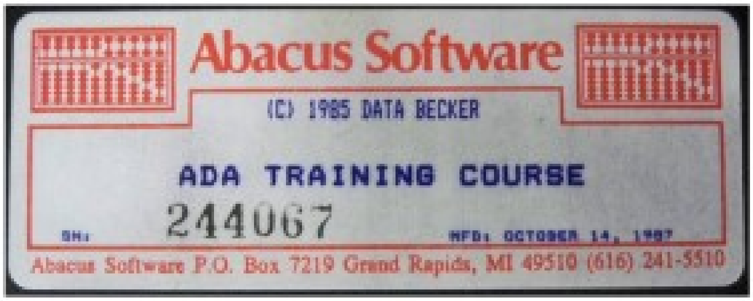
ADA TRAINING COURSE

Learn the NEW Structured "Language of the Future" includes Compiler and Editor

A DATA BECKER Product from Abacus



Abacus Software

(C) 1985 DATA BECKER

ADA TRAINING COURSE

SN: 244067

MFD: OCTOBER 14, 1987

Abacus Software P.O. Box 7219 Grand Rapids, MI 49510 (616) 241-5510

ADA TRAINING COURSE

Structured Language of the Future for the Commodore-64

By: V. Sasse

A DATA BECKER PRODUCT

Published By:

You Can Count On Abacus Software

Second Printing,	February 1985
Printed in U.S.A.	Translated by Greg Dykema, Lauren Thompson
Copyright (C) 1985	Abacus Software, Inc.
	P.O. Box 7211
	Grand Rapids, MI 49510
Copyright (C) 1984	Data Becker, GmbH
	Merowingerstr. 30
	4000 Dusseldorf, W. Germany

ISBN # 0-916439-15-1

Table of Contents

The Ada Training Course

Introduction

What is Ada?

Ada is one of a new generation of programming languages. It gets its name from from the Countess Ada Lovelace, the daughter of the English author Lord Byron. The Countess lived in England during the 18th century and is the first person to determine how a calculating machine, developed by Charles Babbage, could be programmed. She is considered to be the first programmer in the world.

Until now ADA was only known in the higher levels of data processing (on mainframes), largely because there was no ADA compiler for the computers which "people like you and I" own. ADA is the language of the future and one should at least become acquainted with it. This is exactly what the Ada training course allows you to do. Part of the training course is an Ada compiler which compiles a subset of this language into machine language.

This training course includes:

1) The program diskette
2) The text book

The program diskette:

There are five main programs on the diskette.

A) The **EDITOR**

You write your ADA programs with the editor. The editor also includes complete disk management capabilities. You can easily send commands to your disk drive, display the directory listing or send it to a printer, etc.

B) The **syntax-checker** for your ADA programs

This program tests your ADA programs for syntactic correctness. If you are not sure what the syntax of the ADA programming language is, refer to the text book.

C) The **semantics-checker** and code generator for your ADA programs.

The program checks your ADA programs for syntactic correctness and a creates a very fast assembler program.

D) The **assembler**

The assembler can be used together with the ADA compiler, or may be used separately from it. You can use it to assemble the assembler programs produced by the ADA compiler or your own machine language programs.

E) The **disassembler**

With the disassembler you can convert op-codes in the computer's memory back into the assembler mnemonics. This allows you to analyze machine language programs.

The text book:

The operation of the programs are described in detail in this text. You will receive an introduction to the ADA language including examples, problems, and the corresponding solutions.

This is a true training course with which you will acquaint yourself with data processing fundamentals. The knowledge acquired can also be transferable to other programming languages.

You will not only learn the basics of a new language, ADA, but also how programming-language compilers work, what methods they use, and what they in principle can and cannot do.

You will certainly become better acquainted with your computer and even enter into the world of machine language programming. The most important utilities necessary to do so are included in this training course.

1. The Editor

The editor is the program that you will be using most as a user of the ADA Training Course. You will write your programs with it and carry out compilation from it.

The editor offers a number of other capabilities as well. It can:

- save your programs to diskette
- load your programs from diskette
- print your programs
- inform you of memory space remaining
- display the disk directory
- scratch files on the diskette
- transmit commands to the disk drive

Let's try out the various functions of the editor so that we may acquaint ourselves with the most important program in the ADA training course.

Turn on your computer, disk drive and (if present) printer. Insert the distribution diskette into the disk drive (By distribution diskette we mean the disk which you received with your Ada training course). Load the editor with the command:

```
LOAD"EDITOR",8,1  <RETURN>
```

After about a minute the program will be completely loaded.

Remove the distribution diskette from the disk drive and replace it with a new formatted disk or one containing data you no longer need. Press <RETURN>

The **START** menu appears on the screen.

The editor has a total of three menus:

Menu - **START**
- **COMMANDS**
- **WRITE/EDIT**

These three menus can be reached with the keys <@>, <*>, and <↑> (up arrow) respectively. These keys are operational when the computer has finished the task you have instructed it to perform.

The options in the **START** menu allow you to select a function so that all of the keys repeat and to select the colors for the characters, the screen border, and the background. Press the <f1> and then select your preference in color combinations. Now press the <*> key.

The **COMMAND** menu appears on the screen. From this menu you can access all of the general commands listed below.

- save your programs on diskette
- load your programs from diskette
- print your programs
- inform you of memory space remaining
- display the disk directory
- scratch files on the diskette
- transmit commands to the disk drive

The <↑> (up arrow) key brings us to the WRITE/EDIT menu. This menu allows you to create a new program or edit an existing one.

We will now discuss the menus individually. Each menu option can be selected by pressing the appropriate key. Press the <@> to return to the START menu.

1.1. START menu

By pressing the function key <f1> we can make all of the keys repeat, meaning that holding a key down will cause that character to be entered repeatedly. The <f3> key allows us to turn this feature off.

The function key <f2> (obtained by pressing <SHIFT> and <f1> at the same time) changes the color of the screen border. Simply press <f2> until you get the color which is most pleasing to you. <f4> changes the color of the background in a similar manner and <f6> changes the character color.

1.2. WRITE/EDIT menu

This menu is accessed with <↑> (up arrow). Press the <↑> (up arrow) key to enter the WRITE/EDIT menu.

Now we will learn how we can create and edit a program with the editor. We will go through each command of the editor and see what effects they have.

The operation of a key pressed in error can be undone by immediately pressing the <RETURN> key. Wherever a particularly damaging error may occur, the computer will ask to make sure that the function is really intended.

Pressing the <**f2**> key prepares the editor for entering a new program. A message confirming the selection of the option "Input" appears on the screen followed by three lines containing other information with which we need not concern ourselves with at the moment. The number "00010" appears in the fifth line followed by a reverse question mark. This is the first line number of our text. These line numbers are irrelevant to the Ada program! They are used only so that the user can quickly find a given program line. This Ada Training Course makes references to the line numbers to make corrections easier. Behind the line number is a field with a question mark in the color which we chose for the characters. This is the **CURSOR**. It indicates the place at which the next input will appear. Please enter the sentence:

"This is supposed to be an Ada program."

If you made a mistake while typing, you can erase the last character or with repeated use, the last characters, on the line by using the **DEL** (delete) key.

The editor will accept only those characters which make sense in an Ada program. It works as a filter, filtering out nonsensical input. If the cursor fails to move and no character is entered, you have pressed an illegal key or key combination.

We move to the next input line by pressing the <RETURN> key. The line number "00020" appears on the screen. Assuming we do not want to enter any more lines, we can exit the input mode by pressing the <RETURN> key again. Please press the <RETURN> key now. The computer confirms the exit from the INPUT mode with the message:

```
****            Input done          ****
```

And the cursor disappears.

In place of our sentence we could have entered an Ada program consisting of a set of instructions. The sentence "This is supposed to be an Ada program." will suffice in order to acquaint ourselves with the editor.

Please press the <f1> key now. The <f1> key returns us to the start of the text and informs us of this with the message:

```
****            Beginning           ****
```

Please press the <f7> now. With the help of the <f7> key we can we can view our program line by line. Please press the <f7> again. At the end of the text the computer responds with the message:

```
****            End                  ****
```

Press the <↑> (up arrow) key now. The <↑> (up arrow) key will return you to the WRITE/EDIT menu.

You can go immediately to the end of the program by pressing **<f3>**. Function key **<f5>** allows you to step backwards through the program. Feel free to try out each of the keys and become accustomed to their use. If you forget any of the keys meanings, you can see the WRITE/EDIT menu again with <↑> (up arrow).

If you want to add additional lines to the program, press the **<f2>** key. Press the **<f2>** key now. The editor gives you the next possible line number and allows you to enter additional lines. Enter the line "Sentence 2", press <RETURN> and enter the line "Sentence 3" , press <RETURN>. To exit the input mode press <RETURN>. The computer leaves the input mode when you press <RETURN> over an empty line.

If you want to edit an already existing line, do the following: List the line to be changed using **<f5>** or **<f7>** and then press **<f6>**. The cursor will appear in reverse. You can move through the line with the cursor keys and change characters by simply writing over them. When you are done editing the line, press <RETURN>. Try changing the number "3" in line 00030 to "4". To do so press the **<f5>** key to list line 00030 then press the **<f6>** key. Check to see if line 0030 is changed by entering the menu mode <↑> (up arrow), then list the complete text by pressing the **<f7>** key four times.

The editor also allows us to insert lines between existing lines. If, for instance, we want to insert a line between the second and third lines, we list line two (0020) and then press **<f8>**. Do so now. The computer confirms this by printing the line number "00021" and the cursor reappears. We enter the line as we did before under the "Input" command. Something like "This line follows line 00020". We terminate the input with <RETURN> and the line number "00022" appears . We again exit this mode by pressing <RETURN> before typing anything else on the line. Up to nine lines can be inserted since the editor numbers the lines by ten. The editor saves the inserted lines differently and inserts them into our program at the end of the command. It gives us the appropriate messages on the screen.

If we would like to have all of the lines numbered by ten again, we simply press <f4>. Using this option we can insert as many lines as desired. Do so now, then list the renumbered text using the <f7> key.

There are two possibilities for erasing lines: with the "pound" key or the left-arrow key. You can erase individual lines with the pound key and entire blocks with the left arrow. Pressing the pound key erases the line which you last listed with <f5> or <f7>. List line 00030 and then delete it using the "pound key". Check to be sure line 00030 was deleted, then renumber the text using the <f4> key.

Now press the left-arrow key to delete a range of lines. After pressing the left-arrow key you will be asked "from line :", you must then enter a line number and press <RETURN>. Enter 20 and press <RETURN>. The question "to line :" is answered in the same way. Enter 30 and press <RETURN>. List the text to be sure the lines "from line 20" "to line 30" were deleted.

With this we complete our discussion of the WRITE/EDIT menu and all that remains is the COMMAND menu.

1.3. COMMAND menu:

In order to follow the examples in this section, you should have at least one line of text in memory.

Press <*> and we enter the **COMMAND** menu.

First we would like to ask the computer how much space is left in memory so that we know how much we can add to our program. The function key <**f7**> does this for us. Please press the <**f7**> key now to view the available memory. The computer responds with the message (free memory may be different):

```
****   20045  Characters free   ****
```

If you are certain that you no longer need the contents of the diskette in the disk drive, you will want to format this diskette and become acquainted with the function "Send command to disk drive." Press <**f6**> and the following message will appear:

```
****   Command to disk          ****
****   Command  ?
```

We enter: "n:data,01" and press the <RETURN> key, thereby sending the command to format a disk to the disk drive. The name "data" and the identification code "01" are placed on the disk. The disk drive requires some time to execute this command. If you made an error while typing the command, the drive will usually respond with a "SYNTAX ERROR." You simply correct the command in this case. In general, you can transmit any command found in chapter 4 of the disk drive manual to the drive in this manner.

We need only press the **<f1>** key in order to save our Ada program to the diskette. You will be asked for a name. After entering this name, press <RETURN> and the disk drive will proceed to save the program.

Press **<f5>** to make sure that the file was saved correctly. You will receive information concerning the name of the disk, its identification number and the DOS version the disk was formatted under.

A program can be loaded back into the editor with the **<f3>** key. You are asked for the name of the program, and after this input and subsequently pressing <RETURN> the command will be executed.

Programs on the diskette can be removed with the command "Delete file." After pressing **<f8>** you are asked for the name of the file which is to be erased. After entering the name of the file and pressing <RETURN> the file will be deleted from the disk.

The **<f4>** key is used to print a program on the printer. You **must** enter a comment which appears as a header for the listing. Leading spaces may be entered by pressing <SHIFT> and the space bar together.

With the function key **<f2>** we start the Ada compiler. The program currently in memory is compiled. You will be asked if you want to first save the program because the memory will be cleared after the compiler has done the first part of its work. See the following section "Using the compiler" for more information.

This concludes the section on the Ada editor. It would be a good idea to practice using the editor, so you may become accustomed to using it.

2. Using the compiler

After you have written a program with the editor, enter the **COMMAND** menu and press the **<f2>** key. You will be asked if this key has been pressed in error. If you enter a character other than "y" and press <RETURN>, the command will be terminated. The compiler will then ask if you would like to include a **TRACE** function on the compiled program. This will cause the compiled program to print the sequence of line numbers from the Ada source code as it executes. Unless you are having problems with a program the usual answer is **NO**. When you press <RETURN>, you will then be asked if you want to save the program. This can be skipped by entering a character other than "y" and pressing <RETURN>. If you press only <RETURN>, you must give a name for your program. If a program by the same name is found on the disk, it will be erased and the new program saved in it's place.

After saving the program (or skipping this option), the compiler begins with the **lexical analysis**. When this is done, the computer will require the distribution diskette in order to load the **syntactic analysis** program. After this program is loaded you must reinsert your data disk.

After the the syntactic analysis the computer again requires the distribution diskette in order to load the **semantic analysis** program. Then your data disk is again needed in order to continue the compilation.

If an error was encountered during the syntactic analysis of your program, you can load the editor directly in order to correct the program.

The program for semantic analysis creates an assembly language program with the name "ADA.SRC".

If the semantic analyzer discovers an error, you can reload the editor at the end of the semantic analysis.

If the program compiled successfully, load the assembler in order to assemble the ADA.SRC program. A third option is to end the program and load ADA.SRC into the computer.

Load the assembler with:

```
LOAD "ASSEMBLER",8,1
```

Load the assembly language program with:

```
LOAD "ADA.SRC",8
```

If you load the assembler, you will be asked for the name of the program which you would like to assemble. If you press <RETURN>, ADA.SRC will be assembled. You can also enter the name of a different assembly language program, however.

The finished machine language program receives the extension ".OBJ". A machine language program created from an Ada program can be loaded with:

```
LOAD "ADA.OBJ",8,1
```

and your own programs with:

```
LOAD "name.OBJ",8,1
```

"ADA.OBJ" can be started with RUN, your own programs with SYS start-address.

Ada on the Commodore 64 is very disk intensive so it is a good idea to re-format your data diskettes at regular intervals. Remember that reformatting a disk removes all information from it, so don't do this to disks which still contain information you might want. It has happened to me that the disk drive can no longer properly read the files on the diskette because the read/write head is misaligned. This can produce quite peculiar compilation results!

{This page left blank intentionally}

3. About the Ada training course

In order to better understand the contents of this training course, I would first like to present the goals I had when I wrote this training course package.

The goal of the Ada training course is to acquaint you with a structured programming language. The language at hand is the very new language Ada. You will become acquainted with fundamental structures which are present in most modern programming languages. The training course is not tied exclusively to Ada; you will also learn things about your CBM 64.

A large portion of the training course is concerned with programming in assembly language and the use of operating system routines. This is because the assembler serves as an interface between the "higher level" programing language Ada and the 6510 microprocessor which forms the "heart" of your computer.

The operation of the assembler routines will give you valuable insight into how you can write your own assembly language programs. The assembler and disassembler programs included allow you to begin immediately with this.

Thus the training course operates on two planes, the plane of the high-level programming language and the plane of machine language. One seeks to develop a language with which one can program easily and elegantly, with whose help programs (sets of instructions) can be written which can be understood by others.

Our sets of instructions must be carried out by microprocessors. In the construction of the microprocessors, the goal is to manipulate memory locations and process their contents as quickly as possible.

The goal of our course can be abstractly formulated as follows: Proceeding from an initial condition of our memory locations, their contents are to be conveyed to a desired condition with the help of the program. The microprocessor should carry this out as quickly as possible. Microprocessors, however, understand only their own machine language and since their are many different microprocessors, their are also many different machine languages. If one then wants to program in a higher-level language, some connection between the high-level language and the machine language of the microprocessor is required. The compiler represents this connection. The compiler is a program which is designed to work with the microprocessor. It translates our program into the machine language of the processor in question. A compiler generally consists of several programs which convert the the program to be translated into machine language step by step.

If one wants to write a compiler which will run on more than one computer (microprocessor), a procedure like the following will help him to do so: One translates the high-level program into the assembly language of a "fictitious" microprocessor. A microprocessor which does not exist, but which has the fundamental properties of real microprocessors. From this language it is no great step to the machine language of the individual microprocessor. For each new microprocessor, "only" this portion of the compiler need be rewritten to make the entire compiler work. One can use the fact that the machine languages of the various microprocessor are related. For our compiler, this interface is for the 6510 microprocessor.

You can learn how the programs which create this assembler code operate in the sections concerning the operation of the compiler. You will learn how programs can be in the situation to analyze other complex programs. A few sentences about the individual analysis steps:

The program for lexical analysis checks all Ada programs for lexical correctness. The syntactic analysis works with a grammar which represents a large portion of the Ada definition. It can check almost all programs used in normal work for syntactic correctness.

Ada is a very young language which has been changed in parts many times over the last years. So far, complete versions of Ada are only available for mainframe computers, and as far as I know, only test versions are available. This is attributable to the complexity of the language. With this background it is quite nice to be able to check many programs for syntactic correctness. During the development of the syntax checker I have checked a variety of programs from Ada books for syntactic correctness and with many programs, which probably could not have been tested before, discovered discrepancies.

There are stringent limitations on the subsequent semantic check and the creation of the assembly language programs. This is first of all due to the fact that the syntactic checking of Ada programs is very time-consuming, and second because of the structure of the compiler itself. Since all of the compiler programs cannot fit into memory at the same time, the programs run in sequence. This means that information required by programs following one another must be saved on disk. Each program works only with the information of the previous program. Memory can be saved this way but in order to create a piece of an assembly language program, all of the necessary information must be present. But because memory space must be saved, of the excellent capabilities of Ada, only those which can be made to work under these limitations are included.

I hope that I have attained my goal of offering you an elegant option for creating short, fast assembly language programs.

In addition it is possible to combine several programs with each other and to address Ada programs from BASIC.

4. Writing our first Ada program

After you have loaded the editor and started it, select the menu WRITE/EDIT and enter the following program (the underline character "_" as in ADA_1 is obtained by pressing the Commodore key and the P key at the same time.):

```
procedure ADA_1 is
 --
 --  The data objects that our program
 --  uses will be declared here.
 --
begin
 --
 --  The executable statements
 --  of our program appear here.
 --
 null ;
 --
end ADA_1 ;
```

This is the smallest possible Ada program (without the comments). It has the name "ADA_1". The name appears at the start of the program and at the end. It is not absolutely necessary at the end, but when it appears it must be the same name as at the beginning. A discrepancy will be seen as an error by the compiler. It is a good idea to include the name at the end of the program as well as at the beginning so that you always know the program is at an end. This is not necessary for small programs, but we want to learn how to write large programs clearly and use the capabilties which Ada offers.

The program begins at the keyword **"procedure"**. Keywords are words which have a predetermined meaning in Ada. Keywords are part of the language. The keyword **"procedure"** means that a program to be executed begins at this spot.

After **"procedure"** comes the name of the program, which we may choose freely. The name chosen may not be an Ada keyword because keywords are predetermined and therefore protected. Names of data objects may and should be longer than the maximum of two letters to which we are accustomed to in BASIC. Therefore it is possible to give sensible and suggestive names to data objects. This increases the readability of a program. Names (also called identifiers) can be a maximum of 250 characters long. In practice, you will probably not reach this limit, but it is given for the sake of completeness. Identifiers must begin with a letter and may contain letters, digits and the character "_". This is the underline character (**Commodore P**) used to separate words in the identifier, making it easier to read.

The computer does not distinguish between upper and lower case, "procedure" or "PROCEDURE" or "PROCedure" all have the same meaning as far as the compiler is concerned. To easily distinguish Ada keywords we will write the keywords in boldfaced lower case.

A separating character must be between keywords and identifiers. A space is a separator; we will learn others later.

Comments in Ada are denoted by two consecutive minus signs (--, hyphens, dashes). Comments make use of the entire line. Ada instructions can not follow a comment in a line.

Examples of valid and invalid comments:

```
-- This is a valid comment
-- Comments can extend over several
-- lines, or may be empty
--

- - This is not a valid comment!
```

No spaces may be between the minus signs.

Back to our example: After the name of our program follows the keyword **is**. We will later put the data objects (variables, constants, ...) which our program will use between this word and the keyword **begin**.

After **begin** follows the part of the program which contains the executable instructions. These are instructions which tell the computer to perform a specific action. The instruction I have chosen here is the instruction **null**. This instruction serves as a place holder in Ada. We will use it wherever an Ada instruction must be, but we do not yet know which instruction we will chose, or when the computer should execute the **null** instruction.

Ada instructions are ended with a semicolon.

The program ends with the keyword **end**. The name of the program follows, terminated by a semicolon.

You probably never thought that one could say so much about a program which doesn't do anything.

If you wish, you can compile this program. It will create a machine language program which contains no instructions, but it will allow you to test the operation of the compiler.

5. Text output

In this section we will learn how to output text and receive our first exposure to the compiler.

One property of Ada is that the possibility exists to break complex problems down into smaller ones. One writes a program which we call in ADA a **package** for each smaller program and then assembles the total solution out of these partial ones. The advantages of this are clear: We write a program for a specific problem, test it out, and save it. For the moment all that interests us is what data must be passed to the program in order to get certain information back. The commands the computer executes to do this are unimportant for us. Furthermore, we need no longer give any consideration to this partial solution; we only use it. Several persons can work on one large program without one being dependent on any other. The programmers simply agree on the functions of the program parts and the manner in which they are accessed and then they start programming independently.

The program which is responsible for the input and output of data is also such a **package**. It is agreed upon in the language as a so-called "standard package," which means that it is part of the equipment of the compiler. This **package** must be rewritten for each computer, so that the same commands perform the same operations in each implementation of the language. The agreed-upon scope of this package is too large for the CBM 64. I have included the capabilities for you which were most important to me, without making this **package** consume the entire memory of the computer.

Don't worry--you will find everything you need for writing useful programs.

If we want to use the input/output **package**, we must inform the compiler of this. This is done with the commands:

```
with TEXT_IO; use TEXT_IO;
```

The **package** for text input/output is called "TEXT_IO". These commands must be at the start of the program. The **with** command must appear before the keyword **procedure**. The **use** command can also appear at other places.

Let us begin with text output.

The predetermined receiver of output is the screen. If we want the sentence "Hello, this works quite well!" on the screen, we must write the following program:

```
with TEXT_IO; use TEXT_IO;
--
procedure OUTPUT_1 is
--
begin
--
      PUT ( "Hello, this works quite well!" );
--
end OUTPUT_1;
```

The command which we use to output strings (text) is called PUT. PUT is not a keyword, but it has a specified meaning in connection with the previous **with** and **use** commands. This command will be the same in all implementations of Ada.

We place the output text in parentheses and enclose the actual string in quotation marks. This is called a "string literal" in Ada. There also "character literals" in Ada. These are individual characters and are enclosed in apostrophes in Ada. For example, 'A' or 'h' or '_' are character literals.

The output of characters is done in the same manner as the output of string literals.

```
PUT ( 'b' ) ;
```

The PUT command outputs the character or string at the current position of the cursor. At the end of the command the cursor is placed at the next output position.

If you want to output a character or string literal and set the cursor at the start of the next line, use the following command:

```
                    PUT_LINE ( "......" );
or                  PUT_LINE ( '.' );
```

The periods stand for a character or string literal. To place the cursor one line lower, end the output on the current line with this command:

```
NEW_LINE ;
```

To skip lines or to print blank lines, use the command in following form: The number must a natural number (integer greater than zero). (NUMBER-1) lines will then be printed.

Example:

```
NEW_LINE ( 4 ) ;
```

This ends the output on the current line and prints three blank lines.

In order to position a cursor within a line, use this command:

```
SET_COL ( column );
```

The cursor is placed at the column specified in place of the word "column". Example: We want to place the cursor at column 35.

```
SET_COL ( 35 ) ;
```

The output can be sent to the printer instead of the screen with this command:

```
SET_OUTPUT ( printer ) ;
```

Now all output will be sent to the printer. Output can be redirected to the screen with:

```
SET_OUTPUT ( screen ) ;
```

Make sure that you do not direct the output to the same device twice in a row. The compiler can first recognize this error at execution time, after it has compiled the entire program, at the time you have started the machine language program.

Exercise:

We now have learned enough in order to write a small Ada program. Solve the following task, one possible solution can be found in the "Solutions" section. The solution has the name "OUTPUT 2". The given solution does not mean that our problem can only be solved in this manner or that it is necessarily the best solution. It means only that it is the solution which I have worked out for you. Write a program which outputs the following sentences in the given form:

1. Output the string literal "This is our first task."

2. Move to the next output line.

3. Output the string "This string starts in the second line and extends into line number "

4. Output the character '3' directly behind the previous string.

5. Output 5 blank lines.

6. Output the line "Now everything goes to the printer."

7. Direct the following output to the printer.

8. Output "Is the printer working?"

9. Switch the output back to the screen.

10. Output the character 'E' in column 35.

{This page left blank intentionally}

6. Screen control

The following capabilities are not part of the Ada standard but our computer places them at our disposal, so it seems a shame not to use them.

These capabilities are very interesting because a compiled program executes these functions very quickly. They are so fast in part because neither the video interface chip (the device responsible for the screen output) nor your television, to say nothing of your eyes, can follow an output stream which consists only of these functions. Programs with these functions are executed with maximum speed, very quickly indeed in comparison with BASIC.

Our Ada program uses the CBM-64 operating system routines for these functions. I have written a **package** for the CBM-64 called CBM_64 so that you can use these functions. Please include this package at the start of all your programs in the future.

```
with CBM_64 ; use CBM_64;
```

The following functions are available for screen control:

```
SET_ROW ( line );
```

This command is related to the command SET_COL. The command SET_ROW is not included in the Ada standard, however. SET_ROW sets the cursor to the given line. You can choose a line number between 1 and 24.

Example: Set the cursor to line 15.

```
SET_ROW ( 15 );
```

We can clear the screen with the command:

```
SCREEN_CLR ;
```

We set the cursor in the upper left-hand corner of the screen with:

```
CURSOR_HOME ;
```

The following commands allow us to change the color of the screen border, background, and characters. For the sake of simplicity I will enumerate all of the possibilities.

Note the way each color is designated, otherwise the compiler will respond with **"Unrecognized color"**.

Selecting the type (character) color:

```
SET_TYPE ( black );
SET_TYPE ( white );
SET_TYPE ( red );
SET_TYPE ( green );
SET_TYPE ( blue );
SET_TYPE ( purple );
SET_TYPE ( yellow );
SET_TYPE ( cyan );
```

Selecting the border color:

```
                    SET_BORDER ( black );
                    SET_BORDER ( white );
                    SET_BORDER ( cyan );
                    SET_BORDER ( red );
                    SET_BORDER ( purple );
                    SET_BORDER ( green );
                    SET_BORDER ( blue );
                    SET_BORDER ( yellow );
                    SET_BORDER ( orange );
                    SET_BORDER ( brown );
                    SET_BORDER ( light_red );
                    SET_BORDER ( grey_1 );
                    SET_BORDER ( grey_2 );
                    SET_BORDER ( light_green );
                    SET_BORDER ( light_blue );
                    SET_BORDER ( grey_3 );
```

Selecting the background color:

```
SET_BKGND ( black );
SET_BKGND ( white );
SET_BKGND ( cyan );
SET_BKGND ( red );
SET_BKGND ( purple );
SET_BKGND ( green );
SET_BKGND ( blue );
SET_BKGND ( yellow );
SET_BKGND ( orange );
SET_BKGND ( brown );
SET_BKGND ( light_red );
SET_BKGND ( grey_1 );
SET_BKGND ( grey_2 );
SET_BKGND ( light_green );
SET_BKGND ( light_blue );
SET_BKGND ( grey_3 );
```

Exercise:

Write a program which does the following:

1. Clears the screen.
2. Set the border color to "grey_2"
3. Set the background color to "white"
4. Set the cursor in line 10, column 20.
5. Output "L 10, C 20" in black type.
6. Set the cursor in the upper left-hand corner of the screen.

The solution for this task can be found under the name "screen control."

Try this out and see how fast your CBM-64 can be. You will be surprised.

{this page left blank intentionally}

7. Data objects

By data objects we mean objects in our program to which we can assign values. We distinguish between constants and variables.

Data objects are assigned a specific type. They then assume the characteristics of that type.

8.1 Types:

We will be working with three different data types in our Ada training course:

Type : INTEGER

This data type represents all whole numbers in the range -32768 to +32767.

Type : FLOAT

This data type represents all floating-point numbers in the range +/-1.701411183E+38 and +/-2.93873588E-39.

Type : STRING

Objects of this type can be assigned strings of characters up to 80 characters long.

Constants:

Constants are objects which assume a value at their declaration. This value cannot be changed during the course of the program. The compiler checks for this and refuses a value assignment.

Constants of any of the previously named types can be declared.

Examples:

Constants of type INTEGER:

```
INTEGER_1                  : constant INTEGER := 15;
START_QUANTITY_CARS        : constant INTEGER := 3576;
ACCELERATION_FACTOR        : constant INTEGER := -15;
```

The declaration is constructed as follows:

At the beginning of the line stands the name or identifier of the data object. Then follows the colon and the keyword **constant**. Next comes the type identifier and, preceded by a colon and equals sign, the value our constant is to have.

With the declaration of any data object it is possible to define several objects at once.

Example:

```
OBJECT_1, OBJECT_2, OBJECT_3 : constant INTEGER := -1000;
```

The data objects are separated from each other by a comma.

Constants of type FLOAT:

```
FLOAT_123                 : constant FLOAT := -3.98E-22;
PI                        : constant FLOAT := 3.1415;
SHERRY, LIQUEUR, WHISKEY : constant FLOAT := 35;
```

Constants of type string:

```
STR                       : constant STRING := "pearl";
ADDRESS                   : constant STRING := "Grand Rapids";
SNTNCE_START, SNTNCE_END: constant STRING := "Hi there!";
```

For constants of type string, the amount of memory taken up by the constant depends on its length.

When we declare data objects which will have only one value throughout the program and are never to be reassigned, we declare these as constants.

If we want to perform calculations and assign a value to a data object during the course of the program, we need variables. These objects can be defined and redefined during the execution of a program. They can also be assigned an initial value. If we are starting our program, the variables are assigned values.

Variables of type INTEGER:

```
SUSAN                     : INTEGER := 22;
PETER_MEIER               : INTEGER := 18;
SAM, JOE, ANN             : INTEGER := 172;
```

Variables of type FLOAT:

```
PRICE                      : FLOAT ;
SUM                        : FLOAT := 2E+10;
PROJ, EXIST, MOVE          : FLOAT := 0;
```

Variables of type string:

```
FIRST_NAME_1               : STRING:= "Mike";
FIRST_NAME_2, FIRST_NAME_3 :  STRING:= "Harold";
LAST_NAME                  : string;
```

Now we know how data objects are declared.

The types which you have become acquainted with are predefined in Ada. They belong to the language standard. An Ada compiler for larger computers would have additional data types available. Also missing in this training course is the ability to form user-defined types from those already existing. Due to memory limitations these were not implmented in the Ada Training Course compiler, but you may run the lexical analysis and syntactical analysis on programs using the entire ADA language. You will not be able to run the semantic analysis or compile these programs on the CBM-64.

Exercise

Write the declaration portion of a program to work with the following data objects:

1. A whole number constant with the name WHOLE and the value -1.

2. A floating-point number with the name FLOATP and the value 0.3E-6.

3. A string constant with the name STR and the value "Hi there!"

4. An integer variable with the name INT_VAR.

5. Two floating-point variables with the names PRICE_CHEESE and PRICE_SAUSAGE and the initial values 0 and 0.

6. A string variable with the name HOUSENAME and your last name as the initial value.

The model solution has the name "DECLARATIONS".

{this page left blank intentionally}

8. Data input and output

The input and output of data is handled by the computer-dependent **package** CBM_64. Don't forget to specify this package before the declaration portion.

We use the following command to read data objects from the keyboard:

```
GET ( data object );
```

You replace the words "data object" with the name of a variable to which you want to assign a new value. The program then stops at the point in the program where this command is found and requests input from the keyboard with a question mark (?). Be sure that you enter a value of the appropriate type.

In order to display the values of data objects on the screen, use the following command:

```
PUT ( data object );
```

You are already familiar with the PUT command from text output.

It is good style to make inputs immediately visible with an output (echo the input) in order to provide a check. Also, do not forget to comment your programs so that you can understand them later. Take a look at the following example:

Example:

```
with TEXT_IO; use TEXT_IO;
with CBM_64; use CBM_64;
--
-- Example for the input and output of data.
-- The name and year or birth of the user
-- will be entered and printed.
--
procedure DATA_IN_DATA_OUT is
--
-- Declaration of the string variable for
-- the name of the user.
--
   NAME : STRING;
--
-- Declaration of the integer variables for
-- the birth year of the user.
--
   BIRTH_YEAR : INTEGER;
--
begin
--
   SCREEN_CLR;
--
   SET_COL (5);
--
  PUT ( "   Please enter your name:");
--
   SET_ROW (8); SET_COL (4);
--
   GET ( NAME );
--
```

```
      NEW_LINE; PUT ( "   Your name is :" );
      PUT ( NAME );
 --
      NEW_LINE (3);
      PUT ( "   Please enter the year or your birth:");
 --
      NEW_LINE; SET_COL (4); get ( BIRTH_YEAR );
      NEW_LINE (2);
      PUT_LINE ( "   You were born in the year :" );
      PUT ( BIRTH_YEAR );
 --
 end DATA_IN_DATA_OUT ;
```

As you have noticed, more than one instruction may be placed on a line in Ada. The semicolon separates the instructions from each other. You should make sure that the program does not become too cluttered. In some cases it is even advisable to place instructions which belong together on a single line.

Exercise

The solution has the program name "output 2".

Write a program which asks for your body weight and then outputs this again.

{this page left blank intentionally}

9. Value assignment

Ada is a strongly-typed language, which means that a data object of a certain type may only be assigned values which are compatible with that type. A variable of type integer may not be assigned a floating-point value because the floating point value would first have to be converted to an integer before the assignment. The individual types are logically distinguished. Not only variables but also operations such as addition, multiplication, etc. are logically distinguished by type.

Nevertheless, it is often necessary to assign the value of an integer variable to a floating-point variable, for example. The value of the integer variable must first be converted to a floating-point value. You can convert the values of integer variables to floating-point variables and vice versa. How this is done will be explained later.

First we want to see what a value assignment in Ada looks like.

Examples:

```
SAM := TOM + 2;
```

We assume that both SAM and TOM are data objects of the same type. If this were not the case, the compiler would tell us so. On the left side of the value assignment stands the data object whose value will be changed. Then follows a colon and the equals sign. This character combination can be read as "receives the value of." The ":=" tells the compiler that this instruction is an assignment. On the right side is an arithmetic expression. At the end follows the semicolon which signals the end of the instruction. In our example the data object SAM is assigned the value of the data object TOM, plus 2.

Exponents in Ada are designated by the string "**".

Conversion:

Floating-point values can be converted to integer values with the following construction:

```
INTEGER ( floating-point value )
```

The value of the data object used in place of "floating-point value" is converted to type integer.

The opposite conversion of an integer value to a floating-point value is accomplished with:

```
FLOAT ( integer value )
```

Example:

ERIKA is a data object of type integer and JOHN is a data object of type FLOAT. ERIKA is to be assigned the value of JOHN, therefore JOHN must be converted to an integer:

```
ERIKA := INTEGER ( JOHN );
```

This covers the value assignment of types float and integer, but what about the value assignments of type string?

Value assignment with data type string:

Here things are done a bit differently than usual. Data objects of type string have a length of 80 characters. The compiler reserves this space in memory. It can therefore access the individual strings very quickly because it does not have to search for memory.

We can represent every string variable in the following form: NAME (1..80). This means that we can access the places 1 through 80 for this variable. If, for example, we want to fill positions 1 to 10 with a certain string, we do it as follows:

```
PETER ( 1..10 ) := "abcdefghij";
```

At the start of the program execution, all string variables are filled with binary nulls so that they are considered to be empty. If you want to return a string to its initial condition, enter the following command:

```
PETER ( 1..80 ) := "";
```

The following procedure is used to assign string variables with the values of other string variables:

```
PETER ( 5..15 ) := EDWARD ( 3..13 );
```

Here the string variable PETER at position 5 is assigned the value of the string variable EDWARD at position 3. Eleven characters are copied.

If the receiving string is shorter than the sender, the copied string will be truncated. If the receiving string is longer than that sent, the string copied is padded with blanks.

Exercise

The solution has the name "VALUE ASSIGNMENT".

Write a program which perform the following task:

A merchant sells diskettes and wants a program that will write a bill giving him the total of the purchase, including sales tax. The name of the customer must also be on the bill in order to keep the finances straight. Below is a sample bill. Try to use everything you learned in this section.

Sample bill:

```
Sam Harris                          bought on 10/05/84

10   diskettes at a price of            $ 29.95

4% sales tax                               1.20
```

10. Functions:

A number of numeric functions which support the operating system have been implemented in the CBM 64 package.

The operand, the variable or constant, used for these functions can be of type float or integer. The syntactic form is the same for all of the functions. Simply replace "function name" with the actual name of the function.

Command construction:

```
VARIABLE_1 := function name (VARIABLE_2 );
```

Examples:

```
SQUARE_ROOT := SQR ( TOM );
SQUARE_ROOT := SQR ( 4 );
```

The functions:

Function: **ABS**

The absolute value of the argument (operand) is calculated.

Function: **ATN**

The arctangent of the operand is calculated. The operand is given in radians.

Function: **COS**

Returns the cosine of the value given in radians.

Function: **EXP**

Returns the value e ** operand in which e=2.71827183.

Function: **INT**

The "INT" of a value returns its integer portion (greatest integer function). For example, INT (2.34) is 2, while INT (-4.6) is 5.

Function: **LOG**

"LOG" returns the natural logarithm (base e).

Function: **PEEK**

Returns the contents of the given memory location.

Function: RND

"RND" returns a random number depending on the value of the argument. If the argument is negative, a new set of random numbers is produced. This set is dependent on the negative number, so the same negative value produces the same set of numbers. If the value is greater than or equal to zero, a new number will be generated.

Function: SGN

Returns the following values:
-1 if the argument is less than zero.
 0 if the argument is equal to zero.
+1 if the argument is greater than zero.

Function: SIN

Returns the sine of the angle given in radians.

Function: SQR

The square root of the value is calculated. The argument must be positive.

Function: TAN

The tangent of the angle given in radians is the result.

{this page left blank intentionally}

11. Decision Making

Ada is a block-structured language. Instructions which logically belong together are collected together into a block. For example, we write the keyword at the beginning of the executable instructions and the word **end** at the end of the program. The instructions in between belong to a program, they form a block.

Up to now we have only concerned ourselves with programs which are executed sequentially, meaning that we do not know how to make a program execute its instructions in an order other than one pass through all of them, one after the other. In our previous programs, each instruction was executed exactly once. We could not skip any instructions.

One often faces the problem of having to choose between two sets of instructions based on a condition. In English we would formulate this as follows: "If the condition is fulfilled, then execute these instructions, else execute this other set." Two different instruction blocks exist which make up the structure of this program portion. One speaks of structured programming if such structures determine the program. An additional method of structuring programs involves loops, which we will discuss in the next section.

What does a condition in Ada look like?

Example:

```
if HAL > 0 then
--
-- A set of instructions
-- can be placed here.
-- It will be referred to as block_1.
--
else
--
-- Instructions for block_2 can be
-- placed here.
--
end if;
```

We can clearly recognize two blocks. At least one statement must be placed in each block, even if it is just the empty instruction **null**.

The decision statement begins with the keyword **if**. Then follows the condition which determines the branch to the individual blocks. If the condition is fulfilled, in our case **if** the value of HAL is greater than 0, the first block is executed. The first block is comprised of statements from the **if** statement to the **else** statement. Here the program execution branches to the instruction following the end of the decision **end if;**.

If the condition was not fulfilled, the instructions in block_2 are executed.

If no instructions are necessary for block_2, we can place the instruction **null**; there. Another possibility is to leave off this block altogether.

```
if HAL >0 then
--
-- instruction block
--
end if ;
```

Do not forget the semicolon after the **end if** because the conditional is also a statement in Ada and must be separated from following statements by the semicolon.

The following operators are available for forming the condition:

Operator	Meaning
=	**equal to**
/=	**not equal to**
<	**strictly less than**
<=	**less than or equal to**
>	**strictly greater than**
>=	**greater than or equal to**

Exercise

Ask the user if the sentence "Block structures are great!" should be sent to the screen or printer and then do so. The solution has the name "DECISIONS".

{this page left blank intentionally}

13. LOOPS

The loop structure is used to execute a block of instructions more than once without having to retype the block. Let's look first at the endless loop.

```
loop

--

--

-- A sequence of instructions

--

--

end loop;
```

In Ada one calls this construction the "basic loop." This is the simplest form of a loop, but also the one you will need the least. Once you are in this loop you can carry out the sequence of given instructions as long as you want until the computer is turned off. Your computer makes use of such a loop when it is turned on. It waits for a command from you and returns again to the loop when it has carried out the command. This interpretation loop is the principal structure in the computer and all other structures are subordinate to it. This loop reads the keyboard, it will not do you any good to escape from this loop.

It is possible to escape from an endless loop in Ada with the following command:

```
          exit loopname when condition;
```

This instruction means **exit** the loop with the name "loopname" **when** the condition is fulfilled. This condition is similar to a BASIC IF statement. How do we give a name to a loop?

Example:

```
     ROUND:   loop
              --
              --
              --
              -- A sequence of instructions
              --
              --
              --
                   exit ROUND when A /= B ;
              --
              -- A Not Equal To B    "A /= B"

              -- A sequence of instructions


              --


              end loop ROUND;
```

You must write the name of the loop in at least two places: before the keyword **loop**, followed by a colon, and after the keywords **end** and **loop**, followed by a semicolon. In our example the loop "ROUND" will be exited if the value of A is different from the value of B, then the **exit** statement is executed. The program execution will pick up again after the instruction "**end loop** ROUND;".

Take a look at the following example:

```
OUTSIDE : loop
--
-- A sequence of instructions
--
     INSIDE : loop
              --
              --
              -- A sequence of instructions
              --
              --
                     exit OUTSIDE when MM < 3;
              --
              --
              -- A sequence of instructions
              --
              end loop INSIDE ;
              --
--
end loop OUTSIDE;
```

Both loops can be exited by proper selection of the exit criteria in the inner loop.

If you want to run through a loop only a few times, Ada offers the following possibility:

Example:

```
for I in 1..10 loop

--

--

-- the sequence of instructions which is to

-- be executed ten times.

--

--

end loop ;
```

The loop parameter, in our case "I", can only be read within the loop. Upon entry into the loop the parameters will be defined, and will cease to exist after the completion of the loop. In our case the loop parameter assumes the values, one after another: 1,2,3,4,5,6,7,8,9,10. The loop will be carried out ten times.

EXERCISE

You will find the suggested solution under the name "loop" on the Ada Training Course diskette and in section 27. Problem Solutions.

Write a program that prints all the even numbers up to 100, and then all the odd numbers from 100 to 200. The output should be commented.

{This page left blank intentionally}

13. Jumps

Have you already missed the **goto** command? I believe that this command is unnecessary, because in principle all the problems can be solved with sequential procedures, conditionals, and loops. But there is also a "goto" command in Ada. You must indicate the place in the program to which you would like to **jump**. There are no line numbers in Ada like there are in Basic. The provision for such jump destination markers is as follows:

```
<< JUMP LABEL >>
```

In place of JUMP LABEL insert a name of your own. A jump label can be inserted before any instruction in the executable part of your program. The **goto** instruction has the following construction:

```
goto JUMP LABEL;
```

I probably don't need to put an exercise here for you, an example of the **goto** command may be found in the program named DEMO on the Ada diskette. DEMO.OBJ is the compiled and assembled version of the DEMO program, you may simple load and run this program.

{This page left blank intentionally}

14. The Operation of the Compiler

A section not just for experts

Surely you have asked yourself what actually happens after you have told the editor to compile a program. I will answer this on the next pages. For me this is one of the most interesting parts of data processing. I find it simply fascinating to discover the means by which a machine is in a position to analyze a language.

A few things to consider: If you want to express something in a language, you put the words together in sentences. You do this whether you are speaking English or writing a program in Ada. By words in Ada we mean keywords, special characters, and names. The programs which you pass on to the Ada compiler are nothing other than Ada sentences.

Ada is an artificially created language, but nevertheless it is a language which is in the position to form an endless number of sentences. If you want to take only a certain length for your programs, then you can set a maximum length of 5000 sentences. I am convinced that the Ada compiler in this Ada training courses can never come in contact with all the possible programs of this length.

Even more surprising is the fact that it is possible to write a program (the Ada compiler) that can analyze and compile all these sentences. You will perhaps say that this cannot be that difficult, because our language is based on clear cut, definite rules. We need only to write a program that knows these rules and analyzes our sentences following these rules. Easier said than done! Do you have all the rules in your head? Or do you often have to look these things up, as I do? Have you developed certain methods which you follow when you check whether the program is in keeping with the rules?

Now, according to which methods does the compiler of programming languages proceed? You will get to know the methods which the Ada-compiler proceeds. Programming languages are compiled using these methods, so don't quickly forget what I tell you but keep it in the back of your mind, for it will clarify many messages the computer is giving you.

Perhaps you ask why I write about program analysis when all we want is to compile the programs. You will see that the information we need in order to compile a program will be obtained by program analysis. It is noteworthy that the computer requires more time for the analysis of your program than for the production of the workable machine language program.

The program analysis can be divided into three major parts which are executed one after the other on the CBM-64. If more memory were available these parts could be executed in parallel to each other without having to save the data on the diskette. Saving the data naturally takes time, and you can cut down on this time in larger computer systems by the parallel execution of the three parts.

The three parts of the analysis are:

1) **The lexical analysis**

2) **The syntactical analysis**

3) **The semantic analysis**

The subjects of parts 1-3 can be roughly summarized as follows:

The lexical analysis should recognize particular words of the program and filter out the words which don't make sense in Ada.

The syntactical analysis should examine whether a program follows the grammatical rules of Ada. We will later learn what this grammar is like.

The semantic analysis checks whether your program basically makes sense and whether you've followed the rules which in the previous examinations had not been detected.

As you've already recognized, the Ada program goes through ever closer examinations. If the compiler recognizes mistakes in syntax, then a semantic examination is no longer necessary. We'll overlook these points in particular until they're understandable.

{This page left blank intentionally}

15. The Lexical Analysis

A program which executes a lexical check is called a "scanner". The strongest ally with the lexical examination in our case is the editor. The scanner's job is to take only the characters which result in a sensible Ada-program.

We call upon the scanner when we use the function "Compile the program" from the editor. The scanner is part of the editing program and is already located in the memory of the computer.

The task of the scanner is to put our program into a "standard form" which can be processed by the succeeding program, which executes the syntactical test. To do that the scanner takes all of the comments out of our program because they are only for our benefit and are not needed by the compiler. If spaces appear in the program they are removed. This does not apply to spaces in character strings. The scanner prints out the line numbers of the program and changes all the uppercase letters into lowercase. Its main task however is to break the program into Ada words.

What are then all of the Ada words?

1) Ada keywords, i.e. **loop**, **procedure**, etc.

2) The identifiers of the programs

3) The separators, i.e. =, >, etc.

How does the scanner go about this? It reads the program character by character until it encounters one of the following cases:

(a) A comment follows (--)
(b) A space follows
(c) A separator follows
(d) The end of the line is reached

When it reaches one of these it knows that the sequence of characters read was an Ada word.

An example:

The program shall be:

```
00010 procedure LEX_EXAMPLE is
00020 -- Example 1 for lexical analysis
00030       A,B : INTEGER ;
00040 begin
00050       A := B;
00060 end LEX_EXAMPLE ;
```

Line 00010:

The line number 00010 will be printed. The scanner reads over the spaces after the line number, then reads **procedure** and recognizes the space following it. It enters case (b). The scanner knows that **procedure** is an Ada word. It does one more thing though: it determines whether it is a keyword or whether it is dealing with a word selected by the user. When **procedure** is a keyword, a coded message will be generated. This message has the following contents: Here comes a keyword. The keyword is **procedure**. This message is in reality only two characters long and can be interpreted by the following analysis program. We can recognize the meaning of this action when we talk about the syntactical analysis.

The spaces after **procedure** will not be printed. The scanner then reads "LEX_EXAMPLE", recognizes the space, notices that the word is not a keyword and outputs in lowercase letters "lex_example".

With the keyword **is** case (d) occurs, the end of the line is reached. The scanner prints a corresponding message as for **procedure** and moves to the next line.

Line 00020:

The line number 00020 will be printed. Then a comment follows, recognizable by the two minus signs (--), and the rest of the line will be skipped over.

Line 00030:

The line number 00030 will be printed. The following spaces will overlooked. Since a comma, also a separator, follows "A," a word ends here. The scanner recognizes it as one chosen by the user and prints "a". And so on...

Lines 00040 - 00060:

Nothing more will be said about these lines since we have already covered all of the cases. But you should "scan" over these lines for practice.

Upon completion the scanner prints the message that the program is finished and instructs you to insert the distribution disk.

16. The Syntactical Analysis

The program that is responsible for the syntactical analysis is called the Parser.

The Ada-Parser is the parser called by the scanner after its work has been carried out. The parser reads the output of the scanner and checks the program for syntactical accuracy.

First we will clarify what is meant when a program is syntactically correct. Every language is based on certain rules, which determine how sentences are formed out of words. A collection of such rules is called grammar. Most of our recollections of grammar come from school, but have no fear because in programming languages the grammar is much easier to understand than that for other languages like English. The reason for this is that with natural languages we must infer the rules from the language. One examines for example a thousand English sentences and tries to understand the construction of these sentences in terms of rules. If you add to the thousand sentences you will probably have to add new and different rules as well. We can never be sure that we have found all the rules. Of course there are always exceptions to the rules.

This method is not possible for programming languages because the people who write the compiler don't know which programs the user will devise later. The opposite course can also be taken. We first define what the programming language should do and then the grammar is developed. Sentences using correct grammar are syntactically correct, all others will not be accepted by the parser.

In reality the way of proceeding is somewhat more complicated. One would think that once the grammar has been worked out that the parser should be able to work with this grammar. Not all grammar can be processed by every parser, however. The grammar must either be adapted to the parser or the parser to the grammar. Unfortunately there are restrictions on the side of the computer because different parsers require varying amounts of memory space and compiling time.

With the Ada Training Course compiler I have proceeded as follows: I have sought a method for the parser which requires as little memory as possible. Then I devised the parser and rewrote the Ada grammar so that it is more workable. This can be said in two sentences, but it required a great deal of time spent in working out the details since the inconsistency of the new grammar became noticeable only after a great deal of computation.

How does the parser check a program for syntactical accuracy? There are many different methods for doing so. I'd like to present those that the Ada compiler uses. In the literature these methods are known as LL(1) - parsing.

Before we can understand them a few considerations are necessary. So that these don't become much too dry, let's get acquainted with these methods by means of an example.

16.1 The LL(1) Pleasure Garden Part 1:

We take a spacious garden which we will call the LL(1) Pleasure Garden. Within this garden there is an array of amusements such as carrousels, water games, old statues, etc. By every scene stands a mailbox with an inscription, designating the amusement. Further in the park is a whole set of paths and at their intersections, markers directing the way to the next attractions. We'll imagine that our Sunday walk leads us into this pleasure garden. At the entrance we receive a package with cards which will mark out our walk through the garden. We turn up the top card and follow the signs. For example, card 1:" "monkeyhouse". We follow the signs which are at the entrance and show us the way. Arriving at the monkeyhouse we see a mailbox labeled "monkeyhouse" in which to put our card. After we've looked around we turn up the next card and follow the respective instructions. So we wander through the garden until we turn up the card with **exit** written on it, and the walk ends there.

Back to the analysis of our program language: the deck of cards represents our program and on every card is a word of the program. The garden is the grammar according to which the program should be written, and the paths through the garden are the grammatical rules. In the section "19. Ada Grammar" you will find a complete list of applicable grammar with an index.

We will move through one case and parser the following small program.

```
procedure A is

begin

null;

end A;
```

We imagine the keywords **procedure**, **is**, **begin** and **end** in coded form and imagine the empty spaces as not being there, this is the form of the program that the parser receives from the scanner.

After the parser has been loaded and started, the parser program runs through an initialization phase. Here the parser prepares itself for its work. The first rules of the grammar will be read along with others which every program must fulfill. These rules characterize the entire "future" of our Ada programs.

The rules read: compilation ::=compilation_1 E_O_F . The name of the rule is "compilation". The name is the left part of the rule, the part which stands before the "::=". On the right side are two different types of words: One type is the name of other rules and the other consists of Ada words.

This applies with three exceptions:

1) The word E_O_F indicates the end of the program; the scanner adds this word to our program.

2) If there are rule alternatives they are separated from one another by the character "¦". Alternatives mean that the rules are allowed to split up into more cases. This will soon become apparent on its own.

3) If an alternative is empty ,it can be identified by an upper case "L". An empty alternative in a rule means that the rule cannot be used.

To "apply" a rule means to replace the name of the rule with the right side of the definition of that rule. The choice of a blank alternative means that the name is simply erased.

The application of the rule "compilation" results in the following: First we must use the rule "compilation_1", followed by the word E_O_F. If the word "E_O_F" doesn't appear then the parser will interpret it as an error.

How then does the rule "compilation_1" appear? It appears as:

```
compilation_1 ::= context_clause compilation_unit
                  compilation_1
                   ¦ L
```

The use of "compilation_1" allows us two possibilities:

1) The use of the first series, which begins with "context_clause";

2) The use of the second line, the empty rule.

We will consider case 2: we would be finished with the rule "compilation_1" and return to the rule "compilation". Now the word E_O_F follows which means that at the beginning of the program the information must have been given that the program is already at the end. We proceed through the grammar in this manner if we want to compile an Ada program, which does not consist of any instructions. The parser in this case follows the motto "He who does nothing also makes no mistakes". We are working with our program, however, so we must take case 1.

Case 1 begins with "context_clause". Let's look at this rule:

```
context_clause  ::=  "with" identifier  with_1  ";"
context_2  context_clause I  L
```

Line 1 begins with the word **"with"**. To be able to choose this alternative we must have begun our program with **"with"**.

Now there remains only line 2 with the blank empty alternative. The rule "context_clause" is thus processed. How does this appear in the "bookkeeping" of the parser? It has not yet noticed this still unprocessed rule and has marked the corresponding substitutions. For the program then it has:

```
compilation  ::= compilation_unit compilation_1 E_O_F
```

Back to the rule "compilation_1": the next working rule is "compilation_unit".

```
compilation_unit  ::=  "procedure"  identifier  formal_part

                                        subprogram_spe

                              "package" package_se
```

Our program begins with **"procedure"**, therefore we choose alternative 1. We have found an applicable rule and can view the first word in our program as processed. We move on to the next word. This is "a"". We work out further the rules of "compilation_unit". In the meantime we must notice the rest of the rules not yet worked out for compilition. There we must again take up the work when we are finished with the rule. You may think that this could go on forever, but it eventually does come to an end, although this may come after a few hundred steps for even a small program. That is much too much to execute by hand but the computer performs this work faithfully and diligently. In the section "17. Watching the parser do its work" you will find a complete record of our small program. You should go through this record once because it will clear up any questions you might still have.

One question I have not dealt with as of yet will lead us into a new section: Suppose the parser comes to a rule with several alternatives, none of which begin with an Ada word. Which alternative does the parser follow and according to which criteria does it proceed? For each alternative in every rule one can determine which words can occur if a given alternative is selected and followed. I will list the words which are possible and are an alternative to the rule in the following group of words because they fall under the same category and have similar characteristics. We know that every time we need to find a word it is always the word that we last encountered. In the LL(1) pleasure garden this was always the card which we had uncovered and which gave us an intermediate destination on our way. The mailboxes in our garden are now replaced with the Ada words in our grammar. The guideposts in our garden are the family of words from which it is possible to select an alternative to the rule.

I also owe you an answer to the question of why it is possible to analyze a program labeled with LL(1) Parsing. Here is the answer: We always direct our analysis of a sentence (program) from the furthest word on the left, which we haven't found yet on our way through the grammar. This explains the two uppercase "L"s. With the "1" it is a different matter: In each case it sufficies to see only one word into the program. Therefore we insert the word **procedure** and look for this in the grammar, not needing at the moment any information about the words in our program which follow after **procedure**. At first if we have found **procedure**, we need the next word in order to find the rest of the way through the grammar. Perhaps you've already run across this case yourself. It is naturally just another set of analysis procedures for programming languages. You can read about the most current procedures for analysis in many publications.

Until now we have just assumed that the programs which we analyze will be syntactically correct, but that isn't just exactly what we wanted to find out. Let us return in this case to the model of our pleasure garden.

16.2 The LL(1) Pleasure Garden, Part 2

We've gone astray!

We run excitedly through the pleasure garden and come to a new attraction. After we've amused ourselves with it we look for the mailbox and throw in the card. We turn up the next card, look for the signs directing the way to our new goal, look in every direction, whirl around once more and despair! We can't find any signs which show us the right direction in which to go.

But what is wrong and how can we save ourselves? It could be that we lost a card, or there could be one too many cards, or someone could have mixed in a wrong card, or...

What do we do? We assume that the mistake in the cards happened earlier and only just now showed up. Then we have to go back and at an earlier crossway look for our destination. If we assume a card is missing then we must go further and keep an eye out for the destination. If we decide that a wrong card has been slipped in then we simply take the next one and go on as usual. What would you do? Think it over once. I know you'll find another way by which to continue your walk. Will we reach the end or must we resign ourselves to going back to the beginning? Think over what method you're going to use and think about how this will be carried out by the computer.

We will show how this takes place in the analysis of our program. The parser has a program which works through to a certain place, finds a keyword of the grammar or works at a rule where it must move on to a new word. The parser tackles a new rule in every case. This rule is determined by an antecedent, so it can choose only between the alternatives of this rule. It won't find the new word in the group of grammar words, so it comes up with an error. We must now find another way to continue upon our walk.

Which avenues does the parser have to pursue in continuing the analysis, and which is the most promising? If the parser wants to go back on its way through the grammar then it is important for it to have marked the way. We know that the analysis of a program can consist of many steps. That easily requires more memory area than we have available, so we can eliminate this possibility. The parser has noted the possible future for our program which limits this future step by step until only the end symbol is possible. The parser also has the option of determining which Ada words should be anticipated in the future. Another possibility for the parser would be to continue to read our program until this word has been found. What this means in our walk through the garden is that we turn over a new card until we come to a card for which we see a sign. We could have bad luck with that though and stand at the end with no cards. The parser in this case would read to the end and then stop working, which means that the rest of the program won't be checked for syntactical accuracy.

There are many possibilities which the Parser can make use of. I'd like to outline for you at the end of this chapter the one which appears most promising but which unfortunately cannot be realized on the C-64 because of the limited memory. Here now are the possibilities for the Ada Compiler which I feel present themselves in every case as very good options and can manage with little memory space.

Let us suppose that the way through our garden would be marked out so that a visible trail announced your presence, and that in our card deck were cards which were marked to be thrown into the mailboxes along the way. Then I hold the following way of proceeding to be the most sensible: We go along until the next intersection and turn up a number of cards until we reach the card that stands for that intersection. From that card it is very probable that we will be able to find the rest of the way to our goal. We relay this to the parser. First comes the question of "What are the crossroads/intersections in Ada?". You've noticed how every instruction in Ada is separated with a semicolon. Therefore what comes closer to a crossroad than a semicolon? So what is it to forget the semicolon when it only stands as a "tag" behind our instructions? We suppose then that the user begins a new instruction on a new line. We take a new line to be sort of a main intersection. This way of proceeding will not absolutely guarantee results, but it is reasonable and promises the greatest results with the smallest amount of memory allowable.

It is important that we try to make the rest of our program accessible to a syntactical check. Interpreters, such as the CBM-64 has, make this task easy because it simply interrupts the program execution when an error shows up and prints the meaningless message "Syntax Error". The compiler must read over the entire program until it reaches the mistake which is a time consuming process with several errors.

The Ada parser tries next to find the symbol which it has been looking for next. It looks only until the next semicolon or a new line.

Here then is the possibility which I belive to be most promising: If the the parser comes across a symbol which it didn't expect, it looks up in the index whether the symbol has been changed in some way by the user, thereby limiting the future of our program. The index takes up a large place and doesn't fit in the memory of the computer.

We know at the end of the syntactical check whether our program compiles with the rules of Ada grammar. This test is a prerequisite for the semantic test, which also locates the last discrepancies.

In English we know that the sentence "The ducks trills." is grammatically correct, but does it also make sense?

17. Watching the parser do its work

We have selected the following small program as an example:

```
procedure A is

begin

null;

end A;
```

It is obvious that no one would write such a program. If we follow the path of the parser through the grammar we will see that it is already long enough to give us an idea about the syntactical analysis of larger programs.

In this section we will learn a way to analyze a program filled with syntactical errors. The parser is orientated to the grammar and the index, both of which you will find in this manual.

How do we get to know these tools? For that we will run through a small example program. At the end you will surely agree that this is a place in data processing where one needs many words to describe simple facts.

Every program must fulfill Rule 001 of the grammar:

```
001 compilation ::= compilation_1 E_O_F
```

This means that after the application of the rule "compilation_1" you can come to the end of the program. The characters E_O_F stand for the end. These characters lie in the future of our trip through the grammar and as a result do not concern us at the moment. Now we mustn't forget them because we will still need them, so we take note of these characters.

"How does the Parser do this?" will be your next question. The parser notes information about the future of a program in a "memory stack", also just referred to as a "stack".

How does one explain a "stack"? My suggestion is the following: Let us imagine a skyscraper with 2000 stories. This is the capacity of the stack in the Ada Parser. In this skyscraper we find an elevator with only two buttons in the car. Button 1 goes up a floor and button 2 goes down a floor. We can deposit information on every floor but we can't go from the 999th floor to the 700th floor and look at the information on them. We can only go one floor up or down at a time. At first we go into the skyscraper and naturally find ourselves on "Floor 0". And if we go down a floor we negate all of the information on the last floor. We will see that this "construction" is sufficient to point out our way through the grammar. Now E_O_F lies in the future. Therefore we discard the information from "Floor 0" and go up a floor.

We find ourselves now on the second floor and must use rule 002 ("compilation_1").

```
002  compilation_1 ::= context_clause compilation_unit
                        compilation_1
```

```
003                    ; L
```

Here we have two possibilities:

1) We choose 003, but this means that in our program the ending characters E_O_F stand beginning to end, which is not the case;

2) We use rule 002, which means that we must put away the information of the future. "Compilation_1" put aside, and up another floor. "Compilation_unit" put aside and up another floor. Now we can use "context_clause".

```
006  context_clause ::= "with" identifier .........

007                    ; L
```

With these rules we again have two possibilities:

1) Use of 006 : 006 begins with **with**. This is an Ada word and our program must begin with **with**;

2) If our program doesn't begin with **with**, we choose possibility 007. An "L" always means that this rule is blank. We must then turn to the next rule that lies in the future of the program. Therefore we go another floor down, and read the information. We read "compilation_unit". Now we must choose this rule:

```
004  compilation_unit ::= "procedure" identifier formal_part
                                    subprogram_spe

005                      | "package" package_se
```

Our program can begin with the Ada words **procedure** or **package**. If it begins with **procedure** then we choose rule 004. We cannot forget to start our program with **procedure** because it is already tested. I'll abandon the "elevator" and confine myself to telling you which rules will be chosen by the parser.

```
Production  :  060
               063
               019
               015
               Move to line 00020 :
               043
               Move to line 00030 :
               153
               157
               158
               Move to line 00040 :
               155
               039
               016
               060
               063
               003
```

End of the syntactical check.

17.1. Error handling:

If the parser finds a mistake, the possibility exists for us to print the stack. The parser goes down with us from the floor where it now is and prints the respective information. It shows the number of the floor and a number which stands for the stored rules. These numbers correspond to the numbers which stand before the names of the rules in the index. For example, 010 block_statement. Here the number 10 is shown. The Ada words are coded in the memory. They are preceded by the number 255 so that the parser can distinguish them from the rules. The list of keywords is found at the end of this chapter.

Further in the stack is information which directs the parser to carry out certain work. Your program should not only have been checked for syntactical accuracy but also later be converted into a machine language program. So that the semantic analysis and the assembler will be provided with the necessary information, the grammar is expanded in characters. If these appear on the stack, the parser knows that it must supply the information for the following work. Every one of these characters is preceded by the number 252.

17.2. The list of coded Ada words:

Number	Word
97	at
98	do
99	if
100	in
101	is
102	of
103	or
104	abs
105	and
106	end
107	for
108	mod
109	new
110	not
111	out
112	rem
113	use
114	xor
115	body
116	case
117	else
118	exit
119	goto
120	loop
121	null
122	task
123	then
124	type
125	when
126	with

127	abort
128	array
129	begin
130	delay
131	raise
132	elsif
133	entry
134	range
135	while
136	accept
137	access
138	digits 10
139	others
140	pragma
161	record
162	return
163	select
164	declare
165	generic
166	limited
167	package
168	private
169	renames
170	reverse
171	subtype
172	constant
173	function
174	separate
176	procedure
177	terminate
178	=>
179	..
180	**

181	:=
182	/=
183	>=
184	<=
185	<<
186	>>
187	<>
188	exception
254	E_O_F

18. The Semantic Analysis

At the conclusion of the syntactical analysis follows the semantic analysis. The semantic analysis checks whether the program is correct according to program structure. For example a program can be syntactically correct making use of the output routine but missing the assignment of the input and output packages. It is not, however, semantically correct. In the semantic analysis all of the tests are now conducted which could not be done during the syntactical analysis.

Examples:

You want to direct the output to the printer with set_output (PRINTER), but typed "PRONTER" instead of "PRINTER". Syntactically the command is correct because the Parser looks for an identifier in parentheses. However if the machine program is produced, the computer must recognize the device "PRONTER" and know how it should be addressed. There is no such device and it is clear that the command must be rejected. The semantic test undertakes this job.

You have in your program forgotten to declare a data object, or you have incorrectly nested loops or tried out a possibility for which no machine program can be produced - in all of these cases the semantic check can give information as to their correctness.

One can say in simplified terms: Everything that can only be formulated with words rather than additional rules is subject to semantic examination.

How does the program do this semantic test?

When the parser checks a program for syntactical accuracy, it starts on its way with the first rule of grammar. It follows a path through the grammar, which is characteristic for the program. The semantic analysis follows this path and can then carry out the testing.

How does the program for the semantic analysis obtain the necessary information? The grammar, as it is printed in this book, is expanded with additional symbols. If the parser comes upon such a symbol, it then has a specific action to perform.

Example:

It has just processed the rule which means that an identifier is at the end. Then it comes upon a symbol that instructs: Pass on this identifier to the semantic test. It is in this way that the semantic analysis obtains your information.

Closely related to the semantic analysis is the production of the assembler program. If the semantic check is successful, we know that a correct machine program can be produced. Moreover, the program for the semantic analysis is ready to recall all the information for the production of the assembler program. For that reason it is easy to produce the assembler program parallel to the semantic analysis.

If you want to do this then you will have to save all of the necessary information on the disk again. This would mean a longer compiling time. In this method you would produce an executable program right after the semantic check by running an assembler program. This will give you the possibility of actively engaging in the compiling process. This method enables you to combine your own assembler programs with Ada programs or you could change programs produced by the compiler as you wish. So I have settled on this method. I find it good if one not only gives instructions to the compiler "for better or for worse", but also can see his own ideas realized in the produced machine program.

With modern programming languages the cost for the semantic analysis is very high, because one wants to inform the user of all possible errors. In earlier programming languages this was not always so. The compiler in question compiled programs which did not work in every case. They had gaps as it were in the working of the rules. It is possible to use these gaps and draw from the computer possibilities over which the language actually doesn't have control. One programs with "tricks", fully aware of the risk of failure. It is only bad luck if the programmer misses a "gap" by mistake, receives no error message from the compiler and as a result has no idea where he should look for the error in the program. Even the "self-proclaimed" computer experts couldn't help him. I know of a mainframe computer with which the utilization of such a "gap" began to execute a program that quit after awhile and printed the message "computer defect", although the computer was in perfect working order.

With Ada one is protected from such undesirable surprises. The semantic analysis together with the production of the assembler program occupies a great deal memory. The compiler in the C-64 occupies almost all of the memory space available. The great memory area results in a reduction in size of the compiling languages.

19. Ada Grammar

Why do I need the grammar?

The grammar of the language gives information about how a program can be made syntactically correct. It describes the syntax of all possible Ada programs which can be compiled by the Ada compiler in this Ada training course. it can give you very helpful information when the compiler gives you an error message which you do not immediately understand. It can also give information about whether a specific command construction is possible or not. Such grammars exist for most programming languages, and they represent the single greatest aid when writing compilers. They describe what demands the user may make on the compiler. It is a part of the standard of a programming language. Learn to use the grammar! This knowledge will be invaluable when learning to use a new programming language. You can recognize the key points of a language by studying the grammar. You can recognize what possibilities the language offers you, and whether it would pay to learn more about the language. The grammar has the advantage that it yields a great deal of information in a very brief form. I always have the grammar of the programming language I am working with in reach when programming. For programming languages with a relatively small scope, such as BASIC, you can keep the grammatical rules in your head, but you should learn the possibilities and capabilities which programming languages like Ada, FORTRAN, or COBOL offer.

Information on use of the grammar can be found in the sections "14. The compiler operation -- 16. The syntactic analysis." There you will find an example of the path you might take through the grammar when you analyze a program.

The individual rules of the grammar are numbered and you can find them quite quickly with the help of the index.

19.1. The rules of the grammar:

```
001 compilation       ::= compilation_1 E_O_F
002 compilation_1     ::= context_clause compilation_unit
                          compilation_1
003                     | L
004 compilation_unit ::= "procedure" identifier format_part
                          subprogram_spe
005                     | "package" package_se
006 context_clause::= "with" identifier with_1 ";" context_2
                        context_clause
007                     | L
008 context_2        ::= "use" identifier use_1 ";" context_2
009                     | L
010 with_1            ::= "," identifier with_1
011                     | L
012 use_1             ::= "," identifier use_1
013                     | L
014 subprogram_spe    ::= ";"
015                     | "is" declarative_part "begin"
                          sequence_of_statements package_4
                          "end" subprogram_spe_1 ";"
016 subprogram_spe_1 ::= identifier
017                     | L
018 formal_part       ::= "[" parameter_specification
                          parameter_spe_1 "]"
019                     | L
020 parameter_spe_1 ::= ";" parameter_specification
                          parameter_spe_1
021                     | L
022 parameter_specification ::= identifier_list ";" mode
                                type_mark expre_1
```

```
023 mode                ::= "in" mode_1
024                       | "out"
025                       | L
026 mode_1              ::= "out"
027                       | L
028 expre_1             ::= ":=" expression
029                       | L
030 package_se          ::= identifier "is" declarative_part
                            package_1 "end" package_2 ";"
031                     | "body" identifier "is" declarative_part
                            package_3 "end" package_2 ";"
032 package_1           ::= "private" declarative_part
033                       | L
034 package_2           ::= identifier
035                       | L
036 package_3    ::= "begin" sequence_of_statements package_4
037                       | L
038 package_4           ::= "exception" exception_handler
                            exception_1
039                       | L
040 exception_1         ::= exception_handler exception_1
041                       | L
042 declarative_part ::= declarative_1 declarative_part
043                       | L
044 declarative_1       ::= "procedure" identifier formal_part
                            subprogram_spe
045                       | "package"  package_se
046                       | "use"      identifier use_1 ";"
047                       | "type"     identifier "is"
                                       type_definition ";"
048                       | "subtype"  identifier "is"
                                       subtype_indication ";"
049                       | identifier_list ":" switch_decl_1
```

```
050 switch_decl_1     ::= "exception" ";"
051                     | "constant" switch_decl_2
052                     | subtype_indication expre_1 ";"
053                     | array_type_definition expre_1
054 switch_decl_2     ::= subtype_indication expre_1 ";"
055                     | array_type_definition expre_1 ";"
056                    | ":=" universal_static_expression ";"
057 identifier_list   ::= identifier identifier_1_1
058 identifier_1_1    ::= "," identifier identifier_1_1
059                     | L
060 identifier        ::= letter ident_1
061 ident_1           ::= "_" letter_or_digit ident_1
062                     | letter_or_digit ident_1
063                     | L
064 letter_or_digit   ::= letter
065                     | digit
066 character_literal ::= "'" graphic_character "'"
067 string_literal    ::= """ string_1 """
068 string_1          ::= graphic_character string_1
069                     | L
070 graphic_character ::= letter
071                       | digit
072                       | space
073                       | special_character
074 type_definition ::= "[" enumeration_literal type_d_1 "]"
075                     | range_constraint
076                     | "digits 10" range_constraint
077                     | array_type_definition
078                     | "new" subtype_indication
079 type_d_1          ::= "," enumeration_literal type_d_1
080                     | L
081 subtype_indication ::= type_mark constraint
082 type_mark         ::= identifier
```

```
083 constraint         ::= range_constraint
084                      ! index_constraint
085                      ! L
086 range_constraint ::= "range" range
087 range        ::= simple_expression ".." simple_expression
088 enumeration_literal ::= identifier
089                        ! character_literal
090 array_type_definition ::= "array" index_constraint "of"
                              component_subtype_indication
091 index_constraint ::= "[" range index_c_1 "]"
092 index_c_1          ::= "," range index_c_1
093                      ! L
094 exception_handler ::= "when" exception_choice
                          exception_h_1 "=>"
                          sequence_of_statements
095 exception_h_1      ::= "!" exception_choice exception_h_1
096                      ! L
097 exception_choice ::= exception_identifier
098                      ! "other"
099 name               ::= identifier name_1
100                      ! character_literal
101 name_1             ::= "'" identifier
102                      ! "[" simple_expression name_2
103                      ! L
104 name_2             ::= ".." simple_expression "]"
                         ! name_3 name_4 "]"
106 name_3           ::= relational_operator simple_expression
107                      ! L
108 name_4             ::= "," expression name_4
109                      ! L
110 expression         ::= relation expre_2
111 expre_2            ::= logical_operator relation expre_2
112                      ! L
```

```
113 relation           ::= simple_expression rel_1
114 rel_1            ::= relational_operator simple_expression
115                     | L
116 simple_expression ::= simp_1 term simp_2
117 simp_1             ::= unary_operator
118                     | L
119 simp_2             ::= adding_operator term simp_2
120                     | L
121 term               ::= factor term_2
122 term_2             ::= multiplying_operator factor term_2
123                     | L
124 factor             ::= primary fac_2
125 fac_2              ::= "**" primary
126                     | L
127 primary            ::= numeric_literal
128                     | string_literal
129                     | name prim_1
130                     | "[" expression "]"
131 prim_1             ::= "[" expression "]"
132                     | L
133 logical operator   ::= "and"
134                     | "or"
135                     | "xor"
136 relational_operator ::= "="
137                        | "/="
138                        | "<"
139                        | "<="
140                        | ">="
141                        | ">"
142 adding_operator   ::= "+"
143                     | "-"
144                     | "&"
145 unary_operator    ::= "+"
```

```
146                        | "-"
147                        | "abs"
148                        | "not"
149 multiplying_operator ::= "*"
150                            | "/"
151                            | "mod"
152                            | "rem"
153 sequence_of_statements ::= label_1 statement seq_1
154 seq_1                ::= label_1 statement seq_1
155                        | L
156 label_1              ::= "<<" identifier ">>" label_1
157                        | L
158 statement            ::= "null" ";"
159                        | state_1
160                        | "exit" exit_1 exit_2 ";"
161                        | "return" return_1 ";"
162                        | "goto" identifier ";"
163                        | "raise" raise_1 ";"
164                        | if_statement
165                        | case_statement
166                        | block_statement
167 state_1              ::= identifier name_1 state_2
168                        | character_literal state_3
169                        | loop_statement
170 state_2              ::= ":=" expression ";"
171                        | ":" loop_statement
172                        | actual_parameter_part ";"
173 state_3              ::= ":=" expression ";"
174                        | actual_parameter_part ";"
175 if_statement         ::= "if" condition "then"
                           sequence_of_statements if_1 if_2
                           "end" "if" ";"
176 if_1                 ::= "else if" condition "then"
```

```
                          sequence_of_statements if_1
177                     ! L
178 if_2              ::= "else" sequence_of_statements
179                     ! L
180 condition         ::= boolean_expression
181 case_statement    ::= "case" expression "is"
                          case_statement_alternative case_1
                          "end" "case" ";"
182 case_statement_alternative ::= "when" choice case_2 "=>"
                                  sequence_of_statements
183 case_1            ::= case_statement_alternative case_1
184                     ! L
185 case_2            ::= "!" choice case_2
186                     ! L
187 loop_statement    ::= loop_2 basic_loop loop_3 ";"
188 basic_loop        ::= "loop" sequence_of_statements
                          "end" "loop"
189 iteration_rule    ::= "while" condition
190                     ! "for" identifier "in" loop_4 range
191 loop_2            ::= iteration_rule
192                     ! L
193 loop_3            ::= identifier
194                     ! L
195 loop_4            ::= "reverse"
196                     ! L
197 block_statement::=block_1 "begin" sequence_of_statements
                          package_4 "end" ";"
198 block_1      ::= "declare" declarative_1 declarative_part
199                     ! L
200 exit_1            ::= identifier
201                     ! L
202 exit_2            ::= "when" condition
203                     ! L
```

```
204 return_1            ::= expression
205                       | L
206 actual_parameter_part ::= "[" identifier "=>"
                                actual_parameter actual_1 "]"
207                       | L
208 actual_1            ::= "," para_1 actual_1
209                       | L
210 para_1              ::= identifier "=>" actual_parameter
211                       | L
212 actual_parameter ::= name actu_1
213 actua_1             ::= "[" name "]"
214                       | L
215 raise_1             ::= identifier
216                       | L
217 choice              ::= simple_expression
218                       | "others"
219 numeric_literal     ::= integer num_1 num_2
220 num_1               ::= "." integer
221                       | L
222 num_2               ::= exponent
223                       | L
224 integer             ::= digit int_1
225 int_1               ::= "_" digit int_1
226                       | digit int_1
227                       | L
228 exponent            ::= "E" exponent_1
229 exponent_1          ::= "+" integer
230                       | "-" integer
231                       | integer
```

19.2.. Index to the grammar:

The index is constructed as follows:

The number of the rule is the first thing on the line. This appears if you output the stack during the syntactical analysis. Then follows the name of the rule. The number after this gives the number with which the rule is defined in the grammar. The numbers after the slash indicate the grammatical rules in which the given rule is used.

```
001 actua_1                   213 / 212
002 actual_1                  208 / 206,208
003 actual_parameter          212 / 206,210
004 actual_parameter_part 206 / 172,174
005 adding_operator           142 / 119
006 array_type_definition 090 / 53,55,57
007 ---------------------------------------------------------
008 basic_loop                188 / 187
009 block_1                   198 / 197
010 block_statement           197 / 166
011 ---------------------------------------------------------
012 case_1                    183 / 181,183
013 case_2                    185 / 182,185
014 case_statement            181 / 165
015 case_statement_alternative  182 / 181,183
016 character_literal         066 / 89,100,168
017 choice                    217 / 182,185
018 compilation_1             002 / 1,2
019 compilation_unit          004 / 2
020 condition                 180 / 175,176,189,202
```

```
021 constraint               083 / 81
022 context_2                008 / 6,8
023 context_clause           006 / 2,6
024 -------------------------------------------------------
025 declarative_1            044 / 42,198
026 declarative_part         042 / 15,30,31,32,42,198
027 digit                        / 65,71,224,225,226
028 -------------------------------------------------------
029 enumeration_literal      088 / 74,79
030 exception_1              040 / 38,40
031 exception_choice         097 / 94,95
032 exception_h_1            095 / 94,95
033 exception_handler        094 / 38,40
034 exit_1                   200 / 160
035 exit_2                   202 / 160
036 exponent                 228 / 222
037 exponent_1               229 / 228
038 expre_1                  028 / 22,52,53,54,55
039 expre_2                  111 / 110,111
040 expression               110 / 28,56,108,130,131,170,173,180,
                                   181,204
041 -------------------------------------------------------
042 fac_2                    125 / 124
043 factor                   124 / 121,122
044 formal_part              018 / 4,44
045 -------------------------------------------------------
046 graphic_character        070 / 66,68
047 -------------------------------------------------------
048 ident_1                  061 / 60,61,62
049 identifier            060 / 4,6,8,10,12,16,30,31,34,44,46,47,
                               48,57,58,82,88,97,99,101,156,162,
                                   167,190,193,200,206,210,215
050 identifier_l_1           058 / 57,58
```

051	identifier_list	057 / 22,49
052	if_1	176 / 175,176
053	if_2	178 / 175
054	if_statement	175 / 164
055	index_c_1	092 / 91,92
056	index_constraint	091 / 84,90
057	int_1	225 / 224,225,226
058	integer	224 / 219,220,229,230,231
059	integer_rule	189 / 191
060	--	
061	label_1	156 / 153,154,156
062	letter	/ 60,64,70
063	letter_or_digit	064 / 61,62
064	logical_operator	133 / 111
065	loop_2	191 / 187
066	loop_3	193 / 187
067	loop_4	195 / 190
068	loop_statement	187 / 169,171
069	--	
070	mode	023 / 22
071	mode_1	026 / 23
072	multiplying_operator	149 / 122
073	--	
074	name	099 / 129,212,213
075	name_1	101 / 99,167
076	name_2	104 / 102
077	name_3	106 / 105
078	name_4	108 / 105,108
079	num_1	220 / 219
080	num_2	222 / 219
081	numeric_literal	219 / 127
082	--	
083	package_1	032 / 30

084	package_2	034 / 30,31
085	package_3	036 / 31
086	package_4	038 / 15,36,197
087	package_se	030 / 5,45
088	para_1	210 / 208
089	parameter_spe_1	020 / 18,20
090	parameter_specification	022 / 18,20
091	prim_1	131 / 129
092	primary	127 / 124,125
093	---	
094	range	087 / 86,91,92,190
095	range_constraint	086 / 75,76,83
096	raise_1	215 / 163
097	rel_1	114 / 113
098	relation	113 / 110,111
099	relational_operator	136 / 106,114
100	return_1	204 / 161
101	---	
102	seq_1	154 / 153,154
103	sequence_of_statements	153 / 15,36,94,175,176,178,182, 188,197
104	simp_1	117 / 116
105	simp_2	119 / 116,119
106	simple_expression	116 / 87,102,104,106,113,114,217
107	space	/ 72
108	special_character	/ 73
109	state_1	167 / 159
110	state_2	170 / 167
111	state_3	173 / 168
112	statement	158 / 153,154
113	string_1	068 / 67,68
114	string_literal	067 / 128
115	subprogram_spe	014 / 4,44

```
116 subprogram_spe_1          016 / 15
117 subtype_indication        081 / 48,52,54,78,90
118 switch_decl_1             050 / 49
119 switch_decl_2             054 / 51
120 ---------------------------------------------------------
121 term                      121 / 116,119
122 term_2                    122 / 121,122
123 type_d_1                  079 / 74,79
124 type_definition           074 / 47
125 type_mark                 082 / 22,81
126 ---------------------------------------------------------
127 unary_operator            145 / 117
128 use_1                     012 / 8,12,46
129 ---------------------------------------------------------
130 with_1                    010 / 6,10
131
-------------------------------------------------------------
```

{This page left blank intentionally}

20. The Assembler

The assembler is required when one wishes to convert assembly language programs into machine language programs. I would not like to delve any deeper into programming the microprocessor in machine language; you can find that information in numerous other places. What I would like to do is to acquaint you with the characteristics of this assembler.

What does an assembly language program consist of?

1) Instructions which will be translated into machine code by the assembler.

2) Instructions which provide the assembler with information about the program and so control the assembly. These instructions are also called pseudo-instructions or pseduo-operations (pseudo-ops) because they do not correspond to machine language instructions as do regular assembly instructions and do not appear in the machine code. The disassembler cannot reproduce these instructions in its conversion from machine code into assembly language mnemonics. This is possible for instructions of type 1).

I would like to make a few comments about the notation of assembly language programs:

Assembly language programs can be written and stored like BASIC programs. This allows you to view and analyze the assembly language programs which the Ada compiler produces. This is perhaps the greatest aid to you. I left this interface to the Ada compiler open, even though there are faster ways of compiling an Ada program. This allows you to see how the compiler goes about analyzing an Ada program, and exactly what the results of this analysis are.

Comments in an assembly language program begin with a semicolon. A semicolon tells the assembler to ignore the rest of the line. Comments may begin at any point on the line.

Example:

```
10 ; This is a comment which
20 ; stretches over several
30 ; lines.
40 LDA   12     ; load acc with contents
50              ; of memory location 12
```

Spaces function as separators. They separate the basic elements of assembly language programs from each other on the line. An instruction ends with the end of the line. Only one assembler instruction is possible per line.

20.1 Operands

Operands can be decimal numbers, hexadecimal numbers, and symbols (labels, names) of arbitrary length. Symbols must begin with a letter.

Examples:

```
Decimal numbers:          15
                        1000
Hexadecimal numbers:
                       $FFFF
                         $0D
                       $1234

Symbols:
                       OTTO
          JUMPDESTINATION1
               TEXT-OUTPUT
```

Concerning type 1) commands:

The mnemonic abbreviation of commands corresponds to the MOS standard. The notation for the various addressing modes is explained below.

The shift and rotate commands which involve the accumulator:

```
ASL ACCU
LSR ACCU
ROL ACCU
ROR ACCU
```

One-byte commands such as BRK as written as usual.

Direct addressing:

Command construction: First comes the mnemonic abbreviation, then a space, a number sign (#), a space if desired, and finally the operand.

Examples of direct addressing:

```
LDA # OTTO
AND #OTTO
ADC # 13
ADC #13
CMP # $12FF
```

Zero-page and absolute addressing without index:

Command construction: The mnemonic abbreviation, at least one space, operand.

Either zero-page or absolute addressing is chosen based on the size of the operand. If the operand is symbol which has not been defined up to the current point in the assembly language listing, absolute addressing is chosen. This is done because the assembler reads the source code only once in order to save time.

Examples:

```
ORA OTTO
STA 234
LDA $FE
STX 12345
```

Zero-page and absolute addressing with index:

Command construction: Mnemonic, space, operand, comma, and "X" for the X index-register or a "Y" for the index register Y.

Examples:

```
STX OTTO,Y
STY OTTO,X
STA $44,X
LDA 123,X
```

Indexed indirect addressing:

Command construction: mnemonic, as many spaces as desired (but at least one), open parenthesis, arbitrary number of spaces, operand, arbitrary number of spaces, comma, arbitrary number of spaces, an "X", close parenthesis.

Examples:

```
LDA ( OTTO ,X)
STA ( $AA,  X )
```

Indirect indexed addressing:

Command construction: mnemonic, at least one space, open parenthesis, space(s), operand, space(s), closing parenthesis, space(s), comma, space(s), the character "Y".

Examples:

```
LDA ( OTTO ),Y
STA ( 123 ) , Y
```

Indirect absolute addressing:

This type of addressing can be used only with the JMP command.

Example:

```
JMP ( 12345 )
```

Relative addressing:

This method of addressing is used for the relative jumps. Command construction: mnemonic, at least one space, operand. The operand must in this case be a label marking a jump destination. You will learn in the next section how this works.

Examples:

```
BCC LABEL-1
BPL OUTPUT
```

20.2 Pseudo-instructions

The pseudo-ops control the assembler and have only an indirect effect on the corresponding machine language program. Pseudo-ops are denoted by a preceding period. There are also abbreviations for most of the pseudo-ops in order to allow you to write as short an assembly source file as possible.

Take a look at the assembly language programs the compiler creates. This alone should clarify many questions which you might have and you have a collection of examples which you can refer to and expand at any time. Once you have practice in programming in Ada and assembly language, and are familiar with how the compiler works, you can try to optimize the assembly source code. This Ada compiler makes no attempt at optimization.

The instruction: .START

(.START) sets the address at which your machine language program will begin. The operand following determines the start address.

Example:

```
.START 2047
```

The instruction: .END

(.END) tells the assembler that the assembly language program is now done. No example is required.

The instruction: .LABEL or .L

With this instruction you can define symbols as jump destinations. The symbol is assigned the address of the memory location at which the next machine language command will be placed. If you like, you can also you use this symbol to provide the accumulator with the contents of this memory location, for instance.

Examples:

```
Label-1 .LABEL
Label-1 .L
```

The instruction: .EQU or .E

This instruction permits values to be assigned to symbols. In the assembly, the symbol will be replaced by its value. A symbol may be assigned a value only once with .EQU.

Examples:

```
CHARLOTTE .EQU $FEFE
HANS      .EQU 123
JOHN      .E   MONICA
```

The instruction: .VAREQU or .V

This instruction is used in order to change the value of a symbol.

Examples:

```
                    JOHN      .VAREQU CHARLOTTE
                    JOHN      .V      SUSANNE
```

The instruction: .BLOCK or .BL

You need this instruction to reserve space for data in an assembly language program. The operand behind the instruction gives the number of memory locations (bytes) to be reserved.

Examples:

```
                              .BLOCK 555
                              .BL HANS
```

The instruction: .TEXT or .T

If you want to save character strings, you would use this command. The character string is saved at the location at which the instruction occurs. The string is enclosed in quotation marks. The first quotation mark is not saved, although the last is. A character with value zero is also added. This command is most often used to later output the character string. To do this we need only the address at which the text can be found, pass this to a ROM routine, and jump to this routine in order to output the text. See also the examples for the command .COUNT.

Examples:

```
                              .TEXT "Hello, I'm here."
                              .T    "That's just great."
```

The instruction: .BYTE or .B

This command places the value of the operand into the next memory location and reserves it. The value of the operand must correspondingly lie between 0 and 255.

Examples:

```
.BYTE 66
.B    CARLA
```

The instruction: .DBYTE or .DB

The value of the 16-bit operand is broken into two 8-bit quantities. Then the most-significant of the two is placed into memory, followed by the least-significant byte. These memory locations are also reserved.

Examples:

```
.DBYTE 256
.DB    254
```

The first command places the values 255 and 1 in memory.

The second command places the values 254 and 0 in memory.

The instruction: .WORD or .W

This instruction corresponds to the .DBYTE instruction, but it stores first the least-significant byte and then the most-significant.

The instruction: .COUNT or .C

If the assembler encounters this command, the following happens: When the assembler is started, it places the symbols CL and CH in its symbol table. .COUNT assigns values to these symbols. The address at which the next data will be placed is divided into two 8-bit pieces. CL is assigned the least-significant byte and CH the most-significant. If CL or CL appears in the next instructions, these values are substituted. .COUNT actualizes these values.

Example:

Output the sentence "John is a bad boy!"

```
        JMP TEXT-1    ; jump over the
                      ; sentence
        .COUNT
        .TEXT "John is a bad boy!"
TEXT-1  .LABEL       ; jump
                     ; destination
        .LDY # CL    ; load the pntrs
        .LDA # CH    ; for the jump
                     ; to the ROM
                     ; routine
        JSR          ; jump to ROM
        LDA # 13     ; load CR
                     ; character
        JSR          ; jump to the
                     ; kernal output
                     ; routine
```

I hope that you have fun programming in assembly language!

(this page left blank intentionally)

21. The Disassembler:

The disassembler is required when you want to analyze machine language programs. With the help of the assembler you can write machine language programs which you can either run separately or use in a BASIC or Ada program.

A disassembler converts machine code back into the assembly language mnemonics which produced it (or more exactly, to the mnemonics to which the codes correspond). It is not within the scope of this book to discuss programming 65XX family microprocessors. There are a number of good books available on this topic. I would like to recommend the book by Lothar Englisch The Machine Language Book for the Commodore 64. Englisch has a very good programming style. Also worthy of recommendation are the "classics" by Rodney Zaks and Lance A. Leventhal. These two concern only the 6502 microprocessor in general and are neither limited to nor do they give specific information about the Commodore 64. The Programming Manual for the R6500 family from Rockwell International is also good.

The disassembler is stored as a compressed BASIC program on the disk. This has the advantage that you can move the disassembler around in memory as desired. This is not possible with a compiled program. This makes up for the decreased speed in my opinion. If you have a machine language program at locations 2047 to 10000, for example, you can load the disassembler at location 10002. To do this, enter the following lines in command mode:

```
POKE 44, INT( 10002/256 )
POKE 43, 10002 - 256 * PEEK( 44 )
POKE 10002 - 1 , 0
```

You can then load the disassembler with:

```
LOAD "DISASSEMBLER",8
```

If the machine language program lies outside the range 2047 - 12000, you can omit the first three lines of this procedure.

If you have loaded the disassembler at location other than normal (other than typing simply LOAD "DISASSEMBLER",8), you must be sure to return the computer to its original condition when you are finished. This is done with the following lines:

```
POKE 43,1
POKE 44,8
```

If you want to know how far the program which you have in memory extends, enter:

```
PRINT PEEK(45) + PEEK(46) * 256
```

Load the disassembler and start it with:

```
RUN
```

A menu appears from which you can select the various commands of the disassembler. Let us go through the commands one by one.

M : MENU

By pressing the <M> key the menu reappears. This allows you to be informed of the commands at your disposal.

F : FREE SPACE

This command tells you how many free memory locations are left, memory locations whose addresses are higher than the end address of the disassembler. You can get more space for machine language programs by reducing the space required by the disassembler. You must POKE the appropriate values into memory locations 45 and 46 in order to do this.

D : DECIMAL TO HEX

With this command you can convert a decimal number into its hexadecimal equivalent. Hexadecimal numbers are often required when working in machine language, but people still prefer to work with decimal. This command and the one that follows are therefore two of my favorite commands.

H : HEXADECIMAL TO DECIMAL

You can convert a hexadecimal number into a decimal number.

A : SET ADDRESSES

Here you can tell the disassembler in which memory range you would like to work in.

F : MOVE POINTER FORWARD

At the start of the program the work pointer points to the memory location set previously by the preceding command. By pressing the <F> key you increment the pointer by one and output the contents of the location to which it points on the screen.

B : MOVE POINTER BACKWARD

With this command you can decrement the pointer by one and output the contents of the memory location in question.

P : POKE

By pressing this key you can change the contents of the memory location to which the work pointer points. You will be asked for the new contents of the address. Enter this and press <RETURN>. The contents of the memory location are then changed and the pointer is incremented by one.

I : INSERT BYTES

You will be asked for the number of bytes to be inserted. Enter the number and press <RETURN>. Within the selected memory range, all the contents of the memory locations at the current pointer position will be moved upwards in memory by the number of bytes to be inserted. The memory locations so freed are filled with the decimal value 234. This is the op-code for the microprocessor command NOP : NO OPERATION.

D : DELETE BYTES

You you must enter the number of bytes to be deleted. This many bytes will then by deleted at the pointer position. The rest of the selected memory area is then moved down correspondingly.

Y : SYS(xxxxx)

With the <Y> key you can execute a machine language program which starts at the memory location indicated. The address corresponds to the start address of the previously-chosen memory range.

D : DISASSEMBLE & PRINT

Now we come to the disassembling. With <D> we can output a disassembled program to a printer. It appears in hexadecimal as well as decimal notation. We first decide whether we want to enter the start and end addresses in hexadecimal or decimal. If we enter a character other than "Y", we must enter the addresses in decimal. We can end the output at any time by pressing <RETURN>.

F5 : DISASSEMBLE AND PRINT DEC

This command outputs the disassembled program which begins at the current pointer position on the screen in decimal form.

F7 : DISASSEMBLE AND PRINT HEX

Outputs the disassembled program in hexadecimal form, otherwise as command F5.

S : SAVE TO DISK

With this command you can save the contents of a memory range on a diskette.

L : LOAD FROM DISK

With this command you can load the contents of a saved memory range into the memory of the computer from disk.

Try out all of the disassembler commands. Practice is the best way to become familiar with anything, and the best way to be able to work efficiently with the disassembler.

22. Compiler error messages:

When you compile a program, you will certainly find that the compiler has discovered one or more errors in your program. There is no reason to doubt that these errors are valid, although sometimes one would like to.

Errors which will be discovered in the syntactic analysis.

If the compiler discovers an error during the syntactic analysis, it interrupts the analysis. It outputs the line in which it discovered the error. The line can only be output in the form in which the lexical analysis left it. The line therefore does not have its original form, but it can still be easily read. The last character printed on the line is the one which caused the error. The computer will also tell you which characters (or keywords) would be possible at the given place. This does not mean that any of these character would work in this spot, but that the compiler carried its analysis one step further. In the next step it was able to reduce the number of possible characters. The characters given are intended to be suggestions to the programmer as to what should go in the line.

The compiler then informs you which character it would have expected in the course of the continuing analysis. This character must appear in your program. It is also possible, however, that the compiler has gotten so far off track in the analysis up to this point that this message is of no help. You do have an idea of what the compiler expected and how it understood the last instruction.

The compiler now asks you if you want it to output the stack. Refer to the section on working with the compiler for more information about the stack. If you enter a character other than "Y" followed by the <RETURN> key, the stack will not be printed. If you press only the <RETURN> key, you can proceed step by step through the stack by pressing any key.

Having done all this, the compiler attempts to continue with the syntactic analysis. It is possible that one error may result in the compiler getting off track and printing many more error messages which are really only indirect results of the first real error. This is a fault of all compilers, however. You can learn why this is so in the sections dealing with the compiler.

The most common error message during the semantic analysis is "This possibility not implemented!" This indicates that you have chosen a program construction which is syntactically correct but for which no machine code can be created. Otherwise you will get information on what you have done wrong.

Don't despair! Only through practice can one make any progress in data processing. Only he who knows all the error messages of his compiler is really acquainted with it!

23. Run-time Errors:

Run-time errors are those which occur while the program is running, not while it is being compiled.

If your program contains a lexical, syntactic, or semantic error, you get an error message already at compile time. You can then correct your program according to the error message. The most concrete error messages are those produced during the syntactic analysis. The program changes a great deal in form from step to step during the compilation, although the logic does not change.

The machine language program created contains only the necessary information. Anything not absolutely required for its creation has been lost along the way. It is then very short and can be executed quickly. The names of your data objects are of no interest to the machine language program. It knows only at which memory location it can find the data object.

If an error occurs during the execution of the program, the microprocessor stops executing the program and the operating system outputs an error message. For example, if a floating-point variable is assigned the value 6E+50 during the course of a program, the program will be interrupts and the message "overflow" will be printed. You generally do not know where this error occurred and what line to search for the error. The computer cannot give you this information because it does not know it anymore.

23.1 TRACE

In order to make it easier to find these sorts of errors and also allow you follow the execution of your program, there is the possibility to output a "trace" of your program. The trace consists of outputting the numbers of the lines as they are executed. This way you always know which line is by executing at any given time and can so follow the program course.

The compiler must be told from the start that a program is to be created which will leave a trace. The editor will ask you when you tell it to compile a program if you want have a trace or not. If so, type a "Y" and then press <RETURN>. If you press <RETURN> without typing anything, you will get a program without a trace. You can output the trace to a printer with the **set_output** command.

A program with trace is somewhat longer than without because the information about the line numbers must be present in the machine language program. The assembly time is also correspondingly longer.

With the help of the trace and additional output with the put command, you can narrow down the possible locations for an error.

24. List of Ada Keywords:

This list contains all Ada keywords, including those which are not supported in our Ada training course. The lexical analysis, however, recognizes all valid Ada keywords, so that we cannot use one which might interfere with the program's successful compilation on a more comprehensive compiler. This is done to improve the portability of the programs created with this compiler.

The keywords are protected or "reserved." This means that they cannot be used as names by the programmer.

abort	accept	access	all
and	array	at	begin
body	case	constant	declare
delay	delta	digits	do
else	elsif	end	entry
exception	exit	for	function
generic	goto	if	in
is	limited	loop	mod
new	not	null	of
or	others	out	package
pragma	private	procedure	raise
range	record	rem	renames
return	reverse	select	separate
subtype	task	terminate	then
type	use	when	while
with	xor		

{This page left blank intentionally}

25. Problem Solutions:

These solutions are intended to help you if you find that you are not able to formulate a working solution to the practice problems posed at various points in this book. Look through the listings and study my suggestions. These are not intended to represent the best solutions to the problems but they are reasonably efficient and do solve the problems. I have included plenty of comments to help you follow the program flow.

As I said, there are theoretically many possible ways to write a program which yield the same result, an I am convinced that you will find a number of elegant possibilities.

The suggested program solutions are included on the Ada Training Course diskette. These may be loaded from the "editor", they must be saved and compiled on a separate data diskette. **DO NOT COMPILE THESE PROGRAMS ON THE MASTER DISKETTE!!**

Also included on the master diskette is a DEMO program and the compiled version of the program, DEMO.OBJ. The DEMO.OBJ program may be simply loaded and RUN.

output 2

```
with TEXT_IO; use TEXT_IO;
with CBM_64 ; use CBM_64;
--
-- Example for the input and output of data
-- The name and weight of the user
-- will be entered and printed
--
 procedure IN_OUT is
--
-- Declaration of the string variable for
-- name of the user
--
   NAME : string;
--
-- Declaration of the floating-
-- point variables for the weight
--
   WEIGHT : float;
--
begin
--
    screen_clr;
--
   set_row (5);
--
   put ( "  Please enter your name:" );
--
   set_row (8); set_col (4);
--
   get ( NAME );
--
   new_line; put ( "   Your name is :" );
   put ( NAME );
--
   new_line (3);
   put ( "   Please enter your weight:" );
--
   new_line; set_col (4); get (WEIGHT );
new_line (2);
   put_line ( "  You weigh : " );
   put ( WEIGHT );
--
end IN_OUT ;
```

value assign

```
with TEXT_IO; use TEXT_IO;
with CBM_64 ; use CBM_64 ;
--
-- This Program prints a reciept.
--
-- It asks for information about the transcation.
-- Data is entered on the keyboard and then
-- sent to the Printer.
--
procedure VALUE_ASSIGN is
--
-- Declare the string variables :
--
  BUY : constant string := "bought on";
  TAX_RATE : constant string := "4% sales tax
         : ";
  DISKETTE : string ;
  DATE : string;
  NUMBER_DISK : string;
  NAME : string;
--
-- The Price as floating-point
-- variables.
--
  PRICE : float := 0;
  STATE_TAX : float := 0.04;
--
begin
--
  screen_clr;
--
  put_line [ "Enter the buyer    :" ];
  get [ NAME ];
--
  new_line; put_line [ "Enter the date of the sale
 :"];
  get [ DATE ];
--
--  Build the first line.
--
   NAME [ 35..43 ] := BUY [ 1..9];
   NAME [ 46..54 ] := DATE [ 1..9];
--
--  Output the first line to the screen.
```

```
--
   put_line [ NAME ];
   new_line ;
-- Build the second line.
--
   put_line [ "Number if Diskettes purchased?    "];
   get [ NUMBER_DISK ];
--
   put_line [ "Total amount?        "];
   get [ PRICE ];
--
   DISKETTE [ 1..4 ] := NUMBER_DISK [ 1..4 ];
   DISKETTE  [ 6..35 ] := "Diskettes at a Price of
" ;
--
   put [ DISKETTE ]; put [ PRICE ];
   new_line ;
--
--
-- Build the third line :
--
   STATE_TAX := PRICE * STATE_TAX;
--
   put [ TAX_RATE ]; put [ STATE_TAX ] ;
   new_line ;
--
--
-- Output to the Printer.
--
   set_output [ printer ];
   put_line [ NAME ];
   put [ DISKETTE ]; put [ PRICE ];
   new_line ;
   put [ TAX_RATE ]; put [ STATE_TAX ];
   new_line ;
--
   set_output [ screen ];
--
end VALUE_ASSIGN ;
```

loops

```
with TEXT_IO; use TEXT_IO;
with CBM_64 ; use CBM_64;
--
procedure NUMBER_LOOPS is
--
--declare the number variables
--
   NUMBER : float   :=0;
   HILF     : float    ;
--
begin
--
-- output comments.
--
   screen_clr; new_line (5);
   put_line ( "Output the even numbers :");
--
-- Setup the first_loop.
--
   FIRST :  loop
              --
              NUMBER := NUMBER + 1;
              --
              -- The first_loop will quit
              -- when the NUMBER is greater than 50
              --
                 exit FIRST when NUMBER > 50;
              --
              -- Output the even numbers.
              --
              HILF := NUMBER * 2;
              put ( HILF ); new_line;
              --
              end loop FIRST;
--
-- Output the odd numbers.
--
   put_line ( "Output the odd nmubers!");
--
-- Setup the second_loop.
--
   for I in 50..99 loop
   --
   HILF := float ( I ); HILF := HILF * 2 ; HILF :=
HILF + 1;
   -- Output the odd numbers.
   --
   put ( HILF ); new_line;
   --
   end loop;
--
end NUMBER_LOOPS ;
```

decision

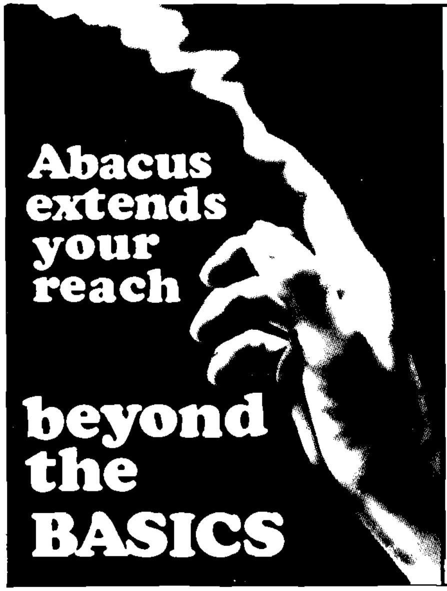
```
with TEXT_IO; use TEXT_IO;
with CBM_64 ; use CBM_64;
--
procedure DECISION is
--
--
-- define the test variable.
--
   TEST : float;
--
--
begin
--
   screen_clr;
--
   new_line ( 5 );
--
   put_line ( "Output  Printer (1) / Screen (2) ?");
--
   get ( TEST );
--
   if TEST=1 then
   --
   -- Output to the Printer.
   --
      set_output ( printer );
      put_line ( "Block structures are great!");
      set_output ( screen );
   --
   --
   else
   --
   -- Output to the Screen.
   --
      put_line ( "Block structures are great!");
   --
   --
   end if;
--
end DECISION ;
```

screen control

```
with TEXT_IO; use TEXT_IO;
with CBM_64 ; use CBM_64;
--
procedure SCREEN_CONTROL is
--
begin
--     Clear the screen.
       screen_clr;
--     Set border to grey_2.
--
       set_border ( grey_2 );
--
--     Set background to white.
--
       set_bkgnd ( white );
--
--     Set the cursor.
--
       set_row ( 10 );
       set_col ( 20 );
--
--     Set the character color to black.
--
       set_type ( black );
--
--     Output  "R 10 , C 20 ".
--
       put ( "R 10 , C 20 " );
--
--     Set cursor in upper left hand corner
--     of the screen.
       cursor_home;
--
end SCREEN_CONTROL ;
```

declarations

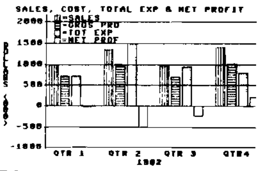
```
with TEXT_IO; use TEXT_IO;
with CBM_64 ; use CBM_64 ;
--
procedure DECLARATIONS is
--
-- Declare the Integer Constant.
--
   WHOLE : constant integer := -1 ;
--
-- Declare the floating-point number.
--
   FLOATP : constant float := 0.3e-6 ;
--
-- Declare the String constant.
--
   STR : constant string := "Hi there!" ;
--
-- Declare the Integer variable.
--
   INT_VAR : integer ;
--
-- Declare the floating-point variables.
--
   PRICE_CHEESE, PRICE_SAUSAGE : float :=0 ;
--
-- Declare the string variable.
--
   HOUSENAME : string := "Sasse" ;
--
-- End of the Declarations.
--
begin
--
   null;
--
end DECLARATIONS ;
```


Abacus
extends
your
reach
beyond
the
BASICS

Make your '64 work fulltime

MAKE YOUR OWN CHARTS...

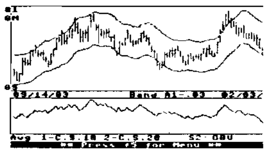
CHARTPAK-64
produces professional quality charts and graphs instantly from your data. 8 chart formats. Hardcopy in two sizes to popular dot matrix printers. **$39.95**
ISBN# 0-916439-19-4



Also Available **CHARTPLOT-64** for unsurpassed quality charts on plotters. **ISBN# 0-916439-20-8 $84.95**

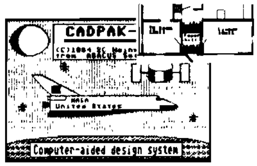
CHART YOUR OWN STOCKS...

TAS-64
sophisticated technical analysis charting package for the serious stock market investor. Capture data from DJN/RS or Warner services or enter and edit data at keyboard. 7 moving averages, 3 oscillators, trading bands, least squares, 5 volume indicators, relative charts, much more. Hardcopy in two sizes, most printers. **ISBN# 0-916439-24-0 $84.95**



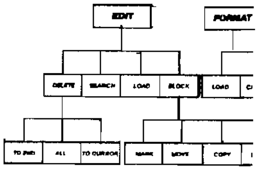
DETAIL YOUR DESIGNS...

CADPAK-64
superb lightpen design tool. exact placement of object using our Accu-Point positioning. Has two complete screens. Draw LINEs, BOXes, CIRCLEs, ELLIPSEs; pattern FILLing; freehand DRAW; COPY sections of screen; ZOOM in and do detail work. Hard copy in two sizes to popular dot matrix printers. **ISBN# 0-916439-18-6 $49.95**



DO YOUR OWN WORD PROCESSING

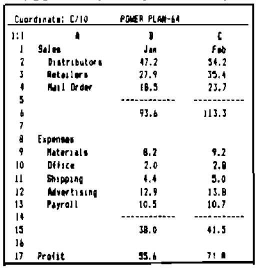
TEXTOMAT-64
flexible worprocessing package supporting 40 or 80 columns with horizontal scrolling. Commands are clearly displayed on the screen awaiting your choice. Quickly move from editing to formatting to merging to utilities. Will work with virtually any printer.



ISBN# 0-916439-12-7 $39.95

CREATE SPREADSHEETS & GRAPHS...

POWER PLAN-64
not only a powerful spreadsheet packages available, but with built in graphics too. The 275 page manual has tutorial section ahd HELP screens are always available. Features field protection; text formatting, windowing; row and column copy, sort; duplicate and delete.
ISBN# 0-916439-22-4
$49.95

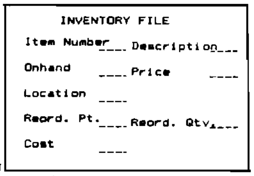


```
Coordinate: C/10        POWER PLAN-64
1:1          A              B           C
 1  Sales                  Jan         Feb
 2    Distributors         47.2        54.2
 3    Retailers            27.9        35.4
 4    Mail Order           18.5        23.7
 5                     ----------- -----------
 6                         93.6       113.3
 7
 8  Expenses
 9    Materials             8.2         9.2
10    Office                2.0         2.8
11    Shipping              4.4         5.0
12    Advertising          12.9        13.8
13    Payroll              10.5        10.7
14                     ----------- -----------
15                         38.0        41.5
16
17  Profit                 55.6        71.8
```

FREE PEEKS & POKES POSTER WITH SOFTWARE
For name & address of your nearest dealer call (616) 241-5510

ORGANIZE YOUR DATA...

DATAMAT-64
powerful, yet easy-to-use data management package. Free form design of screen using up to 50 fields per record. Maximum of 2000 records per diskette. Complete and flexible reporting. Sorting on multiple fields in any combination. Select records for printing in desired format. **ISBN# 0-916439-16-X $39.95**



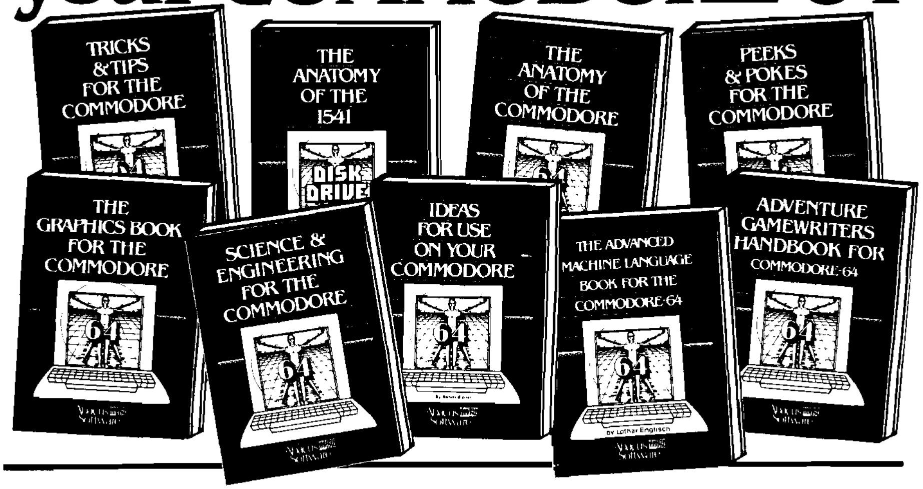
```
        INVENTORY FILE
Item Number____ Description____
Onhand     ____ Price      ____
Location   ____
Reord. Pt. ____ Reord. Qty.____
Cost       ____
```

Other titles available. For FREE CATALOG and name of nearest dealer, write or call (616) 241-5510. For postage and handling, include $4.00 ($6.00 foreign) per order. Money Order and checks in U.S. dollars only. Mastercard, VISA and American Express accepted. Michigan residents include 4% sales tax.

CANADA: Book Center, Montreal (514) 332-4154

You Can Count On
Abacus Software

P.O. Box 7211 Grand Rapids, MI 49510 - Telex 709-101 - Phone 616/241-5510


TRICKS & TIPS FOR THE COMMODORE
THE ANATOMY OF THE 1541
DISK DRIVE
THE ANATOMY OF THE COMMODORE
PEEKS & POKES FOR THE COMMODORE
THE GRAPHICS BOOK FOR THE COMMODORE
SCIENCE & ENGINEERING FOR THE COMMODORE
IDEAS FOR USE ON YOUR COMMODORE
THE ADVANCED MACHINE LANGUAGE BOOK FOR THE COMMODORE 64
ADVENTURE GAMEWRITERS HANDBOOK FOR COMMODORE 64
64
By Lothar Englisch

ADA TRAINING COURSE

Language of the Future

Learn ADA, official development language for the U.S. Department of Defense. The ADA TRAINING COURSE is an introduction to this language for the Commodore 64 user. It presents the basics of the ADA language with an in-depth user's manual and disk-based assembler.

You'll learn the fundamentals of the ADA programming language, which usually runs on mainframe computers much larger than the '64.

The ADA TRAINING COURSE supports a comprehensive subset of the high-level ADA language. You'll learn about structured programming, modular program construction, lexical, semantic and syntatical analysis, constant & variable definitions, types and much more.

The ADA TRAINING COURSE includes a **source file editor, syntax checker, semantics checker, assembler, dis-assembler** and a comprehensive 140+ page step-by-step manual.

For Commodore 64 with 1541 disk drive. Printer optional.

A DATA BECKER Product