C O M P A S

Pascal Program Development System

Version 3.0

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

Poly-Data microcenter ApS
Aaboulevarden 13
DK-1960 Copenhagen V

COPYRIGHT

DISCLAIMER

Poly-Data microcenter ApS makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Poly-Data microcenter ApS reserves the right to revise this publication without obligation of Poly-Data microcenter ApS to notify any person of such revision.

TRADEMARKS

COMPAS, COMPAS Pascal, COMPAS-80 and COMPAS-86 are trademarks of Poly-Data microcenter ApS. CP/M, CP/M-80 and CP/M-86 are trademarks of Digital Research Inc. MS-DOS is a trademark of Microsoft Inc.

Poly-Data microcenter ApS
Aaboulevarden 13
DK-1960 Copenhagen V, DENMARK

Telephone: +1 35 61 66
Telex: 16600 FOTEX DK, Att: microcenter

TABLE OF CONTENTS

Section 0

Introduction

COMPAS is a program development package based upon the block structured programming language Pascal. COMPAS is available in three different versions: For CP/M-80 running on a Z-80 processor and for CP/M-86 or MS-DOS running on an 8086 (or 8088) processor. The 8086 versions support the 8087 floating point co-processor. Throughout the manual, the Z-80 version is referred to as COMPAS-80 and the 8086 versions are referred to as COMPAS-86.

COMPAS includes all facilities required to create, edit, compile, run and debug programs written in Pascal. The system consists of a run time package, an on-screen editor, and a Pascal compiler, and it is fully contained in a single program occupying only 28K bytes for the Z-80 or 32K bytes for the 8086.

COMPAS Pascal closely follows the definition of Standard Pascal as contained in the "User Manual and Report" by K. Jensen and N. Wirth. In addition COMPAS Pascal offers some extensions to furthermore increase the versatility of the language.

This manual describes how to operate the COMPAS package. In programming matters you are referred to the "COMPAS Pascal Programming Manual".

The COMPAS Pascal language system and its documentation is written by Anders Hejlsberg.

0.1 System requirements

To use COMPAS-80 the following requirements must be fulfilled by your computer system:

- Z-80 microprocessor.
- CP/M 2.2 (or later) operating system.
- At least one disk drive.
- At least 48K bytes of RAM available to programs.

To use COMPAS-86 the following requirements must be fulfilled by your computer system:

- 8086 or 8088 microprocessor.
- CP/M-86 or MS-DOS operating system.
- At least one disk drive.
- At least 64K bytes of RAM available to programs.

Note that COMPAS will not run on systems with an 8080 or an 8085 microprocessor.

0.2 The distribution disk

The distribution disk contains the following files (the '.COM' expension is used by CP/M-80 and MS-DOS and the '.CMD' extension is used by CP/M-86):

README.DOC	A text file which describes the current version of the COMPAS. Before using COMPAS please display and read this file, for instance using a 'TYPE' command from the operating system.
COMPAS.COM COMPAS.CMD	This file contains the COMPAS system itself, i.e. the run-time package, the editor, and the Pascal compiler.
COMPAS.ERM	The error messages file. This file contains a list of error messages used by the compiler for reporting compilation errors.
COMPAS.HLP	The help texts file. This file contains the help texts displayed by the HELP command and by the ^J editor command.
CPAS87.COM CPAS87.CMD	The 8087 version of COMPAS (shipped wirth COMPAS-86 only. This version uses the 8087 NDP (numeric data processor) for floating point calculations, but in all other aspects it is equivalent to the standard version. If your system is equipped with an 8087 co-processor, you will probably want to use this version instead of the standard version.
INSTALL.COM INSTALL.CMD	The COMPAS install program. If you have bought an uninstalled version of COMPAS, use this program to configure the package for use on your specific system. INSTALL may also be used to modify a pre-installed version to suit your individual needs. The INSTALL program is fully self-explanatory.

INSTALL.TRM
INSTALL.DAT — Install program data files. These files contain configuration data for up to 50 different computer systems and terminals.

The distribution disk may furthermore contain various demonstration programs as source texts (i.e. as '.PAS' files).

0.3 Notations used in this manual

Whenever the term "filename" is used it refers to a CP/M or MS-DOS disk file name. The general format of a disk file name is:

<drive>:<name>.<type>

where <drive> is the disk drive identifier (A-P for CP/M, A-O for MS-DOS), <name> is any combination of up to 8 letters or digits, and <type> is any combination of up to three letters or digits. The <name> field must always be specified, whereas the <drive> field and the <type> field are optional. If the <drive> field (and the colon following it) is omitted, the currently logged drive is assumed. If the <type> field (and the period preceding it) is omitted, a default type is assumed depending on the context.

Throughout the manual, hex numbers (numbers to base 16) are preceded by a '$' character, e.g. $16E0.

Section 1

Running COMPAS

1.1 Invoking COMPAS

To invoke COMPAS enter the command line:

COMPAS

If the COMPAS.COM or COMPAS.CMD file is not located on the currently logged drive, first log in this drive (for instance by entering 'B:', if COMPAS is on the disk in drive B). Once loaded the system prompts:

```
COMPAS V3.XY (rrrrrrr version, ssss CPU)
Copyright (C) 1983 Poly-Data microcenter ApS
Include error messages (Y/N)?
```

where Y and X are the release and revision numbers respectively, rrrrrrr is the name of the operating system, and ssss is the CPU type. Now type 'Y' to load the error message file (COMPAS.ERM), or any other character to omit it. If you type 'Y' at this point, the compiler will display an error message on locating an error. Otherwise, only the error number is displayed, and you will yourself have to look up the error in the "COMPAS Pascal Programming Manual".

Following a cold-start (as described above), COMPAS may be warm-started from CP/M or MS-DOS using the command line:

COMPAS *

This of course requires that no vital memory areas have been overwritten by other programs run in the meantime.

1.2 Command lines

COMPAS prompts by printing two angle brackets ('>>'). Each time this prompt appears, COMPAS is ready to accept and process a command line. When you enter a command line, you may use the following editing keys:

BACKSPACE	Backspaces one character. On most keyboards this code is generated by pressing the key marked BS, BACK, or BACKSPACE, or by pressing CTRL/H.
DEL	Same as BACKSPACE described above. On most keyboards this code is generated by pressing the key marked DEL or RUBOUT.
CTRL/X	Backspaces to the beginning of the line.
RETURN	Terminates the input line. On most keyboards this code is generated by pressing the key marked RETURN or ENTER.

Below is shown a list of the commands recognized by COMPAS (each command is described in full in the subsequent sections). All commands may be abbreviated to their first letter.

HELP	Display help texts.
LOAD	Load a source text.
SAVE	Save the source text.
NAME	Set current file name.
EDIT	Invoke the on-screen editor.
COMPILE	Compile the source text.
RUN	Run the current program.
PROGRAM	Compile into a program file.
OBJECT	Compile into an object file.
FIND	Find a run time error.
WHERE	Find error location in an include file.
DIR	Display disk directory.
USE	Change current logged drive and user.
MEMORY	Display size of text and free memory.
ZAP	Delete the source text.
QUIT	Return to the operating system.

1.3 The HELP command

The HELP command is used to display help texts. Use it whenever you are in doubt as to what to type on a command line. The command line format of the HELP command is:

HELP <command>

Where <command> is one of the commands shown above (or the first letter of one of them). If <command> is not specified, HELP displays a command summary. Otherwise, it displays a complete description of that particular command.

The HELP command only works if the COMPAS.HLP text file is present on the disk from which COMPAS was executed. If this is not the case, HELP displays:

No COMPAS.HLP file on disk

If you try to obtain help on an unexisting command, HELP displays:

No such help text

Section 2

Loading, saving and naming files

2.1 The LOAD command

The LOAD command is used to load a source text into the memory buffer. The command line format is:

LOAD <filename>

The file type defaults to '.PAS'. When a file is loaded, it is appended to the end of the source text already held within the memory buffer. On loading the file, COMPAS displays:

Loading d:filename.typ

If the file specified does not exist, COMPAS displays:

No such file

If loading the entire file would overflow the memory buffer, COMPAS displays:

File too big

For both error conditions the text already held within the memory buffer remains unchanged.

When a file is successfully loaded, the current file name is set to the name of that file. The current file name is used by the SAVE, PROGRAM, and OBJECT commands if a file name is not explicitly stated.

2.2 The SAVE command

The SAVE command is used to save the text held within the memory buffer in a disk file. The command line format is:

SAVE <filename>

The file type defaults to '.PAS'. If <filename> is omitted entirely, the current file name is used. On saving the file, COMPAS displays:

Saving d:filename.typ

If a file of the same name and type exists on the disk specified, its type is changed to '.BAK' before the new file is created (the backup file facility may be disabled using the INSTALL program, so that duplicate files are simply deleted). If the disk directory is full, COMPAS displays:

Directory is full

If there is not enough room on the disk to create a new file, COMPAS displays:

Disk is full

If one of the above errors are reported, insert a new disk, log it in using the USE command, and try SAVE again.

2.3 The NAME command

The NAME command is used to display and optionally change the current file name. The command line format is:

NAME <filename>

If <filename> is omitted entirely, the current file name is not changed, but only displayed. Otherwise the current file name is set to <filename>. The default file type is '.PAS'. The NAME command ends by displaying:

Current file is d:filename.typ

Section 3

The editor

The COMPAS on-screen editor is used to enter and edit source texts. In COMPAS-80 the size of a source text is limited only by the amount of memory available (up to 35K bytes or so depending on your system). In COMPAS-86 the maximum size of a source text is up to 60K bytes depending on the amount of memory available (usually enough memory is available to allow source texts of all 60K bytes). If a program grows to be too large for the editor to handle it in one piece, you must break it into one or more seperate texts and use include files.

The editor is invoked using the command:

EDIT

The on-screen editor is specifically designed for use with video displays. On entering the editor, the start of the text held within the memory buffer is displayed on the screen. If the text is too long for the screen, which it usually is, then only the first portion is displayed. This is the concept of a "window". The whole text is there and accessible by editor commands, but only a portion of it can be seen through the "window" of the screen. When any editor command would take you to a position in the text which is not displayed, the "window" is moved to show that portion of the text.

The cursor marks a position in the text and can be moved to any position occupied by text. The window shows the portion of text near the cursor. To see another portion of the text, simply move the cursor.

Lines of a text may be as long as you wish, but the editor is only able to display the first part of a line. If a line is longer than the screen width, a '+' is displayed in the last position of the line on the screen.

When you use the COMPAS editor, you will notice that it has a great deal of "intelligence" built in. For instance, it updates the display only when it has nothing else to do, i.e. only when you are not entering commands or characters. Furthermore, when the editor is in the process of updating the display, it still scans the keyboard for your input. If the editor cannot keep up with your input, i.e. if you type characters faster than the editor can process them, the characters are stored in a "type-ahead" buffer, and once you relax, the buffer is emptied one character at a time. Up to 64 characters can be waiting in the "type-ahead" buffer.

To enter characters into the text you simply type them. Depending on the current mode of operation, new characters will either be inserted (pushing the remainder of the line to the right), or they will replace old characters.

You command the editor through control characters. A control character is entered from the keyboard by pressing the CTRL key and a letter key simultaneously. In this manual, an up arrow prefixing a letter indicates a control character, e.g. ^A for control-A.

Since there are more editor commands than there are single control characters in the ASCII alphabet, some editor commands are invoked by entering two characters, for instance ^K^D which means control-K followed by control-D. Note that the second character of a two-character control sequence need not be entered as a control character. Thus control-D in the above example might as well have been entered as 'D' or 'd'.

To allow you to benefit fully from the special keys offered by your keyboard, COMPAS allows you to define alternate keys for invoking specific editor functions. A typical example is defining your cursor arrows to do the same as the standard cursor controls (^S, ^D, ^E, and ^X), and in that case, whenever the manual refers to a standard key, the alternate key may be used instead. Alternate keys are defined using the INSTALL program supplied on the master disk. INSTALL may also be used to display a list of the alternate keys defined for your computer.

The COMPAS editor offers four different options (or switches) called INSERT, AUTO, TABS, and ADJUST. Together they determine the current mode of operation. Each option may be activated or passivated independantly using editor commands.

When INSERT is on, characters typed at the keyboard are inserted into the text and the remainder of the current line is pushed to the right. When INSERT is off, new characters simply replace old characters on the line.

To improve the readability of Pascal programs, the lines of a source text are often indented, for instance according to the number of BEGINs preceding the line. For this purpose, COMPAS provides an AUTO (automatic tabulator) option, which, when activated, causes each new line to automatically start at the same indentation as the line above.

When TABS is on, the TAB (^I) command will insert ASCII TAB characters into the text. Otherwise, the TAB command will insert an appropriate number of blanks to move the cursor to the next TAB stop.

The ADJUST mode is used to quickly adjust the indentation of a line or a block of lines. See section 3.6 for a complete description.

The top line of the screen is reserved for the status line, which shows the current status of the editor at all times. More specifically, it shows the number of bytes in use and free, and the state of the INSERT, AUTO, TABS, and ADJUST options. When an option is on, its name appears on the status line.

3.1 Cursor movement commands

^S Move cursor left one character.

^D Move cursor right one character.

^A Move cursor left one word.

^F Move cursor right one word.

^Q^S Move cursor to beginning of line.

^Q^D Move cursor to end of line.

^E Move cursor up one line.

^X Move cursor down one line.

^Q^E Move cursor to top of screen.

^Q^X Move cursor to bottom of screen.

^R Move cursor up one page.

^C Move cursor down one page.

^Q^R Move cursor to start of text.

^Q^C Move cursor to end of text.

3.2 Mode selection commands

^V INSERT on/off. When INSERT is on and a character is typed at the keyboard, the remainder of the line is pushed to the right to make room for the character. When INSERT is off and a character is typed at the keyboard, the new character simply replaces the character under the cursor.

^Z AUTO (automatic tabulator) on/off. When the automatic tabulator is on, every new line will automatically start at the same indentation as the line above.

^B TABS on/off. When TABS is on, the TAB (^I) command will insert ASCII TAB characters into the text. Otherwise, the TAB command will insert an appropriate number of blanks to move the cursor to the next TAB stop. Using ASCII TAB characters often saves memory, but it prevents use of the ADJUST mode to adjust the indentation of blocks of lines.

^W ADJUST mode on/off. The ADJUST mode is used to adjust the indentation of blocks of lines. For further details, see section 3.6.

3.3 Editing commands

RETURN Newline (same as ^M). This command depends on whether INSERT is on or off. When INSERT is on, a CR/LF (carriage-return line-feed) sequence is inserted into the text, which causes a blank line to appear, or which breaks the current line into two if the cursor is not at the end of the line. The cursor is then moved to the beginning of the new line. If AUTO is also on, the new line will automatically start at the same indentation as the line above. If INSERT is off, the cursor is simply moved to the beginning of the next line.

^N Insert carriage return. Inserts a CR/LF (carriage-return line-feed) sequence at the cursor, which causes a blank line to appear, or which breaks the current line into two if the cursor is not at the end of the line. The cursor does not move.

TAB Tabulate (same as ^I). If TABS is on, an ASCII TAB character is inserted into the text, which causes the cursor to move to the next multiple of eight column. If TABS is off, enough blanks are inserted to move the cursor to the next multiple of eight column. Note that TABS should be off if you plan to use the ADJUST mode to adjust the indentation of your program lines later on.

DEL Delete character left. Deletes the character before the cursor, and moves the cursor left one column.

^G Delete character right. Deletes the character under the cursor. The cursor does not move.

^T Delete word. Deletes the word that starts at the current character position (a word is any group of non-blank characters). The cursor does not move.

^Y Delete line. Deletes the entire line holding the cursor, and scrolls the remainder of the screen up one line. The cursor moves to column one of the next line.

^Q DEL Delete to beginning of line. Deletes all characters before the cursor on the current line.

^Q^Y Delete to end of line. Deletes all characters after the cursor on the current line.

3.4 Block commands

The block commands operate on blocks of text. A block is delimited by a start block marker and an end block marker, and these are shown on the screen as '>' and '<' in reverse. Before issuing a block manipulation command, you must first set a start and an end block marker, and furthermore you must make sure that the cursor is not within the block.

^K^B Set start block marker. Inserts a start block marker at the current cursor position, and moves the cursor right one column. If a start block marker is already set somewhere else, it is removed.

^K^K Set end block marker. Inserts an end block marker at the current cursor position, and moves the cursor right one column. If an end block marker is already set somewhere else, it is removed.

^K^V Move block. Moves the marked block to the current cursor position, i.e. copies it to the current cursor position and removes it at its original position. The block markers are removed. This command will not operate if the cursor is within the marked block.

^K^C Copy block. Copies the marked block to the current cursor position. The block markers are not removed. This command will not operate if the cursor is within the marked block.

^K^Y Delete block. Delete the marked block as well as the block markers. This command will not operate if the cursor is within the block.

^K^P Print block. Outputs the marked block to the printer. This command will not operate if the cursor is within the block.

^K^H Remove block markers. Note that when you leave the editor, using the ^K^D command, the block markers are automatically removed.

3.5 Search/replace commands

The search/replace commands are used to quickly locate and optionally replace occurrances of a string in the text. Note that search and replace strings cannot contain CR/LF (carriage-return line-feed) sequences, and that the maximum length of such strings is 32 characters.

^Q^F Find string. On entering this command, the status line is cleared, and a 'Find?' prompt appears in its place. Now type the string to be found, and end by pressing RETURN. You may use the DEL key to correct errors. Note that TAB characters are displayed as '^I'. Once the string is input, the scanning of the text starts. If a matching string is found, the cursor is moved to the character position just after the string. Otherwise the cursor does not move. The scan only includes the text after the cursor. To include all of it, type ^Q^R before ^Q^F.

^Q^A Find and replace string. This command is an extended version of ^Q^F, which furthermore allows you to replace the string(s) found with another string. The command will prompt you for a find string (the 'Find?' prompt), a replace string (the 'Replace with?' prompt), and an option list (the 'Options (G,N)?' prompt). The 'G' option indicates a global search, and the 'N' option indicates that the string(s) found should be replaced without asking. On entering options, type the letters with no delimiters in between. If the 'N' option is not selected, a 'Replace (Y/N)?' prompt will appear each time a matching string is found, and the cursor will move between the text and the prompt in short intervals. Typing 'Y' will replace the string, whereas 'N' will leave it unchanged. If the 'G' option is selected, the search will not stop at the first occurrance, but continue until the entire text (after the initial cursor positon) has been scanned, or RETURN is entered in response to the 'Replace (Y/N)?' prompt.

^L Continue search. Repeats the last ^Q^F or ^Q^A command with the same parameters.

3.6 The ADJUST mode

The ADJUST mode is designed to make it easy to adjust the indentation of a line or a whole group of lines. You enter the ADJUST mode by pressing ^W.

Once you are in the ADJUST mode, each time ^S is typed, the whole line moves one position to the left, and each time ^D is typed, the whole line moves one position to the right. Moving the cursor up or down, using ^E or ^X, makes the same adjustment to lines above or below. Note that once a direction has been chosen (either up or down), you cannot move backwards in the opposite direction.

When the line (or group of lines) is adjusted to the desired indentation, press ^W to leave the ADJUST mode.

Note that the ADJUST mode will not correctly adjust lines containing ASCII TAB characters. Therefore, set TABS off before entering lines that may require adjustments later on.

3.7 Other editor commands

^J Help. Displays a summary of all editor commands on the screen. This command only works if the COMPAS.HLP file is present on the disk from which COMPAS was started.

^K^D Terminate editor. On entering this command, the screen is cleared and you are returned to the command mode (the '>>' prompt). If you have been correcting an error in an include file, it will be saved, and the original file will be reloaded. For further details on this, see section 4.7.

^K^X Exit editor. Under normal circumstances this command does exactly the same as ^K^D. However, if you have been correcting an error in an include file, ^K^X will not restore the original file upon exit.

3.8 Editor error messages

Editor error messages are displayed on the top line (where the status line is normally located). An example:

ERROR: No room to insert. Press <RETURN>

To reset from an error condition and restore the status line, press RETURN. There are four different error messages:

No room to insert

This message is displayed if you try to insert characters when there is no memory left.

Block not found

This message appears if you invoke a block command when no or only one block marker is set. It will also be reported if you are within the marked block on invoking the block command. In the latter case, simply move the cursor outside the block, and re-enter the command.

No COMPAS.HLP file on disk

This message is displayed if the ^J (help) command is unable to locate the COMPAS.HLP text file.

No such help text

This message is displayed if the COMPAS.HLP file does not contain an editor command summary. Under normal circumstances you will never see this message.

Section 4

The compiler

The compiler is the heart of the COMPAS Pascal language system. It is capable of translating COMPAS Pascal, as defined in the "COMPAS Pascal Programming Manual", into native machine code instructions.

When the compiler is invoked from a COMPILE or a RUN command, the object code is stored directly into memory in succession of the source text. This mode is extremely fast (up to 5000 lines are processed per minute), and once the program is compiled it can be executed immediately. COMPAS-80 users should however note that since the system requires memory for both source text and the object code at the same time, it is likely that very large programs cannot be compiled in this mode. This also applies to COMPAS-86 users running on systems with small amounts of RAM.

The PROGRAM and OBJECT commands instruct the compiler to write the object code to a disk file. This mode is of course somewhat slower than the above, but it requires less memory, and makes possible the generation of '.COM' or '.CMD' files which may be executed directly from the operating system.

When activated from a FIND command, the compiler may be used to locate a statement in the source text which corresponds to a specific address in the object code, typically the address of a run-time error. This mode is invaluable help for the debugging of a program.

4.1 The COMPILE command

When the compiler is invoked from a COMPILE command, the object is stored directly into memory in succession of the source text. Note that whenever you invoke the editor, the code produced by the COMPILE command is erased. The actions performed by the COMPILE command depends on the version of COMPAS in use.

COMPAS-80

On entry the compiler displays:

```
Compiling
```

Following a successful compilation, you are informed of the size of the object code, the size of free memory, and the size of the data area:

```
Code: rrrrr bytes (aaaa-bbbb)
Free: sssss bytes (cccc-dddd)
Data: ttttt bytes (eeee-ffff)
```

where the numbers in parentheses are the start and end addresses (in hex) of each specific area. The size of the code section does not include the run-time package.

COMPAS-86

On entry the compiler displays:

```
Compiling
```

Following a successful compilation, you are informed of the size of the code segment, the size of the data segment, and the size of free memory (used for the stack segment):

```
Code: rrrrrr bytes (aaaa paragraphs)
Data: ssssss bytes (bbbb paragraphs)
Free: tttttt bytes (cccc paragraphs)
```

where the numbers in parentheses are the paragraph sizes (in hex) of each segment. One paragraph corresponds to 16 bytes. The code segment size includes both the run-time package and the actual program code.

4.2 The RUN command

The RUN command is used to execute a program. If no object code is present, the compiler is invoked to compile the program. Assuming a successful compilation, or if the object code was already present, the message:

```
Running
```

is output, and control is transferred to the program. When the program ends, it automatically enters the command mode of COMPAS.

If a run-time error occurs, or if you interrupt the program by pressing ^C, the program will terminate displaying a status message, for instance:

```
EXECUTION ERROR 04 AT PC=254E
Program terminated
```

You may then use the FIND command to locate the statement that caused the error or was interrupted.

4.3 The PROGRAM command

The PROGRAM command is used to compile the program into a machine code program file on a disk. The command line format and the actions performed by the command depends on the version of COMPAS in use.

COMPAS-80

For COMPAS-80 the command line format is:

```
PROGRAM <filename>,<origin>,<top>
```

where <filename> is a disk file name, and <origin> and <top> are hex addresses (without the preceding '$' character). The default file type is '.COM'. If <filename> is omitted entirely, the current file name is used with its type changed to '.COM'.

<origin> specifies the start address of the object code. If it is omitted, the end address of the run-time package is assumed. <origin> values should never be less than the end address of the run-time package (to find this address, simply use the PROGRAM command without the <origin> parameter, and note the start address of the code area).

<top> specifies the address of top of memory for the program. Programs will never access locations above this address. If <top> is omitted, the current logical top of memory is assumed. Since the compiler allocates storage for variables starting at the top of memory and working downwards, programs compiled for a given memory size cannot be run on systems with smaller memory sizes.

Before compiling the program COMPAS displays:

```
Compiling to d:filename.typ
```

On compiling the program, COMPAS also writes a copy of the run-time package into the command file. The run-time package always occupies the first portion of a program file. If an origin address greater than the end address of the run-time package is specified, a gap is left in the program file. Since this area is neither accessed by the run-time routines, nor by the program code, it is a suitable place for EXTERNAL specified machine code subroutines. These may be inserted into the program file using the DDT utility supplied on your CP/M master disk.

Following a successful compilation you are informed of the size of the program code, the size of free memory, and the size of the data area:

```
Free: rrrrr bytes (aaaa-bbbb)
Code: sssss bytes (cccc-dddd)
Free: ttttt bytes (eeee-ffff)
Data: uuuuu bytes (gggg-hhhh)
```

where the numbers in parentheses are the start and end addresses (in hex) of each specific area. The size of the code section does not include the run-time package. The first line is displayed only if an origin value was specified on the command line.

COMPAS-86

For COMPAS-86 the command line format is:

```
PROGRAM <filename>,<ssegmin>,<ssegmax>,<csegmin>,<dsegmin>
```

where <filename> is a disk file name, and <ssegmin>, <ssegmax>, <csegmin>, and <dsegmin> are hex addresses (without the preceding '$' character). The default file type is '.CMD' for CP/M-86 and '.COM' for MS-DOS. If <filename> is omitted entirely, the current file name is used with its type changed to '.CMD' or '.COM'. Any one of the four hex parameters may be omitted, for instance:

```
PROGRAM B:TEST,800          (only <ssegmin>)
PROGRAM ,,,CD8,12E4         (only <csegmin> and <dsegmin>)
```

<ssegmin> and <ssegmax> specify the minimum and maximum sizes (in paragrahps) of the stack segment. <ssegmin> defaults to 100 hex (4K bytes), and <ssegmax> defaults to the value of <ssegmin>. <csegmin> and <dsegmin> specify the minimum sizes (in paragrahps) of the code and data segments. They default to the lowest possible values. They must not be larger than hex FFF (64K bytes), and usually they are only specified for programs that will chain to other programs with larger code and/or data segments.

Before compiling the program, COMPAS displays:

```
Compiling to d:filename.typ
```

On compiling the program, COMPAS also writes a copy of the run-time package into the command file. The run-time package always occupies the first portion of the code segment.

Following a successful compilation, you are informed of the size of the code segment, the size of the data segment, and the minimum size of the stack segment:

```
Code: rrrrrr bytes (aaaa paragraphs)
Data: ssssss bytes (bbbb paragraphs)
Free: tttttt bytes (cccc paragraphs)
```

where the numbers in parentheses are the paragraph sizes (in hex) of each segment. One paragraph corresponds to 16 bytes. The code segment size includes both the run-time package and the actual program code.

In COMPAS-86 the PROGRAM command actually works in two different modes. In the "compile" mode, it generates the object code and at the same time writes it to the program file. This mode only requires room for the source text and the symbol table. In the "dump" mode on the other hand, the PROGRAM command simply dumps an already existing object code into the program file, without actually compiling the source text. This mode is extremely fast and only limited by the speed of your disk system.

The PROGRAM command automatically selects the proper mode of operation. If a COMPILE command is issued before the PROGRAM command, the PROGRAM command realizes that the object code already exists within memory, and thus selects the "dump" mode. On the other hand, if no object code is present within memory prior to the PROGRAM command, the "compile" mode is selected. Since the "dump" mode is significantly faster than the "compile" mode, especially for large programs, it is recommended that you always issue a COMPILE command immediately before a PROGRAM command.

4.4 The OBJECT command

The OBJECT command is used to create object (chain) files, i.e. files that do not contain the run-time package but only the actual program code. Object files may only be activated through the chain procedure of COMPAS Pascal - they cannot be executed directly from the operating system. For further details on program chaining, please refer to the "COMPAS Pascal Programming Manual". The command line format of the OBJECT command depends on the version of COMPAS in use.

COMPAS-80

For COMPAS-80 the command line format is:

OBJECT <filename>,<origin>,<top>

where <filename> is a disk file name, and <origin> and <top> are hex addresses (without the preceding '$' character). The default file type is '.OBJ'. If <filename> is omitted entirely, the current file name is used with its type changed to '.OBJ'. For a description of <origin> and <top>, please refer to the PROGRAM command.

COMPAS-86

For COMPAS-86 the command line format is:

OBJECT <filename>

where <filename> is a disk file name. The default file type is '.CHN' (short for chain). If <filename> is omitted entirely, the current file name is used with its type changed to '.CHN'.

Since the memory allocation state is not changed by a call to the chain procedure, you need not specify segment size information when creating an object file.

It is up to you, however, to specify sufficiently large minimum segment sizes on compiling the "root" program (using the PROGRAM command), as the memory allocation state is established once and for all when the "root" program is executed from the operating system. Therefore, note the code and data segment paragraph sizes output at the end of each object file compilation, and specify the largest values when compiling the "root" program.

4.5 The FIND command

The FIND command is used to locate a statement in the source text that corresponds to an offset address in the object code. In this mode the compiler generates no object code. The command line format is:

FIND <offset>

where <offset> is the offset address of the statement to be located. The offset address must be specified in hex with no preceding '$' character. For COMPAS-80 the offset address is relative to the start address of the program code. Thus, if the program code starts at address $1E80, then 'FIND 348' will locate the statement that resides at $21C8. For COMPAS-86 the offset address is always the true program counter offset within the code segment.

If <offset> is omitted, the offset address of the most recent run-time error is substituted instead. Thus, to locate the statement that caused a run time error, simply enter a FIND command when the error is reported. On entry the compiler displays:

```
Searching
```

If the offset address is passed during compilation, the compiler stops and displays:

```
Target address found
Press <RETURN> to edit or <ESC> to abort
```

When you press RETURN the editor is invoked, and the cursor is placed at or just after the relevant section. If you press ESC you are returned to the command mode. If the offset address is out of range, the compiler outputs:

```
Target address not found
```

before returning you to the command mode.

If a run-time error occurs within an overlay subroutine (a disk resident procedure or function), the FIND command will not always correctly locate the statement that caused the error. For a discussion of this problem and a method to avoid it, please refer to section 15.9 of the "COMPAS Pascal Programming Manual".

4.6 The WHERE command

The WHERE command invokes the editor, and moves the cursor to a specific position in the text. The command line format is:

```
WHERE <offset>
```

where <offset> is a hex number (with no preceding '$' character) specifying the offset address of the spot to be located. Whenever you leave the editor, the offset address of the cursor is recorded as the default <offset> value. Thus, if you use WHERE with no argument instead of EDIT to invoke the editor, the cursor will be moved to the spot you left previously instead of to the beginning of the text. In addition, when the compiler reports an error, it also records the offset address of the error as the default WHERE argument. In this case, a WHERE command will invoke the editor and move the cursor to the spot in error.

4.7 Error handling

If an error is found during a compilation, the compiler stops and displays an error number. If the error message file was loaded on running COMPAS, an error message is displayed as well:

```
Error 04: Duplicate identifier
Press <RETURN> to edit or <ESC> to abort
```

On pressing RETURN, the editor is invoked and the cursor is moved to the spot in error. You may then edit the source text in the same way as usual. If you press ESC you are returned to the command mode.

If an error is spotted within an include file the situation is a bit more complicated. In this case the compiler displays the name of the file and the offset address of the spot in error. Assuming that the current file name is A:MAIN.PAS and that the include file name is A:FUNCLIB.PAS. The error message might then read:

```
Include file A:FUNCLIB.PAS at CC=07B2
Error 25: Unknown or invalid variable identifier
Press <RETURN> to edit or <ESC> to abort
```

The offset address (CC) is the number of characters (in hex) read from the file before the error occurred. If you press RETURN at this stage, COMPAS will proceed by saving the text currently held within memory. On doing so, it would in this case display:

```
Saving A:MAIN.PAS
```

since A:MAIN.PAS is the current file name (set through LOAD or NAME). The file is only saved if it has not been modified since it was loaded or saved the last time. Next thing the include file is loaded. In this case COMPAS would display:

```
Loading A:FUNCLIB.PAS
```

Finally COMPAS will automatically invoke the editor and move the cursor to the spot in error. You may then correct the error. If you exit the editor through ^K^D, COMPAS will automatically save the include file and reload the original file before returning to the command mode. In this case, the display would be:

```
Saving A:FUNCLIB.PAS
Loading A:MAIN.PAS
```

If you however exit the editor through ^K^X, the include file will remain the current file.

Section 5

Further commands

5.1 The DIR command

The DIR command is used to display the directory of a disk. The command line format is:

DIR <afn>

where <afn> is an ambiguous file name as the one used in a CP/M or MS-DOS DIR command, i.e. question marks (?) and asterisks (*) may be interspersed throughout the file name and type fields.

A question mark will match any character in that position, and an asterisk will match any combination of characters within the field in which it is used (actually, an asterisk in the name field is equivalent to eigth question marks, and an asterisk in the type field is equivalent to three question marks).

If both the name field and the type field are left out, leaving only the drive identifier and a colon, then all files on that drive are listed. If <afn> is omitted entirely, then all files of the currently logged drive are listed.

5.2 The USE command

The use command is used to display and set the currently logged drive (the default drive). In the CP/M versions of COMPAS it is furthermore used to log in new disks, and to set and display the current user number. The command line format depends on the version of COMPAS in use.

CP/M versions

For the CP/M versions of COMPAS the command line format is:

USE <drive><user>

where drive is a drive identifier (A-P) and user is a user number (0-15). If <drive> is specified, the currently logged drive is changed to that drive, and if <user> is specified, the current user number is changed to that number.

The USE command is furthermore used to log in new disks. Whenever a disk is changed in one of the drives, a USE command should be issued. Otherwise CP/M will report an R/O error if you try to write to that disk).

Before returning to the command level, the USE command displays the currently logged drive and user number, for instance:

Current drive is A, user 0

MS-DOS version

For the MS-DOS version the command line format is:

USE <drive>

Where <drive> is a drive identifier (A-O). If <drive> is specified, that drive becomes the default drive. Before returning to the command level, the USE command displays the identifier of the default drive, for instance:

Current drive is A

5.3 The MEMORY command

The MEMORY command is used to display the current memory allocation state. The actual display depends on the version of COMPAS in use.

COMPAS-80

COMPAS-80 displays:

```
Code: rrrrr bytes (aaaa-bbbb)
Free: sssss bytes (cccc-dddd)
Data: ttttt bytes (eeee-ffff)
```

The 'Code' and 'Data' fields are displayed only if an object code version of the current program is present in memory. The numbers in parentheses are the start and end addresses (in hex) of each specific area.

COMPAS-86

COMPAS-86 displays:

```
Code: rrrrrr bytes (aaaa paragraphs)
Data: ssssss bytes (bbbb paragraphs)
Free: tttttt bytes (cccc paragraphs)
```

The 'Code' and 'Data' fields are displayed only if an object code version of the current program is present in memory. The numbers in parentheses are the paragraph sizes (in hex) of each area. One paragraph corresponds to 16 bytes.

5.4 The ZAP command

The ZAP command erases the text held within the memory buffer, and changes the current file name to 'WORK.PAS'. As a safety precaution, ZAP prompts:

Are you sure (Y/N)?

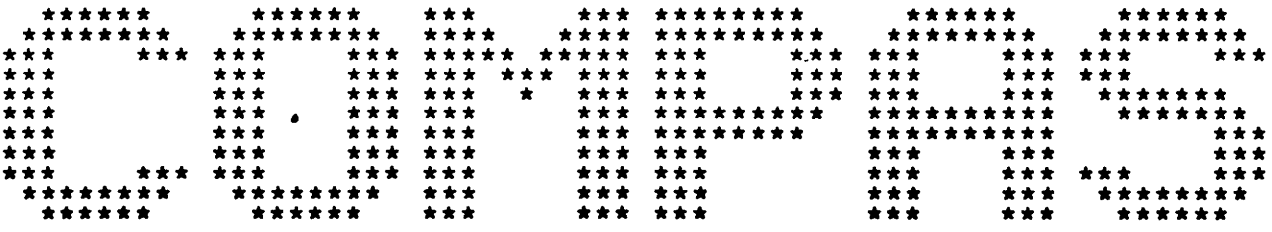
Any answer but 'Y' or 'y' will leave the text unchanged.

5.5 The QUIT command

The QUIT command transfers control to CP/M. If the source text has been edited but not saved, COMPAS prompts:

```
Text not saved. Quit (Y/N)?
```

and any answer but 'Y' or 'y' will return you to the command mode. You may later warmstart COMPAS as described in section 1.1.



C O M P A S

Pascal Program Development System

Version 3.0

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

Poly-Data microcenter ApS
Aaboulevarden 13
DK-1960 Copenhagen V

C O M P A S

Pascal Program Development System

Version 3.0

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

Poly-Data microcenter ApS
Aaboulevarden 13
DK-1960 Copenhagen V

COPYRIGHT

DISCLAIMER

Poly-Data microcenter ApS makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Poly-Data microcenter ApS reserves the right to revise this publication without obligation of Poly-Data microcenter ApS to notify any person of such revision.

TRADEMARKS

COMPAS, COMPAS Pascal, COMPAS-80 and COMPAS-86 are trademarks of Poly-Data microcenter ApS. CP/M, CP/M-80 and CP/M-86 are trademarks of Digital Research Inc. MS-DOS is a trademark of Microsoft Inc.

Poly-Data microcenter ApS
Aaboulevarden 13
DK-1960 Copenhagen V, DENMARK

Telephone: +1 35 61 66
Telex: 16600 FOTEX DK, Att: microcenter

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

Section 0

Introduction

The purpose of this manual is to define the programming language COMPAS Pascal. The manual is not meant as a tutorial. It is however, as far as possible, organized in such a way that the features of the languare are introduced in a logical order. Newcomers to Pascal are recomended to supplement the reading of this manual with a Pascal tutorial. Once you know your way about the COMPAS Pascal language, you will find the appendices in the back of the manual to be of help as a quick reference guide.

COMPAS Pascal is a superset of Standard Pascal, which is defined by K. Jensen and N. Wirth in the "Pascal User Manual and Report". COMPAS Pascal closely follows the definition of Standard Pascal - only few and minor differences exist and these are thoroughly described in section 26. Among the extensions introduced by COMPAS Pascal are:

- Dynamic strings
- Random access data files
- Structured constants
- Overlay procedures and functions
- Free ordering of sections within declaration part
- Control characters in string constants
- Type conversion functions
- Program chaining with common variables
- Include files
- Full support of operating system facilities
- Logical operations on integers
- Bit/byte manipulation
- Non-decimal integer constants
- Direct access to CPU memory and data ports
- Absolute address variables
- In-line machine code generation

In addition, some extra standard procedures and functions are offered to furthermore increase the versatility of COMPAS Pascal. All language extensions are characterized either by being absolutely necessary for the intended application area, or by being highly convenient as compared to the unextended language.

Throughout the manual, the Z-80 version of COMPAS is referred to as COMPAS-80 and the 8086 version of COMPAS is referred to as COMPAS-86.

The COMPAS Pascal language system and its documentation is written by Anders Hejlsberg.

Section 1

Basic language elements

1.1 Basic symbols

The basic vocabulary of COMPAS Pascal consists of basic symbols divided into letters, digits, and special symbols:

```
Letters:    A to Z, a to z, and underscore '_'
Digits:     0 1 2 3 4 5 6 7 8 9
Specials:   + - * / = < > ( ) [ ] { } . , ; : ' ^ @ $
```

The compiler does not distinguish between upper and lower case of letters. Certain operators and delimiters are formed using two special symbols:

1. <> <= >= := ..
2. (. and .) may be used instead of [and]
3. (* and *) may be used instead of { and }

1.2 Reserved words and standard identifiers

Reserved words are integral parts of COMPAS Pascal and cannot be redefined. Thus, reserved words should never be used as user defined identifiers. The reserved words are:

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	OVERLAY	PACKED
PROCEDURE	PROGRAM	RECORD
REPEAT	SET	SHL
SHR	STRING	THEN
TO	TYPE	UNTIL
VAR	WHILE	WITH

Throughout the manual reserved words are written in upper case letters. COMPAS Pascal also defines some standard identifiers giving names of predefined types, constants, variables, procedures, and functions. Standard identifiers can be redefined but this is strongly discouraged as it means the loss of the facilities offered by that specific identifier and often leads to confusion. The following standard identifiers are common to all versions of COMPAS:

arctan	addr	allocate
assign	aux	blockread
blockwrite	boolean	buflen
byte	chain	char
chr	close	clreol

clreos	clrhom	con
concat	copy	cos
delete	dellin	eof
eoln	erase	execute
exp	false	fill
flush	frac	gotoxy
hi	hptr	input
insert	inslin	int
integer	iores	kbd
keypress	len	length
ln	lo	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	rvsoff
rvson	seek	sin
size	sqr	sqrt
str	succ	swap
text	trm	true
trunc	usr	val
write	writeln	

The following standard identifiers are unique to COMPAS-80:

aoaddr	aiaddr	bdos
bdosb	bios	biosb
ciaddr	coaddr	csaddr
loaddr	rptr	sptr
uiaddr	uoaddr	

The following standard identifiers are unique to COMPAS-86:

aoofs	aiofs	ciofs
coofs	cseg	csofs
dseg	loofs	memw
ofs	portw	seg
sseg	swint	uiofs
uoofs		

1.3 Separators

Blanks, end of lines, and comments are considered as separators. At least one separator must occur between any two consecutive language elements.

1.4 Program lines

The maximum length of a program line is 127 characters. If a line is longer than 127 characters, all characters beyond the 127'th are ignored.

Section 2

User defined language elements

2.1 Identifiers

Identifiers serve to denote labels, constants, types, variables, procedures, and functions. An identifier consists of a letter or underscore followed by any combination of letters, digits, or underscores. All characters are significant. Some examples:

```
COMPAS    sum    root3    last_item
```

Note that COMPAS Pascal does not distinguish between upper and lower case of letters.

2.2 Numbers

Numbers are constants of type integer or of type real. Integer constants are whole numbers expressed either in decimal or hexadecimal notation. When the '$' symbol preceeds the constant it is considered a hexadecimal constant. The decimal integer range is -32768 through 32767 and the hexadecimal integer range is $0000 through $FFFF. Some examples of integer constants:

```
1    3741    -3    $20    $E7B4
```

The real magnitude is 1E-38 through 1E+38 (1E-308 through 1E+308 for the 8087 version) with a mantissa of up to 11 significant digits (15 for the 8087 version). Exponential notation may be used, in which case the letter 'E' preceding the scale factor has the meaning of "times ten to the power of". Whenever a real constant is allowed an integer constant is allowed as well. Some examples of real constants:

```
1.0    -0.025    5E10    2E-5    -3.7833654719E+12
```

No separators may occur within numbers.

2.3 Strings

Text strings are sequences of characters enclosed in single quotation marks. If the string is to contain a quote mark it should be written twice. Strings consisting of a single character are constants of the standard type char. Strings containing n (where n is greater than one) characters are constants of the types ARRAY [m..n+m-1] OF char. All string constants are compatible with all STRING types. Some examples of text strings:

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

Note the last example - the quotes enclose no characters and the string constant denotes the empty string which is only compatible with STRING types.

COMPAS Pascal also allows for control characters to be embedded in strings. Two notations for control characters are supported: The '@' symbol followed by an integer constant (in range 0..255) denotes a character of the ASCII value given by the constant, and the '^' symbol followed by a character, of ASCII value n, denotes the control character of ASCII value n-64. Some examples of control characters:

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

Control characters may be concatenated into strings if they are written with no separators in between:

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

The above strings contain two, four, two, three, and four characters respectively. Control characters may also be mixed with text strings:

```
'ERROR '^G^G^G' TRY AGAIN'    @27'[1;1f'    'Line 1'^M^J
```

The above strings contain nineteen, six, and eight characters respectively.

2.4 Comments

A comment can be inserted anywhere in the program between two language elements. It is bounded by the symbols { and } or by the symbols (* and *). Some examples:

```
{this is a comment}    (* this is also a comment *)
```

2.5 Compiler directives

Certain features of the COMPAS Pascal compiler are controlled through compiler directives. A compiler directive is actually a special form of a comment. Thus, whenever a comment is allowed, a compiler directive is allowed as well. A compiler directive list is introduced by a $ character immediately following the opening comment bracket. The syntax of the directive list depends upon the directives selected. A full description of each of the compiler directives follow later in this manual, and a summary of compiler directives is located in Appendix C. Some examples of compiler directives:

```
{$I-}    {$A+,R-,B+}    {$I STRINP.LIB}    (*$W5*)
```

Note for now, that there must be no spaces in the {$ or (*$ or immediately after the $ character.

Section 3

Standard scalar types

A data type defines the set of values a variable may assume. Every variable occurring in a program must be associated with one and only one type. Although data types in COMPAS Pascal can be quite sophisticated, each must be ultimately built from simple (unstructured) types. A simple type is either defined by the programmer, and then called a declared scalar type, or one of the five standard scalar types, integer, real, boolean, char, or byte.

3.1 The type integer

An integer is a whole number within the range -32768 through 32767, or within the range $0000 through $FFFF. Variables of type integer occupy two bytes of memory.

Overflow on integer arithmetic operations is not detected. Furthermore, partial results in integer expressions must be kept within the integer range. For instance, the expression 4000*50/25 will not yield 8000 as the multiplication causes an overflow to occur.

3.2 The type real

For COMPAS-80 and the standard version of COMPAS-86 the real magnitude is 1E-38 through 1E+38 with a mantissa of up to 11 significant digits, and variables of type real occupy 6 bytes of memory. For the 8087 version of COMPAS-86 the magnitude is 1E-308 through 1E+308 with a mantissa of up to 15 significant digits, and variables of type real occupy 8 bytes of memory.

If an overflow occurs during an arithmetic operation involving reals, the program will halt, displaying an execution error. If an underflow occurs, a result of zero is returned.

Although real is included as a standard scalar type, it cannot always be used in the same context as other scalar types. In particular, the functions pred and succ cannot take real arguments, and values of type real cannot be used in array indexing, nor for defining the base type of a set, nor in controlling FOR and CASE statements. Furthermore, subranges of type real are not allowed.

3.3 The type boolean

A boolean value is one of the logical truth values denoted by the predefined identifiers true and false. The byte boolean is defined such that false < true. A boolean variable occupies one byte of memory.

3.4 The type char

A char value is a character in the ASCII character set. Characters are ordered according to their ASCII value, such that for instance 'A' < 'B'. The ordinal (ASCII) values of characters may range from 0 to 255 (i.e. from @0 to @255). A char variable occupies one byte of memory.

3.5 The type byte

The type byte is actually a subrange of the type integer, defined such that TYPE byte = 0..255. Thus, bytes are compatible with integers, i.e. bytes and integers may be mixed in expressions and byte variables may be assigned integer values. A variable of type byte occupies one byte of memory.

Section 4

The program heading and the program block

Every program consists of a program heading followed by a program block. The block contains a declaration part, in which all objects local to the program are defined, and a statement part, which specifies the actions to be executed upon these objects.

4.1 The program heading

The program heading gives the program a name, and lists its parameters, through which the program communicates with the environment. The list consists of a sequence of identifiers separated by commas and enclosed in parentheses. Some examples of program headings:

```
PROGRAM squares;
PROGRAM calculator(input,output);
PROGRAM convert(printer,disk);
```

In COMPAS Pascal the program heading is purely optional and of no significance to the program.

4.2 The declaration part

The declaration part of a block declares all identifiers used within the statement part of that block, and other blocks within it. The declaration part is divided into 5 different sections:

1. Label declaration part
2. Constant definition part
3. Type definition part
4. Variable declaration part
5. Procedure and function declaration part

COMPAS Pascal allows each of the above sections to occur any number of times in any order in the declaration part. Standard Pascal, however, specifies that each section may only occur zero or one time, and only in the above order.

4.2.1 Label declaration part

Any statement in a program may be marked by prefixing the statement with a label followed by a colon, making possible a reference by a GOTO statement. However, the label must be declared in a label declaration part before its use. The reserved word LABEL heads this part, and it is followed by a list of label identifiers separated by commas and ended by a semicolon. An example:

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

Note that Standard Pascal specifies that labels may only be numbers of at most 4 digits, whereas COMPAS Pascal allows both numbers and identifiers to be used.

4.2.2 Constant definition part

A constant definition introduces an identifier as a synonym for a constant. The reserved word CONST heads the constant definition part, and it is followed by a list of constant assignments separated by semicolons. Each constant assignment consists of an identifier followed by an equal sign and a constant. Constants are either strings or numbers as defined in sections 2.2 and 2.3. An example:

```
CONST
  number = 45;
  max = 193.158;
  min = -max;
  name = 'Michael';
  home = @27'[1;1f]';
```

The following constants are predefined in COMPAS Pascal, i.e. they can be referred to without previous definition:

pi	real	3.1415926536E+00.
false	boolean	The truth value false.
true	boolean	The truth value true.
maxint	integer	32767.

A constant definition part may also define typed constants. This is described in full in section 12.

4.2.3 Type definition part

A data type in Pascal may be either directly described in the variable declaration or referenced by a type identifier. Not only are several standard type identifiers provided, but also a method, the type definition, for creating new types. The reserved word TYPE heads the type definition part, and it is followed by a number of type assignments separated by semicolons. Each type assignment consists of a type identifier followed by an equal sign and a type. Some examples:

```
TYPE
  word = integer;
  day = (mon,tues,wed,thur,fri,sat,sun);
  list = ARRAY[1..10] OF real;
  digits = SET OF 0..9;
  complex = RECORD re,im: real END;
```

More examples of type definitions are found in the subsequent sections.

4.2.4 Variable declaration part

Every variable occurring in a program must be declared in a variable declaration. This declaration must textually precede any use of the variable, i.e. the variable must be "known" to the compiler before it can be referred to.

A variable declaration associates an identifier and a data type with a new variable simply by listing the identifier followed by a colon and its type. This identifier/type association is valid throughout the entire block containing the declaration, unless the identifier is redefined in a subordinate block. The reserved word VAR heads the variable delcaration part. An example:

```
VAR
  root1,root2,root3: real;
  count,i: integer;
  found: boolean;
  d1,d2: day;
  buffer: ARRAY[0..127] OF byte;
```

Variables may be declared to reside at specific memory addresses. This is done by adding an AT clause to the variable declaration. The specific syntax of an AT clause depends on the version of COMPAS in use.

COMPAS-80

The reserved word AT must be followed by an integer constant giving the address of the first byte to be occupied by the variable. Some examples:

```
VAR
  memtop: integer AT $0006;
  cmdline: STRING[127] AT $80;
```

COMPAS-86

The reserved word AT must be followed by two integer constants, separated by a colon, which specify the segment base and offset of the first byte to be occupied by the variable. Some examples:

```
VAR
  int_3_ofs: integer AT $0000:$000C;
  int_3_seg: integer AT $0000:$000E;
```

To place variables at absolute offsets within the code segment or the data segment, use the identifier cseg or the identifier dseg followed by a colon and an offset:

```
VAR
  cmdline: STRING[127] AT dseg:$80;
```

The AT specification may also be used to declare a variable "on top" of another variable, or more specifically, to specify that a variable should start at the same address as another variable. When AT is followed by the identifier of a variable (or a parameter), the new variable will start at the address of that variable (or parameter). An example:

```
VAR
  str: STRING[32];
  strlen: byte AT str;
```

The above declaration specifies that strlen should start at the same address as str (and since the first byte of a string variable gives the length of the string, strlen will contain the length of str).

A variable declaration which employs an AT specification may only list one identifier before the colon.

Further details on variable space allocation are given in sections 23 and 24.

4.2.5 Procedure and function declaration part

A procedure declaration serves to define a procedure within the current procedure or program (see section 15.2). A procedure is activated from a procedure statement (see section 6.1.2).

A function declaration serves to define a program part which computes and returns a value (see section 15.3). A function is activated by the evaluation of a function designator which is a constituent of an expression (see section 5.2).

4.3 The statement part

The statement part is the ending part of a block. It specifies the algorithmic actions to be executed upon activation of the program. The statement part takes the form of a compound statement followed by a period ('.'). A compound statement consists of the reserved word BEGIN, followed by a list of statements separated by semicolons, and it is ended by the reserved word END.

A sample program:

```
PROGRAM convert(output);
CONST
  addin = 32;  mulby = 1.8;
  low = 10;  high = 19;
  separator = '----------';
TYPE
  degreetype = low..high;
VAR
  degree: degreetype;
```

```
BEGIN
  writeln(separator);
  FOR degree:=low TO high DO
  BEGIN
    write(degree:10,'c',round(degree*mulby+addin):10,'f');
    IF odd(degree) THEN writeln;
  END;
  writeln(separator);
END.
```

The program produces the following output on the screen:

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

Section 5

Expressions

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators. Expressions consist of operators and operands, i.e. variables, constants, and function designators.

This section describes how to construct expressions of the standard scalar types integer, real, boolean, and char. Expressions of declared scalar types, string types, and set types may also be constructed - this is described in sections 7.1, 8.2, and 11.2 respectively.

The rules of composition specify operator precedences according to five classes of operators. The unary minus (minus with one operand only) has the highest precedence, followed by the NOT operator, then the multiplying operators, then the adding operators, and finally with the lowest precedence the relational operators. Sequences of operators of the same precendece are evaluated from the left to the right. Any expression enclosed within parentheses is evaluated independent of preceding or succeeding operators.

5.1 Operators

If both of the operands of the arithmetic operators of addition, subtraction, and multiplication are of type integer, then the result is of type integer. If one (or both) of the operands is of type real, then the result is also of type real.

5.1.1 The unary minus

The unary minus denotes negation of its operand which may be of type real or of type integer.

5.1.2 The NOT operator

The NOT operator denotes negation (inversion) of its boolean operand:

```
NOT true      = false
NOT false     = true
```

COMPAS Pascal also allows the NOT operator to be applied to an integer operand, in which case it denotes bitwise negation. Some examples:

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

5.1.3 Multiplying operators

Operator	Operation	Type of operands	Type of result
*	Multiplication	real, integer	real, integer
/	Division	real, integer	real
DIV	Integer division	integer	integer
MOD	Modulus	integer	integer
AND	Logical AND	integer, boolean	as operand
SHL	Shift left	integer	integer
SHR	Shift right	integer	integer

5.1.4 Adding operators

Operator	Operation	Type of operands	Type of result
+	Addition	real, integer	real, integer
-	Subtraction	real, integer	real, integer
OR	Logical OR	integer, boolean	as operand
EXOR	Logical EXOR	integer, boolean	as operand

5.1.5 Relational operators

The relational operators work on all standard scalar types, real, integer, boolean, and char. Integer and real operands may be mixed. The type of the result is always boolean, i.e. true or false. The relational operators are:

```
a = b      true if a is equal to b.
a <> b     true if a is not equal to b.
a > b      true if a is greater than b.
a < b      true if a is less than b.
a >= b     true if a is greater than or equal to b.
a <= b     true if a is less than or equal to b.
```

5.2 Function designators

A function designator specifies the activation of a previously declared function or a standard function. It may contain a parameter list, which is a sequence of variables or expressions separated by commas and enclosed in parentheses. Some examples of expressions involving function designators:

```
round(degree)
sqrt(sqr(x)*sqr(y))
(max(a,b)<10) and (c>100)
volume(radius,height)
```

Section 6

Statements

Statements denote algorithmic actions and are said to be executable. The statement part of a program, a procedure, and a function is a compound statement, i.e. a sequence of statements separated by semicolons (;) and enclosed within the reserved words BEGIN and END. Statements in Pascal are either simple statements or structured statements.

6.1 Simple statements

A simple statement is a statement of which no part constitutes another statement. In this group are the assignment, procedure, GOTO, and empty statements.

6.1.1 Assignment statements

The most fundamental of all statements is the assignment statement. It specifies that a computed value is to be assigned to a variable. An assignment consists of a variable followed by the assignment operator (:=) and an expression.

Assignment is possible to variables of any type, except files. However, the variable (or the function) and the expression must be of identical type, with the exception that if the type of the variable is real, the type of the expression may be integer. Some examples of assignment statements:

```
count:=count+1;
degree:=degree+10;
found:=false;
dist:=sqrt(sqr(x)+sqr(y))
digit:=(num>='0') and (num<='9')
root:=(-b+sqrt(sqr(b)-d))/(2*a)
a:=max3(a,b,100);
```

6.1.2 Procedure statements

A procedure statement serves to denote the activation of a standard procedure or a user defined procedure. The statement consists of a procedure identifier, optionally followed by a parameter list. The parameter list is a list of variables or expressions separated by commas and enclosed in parentheses. Some examples of procedure statements:

```
seek(f,r);
sort(names);
exchange(x,y);
plot(x,round(sin(x*f)*20.0)+24);
```

6.1.3 GOTO statements

A GOTO statement consists of the reserved word GOTO followed by a label identifier. It serves to indicate that further processing should continue from that point in the program text which is marked by the label. On using GOTO statements the following rules should be observed:

The scope of a label is the block within which it is declared. It is therefore not possible to jump into and out of a procedurs and functions.

Every label must be specified in a label declaration in the declaration part of the block in which the label marks a statement.

6.1.4 Empty statements

An empty statement denotes no action and occurs whenever the syntax of Pascal requires a statement but no statement appears. Some examples:

```
BEGIN END;
WHILE digit AND (a>17) DO {nothing};
REPEAT {wait} UNTIL keypress;
```

6.2 Structured statements

Structured statements are constructs composed of other statements which are to be executed in sequence (compound statements), conditionally (conditional statements), or repeatedly (repetitive statements).

6.2.1 Compound statements

In some instances, the syntax of Pascal allows for only one statement to be specified. If more statements are to be executed in such a situation, a compound statement may be used. It consists of any number of statements separated by semicolons and enclosed within the reserved words BEGIN and END, and specifies that the component statements be executed in sequence. An example:

```
IF x>y THEN
BEGIN
  temp:=x; x:=y; y:=temp;    {exchange x and y}
END;
```

The last component statement of a compound statement need not be followed by a semicolon.

6.2.2 Conditional statements

A conditional statement selects for execution a single of its component statements.

6.2.2.1 IF statements

The IF statement specifies that a statement be executed only if a certain condition (a boolean expression) is true. If it is false, then either no statement or the statement following the reserved word ELSE is to be executed.

The syntactic ambiguity arising from the construct:

```
IF <e1> THEN IF <e2> THEN <s1> ELSE <s2>
```

is resolved by evaluating:

```
IF <e1> is false, no statement is executed.
IF <e1> is true and <e2> is true, <s1> is executed.
IF <e1> is true and <e2> is false, <s2> is executed.
```

In general, an ELSE part belongs to the last IF statement that misses and ELSE part. Some examples of IF statements:

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

IF number<0 THEN
BEGIN
  writeln('Negative numbers are not allowed');
  number:=0;
END;
```

6.2.2.2 CASE statements

The CASE statement consists of an expression (the selector) and a list of statements, each preceded by a case label. It specifies that the one statement be executed whose case label contains the current value of the selector. If none of the case labels contain the value of the selector, then either no statement is executed or the statements between the reserved words OTHERWISE and END are executed.

A case label consists of any number of constants or subranges separated by commas followed by a colon. A subrange is written as two constants separated by a lazy colon (..). The type of the constants must be the same as the type of the selector. The statement following the case label is executed if the value of the selector equals one of the constants or if it lies within one of the ranges.

Valid selector types are all simple types, i.e. all scalar types except real. Some examples of CASE statements:

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

```
CASE number OF
  1,3,5,7,9: writeln('Odd digit');
  2,4,6,8: writeln('Even digit');
  0,10..255: writeln('Zero or between 10 and 255');
OTHERWISE
  writeln('Negative or greater than 255');
  count:=count+1;
END;
```

The last statement before the reserved word OTHERWISE and the last statement before the reserved word END need not be followed by a semicolon.

6.2.3 Repetitive statements

Repetitive statements specify that certain statements are to be executed repeatedly. If the number of repetitions is known on beforehand (i.e. before the repetitions are started), the FOR statement is the appropriate construct to express this situation. Otherwise the WHILE or the REPEAT statement should be used.

6.2.3.1 WHILE statements

The expression controlling the repetition must be of type boolean. The statement is repeatedly executed as long as the expression yields true. If its value is false at the beginning, the statement is not executed at all. Some examples:

```
WHILE number<1000 DO number:=sqr(number);

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 statements

The expression controlling the repetition must be of type boolean. The sequence of statements between the reserved words REPEAT and UNTIL is repeatedly executed (and at least once) until the expression becomes true. An example:

```
REPEAT
  readln(n); sum:=sum+n;
UNTIL n=0;
```

The last component statement of a REPEAT statement need not be followed by a semicolon.

6.2.3.3 FOR statements

The FOR statement indicates that the component statement is to be repeatedly executed while a progression of values is assigned to a variable which is called the control variable. The progression can be up TO (succeeding) or DOWNTO (preceding) a final value.

The control variable, the initial value, and the final value must all be of the same type. Valid types are all simple types, i.e. all scalar types except real.

If the initial value is greater than the final value when using the TO clause, or if the initial value is less than the final value when using the DOWNTO clause, the component statement is not executed at all. Some examples of FOR statements:

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

FOR i:=1 TO n DO
BEGIN
  readln(number);
  IF number=0 THEN
  numzeroes:=numzeroes+1 ELSE
  IF number>0 THEN
  positivesum:=positivesum+number ELSE
  negativesum:=negativesum-number;
END;
```

Note that the component statement of a FOR statement may not contain assignments to the control variable. If the repetition is to be terminated before the final value is reached, a GOTO statement must be used (such constructs are however not recommended - use a WHILE or a REPEAT statement instead).

Upon completion of a FOR statement, the control variable equals the final value, unless the component statement was not executed at all, in which case no assignments are made to the control variable.

Section 7

Scalar and subrange types

The basic data types of Pascal are the scalar types. Scalar types are characterized by the fact that they constitute a finite and linear ordered set of values.

Although the standard type real is included as a scalar type, it does not conform to the above definition. Therefore, reals may not always be used in the same context as other scalar types.

7.1 Scalar types

Apart from the standard scalar types (integer, real, boolean, and char), COMPAS Pascal supports user defined scalar types, also called declared scalar types. The definition of a scalar type specifies, in order, all of its possible values. The values of the new type will be represented by identifiers, which will be the constants of the new type. Some examples:

```
TYPE
  card = (club,diamond,heart,spade);
  day = (mon,tues,wed,thur,fri,sat,sun);
  operator = (plus,minus,times,divide);
  sex = (male,female);
```

Variables of the above type card can assume one of four values, namely club, diamond, heart, or spade. You are already acquainted with the standard scalar type boolean, which is defined as:

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

The relational operators (=, <>, >, <, >=, and <=) are applicable on all scalar types, provided that both operands are of the same type (reals and integers may be mixed however). The ordering of the scalar type, i.e. the order in which the values of the type are introduced in the type definition, is used as the basis of the comparison. For the above type card, the following is true:

```
club < diamond < heart < spade
```

Standard functions with arguments of scalar type are:

```
succ(x)     The successor of x.
pred(x)     The predecessor of x.
ord(x)      The ordinal value of x.
```

The result type of succ and pred is the same as the argument type. The result type of ord is integer. The ordinal value of the first value of a scalar type is 0. Assuming the above definitions, then:

```
succ(diamond) = heart
pred(fri) = thur
ord(times) = 2
```

7.2 Subrange types

A type may be defined as a subrange of any other already defined scalar type. Such types are called subranges. The definition of a subrange simply indicates the least and the largest constant value in the subrange, where the lower bound must be greater than the upper bound. A subrange of type real is not allowed. Some examples:

```
TYPE
  day = (mon,tues,wed,thur,fri,sat,sun);
  workday = mon..fri;
  weekend = sat..sun;
  digit = '0'..'9';
  letter = 'A'..'Z';
  range = -99..99;
  monthlength = 28..31;
```

The types workday and weekend shown above are both subranges of the scalar type day, also known as their associated scalar type. The associated scalar type of digit and letter is char, and the associated scalar type of range and monthlength is integer. You are already acquainted with the standard subrange type byte, which is defines as:

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

A subrange type retains all the properties of its associated scalar type with a restriction on the range of its values.

The use of declared scalar types and subrange types is strongly recommended as it greatly improves the readability of programs. Furthermore, run time checks are included in the program code, unless otherwise specified, to verify the values assigned to declared scalar variables and subrange variables. Another advantage of declared scalar types and subrange types is that they often save storage space. COMPAS Pascal allocates only one byte of storage for variables of a declared scalar type or a subrange type with a total number of elements less than 256. Similary, integer subrange variables, where lower and upper bounds are both within the range 0 through 255, occupy only one byte of storage.

7.3 Type conversion

As stated earlier in this section, the ord function may be used to convert scalar types into values of type integer. Standard Pascal does however not provide a way to reverse this process, i.e. a way of converting an integer into a scalar value.

In COMPAS Pascal, a value of any scalar type may be converted into a value of any other scalar type, with the same ordinal value, by means of the retype facility. Retyping is achieved by using the type identifier of the desired type in a function designator. The function designator must specify one parameter enclosed in parentheses which may be a value of any scalar type. Assuming the type definitions given in section 7.1, then:

```
integer(heart) = 2
day(4) = fri
operator(0) = plus
char(65) = 'A'
integer('0') = 48
boolean(female) = true
```

The retype facility will not accept reals, neither as the desired type nor as the argument type.

7.4 Range checking

The generation of code to perform run time range checks on scalar and subrange variables is controlled through the R compiler option. The default setting is {$R+}, and when an assignment is made to a scalar or a subrange variable in this mode, the value assigned is checked by the program code. Otherwise, no run time check code is generated. An example:

```
PROGRAM rangecheck;
TYPE
  digit = 0..9;
VAR
  d1,d2,d3: digit;
BEGIN
  d2:=5;                {valid}
  d1:=d2+1;             {valid since d2<9}
  {$R-} d3:=167;        {invalid but causes no error}
  {$R+} d3:=55;         {invalid and causes a run time error}
END.
```

The {$R+} setting also causes range check code to be generated to check actual parameter values in procedure statements and function designators, when the formal parameter is a value parameter of a scalar or a subrange type.

Range checking should only be passivated in a properly debugged program.

Section 8

String types

The character string, which is a sequence of characters, is frequently needed in programming. For character string processing, COMPAS Pascal introduces string types. String types are structured types, and in many ways similar to array types (see section 9). There is however one major difference as the number of characters in a string (also known as the length of the string) may vary dynamically between 0 and a specified upper limit whereas the number of elements in an array is fixed.

8.1 String type definitions

The definition of a string type specifies the maximum number of characters it can contain, i.e. the maximum length of strings of that type. The maximum length follows, enclosed in square brackets, the reserved word STRING in the definition, and it must be an integer constant in the range 1 through 255. Some examples:

```
TYPE
  filename = STRING[8];
  line = STRING[72];
  hexstr = STRING[4];
```

Variables of a given string type occupy the maximum length plus one bytes of storage space. The individual characters within a string are indexed from 1 to the length of the string.

8.2 String expressions

Character strings may be computed from other character strings using string expressions. String expressions are built from string constants, string variables, function designators, and operators.

The relational operators (=, <>, >, <, >=, and <=) may be applied to string operands, the result being a boolean value (true or false). To compare two strings, single characters are compared from the left to the right. If the strings are of different lengths, but equal to the length of the shortest string, then the shortest string is considered the smaller. Strings are equal if and only if their lengths as well as their contents are identical. Some examples of string comparisons yielding true:

```
'string' = 'string'
'B' > 'A'
'ABC' < 'ABCD'
'test ' <> 'test'
'12' < '2'
```

Of higher precedence than the relational operators is the concatenation operator, which is written as a plus sign (+). The concatenation operator returns the concatenation of its two operands. Some examples:

```
'Michael '+'Jones' = 'Michael Jones'
'132'+'.'+'377' = '132.377'
'A'+'B'+'C'+'D' = 'ABCD'
```

8.3 String assignments

The assignment operator is used to assign the value of a string expression to a string variable. Some examples:

```
digits:='0123456789';
line:='this is a character string';
```

If a string variable is assigned a string which is longer than its maximum length, then only the leftmost characters are transferred. If, for instance, the variable digits above is defined to be of type STRING[4], then following the assignment it will only contain the four leftmost characters, i.e. '0123'.

8.4 String functions and procedures

The following standard string functions are available in COMPAS Pascal:

len(s) — Returns the length of the string expression s, i.e. the number of characters in s. The type of the result is integer.

pos(p,s) — p and s are string expressions, and the type of the result is integer. The pos function scans s to find the first occurrence of p within s. pos returns the index within s of the first character in the matched pattern (the index of the first character in a string is 1). If the pattern is not found, pos returns 0.

copy(s,i,n) — s is a string expression and both i and n are integer expressions. copy returns a string containing n characters from s starting at the i'th position in s. If i is greater than len(s), an empty string is returned. If i+n-1 is greater than len(s), a string of len(s)-i+1 characters is returned. If i is outside the range 1..255, a run time error occurs.

concat(strs) — strs is any number of string expressions separated by commas. The result is a string which is the concatenation of the parameters in the same order as they are written. If the length of the result is greater than 255, a run-time error occurs. Note that string concatenation may also be achieved using the plus (+) operator. concat is included only to maintain compatibility with other Pascal compilers.

The following standard string procedures are available in COMPAS Pascal:

delete(s,i,n) s is a string variable and both i and n are integer expressions. The procedure removes n characters from s starting at the i'th position. If i is greater than len(s), no characters are removed. If i+n-1 is greater than len(s), then len(s)-i+1 characters are removed. If i is outside the range 1..255, a run time error occurs.

insert(p,s,i) p is a string expression, s is a string variable, and i is an integer expression. The procedure inserts p into s at the i'th position. If i is greater than len(s), then p is concatenated to s. If the result is greater than the maximum length of s, then s will only contain the leftmost characters. If i is outside the range 1..255, a run time error occurs.

val(s,x,p) s is a string expression, x is either an integer or a real variable, and p is an integer variable. The numeric string contained in s is converted to a value of the same type as x, and stored in x. The numeric string should follow the rules that apply to numeric constants (see section 2.2). Neither leading nor trailing spaces are allowed. If no errors are detected, val sets p to 0. Otherwise p is set to the position of the first character in error (in this case the value of x is undefined).

str(p,s) p is a write parameter of type integer or of type real, and s is a string variable. For a full description of write parameters, please refer to section 16.3. The procedure converts the numeric value into a string and stores the result in s.

Below is shown a program which demonstrates the string handling facilities of COMPAS Pascal:

```
PROGRAM stringdemo(output);
VAR
  st,more,pattern,less: string[64];
  i,p: integer;
  r: real;
BEGIN
  st:='up';
  more:='what''s '+st+' doc';
  writeln('line 1: ',more);
  st:='hello there';
  writeln('line 2: ',len(st),' ',len(''));
  st:='this is a character string';
  pattern:='ch';
  writeln('line 3: ',pos(pattern,st),' ',pos('7','12345'));
  st:='keep something here';
  less:=copy(st,pos('s',st),9);
  writeln('line 4: ',less);
  writeln('line 5: ',copy('12345',3,255));
  st:='this is a very long character string';
```

```
  delete(st,pos('a',st)+2,10);
  writeln('line 6: ',st);
  st:='a is equal to b';
  insert('less than or ',st,6);
  writeln('line 7: ',st);
  st:='-1547';
  val(st,i,p);
  writeln('line 8: ',i,' ',p);
  r:=pi;
  str(r:10:6,st);
  writeln('line 9: ',st);
END.
```

When the program is run, it produces the following output:

```
line 1: what's up doc
line 2: 11 0
line 3: 11 0
line 4: something
line 5: 345
line 6: this is a character string
line 7: a is less than or equal to b
line 8: -1547 0
line 9:   3.141593
```

8.5 Strings and characters

String types and the standard scalar type char are compatible. Thus, whenever a string value is expected, a char value may be specified instead and vice versa. Furthermore, strings and characters may be mixed in expressions. When a character is assigned a string value, the length of the string must be exactly 1, or otherwise a run time error occurs.

The characters of a string variable may be accessed individually through string indexing. This is achieved by following the string variable with an index expression, of type integer, enclosed in square brackets. Some examples:

```
st[5]    line[37]    digits[i]    name[len(name)-1]
```

The character at index 0 contains the length of the string. Thus, len(s) is the same as ord(s[0]). Assignments to the length indicator are not checked to be less than the maximum length of the string variable - this is the responsibility of the programmer.

When the range check compiler option is active, i.e. when statements are compiled in the {$R+} mode, code is generated which insures that the value of a string index expression does not exceed the maximum length of the string variable. It is, however, possible to index a string beyond its current dynamic length - in such cases the characters read are random, and assignments will not affect the actual value of the string variable.

8.6 Predefined strings

COMPAS provides a number of predefined strings which may be used to control the screen from within your programs. The strings are defined using the INSTALL program. Below follows a description of what happens when they are printed.

clrhom — Clear home. Clears the screen and returns the cursor to the home position, i.e. the upper left corner.

clreos — Clear to end of screen. Clears all character positions from the cursor to the end of the screen.

clreol — Clear to end of line. Clears all character positions from the cursor to the end of the line.

inslin — Insert line. Scrolls the current line, and all lines below it, one line down, and inserts a blank line at the current line.

dellin — Delete line. Deletes the current line and scrolls up the following lines, with a blank line appearing at the bottom of the display.

rvson — Reverse on. Causes subsequent characters to be printed in reverse video (or highlight).

rvsoff — Reverse off. Cancels the attribute(s) activated by the rvson string.

Note that all strings, except clrhom, are optional, i.e. there is no guarantee that anything happens when they are printed. However, if a string is not defined, its length is zero, and the following construct can then be used to test it:

```
IF rvson='' THEN writeln('Reverse video not supported');
```

Section 9

Array types

An array is a structured type consisting of a fixed number of components (defined when the array is introduced) where all are of the same type, called the component or the base type. Each component can be explicitly denoted and directly accessed by the name of the array variable followed by the so-called index in square brackets. Indices are computed from expressions, and their type is called the index type.

9.1 Using arrays

The definition of an array type specifies both the component type and the index type. An array type is introduced by the reserved word ARRAY. Following this comes the index type, enclosed in square brackets, then the reserved word OF, and finally the component type. Some examples:

```
TYPE
  digit = 0..9;
  colour = (red,green,blue);
  list = ARRAY[1..100] OF real;
VAR
  digitname: ARRAY[digit] OF STRING[10];
  intensity: ARRAY[colour] OF digit;
  a,b: list;
```

The selection of an array component is achieved through following the array variable identifier by an index expression enclosed in square brackets. Some examples:

```
digitname[5]:='five';
intensity[green]:=7;
a[i]:=a[i]*2.5/b[j+1];
a:=b;
```

Since assignment is allowed between any two variables of identical type, entire arrays can be copied using the assignment operator as shown above (a:=b).

The R compiler option controls the generation of code which will perform range checks on array index expressions during run time. The default state is {$R+}, and when statements are compiled in this mode, all index expressions are checked against the bounds of their corresponding index type.

COMPAS-80 provides an S compiler option which allows the programmer to decide whether the code generated to subscribe arrays should be optimized with respect to code size or execution speed. The default state is {$S-}, which indicates optimization with respect to code size. The opposite state, {$S+}, indicates optimization with respect to execution speed. The S compiler option has no effect in COMPAS-86, since the code generated by the COMPAS-86 compiler is always optimized with respect to both size and execution speed.

9.2 Multidimensional arrays

The component type of an array may be any data type. In particular, the component type may again be an array, in which case the structure is said to be a multidimensional array. Some examples:

```
TYPE
  chesspiece = (king,queen,rook,knight,bishop,pawn);
  chessboard = ARRAY[1..8] OF ARRAY[1..8] OF chesspiece;
  size = 0..9;
  cube = ARRAY[size] OF ARRAY[size] OF ARRAY[size] OF real;
VAR
  board: chessboard;
  c: cube;
```

The definition of a multidimensional array can be contracted to a more convenient form by specifying all indices right away, thus:

```
  chessboard = ARRAY[1..8,1..8] OF chesspiece;
  cube = ARRAY[size,size,size] OF real;
```

A similar abbreviation is also provided for the application of more than one array selector:

```
board[3,7]      is equivalent to      board[3][7]
cube[0,4,3]     is equivalent to      c[0][4][3]
```

It is, of course, possible to define multidimensional arrays in terms of previously defined array types. An example:

```
TYPE
  vector = ARRAY[1..n] OF real;
  matrix = ARRAY[1..m] OF vector;
VAR
  v1,v2: vector;
  m: matrix;
```

In this case all of the following assignments are legal:

```
v1[i]:=m[i,j]+1.5;
m[i][j]:=v1[i]+v2[j];
v1:=v2;
v2:=m[i];
m[i]:=m[j];
```

9.3 Predefined arrays

COMPAS Pascal implements two predeclared byte arrays, called mem and port, which are used to access CPU memory and data ports.

9.3.1 The mem array

The mem array is used to access memory. Each component of the mem array is a byte, whose index corresponds to its address in memory. The syntax of references to the mem array depends on the version of COMPAS is use.

COMPAS-80

In COMPAS-80 the index type of the mem array is integer. When a value is assigned to a component of mem, it is stored at the address given by the index expression. When a component of mem is referred to in an expression, the value stored at the address given by the index expression is loaded. Some examples:

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

COMPAS-86

In COMPAS-86 the mem array requires both a segment and an offset to be specified, separated by a colon. Some examples:

```
data:=mem[seg(v):ofs(v)+32];
mem[dseg:$80]:=len(s);
```

The first expression specifies the segment base address and the second specifies the offset within that segment. Both expressions must be of type integer.

For direct access to words (integers) in the 8086 address space, COMPAS-86 provides a memw pseudo-array. It corresponds to the mem array in every aspect, except that the value loaded or stored is a word (with the least significant byte first as is standard on the 8086). Some examples:

```
bdos_segment:=memw[$0000:$0380];
stack_base:=memw[dseg:$15];
```

9.3.2 The port array

The port array is used to access the data ports of the CPU. Each element of the array represents a data port, whose port address corresponds to its index. For COMPAS-80 the index type is byte (8-bit port addresses) and for COMPAS-86 the index type is integer (16-bit port addresses). When a value is assigned to a component of port, it is OUTput to the port specified. When a component of port is referenced in an expression, its value is INput from the port specified. Some examples:

```
port[$46]:=$FF;
port[base]:=port[base] EXOR mask;
while port[$B2] AND $80=0 DO {wait};
```

The use of the port array is restricted to assignment and reference in expressions only. Hence, port elements cannot function as variable parameters to procedures and functions. Furthermore, operations referring to the entire port array (reference without index) are not allowed.

COMPAS-86 also offers a portw array which may be used for word I/O. It behaves exactly as the port array, except that 16-bits values are input and output.

9.4 Character arrays

Character arrays are arrays with one index and components of the standard scalar type char, i.e. arrays conforming to the definition:

```
ARRAY[n..m] OF char
```

where m is greater than n. Character arrays may be thought of as strings with a constant length (m-n+1). String constants may be assigned to character arrays, provided that the length of the string constant equals the length of the character array.

COMPAS Pascal allows character arrays to participate in string expressions, in which case the character array is converted into a string of length m-n+1 (where n and m are the lower and upper bounds of the index type of the character array). Thus, character arrays may be compared and manipulated in the same way as strings. Note, however, that values computed from string expressions cannot be assigned to character arrays - the only values that may be assigned are string constants and other character arrays variables of the same type.

Section 10

Record types

A record is a structure consisting of a fixed number of components, called fields. Unlike the array, components are not constrained to be of identical type. Hence, because there may be several different component types in a record, a computed selector, i.e. an index expression, is not allowed. Instead, each field is given a name, the field identifier, which is used to select it.

10.1 Using records

The definition of a record type specifies for each component its type and an identifier, called the field identifier. A record type is introduced by the reserved word RECORD followed by a field list and, to end the definition, the reserved word END. The field list is a sequence of record sections separated by semicolons. Each record section consists of one or more identifiers separated by commas followed by a colon and a type. Some examples:

```
TYPE
  complex = RECORD
              re,im: real;
            END;
  month = (jan,feb,mar,apr,may,jun,jly,aug,sep,oct,nov,dec);
  date = RECORD
           d: 1..31;
           m: month;
           y: 1900..1999;
         END;
  fullname = RECORD
               surname: STRING[32];
               noofnames: 1..3;
               forenames: ARRAY[1..3] OF STRING[16];
             END;
VAR
  x,y: complex;
  birthday: date;
  pers: fullname;
  vacation: ARRAY[1..14] OF date;
```

re, im, d, m, y, surname, noofnames, and forenames are field identifiers. All field identifiers within a record must be unique to that record. To reference a record component, the name of the record is followed by a point and the respective field identifier. Some examples:

```
x.re:=5.9;
y.im:=y.im+x.im;
birthday.d:=2;
birthday.m:=dec;
birthday.y:=1960;
pers.surname:='Jones';
pers.noofnames:=2;
```

```
pers.forenames[1]:='John';
pers.forenames[2]:='Raymond';
vacation[10]:=birthday;
x:=y;
```

Note that, similar to array types, assignment is allowed between records of identical types. Record components may be of any type. Hence, a record type may itself contain records. An example:

```
TYPE
  person = RECORD
             name: fullname;
             birthday: date;
           END;
VAR
  p1,p2,p3: person;
```

In this case the following assignments to the variables p1, p2, and p3 are all legal:

```
p1.name.surname:='Smith';
p1.name.noofnames:=1;
p1.name.forename[1]:='David';
p1.birthday.d:=17;
p1.birthday.m:=mar;
p1.birthday.y:=1951;
p2.name:=p1.name;
p3:=p2;
```

10.2 WITH statements

The above notation can be a bit tedious, and the programmer may wish to abbreviate it using the WITH statement. The WITH clause effectively opens the scope containing the field identifiers of the specified record variable, so that field identifiers may be used as variable identifiers. A WITH statement starts with the reserved word WITH. Then comes a list of record variables separated by commas followed by the reserved word DO and a statement. Within the component statement one designates a field of a record variable by designating only its field identifier (without preceding it with the notation of the entire record variable). The WITH statement below is equivalent to the series of assignments shown above:

```
WITH p1,name,birthday DO
BEGIN
  surname:='Smith';
  noofnames:=1;
  forename[1]:='David';
  d:=17;
  m:=mar;
  y:=1951;
  p2.name:=name;
  p3:=p2;
END;
```

Likewise, assuming the declaration:

```
VAR
  dates: ARRAY[1..7] OF date;
```

then the statement:

```
WITH dates[4] DO
IF m=dec THEN
BEGIN
  m:=jan; y:=y+1;
END ELSE m:=succ(m);
```

is equivalent to the statement:

```
IF dates[4].m=dec THEN
BEGIN
  dates[4].m:=jan; dates[4].y:=dates[4].y+1;
END ELSE dates[4].m:=succ(dates[4].m);
```

The form:

```
WITH r1,r2,....,rn DO
```

is equivalent to:

```
WITH r1 DO WITH r2 DO .... WITH rn DO
```

For COMPAS-80 the maximum number of nested WITH statements allowed, i.e. the maximum number of records that may be "opened" at one time, is controlled through the W compiler register. The default setting is {$W4}, which means that at most four records may be "opened" at any one time. The maximum nesting level within a specific block can be from 0 to 9 according to the setting of the W compiler register at the beginning of that block. Thus, if a {$W8} compiler directive is placed before the declaration part of a block, the statement part of that block will allow for up to 8 nested WITH statements (note that this does not mean that only 8 WITH statements may occur within that statement part - it means that no more than 8 WITH statements may active at one time). At the beginning of every block, the compiler reserves two bytes of workspace for each nesting level allowed. Thus, the {$W4} defualt setting will cause 8 bytes of workspace to be reserved for each block in the program if it is not changed.

The W compiler register has no effect in COMPAS-86, although W compiler directives are allowed. The maximum nesting level for WITH statements in COMPAS-86 is always 16.

10.3 Record variants

The syntax of a record type also makes provision for a variant part, implying that a record type may be specified as consisting of several variants. This means that different variables, although said to be of the same type, may assume structures which differ in a certain manner. The differences may consist of a different number and different types of components.

Each variant is characterised by a list, in parentheses, of declarations of its pertinent components. Each list is headed by one or more labels, and the set of lists is preceded by a CASE clause specifying the type of these labels. As an example assume the existence of:

```
TYPE
  fig = (rect,tri,circ);
```

Then one could describe figures by data of the following type:

```
TYPE
  figure = RECORD
             <fields common to all figures>;
             CASE fig OF
               rect: (<rectangle fields only>);
               tri: (<triangle fields only>);
               circ: (<circle fields only>);
           END;
```

Normally, a component (field) of the record itself indicates its currently valid variant. For example, the above defined figure type is likely to contain a common field:

```
kind: fig
```

This frequent situation can be abbreviated by including the declaration of the discriminating component, the tag field, in the CASE clause itself:

```
TYPE
  figure = RECORD
             colour: (red,green,blue);
             CASE kind: fig OF
               rect: (height,width: real);
               tri: (side1,side2,angle: real);
               circ: (radius: real);
           END;
```

The fixed part of a record, i.e. the part containing common fields, must always precede the variant part. In the above example, the colour field is the only field in the fixed part. A record can only have one variant part. In a variant, the parentheses must be present, even if they will enclose nothing.

The maintenance of tag field values is the responsibility of the programmer and not of the Pascal system. Thus, assuming the figure type above, the height field can be referred to even if the kind field does not yield the value rect. Actually, the tag field identifier may be omitted, leaving the type identifier only. Record variants which have no tag fields are known as free unions, as opposed to those who do, which are called discriminated unions. The use of free unions is infrequent and it should only be practiced by experienced programmers.

Section 11

Set types

A set is any collection of objects which are to be conceived of as a whole. The objects within a set are known as the members or the elements of that set. Some examples of sets:

A. All whole numbers between 0 and 100
B. The prime numbers whose magnitude is less than 100
C. The letters of the alphabet
D. The consonants of the alphabet

Two sets are the same if and only if their elements are the same. There is no ordering involved here, so the sets [1,3,5], [5,3,1] and [3,5,1] are all the same. If the members of one set are also members of another set, then the first set is said to be included in the second. Referring to the examples above, B is included in A and D is included in C. D is also included in D, but this is not a strict inclusion.

There are three operations involving sets, which are similar to the operations of addition, multiplication and subtraction for numbers. The union (or sum) of two sets A and B is that set whose members are members of either A or B. For instance, the union of [1,3,5,7] and [2,3,4] is [1,2,3,4,5,7]. The intersection (or product) of two sets A and B is that set whose members are the members of both A and B. Thus, the intersection of [1,3,5,7] and [2,3,4] is [3]. The relative complement of B with respect to A (written A-B) is that set whose members are members of A but not of B. For instance, [1,3,5,7]-[2,3,4] is [1,5,7].

11.1 Set types

In theory there are no limitations placed on the objects which may be members of a set. In Pascal, only a restricted form of sets are available. The members of a set must all be of the same type, called the base type of the set, and the base type must be a simple type, i.e. any scalar type except real. A set type is introduced by the reserved words SET OF followed by a simple type. Some examples:

```
TYPE
  smallinteger = SET OF 0..50;
  digit = SET OF 0..9;
  letter = SET OF 'A'..'Z';
  colourset = SET OF (red,green,blue);
  charset = SET OF char;
```

In COMPAS Pascal, the maximum number of elements in a set is 256, and the ordinal values of the bounds of the base type must both be within the range 0 through 255.

11.2 Set expressions

Set values may be computed from other set values through set expressions. Set expressions are built from set constants, set variables, set constructors, and operators.

11.2.1 Set constructors

A set constructor consists of a list of set elements or set ranges separated by commas and enclosed in square brackets. A set element is an expression of the same type as the base type of the set, and a set range is two such expressions separated by a lazy colon (..). Some examples of set constructors:

```
[1,3,5]
['C','O','M','P','A','S']
[x]
[i,j,k+3]
[1..5]
[i..j]
['A'..'Z','a'..'z','0'..'9']
[1,3..10,12]
```

The set [1..5] is the same as [1,2,3,4,5]. If i>j then [i..j] denotes an empty set. A special set constructor is [], which denotes the empty set. Since the empty set contains no expressions to indicate its base type, it is compatible with all set types.

11.2.2 Set operators

The rules of composition specify set operator precedences according to three classes of operators. Of the highest precedence is the intersection operator (*). Then comes the union and difference operators (+ and -), which are of the same precedence, and finally, with the lowest precedence, comes the relational operators and the inclusion operator. To summarize, the operators, grouped in order of precedence, are:

*	Set intersection.
+	Set union.
-	Set difference.
=	Test on equality.
<>	Test on inequality.
>=	True if the second operand is included in the first operand.
<=	True if the first operand is included in the second operand.
IN	Test on set membership. The second operand is of a set type, and the first operand is an expression of the same type as the base type of the set. The result is true if the first operand is a member of the second operand, otherwise false.

There is no operator for strict unclusion, but it may be programmed as A*B=[].

Set expressions are often useful to clarify complicated tests. For instance, the test:

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

can be expressed much clearer as:

```
IF ch IN ['E','C','S','I'] THEN
```

Likewise, the test:

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

may be abbreviated to:

```
IF ch IN ['0'..'9'] THEN
```

11.3 Set assignments

The assignment operator (:=) may be used to assign set values, computed from set expressions, to set variables. Some examples:

```
VAR
  digits: SET OF 0..9;
  vowels,consonants: SET OF 'A'..'Z';
  oddnumbers: SET OF 1..100;
BEGIN
  digits:=[0..9];
  vowels:=['A','E','I','O','U','Y'];
  consonants:=['A'..'Z']-vowels;
  oddnumbers:=[];
  FOR i:=1 TO 50 DO oddnumbers:=oddnumbers+[i+i-1];
END.
```

Section 12

Typed constants

Contrary to an untyped constant (see section 4.2.2), the definition of a typed constant speficies both the type and the value of the constant. A structured constant may be compared to a variable of the same type, with the exception that assignments to typed constants should not be made. Furthermore, the value of a typed constant is "known" upon the activation of the block within which it is defined, whereas the value of an ordinary variable is indeterminable.

Typed constants are introduced in a constant definition part in the declaration part of the block within which they are used. The typed constant identifier must be followed by a colon and a type, which again is followed by an equal sign and the actual constant.

12.1 Typed constants of unstructured types

Whenever a variable is expected, a typed constant may be specified instead. Thus, contrary to an untyped constant, a typed constant may be used as a variable parameter to a procedure or a function. Some examples of typed constants of simple types:

```
CONST
  maximum: integer = 9999;
  factor: real = -6.32774526;
  name: STRING[13] = 'COMPAS Pascal';
  crlf: STRING[2] = ^M^J;
```

The use of a typed constant is recommended if the constant is referred to often throughout the program. The reason for this is that a typed constant is included in the program code only once, whereas an untyped constant is included every time it is referred to. Since a typed constant is actually a variable with a constant value, it cannot be used in the definition of other constants or types. For instance, the following construct is not allowed:

```
CONST
  low: integer = 0;
  high: integer = 100;
TYPE
  list: ARRAY[low..high] OF real;
```

since low and high are not untyped constants.

12.2 Structured constants

Structured constants are often needed in programming to provide initialized tables and sets for tests, conversions, mapping functions, etc.

12.2.1 Array constants

An array constant consists of a set of constants separated by commas and enclosed in parentheses. Some examples:

```
TYPE
  color = (red,green,blue);
  namelist = ARRAY[color] OF STRING[5];
CONST
  colorname: namelist = ('red','green','blue');
  fact: ARRAY[1..7] OF integer = (1,2,6,24,120,720,5040);
```

The above example introduces the array constant colorname, which may be used to convert values of the scalar type color into their corresponding string representations, and fact, which may be used as a mapping function for high-speed calculations of the factorial of an integer. The components of colorname are:

```
colorname[red] = 'red'
colorname[green] = 'green'
colorname[blue] = 'blue'
```

Multidimensional array constants are also allowed, in which case the constants of each dimension must be enclosed separatly in parentheses. The innermost constants correspond to the rightmost dimensions:

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

The components of c are:

```
c[0,0,0] = 1        c[0,0,1] = 7
c[0,1,0] = 2        c[0,1,1] = 9
c[1,0,0] = 0        c[1,0,1] = 8
c[1,1,0] = 6        c[1,1,1] = 3
```

The component type of an array constant may be any type except file types and pointer types. An example of an array constant of a structured type is found in section 12.2.2 (the array constant called numbers).

Character array constants may be specified as single characters and as strings. Thus, the definition:

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

is equivalent to the more convenient form:

```
CONST
  digits: ARRAY[0..9] OF char = '0123456789';
```

12.2.2 Record constants

A record constant consists of a list of field constants separated by semicolons and enclosed in parentheses. Each field constant states the field identifier followed by a colon and a constant. Some examples:

```
TYPE
  complex = RECORD
              re,im: real;
            END;
  month = (jan,feb,mar,apr,may,jun,jly,aug,sep,oct,nov);
  date = RECORD
           d: 1..31; m: month; y: 1900..1999;
         END;
  fullname = RECORD
               surname: STRING[32];
               noofnames: 1..3;
               forenames: ARRAY[1..3] OF STRING[16];
             END;
CONST
  c1: complex = (re: 4.75; im: -8.0);
  currentdate: date = (d: 12; m: jan; y: 1982);
  jones: fullname = (surname: 'Jones';
                     noofnames: 2;
                     forenames: ('John','Raymond',''));
  numbers: ARRAY[1..3] OF complex =
    ((re: 1; im: 2),(re: 0; im: 37),(re: -3; im: 8.5));
```

Field constants must be specified in the same order as they appear in the definition of the record type. If a record contains fields of file types or pointer types, then constants of that record type cannot be specified. If a record constant contains a variant, then it is the responsibility of the programmer to speficy only the fields of the valid variant. If the variant contains a tag field, its value must be specified as well.

12.2.3 Set constants

The syntax of a set constant is exactly the same as that of a set constructor (see section 11.2.1), except that the elements or range bounds may only be constants. Some examples of set constants:

```
TYPE
  digitset = SET OF 0..9;
  letterset = SET OF 'A'..'Z';
CONST
  evendigits: digitset = [0,2,4,6,8];
  vowels: letterset = ['A','E','I','O','U','Y'];
  alphanum: SET OF char = ['A'..'Z','a'..'z','0'..'9'];
```

Section 13

File types

Files are the channels through which a program can store data for later retrieval by the program itself or by another program. As opposed to other data types, the data stored in a file is not kept within memory but recorded in a disk file. Similary, the data read from a file is obtained from a disk file.

A file consists of a sequence of components, all of the same type. The number of components in a file, called the length of the file, is not fixed by the definition of the file. Internally, the system keeps track of file accesses through a file pointer. Each time a component is written to or read from a file, the file pointer of that file is advanced to the next component. Since all components of a file are of equal length, the position of a specific component can be calculated. Hence, the file pointer can be moved randomly to any component in the file.

13.1 File type definitions

A file type is introduced by the reserved words FILE OF, followed by the type of the components of the file. Some examples:

```
TYPE
  date = RECORD
           d: 1..31;
           m: 1..12;
           y: 1900..1999;
         END;
  dfile = FILE OF date;
  list = ARRAY[1..10] OF real;
VAR
  intfile: FILE OF integer;
  listfile: FILE OF list;
  datefile: FILE OF date;
```

The component type of a file may be any type, except a file type. (i.e. FILE OF FILE like type is not allowed). File variables may neither appear in assignments nor in expressions.

13.2 Operations on files

The following file handling procedures are available in COMPAS Pascal (f denotes a file variable identifier):

assign(f,s) s is a string expression which yields a CP/M or MS-DOS file name of proper format. The file name is assigned to the file variable, and all further operations on f will operate on the disk file of name s. assign should never be used on a file which is not closed.

rewrite(f) A new disk file of the name assigned to f is created and prepared for processing, and the file pointer is set to the beginning of the file. A disk file created by rewrite is initially empty, i.e. it contains no elements.

reset(f) The disk file of the name assigned to f is prepared for processing, and the file pointer is set to the beginning of the file. f must name an existing file, or otherwise an I/O error occurs.

read(f,vs) vs denotes one or more variables of the component type of f, separated by commas. Each variable is read from the disk file, and following each read operation, the file pointer is advanced to the next component.

write(f,vs) vs denotes one or more variables of the component type of f, separated by commas. Each variable is written to the disk file, and following each write operation, the file pointer is advanced to the next component.

seek(f,n) n is an integer expression. The file pointer is moved to the n'th component of the disk file. The position of the first component is 0. Note that it is not possible to seek beyond the current length of the file.

flush(f) A call to flush empties the internal sector buffer of the file f, i.e. if any write operations have taken place since the last disk update, the sector buffer is written to the disk file. Furthermore, flush insures that the next read operation will actually perfom a physical read from the disk file. Normally, calls to flush are only required by programs which run on multi-user systems, where several users may access the same disk file. flush should never be used on a closed file. Note that flush has no effect in the MS-DOS version of COMPAS-86, since an MS-DOS random access file variable does not contain a sector buffer.

close(f) The disk file associated with f is closed, and the disk directory is updated to reflect the new status of the disk file.

erase(f) The disk file associated with f is erased. If the file is open, i.e. if the file has been reset or rewritten but not closed, it is generally good programming practice to call close first.

rename(f,s) The disk file associated with f is renamed to a new name given by the string expression s. rename should never be used on an open file. Further operations on f will operate on the disk file given by the new name.

The following standard functions are applicable to files:

eof(f) A boolean function which returns true if the file pointer is positioned at the end of the disk file, i.e. beyond the last component of the file. If the file pointer is not at the end of the file, eof returns false.

position(f) An integer function which returns the current position of the file pointer. 0 corresponds to the first component of a disk file.

length(f) An integer function which returns the length of the disk file, i.e. the number of components in the file. If length(f) is zero, then the file is said to be empty.

The MS-DOS version of COMPAS-86 offers three additional file handling routines, called longseek, longpos and longlen. They correspond to seek, position and length but use reals to provide an extended range (30 bits) - thus, the second parameter in a call to longseek must be an expression of type real, and the values returned by longpos and longlen are of type real.

The use of a file should always be preceded by a call to assign, to assign a disk file name to the file variable. If read and/or write operations are to be performed, the file should then be opened through a call to rewrite or reset. As a result of this call, the file pointer is set to point at the first component of the disk file, i.e. position(f)=0. When a file is rewritten, length(f) becomes 0.

A disk file can only be expanded by adding components to the end of it. The file pointer can be moved to the end of the file by executing:

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

When a program is through performing its read/write operations on a file, it should always call the close procedure. Failing to do so may result in the loss of data, since the new status of the disk file is not properly recorded in the disk directory.

The program shown below creates a disk file called A:RANDOMS.DAT, and writes 1000 random integers between 0 and 999 to the file.

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

This program will read the random numbers from the disk file created by the program above, and calculate their average.

```
PROGRAM readrandoms;
VAR
  i,n: integer;
  sum: real;
  intfile: FILE OF integer;
BEGIN
  assign(intfile,'A:RANDOMS.DAT');
  reset(intfile);
  sum:=0.0; n:=0;
  WHILE NOT eof(intfile) DO
  BEGIN
    read(intfile,i); sum:=sum+i; n:=n+1;
  END;
  close(intfile);
  writeln('the average is ',sum/n:0:3);
END.
```

The program shown below creates a new disk file called A:TEMP.DAT and writes to it the components of the disk file A:RANDOMS.DAT in reversed order. Finally, A:RANDOMS.DAT is erased, and A:TEMP.DAT is renamed to A:RANDOMS.DAT. Note the use of the seek procedure and the length function in order to read the file backwards.

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

Users of COMPAS-86 under MS-DOS should note that if the length of a disk file does not match the record size in transfers, meaning that the last record of the disk file is not entirely filled, the last record will be padded with zeros when it is read. However, this situation will never occur if the file is always processed using file variables of the same type.

13.3 Textfiles

Textfiles are unlike all other file types in that they are not simply sequences of values of some type. The basic components of a textfile are characters, but these are internally structured into lines. Each line consists of any number of characters ended by an end-of-line marker. The file itself is ended by an end-of-file marker. The CP/M and MS-DOS operating systems use a CR/LF sequence to mark the end of a line, and a CTRL/Z character to mark the end of a file. Since the lengths of lines may differ, the position of a given line in a file cannot be calculated.

Textfiles can therefore only be processed sequentially. Furthermore, textfiles cannot be written to and read from at the same time.

13.3.1 Operations on textfiles

A textfile variable is declared by referring to the standard type identifier text. Similar to defined files, read/write operations on a textfile must be preceded by a call to assign and a call to reset or rewrite.

If a textfile is to be created, rewrite should be used to open it. In this case the only operation allowed on the file is the appending of new components to the end of it. If a textfile is to be examined, reset should be used to open it. In this case the only operation allowed on the file is the sequential reading of components. When close is called on a new textfile, i.e. a textfile which was opened using rewrite, an end-of-file mark is automatically appended to the file.

Characters are written to a textfile using the standard procedure write, and read from a textfile using the standard procedure read. For the processing of lines in a textfile, Pascal introduces the following special textfile operators (where t denotes a textfile variable identifier):

readln(t)	Skips to the beginning of the next line, i.e. skips all characters up to and including the next CR/LF sequence.
writeln(t)	Writes a line marker, i.e. a CR/LF sequence, to the textfile.
eoln(t)	A boolean function which returns true if the end of the current line has been reached, i.e. if the file pointer is positioned at the CR character of the CR/LF line marker. If eof(t) is true, eoln(t) is also true.

When applied to a textfile, the eof function returns the value true if the file pointer is positioned at the end-of-file mark (the CTRL/Z character ending the file). The seek and flush procedures and the position and length functions are not applicable to textfiles.

The program shown below will do a frequency count of the alphabetic characters in the textfile A:LETTER.TXT.

```
PROGRAM freqcount;
CONST
  letters: SET OF 'A'..'Z' = ['A'..'Z'];
  lowercase: SET OF 'a'..'z' = ['a'..'z'];
VAR
  count: ARRAY['A'..'Z'] OF integer;
  ch: char;
  t: text;
```

```
BEGIN
  FOR ch:='A' TO 'Z' DO count[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 lowercase THEN ch:=chr(ord(ch)-32);
      IF ch IN letters THEN count[ch]:=count[ch]+1;
    END;
    readln(t);
  END;
  close(t);
  writeln('character   count');
  FOR ch:='A' TO 'Z' DO
  writeln('    ',ch,count[ch]:11);
END.
```

Further extensions of the procedures read and write (for the convenient handling of legible input and output of other data types than characters) are described in section 16.

13.3.2 Logical devices

A textfile may be used to communicate with a CP/M or MS-DOS logical device such as a terminal, a printer, or a modem. This is achieved by assigning the symbolic name of the logical device to the textfile. The following logical devices are supported by COMPAS Pascal:

CON: The console device. Output is sent to the CP/M or MS-DOS console output device, typically the CRT, and input is obtained from the console input device, typically the keyboard. Contrary to the TRM: device (see below), the CON: device provides buffered input. Briefly, this means that each read or readln from a textfile assigned to the CON: device will input an entire line into a line buffer, and that the operator is provided with a suitable set of editing facilities during line input. For more details on console input, please refer to sections 13.3.3 and 16.1.

TRM: The terminal device. Output is sent to the CP/M or MS-DOS console output device, typically the CRT, and input is obtained from the console input device, typically the keyboard. Input characters are echoed, unless they are control characters. The only control character which causes output is a carriage return (CR), which is echoed as CR/LF.

KBD: The keyboard device (input only). Input is obtained from the CP/M or MS-DOS console input device, typically the keyboard. Input characters are not echoed.

LST: The list device (output only). Output is sent to the CP/M or MS-DOS list device, typically the line priner.

AUX: The auxiliary device. Output is sent to the CP/M punch device or the MS-DOS auxiliary device, and input is obtained from the CP/M reader device or MS-DOS auxiliary device. Typically, these devices refer to a modem.

USR: The user device. Output is sent to the user output routine, and input is obtained from the user input routine. For further details on user input and output, please refer to section 22.

Users of COMPAS-86 under MS-DOS should note that even though a colon is not normally needed when referring to logical devices under MS-DOS, it should be specified. In this way the COMPAS I/O system will know that the file is a device and not a disk file, and so input and output characters one at a time instead of in blocks of 128.

Similar to disk files, files assigned to logical devices must be opened through reset or rewrite before they can be used. In the case of a logical device there is no difference between the functions performed by reset and rewrite. The close procedure performs no function when applied to a file which is assigned to a logical device. If erase or rename is attempted on a logical device, an I/O error occurs.

The program shown below will list the disk file called B:MSG.TXT on the line printer.

```
PROGRAM listfile;
VAR
  ch: char;
  infile,outfile: text;
BEGIN
  assign(infile,'B:MESSAGE.TXT'); reset(infile);
  assign(outfile,'LST:'); rewrite(outfile);
  WHILE NOT eof(infile) DO
  BEGIN
    WHILE NOT eoln(infile) DO
    BEGIN
      read(infile,ch); write(outfile,ch);
    END;
    readln(infile); writeln(outfile);
  END;
  close(infile); close(outfile);
END.
```

The standard functions eof and eoln operate differently on logical devices than on disk files. In the case of a disk file, eof returns true when the next character in the file is a CTRL/Z, and eoln returns true when the next character is a CR or a CTRL/Z. Thus, eof and eoln are actually "look ahead" routines. In the case of a logical device it is however not possible to look ahead, so eof and eoln therefore operate on the last character read instead of the next character. In other words, eof returns true when the last character read was a CTRL/Z, and eoln returns true when the last character read was a CR or a CTRL/Z. As an example, consider the following program, which inputs characters from the terminal and echoes them on the printer until the operator enters a CTRL/Z.

```
PROGRAM printer;
VAR
  terminal,printer: text; ch: char;
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);
END.
```

Similar to eof and eoln, the readln procedure differs between logical devices and disk files. In the case of a disk file, readln reads all characters up to and including the CR/LF sequence, whereas on a logical device it only reads up to and including the next CR. Again, the reason for this is the inability to look ahead on logical devices, which means that the system has no way of knowing whether a LF follows the CR or not.

13.3.3 Standard files

Below is shown a list of the predefined textfiles of COMPAS Pascal. These files may be used instead of declared textfiles to save the code and data space otherwise needed for the files and their initialization. All files are preassigned to a specific logical device, and need not be rewritten or reset prior to their use (actually assign, reset, rewrite, and close may never be applied to one of these files):

input — The primary input file. This file is preassigned either to the CON: device or to the TRM: device (see below for further details).

output — The primary output file. This file is preassigned either to the CON: device or to the TRM: device (see below for further details).

con — Preassigned to the console device (CON:).

trm — Preassigned to the terminal device (TRM:).

kbd — Preassigned to the keyboard device (KBD:).

lst — Preassigned to the list device (LST:).

aux — Preassigned to the auxiliary device (AUX:).

usr — Preassigned to the user device (USR:).

The logical device referred to by the standard files input and output is determined through the use of the B compiler option. The default state is {$B+}, and in this mode the console device (CON:) is used. If a {$B-} compiler directive is placed at the beginning of the program text (note: before the declaration part), input and output will instead refer to the terminal device (TRM:). The terminal device offers no editing facilities during input, but entries may follow the formats defined by Standard Pascal. The console device, on the other hand, provides buffered input with editing facilities (see section 16.1), but it does not confirm to the standard in all aspects. Note that no differences exist between the console device and the terminal device on output operations.

Since the standard files input and output are used very frequently, they are considered the default values in textfile operations when no textfile identifier is explicitly stated. Below is shown a list of shortened textfile operations and their equivalents:

```
write(ch)     is equivalent to     write(output,ch)
read(ch)      is equivalent to     read(input,ch)
writeln       is equivalent to     writeln(output)
readln        is equivalent to     readln(input)
eof           is equivalent to     eof(input)
eoln          is equivalent to     eoln(input)
```

Further extensions of the procedures read and write (for the convenient handling of legible input and output of other data types than characters) are described in section 16.

13.4 Untyped files

An untyped file is a low-level I/O channel primarily used for direct access to a CP/M or MS-DOS disk file. An untyped file is declared with the reserved word FILE and nothing more, e.g.:

```
VAR
  datafile: FILE;
```

13.4.1 Operations on untyped files

All standard file handling procedures and functions, except read, write and flush, are allowed on untyped files. For MS-DOS however the record size must be specified when the file is reset or rewritten:

```
reset(f,s)        or        rewrite(f,s)
```

where s is an integer expression which specifies the record size in bytes. For CP/M the record size is always 128 bytes - thus, the syntax of reset and rewrite is the same as for an ordinary file.

Instead of read and write, two procedures, called blockread and blockwrite, are introduced for high-speed data transfers. The format of calls to these procedures are:

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

where f is an untyped file variable identifier, v is any variable, and n is an integer expression. The result of a call to one of these procedures is that n records are transferred from/to the disk file to/from memory starting at the first byte occupied by the variable v. It is the responsibility of the programmer to insure that enough memory space is occupied by v to accomodate the entire data transfer. A call to blockread or blockwrite also has the effect of advancing the file pointer n records.

Similar to a defined file, read/write operations on an untyped file must be preceded by a call to assign and a call to rewrite or reset. If rewrite is used, a new disk file is created and opened. If reset is used, an existing disk file opened. If rewrite or reset is used on an untyped file, close should also be used to secure proper termination of the processing of the file.

The effect of a call to the seek (or longseek) procedure or one of the functions position (or longpos), length (or longlen), and eof is exactly the same as with a defined file.

The use of an untyped file is demonstrated by the program shown below. It will output a hex dump of any disk file to a logical device or another disk file.

```
PROGRAM hexdump; {$R-,A+,S+}
TYPE
  hexstr = STRING[4];
  filename = STRING[14];
  sector = ARRAY[0..7,0..15] OF byte;
  sysfile = FILE;
VAR
  i,j,address: integer;
  inname,outname: filename;
  infile: sysfile;
  outfile: text;
  buffer: sector;

FUNCTION hex(number,digits: integer): hexstr;
CONST
  hexdigits: ARRAY[0..15] OF char = '0123456789ABCDEF';
VAR
  h: hexstr;
  d: integer;
BEGIN
  h[0]:=chr(digits);
  FOR d:=digits DOWNTO 1 DO
  BEGIN
    h[d]:=hexdigits[number AND 15];
    number:=number SHR 4;
  END;
  hex:=h;
END;

BEGIN
  write('Input file? '); readln(inname);
  write('Output file? '); readln(outname);
  assign(infile,inname); reset(infile);
  assign(outfile,outname); rewrite(outfile);
```

```
    address:=0;
    WHILE NOT eof(infile) DO
    BEGIN
      blockread(infile,buffer,1);
      FOR i:=0 TO 7 DO
      BEGIN
        write(outfile,hex(address,4));
        FOR j:=0 TO 15 DO
        write(outfile,hex(buffer[i,j],2):3);
        writeln(outfile);
        address:=address+16;
      END;
    END;
    close(infile); close(outfile);
  END.
```

The above program is written for CP/M. If it is to be run on COMPAS-86 under MS-DOS, the statement 'reset(infile)' should be changed to 'reset(infile,128)'.

MS-DOS users should note that if the length of a disk file does not match the record size used in transfers, meaning that the last record of the file is not entirely filled, the last record will be padded with zeros when it is read.

In read/write operations on untyped files, data is transferred direcly from/to the disk file to/from the variable, as opposed to other files, where data passes through a sector buffer in the file variable. Since the sector buffer is not needed for an untyped file, an untyped file variable occupies less memory than other files. If a file variable is required only to use erase, rename or other non input/output operations, an untyped file is therefore recommendable.

13.5 I/O checking

The I compiler option controls the generation of code which will check the result of I/O operations during run time. The default state is on, i.e. {$I+}. In this mode the COMPAS Pascal compiler generates calls to an I/O check routine following the code of each I/O operation. If I/O checking is disabled, using an {$I-} compiler directive, no run time checks are performed. Instead the result of each I/O operation may be monitored by calling the standard function iores. Appendix F lists the possible values returned and their meaning. Note for now that if iores is zero the operation was successful. Also note that if an error occurred all I/O operations are suspended until iores is called to examine the result. Once this happens, the error condition is reset and I/O may be performed again.

The use of the iores facility is applicable in many situations. Below is shown an example of iores use to determine whether a given disk file exists or not:

```
assign(f,'B:LETTER.TXT');
{$I-} reset(f) {$I+};
IF iores>0 THEN writeln('File does not exist.');
```

When operating in the {$I-} mode, the following standard procedures should be followed by an iores check:

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

Section 14

Pointer types

A static variable (staticly allocated) is a variable which is declared in a program and subsequently denoted by its identifier. It is called static since it exists during the entire execution of the block to which it is local. A variable may, however, be generated dynamically, and it is then called a dynamic variable.

Dynamic variables do not occur in an explicit variable declaration and cannot be referenced directly by identifiers. Instead, a pointer variable is introduced, which is merely a variable containing the memory address of the dynamic variable.

14.1 Pointer type definitions

A pointer type is introduced by the pointer symbol (^) followed by the type identifier (note: type identifier, not type) of the dynamic variables which may be allocated and referenced through pointer variables of the new type. Some examples:

```
TYPE
  intptr = ^integer;
  str40 = STRING[40];
  strptr = ^str40;
  data = RECORD
           i: integer; ch: char;
         END;
  dataptr = ^data;
```

The type identifier in a pointer type definition may refer forwards to an as yet undeclared identifier. This is the only case where the use of an identifier is allowed prior to its declaration. An example:

```
TYPE
  link = ^person;
  person = RECORD
             firstname,lastname: STRING[32];
             age: 0..100;
             next: link;
           END;
```

The above type definitions also demonstrates one very common use of pointers, wherein the dynamic variables contain a link to the next record in a chain of records.

14.2 Using pointers

The dynamic variable referenced by a pointer is accessed by following the pointer variable by the pointer symbol (^). Assuming the following declarations:

```
TYPE
  item = RECORD
           name: STRING[20];
           cost: real;
         END;
  itemptr = ^item;
  list = ARRAY[1..5] OF integer;
VAR
  pint: ^integer;
  firstitem: itemptr;
  plist: ^list;
  i: integer;
```

then the following assignments are valid:

```
pint^:=4;
firstitem^.name:='hammer';
firstitem^.cost:=2.95;
plist^[3]:=7;
plist^[1]:=pint^;
plist^[i]:=plist^[i+1]*2;
```

A dynamic variable is allocated using the standard procedure new. Referring to the declarations above, the procedure statement:

```
new(firstitem);
```

allocates a new dynamic variable of type item, and assigns its address to the pointer variable firstitem.

A pointer variable may be assigned the value of another pointer variable, provided that both pointers are of the same type. Two pointers of identical type may be tested for equality or inequality using the relational operators = and <>. The operators return a boolean result (true or false).

The pointer value NIL (note that NIL is a reserved word and not a predefined constant) is compatible with all pointer types. NIL points to no dynamic variable, and may be assigned to pointer variables to indicate the absence of a usable pointer. NIL may also be used in comparisons.

As stated earlier, one very common use of pointers is the generation of linked chains, wherein each record contains a link to the next record in the chain. The program below gives an example of this:

```
PROGRAM chain(input,output);
TYPE
  nametype = STRING[30];
  personptr = ^person;
  person = RECORD
             name: nametype;
             next: personptr;
           END;
VAR
  chain,pp: personptr;
  newname: nametype;
```

```
BEGIN
  chain:=NIL;
  WRITELN('Enter names and end with a blank line:');
  REPEAT
    readln(newname);
    IF newname<>'' THEN
    BEGIN
      new(pp);
      pp^.name:=newname;
      pp^.next:=chain;
      chain:=pp;
    END;
  UNTIL newname='';
  WRITELN('The following names were entered:');
  pp:=chain;
  WHILE pp<>NIL DO
  BEGIN
    writeln(pp^.name); pp:=pp^.next;
  END;
END.
```

Variables created by the standard procedure new are stored in a stack-like structure called the "heap". The COMPAS Pascal system controls the heap by maintaining a heap pointer which, at the beginning of a program, is initialized to the address of the first free byte in memory. On each call to new, the heap pointer is moved up towards the top of free memory a number of bytes corresponding to the size of the dynamic variable newly allocated.

If one or more dynamic variables are for some reason no longer required by a program, the standard procedures mark and release may be used to reclaim the memory allocated for these variables. The mark procedure records the state of the heap pointer (i.e the address contained in the heap pointer) in a variable. The form of a call to mark is:

```
mark(v);
```

where v is any pointer variable. The release procedure sets the heap pointer to the address contained in its argument. The form of a call to release is:

```
release(v);
```

where v is any pointer variable, previously set by mark. The following program is a simple demonstration of how mark and release can be used to control the heap:

```
PROGRAM heap;
TYPE
  item = RECORD
           name: STRING[30];
           cost: real;
         END;
VAR
  pitem: ^item;
  heapmark: ^integer;
```

```
BEGIN
  mark(heapmark);
  new(pitem);
  pitem^.name:='screwdriver';
  pitem^.cost:=1.35;
  release(heapmark);
END.
```

At the beginning of the program mark is used to record the initial state of the heap pointer in the variable heapmark (the associated type of heapmark is irrelevant, since heapmark is never used in a call to new). Then new is called to allocate a dynamic variable of type item, and the variable is initialized. Finally, release is used to reset the heap pointer to its initial state, thereby discarding the dynamic variable pointed to by pitem.

If new had been called several times between the calls to mark and release, the storage occupied by several variables would have been released at once. Note that because of the stack nature of the heap, it is not possible to release the memory space used by a single item in the middle of the heap. Also note that careless use of mark and release can leave "dangling pointers", pointing to areas of memory which are no longer part of the defined heap space.

To allocate a variable number of bytes on the heap, COMPAS provides a standard procedure called allocate. The format of a call to allocate is:

```
allocate(p,n);
```

where p is any pointer variable and n is an integer expression which specifies the number of bytes to allocate. p is set to point at the first byte of the allocated area.

To find the number of bytes available on the heap at a specific time, a program may call the standard function memavail, which requires no parameters and returns an integer result. If more than 32767 bytes of memory are available, then memavail returns a negative number. In this case, the correct number of bytes free may be calculated from 65536.0+memavail (note the use of a real constant to generate a real result - this is required since the result is greater than maxint).

Since COMPAS-86 allows the heap to be larger than 64K bytes, the integer range would not be sufficient if memavail were to return the exact number of bytes free. Therefore, the COMPAS-86 version of memavail returns the number of paragraphs free. One paragraph corresponds to 16 bytes.

More details on memory management may be found in section 24.

14.3 Direct access to pointers

COMPAS Pascal allows the programmer to directly access the address stored in a pointer. This facility may prove to be extremely valuable to the experienced programmer, since it enables pointers to point anywhere into memory. If used carelessly it is however also very dangerous, as a dynamic variable, through use of the ptr function, may be placed on top of other variables, or even worse on top of the program code.

Since the memory formats of pointers in COMPAS-80 and COMPAS-86 differ (COMPAS-80 uses 16-bit pointers whereas COMPAS-86 uses 32-bit pointers), the methods used to directly control pointers depend on the version of COMPAS in use.

COMPAS-80

For pointer manipulations COMPAS-80 provides two standard functions, called ord and ptr. ord returns an integer, which is the address contained in its pointer argument, and ptr converts its integer argument into a pointer, which is compatible with all pointer types.

COMPAS-86

Pointers in COMPAS-86 are 32-bit quantities. Therefore, a pointer cannot be converted an integer using the ord function, and the ptr function cannot convert a single integer to a pointer. Alternatives are however provided: The ofs and seg functions return the offset and segment addresses of any variable. To get at the offset and segment addresses stored in a pointer variable simply specify the variable followed by the pointer arrow, for example:

```
offset:=ofs(p^);
segment:=seg(p^);
```

COMPAS-86 pointer values are created using the addr and ptr functions. addr takes any variable as its argument and returns a pointer to that variable. ptr creates a pointer from two integer expressions, for instance:

```
p:=ptr(dseg,$80);
```

The first expression specifies the segment address and the second specifies the offset address. Both expressions must be of type integer. The pointer values returned by addr and ptr are compatible with all pointer types.

14.4 Summary of pointer related routines

The following dynamic allocation procedures are available in COMPAS Pascal:

new(p) p is a variable of any pointer type. The procedure allocates a dynamic variable of the type bound to p, and assigns its address to p.

allocate(p,n) p is a variable of any pointer type and n is an integer expression. The procedure allocates an area of n bytes on the heap and assigns its address to p.

mark(p) p is a variable of any pointer type. The procedure records the address contained in the heap pointer in the variable p.

release(p) p is a variable of any pointer type. The procedure sets the heap pointer to the address contained in the variable p.

The following heap and pointer related functions are available in COMPAS Pascal:

memavail An integer function which returns the number of bytes (COMPAS-80) or paragraphs (COMPAS-86) currently available between the heap pointer and the top of free memory. If more than 32767 bytes or paragraphs are available, a negative number is returned. In this case, the correct value is 65536.0+memavail.

ord(p) COMPAS-80 only. A function which converts the address given by the pointer value p into an integer. p may be of any pointer type. Note that ord(NIL)=0.

ptr(i) COMPAS-80 only. A function which converts the address given by the integer expression i into a pointer value compatible with all pointer types. Note that NIL=ptr(0).

ptr(s,o) COMPAS-86 only. A function which converts the segment and offset addresses given by the integer expressions s and o into a pointer value compatible with all pointer types. Note that NIL=ptr(0,0).

addr(v) COMPAS-86 only. A function which returns a pointer a variable. v is any variable. Note that if v is an array, indexing is allowed, and if v is a record, specific fields may be selected.

Also see section 15.3.2.4 which contains a list of 'unclassified' standard functions.

Section 15

Procedures and functions

A procedure is a seperate program part which may be activated from a procedure statement (see section 6.1.2). A function is very similar to a procedure, except that it computes and returns a value. A function is activated from a function designator (see section 5.2).

15.1 Parameters

Procedures and functions (in common referred to as subprograms) may have parameters. Parameters provide a substitution mechamism that allows the algorithmic actions of the subprogram to be repeated with a variation of its arguments.

A procedure statement or a function designator may contain a list of actual parameters. These are substituted for the corresponding formal parameters that are defined in the heading of the subprogram. The correspondence is established by the positioning of the parameters in the lists of actual and formal parameters. Two kinds of parameters are supported by COMPAS Pascal: Value parameters and variable parameters.

When no symbols heads a formal parameter part of a subprogram heading, the parameter(s) of this section are said to be value parameters. In this case the actual parameter must be an expression (of which a variable is a simple case). The corresponding formal parameter represents a local variable in the subprogram. As its initial value this variable receives the current value of the actual parameter (i.e. the value of the expression at the time of the call). The subprogram may then change the value of this variable by assigning to it, but this will not affect the value of the actual parameter. Hence, a value parameter can never represent a result of a computation.

When the symbol VAR heads a formal parameter part of a subprogram heading, the parameter(s) of this section are said to be variable parameters. In this case the actual parameter must be a variable. The corresponding formal parameter represents this variable during the entire execution of the subprogram. Any operation involving the formal parameter is performed directly upon the actual parameter. Hence, whenever a parameter is to represent a result of the subprogram it must be declared as a variable parameter.

All address calculations are done at the time of the call. Thus, if a variable is a component of an array, its index expression(s) are evaluated when the subprogram is called.

Note that file parameters must always be specified as variable parameters.

When a large data structure, such as an array, is to function as a parameter, the use of a variable parameter is preferrable. This will save both time and storage space, since the only information passed on to the subprogram is a word (two bytes) giving the

address of the actual parameter, as opposed to the use of a value parameter, where storage must be allocated and initialized with a copy of the entire structure.

15.2 Procedures

A procedure is either a standard procedure or a procedure declared by the programmer. Opposed to user declared procedures, standard procedures can be referred to without previous declaration. If a procedure is declared with the same name as a standard procedure, that standard procedure cannot be used within that block.

15.2.1 Procedure declarations

A procedure declaration consists of a procedure heading, a declaration part, and a statement part.

The procedure heading specifies the identifier naming the procedure and an optional formal parameter list. The formal parameter list is a sequence of formal parameter parts separated by semicolons and enclosed in parentheses. A formal parameter part is a list of identifiers separated by commas followed by a colon and a type identifier. If a formal parameter part is preceded by the reserved word VAR, the parameters of this section are variable parameters. Otherwise the parameters are value parameters. Some examples of procedure headings:

```
PROCEDURE switchoff;
PROCEDURE drawto(x,y: integer);
PROCEDURE scale(VAR data: matrix; factor: real);
```

The declaration part of a procedure has the same form as that of a program. All identifiers introduced in the formal parameter list and the declaration part are local to the procedure declaration, which is called the scope of these identifiers. They are not known outside their scope. A procedure declaration may reference any constant, type, variable, procedure, or function global to it (i.e. defined in an outer block), or it may choose to redefine the name.

The statement part specifies the algorithmic actions to be executed upon activation of the procedure by a procedure statement. The statement part takes the form of a compound statement (see section 6.2.1). The use of a procedure identifier within the statement part of the procedure itself implies recursive execution of the procedure.

Below is shown an example of a program which uses a procedure with a value parameter. Since the actual parameter passed to the procedure is in some instances a constant (a simple expression), the formal parameter must be a value parameter.

```
PROGRAM histograms;
VAR
  i,k,n: integer;
  number: real;
```

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

BEGIN
  readln(n);
  FOR i:=1 TO n DO
  BEGIN
    readln(number);
    k:=round(number);
    IF k<0 THEN drawline(0) ELSE
    IF k>100 THEN drawline(100) ELSE
    drawline(k);
  END;
END.
```

Here is another program which uses a procedure with two variable parameters. Since a call to the procedure is to affect the values of the actual parameters, only variable parameters are usable.

```
PROGRAM compare;
VAR
  a,b: integer;

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

BEGIN
  readln(a,b);
  IF a=b THEN writeln('equal numbers') ELSE
  BEGIN
    IF b>a THEN exchange(a,b);
    writeln('the largest number is ',a);
    writeln('the smallest number is ',b);
  END;
END.
```

Note that the type of the parameters in a parameter part must be specified as a type identifier. Thus, the construct:

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

is not allowed. The correct method is to associate a type identifier with the parameter type, using a TYPE definition, and then use that type identifier as the parameter type, for instance:

```
TYPE
  list = ARRAY[1..100] OF integer;

PROCEDURE sort(data: list);
```

15.2.2 Standard procedures

A number of standard procedures are implemented by COMPAS Pascal. The standard procedures for string handling are described in section 8.4, the standard procedures for file handling are described in sections 13.2, 13.3.1, and 13.4.1, the standard procedures for the allocation of dynamic variables are described in section 14.4, and the standard procedures for input and output are described in section 16. In addition, the following standard procedures are available:

gotoxy(x,y) Moves the cursor to a specified position on the screen. x and y are integer expressions giving the new coordinates of the cursor. The upper left corner corresponds to (0,0). The call is ignored if x or y is outside range.

randomize Initializes the random generator with a random value. COMPAS-80 uses the Z-80 refresh register for the purpose of obtaining the random seed, but in COMPAS-86 the randomize procedure does nothing. To randomize the random number generator in this case, do a random number of calls to the random function.

move(s,d,n) Does a mass move of a specified number of bytes. s and d are two variables of any type, and n is an integer expression. A block of n bytes, starting at the first byte occupied by s, is copied to the block starting at the first byte occupied by d.

fill(v,n,d) Fills a specified range of memory with a specified value. v is a variable of any type, n is an integer expression, and d is an expression of type byte or of type char. n bytes, starting at the first byte occupied by v, are filled with the value d.

15.3 Functions

Similar to a procedure, a function is either a standard function or a function declared by the programmer.

15.3.1 Function declarations

A function declaration consists of a function heading, a declaration part, and a statement part.

The function heading is equivalent to the procedure heading, except that the formal parameter list must be followed by a colon and a type identifier defining the function result type. Some examples:

```
FUNCTION busy: boolean;
FUNCTION average(VAR m: matrix): real;
FUNCTION max2(a,b: real): real;
```

The result type of a function must be a scalar type (integer, real, boolean, char, declared scalar or subrange), a string type, or a pointer type.

The declaration part of a function is the same as that of a procedure.

The statement part takes the form of a compound statement (see section 6.2.1). Within the statement part at least one statement assigning a value to the function identifier must occur. This assignment determines the result of the function. The appearance of the function identifier in an expression within the function itself implies recursive execution of the function.

Below is shown an example of a program which uses a function to find the largest of four values:

```
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('the largest value is ',max4(i,j,k,l));
END.
```

Since the function max2 in the above example is nested within the function max4, max2 can only be called from max4, and not from the main program.

The program shown below demonstrates the use of a recursive function to calculate the factorial of an integer number:

```
PROGRAM calcfac;
VAR
  n: integer;

FUNCTION fac(i: integer): integer;
BEGIN
  IF i<=1 THEN fac:=1 ELSE fac:=i*fac(il);
END;

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

Note that the result type of a function must be specified as a type identifier. Thus, the construct:

```
FUNCTION hex(number,digits: integer): STRING[4];
```

is not allowed. Instead, a type identifier should be associated with the type STRING[4], using a TYPE definition, and that type identifier should then be used to define the function result type, for instance:

```
TYPE
  str4 = STRING[4];

FUNCTION hex(number,digits: integer): str4;
```

Note that if a function uses any of the procedures read, readln, write, or writeln, then this function must never be referenced by an expression within a write or writeln statement. The reason for this lies in the nature of the implementation of write and writeln - at the beginning of a call to one of these procedures, certain informations are set up for the entire call, and if a function, referenced by an expression within the write or writeln procedure statement, were to call write or writeln "again", then these informations would be corrupted.

15.3.2 Standard functions

A number of standard functions are implemented by COMPAS Pascal. The standard functions for string handling are described in section 8.4, the standard functions for file handling are described in section 13.2 and 13.3.1, and the standard functions relating to pointers are described in section 14.4.

15.3.2.1 Arithmetic functions

In the functions listed below the type of x must be either real or integer, and the type of the result is the type of x.

abs(x)	Computes the absolute value of x.
sqr(x)	Computes x*x.

In the functions listed below the type of x must be either real or integer, and the type of the result is real.

sin(x)	Computes the sine of x, where x is in radians.
cos(x)	Computes the cosine of x, where x is in radians.
arctan(x)	Computes the arccus tangent, in radians, of x.
ln(x)	Computes the natural logarithm of x.
exp(x)	Computes the square root of x.
int(x)	Computes the whole part of x, i.e. the greatest whole number less than or equal to x for x>=0, and the smallest whole number greater than or equal to x for x<0.

frac(x) Computes the fractional part of x with the same sign as x, i.e. frac(x)=x-int(x).

15.3.2.2 Scalar functions

succ(x) x is of any scalar type and the result is the successor of x (if it exists).

pred(x) x is of any scalar type and the result is the predecessor of x (if it exists).

odd(x) x is of type integer. The boolean value true is returned if x is odd, and the boolean value false if x is even.

15.3.2.3 Transfer functions

In addition to the functions described below, the retype facility (see section 7.3) may be used to convert values of any scalar type to values of any other scalar type.

trunc(x) The type of x is real. The result is the greatest integer less than or equal to x for x>=0, and the smallest integer greater than or equal to x for x<0.

round(x) The type of x is real, and the result is the value of x rounded, i.e.:

round(x) = trunc(x+0.5) for x>=0
round(x) = trunc(x-0.5) for x<0

ord(x) x may be of any scalar type, and the result (of type integer) is the ordinal number of the value x in the set defined by the type of x. Note that ord(x) is equivalent to integer(x).

15.3.2.4 Further standard functions

pwrten(i) i is an integer within the range -37 through 37 (-307 through 307 for the 8087 version of COMPAS-86). The result, of type real, is 10 raised to the i'th power.

random Returns a random number within the range 0<=r<1. The type of the result is real.

random(i) Returns a random integer within the range 0<=r<i.

keypress Returns the boolean value true if a key is pressed at the console, or false if no key is pressed. The result is obtained by calling the CP/M console status routine.

hi(i) The type of i is integer and the type of the result is integer. The value returned is the high order byte of i moved to the low order byte. The high order byte of the result is zero.

lo(i) The type of i is integer and the type of the result is integer. The value returned is the low order byte of i with the high order byte forced to zero.

swap(i) The type of i is integer and the type of the result is integer. The value returned is constructed from exchanging the high and low order bytes of i.

size(v) v is any variable or a type identifier. The value returned is the size of v (in bytes), or the size of a variable of type v. The type of the result is integer.

addr(v) COMPAS-80 only. v is any variable or the identifier of a procedure or a function. The value returned is the memory address of v. The type of the result is integer. Note that if v is an array, indexing is allowed, and if v is a record, specific fields may be selected.

ofs(v) COMPAS-86 only. v is any variable or the identifier of a procedure or a function. The value returned is the offset of v within its segment (procedures and functions always reside within the code segment, the base address of which may be found using the cseg function described below). If v is an array variable, indexing is allowed, and if v is a record variable, specific fields may be selected. The type of the result is integer.

seg(v) COMPAS-86 only. Returns the segment base address of any variable. As with addr and ofs, indexing and field selection is allowed. The type of tne result is integer.

cseg COMPAS-86 only. Returns the base address of the code segment. The result type is integer.

dseg COMPAS-86 only. Returns the base address of the data segment. The result type is integer.

sseg COMPAS-86 only. Returns the base address of the stack segment. The result type is integer.

15.4 FORWARD references

Calls to a procedure or a function may occur before the actual definition of the subprogram by use of a FORWARD reference. This may be convenient if two procedures or functions are mutually recursive, since it is impossible for both of them to appear before their calls. A subprogram is FORWARD referenced by separating its heading from the block. The heading is specified first,

followed by the reserved word FORWARD, and the body is then given at a later time within the same declaration part. Note that the parameters and function result type are not repeated at the body. The program shown below demonstrates the use of a FORWARD referenced procedure.

```
PROGRAM flipflop;

PROCEDURE flip(n: integer); FORWARD;

PROCEDURE flop(n: integer);
BEGIN
  writeln('entry flop. n equals ',n);
  IF n>0 THEN flip(n-1);
  writeln('exit flop. n equals ',n);
END;

PROCEDURE flip;
BEGIN
  writeln('entry flip. n equals ',n);
  IF n>0 THEN flop(n-1);
  writeln('exit flip. n equals ',n);
END;

BEGIN
  flip(3);
END.
```

When the program is executed, it outputs:

```
entry flip. n equals 3
entry flop. n equals 2
entry flip. n equals 1
entry flop. n equals 0
exit flop. n equals 0
exit flip. n equals 1
exit flop. n equals 2
exit flip. n equals 3
```

15.5 Strings as variable parameters

On using a variable parameter (a parameter declared using VAR), the type of the formal parameter and the type of the actual parameter must be one and the same. Normally this means that procedures and functions, which employ strings as variable parameters, will accept only a given string type (i.e. strings of a fixed maximum length). The programmer may override this restriction using the V compiler option. The default setting of this compiler option is {$V+}, which indicates "strict" type checking. In the "relaxed" mode, which is selected through a {$V-} compiler directive, the compiler will allow any string type as the actual parameter type, even though its maximum length is not the same as the maximum length of the formal parameter type. An example:

```
TYPE
  anystring: STRING[255];
VAR
  short: STRING[16]; long: STRING[64];

PROCEDURE uppercase(VAR s: anystring);
VAR
  i: integer;
BEGIN
  FOR i:=1 TO len(s) DO
  IF s[i] IN ['a'..'z'] THEN s[i]:=chr(ord(s[i])-32);
END;

BEGIN {$V-}
  readln(short); uppercase(short); writeln(short);
  readln(long); uppercase(long); writeln(long);
END.
```

15.6 Untyped variable parameters

When a formal parameter is an untyped parameter, the corresponding actual parameter may be any variable, regardless of its type. You already know untyped parameters from the move, fill, blockread, and blockwrite standard procedures. If the type identifier (and the preceding colon) is omitted in the declaration of a variable parameter, that parameter is considered an untyped parameter. An untyped parameter is incompatible with all other types, and it may therefore be used only in contexts where the data type is of no significance, for instance as a parameter to move, fill, blockread, blockwrite, or addr, or as the address specification in a variable declaration using the AT clause.

The blockequal function shown below demonstrates the use of untyped parameters. It compares a block of bsize bytes, starting at v1, with a corresponding block starting at v2. If the blocks are equal, it returns true, otherwise 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;
```

Assuming the declarations:

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

then blockequal may be used in tests for equality, for instance:

```
blockequal(m1,m2,size(matrix))
blockequal(m1[i],m2[j],120)
blockequal(s1[32],s1[64],32)
```

15.7 Absolute procedures and functions

This section applies to COMPAS-80 only. Normally the code generated by COMPAS-80 for procedures and functions supports fully recursive execution. In most cases recursion is however not needed, and the A compiler option is therefore provided to allow the programmer to choose between absolute and recursive subprograms. The default setting is {$A-}, which causes recursive code to be generated. An {$A+} compiler directive instructs the compiler to generate absolute code, which is more compact and executes faster. Absolute procedures and functions will only perform correctly if both of the below listed conditions are satisfied:

The procedure or function identifier must not occur in a procedure statement or an expression within the statement part of the subprogram. In other words, direct recursion must not occur.

Procedures and functions which contain calls to the subprogram must not be activated from within the subprogram. In other words, indirect recursion must not occur.

Procedures and functions in COMPAS-86 always allow for fully recursive execution. Thus, the A compiler option has no effect in COMPAS-86 (it is however allowed to maintain compatibility).

15.8 Stack overflow checks

COMPAS-86 provides a K compiler option which allows the programmer to control the generation of stack check code before subprogram calls. The default state is {$K+}, and in this state stack check code will be generated. In the opposite state, {$K-}, no stack check code is generated. The K compiler directive has no effect in COMPAS-80, but it is allowed to maintain compatibility.

15.9 Overlay procedures and functions

COMPAS allows for groups of procedures and/or functions to be separated from the main program. On developing large systems this is indeed an advantage, since such programs will occupy less memory when they are executed.

All procedures and functions declared using the reserved word OVERLAY will be separated from the main program during compilation, and placed in one or more separate overlay files. During execution of the program, the overlay subprograms are read into memory from the disk when they are called. Since more overlays may share the same memory area in the main program code (i.e. be read into and executed in the same memory area), it is possible to have several procedures and/or functions at the same cost as one ordinary subprogram.

It is important to note the difference between overlays and chained programs: Overlays are compiled together with the main program and may be used exactly as ordinary subprograms, whereas chained programs are compiled separately, and must be invoked through calls to chain or execute.

Overlays can not be used in direct mode programs, i.e. programs invoked with the RUN command. Overlays only work in program files, i.e. programs compiled with the PROGRAM or OBJECT commands.

The declaration of an overlay is identical to the declaration of an ordinary subprogram, except that it must start with the reserved word OVERLAY. Some examples:

```
OVERLAY PROCEDURE initialize;
VAR
  i: integer;
BEGIN
  FOR i:=1 TO 10 DO data[i]:=0;
  count:=0;
END;

OVERLAY PROCEDURE average(d: datalist): real;
VAR
  i: integer;
  a: real;
BEGIN
  a:=0;
  FOR i:=1 TO 25 DO a:=a+d[i];
  average:=a/25;
END;
```

On compiling a program, all overlays declared immediately after one another are placed in a single overlay file. In the main program a gap is then reserved, which is large enough to accomodate the largest of the subprograms. Below follows an example of a program with overlays:

```
PROGRAM overlay_demo_1;

OVERLAY PROCEDURE p1;
BEGIN writeln('procedure 1'); END;

OVERLAY PROCEDURE p2;
BEGIN writeln('procedure 2'); END;

OVERLAY PROCEDURE p3;
BEGIN writeln('procedure 3'); END;

BEGIN p1; p2; p3; END.
```

Assume that the above program is compiled using the PROGRAM command and the name OVDEMO. The compiler will then generate two files, OVDEMO.COM and OVDEMO.000. OVDEMO.COM contains the main program, and thereby the area into which the overlays are loaded. OVDEMO.000 contains the machine code for the procedures p1, p2, and p3. When OVDEMO.COM is invoked, p1, p2, and p3 are read into memory and executed alternately.

When an overlay subprogram is called, the run time system first checks whether the subprogram already resides in the overlay area. If so, the subprogram is invoked immediately, with no prior disk accesses. Thus, if the main program above executed five calls to p1, with no intermediate calls to p2 or p3, p1 would only be read from the disk once, at the first call.

Note that overlay subprograms residing in the same overlay file may not call each other. Thus, referring to the example above, p1, p2, and p3 may not call each other - they can be invoked only from the main program or from other subprograms.

A program is not limited to have only one overlay file. Actually, it can have up to 100 overlay files, numbered from 000 to 099. Every overlay file has a specific overlay area in the main program code, into which its overlay subprograms are loaded. Thus, when there are more overlay files, several overlay subprograms can reside in memory at the same time. As mentioned above, consecutive overlay subprograms are placed in a single overlay file. If, however, other declarations are placed in between the declarations of overlay subprograms, more overlay areas and files are allocated. An example:

```
PROGRAM overlay_demo_2;

OVERLAY PROCEDURE p1;
BEGIN END;

OVERLAY PROCEDURE p2;
BEGIN END;

PROCEDURE p3;
BEGIN END;

OVERLAY PROCEDURE p4;
BEGIN END;

OVERLAY PROCEDURE p5;
BEGIN END;

BEGIN END.
```

If the above program is compiled as OVDEMO, the compiler will produce three files: OVDEMO.COM, OVDEMO.000, and OVDEMO.001. OVDEMO.000 contains the code for p1 and p2, OVDEMO.001 contains the code for p4 and p5, and OVDEMO.COM contains the code for p3 and the main program, and two overlay areas. During execution of the program, p1 or p2 may reside in memory at the same time as p4 or p5. Note that the declaration(s) that separates the two overlay groups need not necessarily be a subprogram - a LABEL, CONST, or VAR declaration will do equally well. A comment, however, is not sufficient, since it is completely ignored by the compiler.

In terms of programming, overlays are identical to ordinary subprograms. Thus, overlays may also be nested (overlays within overlays). An example:

```
PROGRAM overlay_demo_3;

OVERLAY PROCEDURE p1;
BEGIN END;

OVERLAY PROCEDURE p2;

OVERLAY PROCEDURE p21;
BEGIN END;

OVERLAY PROCEDURE p22;
BEGIN END;

BEGIN p21; p22; END;

OVERLAY PROCEDURE p3;
BEGIN END;

BEGIN p1; p2; p3; END.
```

In this case, the compiler generates two overlay files. The 000-file contains the code for p1, p2, and p3, and the 001-file contains the code for p21 and p22. Within the code for p2, an overlay area is reserved for p21 and p22.

Since the subprograms in an overlay file may not call each other, the subprograms may share the same data area. Hence, overlays not only save code space but also data space.

When COMPAS compiles a program with overlays, the overlay files are placed on the same disk as the main program file.

During execution of a program with overlays, the system assumes that the overlay files are situated on the default disk (also known as the currently logged drive). This may however be changed using the Y compiler directive. The syntax for this directive is the letter Y immediately followed by a letter from A to P or a 0 (zero), for instance {$YA}, {$YE}, {$Y0}. A letter designates a specific disk drive and 0 designates the defalt drive. The Y directive must be situated before the first subprogram of an overlay group, and it will affect all following overlay groups, until a new directive is met.

On the first call to an overlay subprogram following the invokation of a program, the overlay file is opened by the run time system, and it remains open during the entire execution of the program. In case that a diskette, on which one or more opened overlay files reside, is to be replaced while the program is running, a re-opening must be enforced. For COMPAS-80 this is achieved by executing:

```
mem[addr(p)+36]:=0; mem[addr(p)+37]:=0;
```

For COMPAS-86 only one statement is required:

```
memw[cseg:ofs(p)+36]:=0;
```

where p is the identifier of one of the subprograms in the overlay file. In connection with the re-opening of an overlay file, it is also possible to select a new drive. For COMPAS-80 this is achieved by executing:

```
mem[addr(p)+3]:=drive;
```

For COMPAS-86 the statement is:

```
mem[cseg:ofs(p)+3]:=drive;
```

where drive is 0 (zero) for the default drive, or between 1 and 16 for drive A to P.

For overlays to be really effecient, the subprograms must be of a substantial size, and they must not be called too often. In addition, each overlay file should contain as many subprograms as possible.

If a run-time error occurs within an overlay, the FIND command will not always correctly locate the statement that caused the error. The reason for this lies in the way FIND works: To locate the statement that caused the error, FIND simply compiles the program until it reaches the specified code address. Its internal source text pointer will then be positioned at the statement in error. In an overlay group several procedures and/or functions share the same addressing range in the code, meaning that several statements occupy the same addresses, but FIND always stops the first time it reaches the target address. Therefore, FIND will usually locate a statement in the first overlay subroutine of an overlay group, although the error may have occurred in one of the following subroutines.

To get around this problem, you must determine which overlay subroutine was active at the time of error and then modify the source text so that this subroutine is the first subroutine of its overlay group. FIND will then operate correctly.

15.10 EXTERNAL specifications

The EXTERNAL specification is used to declare external procedures and functions, typically procedures and functions written in other languages, for instance machine code. An external subprogram has no block (i.e. no declaration part and no statement part). Only the subprogram heading is specified, immediately followed by an EXTERNAL specification. The actual syntax of an EXTERNAL specification depends on the version of COMPAS in use.

COMPAS-80

In COMPAS-80 an EXTERNAL specification consists of the reserved word EXTERNAL followed by an integer constant defining the memory address of the subprogram. External subprograms may have parameters, and the syntax of calls to external subprograms is exactly the same as that of calls to ordinary procedures and functions. Some examples of declarations of external subprograms:

```
PROCEDURE moveto(x,y: integer); EXTERNAL $F000;
PROCEDURE drawto(x,y: integer); EXTERNAL $F003;
FUNCTION point(x,y: integer): boolean; EXTERNAL $F006;
PROCEDURE fastsort(VAR d: namelist); EXTERNAL $1C00;
```

It is the responsibility of the programmer to insure that valid machine code actually exists at the address specified. More details on external subprograms and parameter transfers are given in section 23.3. side 104

COMPAS-86

In COMPAS-86 the EXTERNAL keyword should be followed by a string constant which names a disk file. During compilation, COMPAS will 'link' the code contained in the disk file into the program code. Some examples:

```
PROCEDURE quicksort(VAR d: namelist); EXTERNAL 'QSORT';
FUNCTION point(x,y: integer): boolean; EXTERNAL 'POINT';
```

The default file type is 'CMD' for CP/M-86 and 'COM' for MS-DOS. For 'CMD' files, COMPAS loads all bytes contained in the CODE segment (a CODE segment must be present). All other segments in the 'CMD' file are ignored. For 'COM' files, COMPAS simply loads the entire file.

External subprograms are usually coded in 8086 assembly language and then translated into machine code using the assembler/linker supplied with the operating system (ASM86/GENCMD for CP/M-86 and MACRO-86/LINK-86 for MS-DOS).

Since the 'linking' of an external file involves no relocation (it is a straightforward transfer of data from the file into the program code), it is up to the programmer to insure that the code is position independant. This means that all references to locations in the code segment should be relative to the instruction pointer (jumps are always relative). No direct references should be made to any locations in the code segment, and furthermore no references should be made to any locations in the data segment.

The formats of parameters passed to and from an external subroutine, and the methods used to access them, are described in section 23.

An external subroutine should preserve registers BP, CS, DS and SS, and it must itself provide the ending RET (within segment) instruction.

Section 16

Input and output

The basis of legible input and output are textfiles (see section 13.3) that are assigned to represent a disk file or an input and/or output device. In order to facilitate the handling of textfiles, the four standard procedures read, readln, write and writeln are extended to support a non-standard syntax for their parameter lists, allowing, among other things, for a variable number of parameters. Moreover, the parameters must not necessarily be of type char, but may also be of certain other types, in which case the data transfer is accompanied by an implicit data conversion operation. If the first parameter is a textfile variable identifier, then this is the file to be read or written. Otherwise, the standard files input and output (see section 13.3.3) are automatically assumed as default values.

16.1 The procedure read

The procedure read allows for characters, strings, and numeric values to be input. The format of the procedure statement is:

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

where v1,v2,...,vn denote variables of type char, string, integer or real. In the first case the variables are input from the the console (the standard file input). In the second case the variables are input from the textfile f. Note that f must be assigned to a disk file or a logical device and opened, using the reset procedure, before read is called to input values from it.

For a char variable, read reads one character from the file and assigns that character to the variable. For a disk file, eof is true if the next character is a CTRL/Z, and eoln is true if the next character is a CR or a CTRL/Z. For a logical device, eof is true if the character read was a CTRL/Z, and eoln is true if the character read was a CR or a CTRL/Z.

For a variable of a string type, read reads as many characters as possible into the string, unless the end of the line or the end of the file is reached. The maximum number of characters stored into the string is given by its maximum length. No distinction is made between blanks and other characters. For both disk files and logical devices, eof is true if the string was ended with a CTRL/Z, and eoln is true if the string was ended with a CR or a CTRL/Z.

For a numeric variable (integer or real), read expects to read a string of characters which can be interpreted as a numeric value of the same type (see section 2.2 for the definition of the formats of numeric constants). Any blanks, HTs, CRs, and LFs preceding the numeric string are skipped. The numeric string must not be longer than 30 characters, and it must be followed by a blank, a HT, a CR, or a CTRL/Z. If the numeric string is not of a proper format, an I/O error occurs. Otherwise the numeric string is converted to a value of the appropriate type and stored into

the variable. For a disk file, if the numeric string was ended with a blank or a HT, the next read or readln will start with the character immediately following the blank or the HT. For both disk files and logical devices, eof is true if the numeric string was ended with a CTRL/Z, and eoln is true if the numeric string was ended with a CR or a CTRL/Z. A special case of numeric input is when eof is true at the beginning at the read (or if the first character input from a logical device is CTRL/Z). Instead of assigning a new value to the variable, the current value is retained.

If the input file is assigned to the console device (CON:), or if the standard file input is used in the {$B+} mode (which is the default mode), special rules apply to the reading of variables. Upon a call to read or readln, a line is input from the console and stored into a buffer, and the reading of variables then uses this buffer as the input source. During entry of the input line, the following editing keys are available:

BACKSPACE	Backspaces one character position. On most keyboards this code is generated by pressing the key marked BS or BACKSPACE or by pressing CTRL/H.
DEL	Same as CTRL/H described above. On most keyboards this code is generated by pressing the key marked DEL or RUBOUT.
CTRL/X	Backspaces to the beginning of the line.
RETURN	Terminates the input line. On most keyboards this code is generated by pressing the key marked ENTER or RETURN.

Note that the terminating CR is not echoed. Internally, the input line is stored with a CTRL/Z appended to the end of it. Thus, when less values are specified on the input line than there are parameters in the parameter list, any char variables in excess will be set to CTRL/Z, strings will be empty, and numeric variables will remain unaltered.

Normally the maximum number of characters that can be entered on an input line from the console is 127. You may however control this limit by assigning to the predefined variable buflen. The value assigned must be an integer within the range 0 through 127. Note that assignments to buflen affect only the next read. Once this read completes its input, it restores buflen to 127. An example:

```
write('Enter filename (up to 14 chars)? ');
buflen:=14; readln(filename);
```

16.2 The procedure readln

The procedure readln is identical to the procedure read, except that after the last variable has been read, the remainder of the line is skipped, i.e. all characters up to and including the next CR/LF sequence (or the next CR for a logical device) are skipped. The format of the procedure statement is:

readln(v1,v2,...,vn) or readln(f,v1,v2,...,vn)

After any readln, the next read or readln will start with the first character of the next line. eoln is always false after a call to readln, unless eof is true. readln may also be called with no variables specified:

readln or readln(f)

When readln is used on a file which is assigned to the console device (CON:), the only difference from a corresponding call to read is that the CR terminating the input line is echoed.

16.3 The procedure write

The procedure write allows for characters, strings, booleans, and numeric values to be output. The format of a procedure statement is:

write(p1,p2,...,pn) or write(f,p1,p2,...,pn)

where p1,p2,...,pn denote so-called write parameters, which, according to the type of the value to be output, can take on one of the following formats (m, n, and i denote integer expressions, ch denotes a character expression, s denotes a string expression, b denotes a boolean expression, and r denotes a real expression):

ch ch is output.

ch:n ch is output preceded by an appropriate number of blanks to make the field width n.

s s is output with no preceding blanks. Note that arrays of characters may also be output, since they are compatible with strings.

s:n s is output preceded by an appropriate number of blanks to make the field width n.

b One of the words TRUE or FALSE is output according to the value of b.

b:n One of the words TRUE or FALSE is output preceded by an appropriate number of blanks to make the field width n.

i The decimal representation of i is output with no preceding blanks.

i:n The decimal representation of i is output preceded with an appropriate number of blanks to make the field width n.

r The decimal representation of r is output using floating point format. For COMPAS-80 and the standard version of COMPAS-86 the field width is 18 characters:

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

For the 8087 version of COMPAS-86 the field width is 23 characters:

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

where b stands for a blank, d stands for a digit, and t stands for either '+' or '-'.

r:n The decimal representation of r is output using floating point format in a field of n characters. The generalized format is:

```
r>=0.0:  <blanks>d.<digits>Etdd
r<0.0:   <blanks>-d.<digits>Etdd
```

where <blanks> denotes zero or more blanks and <digits> denotes from one to ten decimal digits. Since at least one digit is output following the decimal point, the field width is always a minimum of 7 (8 for r<0.0) characters. When n is greater than 16 (17 for r<0.0), the number is preceded by n-16 (n-17 for r<0.0) blanks. For the 8087 version of COMPAS-86, the field width is always a minimum of 8 (9 for r<0.0), and when n is greater than 21 (22 for r<0.0), the number is preceded by n-21 (n-22 for r<0.0) blanks.

r:n:m The decimal representation of r is output using fixed point format with m digits after the decimal point. m must be in the range 0<=m<=24, or otherwise floating point format is selected. The number is preceded by an appropriate number of blanks to make the field width n.

16.4 The procedure writeln

The procedure writeln is identical to the procedure write, except that after the last value has been output, a CR/LF sequence is written to the file. The format of the procedure statement is:

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

To produce a single CR/LF sequence, call writeln with no write parameters:

```
writeln     or     writeln(f)
```

Section 17

User interrupts

COMPAS supports two types of user interrupts: Interrupts during console I/O and interrupts during execution.

17.1 User interrupts during console I/O

When COMPAS receives a CTRL/S character from the keyboard during output to the console (CON: or TRM: device), the display is stopped temporarily, until another character is received. Furthermore, when COMPAS receives a CTRL/C character from the keyboard, it displays:

```
USER INTERRUPT AT PC=aaaa
Program terminated
```

and returns control to the command level (or to CP/M). The statement at which the interrupt occurred may then be located using the FIND command.

To make a program uninterruptable, i.e. to deactivate the above described facility, insert a {$C-} compiler directive at the beginning of the program (i.e. before the declaration part). CTRL/S and CTRL/C will then be processed as all other ASCII characters.

17.2 User interrupts during execution

If a program enters an indefinite loop then normally the only way to interrupt it is by pressing RESET. COMPAS however offers a U compiler directive, which instructs the compiler to generate calls to a user interrupt check routine before each statement. The default setting is {$U-} and in this mode no interrupt check calls are generated. If a {$U+} directive is issued, the compiler will generate interrupt check calls before the code of each statement, and when the program is running, such statements can be interrupted by pressing CTRL/C. Contrary to the C compiler option, the U compiler option may be used freely throughout the source text, and all statements compiled in the {$U+} state will be interruptable.

Note that statements compiled with {$U+} will execute significantly slower than uninterruptable statements.

COMPAS-80 uses the RST 38H instruction for interrupt check calls, and COMPAS-86 uses the INT 3 instruction. Normally these interrupt locations are available to user programs, since the CP/M and MS-DOS debuggers (DDT, DDT86 and DEBUG) use them when inserting breakpoints.

Section 18

Include files

There may be times when the source text of a program is too large to fit into the buffer of the COMPAS editor. In a situation like this, the program text may be divided into smaller text segments, which are combined only at the time of compilation, through use of the "include file" compiler directive.

The syntax for instructing the compiler to include another source file into the compilation of the program text in memory is as follows:

{$I filename} or (*$I filename*)

where filename is a CP/M file name. The default file type is '.PAS'. Spaces preceding the file name and following it are ignored. It is recommended that no other objects precede or follow the include file compiler directive on that line.

The compiler cannot keep track of nested include files. If an include file includes another file, the reading of the first file is not continued when the second file ends.

Include files are also of use in maintaining libraries of "subroutines" on disk. These "subroutines" may be be included in the compilation of a program whenever they are needed, simply by specifying their file name in an include file compiler option. As an example, assume that the following "subroutine" exists on drive A: under the filename MINMAX.LIB:

```
TYPE number = 0..99;

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

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

A program which uses the max and min functions may then be written as:

```
PROGRAM minmax;
{$I MAXNUM.LIB}
VAR
  x,y: number;
BEGIN
  write('enter two numbers: '); readln(x,y);
  writeln('the largest number is ',max(x,y));
  writeln('the smallest number is ',min(x,y));
END.
```

Since COMPAS Pascal allows free ordering (and possibly several occurrances) of the individual sections in the declaration part of a block, library "subroutines" may declare types and variables of their own or for use by the main program.

During the processing of an include file, COMPAS preserves the state of all compiler flags and registers. The state of the compiler flags and registers is not changed by an include file directive, but if the include file itself contains compiler directives, these directives will only affect the file and not the "main" program.

Section 19

Program chaining

The COMPAS Pascal system provides a chaining mechanism through which one program can execute another. To chain a program, the standard procedure execute or the standard procedure chain is used. The formats of calls to these procedures are:

execute(f) and chain(f)

where f is a file variable of any type, previously assigned to a CP/M disk file using the assign procedure. Provided that the disk file exists, it is loaded into memory and executed.

The execute procedure may be used to execute any other COMPAS Pascal program file. The chain procedure is used only to activate COMPAS Pascal object files, i.e. files created using the OBJECT command of COMPAS Pascal.

More details on the PROGRAM and OBJECT commands may be found in the COMPAS Pascal Operating Manual. Briefly, the difference between files generated using these commands is that the PROGRAM command includes both the run-time package and the program code in the file (thereby creating a machine code program which will run all by itself), whereas the OBJECT command includes only the program code. Thus, a file generated by OBJECT will run only if the run time package is already in memory.

If the disk file referred to by the file variable specified in a call to execute or chain does not exist, an I/O error occurs, unless the {$I-} option switch is in effect, in which case program execution is resumed at the statement following the call. At this point iores contains a non-zero value, which must be examined before further I/O can be performed. An example:

```
VAR
  f: FILE;
BEGIN
  assign(f,'B:PROG3.COM'); {$I-} execute(f) {$I+};
  i:=iores;
  writeln('B:PROG3.COM does not exist');
END.
```

Users of COMPAS-86 under CP/M-86 should replace the 'COM' file type with 'CMD'.

Data can be transferred from the current program to the chained program using one of two methods: Shared global variables or absolute address variables. Data may of course be transferred using a file, but this method is far less convenient.

Using the shared global variable method you must guarantee that the declaration of global variables occurs as the first variable declarations in both programs, and that the declarations are listed in the same order. Furthermore, both programs must be compiled for the same memory size. An example:

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

PROGRAM nextprog;
VAR
  i,j,k: integer;
BEGIN
  writeln(i,' times ',j,' equals ',k);
END.
```

Users of COMPAS-86 under CP/M-86 should replace the 'COM' file type with 'CMD'.

Using the absolute address variable method you would typically define a record type which contains all relevant information fields, and then declare a variable of this type at an absolute address (using the AT clause) in each program.

Note that when you are operating in the direct mode, a call to execute or chain will cause an I/O error to occur. The chain facility of COMPAS only works if used from one machine code program file or object file to active another machine code program file or object file.

Below follows some notes for each specific version of COMPAS.

COMPAS-80

The COMPAS-80 version of execute allows any CP/M command file to be activated. The file is loaded into memory starting at address $100, and executed at address $100, exactly as the CP/M standard specifies.

The chain procedure loads the object file into memory at the origin of the current program (i.e. at the origin specified when the current program was compiled). Thus, for the chain procedure to function properly, the current program and the object file to be chained must be compiled for the same origin address.

A program can determine whether is was invoked from the CP/M command mode or through a call to execute or chain by examining the byte at location $80. A value of $FF (255) indicates that it was invoked from execute or chain, and other values that it was invoked from CP/M. The program shown below demonstrates this feature:

```
PROGRAM invoketest;
VAR
  chainflag: byte AT $80;
BEGIN
  IF chainflag=255 THEN
  writeln('invoked through execute or chain.') ELSE
  writeln('invoked from CP/M.');
END.
```

COMPAS-86

COMPAS-86 users should note that execute and chain do not alter the memory allocation state. In other words, the base addresses and sizes of the code, data and stack segments are not changed - execute and chain only replace the program code in the code segment. As an effect of this, execute cannot be used to initiate "alien" (i.e. non-COMPAS) programs. Furthermore the programmer must guarantee that the "root" program (i.e. the program executed from the operating system) allocates enough memory for the code, data and stack segments to cater for the largest program to be subsequently chained. The PROGRAM command provides parameters which allow the programmer to explicitly specify the minimum sizes of each of these segments.

A program can determine whether is was invoked from the CP/M-86 or MS-DOS command mode or through a call to execute or chain by examining the byte at location $80 in the data segment (CP/M-86) or the code segment (MS-DOS). A value of $FF (255) indicates that it was invoked from execute or chain, and other values that it was invoked from the operating system. The program shown below demonstrates this feature:

```
PROGRAM invoketest;
VAR
  chainflag: byte AT dseg:$80;
BEGIN
  IF chainflag=255 THEN
  writeln('invoked through execute or chain.') ELSE
  writeln('invoked from CP/M.');
END.
```

Users of COMPAS-86 under MS-DOS should change the 'dseg' identifier above to 'cseg'.

Section 20

In-line machine code

COMPAS Pascal has a very useful built-in feature called CODE statements. Such statements can be used to insert machine code instructions (or other kinds of data) into the program code. The syntax of a CODE statement is very simple: It consists of the reserved word CODE followed by one or more constants, variable identifiers, or location counter references, separated by commas.

The constants are either literal constants or constant identifiers, and they must always be of type integer. If a literal constant is specified it generates one byte of code if it is within the range 0..255 ($00..$FF). Otherwise two bytes of code are generated in the standard byte reversed format. A constant identifier always evaluates into two bytes. The use of a variable identifier will generate two bytes (in byte reversed format) giving the memory address of the variable. A location counter reference consists of an asterisk (*), optionally followed by a plus (+) or a minus (-) sign and an integer constant. In the first case, two bytes (in byte reversed format) of code are generated, containing the current location counter value (i.e. the address of the first byte). In the second case, the offset specified is added or subtracted before coding the address.

The examples that follow depend on the version of COMPAS in use.

COMPAS-80

Below is shown an example of the use of a CODE statement to generate a machine code procedure which will convert all characters in its string argument to upper case.

```
PROGRAM testupcase; {$A+}
TYPE
  str = STRING[64];
VAR
  s: str;

PROCEDURE upcase(VAR strg: str);
BEGIN CODE
  $2A,strg,          {       LD   HL,(strg)   }
  $46,               {       LD   B,(HL)      }
  $04,               {       INC  B           }
  $05,               { L1:   DEC  B           }
  $CA,*+20,          {       JP   Z,L2        }
  $23,               {       INC  HL          }
  $7E,               {       LD   A,(HL)      }
  $FE,$61,           {       CP   'a'         }
  $DA,*-9,           {       JP   C,L1        }
  $FE,$7B,           {       CP   'z'+1       }
  $D2,*-14,          {       JP   NC,L1       }
  $D6,$20,           {       SUB  20H         }
  $77,               {       LD   (HL),A      }
  $C3,*-20;          {       JP   L1          }
END;                 { L2:   EQU  $           }
```

```
BEGIN
  write('enter a str[illegible] [illegible]
  upcase(s); wri[illegible]
END.
```

Note that the JP i[illegible] [illegible]ample only to demonstrate the local[illegible] [illegible]ty. For short jumps like the ones sho[illegible] [illegible] is of course more appropriate.

CODE statements may be [illegible] [illegible] quite freely throughout the statement [illegible] CODE statements may use all CPU registers [illegible] the contents of the stack pointer register [illegible] be [illegible] exit as on entry).

COMPAS-86

In COMPAS-86 the use of [illegible] will generate two bytes of code which is t[illegible] [illegible] the variable within its base segment. The [illegible] [illegible] variables (i.e. variables declared in th[illegible] program) is the data segment, which is access[illegible] [illegible] register. The base segment of local variab[illegible] [illegible] declared within the current subprogram) is [illegible] [illegible] this case the [illegible] offset is relat[illegible] [illegible] register, the [illegible] which automatical[illegible] [illegible] segment to be selected. The base segment o[illegible] [illegible] the code segment, [illegible] is accessible th[illegible] [illegible] CODE statements [illegible] not attempt to ac[illegible] [illegible] not declared in [illegible] main program nor in [illegible]

Below is shown an exam[illegible] [illegible] CODE statement [illegible] generate a machine code [illegible] [illegible] convert all characters in its string argum[illegible] upper [illegible]

```
PROGRAM testupcase[illegible]
TYPE
  str = STRING[64];
VAR
  [illegible]: str;

FUNCTION upcase(VAR [illegible]
BEGIN CODE
  [illegible]4,$BE,strg,           [illegible]ES          [illegible]P]
  $26,$8A,$0[illegible],           [illegible]OV          [illegible]]
  $FE,$C1,                         [illegible]NC
  $FE,$C9,                         [illegible]EC
  $74,[illegible]13,               [illegible]Z
  $47,                             [illegible]NC
  $26,$80,$3D,$[illegible]1,       [illegible]MP        [illegible][DI],'a'
  $72,$F5,                         [illegible]B
  $26,[illegible]80,$3[illegible],$7A,  [illegible]   [illegible][DI],'z'
  $77,[illegible]EF,               [illegible]A
  $26,[illegible]80,$2[illegible],$20,  [illegible]   [illegible]DI[illegible] 20H
  $E[illegible],$E9;               [illegible]MP
END;
```

Section 20

In-line machine code

COMPAS Pascal has a very useful built-in feature called CODE statements. Such statements can be used to insert machine code instructions (or other kinds of data) into the program code. The syntax of a CODE statement is very simple: It consists of the reserved word CODE followed by one or more constants, variable identifiers, or location counter references, separated by commas.

The constants are either literal constants or constant identifiers, and they must always be of type integer. If a literal constant is specified it generates one byte of code if it is within the range 0..255 ($00..$FF). Otherwise two bytes of code are generated in the standard byte reversed format. A constant identifier always evaluates into two bytes. The use of a variable identifier will generate two bytes (in byte reversed format) giving the memory address of the variable. A location counter reference consists of an asterisk (*), optionally followed by a plus (+) or a minus (-) sign and an integer constant. In the first case, two bytes (in byte reversed format) of code are generated, containing the current location counter value (i.e. the address of the first byte). In the second case, the offset specified is added or subtracted before coding the address.

The examples that follow depend on the version of COMPAS in use.

COMPAS-80

Below is shown an example of the use of a CODE statement to generate a machine code procedure which will convert all characters in its string argument to upper case.

```
PROGRAM testupcase; {$A+}
TYPE
  str = STRING[64];
VAR
  s: str;

PROCEDURE upcase(VAR strg: str);
BEGIN CODE
  $2A,strg,          {       LD   HL,(strg)   }
  $46,               {       LD   B,(HL)      }
  $04,               {       INC  B           }
  $05,               { L1:   DEC  B           }
  $CA,*+20,          {       JP   Z,L2        }
  $23,               {       INC  HL          }
  $7E,               {       LD   A,(HL)      }
  $FE,$61,           {       CP   'a'         }
  $DA,*-9,           {       JP   C,L1        }
  $FE,$7B,           {       CP   'z'+1       }
  $D2,*-14,          {       JP   NC,L1       }
  $D6,$20,           {       SUB  20H         }
  $77,               {       LD   (HL),A      }
  $C3,*-20;          {       JP   L1          }
END;                 { L2:   EQU  $           }
```

```
BEGIN
  write('enter a string: '); readln(s);
  upcase(s); writeln(s);
END.
```

Note that the JP instruction is used in the example only to demonstrate the location counter reference facility. For short jumps like the ones above, the JR instruction is of course more appropriate.

CODE statements may be mixed with other statements quite freely throughout the statement part of a block, and CODE statements may use all CPU registers (note however that the contents of the stack pointer register (SP) must be the same on exit as on entry).

COMPAS-86

In COMPAS-86 the use of a variable identifier will generate two bytes of code which is the offset address of the variable within its base segment. The base segment of global variables (i.e. variables declared in the main program block) is the data segment, which is accessible through the DS register. The base segment of local variables (i.e. variables declared within the current subprogram) is the stack segment, and in this case the variable offset is relative to the BP (base page) register, the use of which automatically causes the stack segment to be selected. The base segment of typed constants is the code segment, which is accessible through the CS register. CODE statements should not attempt to access variables that are not declared in the main program nor in the current subprogram.

Below is shown an example of the use of a CODE statement to generate a machine code procedure which will convert all characters in its string argument to upper case.

```
PROGRAM testupcase;
TYPE
  str = STRING[64];
VAR
  s: str;

FUNCTION upcase(VAR strg: str);
BEGIN CODE
  $C4,$BE,strg,      {       LES  DI,strg[BP]           }
  $26,$8A,$0D,       {       MOV  CL,ES:[DI]            }
  $FE,$C1,           {       INC  CL                    }
  $FE,$C9,           { L1:   DEC  CL                    }
  $74,$13,           {       JZ   L2                    }
  $47,               {       INC  DI                    }
  $26,$80,$3D,$61,   {       CMP  ES:BYTE PTR [DI],'a'  }
  $72,$F5,           {       JB   L1                    }
  $26,$80,$3D,$7A,   {       CMP  ES:BYTE PTR [DI],'z'  }
  $77,$EF,           {       JA   L1                    }
  $26,$80,$2D,$20,   {       SUB  ES:BYTE PTR [DI],20H  }
  $EB,$E9;           {       JMP  SHORT L1              }
END;                 { L2:                              }
```

```
BEGIN
  write('enter a string: '); readln(s);
  upcase(s); writeln(s);
END.
```

CODE statements must preserve registers BP, SP, CS, DS and ES.

Section 21

System function calls

21.1 COMPAS-80 system function calls

For the purpose of calling CP/M BDOS and BIOS routines, COMPAS-80 introduces two standard procedures, called bdos and bios, and four standard functions, called bdos, bios, bdosb, and biosb. Note that these routines should only be used by the experienced programmer who fully understands their implications.

bdos(f,p) This procedure is used to invoke CP/M BDOS routines. f and p are integer expressions (p and the preceding comma may be omitted if the routine requires no entry parameter). f is loaded into the C register, p (if specified) is loaded into the DE register pair, and a call is placed to address 5 to invoke the BDOS. bdos may also be used as a function, in which case the result, of type integer, is the value returned in the HL register pair.

bios(f,p) This procedure is used to invoke BIOS routines. f and p are integer expressions (p and the preceding comma may be omitted if the routine requires no entry parameter). f gives the number of the routine to call, with 0 corresponding to the WBOOT routine, 1 to the CONST routine, etc. (in other words, the address of the routine is calculated by adding f*3 to the address contained in locations 1 and 2). If p is specified, it is loaded into the BC register pair prior to calling the routine. bios may also be used as a function, in which case the result, of type integer, is the value returned in the HL register pair.

bdosb(f,p) This function is exactly the same as the bdos function, except that the result, still of type integer, is the value returned in the A register.

biosb(f,p) This function is exactly the same as the bios function, except that the result, still of type integer, is the value returned in the A register.

Details on BDOS and BIOS routines are found in the "CP/M Interface Guide" and the "CP/M Alteration Guide" published by Digital Research.

21.2 COMPAS-86 system function calls

In COMPAS-86 all system calls are performed through a single standard procedure called swint (short for software interrupt). The swint procedure takes two arguments. The first argument must be an integer constant within the range 0 through 255, and it specifies the number of the interrupt to execute. The second argument must be a variable of the type regpack shown below:

```
TYPE
  regpack = RECORD
              ax,bx,cx,dx,bp,si,di,ds,es,flags: integer;
            END;
```

Before executing the INT (software interrupt), swint loads the AX, BX, CX, DX, BP, SI, DI, DS and ES registers from the register pack variable (note that the flags are not initialized before the interrupt). On exit the contents of the registers and the flags are stored into the register pack. The program shown below uses the swint procedure to call the CP/M-86 BDOS (INT 224). It passes function code 9 (print string) in CL and the address of the string in DS:DX.

```
PROGRAM test_system_call;
CONST
  bdos_int = 224;
  print_string = 9;
TYPE
  str18 = ARRAY[1..18] OF char;
CONST
  message: str18 = 'System calls work$';
VAR
  regs: RECORD
          ax,bx,cx,dx,bp,si,di,ds,es,flags: integer;
        END;
BEGIN
  regs.cx:=print_string;
  regs.dx:=ofs(message);
  regs.ds:=seg(message);
  swint(bdos_int,regs);
END.
```

For further details on BDOS calls please refer to the "CP/M-86 Operating System System Guide".

If the above program were to run under MS-DOS, an INT 33 should be used to invoke the MS-DOS function call handler, and the function code should be passed in AH:

```
BEGIN
  regs.ax:=swap(print_string);  {Move to AH}
  regs.dx:=off(message);
  regs.ds:=seg(message);
  swint(33,regs);
END.
```

For further details on MS-DOS system function calls please refer to the "MS-DOS Operating System Programmer's Reference Guide".

Section 22

User written I/O drivers

In certain applications it is desirable or even necessary for a program to define its own low-level input and output drivers, i.e. routines which does the basic inputting and outputting of characters from and to an external device. In the COMPAS environment, the following drivers exist (note that the drivers are not available as standard procedures and functions):

```
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;
```

The const routine is called by the keypress function, the conin and conout routines are used by the CON:, TRM:, and KBD: devices, the lstout routine is used by the LST: device, the auxout and auxin routines are used by the AUX: device, and the usrout and usrin routines are used by the USR: device.

The default drivers for the CP/M versions of COMPAS are the BIOS entry points of the CP/M operating system, i.e. const uses CONST, conin uses CONIN, conout uses CONOUT, lstout uses LIST, auxout uses PUNCH, auxin uses READER, usrout uses CONOUT and usrin uses CONIN.

The default drivers for the MS-DOS version of COMPAS-86 are system function 6 (or 11, 8 and 2 in the {$C-} state) for const, conin and conout, system function 5 for lstout, system function 4 for auxout, system function 3 for auxin and system function 6 (or 2 and 8 in the {$C-} state) for usrout and usrin.

The default settings may be changed by the programmer by assigning the address (COMPAS-80) or offset (COMPAS-86) of a driver procedure or a driver function to one of the following standard variables (the 'addr' postfix is used by COMPAS-80 and the 'ofs' postfix is used by COMPAS-86):

csaddr	csofs	address of const function
ciaddr	ciofs	address of conin function
coaddr	coofs	address of conout procedure
loaddr	loofs	address of lstout procedure
aoaddr	aoofs	address of auxout procedure
aiaddr	aiofs	address of auxin function
uoaddr	uoofs	address of usrout procedure
uiaddr	uiofs	address of usrin function

For COMPAS-86 all offsets are relative to the code segment register (CS).

A driver procedure or a driver function must match the definitions given above, i.e. a const driver must be a boolean function, a conin, auxin, or usrin driver must be a char function, and a conout, lstout, auxout, or usrout driver must be a procedure with a char value parameter.

Below is shown a program which defines and activates a new driver for the LST: device. Apart from actually outputting characters to the printer, the driver will keep track of the line and column of the print head. Before each new line, a left margin, consisting of a user defined number of blanks, is printed, and at the bottom of a form, the perforation is automatically skipped. Furthermore, form-feeds are converted into an appropriate number of line-feeds. Single characters are output from the driver by calling the LIST routine (routine number 4) in the BIOS.

```
PROGRAM listdriver; {$A+}
CONST
  pagelength = 72;      { overall page length in lines }
  bottommargin = 6;     { lines to skip at bottom }
  leftmargin = 8;       { left margin }
VAR
  lstlin,lstcol: integer;

PROCEDURE prchr(ch: char);
BEGIN
  bios(4,ord(ch));      { COMPAS-80 only }
END;

PROCEDURE lstout(ch: char);
VAR
  i: integer;
BEGIN
  IF ch>=' ' THEN
  BEGIN
    IF lstcol=0 THEN
    BEGIN
      FOR i:=1 TO leftmargin DO prchr(' ');
      lstcol:=leftmargin;
    END;
    prchr(ch); lstcol:=lstcol+1;
  END ELSE
  IF ch=@13 THEN
  BEGIN
    prchr(@13); lstcol:=0;
  END ELSE
  IF ch=@10 THEN
  BEGIN
    prchr(@10); lstlin:=lstlin+1;
    IF lstlin=pagelength-bottommargin THEN
    BEGIN
      FOR i:=1 TO bottommargin DO prchr(@10);
      lstlin:=0;
    END;
  END ELSE
  IF ch=@12 THEN
  BEGIN
```

```
      FOR i:=lstlin TO pagelength-1 DO prchr(@10);
      lstlin:=0;
    END;
  END;

  BEGIN
    loaddr:=addr(lstout); lstlin:=0; lstcol:=0;
    writeln(lst,'LST DRIVER TEST:');
    writeln(lst,'This should produce three blank lines...');
    write(lst,@10@10@10);
    writeln(lst,'This should produce a form-feed...');
    write(lst,@12);
  END.
```

The above program applies to COMPAS-80 only. For the CP/M-86 version of COMPAS-86 the prchr procedure should be changed to:

```
  PROCEDURE prchr(ch: char);
  VAR
    regs: RECORD
            ax,bx,cx,dx,bp,si,di,ds,es,flags: integer;
          END;
  BEGIN
    regs.cx:=5; regs.dx:=ord(ch); swint(224,regs);
  END;
```

For the MS-DOS version of COMPAS-86 the prchr procedure should be changed to:

```
  PROCEDURE prchr(ch: char);
  VAR
    regs: RECORD
            ax,bx,cx,dx,bp,si,di,ds,es,flags: integer;
          END;
  BEGIN
    regs.ax:=$500; regs.dx:=ord(ch); swint(33,regs);
  END;
```

Furthermore, for both versions of COMPAS-86 the first line of the program should be changed to:

```
    loofs:=ofs(lstout); lstlin:=0; lstcol:=0;
```

At the beginning of the program, the lstout vector is modified to reflect the address of the customized LST: output driver, and the line and column counters are reset. A number of strings are then written to the lst file (which is predefined and preassigned to the LST: logical device), and thereby passed on to the lstout routine one character at a time. Note that user written I/O drivers may under no circumstances call any of the procedures read, readln, write, and writeln. Also note that if user written drivers are used in a set of programs which chain each other, then each program must contain the driver definitions and the code needed to activate them.

Note that user written drivers should always be compiled with the U compiler option off, i.e. in the {$U-} state. COMPAS-80 users should furthermore note that it is recommended to compile drivers in the {$A+} state.

Section 23

Internal data formats

In the following descriptions, the symbol 'addr' denotes the address of the first byte that a variable of the given type occupies. The methods used to allocate memory for variables depend on the version of COMPAS in use.

COMPAS-80

In COMPAS-80 the compiler allocates memory for variables from the top of free memory working downwards. All variables (both main program variables and subprogram variables) are statically allocated, i.e. they reside at the same address throughout the entire execution of the program.

In COMPAS-80 the standard function addr may be used to obtain the address of a variable from a program.

COMPAS-86

Variables declared in the declaration part of the main program block reside in the data segment, which is adressed through the DS register. Typed constants reside in the code segment, which is adressed through the CS register. Variables declared in subprograms (procedures and functions) reside in the stack segment, which is addressed through the SS register. Contrary to COMPAS-80, a COMPAS-86 subprogram allocates memory for its variables when it is called and releases this memory when it returns. Thus, subprogram variables in COMPAS-86 are allocated dynamically, and their absolute address is not known at compile time.

Variables are always contained entirely in their base segment, i.e. the offset address of the last byte occupied by a variable will never exceed $FFFF. For this reason, the size of a single variable can never exceed 64K bytes.

In COMPAS-86 the standard function addr returns a pointer to a variable and the standard functions ofs and seg return the offset address and the segment base address of a variable.

23.1 Basic data types

The basic data types may be grouped into structures (arrays, records, and disk files), but this structuring will not affect their internal formats.

23.1.1 Scalars

Integer subranges, where both bounds are within the range 0..255, booleans, characters, and declared scalars, with less than 256 possible values, are stored using a single byte. This byte gives the ordinal value of the variable.

Integers, integer subranges, where one or both bounds are not within the range 0..255, and declared scalars, with more than 256 possible values, are stored using two bytes. These bytes give a 2's complement 16-bit value. The least significant byte is stored at the lowest memory address.

23.1.2 Reals

The data type real is implemented as 6 bytes giving a floating point value with a 40-bit mantissa and an 8-bit 2's exponent. The exponent is stored in the first byte and the mantissa in the next five bytes which the least significant byte first:

```
addr+0      Exponent.
addr+1      LSB of mantissa.
:
addr+5      MSB of mantissa.
```

The exponent uses binary format with an offset of $80. Hence, an exponent of $84 indicates that the value of the mantissa is to be multiplied by 2^($84-$80) = 2^4 = 16. If the exponent is zero, the floating point value is considered to be zero. The value of the mantissa is obtained from dividing the 40-bit unsigned integer by 2^40. The mantissa is always normalized, i.e. the most significant bit (bit 7 of the fifth byte) should be interpreted as a 1. However, the sign of the mantissa is stored in this bit, a 1 indicating that the number is negative, and a 0 indicating that the number is positive.

The 8087 version of COMPAS-86 uses 8 bytes to store a real value. The format used is the 8087 "long real" format. For a complete description of the 8087 and the "long real" floating point format please refer to the "iAPX 86,88 Users Manual" which is published by Intel Corporation.

23.1.3 Strings

A string type occupies its maximum length plus one bytes of memory. The first byte gives the current length of the string. The following bytes contain the actual characters, with the first character stored at the lowest address. In the table shown below, n denotes the current length of the string, and m denotes the maximum length:

```
addr+0      Current length (n).
addr+1      First character.
addr+2      Second character.
:
addr+n      Last character.
addr+n+1    Unused.
:
addr+m      Unused.
```

23.1.4 Sets

An element in a set occupies one bit, and since the maximum number of elements in a set is 256, a set variable will never occupy more than 32 (256/8) bytes.

If a set contains less than 256 elements, some of the bits are bound to be zero at all times and need therefore not be stored. In terms of memory efficience, the best way to store a set variable of a given type would then be to "cut off" all insignificant bits, and rotate the remaining bits so that the first element of the set would occupy the first bit of the first byte. Such rotate operations are however quite slow, and COMPAS therefore employs a compromise: Only bytes which are statically zero (i.e. bytes of which no bits are used) are not stored. This method of compression is very fast and in most cases as memory efficient as the rotation method.

The number of bytes occupied by a set variable is calculated from (max DIV 8)-(min DIV 8)+1, where min and max are the lower and upper bounds of the base type that set. The memory address of a specific element given by:

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

and the bit address, within the byte at memaddr, is given by:

```
bitaddr = e MOD 8
```

where e denotes the ordinal value of the element.

23.1.5 File interface blocks

Each file variable in a program has an associated file interface block (FIB). A FIB occupies 176 bytes of memory and is divided into two sections: The control section (first 48 bytes), and the sector buffer (last 128 bytes). The control section contains various informations on the disk file or device currently assigned to the file. The sector buffer is used to buffer input and output from and to the disk file.

The table shown below defines the format of a FIB for the CP/M versions of COMPAS:

```
addr+0      Flags byte.
addr+1      File type.
addr+2      Character buffer.
addr+3      Sector buffer pointer.
addr+4      Number of records (LSB).
addr+5      Number of records (MSB).
addr+6      Record length in bytes (LSB).
addr+7      Record length in bytes (MSB).
addr+8      Current record number (LSB).
addr+9      Current record number (MSB).
addr+10     Unused (reserved).
addr+11     Unused (reserved).
addr+12     First byte of CP/M FCB.
:
addr+47     Last byte of CP/M FCB.
```

addr+48 First by or bu
:
addr+175 t byte

Under CP/M the Fil buffer, since
data is transferred dire /from the disk
file. Thus, the len f file is only 48
bytes.

The table shown below de FIB for the MS-DOS
version of COMPAS:

addr+0 Flags byt
addr+1 File ype
addr+2 Character
addr+3 Sector b er.
addr+4 Numb of (LSB)
:
addr+7 N ec of (MSB)
addr+8 Unused (re
:
addr+10 Unused (re .
addr+11 First byte S F
:
addr+47 Last byte S FCB
addr+48 First byte r b
:
addr+175 Last byte bu

Unde DOS the secto file variables of
type t y. Ra m le and untyped file
variables occ 48 and always transferred
directly through om able d or written thus
having all blocking and ng o up to MS-DOS.

e flags byte contains of gs which reflect
e curren status of the it 0 input is allowed,
1 set output is . In ersions of COMPAS
t 2 are used to the fo Bit 2 is set
data has been written to the sector, and bit 3 is set if
the contents of the sector buffer are .

The device byte field specifies whether device or a
file is currently assigned to the file. The follo
values can occur:

0 The console devi
1 e terminal dev (CRT).
2 e keyboard devi (KYBD:).
3 e list device (LST:)
4 The auxiliary dev (AUX:)
5 user device (USR:)
6 Disk file.

The sector buffer pointer the first byte of
the sector buffer. When as a logical device,
only the last three byte FFF gnificance.

23.1.6 Pointers

In COMPAS-80 a pointer consists of two bytes giving a 16-bit memory address, and it is stored in memory using byte reversed format, i.e. the least significant byte is stored first. The value NIL corresponds to a zero word.

In COMPAS-86 pointers occupy two words (four bytes). The word at the lowest address contains the offset and the word at the highest address contains the segment base. NIL corresponds to two words of zeros.

23.2 Data structures

Data structures are built from the basic data types using various structuring methods. Three different structuring methods exist: Arrays, records, and disk files. The structuring of data does not in any way affect the internal formats of the basic data types.

23.2.1 Arrays

The components with the lowest index values are stored at the lowest memory addresses. A multidimensional array is stored with the rightmost dimension increasing first.

23.2.2 Records

The first field of a record is stored at the lowest memory address. If the record contains no variant parts, the length is given by the sum of the lengths of each field. If a record contains a variant, the total number of bytes occupied by the record is given by the length of the fixed part plus the length of largest of its variant parts. Each variant starts at the same memory address.

23.2.3 Disk files

Disk files differ from other structures in that data is not stored in memory but instead in a disk file. A disk file is controlled through a file interface block (FIB) as described in section 23.1.5. In general there are two different types of disk files: Textfiles and random access files.

23.2.3.1 Textfiles

A textfile is subdivided into lines. Each line consists of an arbitrary number of characters ended by a CR/LF sequence. CR (carriage return) has the ASCII value 13, and LF (line feed) has the ASCII value 10. The file is ended by a SUB character (CTRL/Z). SUB has the ASCII value 26.

23.2.3.2 Random access files

A random access file consists of a sequence of records, all of the same length and internal format. To optimize file storage capacity, the records of a file are totally contiguous, and not dependant on sector boundaries.

In the CP/M versions of COMPAS the first four bytes of the first sector of the file contains two values giving the number of records in the file and the length of each record (in bytes):

```
sector 0, byte 0:    Number of records (LSB).
sector 0, byte 1:    Number of records (MSB).
sector 0, byte 2:    Record length (LSB).
sector 0, byte 3:    Record length (MSB).
```

The data of the first record in the file immediately follows these control bytes.

In the MS-DOS version of COMPAS the number of records is calculated from the file length recorded in the directory. If the last record is not filled entirely, it is padded with zeros when it is read. Since the record length is neither recorded in the disk file nor in the directory entry, it is up to the programmer to insure that a file is accessed using the same record length always.

23.3 COMPAS-80 parameter transfers

This section describes the methods and formats used to transfer parameters to and from procedures and functions in COMPAS-80.

Parameters are transferred to procedures and functions using the Z-80 stack. Normally, this is of no interest to the programmer, as the machine code generated by COMPAS Pascal will automatically PUSH parameters onto the stack before a call, and POP them at the beginning of the subprogram. If the programmer however wishes to use EXTERNAL subroutines, then such subroutines must themselves POP the parameters from the stack.

On entry to an EXTERNAL subroutine, the top of the stack always contains the return address (a word). The parameters, if any, are located below the return address (that is, at higher addresses on the stack). Therefore, to access the parameters, the subroutine must first POP off the returns address, then all the parameters, and finally it must restore the return address by PUSHing it back onto the stack.

23.3.1 Variable parameters

If a parameter is a variable (VAR) parameter, one word is transferred on the stack giving the absolute memory address of the first byte occupied by the actual parameter.

23.3.2 Value parameters

In the case of a value parameter, the data transferred on the stack depends upon the type of the parameter.

23.3.2.1 Scalars

All scalars except reals, i.e. integers, booleans, characters, and declared scalars, are transferred on the stack as a word. If the variable occupies only one byte when it is stored, i.e. if it is an integer subrange with both bounds in range 0..255, a boolean, a character, or a declared scalar with less than 256 elements, the most significant byte of the parameter word is zero. Normally, a word is POPped off the stack using an instruction like POP HL.

23.3.2.2 Reals

A real is transferred on the stack using three words. If these words are POPped using the instruction sequence:

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

then L will contain the exponent, H the fifth byte of the mantissa (least significant), E the fourth byte, D the third byte, C the second byte, and B the first byte (most significant).

23.3.2.3 Strings

When a string is at the top of the stack, the byte pointed to by SP contains the length of the string. The bytes at memory addresses SP+1 through SP+n (where n is the length of the string) contain the characters of the string. The following machine code instructions may be used to POP the string at the top of the stack and store it in STRBUF.

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

23.3.2.4 Sets

A set always occupies 32 bytes on the stack (set compression only applies to the loading and storing of sets). The following machine code instructions may be used to POP the set at the top of the stack and store it in SETBUF.

```
LD      DE,
LD      HL,
ADD     HL,
LD      BC,
LDIR
LD      SP,
```

This will store the le... ...f... ...of the set at the lowest address in SETBU...

23.3.2.5 Pointers

A pointer value is trans... ... the ... a word giving the memory address of a dyn... ...able ...ue NIL corresponds to zero.

23.3.2.6 Arrays and rec...

Arrays and records are ... ...lly ...to the stack, even though they are used a... ...para... ...nstead, a word is transferred giving the ... ... th... ...e occupied by the value parameter. It is t... ...esp... ...f the subroutine to POP this word, and ... ... th... ...ddress in a block ...py operation.

? Function results

User written EXTERNAL ... ... mu... ...follow the ...les outlined below for retur... ... t...

...es of scalar types ... ...e...)... ...rned using the HL ...ster pair. If the ... ...e re... ...pressed using one ...only, then L should ... ...chi... ...H should by zero.

... are returned using ... DE... ...ister pairs. B, C, ..., and H should conta... ...mos... ...significant byte in B, ... and L should contai... ...leni...

Strings are returned o... ...f ... ...ng the format described in section 2...

Pointer values are retu... ...t... ...er pair.

23.4 COMPAS-86 parameter... ...s

This sect... ...describes ... ...d a... ...ed to transfe... parameter... to and from ... ... ar... ... COMPAS-86.

23.4.1 Parameters

COMPAS-86 transfers pa... ... ... using the stack (addressed through SS:SP) ... ... the top word on the stack always holds ... ... ac... ...thin the code seg... ment). The parameters ... ... ir... ...below the return address (that is, ...stack), and if the

subprogram is a function, space is reserved for the function result variable below the parameters (i.e. at a higher address).

The first instructions executed by an EXTERNAL subprogram will normally be:

```
PUSH    BP              ;Save base page register
MOV     BP,SP           ;Form pointer to stack frame
SUB     SP,varsize      ;Reserve local workspace
```

where 'varsize' is the size of the local workspace required by the subprogram. To exit, the subprogram should execute:

```
MOV     SP,BP           ;Reset stack pointer
POP     BP              ;Reset base page register
RET     parsize         ;Return and adjust stack
```

where 'parsize' is the number of bytes occupied by the parameters.

Assume that an EXTERNAL function has the following subprogram header (str16 is assumed to be string[16]):

```
FUNCTION stack(VAR i: integer; r: real; s: str16): integer;
```

Furthermore assume that the above entry instructions have been executed, with 50 subtracted from the stack pointer to reserve 50 bytes of local workspace. The stack will then conform to the following map (the map extends from high memory to low memory):

```
001F - 0020     Funtion result variable (1 word).
001D - 001E     Segment base address of i (1 word).
001B - 001C     Offset address of i (1 word).
0016 - 001A     Mantissa of r (5 bytes).
0015 - 0015     Exponent of r (1 byte).
0005 - 0014     Characters of s (16 bytes).
0004 - 0004     Length of s (1 byte).
0002 - 0003     Return address (1 word).
0000 - 0001     Saved base page register (1 word).
FFCE - FFFF     Local workspace (50 bytes).
```

The hex values are offsets from the base page register (BP). Note that function result variable is located above the parameters, and that the parameters are stored in reversed order.

Function result variables in COMPAS-86 use the same format as ordinary variables. So do value parameters, except that integers, booleans, characters, and declared scalars transferred as value parameters always occupy one word, even though their range might have been represented using a single byte. Variable parameters are transferred as a pointer (two words), which points to the first byte occupied by the variable. The formats described in section 23.3 do not apply to COMPAS-86 parameters.

Below is shown an example of an EXTERNAL subroutine (a function) which will convert its character argument to upper case. The function is declared in the Pascal program as follows:

```
FUNCTION upcase(ch: char): char; EXTERNAL 'UPCASE';
```

The function is written ... assem... ...age (for instance ...sing the COMPAS editor), ...verte... ... machine code file using the assembler/link... ...plied ... ...erating system. Its source code is shown ...

```
UPCASE: PUSH    BP                      ;Sa...
        MOV     BP,SP                   ;Ec...  ...ck frame
        MOV     AL,[B...                ;Lo...  ...meter
        CMP     AL,'a'                  ;Le...  ...'
        JB      UPC1                    ;Ye...
        CMP     AL,'z'                  ;Gr...  ... 'z'
        JA      UPC1                    ;Ye...
        SUB     AL,20...                ;Co...  ...pper case
UPC1:   POP     BP                      ;Re...
        RET     3                       ;Ad...  ... and return
```

Note that three bytes a... ...f i... ...n return - 1 word occupied by the ch... parame... ...l by... ...d by the function result variable (which is ...ed i... ...se)

23.4.2 Function results

... COMPAS-86 values of s... ...es ... ...s) are returned ...g the AX register. AH ... ...pe se... ... if only one byte ...equired for the resu... that ... ...unctions must in ...ction execute an OR AX, ...ucti... ...he Z flag according to the value returned ... ...funct... ...move all parameters and the function res... ...ible ... ...stack when returning.

Reals are returned on the ... ...with ... ...ment at the lowest address. To accomplish ... ...mply ... function result ...able remain on the st... ...return... ...removing only the ...eters.

... s are more complica... ...e by ... the function must ...e all parameters and ... ...ze the ... that is move it towards high memory so th... ...pied ... ...r of bytes corresponding to its dynamic le... ...one ..., the stack pointer should point at the b... ...cont... ...string length.

Pointer values are return... ...AX ... in DX, offset in AX). The function must r... ...all pa... and the function result variable from the ... ...n re...

Section 24

Memory management

24.1 COMPAS-80 memory management

During compilation of a program, the memory layout is given by the table below:

0000 - 00FF	CP/M and run time package workspace
0100 - EOFR	Run time package
EOFR - EOFC	COMPAS monitor, editor, and compiler
EOFC - EOFW	COMPAS compiler workspace
EOFW - EOFM	Error messages (if loaded)
EOFM - EOFT	Source text
EOFT - >>>>	Object code
<<<< - MTOP	Symbol table (built by compiler)
<<<< - LTOP	CPU stack
LTOP - FFFF	CP/M operating system

If the error message file (COMPAS.ERM) was not loaded on running COMPAS, the source text starts at EOFW. When the compiler is invoked from a COMPILE or a RUN command, it generates object code working upwards from the end of the source text. The CPU stack works downwards from the logical top of memory, and the symbol table works downwards from MTOP. LTOP is the logical top of memory, and MTOP is set to LTOP less 1K bytes (LTOP-$400).

During the execution of a program, compiled using COMPILE or RUN, the memory layout is given by the table shown below:

0000 - 00FF	CP/M and run time package workspace
0100 - EOFR	Run time package
EOFR - EOFC	COMPAS monitor, editor, and compiler
EOFC - EOFW	COMPAS compiler workspace
EOFW - EOFM	Error messages (if loaded)
EOFM - EOFT	Source text
EOFT - EOFP	Object code
EOFP - BOFH	Unused
BOFH - >>>>	Heap (maintained through hptr)
<<<< - BOFR	Recursion stack (maintained through rptr)
<<<< - BOFS	CPU stack (maintained through sptr)
BOFS - FTOP	Unused
FTOP - LTOP	Program variables
LTOP - FFFF	CP/M operating system

EOFP is the end address of the object code, and hptr (the heap pointer) is set to this address at the beginning of the program (BOFH=EOFP). The area between FTOP and LTOP is used for program variables. FTOP is the top of free memory, and sptr (the CPU stack pointer) is set to this address at the beginning of the program (BOFS=FTOP). The recursion stack is used only by recursive procedures and functions to save copies of their entire workspaces. rptr (the recursion stack pointer) is set to the address contained in the stack pointer less 1K bytes at the beginning of the program (BOFR=BOFS-$400).

During the execution of a program file, the memory layout is given by the table shown below:

0000 - 00FF	CP/M and run time package workspace
0100 - EOFR	Run time package
EOFR - SOFP	Unused (user written machine code routines)
SOFP - EOFP	Object code
EOFP - BOFH	Unused
BOFH - >>>>	Heap (maintained through hptr)
<<<< - BOFR	Recursion stack (maintained through rptr)
<<<< - BOFS	CPU stack (maintained through sptr)
BOFS - FTOP	Unused
FTOP - PTOP	Program variables
PTOP - LTOP	Unused
LTOP - FFFF	CP/M operating system

SOFP is the start address of the object code, corresponding to the <origin> parameter in the PROGRAM and OBJECT commands. PTOP is the address of top of memory for the program, corresponding to the <top> parameter in the PROGRAM and OBJECT commands.

As can be seen from the above memory maps, three stack-like structures exist during the execution of a program: The heap, the CPU stack, and the recursion stack.

The heap is used to store dynamic variables, and is controlled through the new, mark, and release standard procedures. At the beginning of a program, the heap pointer (hptr) is set to the address of the bottom of free memory.

The CPU stack is used to store intermediate results during the evaluation of expressions, and to transfer parameters to procedures and functions. Furthermore, an active FOR statement will occupy one word on the CPU stack. At the beginning of a program, the CPU stack pointer (sptr) is set to the address of the top of free memory.

The recursion stack is used only by recursive procedures and functions (i.e. procedures and functions compiled in the {$A-} mode). On entry to a recursive procedure or function, the subprogram copies its workspace onto the recursion stack, and on exit the entire workspace is restored to its original state. At the beginning of a program, the recursion stack pointer (rptr) is set to point 1K bytes ($400 bytes) below the CPU stack pointer.

To allow the programmer to control the positioning of the heap and the stacks within memory, three predefined variables are introduced:

hptr	The heap pointer.
rptr	The recursion stack pointer.
sptr	The CPU stack pointer.

The type of these variables is integer (note that hptr and rptr may be used in the same context as any other integer variable, whereas sptr may only be used in assignments and expressions).

When assignments are made to these variables, always make sure that they point to addresses within free memory, and that:

```
hptr < rptr < sptr
```

Failing to do so may cause unpredictable (and at in some instances rather catastrophic) results. Needless to say, assignments to the heap and stack pointers may never occur once the stacks or the heap are already in use (therefore, always move such assignments to the very beginning of the main program).

On each call to the new procedure and on entering a recursive procedure or function, the system checks that the heap and the recursion stack has not collided, i.e. that hptr is less than rptr. If this is not the case, an execution error occurs.

Note that no checks are made at any time to insure that the CPU stack does not overflow into the bottom of the recursion stack. For this to happen, a recursive subroutine must call itself some 300-400 times, and that is a rather seldom situation. Should a program however require this sort of nesting, simply execute the following assignment at the beginning of the program block:

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

where maxdepth is the maximum depth of calls to the recursive subprogram(s). An extra 512 bytes (or in that region) are needed as a margin to make room for parameter transfers and intermediate results during the evaluation of expressions.

24.2 COMPAS-86 memory management

When a program created by COMPAS-86 is executed (either as a CMD file from CP/M-86 or a as COM file from MS-DOS or through a RUN command), three segments are allocated for the program: A code segment, a data segment, and a stack segment.

CP/M-86 code segment (CS is the code segment register):

```
CS:0000 - CS:EOFR    Run-time package code.
CS:EOFR - CS:EOFP    Program code.
CS:EOFP - CS:EOFC    Unused.
```

CP/M-86 data segment (DS is the data segment register):

```
DS:0000 - DS:00FF    CP/M-86 base page.
DS:0100 - DS:EOFW    Run-time package workspace.
DS:EOFW - DS:EOFM    Main program block variables.
DS:EOFM - DS:EOFD    Unused.
```

MS-DOS code segment (CS is the code segment register):

```
CS:0000 - CS:00FF    MS-DOS base page.
CS:0100 - CS:EOFR    Run-time package code.
CS:EOFR - CS:EOFP    Program code.
CS:EOFP - CS:EOFC    Unused.
```

MS-DOS data segment (DS is the data segment register):

DS:0000 - DS:EOFW	Run-time package workspace.
DS:EOFW - DS:EOFM	Main program block variables.
DS:EOFM - DS:EOFD	Unused.

The unused areas between (CS:EOFP-CS:EOFC and DS:EOFM-DS:EOFD) are allocated only if a minimum size larger than the required size is specified in the PROGRAM command line used to compile the program. The sizes of the code and data segments never exceed 64K bytes each.

The stack segment is slightly more complicated, as it may be larger than 64K bytes. On entry to the program the stack segment register (SS) and the stack pointer (SP) is loaded so that SS:SP points at the very last byte available in the entire segment. During execution of the program SS is never changed but SP may move downwards until it reaches the bottom of the segment, or 0 (corresponding to 64K bytes of stack) if the stack segment is larger than 64K bytes.

The heap grows from low memory in the stack segment towards the actual stack residing in high memory. Each time a variable is allocated on the heap, the heap pointer (which is a double word variable maintained by the COMPAS-86 run-time system) is moved upwards, and then normalized, so that the offset address is always between $0000 and $000F. Therefore, the maximum size of a single variable that can be allocated on the heap is 65521 bytes (corresponding to $10000 less $000F). The total size of all variables allocated on the heap is however only limited by the amount of memory available.

The heap pointer is available to the programmer through the hptr standard identifier. hptr is a typeless pointer, which is compatible with all pointer types (in the same way as NIL is). Assignments to hptr should be excercised only with great care.

Section 25

Interrupt handling

The COMPAS Pascal run-time package and the code generated by the compiler are fully interruptable. If required, interrupt service routines may be written in COMPAS Pascal.

25.1 COMPAS-80 interrupt handling

Interrupt procedures should always be compiled in the {$A+} mode, they should never have parameters, and they must themselves insure that all registers used are preserved. The latter is done by placing a CODE statement, containing the necessary PUSH instructions, at the very beginning of the procedure, and another CODE statement, containing the corresponding POP instructions, at the very end of the procedure. Furthermore, the last instruction of the ending CODE statement should be an EI instruction ($FB), to enable further interrupts. If daisy chained interrupts are used, the CODE statement may also specify a RETI instruction ($ED,$4D), which will then override the RET instruction generated by the compiler.

The general rules for register usage are, that integer operations use only the AF, BC, DE, and HL registers, other operations may use IX and IY, and real operations use the alternate registers.

An interrupt service procedure should not employ any I/O operations using the standard procedures and functions of COMPAS Pascal, since these routines are not re-entrant. Also note that BDOS calls (and in some instances BIOS calls, depending on the specific CP/M implementation) should not be performed from interrupt handlers, as these routines are not re-entrant.

The programmer may disable and enable interrupts throughout a program using DI and EI instructions generated by CODE statements.

If mode 0 (IM 0) or mode 1 (IM 1) interrupts are employed, it is the responsibilty of the programmer to initialize the restart locations in the base page (note that RST 0 cannot be used, since CP/M uses locations 0 through 7). If mode 2 (IM 2) interrupts are employed, the programmer should generate an initialized jump table (an array of integers) at an absolute address, and initialize the I register through a code statement at the beginning of the program.

The program shown below employs an interrupt service routine. It assumes that a mode 1 interrupt (call to 38H on interrupt) occurs every second. The program applies to COMPAS-80 only.

```
PROGRAM interrupt; {$
TYPE
  timestr = STRING[8]
VAR
  rstjump: byte AT $3
  rstaddr: integer AT
  seconds,minutes,hou                    ger;

PROCEDURE inctime;
BEGIN
  CODE $F5,$E5,$D5,$C
  seconds:=succ(secon
  IF seconds=60 THEN
  BEGIN
    seconds:=0; minut                    (min
    IF minutes=60 THE
    BEGIN
      minutes:=0; hou                    (hou
      IF hours=24 THE                    0;
    END;
  END;
  CODE $                 ,$D1,$E1,$F
END;

FUNCTION time: timest
VAR
    timestr;
  N
  ]:=chr(8);
  ]:=chr(hour DIV
      (hour MOD

  [4]:=chr(minutes D
  [5]:=chr(minutes M                     );
      :='
        hr(seconds D                     );
      chr(seconds M
  time                 ;
END;

  IN
   stjump:=$C3; rsted                    inct
   rite('enter hours,                    an
  readln(hours,minute                    );
  CODE $ED,$56,$FB;


  writeln('The time
  :
  CODE $F3;
END.
```

Since the errupt ser ne only us integer operations, it need only AF and HL registers. The CODE statement at the e specifies both the PUPs required and an EI i r further int pts. At the beginning of the a or is p at location $38, which jumps te vice proc The current time is ther i tion ($ED $5 3

executed to select interrupt mode 1, and interrupts are enabled using the EI instruction ($FB). At the end of the program, further interrupts are disabled using the DI instruction ($F3).

25.2 COMPAS-86 interrupt handling

COMPAS-86 interrupt routines must manually preserve registers AX, BX, CX, DX, SI, DI, ES and FLAGS. This is done by placing the following CODE statement as the first statement of the procedure:

```
CODE $50,$53,$51,$52,$56,$57,$06,$9C,$FB;
```

The last byte ($FB) is an STI instruction which enables further interrupts - it may or may not be required. The following CODE statement must be the last statement in the procedure:

```
CODE $9D,$07,$5F,$5E,$5A,$59,$5B,$58,$CF;
```

The last byte ($CF) is an IRET instruction which overrides the RET instruction generated by the compiler.

An interrupt service procedure should not employ any I/O operations using the standard procedures and functions of COMPAS-86, or any operations on real variables, since these sections of the COMPAS-86 run-time package are not re-entrant. CP/M-86 users should also note that no BDOS calls may be performed, since the BDOS is not re-entrant.

It is up to the programmer to initialize the interrupt vector used to invoke the interrupt service procedure. Assuming the declaration:

```
VAR
  int_10_vec: ^integer AT $0000:$0040;
```

then the statement:

```
int_10_vec:=ptr(cseg,ofs(int_10_handler));
```

initializes the INT 10H vector to point at a procedure called 'int_10_handler'.

Section 26

Differences between COMPAS and Standard Pascal

The COMPAS Pascal language adheres closely to the Jensen & Wirth definition of Standard Pascal as contained in the "User Manual and Report". Some minor differences do however exist, and these are described below. Note that this section does not describe the extensions offered by COMPAS Pascal.

Dynamic variables

Dynamic variables and pointers are implemented using the new, mark and release procedures rather than the new and dispose procedures suggested by Standard Pascal. This was done partly to maintain compatibility with other compilers (e.g. UCSD Pascal), and partly because it is far more efficient in terms of execution speed and support code needed. Furthermore, the new procedure will not accept record variant specifications (the allocate standard procedure is easily used to circumvent this restriction).

Get and put I/O

The standard procedures get and put are not implemented. Instead, the read and write procedures have been extended to handle all I/O needs. There are three reasons for this: Firstly the read and write procedures are far more versatile and easier understood that get and put, secondly read and write allow for faster execution of I/O operations, and thirdly variable space overhead is reduced, since file buffer variables are not required.

GOTO statements

A GOTO statement may not leave the current block, since this is generally bad programming practice.

Page procedure

The page procedure is not implemented, since the CP/M and MS-DOS operating systems do not define a form-feed character.

Procedural parameters

Procedures and functions may not be used as parameters. This feature was omitted for two reasons: Firstly the standard definition of procedural parameters is quite weak and almost impossible to implement (to our knowledge, no existing implementation of Pascal on a microcomputer follows the standard in this aspect), and secondly procedural parameters are rarely ever used.

Packed variables

The reserved word PACKED has no specific meaning in COMPAS Pascal (although it is allowed). Instead, packing occurs automatically whenever possible. Furthermore, the pack and unpack procedures are not implemented, since they are not needed.

Appendix A

Summary of standard procedures and functions

This appendix lists all standard procedures and functions available in COMPAS Pascal. The following symbols are used:

<type>	Denotes any type.
<string>	Denotes any string type.
<file>	Denotes any file type.
<scalar>	Denotes any scalar type.
<pointer>	Denotes any pointer type.

Note that some procedures and functions will accept variable parameters of any type. In these cases, no type is specified for that parameter. The following abbreviations may be shown in the right margin:

80	Indicates COMPAS-80 only.
86	Indicates COMPAS-86 only.
MS	Indicates MS-DOS version of COMPAS-86 only.

Input/Output routines

The procedures described below use a non-standard syntax for their parameter lists.

```
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);
```

File handling routines

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

```
FUNC    longlen (VAR f: FILE OF <type>): real;                    MS
        longlen (VAR f: FILE): real;                              MS
FUNC    longpos (VAR f: FILE OF <type>): real;                    MS
        longpos (VAR f: FILE): real;                              MS
PROC   longseek (VAR f: FILE OF <type>; pos: real);               MS
       longseek (VAR f: FILE; pos: real);                         MS
FUNC   position (VAR f: FILE OF <type>): integer;
       position (VAR f: FILE): integer;
PROC     rename (VAR f: <file>; name: <string>);
PROC      reset (VAR f: <file>);
          reset (VAR f: FILE; reclen: integer);                   MS
PROC    rewrite (VAR f: <file>);
        rewrite (VAR f: FILE; reclen: integer);                   MS
PROC       seek (VAR f: FILE OF <type>; pos: integer);
           seek (VAR f: FILE; pos: integer);
```

Arithmetic routines

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

Scalar routines

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

Transfer routines

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

String routines

Note that the str procedure uses a non-standard syntax for its numeric parameter.

```
FUNC     concat (s1,s2,...,sn: <string>): <string>;
FUNC       copy (s: <string>; pos,len: integer): <string>;
PROC     delete (VAR s: <string>; pos,len: integer);
PROC     insert (s: <string>; VAR d: <string>; pos: integer);
FUNC        len (s: <string>): integer;
FUNC        pos (pattern,source: <string>): integer;
```

```
PROC      str (i: integer; VAR s: <string>);
          str (r: real; VAR s: <string>);
PROC      val (s: <string>; VAR i,p: integer);
          val (s: <string>; VAR r: real; VAR p: integer);
```

Pointer related routines

```
FUNC     addr (VAR variable): <pointer>;                    86
PROC allocate (VAR p: <pointer>; size: integer);
PROC     mark (VAR p: <pointer>);
FUNC memavail : integer;
PROC      new (VAR p: <pointer>);
FUNC      ord (p: <pointer>): integer;
FUNC      ptr (i: integer): <pointer>;                       80
          ptr (seg,ofs: integer): <pointer>;                 86
PROC  release (VAR p: <pointer>);
```

Miscellaneous routines

```
FUNC     addr (VAR variable): integer;                       80
         addr (<procedure identifier>): integer;             80
         addr (<function identifier>): integer;              80
PROC     bdos (func,param: integer);                         80
PROC     bios (func,param: integer);                         80
FUNC     bdos (func,param: integer): integer;                80
FUNC     bios (func,param: integer): integer;                80
FUNC    bdosb (func,param: integer): byte;                   80
FUNC    biosb (func,param: integer): byte;                   80
FUNC     cseg : integer;                                     86
FUNC     dseg : integer;                                     86
PROC     fill (VAR dest; length: integer; data: byte);
         fill (VAR dest; length: integer; data: char);
PROC   gotoxy (x,y: integer);
FUNC      hi (i: integer): integer;
FUNC   iores : boolean;
FUNC keypress : boolean;
FUNC      lo (i: integer): integer;
PROC    move (VAR source,dest; length: integer);
FUNC     ofs (VAR variable): integer;                        86
         ofs (<procedure identifier>): integer;              86
         ofs (<function identifier>): integer;               86
FUNC   pwrten (exp: integer): real;
FUNC  random (range: integer): integer;
       random : real;
PROC randomize ;
FUNC     seg (VAR variable): integer;                        86
FUNC    size (VAR variable): integer;
         size (<type identifier>): integer;
FUNC     sseg : integer;                                     86
FUNC     swap (i: integer): integer;
PROC    swint (<constant byte>; VAR regpack);                86
```

Appendix B

Summary of operators

The table shown below gives a summay of all operators available in COMPAS Pascal. The operators are grouped according to their precedences, and the operators of the highest precedence are listed first.

Operator	Operation	Type of operand(s)	Type of result
+ unary	sign identity	integer, real	as operand
- unary	sign inversion	integer, real	as operand
NOT	negation	integer, boolean	as operand
*	multiplication	integer, real	integer, real
	set intersection	any set type	as operand
/	division	integer, real	real
DIV	integer division	integer	integer
MOD	modulus	integer	integer
AND	logical AND	integer, boolean	as operand
SHL	shift left	integer	integer
SHR	shift right	integer	integer
+	addition	integer, real	integer, real
	concatenation	string	string
	set union	any set type	as operand
-	subtraction	integer, real	integer, real
	set difference	any set type	as operand
OR	logical OR	integer, boolean	as operand
EXOR	logical EXOR	integer, boolean	as operand
=	equality	any scalar type	boolean
	equality	string	boolean
	equality	any set type	boolean
	equality	any pointer type	boolean
<>	inequality	any scalar type	boolean
	inequality	string	boolean
	inequality	any set type	boolean
	inequality	any pointer type	boolean
>=	greater or equal	any scalar type	boolean
	greater or equal	string	boolean
	set inclusion	any set type	boolean
<=	less or equal	any scalar type	boolean
	less or equal	string	boolean
	set inclusion	any set type	boolean
>	greater than	any scalar type	boolean
	greater than	string	boolean
<	less than	any scalar type	boolean
	less than	string	boolean
IN	set membership	see below	boolean

The first operand of the IN operator may be of any scalar type, and the second operand must be a set of that type.

Appendix C

Summary of compiler directives

Compiler directives are written as comments and may occur whenever ordinary comments are allowed. A compiler directive list is introduced by a $ character immediately following the opening comment bracket. The generalized format of a comment containing a compiler directive list is:

```
{$<directive list> <any comment>}
```

or:

```
(*$<directive list> <any comment>*)
```

The directive list is a sequence of instructions separated by commas. Each instruction begins with a letter designating the directive. If the letter refers to a compiler option, it must be followed by a plus (+) if the option is to be activated or a minus (-) if the option is to be passivated. If the letter refers to a compiler register, it must be followed by a digit (see W below), and if the letter refers to a special facility of the compiler, it must be followed by a string of characters depending on that facility (see I below). Some examples of compiler directives:

```
{$R-}     {$S+,I+,A-}     (*$W6,B-*)     {$I B:MAX.LIB}
```

The following directives are available:

A COMPAS-80 only. When activated, this compiler option instructs the compiler to generate absolute code for procedures and functions, i.e. code that does not allow for recursive calls, but executes faster and takes up less memory than its recursive equivalent. For further details, please refer to section 15.6.

Default setting is off (A-).

B When this compiler option is active at the beginning of the program block, the CON: device will be used as the default I/O device, i.e. the device assigned to the standard files input and output. Otherwise the TRM: device is used. For further details, please refer to sections 13.3.3 and 16.1.

Default setting is on (B+).

C When this compiler option is active at the beginning of the program block, output to the console can be paused using CTRL/S, and CTRL/C can be used to interrupt a program during console I/O. For further details, please refer to section 17.1.

Default setting is on (C+).

I A compiler option which, when activated, instructs the compiler to include run time tests that check all I/O operations to insure that no I/O errors occurred. For further details, please refer to section 13.5. If the I directive is followed by a filename, it indicates that the file should be included in the source text. For further details on include files, please refer to section 18. Note that when the include file directive is used, no further directives may be specified in that directive list.

Default setting is on (I+).

K COMPAS-86 only. When activated, this compiler option instructs the compiler to generate calls to a stack overflow check routine before calls to procedures and functions. For further details, please refer to section 15.8.

Default setting is on (K+).

R When activated, this compiler option instructs the compiler to include run time tests that check all array indexing operations to insure that the index lies within the specified bounds, and all assignments to variables of scalar and subrange types to make certain that the assigned values lie within the allowable range. For further details, please refer to sections 7.4 and 9.1.

Default setting is on (R+).

S COMPAS-80 only. A compiler option which, when activated, instructs the compiler to optimize array indexing operations with respect to execution speed instead of code size. For further details, please refer to section 9.1.

Default setting is off (S-).

U When this compiler option is activated, the compiler will generate calls to an interrupt check routine before the code of each statement. During run-time, such statements may be interrupted by pressing CTRL/C at the keyboard. For further details, please refer to section 17.2.

Default setting is off (U-).

V A compiler option which in its passive state instructs the compiler to allow string variables of any type as actual variable parameters, even if their maximum length does not agree with that of the formal parameter. For further details, please refer to section 15.6.

Default setting is on (V+).

W COMPAS-80 only. A digit n (0<=n<=9) must immediately follow the W character, and it defines the maximum nesting level of WITH statements. The W register must be set before the declaration part of the block it is to affect. For further details, please refer to section 10.2.

Default setting is 4 (W4).

Y A letter between A and P or a zero digit must immediately follow the Y character, and it defines the drive on which to look for overlay files during run-time. A letter denotes that specific drive, and a zero denotes the currently logged drive. For further details, please refer to section 15.9.

Default setting is 0 (Y0).

Appendix D

ASCII character table

DEC	HX	CHAR	DEC	HX	CHAR	DEC	HX	CHAR	DEC	HX	CHAR
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	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Appendix E

COMPAS syntax

The syntax of the COMPAS Pascal language is presented here using BNF (Backus-Naur Form) formalism. The following symbols are meta-symbols belonging to the BNF formalism, and not symbols of the COMPAS Pascal language:

::=	Means "is defined as".
|	Means "or".
{...}	Denotes possible repetition of the enclosed symbols zero or more times.

The symbol <character> denotes any printable character, i.e. a character with an ASCII value between $20 and $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 identifer>
{ , <file identifer> }

<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> ::= <variable identifier> | <constant> |
       <constant> : <constant> | DSEG : <constant> |
       CSEG : <constant>

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

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

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

<variable identifier> ::= <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 identifer> (<actual parameter>
{ , <actual parameter> })

<function identifier> ::= <identifer>

<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 error messages

An I/O error occurs whenever an error condition arises during an input or output operation. If I/O checking is enabled, an I/O error will cause the program to terminate, displaying an I/O error message:

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

where nn is the I/O error number (in hex) and aaaa is the relative address of the error (with respect to the start address of the program code). If I/O error checking is disabled, an I/O error will not cause the program to halt. Instead the error number is stored so that it can be examined by the program through the iores standard function.

The following I/O errors can occur (note that the error numbers are in hex).

01 Record length mismatch. This error is reported by reset if you try to combine a file variable with a disk file of improper format, or more specifically when the record lengths of the the file variable and the disk file does not agree.

02 File does not exist. This error is reported by reset, erase, rename, execute, or chain if the file name assigned to the file variable does not specify an existing file.

03 Directory is full. This error is reported by rewrite if you try to create a new file when there is no more room in the disk directory.

04 File disappeared. This error is reported by close if the file you are trying to close have disappeared from the disk directory, for instance due to the user changing a disk when he is not supposed to.

05 File not open for input. This error is reported by read (from a textfile or a defined file) or readln if you try read from a file which has not been reset or rewritten, or, in the case of a textfile, from a file prepared using rewrite, or from the LST: logical device.

06 File not open for output. This error is reported by write (to a textfile or a defined file) or writeln if you try write to a file which has not been reset or rewritten, or, in the case of a textfile, to a file prepared using reset, or to the KBD: logical device.

07 Unexpected end-of-file. This error is reported by read or readln (from a textfile) if the physical end-of-file is reached on a disk file before the end-of-file character.

08 No room on disk. This error is reported by write or writeln (to a textfile) if there is no more room on the disk.

09 Error in numeric format. This error is reported by read or readln with a textfile argument when the string read for a numeric value is not of a proper numeric format.

0A Read beyond end-of-file. This error is reported by read (from a defined file) or by blockread if you try read from a file when you are already at the end of the file.

0B File length overflow. This error is reported by write (to a defined file) if you try to store more than 65535 records in a file.

0C Disk read error. This error is reported by read or write (on a defined file) or blockread if the routine is unable to read the next sector from the file. In the case of read, or write it indicates that there is something wrong in the file itself, whereas for blockread it may also indicate that you are trying to read beyond the end of the file.

0D Disk write error. This error is reported by read or write (on a defined file), flush, or blockwrite if the routine is unable to expand the file due to the disk being full.

0E Seek beyond end-of-file. This error is reported by seek if you try seek beyond the end of the file.

0F File not open. This error is reported by blockread or blockwrite if the file referred to has not been reset or rewritten.

10 Operation not allowed on a logical device. This error is reported by erase, rename, execute, or chain (on a textfile), if the file is assigned to a logical device

11 Not allowed in direct mode. This error is reported by execute or chain if you try to invoke another program when you are operating in the direct mode (i.e. when the program was run from a RUN command).

12 Illegal assign parameter. This error is returned by the assign standard procedure if the program attempts to assign a file or a device name to one of the standard files (input, output, con, trm, kbd, lst, aux or usr).

Appendix G

Execution error messages

An execution error indicates a fatal error condition in the system. Execution errors always cause the program to halt and display an error message:

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

where nn is the execution error number. For COMPAS-80, aaaa is the realtive address of the error with respect to the start address of the program code. For COMPAS-86, aaaa is the true offset address of the error.

The following execution errors can occur (note that the error numbers are in hex).

01 String length error. This error is reported by a string concatenation operation (the plus operator or the concat procedure) if the resulting string is longer than 255 characters, or by a string-to-character conversion if the length of the string is not 1.

02 Invalid string index. This error is reported by copy, delete or insert if the index expression is not within 1..255.

03 Floating point overflow.

04 Division by zero attempted.

05 sqrt argument error. The argument passed to the sqrt function was negative.

06 ln argument error. The argument passed to the ln function was zero or negative.

07 Out of integer range. The real value passed to trunc or round was not within the integer range (-32768..32767).

08 Index out of range. The index expression at an array subscription was out of range.

09 Scalar or subrange out of range. The value assigned to a scalar or a subrange variable was out of range.

0A Heap/stack collision. This error occurs at a call to new or allocate if there is not enough memory available on the heap, or at a call to a subprogram if there is not enough memory available on the stack. Note that in COMPAS-86 stack overflow checks are performed only on subprogram calls compiled in the {$K+} state.

0B Overlay file not found. This error is reported on entry to an overlay subroutine if the overlay file that contains its code cannot be found. It is also reported if the overlay file is in some way corrupted.

Appendix H

Compiler error messages

01 '.' expected.

02 BEGIN expected.

03 Invalid function result type. Valid types are all scalar types, string types, and pointer types.

04 Duplicate identifier. This identifier has already been used within the current block.

05 Absolute variables not allowed in records. The AT clause may not be used in a record.

06 Type identifier expected.

07 Files may only be variable parameters.

08 Unknown or invalid type.

09 END expected.

10 Set base type out of range. The base type of a set must be a scalar with no more than 256 possible values or a subrange where both bounds are within the range 0..255.

11 File components may not be files. FILE OF FILE constructs are not allowed.

12 Invalid string length. The maximum length of a string must be within the range 1..255.

13 Invalid subrange base type. Valid base types are all scalar types, except real.

14 '..' expected.

15 Type mismatch in subrange bounds. The type of the lower bound does not agree with the type of the upper bound.

16 Lower bound greater than upper bound. The ordinal value of the upper bound must be greater than or equal to the ordinal value of the lower bound.

17 Unknown or invalid simple type.

18 Simple type expected. Simple types are all scalar types, except real.

19 Unknown pointer type in type definitions. A preceding pointer type definition contains a reference to an unknown type identifier.

20 Undefined label in statement part. The preceding statement part contains a reference to an undefined label.

21 Invalid GOTO in statement part. A GOTO statement in the preceding statement part references a label within a FOR loop from outside the FOR loop.

22 Label already defined. This label already marks a statement.

23 THEN expected.

24 DO expected.

25 Unknown or invalid variable identifier.

26 Variable type is not a simple type. The control variable of a FOR loop must be of a simple type.

27 Type mismatch in FOR statement expressions. One (or both) of the expressions in a FOR statement does not agree with the type of the control variable.

28 TO or DOWNTO expected.

29 Constant and CASE selector type does not agree.

30 END or OTHERWISE expected.

31 Unknown label.

32 Too many nested WITH statements. Use the W compiler directive to increase the maximum number of nested WITH statements.

33 Record variable expected.

34 Unknown or invalid variable.

35 Illegal assignment.

36 Type mismatch in assignment or parameter list. The type of the variable and the expression in an assignment does not agree, or the type of the actual and the formal parameter in a procedure call or a function call does not agree.

37 Expression is not of type integer.

38 Expression is not of type boolean.

39 Expression type is not a simple type. Simple types are all scalar types, except real.

40 Expression is not of type string.

41 Type mismatch in expression. The operands in an expression are not of compatible types.

42 Operand type(s) does not agree with operator.

43 Structured variables are not allowed here. Arrays (except character arrays), records, and files are not allowed here.

44 Type mismatch in set. The types of the elements or ranges in a set does not agree.

45 Unknown identifier or syntax error in expression.

46 Constants are not allowed here.

47 Expression type does not agree with index type.

48 Unknown or invalid field identifier.

49 Unknown or invalid constant.

50 Integer constant expected.

51 Integer or real constant expected.

52 String constant not properly terminated. String constants must be fully contained on a single line.

53 Error in integer constant. The integer constant contains one or more syntax errors or it is not within the integer range, i.e. -32768..32767. Whole real numbers should be followed by a decimal point and a zero, e.g. 14764552.0. For the definition of an integer constant, please refer to section 2.2.

54 Error in real constant. For the definition of a real constant, please refer to section 2.2.

55 Illegal character in identifier. Valid characters are 'A' to 'Z', 'a' to 'z', and underscore '_'.

56 '[' expected.

57 ']' expected.

58 ':' expected.

59 ';' expected.

60 Unknown identifier or syntax error in statement.

61 ',' expected.

62 '(' expected.

63 ')' expected.

64 '=' expected.

65 ':=' expected.

66 OF expected.

67 Unexpected end of source text. This error indicates that your program cannot end in the way it does. Possibly you have more BEGINs than you do ENDs.

68 No such file. The include file specified does not exist.

69 Buffer overflow. This error indicates that there is not enough memory to compile the program. Note that this error may occur even though free memory seems to exist - this storage is however occupied by the stack and the symbol table during compilation. Break your source text into smaller segments and use include files.

70 Memory overflow. You are trying to allocate more storage for variables than available.

71 Variables of this type cannot be input.

72 Variables of this type cannot be output.

73 Textfile expected.

74 File variable expected.

75 Textfiles are not allowed here.

76 Untyped files are not allowed here.

77 String constant expected.

78 String constant length does not agree with type.

79 Invalid ordering of fields in record constant.

80 Type mismatch in structured constant.

81 Constant out of range.

82 Files and pointers are not allowed here.

83 Invalid use of retype facility. The retype facility only applies to simple types, i.e. all scalar types except real.

84 Integer or real expression expected.

85 String variable expected.

86 Textfiles and untyped files are not allowed here.

87 Untyped file expected.

88 Pointer variable expected.

89 Integer or real variable expected.

90 Integer variable expected.

91 Reserved word. These may not be used as identifiers.

92 Label not within current block. GOTO statements may not leave the current block.

93 Procedure or function not properly defined. The subprogram has been FORWARDed, but the body never occurred.

94 Error in CODE statement.

95 Illegal use of AT specification. Only one identifier may appear before the colon in a variable declaration which employs an AT specification.

96 Overlays not supported in direct mode. Overlays may not be used in programs compiled with a COMPILE or a RUN command.

97 Overlays cannot be forwarded. The FORWARD specification may not be used in connection with overlays.

98 PROCEDURE or FUNCTION expected.

99 Can not create overlay file. The compiler can not create a new overlay file due to the disk directory being full or the diskette being write protected.

100 Error in EXTERNAL file. This error is reported by the CP/M-86 version of COMPAS-86 if an EXTERNAL file is not of a proper 'CMD' file format.