (links <= rechts) and (not gefunden) loop
m := (rechts+links) / 2;
6 Schritte
if A(m) = s then gefunden := true;
elsif A(m) < s then links := m+1;
else rechts := m-1; end if;
log(n) mal
2 Schritte
end loop;
if not gefunden then m := 0; end if;
2 Schritte ...; -- "drucke das Ergebnis aus"
end SEARCH1;


3 Schritte
6 Schritte
2 Schritte
log(n) mal
2 Schritte
Gesamt: 6·log(n) + 7 Schritte ∈ O(log(n))

Der ungünstigste oder schlechteste Fall (der "worst case") kann auch tatsächlich eintreten, wenn nämlich das gesuchte Element s nicht im Feld A enthalten ist. Wir sagen: Die *uniforme worst case Zeitkomplexität t(n)* des Programms SEARCH1 lautet

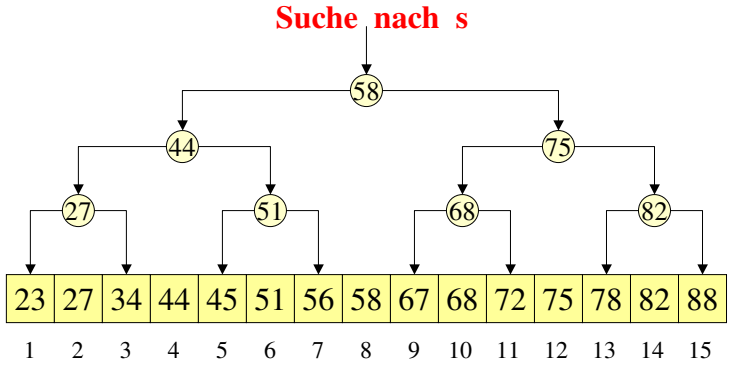
$t(n) = 6 \cdot \log(n) + 7$.

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

Was ist der beste Fall? In diesem Fall wird s im ersten Durchgang der while-Schleife gefunden, d.h. nach 13 Schritten ist der Teil, der das Element s finden soll, beendet.

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

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



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

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

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

Daher braucht man im Mittel:

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

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

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

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

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

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

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

Somit beträgt die *average case Zeitkomplexität* der Intervallschachtelung ziemlich genau $6\cdot\log(n) + 1$ Schritte, also nur einen Schleifendurchlauf weniger als im schlechtesten Fall.

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

Man entscheide erst ganz am Ende, ob A(m) = s gewesen ist; hierzu muss man im Falle A(m) < s in dem rechten Teil des Feldes weitersuchen (links:=m+1), anderenfalls im linken Teil einschließlich des gerade betrachteten Feldes m (rechts:=m).

Programm 2 (in Ada): im Mittel besser als SEARCH1

```
procedure SEARCH2 is
m, links, rechts, s: Integer; gefunden: Boolean;
A: array (1..100_000) of integer;
begin  ...;  -- "lies n, das Feld A und die zu suchende Zahl s ein"
     links:=1; rechts := n;
     while (links < rechts) loop
          m := (rechts+links) / 2;
          if A(m) < s then links := m+1;
                          else rechts := m; end if;
     end loop;
     gefunden := A(m) = s;
     if not gefunden then m := 0; end if;
     ...;                          -- "drucke das Ergebnis aus"
end SEARCH2;
```

Weisen Sie nun nach, dass für diese Version SEARCH2 gilt:

Die *uniforme worst case Zeitkomplexität* beträgt $5 + 4 \cdot \log(n)$; die *uniforme average case Zeitkomplexität* besitzt genau den gleichen Wert.

Hierbei ist n die Zahl der Elemente im array.

Diverse Suchverfahren auf Feldern betrachten wir im Teil "Einführung in die Informatik II" der Grundvorlesung.

Vereinigung disjunkter Typen

Gegeben seien die Mengen M_1, M_2, ..., M_n, die zu den Datentypen T_1, T_2, ..., T_n gehören. Dann kann man nach folgendem Schema hieraus den Datentyp T konstruieren, dessen Wertemenge die disjunkte Vereinigung dieser Mengen

$$M = M_1 \cup M_2 \cup ... \cup M_n$$

ist. Er lautet in Ada:

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

Statt des "index", der hier aus der Menge {1, 2, ..., n} ist, kann auch ein anderer Name und ein anderer Datentyp gewählt werden. Dieses auswählende Element heißt "Diskriminator".

Die Menge ist eine disjunkte Vereinigung, weil durch den Diskrimator "index" genau eine Menge ausgewählt wird, die Mengen also als verschieden angesehen werden, auch wenn verschiedene M_i gleiche Elemente enthalten sollten.

Natürlich können im record noch weitere Komponenten enthalten sein, so dass man bei der disjunkten Vereinigung in der Programmierung von einem "varianten Anteil" ("variant part") spricht. An folgendem Beispiel wird dies klar.

Bei Fahrzeugen kann man an verschiedenen Dingen interessiert sein: Wir nehmen an, dass für alle Fahrzeuge die Länge, die Breite, die Höhe und die Fahrzeugnummer vorliegen müssen, bei Bussen die Zahl der Sitzplätze, bei Lastkraftwagen die Größe der Ladefläche in qm und bei einem PKW die Zahl der Airbags. Dies führt zu folgendem Datentyp:

```
type mass is delta 0.001 range 0.0 .. 50.0;
type art is (PKW, LKW, Bus);
type fahrzeug (auto: art) is record
    länge, breite, höhe: mass;
    nummer: positive;
    case auto is
        when Bus => sitzplätze: 8 .. 150;
        when LKW => ladefläche: positive;
        when PKW => airbagzahl: 0..10;
end record;
```