



C O M P A S
Common Pascal Language System
Version 2.1

P R O G R A M M E R I N G S M A N U A L

Mangfoldiggørelse af denne manual - helt eller delvis - er ifølge loven om ophavsret forbudt, hvis der ikke foreligger en skriftlig tilladelse fra Poly-Data microcenter ApS. Dette gælder enhver form for mangfoldiggørelse, f.eks. fotokopiering, eftertryk, mikrofotografering, etc.

Copyright (C) 1982,1983
Poly-Data microcenter ApS
Strandboulevarden 63
2100 København Ø

INDHOLDSPORTEGNELSE

0	Indledning	5
1	Sprogets grundlæggende elementer	6
1.1	Grundlæggende symboler	6
1.2	Reserverede ord og standard identificere	6
1.3	Separatorer	7
1.4	Programlinier	7
2	Selvdefinerede sprogelementer	8
2.1	Identificere	8
2.2	Tal	8
2.3	Strenge	8
2.4	Kommentarer	9
2.5	Compilerdirektiver	9
3	Standard skalare typer	10
3.1	Integer typen	10
3.2	Real typen	10
3.3	Boolean typen	10
3.4	Char typen	10
3.5	Byte typen	11
4	Programmer	12
4.1	Programoverskriften	12
4.2	Erklæringsdelen	12
4.2.1	Labelerklæringer	12
4.2.2	Konstantdefinitioner	13
4.2.3	Typedefinitioner	13
4.2.4	Variabelerklæringer	14
4.2.5	Procedure- og funktionserklæringer	14
4.3	Sætningsdelen	15
5	Udtryk	16
5.1	Operatorer	16
5.1.1	Negationsoperatoren	16
5.1.2	NOT operatoren	16
5.1.3	Multiplicerende operatorer	17
5.1.4	Adderende operatorer	17
5.1.5	Relationelle operatorer	17
5.2	Funktionskald	17
6	Sætninger	18
6.1	Simple sætninger	18
6.1.1	Tilskrivningssætninger	18
6.1.2	Proceduresætninger	18
6.1.3	GOTO sætninger	19
6.1.4	Tomme sætninger	19
6.2	Strukturerede sætninger	19
6.2.1	Sammensatte sætninger	19
6.2.2	Betingede sætninger	19

INDEHOLDSFORTEGNELSE

6.2.2.1 IF sætninger	20
6.2.2.2 CASE sætninger	20
6.2.3 Repetitionssætninger	21
6.2.3.1 WHILE sætninger	21
6.2.3.2 REPEAT sætninger	21
6.2.3.3 FOR sætninger	22
7 Skalare typer og delintervaller	23
7.1 Skalare typer	23
7.2 Delintervaller	24
7.3 Typekonvertering	24
7.4 Værdikontrol	25
8 Strenge	26
8.1 Strengtyper	26
8.2 Strengudtryk	26
8.3 Strengtilskrivninger	27
8.4 Strengfunktioner og procedurer	27
8.5 Strenge og tegn	29
9 Arraystrukturer	30
9.1 Brug af arraystrukturer	30
9.2 Flerdimensionale arraystrukturer	31
9.3 Prædefinerede arraystrukturer	31
9.3.1 mem arraystrukturen	32
9.3.2 port arraystrukturen	32
9.4 Tegn-arrays	32
10 Poster	33
10.1 Brug af poster	33
10.2 WITH sætninger	34
10.3 Variant-del i poster	35
11 Mængder	37
11.1 Mængdetyper	37
11.2 Mængdeudtryk	38
11.2.1 Mængdeangivere	38
11.2.2 Mængdeoperatorer	38
11.3 Mængdetilskrivninger	39
12 Typeangivne konstanter	40
12.1 Typeangivne konstanter af simple typer	40
12.2 Strukturerede konstanter	40
12.2.1 Arraykonstanter	40
12.2.2 Postkonstanter	41
12.2.3 Mængdekonstanter	42

INDHOLDSFORTEGNELSE

13 Filer	43
13.1 Filtyper	43
13.2 Operationer på filer	43
13.3 Textfiler	46
13.3.1 Operationer på textfiler	46
13.3.2 Logiske I/O enheder	48
13.3.3 Standardfiler	49
13.4 Typeløse filer	50
13.4.1 Operationer på typeløse filer	51
13.5 I/O kontrol	52
14 Pointere	54
14.1 Pointertyper	54
14.2 Brug af pointere	54
14.3 Pointere og integers	57
14.4 Oversigt over pointerrutiner	57
15 Procedurer og funktioner	59
15.1 Parametre	59
15.2 Procedurer	60
15.2.1 Procedureerklæringer	60
15.2.2 Standardprocedurer	61
15.3 Funktioner	62
15.3.1 Funktionserklæringer	62
15.3.2 Standardfunktioner	64
15.3.2.1 Aritmetiske funktioner	64
15.3.2.2 Skalare funktioner	65
15.3.2.3 Konverteringsfunktioner	65
15.3.2.4 Andre standardfunktioner	65
15.4 FORWARD specifikationer	66
15.5 EXTERNAL specifikationer	67
15.6 Strenge som var-parametre	67
15.7 Typeløse var-parametre	68
15.8 Absolutive procedurer og funktioner	69
16 Indlæsning og udlæsning	70
16.1 read proceduren	70
16.2 readln proceduren	71
16.3 write proceduren	72
16.4 writeln proceduren	73
17 Include-filer	74
18 Kædning af programmer	75
19 In-line maskinkode	77
20 CP/M funktionskald	79
21 Brugerdefinerede I/O drive	80

INDHOLDSFORTEGNELSE

22 Interne dataformater	82
22.1 Grundlæggende datatyper	82
22.1.1 Skalarer	82
22.1.2 Reals	82
22.1.3 Strenge	82
22.1.4 Mængder	83
22.1.5 Fil interface blokke	83
22.1.6 Pointere	85
22.2 Datastrukturer	85
22.2.1 Arraystrukturer	85
22.2.2 Poster	85
22.2.3 Diskfiler	85
22.2.3.1 Textfiler	85
22.2.3.2 Random access filer	86
22.3 Parametre	86
22.3.1 Var-parametre	86
22.3.2 Value-parametre	86
22.3.2.1 Skalarer	86
22.3.2.2 Reals	87
22.3.2.3 Strenge	87
22.3.2.4 Mængder	87
22.3.2.5 Pointere	87
22.3.2.6 Arraystrukturer og poster	88
22.4 Funktionsresultater	88
23 Lagerorganisering	89
23.1 Memory maps	89
23.2 Heopen og stakkene	90
24 Interruptstyring	92
25 Forskelle mellem COMPAS og Standard Pascal	94
A Oversigt over standard procedurer og funktioner	95
B Oversigt over operatorer	98
C Oversigt over compilerdirektiver	99
D ASCII tegntabel	101
E COMPAS syntax	102
F I/O fejl	108
G Kørselsfejl	110
H Compilerfejl	111

Kapitel 0

Indledning

Formålet med denne manual er at definere COMPAS Pascal programmeringssproget. Manualen er ikke ment som en lærebog, men den er dog så vidt som muligt ordnet således, at sprogets begreber introduceres i en logisk rækkefølge. Begyndere i programmering anbefales at supplere manualen med en lærebog i Pascal. Når man er blevet fortrolig med sproget, findes der bagest i manualen et par appendices, der er egnede til hurtige opslag.

COMPAS Pascal er en udvidet version af Standard Pascal, der er beskrevet af K. Jensen og N. Wirth i bogen "Pascal User Manual and Report". COMPAS Pascal følger nøje definitionen af Standard Pascal - dog findes der enkelte forskelle, hvilket beskrives nærmere i Kapitel 25. Udvidelser, der gives i COMPAS Pascal, omfatter blandt andet:

- o Dynamiske strenge
- o Random-access filbehandling
- o Strukturerede konstanter
- o Fri ordning af elementer i erklæringsdele
- o Alfanumeriske labels
- o Kontroltegn i strengkonstanter
- o Typekonverteringsfunktioner
- o Kædning af programmer med fælles variable
- o Include filer
- o Logiske operationer på heltal
- o Bit/byte operationer
- o Hexadecimale heltalskonstanter
- o Direkte adgang til lager og dataporte
- o Absolut adresserede variable
- o In-line maskinkode

Desuden findes nye standardprocedurer og funktioner, der yderligere forbedrer sproget. Alle udvidelser er karakteriserede ved, at de enten anses for at være nødvendige for COMPAS Pascals egnethed som et generelt programmeringssprog, eller ved, at de giver væsentlige fordele fremfor det uudvidede sprog. Udvidelserne følger i høj grad de logiske anskuelser, der ligger til grund for Standard Pascal.

COMPAS Pascal systemet og den tilhørende dokumentation er forfattet af Anders Hejlsberg.

Kapitel 1

Sprogets grundlæggende elementer

1.1 Grundlæggende symboler

De grundlæggende symboler i COMPAS Pascal er inddelt i tre grupper - Bogstaver, talcifre og specialtegn:

Bogstaver: A til Z, a til z og underscore '_'

Talcifre: 0 1 2 3 4 5 6 7 8 9

Specialtegn: + - * / = < > () E A æ å . , ; : ' ^ @ \$

Der skelnes ikke mellem store og små bogstaver. Bemærk at de danske tegn E, Ø, A, æ, ø, og å ikke er klassificeret som bogstaver, og, at disse derfor ikke må indgå i identificere. Grunden hertil er, at disse tegn ikke er bogstaver i det internationale ASCII tegnsæt, men i stedet specielle tegn, hvilket illustreres af den nedenfor viste tabel:

Dansk tegn: E Ø A æ ø å

ASCII tegn: [\] { | }

Visse operatorer og separatorer skrives som en sammensætning af to specialtegn:

1. <> <= >= := ..

2. (. og .) kan bruges i stedet for E og A

3. (* og *) kan bruges i stedet for æ og å

Som vist ovenfor er det tilladt at bruge sammensatte symboler i stedet for de danske tegn - dette er bestemt at anbefale, da det gør programtekster mere overskuelige.

1.2 Reserverede ord og standard identificere

Reserverede ord er en integreret del af COMPAS Pascal, og de kan ikke omdefinieres. Således kan de reserverede ord ikke anvendes som brugerdefinerede identificere. De reserverede ord er:

AND	ARRAY	AT
BEGIN	CASE	CODE
CONST	DIV	DO
DOWNTO	ELSE	END
EXOR	EXTERNAL	FILE
FOR	FORWARD	FUNCTION
GOTO	IF	IN
LABEL	MOD	NIL
NOT	OF	OR
OTHERWISE	PACKED	PROCEDURE
PROGRAM	RECORD	REPEAT
SET	SHL	SHR
STRING	THEN	TO
TYPE	UNTIL	VAR
WHILE	WITH	

I denne manual skrives alle reserverede ord med store bogstaver. COMPAS Pascal definerer også et antal standard identificere, der ikke er reserverede, men som er navne på prædefinerede konstanter, typer, variable, procedurer og funktioner. Disse identificere kan eventuelt overskrives med brugerens egne definitioner:

aoaddr	arctan	assign
aux	bdos	bdosb
bios	biosb	blockread
blockwrite	boolean	buflen
byte	chain	char
chr	ciaddr	close
creol	clreos	coaddr
con	concat	copy
cos	csaddr	delete
eof	eoln	erase
execute	exp	false
fill	flush	frac
gotoxy	hi	hptr
input	insert	int
integer	iores	kbd
keypress	len	length
ln	lo	loaddr
lst	mark	maxint
mem	memavail	move
new	odd	ord
output	pi	port
pos	position	pred
ptr	pwrten	random
randomize	read	readln
real	release	rename
reset	rewrite	round
rptr	seek	sin
size	sptr	sqr
sqrt	str	succ
swap	text	trm
true	trunc	uiaddr
uoaddr	usr	val
write	writeln	

I denne manual skrives alle standard identificere med små bogstaver.

1.3 Separatorer

Blanktegn, linieskift og kommentarer anses for separatorer. Der skal altid være mindst en separator mellem to Pascal elementer.

1.4 Programlinier

En programlinie må højst være 127 tegn lang. Hvis en programlinie er længere end 127 tegn, bliver de overskydende tegn ignoreret.

Kapitel 2**Selvdefinerede sprogelementer****2.1 Identificere**

Identificere bruges til at navngive labels, konstanter, typer, variable, procedurer og funktioner. En identifier består af et bogstav efterfulgt af et vilkårligt antal bogstaver eller talcifre. Eksempler:

```
COMPAS      bredde     rod3      antal_tegn
```

Bemærk at COMPAS Pascal ikke skelner mellem små og store bogstaver i identificere.

2.2 Tal

Tal er konstanter af typen integer (heltal) eller real (reelle tal). Heltalskonstanter udtrykkes enten i decimal eller hexadecimal notation. Hexadecimal notation vælges ved at skrive et '\$' tegn foran tallet. Det decimale heltalsområde er -32768 til 32767 og det hexadecimale talområde er \$0000 til \$FFFF. Eksempler på heltalskonstanter:

```
1      3741     -3      $20      $E7B9
```

Talområdet for reelle tal er 1E-38 til 1E+38 med en mantisse på op til 11 betydende cifre. Eksponentiel notation vælges ved at efterfølge mantissen med et 'E', igen efterfulgt af heltallet, der udgør eksponenten. En heltalskonstant kan altid anvendes i stedet for en reel konstant. Eksempler på reelle konstanter:

```
1.0      0.025      5E10      1E-5      -3.7833564719E+12
```

Talkonstanter må ikke indeholde separatorer.

2.3 Strenge

En tekststreng er en følge af tegn omsluttet af enkelte anførelsestegn. Anførelsestegnet kan indgå i strengen, hvis det skrives dobbelt. En streng, der indeholder et enkelt tegn, er en konstant af typen char. En streng, der indeholder flere end et tegn, er en konstant af typen ARRAY(.1..n.) OF char. Strenge, uanset deres længde, er altid kompatible med alle STRING typer. Eksempler på strenge:

```
'COMPAS'      'That''s all folks'      ';'      ''      ''
```

Bemærk det sidste eksempel - anførelsestegnene omslutter ingen (nul) tegn og angiver dermed den tomme streng.

I COMPAS Pascal er det også muligt at lade kontroltegn inddgå i strengkonstanter. Der findes to notationer for kontroltegn: Et snabel-a (@) efterfulgt af en heltalskonstant angiver et tegn af denne ASCII værdi, og en pil (^) efterfulgt af et tegn, med ASCII værdi n, angiver kontroltegnet med ASCII værien n-64. Nogle eksempler på kontroltengsangivelser:

```
@13      @127      @$1C      ^M      ^G      ^L      ^Z
```

Flere kontroltegn kan sammensættes til strenge ved at skrive dem umiddelbart efter hinanden:

```
@13@10    @27@61@32@32    ^M^J    ^G^G^G    ^z@31@31^J
```

De ovenfor viste strenge indeholder henholdsvis to, fire, to, tre, og fire tegn. Kontroltegn og tekststrenge kan også sammen-
sættes:

```
'FEJL '^G^G^G' PRØV IGEN'    @27'Æl;lh'    'Linie 1'^M^J
```

De her viste strenge indeholder henholdsvis atten, seks, og ni tegn.

2.4 Kommentarer

En kommentar kan indsættes hvorsomhelst i et program mellem to sprogelementer. Kommentarer skal omsluttes af symbolerne (*) og *) eller af symbolerne æ og å. Et eksempel:

```
(* Dette er en kommentar *)
```

2.5 Compilerdirektiver

Visse af COMPAS Pascal compilerens faciliteter styres ved hjælp af compilerdirektiver. Et compilerdirektiv er faktisk et special-tilfælde af en kommentar, og kan derfor anvendes når som helst en kommentar er tilladt. En liste af compilerdirektiver startes med et \$ tegn umiddelbart efter den indledende kommentarparentes. Selve listens syntaks afhænger af, hvilke direktiver der angives. En fuld beskrivelse af hvert compilerdirektiv følger senere i manualen, og en oversigt findes i Appendix C. Nogle eksempler på compilerdirektiver:

```
(*$I-*)      ($$A+,R-,B+*)      ($$I MINMAX.LIB*)      ($$W5*)
```

Bemærk, at der ikke må være blanktegn imellem (\$ symbolerne eller umiddelbart efter \$ tegnet.

Kapitel 3

Standard skalare typer

En datatype er afgørende for, hvilke værdier en variabel kan antage. I et program skal enhver variabel være forbundet med een og kun een datatype. Blandt datatyperne findes visse grundlæggende typer, som normalt kaldes standard skalare datatyper. Disse er integer, real, boolean, char, og byte.

3.1 Integer typen

En integer er et heltal i området -32768 til 32767, eller i området \$0000 til \$FFFF. Variable af typen integer optager to bytes i datalageret.

Overløb ved heltalsoperationer rapporteres ikke. Alle mellemresultater i et heltalsudtryk skal holdes inden for det lovlige heltalsområde, da resultatet ellers bliver ukorrekt. For eksempel vil udtrykket $4000 * 50 \text{ DIV } 25$ ikke give 8000, da multiplikationen resulterer i et overløb.

3.2 Real typen

Talområdet for reelle tal er 1E-38 til 1E+38, med en mantisse på 11+ betydende cifre. Variable af typen real optager seks bytes i datalageret.

I tilfælde af overløb i et aritmetisk udtryk af typen real, stopper programmet og udskriver en fejlmeldelse. I tilfælde af underløb returnerer operationen nul (0.0).

Selvom real typen er en skalar datatype, kan den ikke altid bruges i de samme tilfælde som andre skalare typer: Standardfunktionerne pred og succ tillader ikke argumenter af typen real, indextypen i et array må ikke være real, grundtypen i en mængde må ikke være real, kontrolvariablen i en FOR sætning må ikke være af typen real, nøgleudtrykket i en CASE sætning må ikke være af typen real, og et delinterval med grundtype real er ikke tilladt.

3.3 Boolean typen

Boolean variable kan antage to værdier, sand eller falsk, givet ved standardkonstanterne true og false. Boolean typen er defineret således at false < true. En boolean variabel optager en byte i datalageret.

3.4 Char typen

En char værdi er et tegn i ASCII tegnsættet. Tegn er ordnet efter deres ASCII værdi – således gælder der, for eksempel, at 'A' < 'B'. Den ordinale værdi (ASCII værdien) af et tegn må være mellem 0 og 255, dvs. fra @0 til @255. En char variabel optager en byte i datalageret.

3.5 Byte typen

Typen byte er faktisk et delinterval af typen integer, defineret som TYPE byte = 0..255. Således er byte variable kompatible med integer variable, dvs. bytes og integers kan blandes i udtryk, og bytes kan tilskrives værdier af typen integer. En variabel af typen byte optager en byte i datalagret.

Kapitel 4

Programmer

Ethvert program består af en programoverskrift efterfulgt af en programblok. Programblokken indeholder en erklæringsdel, hvori programmets elementer (labels, konstanter, typer, variable, procedurer og funktioner) erklæres, og en sætningsdel, hvori programmets handlinger opskrives i form af programsætninger.

4.1 Programoverskriften

Programoverskriften tildeler programmet et navn, og angiver eventuelt identificere for en eller flere af de I/O kanaler, programmet bruger. Listen af identificere for I/O kanaler er omsluttet af parenteser, og hver enkelt identifier er adskilt fra de omkringstående af kommaer. Eksempler på programoverskrifter:

```
PROGRAM kvadrater;
PROGRAM lommeregner(input,output);
PROGRAM sorterings(printer,disk);
```

I COMPAS Pascal har programoverskriften ingen reel betydning for programmet, og den kan derfor udelades efter behag.

4.2 Erklæringsdelen

I programblokkens erklæringsdel erklæres de identificere, der bruges i blokken og andre blokke inden i denne. Erklæringsdelen er underdelt i fem forskellige afsnit:

1. Labelerklæringer
2. Konstantdefinitioner
3. Typedefinitioner
4. Variabelerklæringer
5. Procedure- og funktionserklæringer

I COMPAS Pascal må hvert afsnit skrives et vilkårligt antal gange, og afsnittenes rækkefølge er uden betydning. Standard Pascal angiver dog, at hvert afsnit kun må forekomme nul eller en gang, og kun i den ovennævnte rækkefølge.

4.2.1 Labelerklæringer

Enhver sætning i et program kan mærkes med en label ved at skrive labelens identifier efterfulgt af et kolon foran sætningen. Det kræves imidlertid, at sådanne labels er erklæret i en labelerklæringsdel, før de bruges. En labelerklæringsdel indledes med det reserverede ord LABEL, efterfulgt af en liste af labelidentificere, adskilt af kommaer, og afsluttes med et semikolon. Et eksempel:

```
LABEL 100,error,999,stop;
```

Bemærk, at Standard Pascal foreskriver, at labels kun må være positive heltal mellem 0 og 9999, medens COMPAS Pascal tillader både tal og identificere.

4.2.2 Konstantdefinitioner

En konstantdefinition erklærer en identifier, og tildeler den en fast værdi. En konstantdefinitionsdel indledes med det reservede ord CONST, efterfulgt af en række konstanttilskrivninger, adskilt af semikoloner. Hver konstanttilskrivning består af en identifier efterfulgt af et lighedstegn og en konstant. Konstanter er enten tal eller strenge, som beskrevet i afsnit 2.2 og 2.3. Et eksempel:

```
CONST
  antal = 45;
  max = 193.158;
  min = -max;
  navn = 'Michael';
  nylinie = ^M^J;
```

De følgende konstanter er prædefinerede i COMPAS Pascal:

pi	real	3.1415926536E+00.
false	boolean	Sandhedsværdien "falsk".
true	boolean	Sandhedsværdien "sand".
maxint	integer	32767.

En konstantdefinitionsdel kan også definere typeangivne konstanter. Dette beskrives i kapitel 12.

4.2.3 Typedefinitioner

En datatype i Pascal kan enten beskrives direkte i en variabelerklæring, eller den kan referes til via en typeidentifier. En del prædefinerede typeidentificere findes som standard (integer, real, boolean, etc.), men udover disse kan brugeren definere nye typer i en typedefinitionsdel. En sådan indledes med det reservede ord TYPE, efterfulgt af en række typetilskrivninger, adskilt af semikoloner. Hver typetilskrivning består af en typeidentifier efterfulgt af et lighedstegn og en datatype. Et eksempel:

```
TYPE
  heltal = integer;
  byte = 0..255;
  dag = (man,tir,ons,tor,fre,lor,son);
  liste = ARRAY(.1..10.) OF real;
  complex = RECORD re,im: real END;
```

Yderligere eksempler på typedefinitioner findes i de følgende kapitler.

4.2.4 Variabelerklæringer

Alle variable, der anvendes i et program, skal erklæres i variabelerklæringsdelen. Erklæring af en variabel skal altid gå forud for dens brug, eller, med andre ord, variablen skal være "kendt" af compileren, førend man kan referere til den.

En variabelerklæring erklærer en ny variabelidentifier og forbinder den med en datatype. Denne sammenhæng mellem identifier og type gælder i hele den blok, hvori variabelerklæringen foretages, med mindre identifieren redefineres i en indre blok. Variabelerklæringsdelen indledes med det reserverede ord VAR, efterfulgt af et antal variabelerklæringer, adskilt af semikolon. Hver variabelerklæring består af en eller flere identificere, adskilt af kommaer, efterfulgt af et kolon og en datatype. Et eksempel:

```
VAR
    rod1,rod2,rod3: real;
    taeller,i: integer;
    fundet: boolean;
    d1,d2: dag;
    buffer: ARRAY(.0..127.) OF byte;
```

Variabele kan eventuelt erklæres til at ligge på faste adresser i lageret. Dette gøres ved at tilføje en AT specifikation til variabelerklæringen efter datatypen. Det reserverede ord AT skal efterfølges af en heltalskonstant, der angiver adressen på den første byte i lageret, der skal optages af variablen. Et eksempel:

```
VAR
    memtop: integer AT $0006;
    kommandolinie: STRING(.127.) AT $0080;
```

AT specifikationen kan også angive, at en variabel skal ligge "oven i" en anden variabel, dvs. at den nye variabel skal starte på den samme adresse som den variabel, der angives efter AT. Et eksempel:

```
VAR
    str: STRING(.32.);
    laengde: byte AT str;
```

Med de ovenfor viste erklæringer kommer str og laengde til at starte på samme adresse i lageret (da den første byte af en strengvariabel indeholder længden af strengen, vil variablen laengde altså indeholde længden af strengen str).

Der må ikke erklæres mere end en variabel ad gangen, når AT specifikationen anvendes.

4.2.5 Procedure- og funktionserklæringer

Procedureerklæringsdelen bruges til at erklære underprogrammer i den nuværende programblok (se afsnit 15.2). En procedure aktiveres via af en proceduresætning (se afsnit 6.1.2).

Funktionserklæringsdelen bruges til at erklære underprogrammer, der udregner og returnerer en værdi (se afsnit 15.3). En funktion aktiveres via et funktionskald, der indgår i et udtryk (se afsnit 5.2).

4.3 Sætningsdelen

Sætningsdelen er den sidste del af en programblok. Den angiver de handlinger, der skal udføres når programmet eksekveres. Sætningsdelen svarer til en sammensat sætning efterfulgt af et punktum. En sammensat sætning består af et vilkårligt antal sætninger, adskilt af semikolonter, og omsluttet af de reserverede ord BEGIN og END.

Et eksempel på et program:

```
PROGRAM konverter(output);
CONST
    offset = 32; faktor = 1.8;
    start = 10; slut = 19;
    separator = '-----';
TYPE
    gradtype = start..slut;
VAR
    grad: gradtype;
BEGIN
    writeln(separator);
    FOR grad:=start TO slut DO
    BEGIN
        write(grad:10,'c',round(grad*faktor+offset):10,'f');
        IF odd(grad) THEN writeln;
    END;
    writeln(separator);
END;
```

Programmet giver den følgende udskrift:

```
-----
 10c      50f      11c      52f
 12c      54f      13c      55f
 14c      57f      15c      59f
 16c      61f      17c      63f
 18c      64f      19c      66f
-----
```

Kapitel 5**Udtryk**

Udtryk består af operander og operatorer. Operander kan være variable, konstanter og funktionskald. Operatorerne er inddelt i fem prioritetsniveauer. Negationsoperatoren (et minus med en operand) har den højeste prioritet, efterfulgt af NOT operatoren, igen efterfulgt af de multiplicerende operatorer, derefter de adderende operatorer, og til sidst de relationelle operatorer. En følge af operatorer med samme prioritet evalueres fra venstre mod højre. Udtryk omsluttet af parenteser evalueres uafhængigt af de omkringstående operatorer.

Dette kapitel beskriver udtryk af de standard skalare datatyper, dvs. integer, real, boolean og char. Udtryk af selvdefinerede skalare typer, strengtyper og mængdetyper kan også konstrueres, hvilket beskrives nærmere i afsnittene 7.1, 8.2 og 11.2.

5.1 Operatorer

Hvis begge operander i en addition (+), subtraktion (-) eller multiplikation (*) er af typen integer, er resultatet også af typen integer. Hvis en eller begge operander er af typen real, er resultatet ligeledes af typen real.

5.1.1 Negationsoperatoren

Negationsoperatoren (et minus med en operand) indikerer, at operandens fortegn skal inverteres. Operanden kan være af typen real eller af typen integer.

5.1.2 NOT operatoren

NOT operatoren angiver at operanden, der er af typen boolean, skal komplementeres:

NOT true	= false
NOT false	= true

COMPAS Pascal tillader også, at NOT operatoren anvendes på en operand af typen integer. I så tilfælde angiver NOT, at samtlige 16 bits i operanden skal inverteres. Nogle eksempler:

NOT 0	= -1
NOT -7	= 6
NOT \$23A5	= \$DC5A

5.1.3 Multiplicerende operatorer

<u>Operator</u>	<u>Operation</u>	<u>Operandtyper</u>	<u>Resultattype</u>
*	Multiplikation	real, integer	real, integer
/	Division	real, integer	real
DIV	Heltalsdivision	integer	som operand
MOD	Modulus	integer	som operand
AND	Logisk AND	integer, boolean	som operand
SHL	Venstreskift	integer	som operand
SHR	Højreskift	integer	som operand

5.1.4 Adderende operatorer

<u>Operator</u>	<u>Operation</u>	<u>Operandtyper</u>	<u>Resultattype</u>
+	Addition	real, integer	real, integer
-	Subtraktion	real, integer	real, integer
OR	Logisk OR	integer, boolean	som operand
EXOR	Logisk EXOR	integer, boolean	som operand

5.1.5 Relationelle operatorer

De relationelle operatorer tillader operander af alle standard skalare typer, dvs. integer, real, boolean og char. Integer og real operander kan eventuelt blandes. Resultattypen er altid boolean, dvs. resultatet er enten true eller false.

a = b	Sandt hvis a er lig med b.
a <> b	Sandt hvis a er forskellig fra b.
a > b	Sandt hvis a er større end b.
a < b	Sandt hvis a er mindre end b.
a >= b	Sandt hvis a er større end eller lig med b.
a <= b	Sandt hvis a er mindre end eller lig med b.

5.2 Funktionskald

Et funktionskald til en standardfunktion eller en selvdefineret funktion kan indgå i et udtryk ved at angive funktionens identificer, eventuelt efterfulgt af en parameterliste. En parameterliste er en følge af variable eller udtryk, adskilt af kommaer, og omsluttet af parenteser. Funktionens resultat kan betragtes som en variabel af samme datatype som funktionen. Eksempler:

```
round(grad)
sqrt(sqr(x)+sqr(y))
(max(a,b)<10) AND (c>100)
volumen(radius,hoejde)
```

Kapitel 6**Sætninger**

Sætningsdelen i et program, en procedure eller en funktion består af en sammensat sætning, dvs. en følge af sætninger, adskilt af semikoloner og omsluttet af de reserverede ord BEGIN og END. Disse sætninger angiver de handlinger, der skal udføres, når programmet, proceduren eller funktionen udføres.

Enhver sætning kan forudgås af en eller flere labels, der kan referes i en GOTO sætning (se afsnit 4.2.1 og 6.1.3).

Sætninger i Pascal er inddelt i simple sætninger og strukturerede sætninger.

6.1 Simple sætninger

En simpel sætning er en konstruktion, der ikke indeholder andre sætninger. I denne gruppe findes tilskrivningssætninger, proceduresætninger, GOTO sætninger og tomme sætninger.

6.1.1 Tilskrivningssætninger

Tilskrivningssætningen er den mest fundamentale af alle sætninger. Den angiver, at en variabel skal tilskrives en ny værdi, der udregnes i et udtryk. Sætningen består af en variabel efterfulgt af tilskrivningsoperatoren (:=) og et udtryk.

Tilskrivninger kan foretages til variable af alle datatyper, dog ikke til filvariable. Bemærk, at variablen og udtrykket skal være af samme type, med den undtagelse, at hvis variabelltypen er real, må udtrykkets type gerne være integer. Eksempler på tilskrivningssætninger:

```
antal:=antal+1;
grad:=grad-10;
fundet:=false;
afstand:=sqrt(sqr(x)+sqr(y));
ciffer:=(num>=0) AND (num<=9);
rod:=(-b+sqrt(sqr(b)-d))/(2*a);
a:=max3(a,b,c);
```

6.1.2 Proceduresætninger

En proceduresætning bruges til at aktivere en standardprocedure eller en selvdefineret procedure. Sætningen består af en procedurereidentifier, eventuelt efterfulgt af en parameterliste. Parameterlisten er en følge af variable eller udtryk, adskilt af kommaer og omsluttet af parenteser. Eksempler på proceduresætninger:

```
seek(f,r)
sorter(navne)
ombyt(x,y)
plot(x,round(sin(x*f)*20.0)+24)
```

6.1.3 GOTO sætninger

En GOTO sætning indikerer, at programudførslen skal fortsættes fra den label, der referes til. Ved brug af GOTO sætninger bør de følgende regler iagttages:

En label kan kun refereres til inden for den blok, hvori den er erklæret. Det er således ikke tilladt at hoppe ind og ud af procedureblokke.

Enhver label skal erklæres i en labelerklæring i den blok, hvori den bruges.

6.1.4 Tomme sætninger

En tom sætning indeholder ingen symboler og angiver således heller ingen handling. En tom sætning forekommer de steder, hvor Pascals syntaks forventer en sætning, men hvor der ingen er. Eksempler:

```
BEGIN END;
WHILE digit AND (a>17) DO (* ingenting *);
REPEAT (* vent *) UNTIL keypress;
```

6.2 Strukturerede sætninger

En struktureret sætning er en konstruktion, der bl.a. indeholder andre sætninger. Disse udføres enten sekventielt (sammensat sætning), betinget (betingede sætninger) eller repeterende (repetitionssætninger).

6.2.1 Sammensatte sætninger

I nogle tilfælde må man kun anvende en enkelt sætning som komponent i en struktureret sætning. Såfremt man i denne situation ønsker at skrive flere sætninger, kan man bruge en sammensat sætning, der er en følge af sætninger, adskilt af semikolonter og omsluttet af de reserverede ord BEGIN og END. Et eksempel:

```
IF x<y THEN
BEGIN
    temp:=x; x:=y; y:=temp;      (* ombyt x og y *)
END;
```

Det er ikke nødvendigt at skrive et semikolon efter den sidste sætning i en sammensat sætning.

6.2.2 Betingede sætninger

En betinget sætning er en sætning, der, på grundlag af et valgudtryk, udfører en af de indeholdte sætninger.

6.2.2.1 IF sætninger

IF sætningen angiver, at den indgående sætning kun skal udføres, hvis et givet boolean udtryk er sand (true). Hvis udtrykket er falsk (false), gælder der enten, at der ikke udføres nogen handling, eller, at sætningen efter ELSE symbolet skal udføres. Konstruktionen:

```
IF <el> THEN IF <e2> THEN <s1> ELSE <s2>
```

skal forstås på følgende måde:

Hvis <el> er falsk, udføres der ingen handling.
Hvis <el> er sand, og <e2> er sand, udføres <s1>.
Hvis <s1> er sand, og <e2> er falsk, udføres <s2>.

Generelt gælder der, at en ELSE-del hører sammen med den sidste IF-del, der mangler en ELSE-del. Eksempler på IF sætninger:

```
IF x<1.5 THEN z:=x+y ELSE z:=1.5;

IF tal<0 THEN
BEGIN
    writeln('Negative tal tillades ikke');
    tal:=0;
END;
```

6.2.2.2 CASE sætninger

En CASE sætning består af et udtryk (kaldet nøgleudtrykket) og en liste af sætninger, hver foregået af en værdiliste. En sætning bliver udført, hvis værdien af valgudtrykket findes i sætningens værdiliste. Hvis nøgleudtrykkets værdi ikke findes i nogen af værdilisterne, gælder der enten, at ingen sætninger udføres, eller, at sætningerne mellem de reserverede ord OTHERWISE og END udføres.

En værdiliste består af et vilkårligt antal konstanter eller intervaller, adskilt med kommaer, efterfulgt af et kolon. Et interval skrives som to konstanter adskilt af et "dovent kolon" (...). Typen af konstanterne skal være den samme som typen af nøgleudtrykket. Sætningen efter værdilisten udføres, hvis nøgleudtrykkets værdi er lig med en af konstanterne, eller indeholdt i et af intervallerne.

Nøgleudtrykket kan være af enhver skalar type, undtagen real. Nogle eksempler:

```
CASE operator OF
  '+': x:=x+y;
  '-': x:=x-y;
  '*': x:=x*y;
  '/': x:=x/y;
END;
```

```

CASE tal OF
  1,3,5,7,9: writeln('ulige ciffer');
  2,4,6,8: writeln('lige ciffer');
  0,10..255: writeln('nul eller mellem 10 og 255');
OTHERWISE
  writeln('negativt eller større end 255');
  taeller:=taeller+1;
END;

```

Den sidste sætning før det reserverede ord OTHERWISE og den sidste sætning før det reserverede ord END behøver ikke efterfølges af et semikolon.

6.2.3 Repetitionssætninger

En repetitionssætning bruges, når man ønsker at udføre en eller flere sætninger gentagne gange. Er antallet af gentagelser på forhånd kendt, bør FOR sætningen bruges. Ellers kan WHILE og REPEAT sætningerne bruges.

6.2.3.1 WHILE sætninger

Udtrykket, der kontrollerer WHILE sætningen, skal være af typen boolean. Den indskrevne sætning gentages sålænge udtrykket er sand (true). Hvis udtrykket er falsk i begyndelsen, bliver sætningen aldrig udført. Nogle eksempler:

```

WHILE tal<1000 DO tal:=sqr(tal);

WHILE i>0 DO
BEGIN
  IF odd(i) THEN z:=z*x;
  i:=i DIV 2;
  x:=sqr(x);
END;

```

6.2.3.2 REPEAT sætninger

Udtrykket, der kontrollerer REPEAT sætningen, skal være af typen boolean. Følgen af sætninger mellem de reserverede ord REPEAT og UNTIL gentages, indtil udtrykket bliver sand (true). Et eksempel:

```

REPEAT
  readln(tal); sum:=sum+tal;
UNTIL tal=0;

```

Det er ikke nødvendigt at skrive et semikolon efter den sidste sætning i en REPEAT sætning.

6.2.3.3 FOR sætninger

FOR sætningen angiver, at den indgående sætning skal udføres gentagne gange, mens en stigende eller faldende række værdier bliver tilskrevet en variabel, kaldet kontrolvariablen. Værdierne kan enten stige i spring af 1 (TO) eller falde i spring af 1 (DOWNTO).

Kontrolvariablen, startværdien og slutværdien skal være af samme skalare type. Typen real er ikke tilladt. Kontrolvariablen skal være en enkeltvariabel (dvs. den må ikke være et element i en struktur).

Hvis startværdien er større end slutværdien ved TO, eller hvis startværdien er mindre end slutværdien ved DOWNTO, udføres den indgående sætning ikke. Nogle eksempler:

```
FOR i:=1 TO 10 DO writeln(i:5,sqr(i):5);

FOR i:=1 TO n DO
BEGIN
  readln(tal);
  IF tal=0 THEN
    antalnuller:=antalnuller+1 ELSE
    IF tal>0 THEN
      positivsum:=positivsum+tal ELSE
      negativsum:=negativsumtal;
END;
```

Bemærk, at den indgående sætning ikke må udføre `tilskrivninger til kontrolvariablen. Hvis FOR løkken skal afbrydes, før den slutter af sig selv, bør en GOTO sætning anvendes (dette er imidlertid dårlig programmeringsteknik - anvend i stedet en WHILE eller en REPEAT sætning).

Efter endt udførelse af en FOR sætning er kontrolvariablen lig med slutværdien, med mindre FOR sætningen blev oversprunget. I sidstnævnte tilfælde bliver der ikke foretaget tilskrivning til kontrolvariablen.

Kapitel 7

Skalare typer og delintervaller

Skalare typer er de grundlæggende datatyper i Pascal. Det, der kendetegner disse typer, er, at de indeholder en endelig og lineært ordnet mængde af værdier.

Datatypen real regnes for en skalar type selvom den ikke opfylder den ovenfor givne definition. Af denne grund kan reals ikke altid anvendes i de sammenhænge, hvor andre skalare typer tillades.

7.1 Skalare typer

Udover at benytte de standard skalare typer (integer, real, boolean og char) kan brugeren selv definere nye skalare typer. Dette gøres ved at angive en liste af identificere, der er de mulige værdier for den nye type, i den rækkefølge, der skal gælde for værdierne. Nogle eksempler:

```
TYPE
  kort = (kloer,ruder,hjerter,spar);
  dag = (man,tirs,ons,tors,fre,loer,soen);
  operator = (plus,minus,gange,dividere);
  tipstegn = (et,kryds,to);
```

Variable af den ovenstående type kort kan antage en af fire værdier, nemlig kloer, ruder, hjerter eller spar. Den standard skalare type boolean er defineret som:

```
TYPE
  boolean = (false,true);
```

De relationelle operatører (=, <>, >, <, >= og <=) kan bruges på alle selvdefinerede skalare typer, forudsat at begge operander er af den samme type (reals og integers kan dog blandes). Den definerede rækkefølge, dvs. den rækkefølge hvori typens konstanter er angivet i typedefinitionen, lægges til grund for sammenligningen. For den ovenstående type kort, gælder der således:

kloer < ruder < hjerter < spar

Der findes tre standardfunktioner, der kan anvendes på alle skalare typer (undtagen real):

succ(x)	returnerer efterfølgeren til x.
pred(x)	returnerer forgængeren til x.
ord(x)	returnerer ordinalværdien af x.

Resultattypen for succ og pred er den samme som argumentets type. Resultattypen for ord er integer. Den ordinale værdi af den første konstant i en selvdefineret skalar type er 0. For de ovenstående skalare typer gælder der:

```
succ(ruder) = hjerter
pred(fre) = tors
ord(gange) = 2
```

7.2 Delintervaller

Man kan definere en ny datatype, som er et delinterval (engelsk " subrange") af en tidligere defineret skalar type (undtagen real). Definitionen angiver blot den mindste og den største værdi i delintervallet, hvor den nedre grænse skal være mindre end eller lig med den øvre. Delintervaller af typen real tillades ikke. Nogle eksempler:

```
TYPE
  dag = (man,tirs,ons,tors,fre,loer,soen);
  arbejdsdag = man..fre;
  fridag = loer..soen;
  ciffer = '0'..'9';
  bogstav = 'A'..'Z';
  skala = -99..99;
  maanedlaengde = 28..31;
```

De ovenstående typer arbejdsdag og fridag er begge delintervaller af den skalare type dag, også kendt som deres overordnede type. Den overordnede type for ciffer og bogstav er char, og den overordnede type for skala og maanedlaengde er integer. Den prædefinerede type byte er givet ved:

```
TYPE
  byte = 0..255;
```

En delintervaltype arver alle de egenskaber, der kendetegner dens overordnede skalare type, blot med en begrænsning af værdiområdet.

Brug af skalare typer og delintervaller kan anbefales af mange grunde. For det første letter de i høj grad forståelsen af et program. For det andet bliver værdier, der tilskrives variable af skalare typer og delintervaltyper, automatisk kontrolleret, når et program udføres (med mindre andet angives), og for det tredje vil skalarer og delintervaller ofte spare lagerplads, da COMPAS Pascal compileren kun reserverer en byte til variable af selvdefinerede skalare typer med op til 256 elementer, og variable af delintervaltyper, hvor både øvre og nedre grænse er mellem 0 og 255.

7.3 Typekonvertering

Som tidligere beskrevet i dette afsnit kan ord funktionen anvendes til at konvertere værdier af skalare typer til værdier af typen integer. Standard Pascal definerer imidlertid ikke en metode til den omvendte situation, dvs. en metode hvormed integers kan konverteres til skalarer.

I COMPAS Pascal kan en værdi af enhver skalar type konverteres til en værdi af enhver anden skalar type, med den samme ordinale værdi, ved hjælp af "retype" faciliteten. Dette opnås ved at anvende typeidentifieren for den ønskede skalare type i et funktionskald. Funktionskaldet skal angive en enkelt parameter, omsluttet af parenteser, der må være en værdi af enhver skalar type. Under forudsætning af typedefinitionerne i afsnit 7.1 gælder der således:

```

integer(njerter) = 2
dag(4) = fre
operator(0) = plus
char(65) = 'A'
integer('0') = 48
operator(ruder) = minus

```

Typekonverteringer må ikke anvende typen real, hverken som den ønskede type eller som argumenttypen.

7.4 Værdikontrol

Medmindre andet angives, genererer COMPAS Pascal compileren kode i det oversatte program til at foretage kontrol af værdier, der tilskrives variable af selvdefinerede skalare typer og delintervaltyper. Dette er ofte en stor fordel når et program skal fejlsøges, men i et færdigt program er det sjældent nødvendigt. Ved hjælp af `(*$R+*)` og `(*$R-*)` compilerdirektiverne er det derfor muligt at styre, om compileren skal generere kode til at foretage kontrol. Ved start af compileren bliver `(*$R+*)` automatisk valgt, og når variable af skalare typer og delintervaltyper tilskrives i denne stilling, vil de tilskrevne værdier blive kontrolleret. I den modsatte stilling, `(*$R-*)`, bliver der ikke foretaget kontrol. Et eksempel:

```

PROGRAM vaerdikontrol;
TYPE
  ciffer = 0..9;
VAR
  c1,c2,c3: ciffer;
BEGIN
  c1:=5;          (* tilladt *)
  c2:=c1+1;       (* tilladt da c1<9 *)
  (*$R-*) c3:=167; (* ulovligt men giver ikke fejl *)
  (*$R+*) c3:=65; (* ulovligt og giver kørselsfejl *)
END.

```

Når værdikontrol er aktiveret bliver der også genereret kode til at checke parameterværdier i procedure- og funktionskald, hvis parameteren er en værdiparameter af en skalar type eller en delintervaltype.

Værdikontrol bør kun passiveres i et grundigt fejlsøgt program.

Kapitel 8

Strenge

Strenge, der er følger af tegn, bruges ofte i programmer. Til strengbehandling i COMPAS Pascal er der mulighed for at definere strengtyper. Antallet af tegn i en strengvariabel, også kaldt længden af strengen, kan variere dynamisk mellem 0 og en given øvre grænse.

8.1 Strengtyper

Når en strengtype defineres, angiver man den maksimale længde af de strengværdier, der kan tilskrives strengvariable af denne type. Strengtypen indledes med det reserverede ord STRING, efterfulgt af en konstant der angiver den maksimale længde. Konstanten skal være et heltal mellem 1 og 255, omsluttet af (. og .) symbolerne. Nogle eksempler:

```
TYPE
  filnavn = STRING(.14.);
  linie = STRING(.72.);
  hexstr = STRING(.4.);
```

Antallet af bytes, der optages i datalageret af en variabel af en given strengtype, svarer til strengvariablens maksimale længde plus en.

8.2 Strengudtryk

Strengværdier kan udregnes fra andre strengværdier ved hjælp af strengudtryk. I lighed med numeriske udtryk er strengudtryk opbygget af operander og operatorer. Operanderne er enten strengkonstanter, strengvariable eller strengfunktioner.

De relationelle operatorer (=, <>, >, <, >= og <=) kan bruges til at sammenligne strenge. Ved en sådan sammenligning bliver tegnene i strengene sammenlignet enkeltvis fra venstre mod højre, indtil to tegn er forskellige. Hvis strengene er af forskellig længde, men ens til længden af den korteste streng, anses den korteste streng for at være den mindste. To strenge er ens, hvis og kun hvis de er af samme længde og indeholder de samme tegn i den samme rækkefølge. Nogle eksempler på sammenligninger, der giver værdien true:

```
'strenge' = 'strenge'
'B' > 'A'
'ABC' < 'ABCD'
'test' <> 'test'
'033' < '12'
```

Sammensætningsoperatoren, der skrives som et plus tegn (+), bruges til at sammensætte strenge. Sammensætningsoperatoren har højere prioritet end de relationelle operatorer. Nogle eksempler:

```
'Jens '+'Hansen' = 'Jens Hansen'
'132'+'.'+'377' = '132.377'
'A'+'B'+'C'+'D' = 'ABCD'
```

8.3 Strengtilskrivninger

Tilskrivningsoperatoren (`:=`) bruges til at tilskrive strengværdier til strengvariable. Nogle eksempler:

```
cifre:='0123456789';
linie:='dette er en streng';
tekst:=""+liniet";
```

Hvis den streng, der tilskrives en strengvariabel, er længere end strengvariablens maksimale længde, er det kun de første tegn, der overføres. Hvis, for eksempel, strengen 'langstreng' tilskrives en strengvariabel af typen STRING(.4.), vil variablen kun indeholde de første fire tegn, dvs. 'lang'.

8.4 Strengfunktioner og procedurer

De følgende strengfunktioner er standard i COMPAS Pascal:

<code>len(s)</code>	Returnerer længden af strengudtrykket <code>s</code> , dvs. antallet af tegn i <code>s</code> . Resultattypen er integer.
<code>pos(p,s)</code>	<code>p</code> og <code>s</code> er strengudtryk, og resultattypen er integer. Funktionen returnerer positionen af den første forekomst af strengen <code>p</code> i strengen <code>s</code> (positionen af det første tegn i en streng er 1). Hvis <code>p</code> ikke findes i <code>s</code> , returneres 0.
<code>copy(s,i,n)</code>	<code>s</code> er et strengudtryk, og <code>i</code> og <code>n</code> er udtryk af typen integer. <code>copy</code> returnerer en streng, der indeholder <code>n</code> tegn fra <code>s</code> , startende fra den <code>i</code> 'te position i <code>s</code> . Hvis <code>i</code> er større end <code>len(s)</code> , returneres en tom streng. Hvis <code>i+n-1</code> er større end <code>len(s)</code> , returneres en streng på <code>len(s)-i+1</code> tegn. Hvis <code>i</code> er udenfor området 1..255, stopper programmet med en kørselsfejl.
<code>concat(strs)</code>	<code>strs</code> er et vilkårligt antal strengudtryk, adskilt af kommaer. Resultatet er en streng, der består af sammensætningen af de angivne strelge. Hvis længden af resultatet er større end 255, stopper programmet med en kørselsfejl. Bemærk, at strengsammensætning også kan foretages med plus (+) operatoren. <code>concat</code> er udelukkende medtaget i COMPAS Pascal for at sikre kompatibilitet med andre versioner af Pascal.

De følgende strengprocedurer er standard i COMPAS Pascal:

delete(s,i,n) s er en strengvariabel og både i og n er udtryk af typen integer. Proceduren fjerner n tegn fra s, startende med det i'te tegn. Hvis i er større end len(s), bliver ingen tegn fjernet. Hvis i+n-1 er større end len(s), bliver len(s)-i+1 tegn fjernet. Hvis i er udenfor området 1..255, stopper programmet med en kørselsfejl.

insert(p,s,i) p er et strengudtryk, s er en strengvariabel, og i er et udtryk af typen integer. Proceduren indsætter strengen p i strengen s på den i'te position. Hvis i er større end len(s), bliver p sat i forlængelse af s. Hvis resultatet er større end den maksimale længde af s, vil s kun indeholde de første tegn. Hvis i er udenfor området 1..255, stopper programmet med en kørselsfejl.

val(s,x,p) s er et strengudtryk, x er en variabel af typen real eller af typen integer, og p er en variabel af typen integer. Den numeriske streng i s konverteres til en værdi af samme type som x, og gemmes i x. Den numeriske værdi skal følge de regler, der gælder for numeriske konstanter (se afsnit 2.2). Hverken foranstændende eller efterfølgende blanktegn tillades. Hvis konverteringen forløber korrekt, sættes p til 0. Ellers sættes p til positionen på det første ulovlige tegn (i så tilfælde er værdien af x ubestemmelig).

str(p,s) p er en write-parameter af typen real eller af typen integer, og s er en strengvariabel. Proceduren konverterer den numeriske værdi til en streng og gemmer resultatet i s. Formatet af write-parametre er beskrevet i afsnit 16.3.

Det nedenfor viste program demonstrerer strengbehandling i COMPAS Pascal:

```

PROGRAM strengdemo;
VAR
  st,mere,delstr,mindre: STRING(.64.);
  i,p: integer;
  r: real;
BEGIN
  st:='sammensat';
  mere:='dette er en '+st+' streng';
  writeln('linie 1: ',mere);
  st:='her står 16 tegn';
  writeln('linie 2: ',len(st),' ',len(''));
  st:='dette er en tekststreng';
  delstr:='tekst';
  writeln('linie 3: ',pos(delstr,st),' ',pos('7','12345'));
  st:='delstrenge udtages med copy funktionen';
  mindre:=copy(st,pos('u',st),7);
  writeln('linie 4: ',mindre);
  writeln('linie 5: ',copy('12345',3,255));
  st:='dette er en meget lang tekststreng';
  delete(st,pos('en',st)+3,11);

```

```
writeln('linie 6: ',st);
st:='a er lig med b';
insert('mindre end eller ',st,6);
writeln('linie 7: ',st);
st:=-1547;
val(st,i,p);
writeln('linie 8: ',i,' ',p);
r:=pi;
str(r:10:6,st);
writeln('linie 9: ',st);
END.
```

Programmet giver den følgende udskrift:

```
linie 1: dette er en sammensat streng
linie 2: 16 0
linie 3: 13 0
linie 4: udtages
linie 5: 345
linie 6: dette er en tekststreng
linie 7: a er mindre end eller lig med b
linie 8: -1547 0
linie 9: 3.141593
```

8.5 Strenge og tegn

Strengetyper og den standard skalare type char er kompatible, dvs. når en strengværdi forventes, kan et tegn (en char værdi) angives i stedet, og omvendt. Desuden kan strenge og tegn blandes i udtryk. Når en strengværdi tilskrives en char variabel, skal strengens længde være præcis 1, ellers stopper programmet med en kørselsfejl.

De enkelte tegn i en streng kan adresseres individuelt gennem indexering. Dette opnås ved at efterfølge strengvariablen med et indexudtryk, af typen integer, omsluttet af (. og .) symbolerne. Nogle eksempler:

```
linie(.37.)    ciffer(.i.)    navn(.len(navn)-1.)
```

Tegnet med index 0 angiver strengens nuværende længde. Således svarer len(s) til ord(s(.0.)). Der føres ikke kontrol med, om værdier, der tilskrives længdeindikatoren, er mindre end den maksimale længde af strengvariablen.

Når værdikontrol er aktiveret, dvs. når sætninger compileres i (*.R+*) stillingen, genereres der tillige kode, der checker, at værdier i indexudtryk ikke overstiger strengvariablenes maksimale længder. Det er imidlertid muligt at indexere en streng uover dens nuværende dynamiske længde. Indholdet af sådanne tegnpositioner er tilfældigt, og tilskrivninger til dem ændrer ikke strengens reelle værdi.

Kapitel 9

Arraystrukturer

En arraystruktur består af et antal dataelementer, der alle er af samme type, kaldet elementtypen. Et element refereres til ved at angive et index, der bestemmer hvor i arraystrukturen, elementet findes. Indices udregnes ved hjælp af indexudtryk, og deres type kaldes indextypen.

9.1 Brug af arraystrukturer

En arraytypes definition angiver både elementtypen og indextypen (og dermed antallet af elementer i arraystrukturen). Arraytypen indledes med det reserverede ord ARRAY. Herefter følger indextypen, der er omsluttet af (. og .) symbolerne, og tilsidst det reserverede ord OF, efterfulgt af elementtypen. Nogle eksempler:

```

TYPE
    ciffer = 0..9;
    farve = (roed,groen,bla);
    liste = ARRAY(.1..100.) OF real;
VAR
    ciffernavn: ARRAY(.ciffer.) OF STRING(.10.);
    intensitet: ARRAY(.farve.) OF ciffer;
    a,b: liste;

```

Et arrayelement udvælges ved at angive arrayvariablens navn efterfulgt af et indexudtryk omsluttet af (. og .) symbolerne. Nogle eksempler:

```

ciffernavn(.5.):='fem';
intensitet(.groen.):=7;
a(.i.):=a(.i.)*2.5/b(.j+1.);
a:=b;

```

Da tilskrivning mellem to variable af samme type altid er tilladt, er det muligt at kopiere hele arraystrukturer ved hjælp af tilskrivningsoperatoren som vist ovenfor (a:=b).

(*SR+*) og (*SR-*) compilerdirektiverne bestemmer, om compileren skal generere kode til at foretage kørselskontrol af indexværdier. Ved start af compileren bliver (*SR+*) automatisk valgt, og når arrayvariable indexeres i denne stilling, bliver indexudtrykkene checket mod grænserne i deres tilhørende indextype.

Ved hjælp af (**S+*) og (**S-*) compilerdirektiverne kan programmøren bestemme, om den kode, der genereres til at udføre indexeringer af arrays, skal optimeres med hastighed eller lagerforbrug for øje. Ved start af compileren vælges (**S-*), og kode til indexering, der genereres i denne stilling, bliver optimeret med henblik på størrelsen af koden. I (**S+*) stillingen bliver koden derimod optimeret med henblik på eksekveringshastighed. Hastighedsgevinsten er tydeligst for flerdimensionale arrays.

9.2 Flerdimensionale arraystrukturer

En arraystrukturs elementtype kan være enhver datatype. Således kan elementtypen igen være en arraytype, og da er arraystrukturen flerdimensional. Nogle eksempler:

```

TYPE
  skakbrik = (konge,dame,taarn,springer,loeber,bonde);
  skakbraet = ARRAY(.1..8.) OF ARRAY(.1..8.) OF skakbrik;
  side = 0..9;
  terning = ARRAY(.side.) OF ARRAY(.side.) OF
             ARRAY(.side.) OF real;
VAR
  braet: skakbraet;
  t: terning;

```

Definitionen af en flerdimensional arraystruktur kan skrives kortere ved at angive alle indextyper på en gang:

```

skakbraet = ARRAY(.1..8,1..8.) OF skakbrik;
terning = ARRAY(.side,side,side.) OF real;

```

På samme måde kan en reference til et element i en flerdimensional arraystruktur også forkortes:

braet(.3,7.)	svarer til	braet(.3.)(.7.)
t(.0,4,3.)	svarer til	t(.0.)(.4.)(.3.)

En flerdimensional arraytype kan naturligvis udtrykkes ved hjælp af tidligere definerede arraytyper. Et eksempel:

```

CONST
  n = 3; m = 4;
TYPE
  vektor = ARRAY(.1..n.) OF real;
  matrix = ARRAY(.1..m.) OF vektor;
VAR
  v1,v2: vektor;
  m1: matrix;

```

I dette tilfælde er de nedenfor viste tilskrivninger tilladte:

```

v1(.i.):=m1(.i,j.)+1.5;
m1(.i.)(.j.):=v1(.i.)+v2(.j.);
v1:=v2;
v2:=m1(.i.);
m1(.j.):=v1;

```

9.3 Prædefinerede arraystrukturer

Til brug for direkte adgang til CPU lager og dataporte findes der i COMPAS Pascal to prædefinerede arraystrukturer kaldet mem og port.

9.3.1 mem arraystrukturen

mem arraystrukturen giver adgang til CPU'ens lager. Til hvert element i mem svarer en byte i lageret, hvis adresse er givet ved indexudtrykket. Når et element tilskrives en værdi, bliver værdien gemt i lageret, og når der i et udtryk refereres til et element, bliver værdien læst fra lageret. Nogle eksempler:

```
mem(.addr(v)+offset.):=$C0;
iobyte:=mem(.3.);
mem(.i.):=mem(.i+1.);
```

9.3.2 port arraystrukturen

port arraystrukturen giver adgang til CPU'ens dataporte. Hvert element i port repræsenterer en dataport, hvis portadresse er givet ved indexudtrykket. Da dataporte adresseres gennem 8-bits adresser, skal indexudtryk give værdier mellem 0 og 255. Når et element tilskrives en værdi, bliver værdien udlæst på porten, og når der i et udtryk refereres til et element, bliver værdien læst fra porten. Nogle eksempler:

```
port(.S46.):=$FF;
port(.base.):=port(.base.) EXOR mask;
WHILE port(.S2.). AND $80=0 DO (* vent *);
```

Visse begrænsninger knytter sig til port arraystrukturen. For det første er det ikke muligt at referere til hele arraystrukturen (reference uden indexudtryk), og for det andet kan elementer ikke anvendes som var-parametre til procedurer og funktioner.

9.4 Tegn-arrays

Tegn-arrays er arraystrukturer med et index og elementer af typen char, dvs. arraystrukturer der er defineret som:

```
ARRAY(.n..m.) OF char
```

hvor n og m er af typen integer, og m er større end n. Tegn-arrays kan opfattes som strenge med en konstant længde ($m-n+1$). Strengkonstanter kan tilskrives til tegn-arrays, forudsat at længden af konstanten er lig med antallet af elementer i arraystrukturen ($m-n+1$).

I COMPAS Pascal kan tegn-arrays indgå i strengudtryk. Et tegn-array konverteres da til en streng af længden $m-n+1$ (hvor n og m er nedre og øvre grænse i tegn-arrayets indextype). Tegn-arrays kan altså sammenlignes og manipuleres på samme måde som strenge. Bemærk dog, at værdier udregnet i strengudtryk ikke kan tilskrives til tegn-arrays - de eneste værdier, der kan tilskrives, er strengkonstanter og andre tegn-array variable af samme type.

Kapitel 10

Poster

En post er en struktureret datatype, og på samme måde som en arraystruktur består den af flere enkelte elementer. I en post kaldes elementerne felter. I et felt kan der lagres værdier af en bestemt datatype, der angives i definitionen af posttypen. Datatypen kan vælges helt frit, og kan eventuelt være endnu en posttype.

10.1 Brug af poster

En posttypes definition angiver en datatype og et feltnavn for hvert felt i posten. Definitionen indledes med det reserverede ord RECORD og afsluttes med det reserverede ord END. Herimellem skrives en feltliste, der består af en eller flere feltsektioner, adskilt af semikoloner. Hver feltsektion består af en eller flere identificere, adskilt af kommaer, efterfulgt af et kolon og en datatype. Nogle eksempler:

```

TYPE
    complex = RECORD
        re,im: real;
    END;
    maanedsnavn = (jan,feb,mar,apr,maj,jun
                    ,jul,aug,sep,okt,nov,dec);
    datopost = RECORD
        dag: 1..31;
        maaned: maanedsnavn;
        aar: 1900..1999;
    END;
    navnepost = RECORD
        efternavn: STRING(.32.);
        antalnavne: 1..3;
        fornavne: ARRAY(.1..3.) OF STRING(.16.);
    END;
VAR
    x,y: complex;
    dato: datopost;
    kunde: navnepost;
    fridage: ARRAY(.1..14.) OF datopost;

```

Identifierne re, im, dag, maaned, aar, efternavn, antalnavne og fornavne er feltnavne. Et feltnavn er lokalt for en posttype, og samme feltnavn kan derfor anvendes i flere forskellige posttyper. Et felt i en post angives ved at skrive et punktum og feltnavnet efter postvariablen. Nogle eksempler:

```

x.re:=5.9;
y.im:=x.im+y.im;
dato.dag:=2;
dato.maaned:=dec;
dato.aar:=1960;
kunde.efternavn:='Hansen';
kunde.antalnavne:=2;
kunde.fornavne(.1.):='Karl';

```

```
kunde.fornavne(.2.):='Erik';
fridage(.10.):=dato;
x:=y;
```

Bemærk, at samtlige værdier i en post kan kopieres direkte over i de tilsvarende felter i en anden post af samme type, som vist i de sidste to tilskrivninger ovenfor.

Felter i en post kan være af enhver datatype. Således kan en felttype være yderligere en posttype, som vist nedenfor:

```
TYPE
  personpost = RECORD
    navn: navnepost;
    foedselsdato: datopost;
  END;
VAR
  pl,p2,p3: personpost;
```

I dette tilfælde er de nedenfor viste tilskrivninger tilladte:

```
pl.navn.efternavn:='Jensen';
pl.navn.antalnavne:=1;
pl.navn.fornavne(.1.):='Peter';
pl.foedselsdato.dag:=17;
pl.foedselsdato.maaned:=mar;
pl.foedselsdato.aar:=1951;
p2.navn:=pl.navn;
p3:=p2;
```

10.2 WITH sætninger

Den ovenfor anvendte notation kan være en smule anstrengende, men den kan da også forkortes ved hjælp af en WITH sætning. WITH sætningen "åbner" en post, således at feltnavnene bliver tilgængelige som almindelige variabelidentificere. En WITH sætning starter med det reserverede ord WITH. Herpå følger en række postvariable adskilt af kommaer, efterfulgt af det reserverede ord DO og en sætning. I den indskrevne sætning kan feltidentifierne i de postvariable, der angives mellem WITH og DO, anvendes som almindelige variabelidentificere. Den nedenfor viste WITH sætning svarer til sætningerne ovenfor:

```
WITH pl,navn,foedselsdato DO
BEGIN
  eternavn:='Jensen';
  antalnavne:=1;
  fornavne(.1.):='Peter';
  dag:=17;
  maaned:=mar;
  aar:=1951;
  p2.navn:=navn;
  p3:=p2;
END;
```

Antag at den følgende erklæring er foretaget:

```
VAR
    datoer: ARRAY(.1..7.) OF datopost;
```

Sætningen:

```
WITH datoer(.4.) DO
IF maaned=dec THEN
BEGIN
    maaned:=jan; aar:=aar+1;
END ELSE maaned:=succ(maaned);
```

svarer da til:

```
IF datoer(.4.).maaned=dec THEN
BEGIN
    datoer(.4.).maaned:=jan;
    datoer(.4.).aar:=datoer(.4.).aar+1;
END ELSE
    datoer(.4.).maaned:=succ(datoer(.4.).maaned);
```

Formen:

```
WITH p1,p2,...,pn DO
```

svarer til:

```
WITH p1 DO WITH p2 DO .... WITH pn DO
```

Det maksimale antal nestede WITH sætninger, der tillades, dvs. det maksimale antal poster, der kan åbnes på en gang, kontrolleres via compilerens W register. Ved start af compileren vælges (*\$W4*), hvilket betyder, at maksimalt 4 WITH sætninger kan nestes. Det maksimale nestingniveau kan varieres mellem 0 og 9, og er givet ved indholdet af W registeret ved begyndelsen af den nuværende blok. Hvis et (*\$W8*) compilerdirektiv placeres før erklæringsdelen i en blok, kan WITH sætninger i denne blok altså nestes op til 8 gange (bemærk, at dette ikke betyder, at der kun må være 8 WITH sætninger i blokken, men derimod, at der højest må være 8 WITH sætninger inden i hinanden). I begyndelsen af en blok reserverer compileren 2 bytes for hver tilladt nesting niveau.

10.3 Variant-del i poster

En posttype kan defineres sådan, at forskellige variable af denne posttype kan indeholde forskellige felter. Dette opnås ved at indføje en variant-del i posttypen. Variant-delen indledes med det reserverede ord CASE, efterfulgt af en skalar datatype, kaldt mærketypen, igen efterfulgt af det reserverede ord OF. Hver variant startes med en eller flere konstanter, hvis type er givet ved mærketypen, efterfulgt af et kolon. Selve varianten angives som en feltliste omsluttet af parenteser. Varianterne adskilles med semikoloner.

Antag, som et eksempel, at man ønsker at beskrive tre forskellige geometriske figurer ved hjælp af en enkelt datatype. Hvis de tre geometriske figurer er:

```
TYPE
    figurtype = (rektaengel,triangel,cirkel);
```

Så kan den det generelle format af den beskrivende type for eksempel være:

```
TYPE
    figur = RECORD
        <felter fælles for alle figurer>;
        CASE figurtype OF
            rektaengel: (<felter for rektaengler>);
            triangel: (<felter for triangler>);
            cirkel: (<felter for cirkler>);
        END;
```

Almindeligvis indeholder en post med en variant-del et felt, der angiver hvilken variant, der bruges i øjeblikket. Dette felt kaldes mærkefeltet. Mærkefeltet for typen figur kunne for eksempel være:

```
art: figurtype
```

Når en posttype defineres, skrives mærkefeltet imellem de reserverede ord CASE og OF:

```
TYPE
    figur = RECORD
        farve: (roed,groen,blaa);
        CASE art: figurtype OF
            rektaengel: (hoejde,bredde: real);
            triangel: (sidel,side2,vinkel: real);
            cirkel: (radius: real);
        END;
```

Den faste del af en posttype skal altid skrives før variant-delen. I typen figur er feltet farve det eneste felt i den faste del. En posttype kan kun indeholde en variant-del. En tom variant-del skrives som to parenteser, der ikke omslutter noget.

Bemærk at COMPAS systemet ikke fører kontrol med, om man i en variant-del, der er beregnet for en type af information, lagrer information af en anden type. I en variabel af den overfor viste type figur giver det derfor ikke fejl at referere til feltet radius, selvom mærkefeltet art ikke er lig med værdien cirkel. Faktisk kan mærkefeltet helt udelades, således at der kun angives en datatype mellem CASE og OF.

Kapitel 11**Mængder**

En samling af objekter, der refereres til under et, kaldes en mængde. Objekterne, der indgår i mængden, kaldes mængdens elementer. Nogle eksempler på mængder:

- A. Alle heltal mellem 0 og 100.
- B. Primtallene mellem 0 og 100.
- C. Alfabetets bogstaver.
- D. Alfabetets vokaler.

To mængder er ens hvis og kun hvis de indeholder de samme elementer. Rækkefølgen af elementerne har ingen betydning, så mængderne (.1,3,5.) og (.5,3,1.) er altså ens. Hvis elementerne i en mængde også indgår i en anden mængde, er den første mængde en delmængde af den anden. For de ovenfor nævnte mængder gælder der, at B er en delmængde af A og D er en delmængde af C. D er også en delmængde af sig selv, men ikke en ægte delmængde, idet der samtidigt gælder, at D er lig med D.

I Pascal findes der tre forskellige mængdeoperationer. Fællesmængden af to mængder A og B, hvilket skrives som A+B, er den mængde, der indeholder alle elementer fra A og alle elementer fra B. For eksempel er fællesmængden af (.1,3,5,7.) og (.2,3,4.) lig med (.1,2,3,4,5,7.). Foreningsmængden af to mængder A og B, hvilket skrives som A*B, er den mængde, hvis elementer både er elementer i A og B. For eksempel er foreningsmængden af (.1,3,5,7.) og (.2,3,4.) lig med (.3.). Differensmængden af to mængder A og B, hvilket skrives som A-B, er den mængde, hvis elementer er elementer i A men ikke i B. For eksempel er differensmængden af (.1,3,5,7.) og (.2,3,4.) lig med (.1,5,7.).

11.1 Mængdetyper

Teoretisk set er der ingen grænse for, hvormange elementer en mængde kan indeholde, ligesom elementer i en mængde i teorien kan være hvad som helst. En mængde i Pascal kan derimod kun indeholde et vist antal elementer, og elementerne skal alle være af samme type (kaldet elementtypen). Elementtypen må være enhver skalar type, undtagen real. En mængdetype angives ved de reserverede ord SET OF efterfulgt af elementtypen. Nogle eksempler:

```
TYPE
  talmaengde = SET OF 0..50;
  cifre = SET OF 0..9;
  bogstaver = SET OF 'A'..'A';
  farver = SET OF (roed,groen,blaa);
  tegnsaet = SET OF char;
```

Det maksimale antal elementer, en mængde i COMPAS Pascal kan indeholde, er 256. Desuden gælder der, at den ordinale værdi af grænserne i elementtypen skal være mellem 0 og 255.

11.2 Mængdeudtryk

Mængder kan udregnes udfra andre mængder ved hjælp af mængdeudtryk. Disse opbygges af mængdekonstanter, mængdevariable, mængdeangivere og operatorer.

11.2.1 Mængdeangivere

En mængdeangiver består af en liste af elementer og intervaller omsluttet af (. og .) symbolerne. En element er et udtryk af mængdens elementtype, og et interval er to sådanne udtryk adskilt af et dovent kolon (...). Nogle eksempler på mængdeangivere:

```
(.1,3,5.)
('C','O','M','P','A','S'.)
(.x.)
(.i,j,k+3.)
(.1..5.)
(.i..j.)
('A'...'A','a'...'å','0'...'9'.)
(.1,3..10,12.)
```

Mængden (.1..5.) svarer til (.1,2,3,4,5.). Hvis i er større end j så angiver (.i..j.) en tom mængde. (...) angiver også en tom mængde, men denne tomme mængde er kompatibel med alle mængdetyper, da den ikke indeholder nogen udtryk, der kan bestemme dens type.

11.2.2 Mængdeoperatorer

I mængdeudtryk findes der tre prioritetsniveauer for operatorer. Foreningsmængdeoperatoren (*) har den højeste prioritet, efterfulgt af fælles- og differensmængdeoperatorerne (+ og -), igen efterfulgt af de relationelle operatorer og IN operatoren. Her følger en oversigt over mængdeoperatorerne:

- * Foreningsmængde.
- + Fællesmængde.
- Differensmængde.
- = Test for lighed.
- <> Test for ulighed.
- >= Sandt hvis den anden operand er en delmængde af den første.
- <= Sandt hvis den første operand er en delmængde af den anden.
- IN Den anden operand er en mængde, og den første operand er et udtryk af samme type som mængdens elementtype. Resultatet er sand hvis elementet, der er givet ved det første udtryk, er indeholdt i mængden.

Mængdeudtryk kan ofte anvendes til at tydeliggøre inåviklede og lange sandhedsudtryk. For eksempel svarer konstruktionen:

```
IF (ch='E') OR (ch='C') OR (ch='S') OR (CH='I') THEN
```

til den noget mere elegante konstruktion:

```
IF ch IN ('E','C','S','I') THEN
```

På tilsvarende måde kan konstruktionen:

```
IF (ch>='0') AND (ch<='9') THEN
```

forkortes til:

```
IF ch IN ('0'..'9') THEN
```

11.3 Mængdetilskrivninger

Tilskrivningsoperatoren kan anvendes til at tilskrive mængdevariable mængdeværdier, der er udregnet i mængdeudtryk. Eksempler:

```
VAR
  cifre: SET OF 0..9;
  vokaler,konsonanter: SET OF 'A'..'A';
  ulige: SET OF 1..100;
  i: integer;
BEGIN
  cifre:=(0..9.);
  vokaler:=(.'A','E','I','O','U','Y','R','Ø','A.');
  konsonanter:=(.'A'..'A.')-vokaler;
  ulige:=(..);
  FOR i:=1 TO 50 DO ulige:=ulige+(.i+i-1.);
END.
```

Kapitel 12

Typeangivne konstanter

I modsætning til definitionen af en typeløs konstant (se afsnit 4.2.2), specificerer definitionen af en typeangiven konstant både konstantens type og dens værdi. En typeangiven konstant svarer til en variabel af samme type, blot med en konstant værdi.

Typeangivne konstanter defineres i en konstantdefintionsdel i en bloks erklærningsdel. Konstantens identifier skal efterfølges af et kolon og en type, der igen skal efterfølges af et lighedstejn og selve konstanten.

12.1 Typeangivne konstanter af simple typer

En typeangiven konstant kan anvendes når som helst en variabel af den samme type forventes. Typeangivne konstanter kan altså, i modsætning til almindelige typeløse konstanter, bruges som variabler til procedurer og funktioner. Nogle eksempler på typeangivne konstanter af simple typer:

```
CONST
  maximum: integer = 9999;
  faktor: real = -6.32774526;
  navn: STRING(.13.) = 'COMPAS Pascal';
  crlf: STRING(.2.) = ^M^J;
```

En typeangiven konstant kan anbefales frem for en typeløs konstant, hvis konstanten anvendes ofte i programmet. Grunden hertil er, at en typeangiven konstant kun inkluderes en enkelt gang i programmets kode, medens en typeløs konstant inkluderes hver gang, der refereres til den. Da en typeangiven konstant faktisk er en variabel med en konstant værdi, kan den ikke anvendes i definitionen af andre konstanter eller typer. Det nedenfor viste eksempel er således ikke tilladt (da min og max ikke er typeløse konstanter):

```
CONST
  min: integer = 0;
  max: integer = 100;
TYPE
  liste = ARRAY(.min..max.) OF real;
```

12.2 Strukturerede konstanter

Ved hjælp af typeangivne konstanter er det muligt at definere konstanter af strukturerede typer, dvs. array-, post- og mængdekonstanter. Sådanne konstanter bruges normalt som prædefinerede tabeller og mængder i konverteringer og tests.

12.2.1 Arraykonstanter

En arraykonstant består af en liste af konstanter adskilt af kommaer og omsluttet af parenteser. Nogle eksempler:

```

TYPE
  farve = (roed,groen,blaa);
  navneliste = ARRAY(.farve.) OF STRING(.4.);
CONST
  farvenavn: navneliste = ('rød','grøn','blå');
  fak: ARRAY(.1..7.) OF integer = (1,2,6,24,120,720,5040);

```

Den ovenfor viste konstantdefinitionsdel definerer en arraykonstant, kaldet farvenavn, der bruges til at omsætte værdier af typen farve til deres tilsvarende tekstrepræsentation, og en arraykonstant, kaldet fak, der bruges til high-speed beregninger af fakulteten af et heltal. Elementerne i farvenavn er:

```

farvenavn(.roed.) = 'rød'
farvenavn(.groen.) = 'grøn'
farvenavn(.blaa.) = 'blå'

```

Flerdimensionale arraystrukturer tillades naturligvis også. Sådanne konstanter defineres ved at omslutte hver dimension med parenteser. Den inderste konstant svarer til dimensionen længst mod højre i arraystrukturen. Nogle eksempler:

```

TYPE
  matrix = ARRAY(.1..2,1..3.) OF real;
  kubus = ARRAY(.0..1,0..1,0..1.) OF integer;
CONST
  m: matrix = ((1.0,1.1,1.2),(6.0,7.5,9.0));
  k: kubus = (((1,7),(2,9)),((0,8),(6,3)));

```

Elementerne i k er:

k(.0,0,0.) = 1	k(.0,0,1.) = 7
k(.0,1,0.) = 2	k(.0,1,1.) = 9
k(.1,0,0.) = 0	k(.1,0,1.) = 8
k(.1,1,0.) = 6	k(.1,1,1.) = 3

Elementtypen i en arraykonstant kan være enhver type undtagen filtyper og pointertyper. Tegn-array konstanter angives enten som enkelttegn eller som strenge. Definitionen:

```

CONST
  cifre: ARRAY(.0..9.) OF char =
    ('0','1','2','3','4','5','6','7','8','9');

```

kan altså også skrives som:

```

CONST
  cifre: ARRAY(.0..9.) OF char = '0123456789';

```

12.2.2 Postkonstanter

En postkonstant består af en liste af feltkonstanter adskilt af semikoloner og omsluttet af parenteser. Hver feltkonstant angiver en feltidentifier efterfulgt af et kolon og en konstant. Nogle eksempler:

```

TYPE
  complex = RECORD
    re,im: real;
  END;
  maanedsnavn = (jan,feb,mar,apr,maj,jun,
                  jul,aug,sep,okt,nov,dec);
  datopost = RECORD
    dag: 1..31;
    maaned: maanedsnavn;
    aar: 1900..1999;
  END;
  navnepost = RECORD
    efternavn: STRING(.32.);
    antalnavne: 1..3;
    fornavne: ARRAY(.1..3.) OF STRING(.16.);
  END;
CONST
  cl: complex = (re: 4.75; im: -8.0);
  kdato: datopost = (dag: 12; maaned: feb; aar: 1983);
  hansen: navnepost = (efternavn: 'Hansen';
                        antalnavne: 2;
                        fornavne: ('Jens','Erik',''));
  tal: ARRAY(.1..3.) OF complex =
    ((re: 1; im: 2),(re: 0; im: 37),(re: -3; im: 8.5));

```

Feltkonstanter skal angives i den rækkefølge, felterne er erklæret i postens definition. Hvis en post indeholder felter af filtyper eller pointertyper, er det ikke muligt at definere konstanter af denne posttype. Hvis en post indeholder en variant, er det op til programmøren kun at angive felter, der indgår i den valgte variant. Hvis varianten indeholder et mærkefelt, skal en værdi også angives for dette.

12.2.3 Mængdekonstanter

Syntaksen for en mængdekonstant er nøjagtig den samme som for en mængdeangiver (se afsnit 11.2.1), bortset fra, at elementerne og intervallernes grænser naturligvis skal være konstanter. Nogle eksempler på mængdekonstanter:

```

TYPE
  cifre = 0..9;
  bogstaver = 'A'..'Z';
CONST
  lige: cifre = (.0,2,4,6,8.);
  vokaler: bogstaver = (.'A','E','I','O','U','Y'.);
  alfanum: SET OF char = (.'A'..'Z','a'..'z','0'..'9'.);

```

Kapitel 13

Filer

Ved hjælp af filer kan et program gemme data, der senere kan læses af programmet selv eller af et andet program. I modsætning til andre datatyper, bliver data, der udlæses til en fil, ikke gemt i lageret, men udskrives i stedet til en diskfil. På samme måde bliver data, der indlæses fra en fil, læst fra en diskfil.

En fil er en følge af elementer, alle af samme datatype. Antallet af elementer i en fil, ofte kaldet længden af filen, er ikke på forhånd fastsat i typedefinitionen, som det for eksempel er tilfældet med en arraystruktur.

Til hver fil hører der en fil-pointer, der kontrolleres af systemet. Hver gang et element udlæses til eller indlæses fra en fil, flytter systemet automatisk fil-pointeren frem til det næste element i filen. Systemet er desuden i stand til at flytte fil-pointeren til ethvert af brugeren valgt element i filen.

13.1 Filtyper

En filtype angives med de reserverede ord FILE OF efterfulgt af typen af filens elementer. Nogle eksempler:

```

TYPE
  datopost = RECORD
    dag: 1..31;
    maaned: 1..12;
    aar: 1900..1999;
  END;
  datofil = FILE OF datopost;
  liste = ARRAY(.0..9.) OF real;
VAR
  hfil: FILE OF integer;
  afil: FILE OF liste;
  dfil: datofil;
```

Elementtypen kan være enhver datatype, undtagen en filtype (FILE OF FILE typer tillades altså ikke). Filvariable må ikke optræde i tilskrivningssætninger eller i udtryk.

13.2 Operationer på filer

Følgende standardprocedurer kan bruges til behandling af filer (f angiver en fil-identifier):

assign(f,s)	s er et strengudtryk, der angiver et CP/M filnavn. Filnavnet tilskrives til filvariablen, og alle efterfølgende filoperationer på f vil arbejde på diskfilen ved navn s. assign må aldrig anvendes på en fil, der er åben.
-------------	--

<code>rewrite(f)</code>	Denne procedure bruges til at oprette nye diskfiler. Diskfilen får det navn, der er tilskrevet f, og fil-pointeren sættes til begyndelsen af filen. Den nye fil indeholder ingen elementer. Hvis der i forvejen findes en fil af samme navn, bliver denne slettet.
<code>reset(f)</code>	Denne procedure bruges til at åbne en eksisterende diskfil. Diskfilen er givet ved det navn, der er tilskrevet f. Hvis diskfilen ikke eksisterer, rapporteres en I/O fejl - ellers åbnes filen, og fil-pointeren sættes til det første element.
<code>read(f,vs)</code>	vs er en eller flere variable, af samme type som f's elementer, adskilt af kommaer. Hver variabel læses fra filen, og efter hver læsning flyttes fil-pointeren frem til det næste element.
<code>write(f,vs)</code>	vs er en eller flere variable, af samme type som f's elementer, adskilt af kommaer. Hver variabel skrives til filen, og efter hver skrivning flyttes fil-pointeren frem til det næste element.
<code>seek(f,n)</code>	n er et udtryk af typen integer. Fil-pointeren flyttes til det n'te element i filen. Det første element i en fil har nummer 0. Bemærk, at det ikke er muligt at flytte ud over længden af en fil.
<code>flush(f)</code>	Et kald til denne procedure sikrer, at f's interne sektorbuffer er udskrevet på disken. Envidere sikres det, at den næste read-operation vitterligt foretager en læsning på disken. Normalt anvendes flush kun i programmer, der skal køres på flerbrugersystemer, hvor flere brugere har adgang til den samme fil på en gang.
<code>close(f)</code>	Diskfilen, der behandles via f, lukkes, og diskettens bibliotek opdateres, så det nu angiver filens nye status.
<code>erase(f)</code>	Diskfilen, der er sammenknyttet med f, slettes. Hvis filen, der skal slettes, er åben, dvs. hvis der er foretaget kald til rewrite eller reset men ikke til close, bør close kaldes først.
<code>rename(f,s)</code>	Denne procedure ændrer navnet på den diskfil, der er sammenknyttet med f, til det navn, der er givet ved strengudtrykket s. rename bør ikke anvendes på åbne filer. Hvis f anvendes efter rename, uden forudgående kald til assign, arbejdes der på filen med det nye navn.

Følgende standardfunktioner kan bruges til behandling af filer:

<code>eof(f)</code>	Returnerer værdien sand (true) hvis f's fil-pointer er placeret ved slutningen af filen, altså "bagved" det sidste element i filen. Hvis fil-pointeren ikke er ved slutningen af filen, returneres værdien falsk (false).
---------------------	---

position(f)	En heltalsfunktion, der returnerer fil-pointerens værdi, altså nummeret på det nuværende element i filen. 0 svarer til det første element.
length(f)	En heltalsfunktion, der returnerer filens længde, altså antallet af elementer i filen. Hvis length(f) er nul, er filen tom.

Der skal altid udføres et kald til assign, før der foretages operationer på en fil. Hvis der skal foretages læsninger og/eller skrivninger fra/til filen, skal der desuden udføres et kald til rewrite eller reset. Efter et sådant kald peger fil-pointeren på det første element i filen, altså position(f)=0. Efter et kald til rewrite gælder der desuden, at length(f)=0.

En fil kan kun udvides ved at føje elementer til slutningen af den. Fil-pointeren kan flyttes til slutningen af filen ved at udføre sætningen:

```
seek(f,length(f));
```

Hvis der er foretaget læsninger og/eller skrivninger på en fil, skal close kaldes efter endt behandling af filen. Undlades dette, kan data gå tabt, da systemet ikke får opdateret filens status korrekt vis.

Det nedenfor viste program udlæser 1000 tilfældige heltal, mellem 0 og 999, til diskfilen A:RANDOM.DAT.

```
PROGRAM skrivtal;
VAR
  i,k: integer;
  talfil: FILE OF integer;
BEGIN
  assign(talfil,'A:RANDOMS.DAT');
  rewrite(talfil);
  FOR i:=1 TO 1000 DO
  BEGIN
    k:=random(1000); write(talfil,k);
  END;
  close(talfil);
END.
```

Det nedenfor viste program indlæser de tilfældige tal fra diskfilen, der blev oprettet af det ovenstående program, og beregner deres gennemsnit:

```
PROGRAM laestal;
VAR
  i,n: integer;
  sum: real;
  talfil: FILE OF integer;
BEGIN
  assign(talfil,'A:RANDOMS.DAT');
  reset(talfil);
  sum:=0.0; n:=0;
  WHILE NOT eof(talfil) DO
  BEGIN
    read(talfil,i); sum:=sum+i; n:=n+1;
  END;
```

```

    close(talfil);
    writeln('gennemsnittet er ',sum/n:0:3);
END.
```

Det nedenfor viste program opretter en ny fil A:TEMP.DAT, og ud læser alle elementer fra A:RANDOM.DAT til den, i omvendt rækkefølge. Derefter slettes A:RANDOMS.DAT, og A:TEMP.DAT omdøbes i stedet til dette navn. Bemærk brugen af seek proceduren og length funktionen til at læse inputfilen baglæns:

```

PROGRAM omvend;
VAR
  p,n: integer;
  indfil,udfil: FILE OF integer;
BEGIN
  assign(indfil,'A:RANDOMS.DAT'); reset(indfil);
  assign(udfil,'A:TEMP.DAT'); rewrite(udfil);
  FOR p:=length(indfil)-1 DOWNTO 0 DO
  BEGIN
    seek(indfil,p); read(indfil,n); write(udfil,n);
  END;
  close(indfil); close(udfil);
  erase(indfil); rename(udfil,'A:RANDOMS.DAT');
END.
```

13.3 Textfiler

Elementerne i en tekstfil er ASCII tegn, og hvert tegn optager en byte i filen. Forskellen på en textfil og en selvdefineret filtype er, at textfilen, udover at være en følge af elementer (tegn), yderligere er inddelt i linier. Hver linie består af en vilkårligt antal tegn afsluttet af et end-of-line mærke. CP/M operativsystemet anvender en CR/LF sekvens (et CR tegn efterfulgt af et LF tegn) som end-of-line mærke og et CTRL/Z tegn som end-of-file mærke. Da længden af en linie kan variere, kan positionen på en given linie ikke beregnes, og det er derfor ikke muligt at foretage tilfældig flytning af en textfils fil-pointer. Desuden er det ikke tilladt at udlæse til og indlæse fra en textfil på en gang.

13.3.1 Operationer på textfiler

En textfilvariabel erklæres ved at referere til den prædefinerede typeidentifier **text**.

I lighed med selvdefinerede filer skal læsninger og skrivninger til og fra en textfil altid forudgås af et kald **assign** efterfulgt af et kald til **reset** eller **rewrite**. Hvis en ny textfil skal oprettes, skal **rewrite** bruges til at klargøre filen. I dette tilfælde tillader systemet kun udlæsning til filen. Hvis en eksisterende tekstfil skal læses, skal **reset** bruges til at klar-gøre filen. I dette tilfælde tillader systemet kun indlæsning fra filen. Når en textfil, der blev klargjort med **rewrite**, lukkes, ved et kald til **close** proceduren, føjer systemet automatisk et end-of-file mærke til filen.

Tegn skrives til en textfil med write proceduren og læses fra en textfil med read proceduren. Pascal har desuden en række specielle procedurer og functioner, der muliggør behandling af linier i en textfil (t angiver en textfil-identifier):

readln(t)	Flytter fil-pointeren frem til begyndelsen af den næste linie ved at læse alle tegn til og med den næste CR/LF sekvens.
writeln(t)	Skriver et end-of-line mærke (en CR/LF sekvens) til filen.
eoln(t)	En funktion, der returnerer sand hvis fil-pointeren er placeret ved slutningen af en linie (nærmere bestemt ved CR tegnet i den CR/LF sekvens der afslutter linien). eoln(t) returnerer desuden sand hvis eof(t) er sand.

De ovennævnte operationer kan kun anvendes på textfiler. Standardfunktionen eof returnerer sand (true) når fil-pointeren peger på textfilens end-of-file mærke, altså det CTRL/Z tegn, der afslutter filen. Da al behandling af textfiler foregår sekventielt, kan seek og flush procedurerne og position og length funktionerne ikke anvendes i forbindelse med textfiler.

Det nedenfor viste program foretager en frekvenstælling af tegnene i filen A:LETTER.TXT.

```

PROGRAM tegnfrekvens;
CONST
  bogstaver: SET OF 'A'..'A' = ('A'..'A'.);
  smaabogst: SET OF 'a'..'å' = ('a'..'å'.);
VAR
  antal: ARRAY('A'..'A'.) OF integer;
  ch: char;
  t: text;
BEGIN
  FOR ch:='A' TO 'A' DO antal(.ch.):=0;
  assign(t,'A:LETTER.TXT'); reset(t);
  WHILE NOT eof(t) DO
  BEGIN
    WHILE NOT eoln(t) DO
    BEGIN
      read(t,ch);
      IF ch IN smaabogst THEN ch:=chr(ord(ch)-32);
      IF ch IN bogstaver THEN antal(.ch.):=antal(.ch.)+1;
    END;
    readln(t);
  END;
  close(t);
  writeln('tegn antal');
  FOR ch:='A' TO 'A' DO
    writeln(' ',ch,antal(.ch.):7);
END.
```

13.3.2 Logiske I/O enheder

En textfil kan bruges til at kommunikere med CP/M operativsystems logiske I/O enheder. Dette opnås ved at sammenknytte textfilen med I/O enhedens symbolske navn via assign proceduren. De følgende I/O enheder er tilgængelige fra COMPAS Pascal:

- CON: Console enheden. Udlæsninger sendes til CP/M console output enheden (normalt systemets skærm), og indlæsninger hentes fra CP/M console input enheden (normalt tastaturet). I modsætning til TRM: enheden (se nedenfor) anvender CON: enheden en liniebuffer under indlæsning. Kort fortalt betyder dette, at input fra CON: enheden altid sker i form af linier, og at disse linier kan editeres når de indtastes. Flere detaljer om indlæsning fra CON: enheden gives i afsnit 13.3.3 og afsnit 16.1.
- TRM: Terminal enheden. Udlæsninger sendes til CP/M console output enheden (normalt systemets skærm), og indlæsninger hentes fra CP/M console input enheden (normalt tastaturet). Med mindre et indlæst tegn er en kontroltegn, udskrives det på skærmen. Dette gælder dog ikke CR tegnet, der udskrives som en CR/LF sekvens.
- KBD: Keyboard enheden (kun indlæsning). Indlæsninger hentes fra CP/M console input enheden (normalt tastaturet). De indlæste tegn udskrives ikke.
- LST: List enheden (kun udlæsning). Udlæsninger sendes til CP/M list enheden (normalt systemets printer).
- AUX: Auxiliary enheden. Udlæsninger sendes til CP/M punch enheden, og indlæsninger hentes fra CP/M reader enheden. Normalt refererer disse enheder til et modem eller en anden ydre enhed.
- USR: User enheden. Udlæsninger sendes til den brugerdefinerbare outputroutine, og indlæsninger hentes fra den brugerdefinerbare inputroutine. Yderligere detaljer om brugerdefinerede I/O rutiner gives i kapitel 21.

Logiske enheder skal, i lighed med diskfiler, åbnes via et kald til reset eller rewrite før de anvendes. Funktionen af reset og rewrite er ens for logiske enheder. close proceduren har ingen effekt på en logisk enhed. Hvis erase eller rename anvendes på en logisk enhed, gives der en I/O fejl.

Det nedenfor viste program udskriver filen B:MESSAGE.TXT på systemets printer:

```

PROGRAM listfil;
VAR
  ch: char;
  indfil,udfil: text;
BEGIN
  assign(indfil,'B:MESSAGE.TXT'); reset(indfil);
  assign(udfil,'LST:'); rewrite(udfil);
  WHILE NOT eof(indfil) DO
    BEGIN
      WHILE NOT eoln(indfil) DO

```

```

BEGIN
  read(indfil,ch); write(udfil,ch);
END;
  readln(indfil); writeln(udfil);
END;
  close(indfil); close(udfil);
END.

```

Standardprocedurerne eof og eoln arbejder forskelligt på diskfiler og logiske enheder. For en diskfil gælder der, at eof returnerer sand, hvis det næste tegn er CTRL/Z, og eoln returnerer sand, hvis det næste tegn er CR eller CTRL/Z. Således kan eof og eoln faktisk "se forud" i filen. På en logisk enhed er det imidlertid ikke muligt at se forud (det næste tegn kendes først når det indlæses), og eof og eoln anvender derfor det sidst indlæste tegn i stedet for det næste tegn. For en logisk enhed gælder der altså, at eof returnerer sand, hvis det sidst indlæste tegn var CTRL/Z, og eoln returnerer sand, hvis det sidst indlæste tegn var CR eller CTRL/Z. Forskellene er illustreret ved det nedenfor viste program, der indlæser tegn fra skærmterminalen og udskriver dem på printeren, indtil der tastes CTRL/Z.

```

PROGRAM printer;
VAR
  ch: char;
  terminal,printer: text;
BEGIN
  assign(terminal,'TRM:'); reset(terminal);
  assign(printer,'LST:'); rewrite(printer);
REPEAT
  REPEAT
    read(terminal,ch);
    IF NOT eoln(terminal) THEN write(printer,ch);
  UNTIL eoln(terminal);
  IF NOT eof(terminal) THEN
  BEGIN
    readln(terminal); writeln(printer);
  END;
  UNTIL eof(terminal);
  close(terminal); close(printer);
END.

```

I lighed med eof og eoln, skelner readln også mellem diskfiler og logiske enheder. For en diskfil gælder der, at readln læser alle tegn til og med den næste CR/LF sekvens. På en logisk enhed læses der derimod kun til og med det næste CR tegn. Grunden er ganske simpel, idet det systemet ikke kan vide, om CR tegnet efterfølges af et LF tegn eller ej (den manglende evne til at "se forud" på logiske enheder).

13.3.3 Standardfiler

Nedenfor vises en oversigt over standardfilerne i COMPAS Pascal. Ved at anvende disse filer fremfor erklærede filer, er det muligt at spare den plads, der ellers skulle bruges til filvariablene og deres initialisering. Alle filerne er præ-tilskrevne til logiske enheder, og det er ikke nødvendigt at åbne dem førend de anvendes (faktisk er det forbudt at anvende assign, reset, rewrite og close på disse filer).

input Den primære inputfil. Denne fil er refererer til CON: enheden eller TRM: enheden (se nedenfor).

output Den primære outputfil. Denne fil refererer til CON: enheden eller TRM: enheden (se nedenfor).

con Refererer til CON: enheden.

trm Refererer til TRM: enheden.

kbd Refererer til KBD: enheden.

lst Refererer til LST: enheden.

aux Refererer til AUX: enheden.

usr Refererer til USR: enheden.

Den logiske enhed, der refereres til af input og output filerne, afgøres af compilerens B flag. Ved start af compileren vælges (*\$B++) automatisk, og i denne stilling refererer input og output til CON: enheden. Hvis et (*\$B-) compilerdirektiv placeres i begyndelsen af programmet (bemærk: før erklæringsdelen), vil input og output i stedet referere til TRM: enheden. TRM: enheden har ingen editeringsfaciliteter under indlæsninger, men inddata kan følge de formater, der gives i Standard Pascal definitionen. CON: enheden har derimod fuld editering (se afsnit 16.1), idet inddata opsamles i en buffer en linie ad gangen, men formatet af inddata følger ikke altid standarden. Bemærk, at der ikke er forskel på udlæsninger til CON: og TRM: enhederne.

Da standardfilerne input og output anvendes meget ofte, vælger compileren automatisk disse filer hvis intet andet angives. Nedenfor vises en liste af textfiloperationer og deres forkortede versioner:

write(ch)	svarer til	write(output,ch)
read(ch)	svarer til	read(input,ch)
writeln	svarer til	writeln(output)
readln	svarer til	readln(input)
eof	svarer til	eof(input)
eoln	svarer til	eoln(input)

Yderligere udvidelser af read og write procedurerne (til ind- og udlæsning af andre datatyper end tegn) beskrives i kapitel 16.

13.4 Typeløse filer

En typeløs fil bruges primært til direkte læsning og skrivning af sektorer fra og til CP/M diskfiler. En typeløs fil erklæreres med det reserverede ord FILE og intet andet, for eksempel:

```
VAR
  datafil: FILE;
```

13.4.1 Operationer på typeløse filer

Alle standard filoperationer, undtagen read, write og flush, kan anvendes på typeløse filer. Til ind- og udlæsninger anvendes to specielle standardprocedurer, blockread og blockwrite. Formatet af kald til disse procedurer er:

```
blockread(f,v,n)      og      blockwrite(f,v,n)
```

hvor f er identifieren for en typeløs fil, v er enhver variabel, og n er et udtryk af typen integer. Ved et kald til blockread eller blockwrite overføres n sektorer (hver på 128 bytes) fra/til diskfilen til/fra lageret, startende med den første byte, der optages af variablen v. Det er op til programmøren at sikre, at v optager nok bytes til at kunne indeholde hele den læste/skrevne datamængde. Efter hver læsning/skrivning af en sektor, flyttes fil-pointeren frem til den næste sektor (ialt flyttes fil-pointeren altså n sektorer frem).

Ind- og udlæsninger fra og til en typeløs fil skal, i lighed med almindelige filer, foregås af et kald til assign og et kald til reset eller rewrite. Hvis rewrite anvendes, oprettes og åbnes en ny fil. Hvis reset anvendes, åbnes en eksisterende fil. Hvis reset eller rewrite anvendes på en typeløs fil, skal close også kaldes ved endt behandling, for at sikre en korrekt afslutning.

Effekten af et kald til seek proceduren eller length eller position funktionerne med en typeløs fil, svarer til en kald med en almindelig fil.

Det nedenfor viste program demonstrerer brugen af en typeløs fil til at udskrive en hex-listing af en diskfil. Hex-listingen udlæses via en textfil til en anden diskfil eller til en logisk I/O enhed.

```
PROGRAM hexlisting;
TYPE
  hexstr = STRING(.4.);
  filnavn = STRING(.14.);
  sektor = ARRAY(.0..7,0..15.) OF byte;
  systemfil = FILE;
VAR
  i,j,adresse: integer;
  indnavn,udnavn: filnavn;
  indfil: systemfil;
  udfil: text;
  buffer: sektor;

FUNCTION hex(tal,cifre: integer): hexstr;
CONST
  hexcifre: ARRAY(.0..15.) OF char = '0123456789ABCDEF';
VAR
  h: hexstr;
  c: integer;
BEGIN
  h(.0.):=chr(cifre);
  FOR c:=cifre DOWNTO 1 DO
    BEGIN
      h(.c.):=hexcifre(.tal AND 15.);
      tal:=tal SHR 4;
```

```

    END;
    hex:=h;
END;

BEGIN
  write('inputfil? '); readln(indnavn);
  write('outputfil? '); readln(udnavn);
  assign(indfil,indnavn); reset(indfil);
  assign(udfil,udnavn); rewrite(udfil);
  adresse:=0;
  WHILE NOT eof(indfil) DO
  BEGIN
    blockread(indfil,buffer,1);
    FOR i:=0 TO 7 DO
    BEGIN
      write(udfil,hex(adresse,4));
      FOR j:=0 TO 15 DO
      write(udfil,hex(buffer(.i,j.),2):3);
      writeln(udfil);
      adresse:=adresse+16;
    END;
  END;
  close(indfil); close(udfil);
END.

```

Ved ind- og udlæsninger fra og til typeløse filer, flyttes data direkte fra/til diskfilen til/fra variablen, i modsætning til andre filer, hvor data først passerer gennem en sektorbuffer i filvariablen. Da der således ikke er behov for en sektorbuffer i en typeløs fil, fylder typeløse filvariable mindre end almindelige filvariable. Hvis en filvariabel kun skal anvendes til kald til erase og rename og andre non-I/O operationer, er en typeløs fil derfor at foretrække.

13.5 I/O kontrol

(`*$I+*`) og (`*$I-*`) compilerdirektiverne bestemmer, om compileren skal generere kode til kørselskontrol af I/O operationer. Ved start af compileren bliver (`*$I+*`) automatisk valgt, og i denne stilling vil alle I/O operationer blive kontrolleret under kørsel af programmet. I den modsatte stilling, (`*$I-*`), bliver der ikke genereret kontrolkode. Status af en I/O operation kan i stedet kontrolleres ved at udføre et kald til standardfunktionen `iores`. I Appendix F findes en oversigt over de mulige statuskoder og deres betydning. Bemærk, at sålænge der ikke forekommer fejl, returnerer `iores` 0. Bemærk også, at i tilfælde af fejl bliver alle I/O operationer suspenderet, indtil `iores` kaldes. Først når dette sker, forlades fejltilstanden, hvorefter yderligere I/O operationer kan foretages.

`iores` faciliteten er brugbar i mange situationer. Den nedenfor viste programstump viser, hvorledes `iores` kan anvendes til at undersøge, om en given diskfil eksisterer eller ej:

```

assign(f,'B:LETTER.TXT');
(*$I-*) reset(f) (*$I+*);
IF iores>0 THEN writeln('filen findes ikke.');

```

Når (*\$I-*) stillingen bruges, bør kald til øe følgende procedurer og funktioner kontrolleres med et kald til iores:

rewrite	reset	read
write	readln	writeln
blockread	blockwrite	seek
flush	close	erase
rename	execute	chain

execute og chain procedurerne er beskrevet i kapitel 18.

Kapitel 14

Pointere

Indtil nu har denne manual kun beskrevet statiske variable, dvs. variable der ligger på en faste plader i lageret og som refereres med faste identificere. En dynamisk variabel er en plads, som man selv kan reservere under udførelsen af et program. Med en sådan plads er der ikke associeret et fast navn, med pladsen refereres i stedet med en pointer. Pointerens "værdi" er da blot startadressen på den dynamiske variabel.

14.1 Pointertyper

En pointertype angives med pointersymbolet (^) efterfulgt af typeidentifieren (bemærk: typeidentifieren, ikke typen) for de dynamiske variable, der kan allokeres og refereres gennem pointervariable af denne type. Nogle eksempler:

```
TYPE
  intptr = ^integer;
  str40 = STRING(.40.);
  strptr = ^str40;
  data = RECORD
    i: integer; ch: char;
  END;
  dataptr = ^data;
```

Det er tilladt typeidentifieren i definitionen af en pointertype at referere til en endnu ukendt typeidentifier. Bemærk, at dette er det eneste tilfælde, hvor referencer til ukendte identificere tillades. Et eksempel:

```
TYPE
  link = ^person;
  person = RECORD
    fornavn,efternavn: STRING(.32.);
    alder: 0..100;
    næste: link;
  END;
```

Dynamiske variable af den ovenfor viste type person er karakteriserede ved, at de indeholder en pointer til den næste dynamiske variabel i en kædet liste. Pointere anvendes meget ofte til at opbygge sådanne kædede lister.

14.2 Brug af pointere

En dynamisk variabel refereres via en pointer ved at efterfølge pointervariablen med pointersymbolet (^). Antag, at de følgende erklæringer er udført:

```

TYPE
  dims = RECORD
    navn: STRING(.20.);
    pris: real;
  END;
  dimsprt = ^dims;
  liste = ARRAY(.1..5.) OF integer;
VAR
  pint: ^integer;
  foerstedims: dimsprt;
  pliste: ^liste;
  i: integer;

```

De følgende tilskrivninger kan da udføres:

```

pint^:=4;
foerstedims^.navn:='hammer';
foerstedims^.pris:=29.75;
pliste^(.3.):=7;
pliste^(.2.):=pint^;
pliste^(.i.):=pliste^(.i+1.)*2;

```

En dynamisk variabel allokeres med standardproceduren new. Under forudsætning af de ovenstående erklæringer, vil sætningen:

```
new(foerstedims);
```

altså allokerer en dynamisk variabel af typen dims, og tilskrive dens adresse i lageret til pointeren foerstedims.

En pointer kan tilskrives værdien af en anden pointer, forudsat begge pointere er af samme type. To pointere af samme type kan desuden testes for lighed og ulighed med de relationelle operatører = og <>.

Pointerværdien NIL (bemærk at NIL er et reserveret ord og ikke en standardkonstant) er kompatibel med alle pointertyper. Når en pointer ikke peger på noget objekt, tilskrives den normalt værdien NIL. NIL kan desuden anvendes i sammenligninger af pointere.

Som tidligere nævnt er sammensætning af poster til kædede strukturer en udbredt anvendelse af dynamiske poststrukturer. Hver post skal da indeholde et felt med en pointer til en anden post, normalt af samme type som posten selv. Det nedenfor viste program opbygger en kædet liste af personer, og udskriver derefter listen:

```

PROGRAM liste;
TYPE
  navnetype = STRING(.30.);
  personptr = ^person;
  person = RECORD
    navn: navnetype;
    naeste: personptr;
  END;
VAR
  personliste, pp: personptr;
  nytnavn: navnetype;
BEGIN
  personliste:=NIL;

```

```

writeln('skriv navne og slut med en blank linie:');
REPEAT
  readln(nytnavn);
  IF nytnavn<>'' THEN
  BEGIN
    new(pp);
    pp^.navn:=nytnavn;
    pp^.naeste:=personliste;
    personliste:=pp;
  END;
UNTIL nytnavn='';
writeln('de følgende navne blev indtastet:');
pp:=personliste;
WHILE pp<>NIL DO
BEGIN
  writeln(pp^.navn); pp:=pp^.naeste;
END;
END.

```

Dynamiske variable, der oprettes ved kald til new proceduren, lagres på en stak-lignende struktur, der kaldes systemets "heap". COMPAS Pascal styrer heapen gennem en heap-pointer. Ved begyndelsen af en program sættes heap-pointeren til at pege på den første ledige byte i lageret. Hvergang new kaldes, bliver heap-pointeren flyttet opad i lageret svarende til det antal bytes, den nye dynamiske variabel optager.

Hvis en eller flere dynamiske variable af en eller anden grund ikke længere anvendes, kan standardprocedurerne mark og release bruges til at frigive det lager, der optages af disse variable. mark proceduren bruges til at gemme heap-pointerens nuværende adresse i en variabel. Formatet af et kald til mark er:

```
mark(v);
```

hvor v kan være enhver pointervariabel. release proceduren sætter heap-pointeren til den adresse, der er indeholdt i argumentet. Formatet af et kald til release er:

```
release(v);
```

hvor v kan være enhver pointervariabel, der tidligere er sat ved hjælp af mark. Det nedenfor viste program demonstrerer på simpel vis brugen af mark og release.

```

PROGRAM heap;
TYPE
  objekt = RECORD
    navn: STRING(.30.);
    pris: real;
  END;
VAR
  objektptr: ^objekt;
  heapmaerke: ^integer;
BEGIN
  mark(heapmaerke); new(objektptr);
  objektptr^.navn:='skruetrækker';
  objektptr^.pris:=13.95;
  release(heapmaerke);
END.

```

I begyndelsen af programmet bruges mark til at gemme heap-pointerens startværdi i variablen heapmaerke (typen af heapmaerke er uden betydning, da heapmaerke aldrig bruges i et kald til new). Derefter udføres et kald til new for at allokerer en dynamisk variabel af typen objekt, og den dynamiske variabel tilskrives en værdi. Til sidst kaldes release med heapmaerke som argument, hvorved heap-pointeren flyttes tilbage til begyndelsestilstanden, og dermed friges det lager, der var optaget af den dynamiske variabel.

Hvis der var udført flere kald til new mellem kaldene til mark og release, ville det lager, der var optaget af alle disse dynamiske variable, blive frigivet. På grund af heaps stak-lignende opbygning, er det ikke muligt at frigive en dynamisk variabel midt i heapen. Bemærk desuden, at forkert brug af mark og release kan efterlade "flydende pointere", der peger på dynamiske variable, der ikke længere er en del af den definerede heap.

Standardfunktionen memavail kan bruges til at undersøge hvormange bytes, der er ledige til yderligere udvidelse af heapen på et givet tidspunkt. memavail tager ingen argumenter, og returnerer en værdi af typen integer. Hvis der er mere end 32767 ledige bytes, returnerer memavail et negativt tal. Den rigtige værdi kan da beregnes af $65536.0 - \text{memavail}$ (bemærk at en konstant af typen real anvendes for at gennemtvinge et resultat af denne type - det er nødvendigt da resultatet er større end maxint).

Yderligere informationer om systemets brug af lageret findes i kapitel 23.

14.3 Pointere og integers

I COMPAS Pascal findes der to standardfunktioner, ord og ptr, der muliggør konvertering mellem pointere og integers. ord returnerer en integer, der svarer til den adresse, der er indeholdt i pointerargumentet, og ptr konverterer en integer til en pointerværdi, der er kompatibel med alle pointertyper. Disse funktioner er yderst anvendelige for en øvet programmør, da de gør det muligt for en pointer at pege på enhver adresse i lageret. Hvis de bruges forkert er de imidlertid også meget farlige, da det er muligt at placere en dynamisk variabel oven i andre variable eller, hvilket er endnu farligere, oven i programmets kode.

14.4 Oversigt over pointerrutiner

I COMPAS Pascal findes de følgende procedurer til styring af dynamiske variable:

new(p)	p er en variabel af enhver pointertype. Proceduren allokerer en dynamisk variabel af den type, der er forbundet med p, og gemmer dens adresse i p.
mark(p)	p er en variabel af enhver pointertype. Proceduren gemmer heap-pointerens nuværende værdi i p.
release(p)	p er en variabel af enhver pointertype. Proceduren sætter heap-pointeren til den værdi, der er indeholdt i p.

I COMPAS Pascal findes de følgende pointerrelaterede funktioner:

memavail	Returnerer antallet af bytes mellem heap-pointeren og toppen af det frie lager. Resultatet er af typen integer. Hvis der er mere end 32767 bytes ledige, returneres et negativt tal, og den rigtige værdi kan da beregnes af 65536.0-memavail.
ord(p)	Konverterer pointerværdien p til en værdi af typen integer. p kan være af enhver pointertype. Bemærk, at ord(NIL)=0.
ptr(i)	Konverterer adressen givet ved integerudtrykket i til en pointerværdi, der er kompatibel med alle pointertyper. Bemærk, at ptr(0)=NIL.

Kapitel 15

Procedurer og funktioner

En procedure er en separat programdel, der aktiveres fra en proceduresætning (se afsnit 6.1.2). En funktion er på mange måder lig en procedure, bortset fra, at funktionen beregner og returnerer en værdi, der indgår i det udtryk, hvorfra funktionen aktiveres (se afsnit 5.2).

15.1 Parametre

Procedurer og funktioner (i fællesskab kaldet underprogrammer) kan have parametre, der tillader underprogrammet at gentage dets handlinger med visse variationer. I underprogrammet kan parametrene anvendes på lige fod med almindelige variable.

Parametre, der nævnes i proceduresætningen eller funktionskaldet, kaldes aktuelle parametre. Parametre, der nævnes i underprogrammets erklæring, kaldes formelle parametre. Under kørsel af underprogrammet bliver de formelle parametre erstattet af de aktuelle parametre. Bemærk, at de formelle og de aktuelle parametre skal stemme overens med hensyn til antal, type og rækkefølge. I COMPAS Pascal er der to typer af parametre: Value-parametre og var-parametre. Når symbolen VAR skrives foran en formel parameter i underprogrammets overskrift, er denne parameter en var-parameter.

Hvis en parameter er en value-parameter, oprettes der ved start af underprogrammet en variabel (den formelle parameter), hvis begyndelsesværdi er givet ved den aktuelle parameter. Den aktuelle parameter er et udtryk, hvoraf en variabel er et simpelt tilfælde. Den aktuelle parameter berøres ikke af, at underprogrammet tilskriver nye værdier til den formelle parameter, og value-parametre kan derfor ikke anvendes til at returnere resultater fra underprogrammet.

Hvis en parameter er en var-parameter, er både den aktuelle og den formelle parameter lig den samme lagerplads. I dette tilfælde skal den aktuelle parameter være en variabel. Under kørsel af underprogrammet repræsenteres den aktuelle parameter af den formelle parameter, og tilskrivninger til den formelle parameter vil ændre den aktuelle parameter tilsvarende. Var-parametre kan derfor anvendes til at returnere resultater fra underprogrammet.

Alle adresseberegninger udføres før et kald. Hvis en parameter for eksempel er et element i en arraystruktur, bliver adressen på dette element udregnet før underprogrammet kaldes.

Filvariable skal altid overføres som var-parametre.

Når en stor datastruktur, såsom et array, skal overføres til et underprogram, bør en var-parameter i videst muligt omfang anvendes. Dette sparer både tid og plads, da der kun skal overføres to bytes, der angiver adressen på strukturen, i modsætning til hele strukturen, der overføres hvis en value-parameter anvendes.

15.2 Procedurer

En procedure er enten en standardprocedure eller en selvdefineret procedure. Standardprocedurer behøver, i modsætning til selvdefinerede procedurer, ikke erklæres før de anvendes. Hvis en selvdefineret procedure erklæres med samme navn som en standardprocedure, kan standardproceduren ikke anvendes i den pågældende blok.

15.2.1 Procedureerklæringer

En procedureerklæring består af en overskrift, en erklæringsdel og en sætningsdel.

Overskriften angiver procedurens navn, i form af en identifier, og eventuelt en parameterliste. Parameterlisten består af et antal parametersektioner, adskilt af semikolonter og omsluttet af parenteser. Hver parametersektion består af en eller flere identificere, adskilt af kommaer, efterfulgt af et kolon og en type-identifier. Det reserverede ord VAR foran en parametersektion angiver, at parametrerne i denne sektion er var-parametre. Nogle eksempler:

```
PROCEDURE afbryd;
PROCEDURE drawto(x,y: integer);
PROCEDURE sorter(VAR navne: liste; fra,til: integer);
```

En procedures erklæringsdel er opbygget på samme måde som et programs. Alle identificere, der nævnes i procedurens overskrift og i dens erklæringsdel, er lokale for proceduren og andre procedurer og funktioner erklæret inden i denne. Dette kaldes identificerens rækkevidde. En procedure kan referere til identificeren for enhver konstant, type, variabel, procedure eller funktion, der er global i forhold til proceduren (erklæret i en ydre blok), eller den kan i stedet redefinere identificeren.

Sætningsdelen angiver de handlinger, der skal udføres når proceduren kaldes. I lighed med et program er en procedures sætningsdel en sammensat sætning, dvs. et antal sætninger adskilt af semikolonter og omsluttet af de reserverede ord BEGIN og END. Hvis proceduren udfører et kald til sig selv, kaldes den for en rekursiv procedure.

Det nedenfor viste program anvender en procedure med en value-parameter. En value-parameter bruges, fordi den aktuelle parameter i nogle tilfælde er en konstant (et simpelt udtryk), der ikke kan overføres som en var-parameter.

```
PROGRAM histogram;
VAR
  i,k,n: integer;
  tal: real;

PROCEDURE skrivlinie(laengde: integer);
VAR
  i: integer;
BEGIN
  FOR i:=1 TO laengde DO write('*');
  writeln;
END;
```

```

BEGIN
  readln(n);
  FOR i:=1 TO n DO
    BEGIN
      readln(tal);
      k:=round(tal);
      IF k<0 THEN skrivlinie(0) ELSE
      IF k>79 THEN skrivlinie(79) ELSE
        skrivlinie(k);
    END;
  END.

```

Her er et andet program, der anvender en procedure med to variabelparametre. Var-parametre bruges, fordi proceduren skal returnere et resultat, hvilket kun kan gøres via parametre af denne art.

```

PROGRAM sammenlign;
VAR
  a,b: integer;

PROCEDURE ombyt(VAR x,y: integer);
VAR
  temp: integer;
BEGIN
  temp:=x; x:=y; y:=temp;
END;

BEGIN
  readln(a,b);
  IF a=b THEN writeln('Tallene er ens') ELSE
  BEGIN
    IF b>a THEN ombyt(a,b);
    writeln('Det største tal er ',a);
    writeln('Det mindste tal er ',b);
  END;
END.

```

Bemærk, at typerne af en procedures parametre skal specifceres som typeidentificere. Konstruktionen:

```
PROCEDURE sort(data: ARRAY(.1..100.) OF integer);
```

er altså ikke tilladt. Den korrekte metode er, at associere en typeidentifier med parametertypen, og derefter at bruge denne identifier i parameterlisten:

```

TYPE
  liste = ARRAY(.1..100.) OF integer;

PROCEDURE sort(data: liste);

```

15.2.2 Standardprocedurer

Standardprocedurerne til strengbehandling er beskrevet i afsnit 8.4, standardprocedurerne til filbehandling er beskrevet i afsnittene 13.2, 13.3.3, og 13.4.1, standardprocedurerne til styling af dynamiske variable er beskrevet i afsnit 14.4, og standardprocedurerne til ind- og udlæsning er beskrevet i kapitel 16. Herudover findes de følgende standardprocedurer:

<code>gotoxy(x,y)</code>	Flytter skærmterminalens markør til linie y position x. x og y er udtryk af typen integer. Skærmens øverste venstre hjørne svarer til 0,0.
<code>clreos</code>	Sletter alle tegn fra markøren til slutningen af skærmen.
<code>clreol</code>	Sletter alle tegn til højre for markøren på den nuværende linie. Tegnet under markøren slettes også.
<code>randomize</code>	Klargør randomgeneratoren ved at anvende et tilfældigt tal som udgangspunkt. Det tilfældige tal fås fra Z-80 processorens refresh register (R).
<code>move(s,d,n)</code>	Foretager en kopiering af en blok i lageret. s og d er variable af enhver type, og n er et udtryk af typen integer. Der kopieres n bytes, fra det lagerområde, der starter med den første byte i s, til det lagerområde, der starter med den første byte i d. move udføres af en Z-80 blokkopieringsinstruktion, og er derfor meget hurtig.
<code>fill(d,n,p)</code>	Udfylder en blok i lageret. d er en variabel af enhver type, n er et udtryk af typen integer, og p er et udtryk af typen byte eller af typen char. Værdien p fyldes ind i n på hinanden følgende bytes, startende med den første byte, der optages af d. fill udføres af en Z-80 blokkopieringsinstruktion, og er derfor meget hurtig.

15.3 Funktioner

En funktion er, i lighed med en procedure, enten en standardfunktion eller en selvdefineret funktion.

15.3.1 Funktionserklæringer

En funktionserklæring består af en overskrift, en erklæringsdel og en sætningsdel.

En funktions overskrift er magen til en procedures, bortset fra, at parameterlisten skal efterfølges af et kolon og en typeidentifikator, der bestemmer typen af den værdi, funktionen returnerer. Nogle eksempler:

```
FUNCTION klar: boolean;
FUNCTION konverter(s: str): integer;
FUNCTION max2(a,b: real): real;
```

Resultattypen skal enten være en skalar type (integer, real, boolean, char, selvdefineret skalar eller delinterval), en pointertype eller en strengtype.

En funktions erklæringsdel er magen til en procedures.

Sætningsdelen er en sammensat sætning, dvs. et antal sætninger såkolt af semikolonter og omsluttet af de reserverede ord BEGIN og END. I sætningsdelen skal der være mindst en sætning, der tilskriver en værdi til funktionsidentifieren. Denne tilskrivning bestemmer funktionens resultat. Hvis funktionen udfører et kald til sig selv, kaldes den for en rekursiv funktion.

Det nedenfor viste program anvender en funktion til at finde den største af fire værdier:

```

PROGRAM findmax;
VAR
  i,j,k,l: integer;

FUNCTION max4(a,b,c,d: integer): integer;

FUNCTION max2(a,b: integer): integer;
BEGIN
  IF a>b THEN max2:=a ELSE max2:=b;
END;

BEGIN
  max4:=max2(max2(a,b),max2(c,d));
END;

BEGIN
  readln(i,j,k,l);
  writeln('Det største tal er ',max4(i,j,k,l));
END.

```

Bemærk, at funktionen max2 i det ovenstående eksempel er lokal indenfor den blok, der udgøres af max4 funktionen, og at den derfor ikke kan kaldes fra hovedprogrammet.

Det nedenfor viste program demonstrerer anvendelsen af en rekursiv funktion til at beregne fakulteten af et heltal:

```

PROGRAM beregnfak;
VAR
  n: integer;

FUNCTION fak(n: integer): integer;
BEGIN
  IF n<=1 THEN fak:=1 ELSE fak:=n*fak(n-1);
END;

BEGIN
  readln(n);
  writeln(n,'! = ',fak(n));
END.

```

Bemærk, at en funktions resultattype skal specificeres som en typeidentifier. Konstruktionen:

```
FUNCTION hex(tal,cifre: integer): STRING(.4.);
```

er derfor ikke tilladt. I stedet bør en typeidentifier associeres med typen STRING(.4.), og denne identifier kan derefter bruges til at angive funktionens resultattype:

```

TYPE
  hexstr = STRING(.4.);

FUNCTION hex(tal,cifre: integer): hexstr;

```

Hvis en funktion kalder en eller flere af procedurerne read, readln, write, og writeln, må denne funktion aldrig aktiveres fra et udtryk i en write eller writeln sætning. Grunden til dette ligger i måden, hvorpå write og writeln er implementeret. Ved starten af et kald til en af disse procedurer sættes visse interne variable til nogle værdier, der skal gælde for hele kaldet. Hvis en funktion, der kaldes fra et udtryk i en write eller writeln sætning, derefter udfører endnu et kald, vil dette kald ødelægge informationerne for det "ydre" kald.

15.3.2 Standardfunktioner

Standardfunktionerne til strengbehandling er beskrevet i afsnit 8.4, standardfunktionerne til filbehandling er beskrevet i afsnittene 13.2 og 13.3.1, og standardfunktioner, der kan relateres til pointere, er beskrevet i afsnit 14.4.

15.3.2.1 Aritmetiske funktioner

I de nedenfor viste funktioner er typen af x enten real eller integer og typen af resultatet den samme som x.

abs(x) Den absolute værdi af x.

sqr(x) x kvadreret (x^2).

I de nedenfor viste funktioner er typen af x enten real eller integer og typen af resultatet real.

sin(x) Sinus til x.

cos(x) Cosinus til x.

arctan(x) Arccus tangens til x.

ln(x) Den naturlige logaritme til x.

exp(x) Grundtallet e opløftet i x'ene potens.

sqrt(x) Kvadratroden af x.

int(x) Heltalsdelen af x, dvs. det største hele tal, der er mindre end eller lig med x, for $x \geq 0$, eller det mindste hele tal, der er større end eller lig med x, for $x < 0$.

frac(x) Decimaldelen af x med samme fortegn som x. Beregnes af $\text{frac}(x) = x - \text{int}(x)$.

15.3.2.2 Skalare funktioner

- succ(x)** x er af en skalar datatype, og resultatet er efterfølgeren til x (hvis en sådan findes).
- pred(x)** x er af en skalar datatype, og resultatet er forgængeren til x (hvis en sådan findes).
- odd(x)** x er af typen integer. Returnerer sand (true) hvis x er ulige eller falsk (false) hvis x er lige.

15.3.2.3 Konverteringsfunktioner

- trunc(x)** x er af typen real, og resultatet er det største heltal, der er mindre end eller lig med x, for $x \geq 0$, eller det mindste heltal, der er større end eller lig med x, for $x < 0$. Resultatet er af typen integer.
- round(x)** x er af typen real, og resultatet, der er af typen integer, er den afrundede heltalsværdi af x:
- round(x) = trunc(x+0.5), for $x \geq 0$
 round(x) = trunc(x-0.5), for $x < 0$
- ord(x)** x kan være af enhver skalar datatype, undtagen real. Resultatet, der er af typen integer, er den ordinale værdi af x.

15.3.2.4 Andre standardfunktioner

- pwrten(i)** i er et heltalsudtryk i området $-37 \leq i \leq 37$. Resultatet, der er af typen real, er 10 opløftet til i'ende potens.
- random** Returnerer et tilfældigt tal i området $0.0 \leq r < 1.0$. Resultatet er af typen real.
- random(i)** Returnerer et tilfældigt heltal i området $0 \leq r < i$. Resultatet er af typen integer.
- keypress** Returnerer sand (true) hvis en tast holdes nede på skærmterminalens tastatur, eller falsk (false) hvis ingen nøgler er aktiverede. Tastaturets status undersøges ved at udføre et kald til CP/M console status rutinen.
- hi(i)** i er af typen integer, og resultatet er af typen integer. Den returnerede værdi er den mest betydende byte af i flyttet til den mindst betydende byte. Den mest betydende byte i resultatet er nul.
- lo(i)** i er af typen integer, og resultatet er af typen integer. Den returnerede værdi er den mindst betydende byte af i. Den mest betydende byte i resultatet er nul.

swap(i)	i er af typen integer, og resultatet er af typen integer. Den returnerede værdi findes ved at ombytte den mindst betydende byte og den mest betydende byte.
addr(v)	v kan være enhver variabel eller en procedure- eller funktionsidentifier. Den returnerede værdi, der er af typen integer, er v's adresse i lageret. Bemærk, at hvis v er en arraystruktur, er det tilladt at angive indexudtryk, og hvis v er en post, er det tilladt at udvælge felter.
size(v)	v kan være enhver variabel (se ovenfor) eller en typeidentifier. Den returnerede værdi, der er af typen integer, er størrelsen af v i bytes.

15.4 FORWARD specifikationer

Ved hjælp af FORWARD specifikationer er det muligt at referere til et underprogram førend den faktiske definition af underprogrammet er givet. Dette bliver nødvendigt, hvis to underprogrammer er indbyrdes rekursive, altså hvis det ene underprogram kalder det andet, der igen kalder det første, da det er umuligt at definere begge underprogrammer fuldtud, førend de kaldes.

En FORWARD erklæring foretages ved at separere underprogrammets overskrift fra dets programblok. Overskriften (og den fuldstændige parameterliste) skrives først, efterfulgt af det reserverede ord FORWARD, og selve blokken følger senere i den samme erklæringsdel. Bemærk, at parameterlisten ikke skal gentages, når programblokken defineres. Det nedenfor viste program demonstrerer brugen af en FORWARD specifikation.

```

PROGRAM flipflop;

PROCEDURE flip(n: integer); FORWARD;

PROCEDURE flop(n: integer);
BEGIN
    write('indgang til flop. n=',n);
    IF n>0 THEN flop(n-1);
    write('udgang fra flop. n=',n);
END;

PROCEDURE flip;
BEGIN
    write('indgang til flip. n=',n);
    IF n>0 THEN flop(n-1);
    write('udgang fra flip. n=',n);
END;

BEGIN
    flip(3);
END.

```

Programmet giver den følgende udskrift:

```
indgang til flip. n=3
indgang til flop. n=2
indgang til flip. n=1
indgang til flop. n=0
udgang fra flop. n=0
udgang fra flip. n=1
udgang fra flop. n=2
udgang fra flip. n=3
```

15.5 EXTERNAL specifikationer

EXTERNAL specifikationen bruges til at erklære eksterne underprogrammer, almindeligvis underprogrammer skrevet i andre sprog, for eksempel maskinkode. Et eksternt underprogram har ingen programblok (dvs. ingen erklæringsdel og ingen sætningsdel). Det eneste, der angives, er underprogrammets overskrift, efterfulgt af det reserverede ord EXTERNAL og en heltalskonstant, der definerer adressen på underprogrammet. Eksterne underprogrammer kan have parametre, og formatet af kald til eksterne underprogrammer svarer fuldstændigt til kald af almindelige procedurer og funktioner. Nogle eksempler på erklæringer af eksterne underprogrammer:

```
PROCEDURE moveto(x,y: integer); EXTERNAL $F000;
PROCEDURE drawto(x,y: integer); EXTERNAL $F003;
FUNCTION point(x,y: integer): boolean; EXTERNAL $F006;
PROCEDURE quicksort(VAR d: navneliste); EXTERNAL $1D00;
```

Det er programmørens ansvar at sikre, at der vitterlig findes et underprogram på den nævnte adresse. Yderligere informationer om eksterne underprogrammer og parameteroverførsler gives i afsnit 22.3.

15.6 Strenge som var-parametre

For var-parametre (parametre, der er erklæret med VAR) gælder der, at den formelle parameter og den aktuelle parameter skal være af en og samme datatype. Dette betyder normalt, at procedurer og funktioner med strenge som var-parametre kun kan anvendes på en bestemt strengtype (dvs. strenge med en bestemt maksimal længde), hvilket ofte er irriterende. Ved hjælp af et (*\$V-*) compilerdirektiv er det derfor muligt, at instruere compileren om, at tillade enhver strengtype som aktuel var-parameter, selvom dens maksimale længde ikke stemmer overens med den formelle var-parameter. Ved start af compileren vælges (*\$V+*), og i denne stilling skal typerne være ens. Et eksempel:

```
TYPE
  streng: STRING(.255.);
VAR
  kort: STRING(.16.); lang: STRING(.64.);

PROCEDURE storebogst(VAR s: streng);
VAR
  i: integer;
```

```

BEGIN
  FOR i:=1 TO len(s) DO
    IF s(.i.) IN ('a'..'å') THEN s(.i.):=chr(ord(s(.i.))-32);
END;

BEGIN (*$V*)
  readln(kort); storebogst(kort); writeln(kort);
  readln(lang); storebogst(lang); writeln(lang);
END.

```

15.7 Typeleøse var-parametre

Til visse specielle formål er det en fordel at kunne skrive procedurer og funktioner, der accepterer alle variable, uanset deres type, som var-parametre. Fænomenet kendes for eksempel fra move, fill, blockread, og blockwrite standard procedurene, hvor visse af parametrene kan være variable af enhver type. Ved at undlade at angive en datatype for en var-parameter i et underprograms overskrift, opnås, at denne parameter er typeløs, og dermed at den aktuelle parameter kan være enhver variabel. En typeløs parameter er inkompatibel med alle datatyper, og den kan derfor kun bruges de steder, hvor datatypen er uden betydning, såsom parameter til move, fill, blockread, blockwrite og addr standard rutinerne, samt som adresseangiver i en AT specifikation.

Nedenfor vises en generel funktion til sammenligning af blokke. Rutinen sammenligner en blok på blocksize bytes, startende fra den første byte, der optages af v1, med en tilsvarende blok, der starter ved v2. Hvis blokkene er ens, returneres true, ellers returneres false.

```

FUNCTION blockequal(VAR v1,v2; bsize: integer): boolean;
VAR
  offset: integer; equal: boolean;
BEGIN
  equal:=true; offset:=0;
  WHILE equal AND (offset<bsize) DO
  BEGIN
    equal:=mem(.addr(v1)+offset.)=mem(.addr(v2)+offset.);
    offset:=succ(offset);
  END;
  blockequal:=equal;
END;

```

Hvis de følgende erklæringer er foretaget:

```

TYPE
  matrix = ARRAY(.1..10,1..20.) OF real;
  sektor = ARRAY(.0..127.) OF byte;
VAR
  m1,m2: matrix; s1,s2: sector; i,j: integer;

```

kan blockequal anvendes som vist nedenfor:

```

blockequal(m1,m2,size(matrix))
blockequal(s1,s2,size(sector))
blockequal(m1(.i.),m2(.j.),120)
blockequal(s1(.32.),s1(.64.),32)

```

15.8 Absolutte procedurer og funktioner

Normalt tillader COMPAS Pascal, at procedurer og funktioner er rekursive. Imidlertid bruges denne facilitet sjældent, og det er derfor muligt, ved hjælp af `(*$A+*)` og `(*$A-*)` compilerdirektiverne, at instruere compileren om at generere absolut kode for underprogrammer. Absolut kode er både hurtigere og mere kompakt end rekursiv kode. Ved start af compileren vælges `(*$A-*)` automatisk, og underprogrammer, der oversættes i denne stilling, tillader rekursion. I den modsatte stilling, `(*$A+*)`, genereres absolut kode. Absolutte procedurer og funktioner fungerer kun korrekt, hvis de nedenstående krav er opfyldt:

Procedurens eller funktionens identifier må ikke forekomme i en proceduresætning eller i et udtryk i underprogrammets sætningsdel (direkte rekursion).

Procedurer eller funktioner, der udfører kald til underprogrammet, må ikke kaldes fra underprogrammet (indirekte rekursion).

Kapitel 16

Indlæsning og udlæsning

Indlæsning og udlæsning i læsbar form foretages via en textfil (se afsnit 13.3), der er sammenknyttet med en diskfil eller en af de logiske I/O enheder. For at lette ind- og udlæsninger, kan standardprocedurerne read, readln, write og writeln anvendes med en speciel syntax, der bl.a. tillader et variabelt antal parameter i et kald. Desuden behøver parametrerne ikke nødvendigvis være af typen char (som beskrevet i afsnit 13.3) - de kan også være strenge og numeriske udtryk af typerne integer og real. I de sidstnævnte tilfælde foretages der automatisk en konvertering fra eller til en numerisk streng. Hvis den første parameter i et kald til read, readln, write eller writeln er en textfilvariabel, bliver ind- eller udlæsningen foretaget via denne textfil. I modsat fald foretages ind- eller udlæsningen via input eller output standardfilen.

16.1 read proceduren

read proceduren anvendes til at indlæse tegn, strenge og tal. Formatet af proceduresætningen er:

```
read(v1,v2,...,vn)      eller      read(f,v1,v2,...,vn)
```

hvor v1,v2,...,vn angiver variable af typerne char, STRING, integer eller real. I det første tilfælde indlæses variablene fra skærmbilledet (standardfilen input) og i det andet tilfælde fra textfilen f. Husk, at f skal sammenknyttes med en diskfil eller en I/O enhed, ved hjælp af assign proceduren, og åbnes, ved hjælp af reset proceduren, førend der indlæses værdier fra den.

For en char variabel gælder der, at read læser et enkelt tegn fra filen og gemmer det i variablen. Hvis inputfilen er en diskfil, er eof sand, hvis det næste tegn er CTRL/Z, og eoln sand, hvis det næste tegn er CR eller CTRL/Z. Hvis inputfilen er en logisk I/O enhed, er eof sand, hvis det læste tegn var CTRL/Z, og eoln sand, hvis det læste tegn var CR eller CTRL/Z.

For en streng gælder der, at read læser så mange tegn som muligt ind i strengen, undtagen hvis et linieskift mødes, eller hvis filen slutter. Det maksimale antal læste tegn er givet ved strengens maksimale længde. Der gøres ikke forskel på blanktegn og andre tegn. Efter læsning af en streng gælder der, at eof er sand, hvis strengen blev afsluttet med CTRL/Z, og eoln er sand, hvis strengen blev afsluttet med CR eller CTRL/Z.

Ved indlæsning af en numerisk værdi (integer eller real) forventer read en tekststreng, der følger reglerne for numeriske konstanter, som beskrevet i afsnit 2.2. Blanktegn, tab-tegn (HT) og linieskift (CR og LF tegn) foran tallet ignoreres. Talstrengen må ikke være længere end 30 tegn, og den skal efterfølges af et blanktegn, et HT tegn, et CR tegn, eller et CTRL/Z tegn. Hvis det numeriske format ikke er korrekt, gives der en I/O fejl. Ellers konverteres talstrengen til en værdi af den givne type, og værdien gemmes derefter i variablen. Hvis inputfilen er en diskfil, og

hvis talstrengen blev afsluttet med et blanktegn eller et HT tegn, gælder der, at det næste kald til read vil starte med tegnet umiddelbart efter blanktegnet eller HT tegnet. Ellers gælder der, at eof er sand, hvis talstrengen blev afsluttet med CTRL/Z, og eoln er sand, hvis talstrengen blev afsluttet med CR eller CTRL/Z. Et specialtilfælde er, når eof er sand før indlæsningen startes (eller når det først læste tegn fra en I/O enhed er CTRL/Z). I stedet for at tilskrive en ny værdi til variablen lader read den gamle værdi blive stående.

Hvis inputfilen refererer til CON: enheden, eller hvis standardfilen input bruges i (*\$B+*) stillingen, gælder der specielle regler for indlæsningen: Ved kaldet til read indlæses en linie fra skærterminalen, og denne gemmes i en intern buffer. Selve læsningen af variablene foretages derefter fra denne buffer. Under indlæsningen af linien kan de følgende editeringsnøgler anvendes:

BACKSPACE	Sletter det sidst indtastede tegn. Denne funktion findes normalt på tastaturet som BS, BACKSPACE eller en venstrepil, men den kan altid genereres ved at trykke CTRL/H.
DEL	Samme som BACKSPACE. På de fleste tastaturer findes denne funktion som DEL eller RUBOUT.
CTRL/X	Sletter hele indtastningen.
RETURN	Afslutter indtastningen. Denne funktion findes normalt på tastaturet som RETURN, ENTER eller NEWLINE.

Bemærk, at det afsluttende CR tegn ikke udskrives. Den indlæste linie bliver automatisk efterfulgt af CTRL/Z tegn. Derfor gælder der, at hvis der gives færre værdier i inputlinien end i parameterlisten, vil overskydende char variable blive sat til CTRL/Z, overskydende strenge vil være tomme, og overskydende numeriske variable vil ikke blive ændret.

Normalt kan der indlæses op til 127 tegn på en linie, men denne grænse kan ændres ved at tilskrive en ny værdi til den prædefinerede variabel buflen. Værdien skal være et heltal mellem 0 og 127. En sådan tilskrivning berører kun den næste indlæsning; når først denne er foretaget, bliver grænsen igen sat til 127. Et eksempel:

```
write('navn (op til 24 tegn)? ');
buflen:=24; read(navn);
```

16.2 readln proceduren

readln proceduren er identisk med read proceduren, bortset fra, at efter endt læsning af den sidste variabel, bliver resten af linien oversprunget, dvs. alle tegn til og med den næste CR/LF sekvens (eller til og med det næste CR tegn for en logisk enhed) bliver læst. Formatet af proceduresætningen er:

```
readln(v1,v2,...,vn)      eller      readln(f,v1,v2,...,vn)
```

Efter et kald til readln gælder der, at det næste kald til read eller readln starter med det første tegn på den næste linie. eoln er altid falsk efter et kald til readln, med mindre eof er sand. readln kan også anvendes uden variable:

```
readln      eller      readln(f)
```

Når readln anvendes på en fil, der refererer til CON: enheden, er den eneste forskel fra et tilsvarende kald til read, at en CR/LF sekvens udskrives til skærmen efter endt indlæsning af linien.

16.3 write proceduren

write proceduren anvendes til at udlæse strenge, booleans, og numeriske værdier. Formatet af proceduresætningen er:

```
write(pl,p2,...,pn)      eller      write(f,pl,p2,...,pn)
```

hvor pl,p2,...,pn angiver såkaldte write-parametre, og f angiver en textfil. Husk, at f skal sammenknyttes med en diskfil eller en I/O enhed, ved hjælp af assign proceduren, og åbnes, ved hjælp af rewrite proceduren, førend der kan foretages udlæsninger til den. Write-parametrene kan antage forskellige former, alt efter hvilke datatyper der udlæses (m, n og i angiver udtryk af typen integer, ch angiver et udtryk af typen char, s angiver et strengudtryk, b angiver et udtryk af typen boolean, og r angiver et udtryk af typen real):

ch Tegnet ch udlæses.

ch:n Tegnet ch udlæses højrejusteret i et felt på n tegn.

s Strengudtrykket s udlæses. Bemærk, at tegn-arrays også kan udlæses, da disse er kompatible med strenge.

s:n Strengudtrykket s udlæses højrejusteret i et felt på n tegn.

b TRUE eller FALSE udskrives alt efter b's værdi.

b:n TRUE eller FALSE udskrives højrejusteret i et felt på n tegn.

i Værdien af udtrykket i udlæses i frit format.

i:n Værdien af udtrykket i udlæses højrejusteret i et felt på n tegn.

r Værdien af udtrykket r udlæses i exponentiel notation i et felt på 18 tegn:

```
r>=0.0:  bbd.dddddddddddEtdd
r<0.0:  b-d.dddddddddddEtdd
```

hvor b angiver et blanktegn, d angiver et ciffer og t angiver enten '+' eller '-'.

r:n Værdien af udtrykket r udlæses i exponentiel notation.
 Det generelle format er:

```
r>=0.0: <blanks>d.<digits>Eddd  

r<0.0: <blanks>-d.<digits>Eddd
```

hvor <blanks> er nul eller flere blanktegn og <digits> er mellem 1 og 10 cifre. Da der altid udlæses mindst et ciffer efter decimalkommaet, er det mindste felt, der kan forekomme, 7 tegn langt (8 for negative tal). Hvis n er større end 16 (17 for negative tal), udlæses der n-16 (n-17 for negative tal) blanktegn foran tallet.

r:n:m Værdien af udtrykket r udlæses højrejusteret i fastkommanotation i et felt på n tegn med m decimaler. m skal være mellem 0 og 24, ellers vælges exponentiel notation.

16.4 writeln proceduren

writeln proceduren er identisk med write proceduren, bortset fra, at der udlæses en CR/LF sekvens efter den sidste værdi. Formatet af proceduresætningen er:

```
writeln(p1,p2,...,pn)      eller      writeln(f,p1,p2,...,pn)
```

Et linieskift kan udlæses alene ved at udelade write-parametrene:

```
writeln      eller      writeln(f)
```

Kapitel 17

Include-filer

Det kan ske, at et programs kildetekst bliver så stor, at den ikke kan være i COMPAS Pascal editorens buffer på en gang. I så tilfælde kan kildeteksten inddeltes i mindre segmenter, der sammensættes under oversættelsen af programmet ved hjælp af "include-fil" compilerdirektiver.

Syntaksen for et compilerdirektiv, der instruerer compileren om, at medtage en anden kildetekst i oversættelsen af den nuværende kildetekst, er som følger:

```
(*$I filnavn*)
```

hvor filnavn er et CP/M filnavn. Hvis filtypen udelades, bliver '.PAS' automatisk valgt. Blanktegn foran og efter filnavnet ignoreres. Det anbefales, at lade include-file compilerdirektivet stå alene på linien.

Compileren kan ikke holde styr på nestede include-filer: Hvis en include-fil medtager en anden fil, fortsætter læsningen af den første fil ikke, når den anden fil slutter.

Include-filer kan også anvendes til at opbygge biblioteker af "macrodefinitioner" på disken. Sådanne macrodefinitioner kan medtages i oversættelsen af et program når som helst det er påkrævet, blot ved at angive deres filnavn i et "include-fil" compilerdirektiv. Antag, som et eksempel, at den følgende macrodefinition findes på disken under filnavnet A:MINMAX.LIB:

```
TYPE tal = 0..99;

FUNCTION max(a,b: tal): tal;
BEGIN if a>b THEN max:=a ELSE max:=b END;

FUNCTION min(a,b: tal): tal;
BEGIN if a<b THEN max:=a ELSE max:=b END;
```

Et program, der anvender typen tal og funktionerne max og min, kan da skrives som:

```
PROGRAM minmax;
(*$I A:MINMAX.LIB*)
VAR
  x,y: tal;
BEGIN
  write('skriv to tal: '); readln(x,y);
  writeln('det største tal er ',max(x,y));
  writeln('det mindste tal er ',min(x,y));
END.
```

Da de enkelte sektioner i en erklæringsdel kan skrives i enhver rækkefølge og eventuelt flere gange, kan macrodefinitioner både erklære konstanter, typer, variable og underprogrammer.

Kapitel 18

Kædning af programmer

I COMPAS Pascal er det muligt for et program at starte et andet. Dette kan udnyttes når programmer bliver så store, at de ikke kan være i lageret på en gang. Kædning af programmer udføres via execute og chain procedurerne. Formatet af kald til disse procedurer er:

```
execute(f)      og      chain(f)
```

hvor f er en filvariabel (af enhver filtype), der tidligere er tilskrevet et filnavn ved hjælp af assign proceduren. Forudsat at det givne program eksisterer, bliver det læst ind i lageret og startet.

execute proceduren kan anvendes til at indlæse og opstarte enhver CP/M programfil ('.COM' fil), for eksempel en fil, der er genereret med COMPAS Pascal's PROGRAM kommando. Filen læses ind i lageret startende fra adresse \$100, og startes i adresse \$100, som CP/M standarden foreskriver.

chain proceduren kan kun anvendes til at aktivere COMPAS Pascal objektfiler ('.OBJ' filer), dvs. filer, der er genereret med COMPAS Pascal's OBJECT kommando. Filen læses ind i lageret på startadressen for det nuværende program (dvs. på den adresse, der blev specificeret som startadresse for det nuværende program, da det blev oversat). Kravet for, at chain fungerer korrekt, er altså, at det nuværende program og det program, der aktiveres, er oversat til den samme startadresse.

Yderligere detaljer om PROGRAM og OBJECT kommandoerne gives i COMPAS Pascal brugermanualen. Kort fortalt er forskellen på disse kommandoer, at PROGRAM kommandoen medtager både run-time programdelen og programmets kode i filen (og skaber dermed en programfil, der kan køre helt af sig selv), medens OBJECT kommandoen kun medtager programkoden. En fil, der er genereret af OBJECT kommandoen, kan derfor kun køre, hvis run-time programdelen allerede findes i lageret.

Hvis den fil, der refereres til i et kald til execute eller chain, ikke findes, gives en I/O fejl, med mindre kaldet er oversat i (*\$I-*) stillingen. I sidstnævnte tilfælde fortsætter programmet med sætningen efter kaldet, og iores funktionen giver nu en værdi forskellig fra nul. Denne værdi skal undersøges af programmet, før yderligere I/O operationer kan foretages. Et eksempel:

```
VAR
  f: FILE;
  i: integer;
BEGIN
  assign(f,'B:PROG3.COM'); (*$I-*) execute(f) (*$I+*);
  i:=iores;
  writeln('B:PROG3.COM findes ikke.');
END.
```

Data kan overføres mellem kædede programmer ved hjælp af fælles globale variable eller absolut adresserede variable. Naturligvis kan en fil også anvendes, men denne metode er noget langsommere end de to andre.

Dataoverførsel ved hjælp af fælles globale variable kræver, at erklæringerne af de fælles variable foretages som de første erklæringer i begge programmer, og desuden, at begge programmer er oversat til den samme lagerstørrelse. Et eksempel:

```

PROGRAM hovedprog;
VAR
  i,j,k: integer;
  f: FILE;
BEGIN
  readln(i,j); k:=i*j;
  assign(f,'A:UNDERPRG.COM'); execute(f);
END.

PROGRAM underprg;
VAR
  i,j,k: integer;
BEGIN
  writeln(i,' gange ',j,' er ',k);
END.

```

Dataoverførsel ved hjælp af absolut adresserede variable foregår typisk ved, at man definerer en postvariabel, der indeholder felter med alle de informationer, der skal overføres, og derefter erklærer en variabel af denne type på en fast adresse (ved hjælp af AT specifikationen).

Bemærk, at det ikke kan lade sigøre at køde programmer i direct mode, eller, med andre ord, at et program, der er startet fra COMPAS med RUN kommandoen, ikke må starte andre programmer.

Et program kan undersøge, om det blev startet fra en CP/M kommandolinie eller fra et kald til execute eller chain ved hjælp af et flag, der gemmes i adresse \$80. Værdien \$FF i denne byte angiver, at programmet blev startet fra execute eller chain, og andre værdier, at programmet blev startet fra CP/M. Det nedenfor viste program demonstrerer dette:

```

PROGRAM starttest;
VAR
  startflag: byte AT $80;
BEGIN
  IF startflag=255 THEN
    writeln('startet fra execute eller chain.') ELSE
    writeln('startet fra CP/M.');
END.

```

Kapitel 19

In-line maskinkode

Til tider kan det være ønskeligt at skrive subrutiner eller dele af programmer i maskinkode for at optimere kørselshastigheden. Dette er muligt i COMPAS Pascal ved hjælp af CODE sætninger. En CODE sætnings syntaks er ganske simpel: Den består af det reserverede ord CODE efterfulgt af en eller flere konstanter, variabelidentificere eller programtællerreferencer, adskilt af kommaer.

Konstanterne er enten skrevne konstanter eller konstantidentifiere. En skreven konstant genererer en byte kode, hvis den er i området 0..255 (\$00..\$FF); ellers genereres to bytes i omvendt format (mindst betydende byte først). En konstantidentifier genererer altid to bytes kode. En variabelidentifier genererer to bytes kode, der angiver adressen på variablen. En programtællerreference består af en stjerne (*), eventuelt efterfulgt af et plus (+) eller minus (-) og en heltalskonstant. I førstnævnte tilfælde genereres to bytes kode, der indeholder programtællerens nuværende værdi (dvs. adressen på den første byte). I det andet tilfælde adderes eller subtraheres det angivne offset, før adressen kodes.

I det nedenfor viste program anvendes en CODE sætning til at definere en procedure, der konverterer alle bogstaver i en streng til store bogstaver.

```

PROGRAM teststore; (*$A+*)
TYPE
  str = STRING(.64.);
VAR
  s: str;

PROCEDURE store(VAR streng: str);
BEGIN CODE
  $2A,streng,      (* LD   HL,(streng)    *)
  $46,              (* LD   B,(HL)      *)
  $04,              (* INC  B          *)
  $05,              (*     Ll: DEC  B      *)
  $CA,*+20,          (*     JP  Z,L2      *)
  $23,              (*     INC  HL      *)
  $7E,              (* LD   A,(HL)      *)
  $FE,$61,           (* CP   'a'        *)
  $DA,*-9,            (* JP  C,Ll      *)
  $FE,$7B,           (* CP   'z'+1      *)
  $D2,*-14,          (* JP  NC,Ll      *)
  $D6,$20,           (* SUB  20H       *)
  $77,              (* LD   (HL),A      *)
  $C3,*-20;          (* JP  Ll          *)
END;                (* L2: EQU  $          *)

BEGIN
  write('indtast en streng: '); readln(s);
  store(s); writeln(s);
END.

```

Bemærk, at den eneste grund til, at der anvendes JP instruktioner, er, at det da bliver muligt at demonstrere programtællerreferencer. I den ovenstående rutine ville det naturligvis være kortere at anvende JR instruktioner.

CODE sætninger kan frit blandes med andre sætninger i en programbloks sætningsdel, og CODE sætninger må anvende alle CPU registre (bemærk dog, at stack-pointeren (SP) skal have det samme indhold ved indgang som ved udgang).

Kapitel 20

CP/M funktionskald

Til kald af CP/M's BDOS og BIOS rutiner findes der i COMPAS Pascal to standardprocedurer, kaldet bdos og bios, og fire standardfunktioner, kaldet bdos, bios, bdosb, og biosb. Disse rutiner bør kun anvendes af øvede programmører, der fuldt ud forstår deres implikationer.

- | | |
|-------------------|--|
| bdos(f,p) | Denne procedure bruges til at kalde CP/M BDOS rutiner. f og p er udtryk af typen integer (hvis rutinen ikke kræver en indgangsparameter, kan p og det foranst  ende komma udelades). f loades ind i C registeret, p (hvis angivet) loades ind i DE registerparret, og derefter udf  res et kald til adresse 5, hvilket aktiverer BDOS rutinen. bdos kan ogs   anvendes som en funktion. I s   tilf  lde er resultatet (af typen integer) den v  rdi, der returneres i HL registerparret. |
| bios(f,p) | Denne procedure bruges til at kalde BIOS rutiner. f og p er udtryk af typen integer (hvis rutinen ikke kr  ver en indgangsparameter, kan p og det foranst  ende komma udelades). f angiver nummeret p   den rutine der skal kaldes, hvor 0 svarer til WBOOT, 1 til CONST, osv. (med andre ord, adressen p   rutinen, der skal kaldes, udregnes ved at l  gge f*3 til den adresse, der er indeholdt i adresse 1 og 2). Hvis p angives, loades denne v  rdi ind i BC registerparret f  r kaldet. bios kan ogs   anvendes som en funktion. I s   tilf  lde er resultatet (af typen integer) den v  rdi, der returneres i HL registerparret. |
| bdosb(f,p) | Denne funktion svarer til bdos, bortset fra, at resultatet er den v  rdi, der returneres i A registeret. |
| biosb(f,p) | Denne funktion svarer til bios, bortset fra, at resultatet er den v  rdi, der returneres i A registeret. |

Informationer om BDOS og BIOS rutiner findes i manualerne "CP/M Interface Guide" og "CP/M Alteration Guide", der begge udgives af Digital Research.

Kapitel 21

Brugerdefinerede I/O drivere

Til visse formål er det ønskeligt eller ligefrem nødvendigt for et program, at definere dets egne low-level I/O drivere, dvs. rutiner der foretager den grundlæggende ind- og udlæsning af tegn fra og til ydre enheder. I COMPAS Pascal findes de følgende grundlæggende I/O drivere (bemærk, at disse subrutiner ikke er tilgængelige som standardprocedurer og funktioner):

```
FUNCTION const: boolean;
FUNCTION conin: char;
PROCEDURE conout(ch: char);
PROCEDURE lstout(ch: char);
PROCEDURE auxout(ch: char);
FUNCTION auxin: char;
PROCEDURE usrout(ch: char);
FUNCTION usrin: char;
```

const rutinen kaldes af keypress funktionen, conin og conout rutinerne bruges af CON:, TRM: og KBD: enhederne, lstout rutinen bruges af LST: enheden, auxout og auxin rutinerne bruges af AUX: enheden, og usrout og usrin rutinerne bruges af USR: enheden.

Hvis intet andet angives, bruger de ovennævnte drivere de tilsvarende rutiner i CP/M systemets BIOS (dvs. const bruger CONST, conin bruger CONIN, conout bruger CONOUT, lstout bruger LIST, auxout bruger PUNCH, auxin bruger READER, usrout bruger CONOUT, og usrin bruger CONIN). Disse definitioner kan imidlertid ændres af et program, ved at tilskrive adressen på en driver til en af de følgende standardvariable:

csaddr	adressen på const funktionen
ciaddr	adressen på conin funktionen
coaddr	adressen på conout proceduren
loaddr	adressen på lstout proceduren
aoaddr	adressen på auxout proceduren
aiaddr	adressen på auxin funktionen
uoaddr	adressen på usrout proceduren
uiaddr	adressen på usrin funktionen

Driverprocedurer og driverfunktioner skal følge de ovennævnte definitioner, dvs. en const driver skal være en boolean funktion, en conin, auxin eller usrin driver skal være en char funktion, og en conout, lstout, auxout eller usrout driver skal være en procedure med en value-parameter af typen char.

Det nedenfor viste program definerer og aktiverer en ny driver for LST: enheden. Udover at udskrive tegn holder den nye driver automatisk styr på printerens linie og position. Foran hver linie udskrives en venstremargin, der består af et brugerdefineret antal blanktegn, og perforeringen mellem siderne overspringes automatisk. Desuden bliver form-feed tegn omsat til et passende antal line-feed tegn. Enkelttegn udskrives fra driveren ved at kalde LIST rutinen (rutine nummer 4) i BIOS'en.

```

PROGRAM listdriver; (*$A+*)
CONST
  sidelaengde = 72;          (* sidelaengde i linier *)
  bundmargin = 6;            (* overspring perforering *)
  venstremargin = 8;         (* venstremargin *)
VAR
  lstlin,lstpos: integer;

PROCEDURE lstout(ch: char);
VAR i: integer;
BEGIN
  IF ch>=' ' THEN
  BEGIN
    IF lstpos=0 THEN
    BEGIN
      FOR i:=1 TO venstremargin DO bios(4,ord(' '));
      lstpos:=venstremargin;
    END;
    bios(4,ord(ch)); lstpos:=lstpos+1;
  END ELSE
  IF ch=@13 THEN
  BEGIN
    bios(4,13); lstpos:=0;
  END ELSE
  IF ch=@10 THEN
  BEGIN
    bios(4,10); lstlin:=lstlin+1;
    IF lstlin=sidelaengde-bundmargin THEN
    BEGIN
      FOR i:=1 TO bundmargin DO bios(4,10);
      lstlin:=0;
    END;
  END ELSE
  IF ch=@12 THEN
  BEGIN
    FOR i:=lstlin TO sidelaengde-1 DO bios(4,10);
    lstlin:=0;
  END;
  END;
  BEGIN
    lstpos:=0; lstlin:=0; loaddr:=addr(lstout);
    writeln(lst,'LST DRIVER TEST:');
    writeln(lst,'Dette giver tre blanke linier...');
    write(lst,@10@10@10);
    writeln(lst,'Dette giver et sideskift...');
    write(lst,@12);
  END.
END.
```

I begyndelsen af programmet sættes printerens linie og position til nul, og LST: driver adressen ændres til adressen på den brugerdefinerede outputrutine. Herefter udskrives en række strenge til lst filen (der er prædefineret, og altid refererer til LST: enheden). Disse strenge bliver af systemet overgivet til lstout proceduren et tegn ad gangen. Bemærk, at brugerdefinerede I/O drivere under ingen omstændigheder må kalde read, readln, write og writeln procedurerne.

Kapitel 22

Interne dataformater

I de følgende beskrivelser angiver symbolet 'addr' den første byte, der optages af den givne variabel. Det er denne adresse, standardfunktionen `addr` returnerer.

22.1 Grundlæggende datatyper

Variablene af de grundlæggende datatyper kan grupperes i strukturer, men dette berører ikke deres interne formater.

22.1.1 Skalarer

Integer delintervaller, hvor begge grænser er mellem 0 og 255, booleans, tegn (char variable), og selvdefinerede skalarer, med mindre end 256 elementer, optager en byte i lageret, der angiver den ordinale værdi af variablen.

Integers, integer delintervaller, hvor en eller begge grænser er udenfor området 0..255, og selvdefinerede skalarer, med mere end 256 mulige værdier, optager to bytes i lageret. Disse bytes udgør en 16-bit 2's komplement værdi. Den mindst betydende byte gemmes først.

22.1.2 Reals

En variabel af typen real optager seks bytes, hvoraf fem bytes (40 bit) udgør mantissen og en byte (8 bits) udgør exponenten. Exponenten gemmes i den første byte, og mantissen i de følgende fem bytes, med den mindst betydende byte først:

<code>addr+0</code>	Exponent.
<code>addr+1</code>	Mantissens mindst betydende byte.
:	
<code>addr+5</code>	Mantissens mest betydende byte.

Exponenten er et binært tal med et offset på \$80, der angiver den potens af 2, mantissen skal multipliceres med. Således svarer en exponent på \$84 til, at mantissen skal multipliceres med $2^{($84 - \$80)} = 2^4 = 16$. Hvis exponenten er \$00, opfattes hele tallet som værende 0. Værdien af mantissen findes ved, at dividere det fortegnsløse 40-bits heltal, der udgøres af de fem bytes, med 2^{40} . Mantissen er altid normaliseret, dvs. den mest betydende bit (bit 7 i `addr+5`) skal altid opfattes som værende 1. Fortegnet gemmes i den mest betydende bit: Er denne 1, er tallet negativt, er den 0, er tallet positivt.

22.1.3 Strenge

En streng optager dens maksimale længde plus en byte i lageret. Den første byte angiver længden, og de efterfølgende bytes indeholder tegnene, med det første tegn på den laveste adresse. I

den nedenfor viste figur angiver n den nuværende længde af strengen og m den maksimale længde.

addr+0	Nuværende længde (n).
addr+1	Første tegn.
addr+2	Andet tegn.
:	
addr+n	Sidste tegn.
addr+n+1	Ubrugt.
:	
addr+m	Ubrugt.

22.1.4 Mængder

Et element i en mængde optager en bit, og da der aldrig er mere end 256 elementer i en mængde, fylder en mængde aldrig mere en 32 (256/8) bytes.

Hvis en mængde indeholder mindre end 256 elementer, er det givet, at nogle af de 256 bits permanent vil være nul, og det er derfor ikke nødvendigt at gemme disse bits. Udfra et lager-økonomisk synspunkt ville den mest effektive måde at gemme mængder på da være at "bortskære" alle unødige bits og rotere de resterende bits, således at det første element i mængden optager den første bit i den første byte. Imidlertid er sådanne rotationer relativt tidskrævende, og COMPAS implementerer derfor et kompromis: Kun de bytes, der er statisk nul (dvs. de bytes hvoraf ingen bits bruges til at repræsentere mængden), gemmes ikke. Denne kompressionsmetode er meget hurtig og i de fleste tilfælde lige så effektiv som den førstnævnte.

Antallet af bytes, der optages af en given mængdevariabel, kan beregnes af $(\text{max DIV } 8) - (\text{min DIV } 8) + 1$, hvor min og max er de nedre og øvre grænser i den grundlæggende skalare type. Lageradressen på et givet element beregnes af:

```
memaddr = addr + (e DIV 8) - (min DIV 8)
```

og bitadressen, i byten på lageradressen memaddr, beregnes af:

```
bitaddr = e MOD 8
```

hvor e er elementets ordinale værdi.

22.1.5 Fil interface blokke

Til enhver filvariabel i et program svarer der en fil interface blok (FIB) i datalageret. En FIB optager 176 bytes, og er delt i to sektioner: En kontrolsektion (de første 48 bytes) og en sektorbuffer (de sidste 128 bytes). Kontrolsektionen indeholder informationer om den diskfil eller den logiske I/O enhed, der er knyttet til filvariablen. Sektorbufferen bruges som buffer under I/O operationer på diskfiler. Formatet af en FIB er som følger:

addr+0	Flag-byte.
addr+1	Filtypen.
addr+2	Tegnbuffer.
addr+3	Sektorbuffer-pointer.
addr+4	Antal elementer i filen (LSB).
addr+5	Antal elementer i filen (MSB).
addr+6	Elementlængde i bytes (LSB).
addr+7	Elementlængde i bytes (MSB).
addr+8	Nuværende elements nummer (LSB).
addr+9	Nuværende elements nummer (MSB).
addr+10	Ubrugt (reserveret).
addr+11	Ubrugt (reserveret).
addr+12	Første byte i CP/M FCB'en.
:	
addr+47	Sidste byte i CP/M FCB'en.
addr+48	Første byte i sektorbufferen.
:	
addr+175	Sidste byte i sektorbufferen.

Flag-byten indeholder fire enkelt-bits flag, der angiver filens status:

bit 0	Input flag. Sat hvis indlæsning er tilladt.
bit 1	Output flag. Sat hvis udlæsning er tilladt.
bit 2	Skrive-semafor. Sat hvis der er skrevet til sektorbufferen.
bit 3	Læse-semafor. Sat hvis indholdet af sektorbufferen er ubrugeligt.

Filtypefeltet angiver, om en diskfil eller en logisk I/O enhed er tilskrevet til filvariablen. De følgende værdier kan forekomme:

0	Console enheden (CON:).
1	Terminal enheden (TRM:).
2	Keyboard enheden (KBD:).
3	List enheden (LST:).
4	Auxiliary enheden (AUX:).
5	Den brugerdefinerede enhed (USR:).
6	En diskfil er tilskrevet til filvariablen.

Tegnbufferen bruges kun ved indlæsning fra textfiler, og indeholder det næste tegn, der skal læses (eller det sidst læste tegn, hvis der læses fra en logisk enhed). Sektorbuffer-pointeren angiver offsettet fra den første byte i sektorbufferen til den byte, der skal behandles næste gang. De følgende tre felter bruges kun af selvdefinerede filer (FILE OF type) og typeløse filer (FILE). Hvert felt angiver en 16-bits værdi i omvendt format. De to bytes på addr+10 og addr+11 anvendes ikke for nuværende, men de er reserverede til fremtidige udvidelser. De 36 bytes fra addr+12 til addr+47 indeholder en CP/M file controller block (FCB). En udførlig beskrivelse af FCB'ens format findes i manualen "CP/M Interface Guide".

Når en filvariabel er sammenkyttet med en logisk I/O enhed, bruges kun de tre første bytes i FIB'en.

Det ovenfor viste FIB format gælder for alle selvdefinerede filer (FILE OF type) og textfiler (text). En typeløs fils FIB har ingen sektorbuffer, da data flyttes direkte fra/til diskfilen til/fra variablen. Længden af en typeløs fils FIB er altså kun 48 bytes.

22.1.6 Pointere

Ett pointei består af to bytes, der angiver en 16-bits adresse i lageret. Den mindst betydende byte lagres på den laveste adresse. Pointerværdien NIL lagres som adressen 0.

22.2 Datastrukturer

De grundlæggende datatyper kan samles i datastrukturer på tre forskellige måder: Arraystrukturer, poster og diskfiler. Gruppeningen berører på ingen måde det interne format af de grundlæggende datatyper. Således har et element i en datastruktur altid det samme interne format som en enkeltvariabel af samme type.

22.2.1 Arraystrukturer

Elementet med den laveste indexværdi lagres på den laveste adresse. Hvis arraystrukturen er flerdimensional, optælles den sidste dimension først. En arraystruktur erklæret ved:

```
a: ARRAY(.1..3,1..3:)
```

bliver lagret på denne måde:

laveste adresse:	a(.1,1.) a(.1,2.) a(.1,3.) a(.2,1.) : højeste adresse:
	a(.3,3.)

22.2.2 Poster

Det første felt i en post lagres på den laveste adresse. Hvis posten ikke indeholder en variant-del, er længden givet ved summen af længderne på hver enkelt felt. I modsat fald er længden givet ved længden af den faste del plus længden af mærkefeltet (hvis det eksisterer) plus længden af den største variant. Hver variant starter på den samme lageradresse.

22.2.3 Diskfiler

I modsætning til andre datastrukturer bliver elementerne i en diskfil ikke gemt i lageret men i stedet i en fil på en disk. En diskfil styres via en FIB som beskrevet i afsnit 22.1.5. COMPAS Pascal arbejder med to forskellige diskfiltyper: Textfiler og random access datafiler.

22.2.3.1 Textfiler

En textfil er inddelt i linier. Hver linie består af nul eller flere tegn og afsluttes af en CR/LF sekvens. ASCII værdien af et CR tegn er \$0D, og ASCII værdien af et LF tegn er \$0A. Filen afsluttes af et CTRL/Z tegn. ASCII Værdien af et CTRL/Z tegn er \$1A.

22.2.3.2 Random access filer

En random access fil består af en følge af elementer, der alle er af samme længde og interne format. Random access filformatet benyttes af selvdefinerede filer. For at opnå fuld udnyttelse af diskfilens kapacitet, lagres elementerne umiddelbart efter hinanden uafhængigt af sektogrænser. De første fire bytes af en random access fil angiver antallet af elementer i filen og længden af elementerne:

sektor 0, byte 0:	Antal elementer (LSB).
sektor 0, byte 1:	Antal elementer (MSB).
sektor 0, byte 2:	Elementlængde i bytes (LSB).
sektor 0, byte 3:	Elementlængde i bytes (MSB).

Det første elements data følger umiddelbart efter disse kontrolbytes.

22.3 Parametre

Parametre overføres til procedurer og funktioner via Z-80 processorens stak. Normalt har dette ingen betydning for programmøren, da COMPAS Pascal sørger for både at PUSHe parametrene før et kald og POPpe dem i begyndelsen af underprogrammet. Skulle programmøren imidlertid ønske at kalde eksterne maskinkoderutiner fra et Pascalprogram, skal disse underprogrammer skrives sådan, at de selv sørger for at POPpe parametrene fra stakken.

Ved indgang til et eksternt underprogram ligger returadressen øverst på stakken. Eventuelle parametre findes "under" returadressen (dvs. på højere adresser i lageret). For at få adgang til parametrene må returadressen POPpes af. Derefter kan parametrene POPpes, og til sidst skal returadressen PUSHes tilbage.

22.3.1 Var-parametre

En var-parameter overføres på stakken som et 16-bits tal, der angiver adressen på den første byte, der optages af den aktuelle parameter.

22.3.2 Value-parametre

Hvis en parameter er en value-parameter, er formatet af de bytes der overføres afhængigt af parametrens type.

22.3.2.1 Skalarer

Alle skalarer, undtagen reals, dvs. integers, booleans, tegn (char), selvdefinerede skalarer, og delintervaller overføres på stakken som 16-bits ord. Hvis værdien er af en type, der normalt kun optager en byte, er den mest betydnende byte nul. Normalt POPpes et 16-bits ord af stakken med en POP HL, POP DE eller POP BC instruktion.

22.3.2.2 Reals

En real overføres på stakken som tre 16-bits ord. Hvis disse ord POPpes med den følgende instruktionssekvens:

```
POP    HL
POP    DE
POP    BC
```

vil L indeholde exponenten, H mantissens femte byte (mindst betydende), E den fjerde byte, D den tredje byte, C den anden byte, og B den første byte (mest betydende).

22.3.2.3 Strenge

Når en streng er placeret øverst på stakken, indeholder den byte, der adresseres af SP registeret, længden af strengen, og bytene fra SP+1 til SP+n (hvor n er længden af strengen) indeholder strengens tegn. Den følgende instruktionssekvens POPper en streng af stakken og lagrer den i STRBUF:

```
LD    DE,STRBUF
LD    HL,0
LD    B,H
ADD  HL,SP
LD    C,(HL)
INC   BC
LDIR
LD    SP,HL
```

22.3.2.4 Mængder

En mængde optager altid 32 bytes på stakken (mængder komprimeres kun når de lagres). Den følgende instruktionssekvens POPper en mængde af stakken og lagrer den i SETBUF:

```
LD    DE,SETBUF
LD    HL,0
ADD  HL,SP
LD    BC,32
LDIR
LD    SP,HL
```

Den mindst betydende byte i mængden lagres på den laveste adresse i STRBUF.

22.3.2.5 Pointere

En pointer overføres på stakken som et 16-bits ord, der angiver en adresse i lageret. Pointerværdien NIL svarer til adressen 0.

22.3.2.6 Arraystrukturer og poster

Arraystrukture og poster bliver faktisk ikke PUSHet på stakken, selvom de overføres som value-parametre. I stedet overføres et 16-bits ord, der angiver adressen på den første byte, der optages af parameteren. Herefter er det op til programmøren at POPpe denne adresse og anvende den som kildeadresse i en blokkopiering.

22.4 Funktionsresultater

Brugerdefinerede eksterne funktioner bør nøje overholde de nedenfor angivne regler, når de returnerer resultater.

Værdier af alle skalare typer, undtagen reals, returneres i HL registerparret. Hvis resultatet er af en type, der normalt kun fylder en byte, skal L indeholde resultatet og H indeholde 0.

Reals returneres i BC, DE, og HL registerparrene. B, C, D, E og H skal indeholde mantissen (mest betydende byte i B), og L skal indeholde exponenten.

Strenge returneres på toppen af stakken, som beskrevet i afsnit 22.3.2.3.

Pointere returneres i HL registerparret.

Kapitel 23

Lagerorganisering

23.1 Memory maps

Under oversættelsen af et program er systemets lager organiseret som vist nedenfor:

0000 - 0OFF	CP/M og run-time arbejdslager
0100 - EOFR	Run-time programdel
EOFR - EOFC	COMPAS monitor, editor og compiler
EOFC - EOFW	Compilerens arbejdslager
EOFW - EOFM	Fejlmeddelelser (COMPAS.ERM)
EOFM - EOFT	Kildetekst
EOFT - >>>	Objektkode
<<< - MTOP	Compilerens symboltabel
<<< - LTOP	CPU stak
LTOP - FFFF	CP/M operativsystem

Hvis COMPAS.ERM filen ikke blev indlæst ved start af COMPAS systemet, starter kildeteksten ved EOFW. Når compileren startes fra en COMPILE eller en RUN kommando, bliver objektkoden gemt direkte ud i lageret umiddelbart efter kildeteksten. CPU stakken arbejder sig nedeften startende fra LTOP (slutadressen på CP/M TPA'en). Symboltabellen arbejder sig nedeften startende fra MTOP, der ligger 1K bytes under LTOP (LTOP-\$400).

Under kørsel af et program, der blev startet med RUN kommandoen, er systemets lager organiseret som vist nedenfor:

0000 - 0OFF	CP/M og run-time arbejdslager
0100 - EOFR	Run-time programdel
EOFR - EOFC	COMPAS monitor, editor og compiler
EOFC - EOFW	Compilerens arbejdslager
EOFW - EOFM	Fejlmeddelelser (COMPAS.ERM)
EOFM - EOFT	Kildetekst
EOFT - EOFP	Objektkode
EOFP - BOFH	Ubrugt
BOFH - >>>	Heap (styres via hptr)
<<< - BOFR	Procedurestak (styres via rptr)
<<< - BOFS	CPU stak (styres via sptr)
BOFS - FTOP	Ubrugt
FTOP - LTOP	Programmets variable
LTOP - FFFF	CP/M operativsystem

EOFP er slutadressen på objektkoden, og hptr (heap-pointeren) sættes til denne adresse ved start af programmet (BOFH=EOFP). Området mellem FTOP og LTOP bruges til programmets variable. FTOP er slutadressen på det frie lager, og sptr (CPU stakpointeren) sættes til denne adresse ved start af programmet (BOFS=FTOP). Procedurestakken bruges kun af rekursive procedurer og funktioner (til at gemme kopier af deres arbejdslager). Ved begyndelsen af et program sættes rptr (procedurestakpointeren) til at pege 1K bytes under CPU stakpointeren (BOFR=BOFS-\$400).

Under kørsel af en programfil er lageret organiseret som vist nedenfor:

0000 - 00FF	CP/M og run-time arbejdsLAGER
0100 - EOFR	Run-time programdel
EOFR - SOFP	Ubrugt (brugerdefinerede maskinkoderutiner)
SOFP - EOFP	Objektkode
EOFP - BOFH	Ubrugt
BOFH - >>>	Heap (styres via hptr)
<<< - BOFR	Procedurestak (styres via rptr)
<<< - BOFS	CPU stak (styres via sptr)
BOFS - FTOP	Ubrugt
FTOP - PTOP	Programmets variable
PTOP - LTOP	Ubrugt
LTOP - FFFF	CP/M operativsystem

SOFP er startadressen på objektkoden, svarende til `<start>` parametren i PROGRAM og OBJECT kommandoerne. PTOP er slutadressen på programmets arbejdsLAGER, svarende til `<slut>` parametren i PROGRAM og OBJECT kommandoerne.

23.2 Heopen og stakkene

Som det fremgår af de ovenfor viste memory maps, eksisterer der tre stak-lignende strukturer i lageret under udførelsen af et program: Heopen, CPU stakken og procedurestakken.

Heopen bruges til at lagre dynamiske variable, og den styres via new, mark og release procedurerne. Ved begyndelsen af et program sættes heap-pointeren (hptr) til startadressen på det frie lager.

CPU stakken bruges til at gemme mellemresultater under udregning af udtryk, og til at overføre parametre til underprogrammer. Desuden bruger en aktiv FOR sætning to bytes på CPU stakken. Ved begyndelsen af et program sættes CPU stakpointeren (sptr) til slutadressen på det frie lager.

Procedurestakken anvendes udelukkende af rekursive procedurer og funktioner (dvs. procedurer og funktioner, der er oversat i `(*$A+*)` stillingen). Ved indgang til en rekursiv procedure eller funktion bliver hele underprogrammets arbejdsLAGER kopieret ned på procedurestakken, og umiddelbart før underprogrammet returnerer, kopieres det tilbage igen. Ved begyndelsen af et program sættes procedurestakpointeren (rptr) til at pege 1K bytes (\$400) under CPU stakpointeren.

Ved hjælp af tre prædefinerede variable er det muligt for programmøren at flytte på heopen og stakkene:

hptr	Heap-pointeren.
rptr	Procedurestakpointeren.
sptr	CPU stakpointeren.

Typen af disse variable er integer. Bemærk, at variablene kun må anvendes i udtryk og i tilskrivninger; de kan altså ikke bruges som var-parametere til underprogrammer.

Når der rettes på heap- og/eller stakpointerne er det programmerens opgave at sikre, at den følgende regel overholdes:

```
hptr < rptr < sptr
```

Hvis denne relation ikke er sand, er følgerne uforudsigelige (og til tider ganske katastrofale for programmet). Det er naturligvis ikke tilladt at ændre på heap- og stakpointervariablene, når den givne struktur først er taget i anvendelse. Derfor bør tilskrivninger til disse variable altid foretages i begyndelsen af et program.

Hvergang new proceduren kaldes, og hvergang der udføres et kald til et rekursivt underprogram, tester systemet, at der ikke er sket en kollision mellem heapen og procedurestakken, altså at hptr er mindre end rptr. Hvis dette ikke er tilfældet, opstår en kørselsfejl.

Bemærk, at der på intet tidspunkt testes for, om CPU stakken har fået overløb ind i "bunden" af procedurestakken. Denne situation opstår først, når rekursive procedurekald nestes en 300-500 gange, hvilket sker meget sjældent. Skulle det imidlertid være påkrævet at neste så "dybt", må programmet flytte procedurestakpointeren ned i lageret for at skabe mere plads. Dette gøres med sætningen:

```
rptr:=sptr-2*maxdybde-512;
```

hvor maxdybde er det højeste nestingniveau, der kan forekomme. De 512 bytes (eller deromkring), der reserveres derudeover, sikrer, at der er plads til mellemresultater under udregning af udtryk.

Kapitel 24

Interruptstyring

Den kode, der genereres af COMPAS Pascal, såvel som run-time programdelen, er fuldt ud interruptbar. Bemærk dog, at interruptrutiner skal gemme samtlige registre, der anvendes (dette gælder også indexregistrene og de alternative registre).

Interruptrutiner kan eventuelt skrives som procedurer i COMPAS Pascal. Sådanne procedurer skal altid oversættes som absolutte procedurer (i `(*$A+*)` stillingen), de må ikke tage parametre, og de skal selv sørge for, at alle registre, der anvendes, bliver gemt. Det sidstnævnte opnås ved, at placere en CODE sætning, der indeholder de nødvendige PUSH instruktioner, i begyndelsen af proceduren, og en anden CODE sætning, der indeholder de tilsvarende POP instruktioner, i slutningen af proceduren. Desuden bør den sidste instruktion i den afsluttende CODE sætning være en EI instruktion (`$FB`), der aktiverer yderligere interrupts. Hvis hardwaren anvender "daisy-chainede" interrupts, kan den afsluttende CODE sætning envidere angive en RETI instruktion (`$ED,$4D`), der så udføres i stedet for den RET instruktion, der genereres af compileren.

De generelle regler for hvilke registre, der anvendes af den kode, der genereres af COMPAS Pascal, er, at operationer på skalarer (undtagen reals) kun benytter AF, BC, DE og HL, at andre operationer tillige benytter IX og IY, og at operationer på reals tillige benytter de alternative registre.

En interruptprocedure må ikke udføre I/O operationer ved hjælp af COMPAS Pascals standardprocedurer og funktioner, da disse rutiner ikke er re-entrante. Bemærk desuden, at kald til BDOS'en (og eventuelt også BIOS'en, afhængigt af den aktuelle implementation af CP/M) bør undgås i interruptprocedurer, da disse rutiner ej heller er re-entrante.

Programmøren kan aktivere og passivere interrupts fra et program ved hjælp af EI og DI instruktioner, der indsættes med CODE sætninger.

Hvis der anvendes mode 0 (IM 0) eller mode 1 (IM 1) interrupts, er det programmørens opgave at initialisere "restart" områderne. Bemærk, at RST 0 interrupts ikke kan anvendes, da RST 0 området fra adresse 0 til 7 bruges af CP/M operativsystemet. Det samme gælder ikke-maskerbare interrupts (NMI'er), da området omkring adresse \$66 er en del af CP/M's arbejdslager. Hvis der anvendes mode 2 interrupts, er det programmørens opgave at erklære en adressetabel (et ARRAY OF integer) på en fast adresse og initialisere den, samt at initialisere I registeret gennem en CODE sætning i begyndelsen af programmet.

Det nedenfor viste program anvender en interruptprocedure. Det antages, at der hvert sekund forekommer en mode 1 interrupt (et kald til adresse \$38):

```

PROGRAM interrupt;
TYPE
  tidstr = STRING(.6.);
VAR
  rsthop: byte AT $38;
  rstadr: integer AT $39;
  sekunder,minutter,timer: integer;

PROCEDURE plustid;
BEGIN
  CODE $F5,$E5,$D5,$C5;
  sekunder:=succ(sekunder);
  IF sekunder=60 THEN
  BEGIN
    sekunder:=0; minutter:=succ(minutter);
    IF minutter=60 THEN
    BEGIN
      minutter:=0; timer:=succ(timer);
      IF timer=24 THEN timer:=0;
    END;
  END;
  CODE $C1,$D1,$E1,$F1,$FB;
END;

FUNCTION tid: tidstr;
VAR
  t: tidstr;
BEGIN
  t(.0.):=@8; t(.3.):=':'; t(.6.):=':';
  t(.1.):=chr(timer DIV 10+48);
  t(.2.):=chr(timer MOD 10+48);
  t(.4.):=chr(minutter DIV 10+48);
  t(.5.):=chr(minutter MOD 10+48);
  t(.7.):=chr(sekunder DIV 10+48);
  t(.8.):=chr(sekunder MOD 10+48);
  tid:=t;
END;

BEGIN
  rsthop:=$C3; rstadr:=addr(plustid);
  write('skriv timer, minutter og sekunder: ');
  readln(timer,minutter,sekunder);
  CODE $ED,$56,$FB;
  :
  writeln('klokken er ',tid,'.');
  :
  CODE $F3;
END.

```

Da plustid interruptproceduren kun udfører heltalsoperationer, behøver den kun gemme AF, BC, DE og HL registerparrene. CODE sætningen i slutningen af plustid angiver både de nødvendige POP instruktioner og en EI instruktion, der akiverer yderligere interrupts. I begyndelsen af programmet klargøres en hopvektor til plustid rutinen i adresse \$38. Derefter indlæses det nuværende klokkeslet, interrupt mode 1 vælges med en IM 1 instruktion (\$ED,\$56), og til sidst aktiveres interrupts med en EI instruktion (\$FB). En DI instruktion (\$F3) bruges til at passivere interrupts umiddelbart før programmet slutter.

Kapitel 25**Forskelle mellem COMPAS og Standard Pascal**

COMPAS Pascal følger nøje Jensen & Wirth's definition af Standard Pascal, der gives i bogen "Pascal User Manual and Report". Der er imidlertid enkelte mindre forskelle, og disse beskrives nedenfor. Bemærk, at dette kapitel ikke omhandler de udvidelser, der er tillagt COMPAS Pascal.

Dynamiske variable

Dynamiske variable oprettes og fjernes med new, mark og release procedurerne i stedet for med de af standarden foreslæede new og dispose procedurer. Dette skyldes til dels ønsket om at sikre kompatibilitet med andre Pascal compilere (f.eks. UCSD Pascal), men primært er grunden, at de nye procedurer er mere effektive, hvad angår kørselshastighed og kodestørrelse. Desuden tillader COMPAS Pascals new procedure ikke angivelser af varianter (dette kan relativt nemt omgås ved at ændre hptr variablen direkte).

Get og put procedurerne

get og put standardprocedurerne findes ikke i COMPAS Pascal. I stedet er read og write procedurerne udvidet, så de nu kan anvendes til alle typer af input og output. Der er tre grunde til dette: For det første er read og write lettere at bruge og nemmere at forstå, for det andet er de hurtigere, og for det tredje kan størrelsen af filvariable reduceres, når der ikke er behov for buffer-variable.

GOTO sætninger

En GOTO sætning må ikke hoppe til en label, der ligger udenfor den nuværende blok (som regel er dette også dårlig programmeringsteknik).

Page proceduren

page standardproceduren findes ikke i COMPAS Pascal, da der i CP/M operativsystemet ikke findes en standard for, hvilket kontroltegn, der skal anvendes til sideskift.

Procedurer og funktioner som parametre

COMPAS Pascal tillader ikke, at procedurer og funktioner anvendes som parametre. Der er to grunde hertil: For det første er definitionen af Standard Pascal meget uklar på dette punkt (så vidt vi ved, findes der ingen implementation af Pascal under CP/M, der følger standarden på dette område) og for det andet bruges denne facilitet næsten aldrig.

Pakkede variable

Det reserverede ord PACKED har ingen betydning i COMPAS Pascal (men tillades dog). Variable pakkes i stedet automatisk, når det er muligt, og som følge heraf, er pack og unpack procedurerne ikke implementerede.

Appendix A

Oversigt over standard procedurer og funktioner

Dette appendix giver en oversigt over COMPAS Pascals standard procedurer og funktioner. De følgende notationer anvendes:

<type>	Angiver enhver type.
<string>	Angiver enhver strengtype.
<file>	Angiver enhver filtype.
<scalar>	Angiver enhver skalar type (undtagen real).
<pointer>	Angiver enhver pointertype.

Bemærk, at visse procedurer og funktioner tillader var-parametre af enhver type. I så tilfælde angives der ingen type for disse parametre.

Input/Output rutiner

De nedenfor viste procedurer benytter en non-standard syntaks i deres parameterlister.

```

PROC      read (VAR f: FILE OF <type>; VAR v: <type>);
          read (VAR f: text; VAR i: integer);
          read (VAR f: text; VAR r: real);
          read (VAR f: text; VAR c: char);
          read (VAR f: text; VAR s: <string>);
PROC      readln (VAR f: text);
PROC      write (VAR f: FILE OF <type>; VAR v: <type>);
          write (VAR f: text; i: integer);
          write (VAR f: text; r: real);
          write (VAR f: text; b: boolean);
          write (VAR f: text; c: char);
          write (VAR f: text; s: <string>);
PROC      writeln (VAR f: text);

```

Filbehandlingsrutiner

```

PROC      assign (VAR f: <file>; name: <string>);
PROC      reset (VAR f: <file>);
PROC      rewrite (VAR f: <file>);
PROC      close (VAR f: <file>);
PROC      erase (VAR f: <file>);
PROC      rename (VAR f: <file>; name: <string>);
FUNC      eof (VAR f: <file>): boolean;
FUNC      eoln (VAR f: text): boolean;
PROC      seek (VAR f: FILE OF <type>; pos: integer);
          seek (VAR f: FILE; pos: integer);
FUNC      length (VAR f: FILE OF <type>): integer;
          length (VAR f: FILE): integer;
FUNC      position (VAR f: FILE OF <type>): integer;
          position (VAR f: FILE): integer;
PROC      blockread (VAR f: FILE; VAR dest; numrec: integer);
PROC      blockwrite (VAR f: FILE; VAR dest; numrec: integer);
PROC      execute (VAR f: <file>);
PROC      chain (VAR f: <file>);

```

Aritmetiske rutiner

```

FUNC      abs (i: integer): integer;
FUNC      abs (r: real): real;
FUNC      sqr (i: integer): integer;
FUNC      sqr (r: real): real;
FUNC      sqrt (r: real): real;
FUNC      sin (r: real): real;
FUNC      cos (r: real): real;
FUNC      arctan (r: real): real;
FUNC      ln (r: real): real;
FUNC      exp (r: real): real;
FUNC      int (r: real): real;
FUNC      frac (r: real): real;

```

Skalare rutiner

```

FUNC      succ (x: <scalar>): <scalar>;
FUNC      pred (x: <scalar>): <scalar>;
FUNC      odd (i: integer): boolean;

```

Konverteringsrutiner

```

FUNC      trunc (r: real): integer;
FUNC      round (r: real): integer;
FUNC      ord (x: <scalar>): integer;
FUNC      chr (i: integer): char;

```

Strengbehandlingsrutiner

Bemærk, at str proceduren benytter en non-standard syntaks til den numeriske parameter.

```

FUNC      len (s: <string>): integer;
FUNC      pos (pattern,source: <string>): integer;
FUNC      copy (s: <string>; pos,len: integer): <string>;
FUNC      concat (s1,s2,...,sn: <string>): <string>;
PROC     delete (VAR s: <string>; pos,len: integer);
PROC     insert (s: <string>; VAR d: <string>; pos: integer);
PROC     str (i: integer; VAR s: <string>);
PROC     str (r: real; VAR s: <string>);
PROC     val (s: <string>; VAR i,p: integer);
PROC     val (s: <string>; VAR r: real; VAR p: integer);

```

Heap- og pointerrutiner

```

PROC      new (VAR p: <pointer>);
PROC      mark (VAR p: <pointer>);
PROC      release (VAR p: <pointer>);
FUNC      memavail : integer;
FUNC      ord (p: <pointer>): integer;
FUNC      ptr (i: integer): <pointer>;

```

Diverse rutiner

```
FUNC      _c (i: integer): integer;
FUNC      hi (i: integer): integer;
FUNC      swap (i: integer): integer;
PROC  randomize ;
FUNC   random (range: integer): integer;
        random : real;
FUNC   pwrtan (exp: integer): real;
FUNC   keypress : boolean;
PROC   gotoxy (x,y: integer);
PROC   clreos ;
PROC   clreol ;
FUNC   iores : boolean;
PROC   move (VAR source,dest; length: integer);
PROC   fill (VAR dest; length: integer; data: byte);
        fill (VAR dest; length: integer; data: char);
FUNC   addr (VAR variable): integer;
        addr (<procedure identifier>): integer;
        addr (<function identifier>): integer;
FUNC   size (VAR variable): integer;
        size (<type identifier>): integer;
PROC   bdos (func,param: integer);
PROC   bios (func,param: integer);
FUNC   bdos (func,param: integer): integer;
FUNC   bios (func,param: integer): integer;
FUNC   bdosb (func,param: integer): byte;
FUNC   biosb (func,param: integer): byte;
```

Appendix B

Oversigt over operatorer

Dette appendix giver en oversigt over alle de operatorer, der findes i COMPAS Pascal. Operatorerne er grupperet efter prioritet, og grupperne er opskrevet efter faldende prioritet.

<u>Operator</u>	<u>Operation</u>	<u>Operandtype(r)</u>	<u>Resultattype(r)</u>
+ alene	ingen	integer, real	som operand
- alene	negation	integer, real	som operand
NOT	komplementering	integer, boolean	som operand
*	multiplikation	integer, real	integer, real
	foreningsmængde	mængde	som operand
/	division	integer, real	real
DIV	heltalsdivision	integer	integer
MOD	modulus	integer	integer
AND	logisk AND	integer, boolean	som operand
SHL	venstreskift	integer	integer
SHR	højreskift	integer	integer
+	addition	integer, real	integer, real
	sammensætning	streng	som operand
	fællesmængde	mængde	som operand
-	subtraktion	integer, real	integer, real
	differensmængde	mængde	som operand
OR	logisk OR	integer, boolean	som operand
EXOR	logisk EXOR	integer, boolean	som operand
=	lighed	skalar	boolean
	lighed	streng	boolean
	lighed	mængde	boolean
	lighed	pointer	boolean
<>	ulighed	skalar	boolean
	ulighed	streng	boolean
	ulighed	mængde	boolean
	lighed	pointer	boolean
>=	større eller lig	skalar	boolean
	større eller lig	streng	boolean
	delmængde-test	mængde	boolean
<=	mindre eller lig	skalar	boolean
	mindre eller lig	streng	boolean
	delmængde-test	mængde	boolean
>	større end	skalar	boolean
	større end	streng	boolean
<	mindre end	skalar	boolean
	mindre end	streng	boolean
IN	element-test	se nedenfor	boolean

IN operatorens første operand kan være af enhver skalar type (undtagen real) og den anden operand skal være en mængde af samme type.

Appendix C

Oversigt over compilerdirektiver

Et compilerdirektiv skrives i en kommentar, og kan derfor anvendes nársomhelst en kommentar tillades. Et \$ tegn, som det første tegn i en kommentar, angiver, at kommentaren indeholder et eller flere compilerdirektiver. Det generelle format af en kommentar, der indeholder compilerdirektiver, er således:

(*\$<compilerdirektiver> <kommentar>*)

Compilerdirektiverne angives som en liste af instruktioner, adskilt af kommaer, og hver instruktion begynder med et bogstav. Hvis bogstavet udvælger et compilerflag, skal det efterfølges af et plus (+) eller et minus (-), alt efter om den funktion, der styres af flaget, skal aktiveres eller passiveres. Hvis bogstavet udvælger et compilerregister (se W nedenfor), skal det efterfølges af et talciffer, der angiver registerets nye værdi, og hvis bogstavet udvælger en speciel facilitet (se I nedenfor), skal det efterfølges af et eller flere tegn, der fungerer som parameter til denne facilitet. Nogle eksempler på compilerdirektiver:

(*\$R-*) (*\$S+,I+,A-*) (*W6,B++) (*\$I MAX.LIB*)

De følgende compilerdirektiver er tilgængelige:

R Når dette compilerflag er aktiveret, genererer compileren kode, der checker alle indexeringer af arraystrukturer og alle tilskrivninger til variable af skalare typer og delintervaltyper, for at sikre, at værdierne ligger indenfor de tilladte grænser. Yderligere detaljer gives i afsnit 7.4 og afsnit 9.1

Værdi ved start af compiler: R+

I Et compilerflag, der, når det er aktiveret, instruerer compileren om at generere kode, der checker alle kald til I/O rutiner for at sikre, at der ikke er sket I/O fejl. Yderligere detaljer gives i afsnit 13.5. Hvis I direktivet efterfølges af et filnavn, bliver denne fil medtaget i oversættelsen af programmet. Yderligere detaljer herom findes i kapitel 17. Bemærk, at et I direktiv med et filnavn ikke kan efterfølges af flere direktiver.

Værdi ved start af compiler: I+

A Når dette compilerflag er aktiveret, instruerer det compileren om, at generere absolut kode til procedurer og funktioner, dvs. kode, der ikke tillader rekursion, men som kører hurtigere og fylder mindre end den modsvarerende rekursive kode. Yderligere detaljer findes i afsnit 15.6.

Værdi ved start af compiler: A-

- S Et compilerflag, der, når det er aktiveret, instruerer compileren om at optimere array-indexeringer med hensyn til kørselshastighed i stedet for kodestørrelse. Yderligere detaljer findes i afsnit 9.1

Værdi ved start af compiler: S-

- B Når dette compilerflag er aktiveres ved begyndelsen af en programblok, vil alle kald til I/O procedurer og funktioner, der ikke angiver en filvariabel, benytte CON: enheden. I modsat fald benyttes TRM: enheden. Yderligere detaljer herom findes i afsnit 13.3.3 og afsnit 16.1.

Værdi ved start af compiler: B+

- V Et compilerflag, der, når det er passiveret, instruerer compileren om at tillade strengvariable af enhver type som aktuelle var-parametre, selvom deres maksimale længde ikke stemmer overens med den formelle var-parameter. Yderligere detaljer findes i afsnit 15.6.

Værdi ved start af compiler: V+

- W Denne instruktion skal efterfølges af et talciffer (0..9), der angiver det maksimale nestingniveau for WITH sætninger. W direktivet skal anvendes før erklæringsdelen for den blok, det skal gælde for. Yderligere detaljer findes i afsnit 10.2.

Værdi ved start af compiler: W4

Appendix D

ASCII tegntabel

DEC	HX	TEGN	DEC	HX	TEGN	DEC	HX	TEGN	DEC	HX	TEGN
0	00	NUL	32	20	SPACE	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	£	67	43	C	99	63	c
4	04	EOT	36	24	\$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(72	48	H	104	68	h
9	09	HT	41	29)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	Æ	123	7B	æ
28	1C	FS	60	3C	<	92	5C	Ø	124	7C	ø
29	1D	GS	61	3D	=	93	5D	A	125	7D	å
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Appendix E
COMPAS syntax

COMPAS Pascal sprogets syntax vises i dette appendix på BNF form (Backus-Naur form). De nedenfor viste symboler indgår i BNF formen, og er ikke en del af Pascal sproget:

- ::= Betyder 'er defineret som'.
- | Betyder 'eller'.
- {...} Angiver, at symbolerne mellem parenteserne kan gentages nul eller flere gange.

Symbollet <character> angiver ethvert skrivbart ASCII tegn, dvs. et tegn med en ASCII værdi mellem \$20 og \$7F.

```

<empty> ::=
<letter> ::= 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 | -
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<hexdigit> ::= <digit> | A | B | C | D | E | F
<program> ::= <program heading> <block> .
<program heading> ::= <empty> | PROGRAM <program identifier>
                      <file identifier list>
<program identifier> ::= <identifier>
<identifier> ::= letter { <letter or digit> }
<letter or digit> ::= <letter> | <digit>
<file identifier list> ::= <empty> | ( <file identifier>
                      { , <file identifier> }
<file identifier> ::= <identifier>
<block> ::= <declaration part> <statement part>
<declaration part> ::= { <declaration section> }
<declaration section> ::= <label declaration part> |
                           <constant definition part> | <type definition part> |
                           <variable declaration part> |
                           <procedure and function declaration part>
<label declaration part> ::= LABEL <label> { , <label> } ;

```

```

label ::= <letter or digit> <letter or digit>

· constant definition part ::= CONST <constant definition>
  { ; <constant definition> } ;

<constant definition> ::= <untyped constant definition> |
  <typed constant definition>

<untyped constant definition> ::= <identifier> = <constant>

<constant> ::= <unsigned number> | <sign> <unsigned number> |
  <constant identifier> | <sign> <constant identifier> |
  <string>

<unsigned number> ::= <unsigned integer> | <unsigned real>

<unsigned integer> ::= <digit sequence> | $ <hexdigit sequence>

<digit sequence> ::= <digit> { <digit> }

<hexdigit sequence> ::= <hexdigit> { <hexdigit> }

<unsigned real> ::= <digit sequence> . <digit sequence> |
  <digit sequence> . <digit sequence> E <scale factor> |
  <digit sequence> E <scale factor>

<scale factor> ::= <digit sequence> | <sign> <digit sequence>

<sign> ::= + | -

<constant identifier> ::= <identifier>

<string> ::= { <string element> }

<string element> ::= <text string> | <control character>

<text string> ::= ' { <character> } '

<control character> ::= @ <unsigned integer> | ^ <character>

<structured constant definition> ::= <identifier> : <type> =
  <structured constant>

<type> ::= <simple type> | <structured type> | <pointer type>

<simple type> ::= <scalar type> | < subrange type> |
  <type identifier>

<scalar type> ::= ( <identifier> { , <identifier> } )

<subrange type> ::= <constant> .. <constant>

<type identifier> ::= <identifier>

<structured type> ::= <unpacked structured type> |
  PACKED <unpacked structured type>

<unpacked structured type> ::= <string type> | <array type> |
  <record type> | <set type> | <file type>

```

```

<string type> ::= STRING | <constant> )

<array type> ::= ARRAY [ <index type> ; , <index type> ] ) OF
    <component type>

<index type> ::= <simple type>

<component type> ::= <type>

<record type> ::= RECORD <field list> END

<field list> ::= <fixed part> | <fixed part> ; <variant part> |
    <variant part>

<fixed part> ::= <record section> { ; <record section> }

<record section> ::= <empty> | <field identifier>
    { , <field identifier> } : <type>

<field identifier> ::= <identifier>

<variant part> ::= CASE <tag field> <type identifier> OF
    <variant> { ; <variant> }

<tag field> ::= <empty> | <field identifier> :

<variant> ::= <empty> | <case label list> : ( <field list> )

<case label list> ::= <case label> { , <case label> }

<case label> ::= <constant>

<set type> ::= SET OF <base type>

<base type> ::= <simple type>

<file type> ::= FILE OF <type>

<pointer type> ::= ^ <type identifier>

<structured constant> ::= <constant> | <array constant> |
    <record constant> | <set constant>

<array constant> ::= ( <structured constant>
    { , <structured constant> } )

<record constant> ::= ( <record constant element>
    { ; <record constant element> } )

<record constant element> ::= <field identifier> :
    <structured constant>

<set constant> ::= [ { <set constant element> } ]

<set constant element> ::= <constant> | <constant> .. <constant>

<type definition part> ::= TYPE <type definition>
    { ; <type definition> } ;

<type definition> ::= <identifier> = <type>

```

```

variable declaration part ::= VAR <variable declaration>
                           ; <variable declaration> ;

<variable declaration> ::= <identifier list> : <type> |
                           <identifier> : <type> AT <address specification>

<identifier list> ::= <identifier> { , <identifier> }

<address specification> ::= <constant> | <variable identifier>

<variable identifier> ::= <variable>

<procedure and function declaration part> ::= 
                           { <procedure or function declaration> }

<procedure or function declaration> ::= <procedure declaration> |
                           <function declaration>

<procedure declaration> ::= <procedure heading> <block> ;

<procedure heading> ::= PROCEDURE <identifier> ; |
                           PROCEDURE <identifier> ( <formal parameter section>
                           { , <formal parameter section> } ) ;

<formal parameter section> ::= <parameter group> |
                           VAR <parameter group>

<parameter group> ::= <identifier list> : <type identifier>

<function declaration> ::= <function heading> <block> ;

<function heading> ::= FUNCTION <identifier> : <result type> ; |
                           FUNCTION <identifier> ( <formal parameter section>
                           { , <formal parameter section> } ) : <result type> ;

<result type> ::= <type identifier>

<statement part> ::= <compound statement>

<compound statement> ::= BEGIN <statement> { ; <statement> } END

<statement> ::= <simple statement> | <structured statement>

<simple statement> ::= <assignment statement> |
                           <procedure statement> | <goto statement> |
                           <code statement> | <empty statement>

<assignment statement> ::= <variable> := <expression> |
                           <function identifier> := <expression>

<variable> ::= <entire variable> | <component variable> |
                           <referenced variable>

<entire variable> ::= <variable identifier> |
                           <typed constant identifier>

<typed constant identifier> ::= <identifier>

<component variable> ::= <indexed variable> | <field designator>

```

```

<indexed variable> ::= <array variable> | <expression>
    , <expression> ) )

<array variable> ::= <variable>

<field designator> ::= <record variable> . <field identifier>

<record variable> ::= <variable>

<field identifier> ::= <identifier>

<referenced variable> ::= <pointer variable> ^

<pointer variable> ::= <variable>

<expression> ::= <simple expression> { <relational operator>
    <simple expression> }

<simple expression> ::= <term> { <adding operator> <term> }

<term> ::= <complemented factor> { <multiplying operator>
    <complemented factor> }

<complemented factor> ::= <signed factor> | NOT <signed factor>

<signed factor> ::= <factor> | <sign> <factor>

<factor> ::= <variable> | <unsigned constant> |
    ( <expression> ) | <function designator> | <set>

<unsigned constant> ::= <unsigned number> | <string> |
    <constant identifier> | NIL

<function designator> ::= <function identifier> |
    <function identifier> ( <actual parameter>
    { , <actual parameter> } )

<function identifier> ::= <identifier>

<actual parameter> ::= <expression> | <variable>

<set> ::= [ { <set element> } ]

<set element> ::= <expression> | <expression> .. <expression>

<multiplying operator> ::= * | / | DIV | MOD | AND | SHL | SHR

<adding operator> ::= + | - | OR | EXOR

<relational operator> ::= = | <> | >= | <= | > | < | IN

<procedure statement> ::= <procedure identifier> |
    <procedure identifier> ( <actual parameter>
    { , <actual parameter> } )

<goto statement> ::= GOTO <label>

<code statement> ::= CODE <code list element>
    { , <code list element> }

```

```

<code list element> ::= <unsigned integer> |
    <constant identifier> | <variable identifier> |
    <location counter reference>

<location counter reference> ::= * | * <sign> <constant>

<empty statement> ::= <empty>

<structured statement> ::= <compound statement> |
    <conditional statement> | <repetitive statement> | |
    <with statement>

<conditional statement> ::= <if statement> | <case statement>

<if statement> ::= IF <expression> THEN <statement> |
    IF <expression> THEN <statement> ELSE <statement>

<case statement> ::= CASE <expression> OF <case element>
    { ; <case element> } END | CASE <expression> OF
    <case element> { ; <case element> } OTHERWISE <statement>
    { ; <statement> } END

<case element> ::= <case list> : <statement>

<case list> ::= <case list element> { , <case list element> }

<case list element> ::= <constant> | <constant> .. <constant>

<repetitive statement> ::= <while statement> |
    <repeat statement> | <for statement>

<while statement> ::= WHILE <expression> DO <statement>

<repeat statement> ::= REPEAT <statement> { ; <statement> }
    UNTIL <expression>

<for statement> ::= FOR <control variable> := <for list> DO
    <statement>

<for list> ::= <initial value> TO <final value> |
    <initial value> DOWNTO <final value>

<control variable> ::= <variable identifier>

<initial value> ::= <expression>

<final value> ::= <expression>

<with statement> ::= WITH <record variable list> DO <statement>

<record variable list> ::= <record variable>
    { , <record variable> }

```

Appendix F

I/O fejl

En I/O fejl rapporteres når der opstår en fejl under en filoperation eller en ind- eller udlæsningsoperation. Hvis sætningen, hvori fejlen opstår, er oversat i (*\$I+*) stillingen, stopper programmet, og en fejlmeddeelse vises:

```
I/O ERROR nn AT PC=aaaa
Program terminated
```

hvor nn er fejlens nummer (i hexnotation), og aaaa er fejlens relative adresse (i forhold til startadressen på programkoden). I (*\$I-*) stillingen stoppes programmet ikke, men i stedet gemmes fejlens nummer i en systemvariabel, der kan undersøges ved at kalde iores funktionen.

De følgende I/O fejl kan forekomme:

- 01 Uoverensstemmende elementlængder. Denne fejl rapporteres af reset, hvis programmet prøver på at sammenknytte en filvariabel af en selvdefineret filtype (FILE OF type) med en diskfil, der ikke har den samme elementlængde (og dermed heller ikke den samme elementtype).
- 02 Filen findes ikke. Denne fejl rapporteres af reset, erase, rename, execute eller chain, hvis der ikke findes en fil af det navn, der er tilknyttet filvariablen.
- 03 Biblioteket er fuldt. Denne fejl rapporteres af rewrite, hvis der ikke er plads til nye filer i diskettens bibliotek.
- 04 Fil forsvundet. Denne fejl rapporteres af close, hvis den diskfil, der referes til, er forsvundet. Fejlen kan eventuelt skyldes, at erase er blevet kaldt, eller, at brugeren har isat en anden diskette.
- 05 Indlæsning ej tilladt. Denne fejl rapporteres af read (fra en textfil eller en selvdefineret fil) eller readln, hvis programmet prøver at læse fra en fil, der ikke er åben, eller, hvis programmet prøver at læse fra en textfil, der er åbnet med rewrite eller sammenknyttet med LST: enheden.
- 06 Udlæsning ej tilladt. Denne fejl rapporteres af write (til en textfil eller en selvdefineret fil) eller writeln, hvis programmet prøver at skrive til en fil, der ikke er åben, eller, hvis programmet prøver at skrive til en textfil, der er åbnet med reset eller sammenknyttet med KBD: enheden.
- 07 Fil pludseligt afsluttet. Denne fejl rapporteres af read eller readln (fra en textfil), hvis filen er slut, selvom der ikke er læst et end-of-file tegn (CTRL/Z).
- 08 Disketten er fuld. Denne fejl rapporteres af write eller writeln (til en textfil), hvis der ikke er mere plads på disketten.

- 09 Fejl i numerisk værdi. Denne fejl rapporteres af read eller readln, hvis en indlæst numerisk værdi ikke er af et korrekt format.
- 0A Filen er slut. Denne fejl rapporteres af read (fra en selvdefineret fil) eller blockread, hvis programmet prøver at læse fra en fil, når fil-pointeren er ved slutningen af filen.
- 0B Fil-overløb. Denne fejl rapporteres af write (til en selvdefineret fil), hvis programmet prøver at lagre mere end 65535 elementer i en fil.
- 0C Læsefejl. Denne fejl rapporteres af read eller write (til eller fra en selvdefineret fil) eller blockread, hvis rutinen er ude af stand til at læse den næste sektor fra diskfilen. Dette er typisk en indikation af, at der er et eller andet galt med filens interne format. For blockread kan fejlen desuden betyde, at programmet prøver at læse fra en fil, der ikke er længere.
- 0D Skrivefejl. Denne fejl rapporteres af read eller write (til en selvdefineret fil), flush eller blockwrite, hvis filen ikke kan udvides yderligere, fordi disketten er fuld.
- 0E Ulovligt elementnummer. Denne fejl rapporteres af seek, hvis programmet prøver på, at flytte fil-pointeren til et element, der ligger ud over filens længde.
- 0F Fil ej åben. Denne fejl rapporteres af blockread eller blockwrite hvis filen ikke er åben.
- 10 Logisk I/O enhed tillades ikke her. Denne fejl rapporteres af erase, rename, execute eller chain (på en textfil), hvis filen er tilskrevet en logisk I/O enhed.
- 11 Ulovlig brug af programkædning. Denne fejl rapporteres af execute eller chain, hvis et program, der blev startet med RUN kommandoen, prøver på, at starte et andet program. execute og chain må kun anvendes fra programfiler.

Appendix G

Kørselsfejl

En kørselsfejl rapporteres, når der opstår en uoprettelig fejltilstand i systemet. Kørselsfejl betyder altid, at programmet stopper og udskriver en fejlmeddeelse:

```
EXECUTION ERROR nn AT PC=aaaa
Program terminated
```

hvor nn er fejlens nummer, og aaaa er fejlens relative adresse (i forhold til programmets startadresse).

De følgende kørselsfejl kan forekomme:

- 01 Streng-overløb. Denne fejl rapporteres ved sammensætning af af to strenge (med plus operatoren eller concat proceduren), hvis den resulterende streng er længere end 255 tegn, eller ved konvertering af en streng til et tegn, hvis strengens længde er forskellig fra 1.
- 02 Ulovlig strengposition. Denne fejl rapporteres af copy, delete eller insert, hvis strengpositionen ikke er en værdi mellem 1 og 255.
- 03 Overløb under floating point operation.
- 04 Division med nul forsøgt. Denne fejl rapporteres, hvis et reelt tal forsøges divideret med 0. Bemærk, at det tilsvarende ikke gælder for heltal (integers).
- 05 Ulovligt argument til sqrt. Programmet forsøger at udregne kvadratroden til et negativt tal.
- 06 Ulovligt argument til ln. Programmet forsøger at udregne den naturlige logaritme til et tal, der er mindre end eller lig med 0.
- 07 Ulovligt argument til trunc eller round. Det reelle tal, der forsøges konverteret til et heltal, er ikke indenfor hel-talsområdet (-32768..32767).
- 08 Over/underløb på index. Værdien af et indexudtryk er udenfor de tilladte grænser.
- 09 Over/underløb på skalar eller delinterval. Værdien, der forsøges tilskrevet til en variabel af en skalar type eller en delintervaltype, er udenfor de tilladte grænser.
- 0A Kollision mellem heap og stak. Denne fejl rapporteres ved et kald til new, eller ved et kald til et rekursivt underprogram, hvis der ikke er tilstrækkelig "luft" (frit lager) mellem heap-pointeren og procedurestakpointeren.

Appendix E

Compilerfejl

- 01 '.' forventet.
- 02 BEGIN forventet.
- 03 Ulovlig resultattype. Funktioner kan kun returnere værdier af skalare typer, strengtyper og pointertyper.
- 04 Dubleret identifier. Dette navn er allerede anvendt.
- 05 Absolutte variable tillades ikke i poster. AT specifikationen kan ikke anvendes inden i poster.
- 06 Typeidentifier forventet.
- 07 Filer må kun være VAR parametre.
- 08 Fejlagtig eller ukendt type.
- 09 END forventet.
- 10 Mængdes grundtype er for stor. Grundtypen i en mængde skal være en skalar eller et delinterval, der ikke har over 256 mulige værdier, og begge grænser skal være mellem 0 og 255.
- 11 Filelementer må ikke være filer.
- 12 Ulovlig strenglængde. En strengs maksimumslængde skal være et heltal mellem 1 og 255.
- 13 Ulovlig grundtype for delinterval. Alle skalare typer, undtagen reals, er tilladte.
- 14 '..' forventet.
- 15 Uens typer i intervalgrænser. Typen af den nedre grænse stemmer ikke overens med typen af den øvre.
- 16 Øvre grænse større end nedre. Den ordinale værdi af den øvre grænse skal være større end eller lig den ordinale værdi af den nedre.
- 17 Fejlagtig eller ukendt simpel type.
- 18 Simpel type forventet. Alle skalare typer, undtagen reals, er simple typer.
- 19 Ukendt pointertype i typedefinition(er). En af de ovenstående typedefinitioner refererer til en ukendt typeidentifier.
- 20 Udefineret label i sætningsdel. Den ovenstående sætningsdel indeholder en reference til en ukendt label.
- 21 Ulovlig GOTO i sætningsdel. En GOTO sætning i den ovenstående sætningsdel refererer til en label inden i en FOR løkke.

- 21 Label er allerede defineret.
- 22 THEN forventet.
- 24 DO forventet.
- 25 Fejlagtig eller ukendt variabelidentifier.
- 26 Variabeltype er ikke en simpel type. En FOR sætnings kontrolvariabel skal være af en simpel type.
- 27 Uens typer i FOR sætnings udtryk. Typen af et (eller begge) af en FOR sætnings kontroludtryk stemmer ikke overens med kontrolvariablens type.
- 28 TO eller DOWNTO forventet.
- 29 Konstant er ikke af samme type som CASE udtryk.
- 30 END eller OTHERWISE forventet.
- 31 Ukendt label.
- 32 For mange nestede WITH sætninger. Brug compilerens W direktiv til at forøge det maksimale antal nestede WITH sætninger.
- 33 Postvariabel forventet.
- 34 Fejlagtig eller ukendt variabel.
- 35 Ulovlig tilskrivning. Tilskrivning til filer og typeløse parametre tillades ikke.
- 36 Uens typer i tilskrivning eller parameterliste. Typen af variablen og udtrykket i en tilskrivning stemmer ikke overens, eller typen af den aktuelle parameter i et kald til et underprogram stemmer ikke overens med typen af den formelle parameter.
- 37 Udtryk er ikke af typen integer.
- 38 Udtryk er ikke af typen boolean.
- 39 Udtryk er ikke af en simpel type. Alle skalare typer, undtagen real, er simple typer.
- 40 Udtryk er ikke af en strengtype.
- 41 Uens typer i udtryk. Typerne af operanderne i udtrykket er ikke kompatible.
- 42 Operandtype(r) stemmer ikke overens med operator.
- 43 Strukturerede variable tillades ikke her. Arraystrukturer (undtagen tegn-arrays), poster og filer tillades ikke her.
- 44 Uens typer i mængde. Typen af elementerne eller intervallerne i en mængde stemmer ikke overens.

- 45 Syntaksfejl eller ukendt identifier i udtryk.
- 46 Konstanter tillades ikke her.
- 47 Udtrykstype er forskellig fra indextype.
- 48 Fejlagtig eller ukendt feltidentifier.
- 49 Fejlagtig eller ukendt konstant.
- 50 Integer konstant forventet.
- 51 Integer eller real konstant forventet.
- 52 Strengekonstant ikke afsluttet. Strengekonstanter må ikke strække sig over flere linier.
- 53 Fejl i integer konstant. Enten er der en syntaksfejl i konstanten, eller også er den udenfor heltalsområdet (-32768 til 32767). Bemærk, at hele reelle tal skal efterfølges af et decimalpunktum og et nul, f.eks. 14765123.0. Formatet af heltalskonstanter er defineret i afsnit 2.2.
- 54 Fejl i real konstant. Formatet af en real konstant er defineret i afsnit 2.2.
- 55 Ulovligt tegn i identifier. Det første tegn i en identifier skal være et bogstav ('A' til 'Z' og 'a' til 'z') eller en underscore ('_'). De følgende tegn skal være bogstaver, talciffer ('0' til '9') eller underscores.
- 56 'E' eller '.' forventet.
- 57 'A' eller ')' forventet.
- 58 ':' forventet.
- 59 ';' forventet.
- 60 Syntaksfejl eller ukendt identifier i sætning.
- 61 ',' forventet.
- 62 '(' forventet.
- 63 ')' forventet.
- 64 '=' forventet.
- 65 ':=' forventet.
- 66 OF forventet.
- 67 Programmet kan ikke slutte her. Denne fejl rapporteres, hvis compileren ikke regner programmet for afsluttet ved slutningen af programteksten. Muligvis er der flere BEGIN'er, end der er END'er.
- 68 Fil findes ikke. Den angivne "include-fil" findes ikke.

- 69 Buffer overflow. Denne fejl rapporteres, hvis der ikke er nok lager til at oversætte programmet. Bemærk, at fejlen kan forekomme, selvom der tilsyneladende er lager tilovers - dette lager bruges imidlertid til oversætterens symboltabel. Del teksten ind i mindre afsnit, og brug "include-filer".
- 70 Memory overflow. Der er ikke plads til programmet og dets variable i lageret.
- 71 Variable af denne type kan ikke indlæses.
- 72 Variable af denne type kan ikke udlæses.
- 73 Tekstfil forventet.
- 74 Filvariabel forventet.
- 75 Tekstfiler tillades ikke her.
- 76 Systemfiler tillades ikke her.
- 77 Strengekonstant forventet.
- 78 Strenghængede stemmer ikke overens med type.
- 79 Fejlagtig rækkefølge af feltkonstanter.
- 80 Uens typer i struktureret konstant.
- 81 Konstant udenfor tilladte grænser.
- 82 Filer og pointere tillades ikke her.
- 83 Fejlagtig brug af retype-facilitet. Retype-faciliteten tillader kun simple typer, altså alle skalarer undtagen reals.
- 84 Integer eller real udtryk forventet.
- 85 Strengevariabel forventet.
- 86 Textfiler og typeløse filer tillades ikke her.
- 87 Typeløs fil forventet.
- 88 Pointervariabel forventet.
- 89 Integer eller real variabel forventet.
- 90 Integer variabel forventet.
- 91 Reserveret ord.
- 92 Label er ikke indenfor denne blok. GOTO sætninger må ikke hoppe ud af den nuværende blok.
- 93 Procedure eller funktion er ikke defineret korrekt. Et underprogram i den ovenstående erklæringsdel er FORWARD specificeret, men programblokken er ikke defineret senere.
- 94 Fejl i CODE sætning. CODE sætninger beskrives i kapitel 19.

95 Clovløg brug af AT specifikation. Der kan kun erklæres en variabel ad gangen med AT specifikationen.