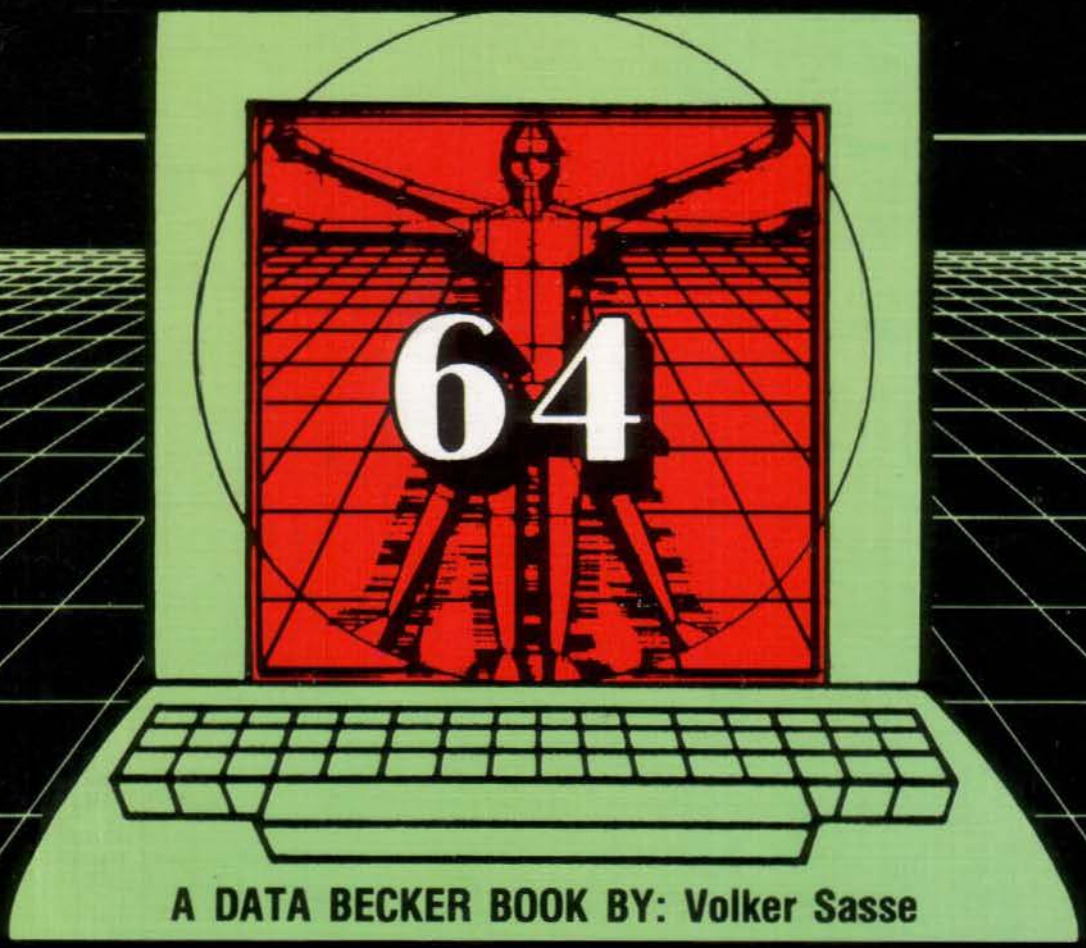
COMPILER DESIGN AND IMPLEMENTATION FOR C-64 & C-128



A DATA BECKER BOOK BY: Volker Sasse

YOU CAN COUNT ON
Abacus Software

B. COPELAND

Compiler Design and Implementation

For the C-64 & C-128

By Volker Sasse

A Data Becker Book
Published by:

Abacus Software
P.O. Box 7211
Grand Rapids, MI 45910

First Printing, July 1985
Printed in USA

Data Becker, GmbH
Merowingerstr. 30
4000 Duesseldorf, W.Germany

Abacus Software, Inc.
P.O. Box 7211
Grand Rapids, MI 49510

ISBN# 0-916439-35-6

PREFACE

Have you ever wanted to know how a compiler changes a high-level language into machine-executable code? Or have you ever played with the idea of writing your own compiler, maybe even writing your own language?

I've found it quite fascinating to learn how one program can translate another program from one language into another. This book illustrates how a computer can transform a program which it is incapable of understanding into one which it can execute. The procedures necessary to do this will be described and implemented in the BASIC language. When you finish this book you'll have a complete compiler, for your study or to change as you desire, perhaps to expand the language.

You can also use this compiler as a model to write one for the language of your choice. Perhaps you'd like to use your computer for a specific purpose, but you haven't yet found the ideal language. Using the ideas presented here, you can develop a language which is suited to your problems and then write the corresponding compiler!

This book is not only for those who want to understand or write compilers, but also for those who want to know more about how their computer works. I would like to direct these readers to the chapter on the operating system.

You can also use this book as an introduction to assembly language since we have included a complete assembler and disassembler. Also included is an introduction to various 6510 machine-language commands needed for compilation.

All of the programs in this book have been tested and run on the C-64 and C-128 in C-64 mode. Using the techniques presented here modifying the programs to use the full capabilities of the C-128 will not be difficult. The decision to present the programs in C-64 format was made to reach a wider audience with the subject matter presented in this book.

TABLE OF CONTENTS

1. WHAT IS A COMPILER?

1.1 WHY DO WE HAVE COMPILERS?

The "heart" of a computer today is its microprocessor. A microprocessor can only execute programs which are written in its specific "machine language". Perhaps you will object and say that your computer also understands the programming language BASIC. This is not really a contradiction. The only reason your computer understands BASIC is because of a machine language program stored in memory which recognizes and executes BASIC commands. When you run a BASIC program, the computer actually executes a machine language program which interprets your BASIC program. That is to say, it interprets the meaning of the BASIC commands and then carries out the corresponding actions. This is a very time-consuming method of program execution. The question arises as to why we don't program in machine language?

Machine language is quite difficult for humans to understand and write programs with. Machine language programs are written as a sequence of binary, decimal, or hexadecimal numbers. This sequence of binary numbers is the true image of the program in the memory of the computer.

Example:

What do you think your Commodore computer does with the following machine language program:

The program in decimal notation:

```
169  65  32  210  255 96
```

The program in hexadecimal notation:

```
A2  41  20  D2  FF 60
```

The program in binary notation:

```
10101001  01000001  00100000  11010010 11111111 01100000
```

It is difficult to determine that this program will output the letter A on the screen. The program below will load the machine language program into memory and then execute it.

```
10 FOR I = 850 TO 855
20 READ X:POKE I, X : NEXT
30 SYS 850
40 DATA 169,65,32,210,255,96
```

The BASIC command 'PRINT "A";' is much easier to understand by the human mind. The BASIC interpreter requires a machine language program to start the necessary machine language commands to 'PRINT "A";'. The microprocessor requires a long time to interpret what must be done before it can be executed. Often the interpretation of a command requires more time than its execution. The situation becomes still more ridiculous when the command 'PRINT "A";' is to be executed a thousand times. The command must then be interpreted a thousand times. Inefficient, but nothing else is possible with an interpreter.

Let us take the program which we have written in a high-level language (such as BASIC) and convert it to a machine language program. Now we have a language with which we can write programs comfortably with and the programs will be quickly processed. The increase in processing speed is because we perform the command recognition process (interpretation) only once. This whole idea makes a compiler!

A compiler therefore is nothing more than a program which has the specific task of translating a high-level language into machine language. This raises the question: "In what language do we write our compiler?".

Let's take another look at our previous example:

We could write:

```
169 + 65          means "Load the accumulator directly
                         with 'A'."

32 + 210 + 255 means "Jump to the subroutine which
                         starts at address 65490."

96                means "Return from subroutine."
```

or abbreviated:

```
LDA # "A"
JSR 65490
RTS
```

We want to write a program which will transform this abbreviated form to machine codes. This effort is certainly

worthwhile since once finished we no longer need concern ourselves with a sequences of numbers.

The abbreviations are called "mnemonics" (memory helps) and a program made up of these abbreviations is called an assembly language program. A program which can convert an assembly language program into a machine language program is called an assembler. More about this later.

A compiler written in assembly language is still not an easy thing to write or comprehend. Since we want to learn how a compiler functions and we own BASIC interpreters, we will write our compiler in the most comfortable language which we have at our disposal--and that is BASIC!

1.2 HOW ARE COMPILERS CONSTRUCTED?

As you already know, a compiler has the task of converting a program from a specific high-level language to machine language. We must now look at the individual tasks to be accomplished. I have divided these tasks into four different areas:

1) **Lexical analysis**
2) **Syntactic analysis**
3) **Semantic analysis**
4) **Code generation**

We proceed with compiler construction in the same manner as for any complex programming project. The primary task is broken into a number of smaller tasks which are easier to solve and implement. Notice that the the word "analysis" occurs three times in the points above, while the word "generation" (here meaning the creation of the machine or assembly language program) appears only once. The task of the first three parts is to check our program for errors and obtain the information necessary for creating the machine or assembly language program.

1.2.1 LEXICAL ANALYSIS:

In the lexical analysis the program is divided into its smallest logical components: the words. In addition, these words are checked to see if they are valid within the language.

This will be clarified using this English sentence:

The elephant practices sack-racing.

In the lexical analysis, this sentence would be divided as follows:

The / elephant / practices / sack-racing / .

Since each word is a valid English word, this sentence would be lexically correct.

1.2.2 SYNTACTIC ANALYSIS:

The syntactic analysis checks to see if the sentence is a correct sentence according to the syntactic or grammatical rules.

Our English is doubtlessly syntactically correct. For a programming language we wish to know if our program is constructed correctly.

An example of a syntactically correct BASIC program:

```
10 FOR J = 1 TO 10
20 FOR I = 1 TO 10
30 PRINT J,I
40 NEXT J
50 NEXT I
60 END
```

This program is certainly syntactically correct, but does it also make sense?

1.2.3 SEMANTIC ANALYSIS:

During the semantic analysis of the previous BASIC program we would determine that the loops are improperly nested. Lines 40 and 50 must be exchanged to get an executable program.

Our English sentence doesn't makes sense either, since elephants cannot participate in sack races.

1.2.4 CODE GENERATION:

During the code generation phase, the program will be converted to machine code (assuming that the program has passed the previous checks). Often the program is translated into an intermediate code. The microprocessor cannot understand the intermediate code but a fast interpreter can be constructed which can execute this intermediate code.

We will generate an assembly language program from our high-level source program which we can then assemble into machine language with an assembler. This has the disadvantage that the compilation time will be longer, but it has deciding advantages, especially for us:

1) We can easily trace the machine commands into which our program will be compiled.
2) The code generation becomes much easier to see.

This method is used to clarify exactly how a compiler operates. Then after you understand how the assembler works, you can rewrite the code generation program so that a machine language program will be created directly.

In closing, I would like to note that there are many different ways of writing a compiler but the fundamental tasks are always the same. In this book I will try to both fully describe these fundamental tasks while providing enough specifics to write a fully-functional compiler.

2. OUR COMPILER LANGUAGE

2.1 AN OVERVIEW

In this chapter I will give a brief overview of the language for which we will write a compiler. Since we have free choice of a language, I simply invented one which is well suited for learning purposes.

It is clear that we want an easily-understood language. We also want to follow the trend in programming language development and choose a block-structured language. I have assembled a selection of capabilities which you will find in every modern programming language.

I have christened our programming language "MINI". First of all because it includes the elementary capabilities of a programming language and second because it is easily extendable, should you so desire.

But what can MINI do?

A MINI program consists of a main program and, if desired, a set of subroutines. In order to be able to communicate with the computer, our language must support the input and output of data, and the output of text or control characters on the screen or printer. To perform calculations, floating-point arithmetic is possible in MINI, including value assignment.

The calculation for the elementary functions such as sine, cosine, logarithm, etc. is also included. We can control the program flow with loops, conditionals, and jumps.

These capabilities will allow us to explain the operation of a compiler. The language is constructed so that you can extend it very easily. It would be very easy, for example, for those who understand floating-point arithmetic to implement integer arithmetic.

Take a look in the following sections and you will surely agree that one can write very nice programs in MINI!

2.2 THE MAIN PROGRAM

Programs in MINI have the following fundamental structure:

```
100    program start is
110    --
120    -- Variables will be declared here later
130    --
140    --
150    begin
160    --
170    -- The executable instructions of our program
180    -- go here.
190    --
200    empty.
210    pend start.
```

This is the smallest possible program in MINI. It has the name "start". The name appears at the beginning and at the end of the program. Our program begins with the keyword **program**.

What are keywords?

Keywords are words which play a special role within a language and are assigned a specific significance. The user cannot change this significance and may use only the declared significance of these words. With the help of the keywords we build the superstructure of our program and inform the compiler of it.

The first word **program** means that a main program is starting. After **program** comes the name of the program which we may choose freely. We will call the name of a program a "program identifier."

What are identifiers in MINI?

Identifiers are arbitrary labels which name objects in MINI, such as programs, subroutines, loops, variables, and so on. Identifiers, in MINI, can have a maximum length of 80 characters and may consist only of letters.

After the name comes the keyword **is**. The declarations of the variables which will be used in our program come between the keyword **is** and **begin**.

Comment lines in MINI are denoted through two successive minus signs (--). Comments occupy the entire line. Lines 110-140 and 160-190 are comment lines.

The line numbers in MINI programs only serve to help us as users find our way in the program. Line numbers also enable us to write programs using the built-in editor on the C-64. We will refer back to the line numbers during the compilation in order to indicate lines containing errors and to allow the user to follow the compiling process.

To write a MINI program, we first switch our computer to upper/lower case mode. We want to write keywords and identifiers in lower case.

The sequence of instructions to be executed will be between the keywords **begin** and **pend** (an abbreviation for "program end"). A sequence of instructions which logically belong

together is called a "block". A block should contain at least one instruction. If we don't know the instruction sequence which will comprise a certain block, we simply insert an instruction, which is a formal instruction, in effect of which is to do nothing. This instruction in MINI is the **empty** instruction.

In MINI, instructions, programs, and subroutines are ended with a period.

After **pend** comes the name of the program again, followed by a period.

The program "start" satisfies the conditions required of a MINI program. The compiled program does not perform any actions.

The following is important to note:

The different line numbers have no meaning in MINI programs. We could have written the program in the following form:

```
1 program start is begin empty. pend start.
```

This form is certainly not very pretty because the structure of the program is no longer apparent. We should always strive to elaborate the structure of a program and not be stingy with comments!

2.3 TEXT OUTPUT

In order to have our programs output their results easily and efficiently, we will create keywords for the output of text.

We have two options for this in MINI:

```
write  "string".
writeln "string".
```

By "string" we mean a sequence of characters which we can type at the keyboard. We enclose the string in quotation marks. This command is concluded with a period. At the start of the statement stands either the keyword **write** or **writeln**. Both commands output the string at the current cursor position.

When using **write**, the cursor is positioned at the end of the last output. **Writeln** places the cursor at the start of the next line, that is, a line feed is sent after the character string. The default output device is the screen. The next section tells you how to output information on the printer.

Here is an example of text output:

The message "Example of" should be printed on one line and on the next line the two strings "the text" & " output" should be printed one after the other.

```
100  program  output  is
110  --
120  --
130  begin
140  --
150  writeln  "Example of".
160  write    "the text".
170  write    " output".
180  --
190  pend  output.
```

2.4 OUTPUT CONTROL

We would like to be able to control the output more than we can with **write** and **writeln**. This may be done by setting the cursor to a desired position on the screen, or clearing the screen or creating a blank line, for example.

We can get a line feed with the command:

linefeed.

Linefeed means that the output on the current line is ended. If you performed output on the current line with **write**, **linefeed** moves the cursor to the next line. If the last output was performed with **writeln**, a blank line is created.

We can clear the screen with the command:

clearscreen.

In order to set the cursor to a specific column, we use the following command:

curscol column.

"column" refers to a number between 1 and 40. Example: Set the cursor to column 25.

curscol 25.

To move the cursor to specific line, we enter the following command:

curslin line.

"line" refers to an integer between 1 and 24. Example: Set the cursor to line 15.

curslin 15.

With our Commodore computers we have the capability to control the screen using control codes. As an example, the cursor moves one position to the right when we send a character corresponding to the ASCII code 29 to the screen. We can output individual characters with the following command:

writeas ASCII code.

"ASCII code" may be a number between 0 and 255. The significance of the individual codes can be found in the user's guide. Here are some of the most important:

3	stop
17	cursor down
18	reverse on
19	cursor home
20	delete character
29	cursor right
34	quotation mark
145	cursor up
146	reverse off
147	clear screen
157	cursor left

A sample program:

```
1000 program  outputex is
1010 --
1020 --
1030 -- This program clears the screen,
1040 -- sets the cursor to line 5, column 5,
1050 -- outputs the sentence "Output on the screen"
1060 -- in reverse type, creates 2 blank lines,
1070 -- and then outputs this sentence
1080 -- once again in normal type.
1090 --
1100 --
1110 begin
1120 --
1130 clearscreen
1140 --
1150 curslin 5.  curscol 5.
1160 --
1170 writeas  18.
1180 --
1190 write "Output on the screen".
1200 --
1210 writeas  146.
1220 --
1230 linefeed. linefeed. linefeed.
1240 --
1250 writeln "Output to the screen".
1260 --
1270 --
1280 pend outputex.
```

2.5 OUTPUT TO THE PRINTER

To direct all output to a different device, such as a printer, we use the keyword **outputdevice**. The instruction to do this is worded:

outputdevice printer.

In order to get the output back on the screen, we use:

outputdevice screen.

This gives us the ability to control a printer within a program.

We must make sure that we do not address the same device twice in a row because we can open a data channel in this form only once.

An example:

```
1000  program  printer  is
1010  --
1020  --
1030  -- This program writes the sentence
1040  -- "Now the printer prints!" on the printer
1050  -- and outputs the sentence "now back
1060  -- to the screen" on the screen.
1070  --
1080  --
1090  begin
1100  --
1110  outputdevice printer.
1120  --
1130  writeln "Now the printer prints!".
1140  --
1150  outputdevice screen.
1160  --
1170  --
1180  writeln "now back to the screen".
1190  --
1200  --
1210  pend printer.
```

2.6 SETTING THE SCREEN COLORS

To make screen control complete we have the capability to change the colors of the border, background, and characters.

The following is a list of the commands with the possible colors. Note the spelling of the colors.

Selecting the border color:

```
border  black.
border  white
border  cyan.
border  red.
border  purple.
border  green.
border  blue.
border  yellow.
border  orange.
border  brown.
border  ltred.
border  greya.
border  greyb.
border  greyc.
border  ltgreen.
border  ltblue.
```

Selecting the background color:

```
background black.
background white.
background red.
background cyan.
background purple.
background green.
background blue.
background yellow.
background orange.
background brown.
background ltred.
background greya.
background greyb.
background greyc.
background ltgreen.
background ltblue.
```

Selecting the character color:

```
type black.
type white.
type red.
type green.
type blue.
type purple.
type yellow.
type cyan.
```

A sample program:

```
100  program  color  is
110  --
120  --
130  -- This program sets the
140  -- background color to black,
150  -- the border color to white,
160  -- and the character color to white;
170  --
180  -- outputs the word "Color" in reverse
190  -- type in line 12 at column 17 and
200  -- sets the background color to white
210  -- and the border color black.
220  --
230  --
240  begin
250  --
260  clearscreen.
270  --
280  background  black.
290  border  white.
300  type  white.
310  --
320  curslin 12.    curscol 17.
330  writeas 18.    writeln "Color".    writeas 146.
340  --
350  background white.
360  border black.
370  --
380  --
390  pend  color.
```

2.7 DECLARATION OF VARIABLES

When we want to use variables in a MINI program, we must first declare them at the start of the program.

In our compiler we want to implement floating-point arithmetic, so we need the declaration for this type.

Floating-point variables can accept values in the range

+/- 1.70141183E+38 and
+/- 2.93873588E-39.

There are two forms of the declaration:

1) Declaration of one floating-point variable:

float variable_identifier.

2) Declaration of several floating-point variables:

float variable_identifier, variable_identifier,...

Notice the underline character ("_", a combination of the Commodore key and the P key) used to join variable_identifer together. In "MINI" a separating character must be between keywords and identifiers. A space is a separator, this is why the underline character is used in the identifier. The Commodore Basic editor will not allow this underline character in a Basic line. We use it only for demonstration purposes because many compilers today, such as the ADA TRAINING COURSE from ABACUS Software, use this syntax.

A sample program:

```
100  program declaration is
110  --
120  --
130  -- tom, dick, and harry will be
140  -- declared as floating-point variables.
150  --
160  --
170  float tom.
180  float dick, harry.
190  --
200  --
210  begin
220  --
230  empty.
240  --
250  pend declaration.
```

2.8 DATA INPUT AND OUTPUT

Now that we know how to declare variables, the next step is the input and output of their values.

For output we use the previously introduced commands, **write** and **writeln**.

```
write   variable_identifier.
writeln variable_identifier.
```

The difference between **write** and **writeln** is that **writeln** moves the cursor to the start of the next line after the output.

For input we use the following command:

```
get variable_identifier.
```

This allows input to be made from the keyboard. When the **get** command is executed, a question mark appears on the screen and a floating-point number is requested. The input is concluded by pressing the return key.

A sample program:

```
100  program in is
110  --
120  --
130  -- This program asks for your
140  -- age and prints it back out.
150  --
160  --
170  float age.
180  --
200  begin
210  --
220  clearscreen.
230  --
240  write "Your age:".
250  get age.  linefeed.
260  write "You are  ".
270  write age.
280  writeln "  years old".
290  --
300  --
310  pend in.
```

2.9 ASSIGNMENT OF VALUES

We can already declare variables, assign values to them via the keyboard, and print out their values. We lack only the ability to change the values of variables within a program, independent of input.

We want to use simple constructions which we can follow through the compilation process. For this reason we will stick to the assignment of strictly positive numbers at first.

Assigning a floating-point number to a variable:

assign floating-point_number **tovar** variable_identifier.

Example:

assign 3.4e+17 **tovar** tom.

After the execution of this command, the variable tom has the value 3.4e+17.

Assigning the value of another variable to a variable:

assign variable_identifier **tovar** variable_identifier.

Example:

assign tom **tovar** dick.

After this command, the value of the variable dick corresponds to that of tom. The value of tom remains unchanged.

Adding two variables:

add variable_identifier **and** variable_identifier **tovar** variable_identifier.

Example:

add tom **and** dick **tovar** tom.

The new value of tom results from the old value of tom plus the value of dick.

Subtracting one variable from another:

subtract variable_identifier **from** variable_identifier **tovar** variable_identifier.

Example:

subtract tom **from** harry **tovar** dick.

The value of dick results from the value of harry minus the value of tom.

Multiplying two variables:

multiply variable_identifier **with** variable_identifier **tovar** variable_identifier.

Example:

multiply harry **with** dick **tovar** dick.

The value of harry is multiplied by the value of dick and the result placed in dick.

Dividing variables:

divide variable_identifier **by** variable_identifier **tovar** variable_identifier.

Example:

divide tom **by** dick **tovar** harry.

The value of harry results from the value of tom divided by the value of dick.

Raising a variable to a power:

power variable_identifier **with** variable_identifier **tovar** variable_identifier.

Example:

power harry **with** dick **tovar** tom.

The value of tom results from the value of harry to the power dick.

A sample program:

```
1000  program arithmetic is
1010  --
1020  --
1030  -- This program demonstrates
1040  -- the floating-point arithmetic.
1050  --
1060  --
1070  -- float monica, thomas, cecilia.
1080  -- float tom, dick, harry.
1090  --
1100  --
1110  begin
1120  --
1130  --
1140  clearscreen.  linefeed.
1150  writeln "Demonstration of floating-point arithmetic".
1160  linefeed.
1170  --
1180  assign 2 tovar monica.
1190  assign 5 tovar thomas.
1200  assign thomas tovar cecilia.
1210  --
1220  --
1230  write "monica = ". writeln monica.
1240  write "thomas = ". writeln thomas.
1250  write "cecilia = ". writeln cecilia.
1260  --
1270  --
1280  add monica and thomas tovar tom.
1290  linefeed.
1300  write "2 + 5 =". writeln tom.
1310  --
```

```
1320  subtract cecilia from tom tovar dick.
1330  write "7 - 2 =". writeln dick.
1340  --
1350  multiply thomas with monica tovar harry.
1360  write "5 x 2 =". writeln harry.
1370  --
1380  divide cecilia by monica tovar dick.
1390  write "5 / 2 =". writeln dick.
1400  --
1410  power monica with thomas tovar tom.
1420  write "2 ↑ 5 =". writeln tom.
1430  --
1440  --
1450  pend arithmetic.
```

2.10 FUNCTIONS

In this chapter we want to become acquainted with the numerical functions which we will implement.

The general command construction goes as follows:

generate function_identifier **from** variable_identifier **tovar** variable_identifier.

or:

generate function_identifier **from** variable_identifier.

The difference between the two instructions is that in the first case the calculated value is assigned to the variable following the keyword **tovar**, while in the second case the calculated value is assigned to the variable from which it was calculated.

In the following list the name of the function will be given followed by a brief description. You may want to refer to the sample program as you go along and read the two together.

Function: absolute

Generates the absolute value of the argument.

Function: arctangent

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

Function: cosine

The cosine of the value given in radians is computed.

Function: exponent

The value e raised to the power given in the argument is calculated. e = 2.71827183.

Function: integer

When we want to convert a floating-point number to an integer, we use this function. For example, the integer 3 results from 3.45 and the integer -5 from -4.6.

Function: logarithm

The natural logarithm (base e) is taken of the value of the variable.

Function: memoryvalue

In this function the variable specifies the address of a memory location and its value is read.

Function: random

With this function you can generate random numbers between 0 and 1. The random numbers are independent of the value of the variable. If the value of the variable is negative, a new set of random numbers is initiated, that is, the same negative arguments will generate the same "random numbers." If the value of the variable is greater than or equal to zero, new numbers will always be generated.

Function: sign

The function "sign" returns the following values:

-1 if the value of the variable is less than zero
0 if the value of the variable is equal to zero
+1 if the value of the variable is greater than zero

Function: sine

"sine" returns the sine of the angle given in radians.

Function: square root

The square root of the value is computed. The value must be positive.

Function: tangent

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

The sample program:

```
1000  program functions is
1010  --
1020  --
1030  -- This program demonstrates
1040  -- the use of functions in
1050  -- the language MINI.
1060  --
1070  --
1080  float a, b, c, d, e, f, g.
1090  float piquarter.
1110  --
1120  begin
1130  assign 4 tovar a.
1140  assign 4 tovar b.
1150   assign 3.1415 tovar c.
1160  divide c by b tovar piquarter.
1170  --
1180  clearscreen.
1190  writeln "Demonstration of functions".
1200  --
1210  linefeed.
1220  generate integer from c tovar d.
1230  --
1240  -- The integer from c was
1250  -- calculated and stored in d.
1260  -- c was not changed.
1270  --
1280  write " c = ". writeln c.
1290  write " d = ". writeln d.
1300  --
1310  generate absolute from a.
1320  --
```

```
1330  -- The contents of a
1340  -- are changed by this form.
1350  --
1360  write " absolute value of 4 = ". writeln a.
1370  --
1380  --
1390  generate arctangent from piquarter tovar e.
1400  write " arctangent of pi/4 = ". writeln e.
1410  --
1420  generate cosine from piquarter tovar e.
1430  write " cosine from piquarter = ". writeln e.
1440  --
1450  generate exponent from b tovar e.
1460  write "exponent of 4 =". writeln e.
1470  --
1480  generate integer from piquarter tovar e.
1490  write " integer of pi/4 =". writeln e.
1500  --
1510  generate logarithm from b tovar e.
1520  write " logarithm of 4 =". writeln e.
1530  --
1540  generate memoryvalue from b tovar f.
1550  write " contents of address 4 =". writeln f.
1560  --
1570  generate random from a tovar g.
1580  write " random of a =". writeln g.
1590  --
1600  generate sign from b tovar g.
1610  write " sign of 4 =". writeln g.
1620  --
1630  generate sine from piquarter tovar f.
1640  write " sine of pi/4 =". writeln f.
1650  --
1660  generate squareroot from b tovar g.
```

```
1670  write " square root of 4 =". writeln g.
1680  --
1690  generate tangent from piquarter tovar f.
1700  write " tangent of pi/4 =". writeln f.
1710  --
1720  writeln "did everything work?"
1730  --
1740  --
1750  pend functions.
```

2.11 CONDITIONALS

Up to now we have not had the ability to skip instructions in our programs. We now want to learn how we can decide which instructions are to be executed based on the value of a variable. To do this we divide a program block into two alternatives and branch to one alternative or the other depending on whether or not our test variable satisfies a certain condition.

Presented formally, it looks like this:

```
if  condition

then
--
--
-- instruction block 1
--
--
else

--
--
-- instruction block 2
--
--
endif.
```

A condition is constructed in the following manner:

```
     variable_identifier  operator  variable_identifier
```

The following operators are available:

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

Here is an example to clarify some of these points:

```
if   tom = dick
then
writeln  "tom is equal to dick".
--
else
--
writeln  "tom does not equal dick".
endif.
```

In this example, the values of tom and dick are compared to each other. If the values are equal, the line "tom is equal to dick" is displayed on the screen. If the values are not equal, the line "tom does not equal dick" is displayed. The program then continues with the instruction following the keyword **endif**.

The instruction "**if then else endif**" encapsules two instruction blocks. Each block is an enclosed set of instructions. In MINI, each block must consist of at least one instruction. Each block has exactly one entry point and one exit point. In the "**if then else endif**" instruction, the entry point is at the "**if**" and the exit at "**endif**".

An advantage of a block-structured language is that you can quickly recognize the instructions which logically belong together.

At this point we should make some comments about the sample and test programs in this book.

Since our goal is to understandg and write a compiler, it is very important for us to have enough sample programs which systematically go through all the possibilities of the compiler. This allows us to see what happens when the compiler encounters a given input. The sample programs in this book are also test programs which we will compile once we have developed our compiler. Since there are a very large number of possible programs which we could write in MINI, it is clear that we can test only the basic capabilities. It is our hope that the compiler also works properly when we exchange two instructions, for example. As you may have noticed in the previous examples, test programs should be written such that errors are made obvious during execution.

The programs on the following pages should be used to test the MINI compiler. The first program should determine if an entered number is odd or even. Either the statement "The number is even." or "The number is odd." should be printed depending on the input.

Sample programs:

```
1000  program even is
1010  --
1020  --
1030  --"This program reads a number
1040  -- from the keyboard and determines
1050  -- whether this number is
1060  -- even or odd.
1070  --
1080  --
1090  float number, temp, two, zero, tempv.
1100  --
1110  begin.
1120  --
1130  clearscreen.  linefeed.  assign 2 tovar two.
1140  --
1150  write "Please enter a positive integer.".
1160  get number.  linefeed.
1170  --
1180  --
1190  assign number tovar temp.
1200  divide temp by two tovar temp.
1210  generate integer from temp.
1220  divide number by two tovar tempv.
1230  subtract temp from tempv tovar temp.
1240  --
1250  assign 0.0 tovar zero.
1260  --
1270  if temp = zero then
1280  --
1290  write number.  writeln "This is an even number.".
1300  --
1310  else
```

```
1320  --
1330  write number.  writeln "This is an odd number.".
1340  --
1350  endif.
1360  --
1370  pend even.
```

```
1000  program selection is
1010  --
1020  --
1030  -- "This program outputs a sentence
1040  -- to the screen or printer
1050  -- depending on user input.
1060  --
1070  --
1080  float  test, one.
1090  --
1100  begin
1120  clearscreen.  linefeed.
1130  --
1140  write "Output to printer (1) or screen (2) ?".
1150  get test.
1160  linefeed.
1170  assign 1.0 tovar one.
1180  --
1190  if test = one then
1200  --
1210  outputdevice printer.
1220  writeln "Output to the printer.".
1230  outputdevice screen.
1240  --
1250  else
1260  --
1270  writeln "Output to the screen.".
1280  --
1290  endif.
1300  --
1310  pend selection.
```

```
1000  program conditional-test is
1010  --
1020  --
1030  -- "This program tests the
1040  -- various conditionals.
1050  --
1060  --
1070  --
1080  float  three, four.
1090  --
1100  begin
1110  --
1120  clearscreen. linefeed.
1130  --
1140  assign 3.0 tovar three.
1150  assign 4.0 tovar four.
1160  --
1170  --
1180  writeln "Conditions:".
1190  --
1200  --
1210  if three = four then
1220  --
1230  writeln "Error on =!".
1240  --
1250  else
1260  --
1270  writeln "No error on =".
1280  --
1290  if three > four then
1300  --
1310  writeln "Error on >!".
1320  --
1330  else
```

```
1340  --
1350  writeln "No error on >".
1360  --
1370  if three >= four then
1380  --
1390  writeln "Error on >=!".
1400  --
1410  else
1420  --
1430  writeln "No error on >=".
1440  --
1450  if three /= four then
1460  --
1470  writeln "No error on /=".
1480  --
1490  if three < four then
1500  --
1510  writeln "No error on <".
1520  --
1530  if three <= four then
1540  --
1550  writeln "No error on <=".
1560  --
1570  else
1580  --
1590  writeln "Error on <=!".
1600  --
1610  endif.
1620  --
1630  else
1640  --
1650  writeln "Error on <!".
1660  --
1670  endif.
```

```
1680  --
1682  else
1684  --
1690  writeln "Error on /=!".
1700  --
1710  endif.
1720  endif.
1730  endif.
1740  endif.
1750  --
1760  pend conditional-test.
```

2.12 LOOPS

In order to be able to execute a block of instructions more than once, we need a program structure called the loop.

In MINI we will have two instructions at our disposal.

1) An infinite loop with an exit instruction
2) A loop instruction with parameters

2.12.1 The infinite loop:

```
loop loop_identifier over
--
--
exit loop_identifier if condition.
--
--
lend loop_identifier.
```

A variant is the infinite loop in the following form:

```
loop loop_identifier over
--
--
-- instruction block
--
--
lend loop_identifier.
```

This is the simplest form of a loop but it is also the form which is used the least. This is because once the loop is

entered, the set of instructions contained within it will be executed over and over until the computer is turned off. Such a loop does have applications, however. One example is one which your computer uses every time you turn it on. It waits for a command from you and returns to the same point after it has executed the command. This interpretation loop is the highest structure in the computer and all other structures are subordinate to it. You will never succeed in escaping this loop.

Or another application:
This program prints out all positive integers sequentially:

```
100  program infinite is
130  -- "This program outputs all
140  -- positive integers in the
150  -- calculation range.
180  float number, one.
190  --
200  begin
210  --
220  assign 1.0 tovar one.
230  --
240  assign 0.0 tovar number.
250  --
260  loop int over
270  --
280  writeln number.
290  --
300  add one and number tovar number.
310  --
320  lend int.
330  --
340  pend infinite.
```

To print out just the first hundred integers, we need the ability to exit the "infinite" loop when the output should be terminated.

The instruction "exit" has the following construction:

exit loop_identifier **if** condition.

We already know how conditions are constructed from the last chapter:

variable_identifier operator variable_identifier

We again have the following operators at our disposal:

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

If the condition is fulfilled, execution jumps to the instruction following the end of the loop whose name is given in the exit instruction.

Let's take a look at an example of this instruction:

```
100  program hundred is
110  --
120  --
130  -- "This program outputs
140  -- all positive integers
150  -- up to one hundred.
160  --
170  --
180  float number, one, hundred.
190  --
200  begin
210  --
220  assign 1.0 tovar one.
230  --
240  assign 0.0 tovar number.
250  --
260  assign 100.0 tovar hundred.
270  --
280  loop int over
290  --
300  writeln number.
310  --
320  add one and number tovar number.
330  --
340  exit int if hundred = number.
350  --
360  lend int.
370  --
380  pend hundred.
```

It may at first seem unnecessary to give a name to each loop within a program, but it has the following advantages:

- The program structure becomes clearer.
- If loops are nested, the nesting is more obvious.
- The exit instruction can then exit nested loops.

Example:

```
100  program exit is
110  --
120  --
130  -- "This program tests
140  -- the exit-loop
150  -- instruction.
160  --
170  --
180  float sam, george, one.
190  --
200  begin
210  --
220  assign 1.0 tovar one.
230  assign 0.0 tovar george.
240  assign 100.0 tovar sam.
250  --
260  --
270  loop outer over
280  --
290  --
300  loop inner over
310  --
320  --
330  writeln george
340  --
```

```
350  exit outer if george = sam.
360  --
370  add one and george tovar george.
380  --
390  --
400  lend inner.
410  --
420  lend outer.
430  --
440  pend exit.
```

Both the inner and outer loops are exited when the condition "george = sam" is satisfied.

The following "small" test program checks the function of the exit instruction.

```
1000  program exittest is
1010  --
1020  --
1030  -- "This program tests
1040  -- the exit instruction.
1050  --
1060  --
1070  float sam, george, harold.
1080  --
1090  begin
1100  --
1105  clearscreen. linefeed.
1110  writeln "Test of the exit instruction.".
1120  linefeed.
1130  --
```

```
1140  assign 2.0 tovar sam.
1150  assign 2.0 tovar harold.
1160  assign 7.0 tovar george.
1170  --
1180  --
1190  loop loopa over
1200  --
1210  loop loopb over
1220  --
1230  loop loopc over
1240  --
1250  loop loopd over
1260  --
1270  loop loope over
1280  --
1290  loop loopf over
1300  --
1310  loop loopg over
1320  --
1330  loop looph over
1340  --
1350  --
1360  exit looph if sam /= george.
1370  --
1380  writeln "Error in exit looph.".
1390  --
1400  lend looph.
1410  --
1420  writeln "Exit looph OK.".
1430  --
1440  exit loopg if sam < george.
1450  --
1460  writeln "Error in exit loopg.".
1470  --
```

```
1480  lend loopg.
1490  --
1500  writeln "Exit loopg OK.".
1510  --
1520  exit loopf if sam <= george.
1530  --
1540  writeln "Error in exit loopf.".
1550  --
1560  lend loopf.
1570  --
1580  writeln "Exit loopf OK.".
1590  --
1600  exit loope if sam <= harold.
1610  --
1620  writeln "Error in exit loope.".
1630  --
1640  lend loope.
1650  --
1660  writeln "Exit loope OK.".
1670  --
1680  exit loopd if george > harold.
1690  --
1700  writeln "Error in exit loopd.".
1710  --
1720  lend loopd.
1730  --
1740  writeln "Exit loopd OK.".
1750  --
1760  exit loopc if george > harold.
1770  --
1780  writeln "Error in exit loopc.".
1790  --
1800  lend loopc.
1810  --
```

```
1820  writeln "Exit loopc OK.".
1830  --
1840  exit loopb if sam >= harold.
1850  --
1860  writeln "Error in exit loopb.".
1870  --
1880  lend loopb.
1890  --
1900  writeln "Exit loopb OK.".
1910  --
1920  exit loopa if sam = harold.
1930  --
1940  writeln "Error in exit loopa.".
1950  --
1960  lend loopa.
1970  --
1980  writeln "Exit loopa OK.".
1990  --
2000  writeln "Test program done.".
2010  --
2020  pend exittest.
```

This test program is much too short because we have not exhausted all of the loop constructions and conditions. Many more cases are possible which may produce errors, but we'll confine ourselves to this for now.

2.12.2 The loop instruction with parameters:

When we know the exact number of times that a loop is to be executed, we would choose this form of the loop instruction:

The general form is:

```
for variable_identifier from variable_identifier
to variable_identifier repeat
--
--
lend.
```

An example:

```
100  program count is
120  --
130  -- "This program outputs the numbers
140  -- from 1 to 10.
160  --
170  float index, lowerbound, upperbound.
180  --
190  begin
200  --
210  assign 1.0 tovar lowerbound.
220  assign 10.0 tovar upperbound.
230  --
240  for index from lowerbound to upperbound repeat
250  --
260  writeln index.
270  --
280  lend.
290  --
300  pend count.
```

Test program for nested loops:

```
1000  program partest is
1020  --
1030  -- "This program tests the nested
1040  -- loops with parameters.
1050  -- "The result should be 1000.
1070  --
1080  float indexa, indexb, indexc.
1090  float lowerbound, upperbound, number, one.
1100  --
1110  begin
1120  --
1130  assign 1.0 tovar lowerbound.
1140  assign 10.0 tovar upperbound.
1150  assign 0.0 tovar number.
1155  assign 1.0 tovar one.
1160  --
1170  --
1180  for indexa from lowerbound to upperbound repeat
1190  --
1200  for indexb from lowerbound to upperbound repeat
1210  --
1220  for indexc from lowerbound to upperbound repeat
1230  --
1240  add one and number tovar number.
1250  --
1260  lend.
1270  lend.
1280  lend.
1290  --
1300  writeln "The result is:". writeln number.
1310  --
1320  pend partest.
```

2.13 JUMPS

Sometimes it is necessary to jump to a specific spot in a program. This is done with a jump command such as the GOTO command in BASIC. Example: GOTO 100. This means "Go to the line with the number 100." In MINI however the line numbers have no significance. We must use a somewhat different method. First we must identify the location to jump to:

```
jumplabel  name.
```

Now we can jump to this point with the statement:

```
jump       name.
```

The "name" here is a combination of letters not found anywhere else in the program (the same name cannot be used in a loop or as a variable, for instance).

An example:

```
100  program jumpex is
110  --
120  --
130  -- "This program is ended
140  -- when 1 is entered.
150  --
160  --
170  float one, test.
180  --
190  begin
200  --
210  assign 1.0 tovar one.
220  --
230  jumplabel start.
240  --
250  get test. linefeed.
260  --
270  if test = one then
280  --
290  empty.
300  --
310  else
320  --
330  jump start
340  --
350  endif
360  --
370  pend jumpex.
```

2.14 SUBROUTINES

We use subroutines when we wish to write a single set of statements that may be used many times throughout the program. Subroutines in MINI do not work with local variables; they may contain only executable instructions.

Subroutines are appended to the main program and are defined as follows:

```
subroutine subroutine_identifier is
--
--
-- instruction block
--
--
srend subroutine_identifier.
```

It is called with the following command:

```
call subroutine_identifier.
```

Once again we offer a sample program which clarifies the instruction better than an extended description of it.

```
100  program subrtest is
110  --
120  --
130  -- "This program calls either
140  -- subroutine a or b.
150  --
160  --
170  float test, one.
180  --
190  begin
200  --
210  assign 1.0 tovar one.
220  --
230  jumplabel start.
240  --
250  writeln "Subroutine a(1) / b(2)".
260  --
270  get test. linefeed.
280  --
290  if test /= one then
300  --
310  call a.
320  --
330  else
340  --
350  call b.
360  --
370  endif.
380  --
390  jump start.
400  --
410  pend subrtest.
420  --
430  ----------------------
```

```
440  subroutine a is
450  --
460  writeln "Subroutine a.".
470  --
480  srend a.
490  --
500  ----------------------
510  subroutine b is
520  --
530  writeln "Subroutine b.".
540  --
550  srend b.
```

3. THE LEXICAL ANALYSIS

3.1 WHY HAVE A LEXICAL ANALYSIS?

The lexical analysis is the first step in the analysis of a program. Its job is to put the program to be compiled into a form that the following steps can work with more easily.

The programs should be put into some kind of standard form which does not contain any parts which are not important to the program. There are many ways to rewrite a MINI program such that the logic of the program does not change but its appearance does. We can convert between upper and lower case, we can use multiple spaces, we can insert comments, select line numbers arbitrarily, and so on.

The purpose of the lexical analysis is to separate the wheat from the chaff, so to speak. This is taking into account the perspective of the compiler since programs without comments are not good at all from the perspective of the user. The program is divided into things called tokens which are the smallest logical units of the language/program. The tokens in MINI programs are the keywords such as **program**, **is**, **=**, **.**, **/**, **pend**, etc. The words selected by the user such as the program identifier, variable identifiers, ... are also designated as tokens, just as are strings or numbers.

A program which searches through a sequence of characters (our MINI program) for tokens (words) is also called a scanner or parser.

We have clarified why we would want to perform a lexical analysis. The details about the operation of the scanner are contained in the following sections. To find out why the scanner preforms an important task, read the chapter on the syntactic analysis.

3.2 HOW DOES THE SCANNER WORK?

Let's take a look at a line from one of the programs in the last section.

340 **exit** int **if** hundred = number.

How does the scanner process this line?

At the start of the line is the line number. Since this is unimportant for the program, it is skipped. Then comes the letter combination "exit". After these letters is a space. The scanner recognizes the end of the word by this space. Now we want to know if "exit" is a keyword or a word selected by the user. The scanner searches through a table of keywords and finds the keyword **exit**. For all of the additional analysis steps it is no longer necessary to retain the entire word. Instead, we assign numbers to all of the keywords. Then we substitute the required number at the spot in the program where the keyword appears. This simplifies the processing of keywords later since we no longer have to search for the word in a table.

After the word "exit" the scanner reads the word "int", the end of which it again signaled by the trailing space. This word is not a keyword so we make note of the word "int". We already know how the scanner recognizes the following word "if" and what actions it performs.

In every language there are characters which clearly mark the end of a word. These characters are called delimiters. In MINI these include the equal sign (=) and the period (.). It should now be clear how the rest of the line is

processed. This has been a very imprecise description, designed to give us a brief look at the scanner operation. But the separate tasks of the scanner must first be stated more exactly. Some preliminary tasks must be carried out.

The individual tasks of the scanner:

- Read program and output status line.
- Recognize delimiters.
- Recognize keywords.
- Remove spaces, except for those in
- strings enclosed in quotation marks, as in "Hello everyone out there."
- Remove comment lines.
- Output tokens.

3.3 READ PROGRAM AND OUTPUT STATUS LINE

In order for the computer to be able to analyze a program, it must first read it. This is often easier said than done.

We have not yet discussed how the programs to be compiled will be written. A program used to write other programs is called an editor.

Editors can be word processing programs, for example, or they can be tailored for writing programs in a specific language. The following case is the simplest for us:

We'll use the editor built in to the computer, that is normally used for writing BASIC programs. These programs can be manipulated, changed, saved, and printed just like BASIC programs. We'll assume that we have our program saved on disk. In a sense, this file is only a text file which we want to read as straight text. By this we mean that all the letters which we typed in should appear exactly as we typed them on the computer. However this is not how the text is found. The BASIC editor stores all BASIC keywords as a number (token). For example, the token 129 replaces the word "for". The BASIC editor performs a lexical analysis of a line, but for a BASIC program and not for MINI. We must undo this analysis before we can perform our own. The lines are also stored in a specific form in memory and transmitted to the disk in this form.

Let's take a look at what we get when we read a text file created by the BASIC editor. At the start of the file we read two characters which contain information specifying the memory locations in the computer in which the program was

found when it was created. These two characters are not of interest to us. Following these come the individual program lines in the following form:

The first two characters form a two-byte address telling the editor at which address the next line begins. We skip these characters as well.

The following two characters contain the line number of the program line. Although the line numbers are irrelevant for our MINI programs, we will use them as reference points when we later output the status of our program. The line numbers are represented as follows:

The ASCII value of the first character is added to 256 times the ASCII value of the second character.

Example:

We read the character A and I. The ASCII values of A and I are: A = 65 and I = 73. This results in a line number of 65 + 73 * 256 = 18751.

Now we read the individual characters of our program line in which the BASIC keywords are represented as tokens whose ASCII values lie in the range between 128 and 203. The normal characters have an ASCII value ranging from 0 to 127, so we always know when we have read a normal ASCII character or a BASIC keyword. From the tokenized BASIC word we can get the full text of the word.

Example:

We read the following ASCII values:

```
80, 128
80 = "p"  128 = "end"
which yields the word "pend"
```

The BASIC editor denotes the end of the line with character of ASCII value zero.

The end of the program is signaled by a link address with the value zero zero (two zeros).

Now we want to write a program which reads a text file, converts it entirely to text, and outputs it to the screen line by line. Do to this we need a list of the assignments of characters to ASCII values and the assignment of the tokens to the BASIC keywords.

When we talk about ASCII values, we are referring to the decimal values of the characters and keywords which the C64 uses internally. This list would look different for other computers, but you can determine the values from the documentation that came with your computer.

Here is the list of characters/words and their ASCII values:

Value	Character	Value	Character
32	space	33	!
34	"	35	#
36	$	37	%
38	&	39	'
40	(	41	)
42	*	43	+
44	,	45	-
46	.	47	/
48	0	49	1
50	2	51	3
52	4	53	5
54	6	55	7
56	8	57	9
58	0	59	;
60	<	61	=
62	>	63	?
64	@	65	a
66	b	67	c
68	d	69	e
70	f	71	g
72	h	73	i
74	j	75	k
76	l	77	m
78	n	79	o
80	p	81	q
82	r	83	s
84	t	85	u
86	v	87	w
88	x	89	y
90	z	91	[
92	English pound	93	]
94	↑	95	<- left arrow
96	shifted space		

(graphics symbols through 127)

128	end	129	for
130	next	131	data
132	input#	133	input
134	dim	135	read
136	let	137	goto
138	run	139	if
140	restore	141	gosub
142	return	143	rem
144	stop	145	on
146	wait	147	load
148	save	149	verify
150	def	151	poke
152	print#	153	print
154	cont	155	list
156	clr	157	cmd
158	sys	159	open
160	close	161	get
162	new	163	tab(
164	to	165	fn
166	spc(	167	then
168	not	169	step
170	+	171	-
172	*	173	/
174	↑	175	and
176	or	177	<
178	=	179	>
180	sgn	181	int
182	abs	183	usr
184	fre	185	pos
186	sqr	187	rnd
188	log	189	exp
190	cos	191	sin

192	tan	193	atn
194	peek	195	len
196	str$	197	val
198	asc	199	chr$
200	left$	201	right$
202	mid$	203	go

Now we can write the program which will convert our program to text and output it line by line. We'll then expand this program to include the rest of the subtasks. We want to write the program so that we can later use it as a subroutine for the syntactic analysis.

The program for syntactic analysis will call the lexical analyzer each time it needs a new token.

The structure of the program:

The lines up to number 1000 will be used for declaration and explanation of the variables used.

Lines between 1000 to 2000 are reserved for initalization tasks.

The lines from 2000 - 50000 contain the program for syntactic analysis.

At line 50000 we'll start writing the program for the lexical analysis.

```
10   rem   program for the lexical
20   rem   and syntactic analysis
30   rem   of mini programs.
40   rem   --
50   rem   declarations
60   rem   --
70   :
```

We are assuming that the program to be compiled is on a disk and that the computer is connected to a disk drive. We want to write the output of the syntactic analysis to a file on the diskette and send the program status to either the screen or printer.

```
 80   fl = 8  :     rem  device number of the disk drive
 90   be = 15 :     rem  logical file number and secondary
                         address of the disk drive
 95   :             rem  command channel
100   in = 8  :     rem  logical file number and secondary
                         address
105   :             rem  of the channel for the input file
110   ou = 7  :     rem  logical file number and secondary
                         address
115   :             rem  of the channel for the output file
120   pr = 3 :      rem  Output channel for the status
125   :             rem  screen is default
130   i1$="":i2$="":i3$="": rem   input variables
140   i1=0  :i2=0  :i3=0  : rem   the corresponding
                                  ASCII values
```

A complete listing of the program without comments will be given in section 4.9.

Let's start with the program initialization:

```
1000   rem  --
1010   rem  preparations
1020   rem  --
1030   :
1040   print chr$(147);chr$(14):rem clear screen
```

Line 1040 will have the effect of clearing the screen, and turn on the upper-lowercase mode.

```
1050   print : print "The MINI compiler:" : print
1060   print "status to the printer or screen (p/s)?"
1070   get i1$ : if i1$ = "p" or i1$ = "s" then 1090
1080   goto 1070
1090   if i1$ = "p" then pr = 4
1100   open pr,pr,7 : rem  open the status channel
```

The corresponding output channel for the program status will be opened. This results in open 3,3,7 for the screen and open 4,4,7 for the printer.

The printer must have the device address 4, if not line 1090 must be changed.

```
1110  print : print "Please insert the data disk"
1120  print "and press RETURN." : print
1130  get i1$ : if i1$<>chr$(13) then 1130
1140  print "Name of program to be compiled"
1150  input i2$
```

Now we have all the information we need to open the appropriate files on the disk. First we'll open the disk drive command channel and initialize the drive. This has the

effect of placing the disk drive in the condition it was in when it was turned on.

We want to write the output of the syntactic analysis to a sequential file with the name "mini-syn". If we have already compiled a program on this disk, this file exists so we must first erase it. It may happen that the user presses the RETURN key when asked to insert the data disk without actually inserting a disk. We want to write a subroutine which we can call whenever we want to know if a disk error has occured.

Opening the command channel:

```
1160   open be,fl,be
1170   print#be,"i"
```

The subroutine for error handling:

```
60000  rem  read the error channel
60010  :
60020  input#be,en,em$,et,es :if en=0 then return
60030  print
60040  print "****  Disk error"
60050  print "****  Error number : ";en
60060  print "****  Error report :"
60070  print "****  ";em$
60080  print "****  Track        : ";et
60090  print "****  Sector       : ";es
60100  print#be,"i"
60110  close be : close pr
60120  print : print "**** Program stopped"
60130  end
```

We are using the new variables en, em$, et, es.

We add a line 150:

```
150    en=0:em$="":et=0:es=0  : rem read error channel
```

Erasing "mini-syn":

```
1180    print#be,"s:mini-syn"
1190    input#be,en,em$,et,es
1200    if en<>1 then 60030
```

After erasing the file, en has the value 1. We cannot call the error subroutine directly, so we read the error channel ourselves.

Now we open the input and output files:

```
1210    open in,fl,in,i2$+"p,r"
1220    get#in,i1$,i1$
1230    open ou,fl,ou,"mini-syn,s,w" : gosub 60000
```

In line 1220 we skip the first two characters of the input file since they have no significance for us.

```
1240    gosub 59000
```

In the subroutine at line 59000 we want to save the tokenized BASIC words in ba$. Since there are a total of 76 of them, line 160 reads:

```
160    dim ba$(75)    : rem text for basic words
```

```
59000  rem text for basic keywords
59005  :
59008  ba$(0)="end"
59010  ba$(1)="for"
59020  ba$(2)="next"
59030  ba$(3)="data"
59040  ba$(4)="input#"
59050  ba$(5)="input"
59060  ba$(6)="dim"
59070  ba$(7)="read"
59080  ba$(8)="let"
59090  ba$(9)="goto"
59100  ba$(10)="run"
59110  ba$(11)="if"
59120  ba$(12)="restore"
59130  ba$(13)="gosub"
59140  ba$(14)="return"
59150  ba$(15)="rem"
59160  ba$(16)="stop"
59170  ba$(17)="on"
59180  ba$(18)="wait"
59190  ba$(19)="load"
59200  ba$(20)="save"
59210  ba$(21)="verify"
59220  ba$(22)="def"
59230  ba$(23)="poke"
59240  ba$(24)="print#"
59250  ba$(25)="print"
59260  ba$(26)="cont"
59270  ba$(27)="list"
59280  ba$(28)="clr"
59290  ba$(29)="cmd"
59300  ba$(30)="sys"
59310  ba$(31)="open"
```

```
59320  ba$(32)="close"
59330  ba$(33)="get"
59340  ba$(34)="new"
59350  ba$(35)="tab("
59360  ba$(36)="to"
59370  ba$(37)="fn"
59380  ba$(38)="spc("
59390  ba$(39)="then"
59400  ba$(40)="not"
59410  ba$(41)="step"
59420  ba$(42)="+"
59430  ba$(43)="-"
59440  ba$(44)="*"
59450  ba$(45)="/"
59460  ba$(46)="↑": rem  "[up arrow]"
59470  ba$(47)="and"
59480  ba$(48)="or"
59490  ba$(49)="<"
59500  ba$(50)="="
59510  ba$(51)=">"
59520  ba$(52)="sgn"
59530  ba$(53)="int"
59540  ba$(54)="abs"
59550  ba$(55)="usr"
59560  ba$(56)="fre"
59570  ba$(57)="pos"
59580  ba$(58)="sqr"
59590  ba$(59)="rnd"
59600  ba$(60)="log"
59610  ba$(61)="exp"
59620  ba$(62)="cos"
59630  ba$(63)="sin"
59640  ba$(64)="tan"
59650  ba$(65)="atn"
```

```
59660  ba$(66)="peek"
59670  ba$(67)="len"
59680  ba$(68)="str$"
59690  ba$(69)="val"
59700  ba$(70)="asc"
59710  ba$(71)="chr$"
59720  ba$(72)="left$"
59730  ba$(73)="right$"
59740  ba$(74)="mid$"
59750  ba$(75)="go"
59760  return
59770  :
```

We execute this subroutine once at the start of the lexical analysis.

Now we are almost far enough to write the program which will be the first thing executed by the syntactic analysis. This program does nothing other than return tokens to the syntactic analyzer. But we must help the lexical analyzer a bit more by reading the first linking address and the first line number. To do this we write a subroutine at line 51000 which we will later call whenever we have reached the end of a line and this task must be performed.

For the lexical analysis we should have the character which we are currently processing in i1$ and the next two characters in i2$ and i3$, this allows us to look a little ways ahead in our program. This is a very practical thing, as we will see later. We will store the ASCII values of the characters in i1, i2, and i3.

We'll write the subroutine which reads a character and shifts the contents of i1$, i2$, and i3$ at line 51500.

```
51500   rem read character
51505   :
51510   i1$=i2$ : i2$=i3$ : get#in,i3$
51520   i1=i2   : i2=i3   : if i3$="" then i3=0 : return
51530   i3=asc(i3$) : return
51540   :
```

We have to call this subroutine twice before we can jump to the subroutine to read the start of the line at 51000.

```
1250    gosub 51500 : gosub 51500 : gosub 51000

51000   rem start of line
```

The linking address for the next line is now in i2$ and i3$. We will skip over it.

Next come the two characters which contain information about the line number. We want to write the line number to the output file so that we can access it later. To do this we first send a character with the ASCII code 254 and then the two characters for the line number to the disk drive. The character 254 cannot occur in any other context, so we know that when we later read the character 254 from our file, that a line number follows.

In addition, the output to the screen or printer should be placed on the next line, so we send a carriage return to channel pr. Finally we output the line number.

Then i1$, i2$, i3$ must be initialized again. The first character of the new line will be placed in i1$.

```
51005  :
51010  gosub 51500 : gosub 51500 : gosub 51000
51020  print#ou,chr$(254);chr$(i1$);chr$(i2$);
51030  print#pr,chr$(13);str$(i1+i2*256);
51040  gosub 51500 : gosub 51500 : return
```

It seems to be the fate of programming that you need a thousand words to describe a few lines of code, but it is often impossible to make sense of a program without the explanation.

One more observation:

In order to make the program clearer, part of the subroutines which will be called will often be placed at the end of the program. This is not very efficent for programs which will be interpreted because a call to these subroutines takes longer than it would if they where not at the end. This effect does not occur in compiled programs and this problem does not occur in assembly language programs.

The program which replaces the syntactic analysis:

```
2000    gosub 50000 : if t$<>chr$(255) then 2000
2005    rem 255 signals end of program
2010    print#pr,chr$(13)"Lexical analysis finished."
2020    close pr
2030    close in
2040    close ou
2050    close be
2060    end
```

This program searches for new tokens until the condition t$=chr$(255) is fulfilled. This character signifies the end

of the program for us. The opened files are then closed to bring the program to a well-defined end.

Back to the task of "Reading program" and "Outputting status line:"

```
50000   rem  lexical analysis
50010   if i1=0 then if i2=0 then if i3=0 then t$=chr$(255)
: return
```

If i1 and i2 and i3 are all zero, this means that we have reached the end of the program. The linking address is zero.

```
50020   if i1=0 then gosub 51000
```

The end of the line is reached and we branch to the appropriate subroutine.

```
50030   if i1>127 and i1<204 then i1$=ba$(i1-128)
```

If i1 is greater than 127 and less than 204, then i1$ is a tokenized BASIC keyword which it must be expanded into text.

i1$ is replaced with the text stored in ba$.

```
50200   print#pr, i1$; : gosub 51500
50210   return
```

We'll leave some line numbers open to leave room for the rest of the tasks of the scanner. You can now run the program which we have developed up to this point. You will then receive a listing of your MINI program.

Removing comment lines:

Next we want to remove all of the comment lines.

Comments in MINI are designated through two successive minus signs. The rest of the line can be ignored.

The following line finds the comment lines:

```
50040   if i1=171 then if i2=171 then gosub 51700 : goto
  50000
```

The following lines read to the end of the program line and outputs the line on the screen:

```
51700   rem  read line to end
51705   :
51710   print#pr,i1$; : gosub 51500 : if i1=0 then return
51720   if i1>127 and i1<204 then i1$=ba$(i1-128)
51730   goto 51710
51740   :
```

Skipping extra spaces:

Any more than one successive space in a MINI program is superfluous. We remove these extra spaces with the following line:

```
50050   if i1$=" "then if i2$=" " then print#pr,i1$; : gosub
51500 : goto 50000
```

Tokens are usually composed of several characters. We recognize the end of a token through a delimiter. This means that we add i1$ to t$ until we encounter a delimiter in i2$.

```
50190   t$=t$+i1$
```

We expand line 2000 as follows:

```
2000    t$=" " : gosub 50000  ... .
```

Now we need two more subroutines: one to find out if a token is a MINI keyword, and another to put strings together since upper case may not be converted to lower case nor blanks removed from strings.

We will write the first subroutine at line 52000 and the second at line 53000.

```
50070   if i1=34 then gosub 53000 : goto 50000
```

If the current character is a quotation mark, we must skip over the string.

```
53000   rem  skip string
53010   t$=t$+i1$ : gosub 51500 : if i1<>34 then goto 53010
53020   t$=t$+i1$ : return
53030   :
```

Delimiters in MINI are =,/,<,>,.,,,,',end-of-line,",space

There are various cases in which delimiters can occur:

1) We have read a character string followed by a period.

```
50080   if i1=34 then if i2$="." then 50200
50085   if i1=32 then if i2=34 then goto 50130
```

2) A delimiter also occurs in i2$:

```
50090   if i2$="=" or i2$="/" or i2$="<" or i2$=">" then
        gosub 52000 : goto 50200
50100   if i2$="." or i2$=" " or i2=0 or i2=34 then gosub
        52000 : goto 50200
50110   if i2$="," then gosub 52000 : goto 50200
```

3) A delimiter is present in i1$:

```
50126   if i1$="." or i1$="," or i1$="=" then t$=i1$ : goto
        50200
50128   if i1$="/" or i1$="<" or i1$=">" then t$=i1$ : goto
        50200
```

4) i1$ is a blank:

```
50130   if i1$=" " then print#pr," "; : gosub 51500 : goto
        50000
```

Now all we lack is the subroutine at line 52000 which will recognize whether or not a token is a MINI keyword. If the token is a keyword, we want to return a number in t which represents the tokenized form of the keyword. We will assign numbers beginning at 128 to the MINI keywords.

Here is the list of keywords and the numbers we will assign to them:

Value	Keyword
128	add
129	outputdevice
130	exit
131	begin
132	generate
133	curscol
134	curslin
135	then
136	by
137	divide
138	float
139	background
140	get
141	is
142	empty
143	with
144	multiply
145	tovar
146	pend
147	power
148	program
149	border
150	call
151	type
152	clearscreen
153	loop
154	lend
155	else
156	jump
157	jumplabel
158	subtract
159	over

```
160       assign
161       srend
162       subroutine
163       from
164       linefeed
165       endif
166       if
167       write
168       writeln
169       writeas
170       and
```

We can simply compare t$ with each keyword.

We must put the MINI keywords in memory. This is not absolutely necessary at the moment, but since we will use them again in the syntactic analysis for error handling, we will save memory space with an array.

There are a total of 49 MINI keywords.

```
170     dim mi$(48) : rem text for mini words

1245    gosub 58000

58000     REM MINI WORDS
58005     :
58008     MI$(0)="ADD"
58009     MI$(42)="AND"
58010     MI$(32)="ASSIGN"
58020     MI$(11)="BACKGROUND"
58030     MI$(3)="BEGIN"
58035     MI$(21)="BORDER"
58040     MI$(8)="BY"
```

```
58042     MI$(22)="CALL"
58045     MI$(24)="CLEARSCREEN"
58050     MI$(5)="CURSCOL"
58060     MI$(6)="CURSLIN"
58090     MI$(9)="DIVIDE"
58091     MI$(27)="ELSE"
58093     MI$(14)="EMPTY"
58094     MI$(37)="ENDIF"
58099     MI$(2)="EXIT"
58100     MI$(10)="FLOAT"
58112     MI$(4)="GENERATE"
58120     MI$(12)="GET"
58121     MI$(38)="IF"
58130     MI$(13)="IS"
58132     MI$(28)="JUMP"
58134     MI$(29)="JUMPLABEL"
58140     MI$(26)="LEND"
58145     MI$(36)="LINEFEED"
58150     MI$(25)="LOOP"
58160     MI$(16)="MULTIPLY"
58161     MI$(35)="OF"
58162     MI$(31)="OVER"
58163     MI$(1)="OUTPUTDEVICE"
58180     MI$(18)="PEND"
58190     MI$(19)="POWER"
58200     MI$(20)="PROGRAM"
58270     MI$(33)="SREND"
58280     MI$(34)="SUBROUTINE"
58290     MI$(30)="SUBTRACT"
58292     MI$(7)="THEN"
58296     MI$(17)="TOVAR"
58299     MI$(23)="TYPE"
58388     MI$(15)="WITH"
58390     MI$(39)="WRITE"
```

```
58400     MI$(40)="WRITELN"
58410     MI$(41)="WRITEAS"
58430     MI$(43)="FOR"
58440     MI$(44)="TO"
58450     MI$(45)="REPEAT"
58460     MI$(46)="/="
58470     MI$(47)="<="
58480     MI$(48)=">="
58490     RETURN

52000   rem  is t$ a token?
52010   :
52020   t$=t$+i1$ : t=0 : a=asc(left$(t$,1))
```

t should contain the ASCII value of the token for the MINI keyword. If the value of t does not change during the course of the subroutine, no keyword is stored in t$. a should contain the ASCII value of the first character of t$.

```
180     t$=" ":t=0: rem variables for token
190     a=0       : rem temp variable for lexical analysis
```

Is a combination of letters stored in t$ at all?

```
52030   if a<65 or a>90 then return
```

Next we call the subroutines for finding the keywords.

```
52035   d=0:c=48:gosub52700: return
52700   for d=b to c
52710   if t$=mi$(d) then t=128+d
52720   next d
52800   if t=0 then return
52810   t$=chr$(t) : return
```

If t=0, t$ is not a keyword, otherwise the tokenized keyword is stored in t$.

The MINI words are stored in mi$. We have entries in mi$ for each of the letters listed above. The lower boundary of these entries is stored in b and the upper boundary in c. We must not forget to initialize b, c, and d.

```
190     a=0:b=0:c=0:d=0:rem temp variables lex. analysis
```

We are now finished with the lexical analysis. The complete listing of the parser can be found later in the book and should be used as the authority if any discrepancies between that listing and the lines given here are found.

4. THE SYNTACTIC ANALYSIS

The question which the syntactic analysis must answer is the following: Does our program follow all the rules of the MINI grammar? We view the MINI program as a sentence in the MINI language and we must check to see if the sentence is a valid MINI sentence. We can compare the task with the problem which results when we want to know if a sentence follows the rules of English grammar. Let's take a look at an example of the analysis of an English sentence and then transfer what we know in that case to our MINI language.

We want to analyze the sentence "The dog barks." We'll abbreviate the rule for an English sentence to the following:

```
sentence ::= subject predicate "."
```

We can read this rule as follows:

A sentence consists of a subject, a predicate, and a period. Now we know what a sentence consists of, but what is a subject and what is a predicate? We know what a period is. It already appears in our sentence.

We want to call symbols (the period is also considered a symbol) which can occur in a sentence of the language, terminal symbols, in contrast to those which we must subject to further analysis in order to reduce them to terminal symbols. We will call this type non-terminal symbols.

```
subject ::= article noun
article ::= 'the'
noun ::= 'car'
     :   'can'
     :   'dog'
```

A subject should be constructed out of an article and a noun. A noun can be "car," "can," or "dog." We have various alternatives for the noun which we show through the use of the colon.

```
predicate ::= verb
verb      ::= 'barks'
          :   'shines'
          :   'rolls'
```

For the predicate we may choose only a verb and for the verb we have 'barks,' 'shines,' and 'rolls' at our disposal.

Let us analyze the following sentence:

```
The       dog       barks       .
  subject          predicate    '.'
article   noun       verb       '.'
'The'     'dog'     'barks'     '.'
```

We have determined that "The dog barks." is a valid English sentence. The grammar which we developed above is very simple and we can proceed through the analysis by "trying out" all of the possibilities. This can take a long time even if we use a computer to help us. Here we must develop an algorithm for the procedure. First, however, we will state the grammar for MINI. To do this we must take a careful look at the MINI instructions.

The grammar:

Every MINI program has the following basic construction:

```
program   program_identifier   is
--
-- definition block
--
begin
--
-- instruction block
--
pend program_identifier.
--
--
subroutines
```

Let's write this as a rule:

```
program ::= 'program' program_identifier 'is'
            definition_block 'begin'
            instruction_block 'pend'
            program_identifier '.' subroutine EOF
```

This defines the rule "program". The following rules are still undefined:

- program_identifier
- definition_block
- instruction_block
- subroutine

Once we have defined these rules and the new rules which occu within these, our grammar is finished.

A few words about the notation:

The words in apostrophes are terminal symbols. All non-terminal symbols which end in 'identifier' we will define in common. The prefix (program, loop, subroutine, etc.) serves only for syntactic distinction. All identifiers have the same syntactic construction while program, subroutine, and so on serve as a prefix indicating the type of identifier. We will need this distinction for the semantic analysis since a program identifier may not be used as a variable identifier, for example.

The non-terminal symbol EOF means End Of File, which means nothing more than the physical end of the file.

The introduction of EOF seems somewhat elaborate, but we will use this symbol in the analysis.

```
subroutine ::= 'subroutine' subroutine_identifier
               'is' instruction_block 'srend'
               subroutine_identifier '.'
               subroutine
             : E
```

With this rule we have a choice. The upper-case "E" indicates that this rule can also be empty, which means we must apply this rule, but it need not be present in our MINI program. A MINI program can contain subroutines, but none have to be present.

Since the rule 'program' is valid for every MINI program and the rule 'subroutine' also occurs in this rule, the rule 'subroutine' must be applied.

If there are no subroutines in our MINI program, the 'subroutine' rule must take this into account. In this case we select the alternative 'E'.

The first alternative describes how a subroutine must be constructed and ends again with the call to the rule 'subroutine'. The rule calls itself again, because it may be that several subroutines are present. If only one is present, the alternative with the 'E' is taken in the second pass. This rule adds no new non-terminal symbols.

Now we turn to the identifiers:

```
identifier   ::= letter identifier_1
identifier_1 ::= letter identifier_1
               :  E

letter       ::= 'a'
               :   'b'
               :   'c'
               :    .
               :    .
               :   'x'
               :   'y'
               :   'z'
```

According to this rule, an identifier may consist of any number of letter characters. We have placed a limit of 80 characters on the identifiers. In MINI we cannot input an identifier which contains more than 80 characters.

```
definition_block        ::= variable_declaration
                          : E
variable_declaration    ::= 'float' variable_identifier
                                  variable_declaration_2 '.'
                                  definition_block
variable_declaration_2 ::= ',' variable_identifier
                                 variable_declaration_2
                          : E
```

Explanation:

A definition block may be empty. If its not, it consists of variable declarations. The rule 'variable_declaration' calls definition_block at its end because an arbitrary number of variable declarations may be appended. Several variables, separated by commas, may be declared within a definition. This is the reason for the introduction of 'variable_declaration_2.'

Now we come to the instruction block:

This is a bit more complex because all possibilities for forming instructions must be taken into account.

```
instruction_block     ::= instruction instruction_sequence
instruction_sequence ::= instruction instruction_sequence
                       :  E
instruction           ::= 'empty' '.'
                       :  'write' write_1 '.'
                       :  'writeln' write_1 '.'
                       :  'linefeed' '.'
                       :  'clearscreen' '.'
                       :  'curscol' integer '.'
                       :  'curslin' integer '.'
```

```
                : 'write as' integer '.'
                : 'outputdevice' device_identifier
                  '.'
                : 'border' color_identifier '.'
                : 'background' color_identifier '.'
                : 'type' color_identifier '.'
                : 'get' variable_identifier '.'
                : 'assign' assign_1 '.'
                : 'add' variable_identifier 'and'
                  variable_identifier
                  'tovar' variable_identifier '.'
                : 'subtract' variable_identifier
                  'from' variable_identifier 'tovar'
                  variable_identifier '.'
                : 'multiply' variable_identifier
                  'with' variable_identifier 'tovar'
                  variable_identifier '.'
                : 'divide' variable_identifier
                  'by' variable_identifier 'tovar'
                  variable_identifier '.'
                : 'power' variable_identifier 'with'
                  variable_identifier 'tovar'
                  variable_identifier '.'
                : 'generate' function_identifier
                  'from' variable_identifier
                  generate_1 '.'
                : 'if' condition 'then'
                  instruction_block 'else'
                  instruction_block 'endif' '.'
                : 'loop' loop_identifier 'over'
                  instruction_block '.'
                : 'exit' loop_identifier 'if'
                  condition '.'
                : 'for' variable_identifier 'from'
```

```
                        variable_identifier
                        'to' variable_identifier
                        'repeat' instruction_block
                        'lend' '.'
                    :   'jumplabel' jump_identifier '.'
                    :   'jump' jump_identifier '.'
                    :   'call' subroutine_identifier '.'
```

The following rules are still undefined:

- string
- integer
- assign_1
- generate_1
- condition
- write_1

```
assign_1    ::= floating_point_number 'tovar'
                variable_identifier 'tovar'
                variable_identifier
generate-1  ::= 'tovar' variable_identifier
             :  E
condition   ::= variable_identifier operator
                variable_identifier
operator    ::= '='
             :  '/='
             :  '<'
             :  '<='
             :  '>'
             :  '>='
string      ::= character string
             :  E
```

We will not define 'character' any farther than to say that it stands for any character from the keyboard, except for the quotation mark.

Still remaining are integer, floating_point_number, and write-1.

```
integer                 ::= digit integer_1
integer_1               ::= digit integer_1
                          :  E
digit                   ::= '0'
                          :  '1'
                          :  '2'
                          :   .
                          :  '9'
floating_point_number ::= integer floating_point_number_1
floating_point_number-1 ::= '.' integer
                              floating_point_number_2
                          :  exponent
                          :  E
floating_point_number-2 ::= exponent
                          :  E
exponent                ::= 'e' exponent_1
exponent_1              ::= '+' integer
                          :  '-' integer
                          :  integer
write_1                 ::= '"' string '"'
                          :  variable_identifier
```

Now we have completely described the language MINI through a grammar. To work with the grammar, we'll number the rules and the alternatives and create an index.

4.1 THE RULES OF GRAMMAR

```
01  program ::= 'program' program_identifier 'is'
                 definition_block 'begin'
                 instruction_block 'pend'
                 program_identifier '.' subroutine EOF

02  subroutine ::= 'subroutine' subroutine_identifier
                    'is' instruction_block 'srend'
                     subroutine_identifier '.'
                     subroutine
03                :  E
04  identifier     ::= letter identifier_1
04a identifier_1   ::= letter identifier_1
05                  :  E
06  letter         ::= 'a'
                    :  'b'
                    :  'c'
                    :   .
                    :   .
                    :  'x'
                    :  'y'
                    :  'z'
07  definition_block        ::= variable_declaration
08                           :  E
09  variable_declaration    ::= 'float' variable_identifier
                                 variable_declaration_2 '.'
                                 definition_block
10  variable_declaration_2 ::= ',' variable_identifier
                                 variable_declaration_2
11                           :  E
12  instruction_block     ::= instruction instruction_sequence
13  instruction_sequence ::= instruction instruction_sequence
```

```
14                          :  E
15  instruction             ::= 'empty' '.'
16                          :  'write' write_1 '.'
17                          :  'writeln' write_1 '.'
18                          :  'linefeed' '.'
19                          :  'clearscreen' '.'
20                          :  'curscol' integer '.'
21                          :  'curslin' integer '.'
22                          :  'write as' integer '.'
23                          :  'outputdevice' device_identifier
                               '.'
24                          :  'border' color_identifier '.'
25                          :  'background' color_identifier '.'
26                          :  'type' color_identifier '.'
27                          :  'get' variable_identifier '.'
28                          :  'assign' assign_1 '.'
29                          :  'add' variable_identifier 'and'
                               variable_identifier
                               'tovar' variable_identifier '.'
30                          :  'subtract' variable_identifier
                               'from' variable_identifier 'tovar
                               variable_identifier '.'
31                          :  'multiply' variable_identifier
                               'with' variable_identifier 'tovar
                               variable_identifier '.'
32                          :  'divide' variable_identifier
                               'by' variable_identifier 'tovar'
                               variable_identifier '.'
33                          :  'power' variable_identifier 'with
                               variable_identifier 'tovar'
                               variable_identifier '.'
34                          :  'generate' function_identifier
                               'from' variable_identifier
                               generate_1 '.'
```

```
35                          :  'if' condition 'then'
                               instruction_block 'else'
                               instruction_block 'endif' '.'
36                          :  'loop' loop_identifier 'over'
                               instruction_block '.'
37                          :  'exit' loop_identifier 'if'
                               condition '.'
38                          :  'for' variable_identifier 'from'
                               variable_identifier
                               'to' variable_identifier
                               'repeat' instruction_block
                               'lend' '.'
39                          :  'jumplabel' jump_identifier '.'
40                          :  'jump' jump_identifier '.'
41                          :  'call' subroutine_identifier '.'
42  assign_1     ::= floating_point_number 'tovar'
                     variable_identifier
43                :  variable_identifier 'tovar'
                     variable_identifier
44  generate_1   ::= 'tovar' variable_identifier
45                :  E
46  condition    ::= variable_identifier operator
                     variable_identifier
47  operator     ::= '='
48                :  '/='
49                :  '<'
50                :  '<='
51                :  '>'
52                :  '>='
53  string       ::= character string
54                :  E
55  integer              ::= digit integer_1
55a integer_1             ::= digit integer_1
56                         :  E
```

```
57  digit                    ::= '0'
                              :  '1'
                              :  '2'
                              :   .
                              :  '9'
58  floating_point_number ::= integer floating_point_number_1
59  floating_point_number-1  ::= '.' integer
                                    floating_point_number_2
60                             :  exponent
61                             :  E
62  floating_point_number-2  ::= exponent
63                             :  E
64  exponent                 ::= 'e' exponent_1
65  exponent_1               ::= '+' integer
66                             :  '-' integer
67                             :  integer
68  write_1                  ::= '"' string '"'
69                             :  variable_identifier
```

4.2 GRAMMAR INDEX

The index is constructed as follows:

At the start of the line is the key number of the rule. This appears if you output the stack during the syntactic analysis. We will come back to this later.

The number after this is the number of the rule with which the rule is defined in the grammar. The numbers after the slash indicate the rules in the grammar in which this rule is used.

```
01  instruction            15/ 12, 3
02  instruction_block      12/ 1, 2, 35, 36, 38
03  instruction_sequence   13/ 12, 13
04  condition              46/ 35, 37
05  identifier               4/ 1, 2, 4, 9, 10, 23, 24, 25, 26,
                               27, 29, 30, 31, 32, 33, 34, 36,
                               37, 38, 39, 40, 41, 42, 43, 44,
                               46
24  identifier_1           4a/ 4, 4a
06  generate_1             44/ 34
07  letter                  6/ 4, 4a
08  definition_block        7/ 1
09  exponent               64/ 60, 62
10  exponent_1             65/ 64
11  floating_point_number    58/ 42
12  floating_point_number_1 59/ 58
13  floating_point_number_2 62/ 59
14  integer                 55/ 20, 21, 22, 58, 59, 65, 66, 67,
23  integer_1              55a/ 55, 55a
15  operator               47/ 46
16  program                 1/
17  assign_1               42/ 28
18  subroutine              2/ 1, 2
19  variable_declaration    9/ 7, 9
20  variable_declaration_2 10/ 9, 10
21  string                 53/ 53, 68
25  write_1                68/ 16, 17
22  digit                  57/ 55
26  character                / 53
```

4.3 HOW DOES THE COMPILER WORK WITH THE GRAMMAR?

In this section and the next we want to finish the preparations for writing the syntactic analysis. A program for syntactic analysis is called a parser. At first we will assume that the MINI program to be checked is syntactically correct and first concern ourselves with the analysis technique before we turn to error handling.

As we said before, the parser is oriented to the grammar and the index. In order to become acquainted with the method of analysis, we will parse the following program:

```
10 program a is
20 begin
30 empty
40 pend a.
```

We know that every MINI program must satisfy rule 01 'program'. This rule encompasses the entire "development" of a MINI program, no matter how complex it may be. The parser takes this rule as the first and "notes" it. The question naturally arise: How does the parser note this rule?

The parser stores the rule in something called a stack.

Next question: What is a stack and how can one picture it?

Let us imagine an elevator in a skyscraper which works on the following principle:

There are only two buttons in the elevator.
Button 1: One floor up.
Button 2: One floor down.

The starting point for a trip is the first floor. From here we can travel one floor up and deposit some information. We forget this "deposited" information as soon as we leave the floor. If we travel higher, this information remains on the floor, but we can retrieve this information only when we return to that floor. Information on higher floors does not exist for us. Furthermore, we can travel only one floor up once we have placed information on the current floor. All lower floors are filled with information.

This "construction" serves only to allow us to save information which we will require at a later point in time.

Let us make note (save) of rule 01 'program.'

It may surprise you that the parser saves this rule from "right to left."

Floor	Information
Floor 1	EOF
Floor 2	subroutine
Floor 3	'.'
Floor 4	program_identifier
Floor 5	'pend'
Floor 6	instruction_block
Floor 7	'begin'
Floor 8	definition_block
Floor 9	'is'
Floor 10	program_identifier
Floor 11	'program'

We find ourselves on the eleventh floor and have the information 'program' at our disposal.

What do we do with this information?

We know that 'program' is a terminal symbol. Further, we know that every MINI program must begin with 'program'. We know this precisely because the information on the "top floor" is a terminal symbol.

It is a good idea to give some thought to these sentences. They contain half of the "secret" of the analysis.

4.4 THE ANALYSIS STRATEGY

During the preparations for analysis, the scanner is instructed to fetch the first token in the MINI program. The first token in our program is 'program'. The parser has only to compare the terminal symbol 'program' with the token 'program'. Since the two are identical, the parser knows that our program corresponds to the syntactical rules of MINI up to the word 'program'. It can then forget the token 'program' and request the next token from the scanner. It contains "a". Now it moves one floor down because the information 'program' has become superfluous and it needs information about the possible future development of MINI programs. It reads "program_identifier" in the syntactic sense of "identifier." But "identifier" is a non-terminal symbol, so it can't compare it with "a". "Identifier" specifies a rule.

What strategy does it employ?

It replaces the name "identifier" with the corresponding rules until a terminal symbol is again the top element on the stack.

The rule for "identifier":

```
04  identifier   ::= letter identifier
05                :  E
```

Now the parser has two possibilities:

It either takes "letter identifier" or "E". How does it decide?

For each possibility it has a list in which is itemized which alternative must be selected based on the current token. We will discuss later how this list is generated.

The parser selects 04 "letter identifier". It clears level 9 and enters the combination:

level 9	identifier
level 10	letter

The information at the top of the stack is letter. By letter we mean all lower-case letters, and since "a" is a lower-case letter, the parser can replace letter with "a" and compare it with the token "a". Since these two agree, it requests the next token and moves one level down. The next token is 'is'. At level 9 is "identifier".

The parser now has the choice between "letter identifier" and "E". This time it selects "E". "E" means that it is to move one level down. At level 8 it finds the information "is", a terminal symbol. It compares this with the next token, and since agreement is found, proceeds to...

This "game" continues until the scanner returns "End Of File" (end of the MINI program) and the state of the parser confirms this by being at level 0. Then our MINI program is syntactically correct and the parser has completed its task.

We hope that it is clear how the parser operates. You should try to parse the rest of the program yourself. If you are not sure which alternative to take for a given rule, just try one out.

As you have no doubt noticed, I have not specified the rule "identifier" as we declared it in the grammar. As practice, take the correct rule and parse the program again.

Three questions are still unanswered:

- How do we create the list which controls the selection of the alternatives?
- How does the parser handle errors in the program?
- What must the parser do so that programs can be compiled into machine language.

4.5 SELECTION OF THE ALTERNATIVES

The parser proceeds through the analysis as follows:

It searches for the rule which it must use based on the information at the top of the stack. Then it looks at the actual token and chooses the alternative with the help of a list. There is a list of possible tokens for every token.

How do we obtain the lists?

The tokens in our program are always terminal symbols. We must search for the terminal symbols in each alternative which can occur first upon the use of the alternative. This is quickly said, but the task can become very complex, especially when one has a large grammar to process in which few alternatives begin with terminal symbols. In some books there are guidelines which one can follow and which always lead to success. In principle these are only suggestions which can serve to write a program which finds the terminal symbols in question. Instead of using some mindless procedure we want to consider which terminal symbols can occur for each alternative.

Let us begin with rule 01 "program" for the sake of completeness, although the rule will never be selected since it is assumed at the start of the parsing.

Alternative	Terminal symbol
01	program
02	subroutine
03	EOF

If an alternative is empty, we must check to see in which rules the rule in question occurs and what the first terminal symbols can be. Example: 'subroutine' occurs only in the rule 'program'. The only terminal symbol which can follow is the "EOF" marker. From this it follows that if the top stack symbol is 'subroutine' and the current token is 'subroutine', alternative 02 is chosen. If the top stack symbol is 'subroutine' and the current token "EOF", rule 03 is applied.

```
04      letter
04a     letter
05      is, '.', ',', and, tovar, from, with, by,
        over, if, to, repeat, '=', '/=', '<', '<=',
        '>', '>=', then
```

Try to follow how we obtained the terminal symbols for rule 05! You will want to use the index to help you.

```
06      letter
07      float
08      begin
09      float
10      ','
11      '.'
12      empty, write, writeln, linefeed, clearscreen,
        curscol, curslin, writeas, outputdevice,
        border, background, type, get, assign, add,
        subtract, multiply, divide, power, generate,
        if, loop, exit, for, jumplabel, jump, call
13      see 12
14      pend, srend, else, endif, '.', lend
15      empty
16      write
```

```
17        writeln
18        linefeed
19        clearscreen
20        curscol
21        curslin
22        writeas
23        outputdevice
24        border
25        background
26        type
27        get
28        assign
29        add
30        subtract
31        multiply
32        divide
33        power
34        generate
35        if
36        loop
37        exit
38        for
39        jumplabel
40        jump
41        call
42        digit
43        letter
44        tovar
45        '.'
46        letter
47        '='
48        '/='
49        '<'
53        character except for '"'
```

```
54      '"'
55      digit
55a     digit
56      '.', 'e', tovar
57      digit
58      digit
59      '.'
60      'e'
61      tovar
62      'e'
63      tovar
64      'e'
65      '+'
66      '-'
67      digit
68      '"'
69      letter
```

Creating this list appears to be a confusing matter, but it is really quite simple. We want to clarify with another example:

Alternative 56:

The alternative is empty, raising the question: What can integer_1 follow?

Integer_1 occurs only in rule 'integer' and there as the last element. This raises the question: What can integer follow? According to the index, integer occurs in alternatives 20, 21, 23, 58, 59, 65, 66, 67. In 20, 21, and 23 the period follows integer.

In 58 integer comes before the rule floating_point_number_1. The successors of integer resulting from this rule come from the first terminal symbols and, since floating_point_number_1 also includes empty alternative 61, the succeeding terminal symbols from floating_point_number_1 as well. We refer to the first terminal symbols as "the first" and the succeeding terminal symbols with "the successors."

The first of floating_point_number_1 are '.' and the first of exponent 'e'. The successors of floating_point_number_1 result from successors of floating_point_number because floating_point_number_1 occurs only as the last element in floating_point_number. But floating_point_number occurs only in alternative 42. Successor is 'tovar'.

Temporary result: Up to now we have '.', 'e', and 'tovar'. This leaves 65, 66, and 67 as alternatives to consider. All three stop with integer and belong to exponent_1.

Question: What are the successors of exponent_1?
Answer: The successors of exponent!
Question: What are the successors of exponent?
Answer: The successors of floating_point_number_2 and floating_point_number_1!
Question: What are the successors of floating_point_number_2?
Answer: The successors of floating_point_number_1!

One can assemble the lists with simple considerations. It will become clear however, that one must consider techniques for larger grammars in order to reach our goal without error. We have not yet mentioned that the parser can only tell exactly which alternative it should choose when the lists for the alternatives to a rule contain all of the

various tokens. If identical tokens occur in the lists for the alternatives to a rule, we must either change our grammar or we must find a different analysis method.

4.6 HANDLING PROGRAM ERRORS

Up to now we have assumed that our MINI program was syntactically correct. If there are syntactic errors in a program, we have two possible ways of handling them.

A) We output the error and halt the syntactic analysis.
B) We try to resume the syntactic analysis in order to check the remainder of the program.

We will not pursue option number A. Imagine a program which contains 20 errors. It would take at least 21 compilation attempts until we know that we have a syntactically correct program.

We will try to implement option B. How does the parser notice that a program contains an error? This can occur in two situations:

1) The top stack symbol is a terminal symbol.
   The current token does not agree with the terminal symbol.
   Example: We made a typing error and entered "progrom" instead of "program"

2) The parser searches for the fitting alternative in a rule and does not find the current token in any of the lists.
   Example: We forgot to specify a program identifier:
   program is ... .
   The current token is then "is", but the parser is expecting an identifier.

Let us consider what strategies we want to use in 1) and 2):

There will be a section of the program which we will not be able to check because the stack contents do not agree with the program structure. We must try to get the stack and the program to agree with each other again so that we can continue with the analysis.

In situation 1)

Here we know exactly what symbol the parser is expecting and what it found instead. We can produce a very precise error message. In addition, we can try to find the expected symbol in the program by requesting new tokens from the scanner. This can cause trouble if the error is typographical because we may not find the expected symbol. In order to prevent this we search no farther than the next '.' which in MINI signals the end of the instruction, which we can view as a small syntactic entity. We must also adjust the stack accordingly by clearing it from the top down until we come to a '.'. After doing so, both stack and program will agree with each other and the analysis can continue.

In situation 2)

Here we can specify a list of tokens which would have assisted in continuing the syntactic analysis.

At this point we want to proceed as follows:

We travel down in the stack and search for the next terminal symbol and continue to read our program in order to determine if this terminal symbol occurs. We read no farther than the next period, however. If we read from the last period, we travel down on the stack until we find the next period. This causes "agreement" again.

You may say that these are very crude methods which will not always lead to success. In addition, we have completely disregarded the possibility of the computer performing any error correction. We agree, but we will implement the error handling in our compiler in such a manner that you can easily put your own ideas into practice, if enough memory remains. There are compilers in which the error correction has the same scope and range as the analysis itself. You must make certain compromises on this point when dealing with microcomputers, however.

Additional tasks:

We can now determine whether any MINI program is syntactically correct or not. After the syntactic analysis comes the semantic analysis and then the code generation. In order for the semantic analysis to take place, we must pass information on to the semantic analysis during the syntactic analysis. Since we establish the structure of the program in the syntactic analysis, we should use this information. The structure of the program can be determined from the path which the parser takes through the grammar. We will use the grammar as a source of information. To do this we will extend the grammar with symbols which we will insert into the grammar but which will not affect the syntactic analysis. These semantic symbols receive a special designation. If a semantic symbol occurs as the top stack element, the parser recognizes this and branches to a subroutine which can process this symbol. The symbol will be removed from the stack and the parser continues with its work.

The subroutines for the semantic symbols do one of two things:

1) They collect information.
2) They output information.

The output of information is directed to the file in which we have already written the line numbers. This file collects all of the information which we need to convert our MINI program into a machine language program. What information do we want to output?

We must go through the grammar again and consider what we want to do for each alternative.

We will put the semantic symbols in parentheses.

```
01 program ::= 'program' (program identifier definition)
               program_identifier 'is' (definition section
               start)
               definition_block (definition section end)
               'begin' instruction_block 'pend' (program
               identifier check) program_identifier '.'
               subroutine (program end) EOF
```

The following happens when this rule is processed:

Information output: "program identifier definition", together with the identifier in question (see 04, 05). This will be entered in a list and during the semantic analysis designated as the program identifier.

"Definition section start" causes all of the following identifiers to be entered in a list.

After "definition section end" no more variables may be declared and all variables which occur must be contained in the list.

The identifier after "program identifier check" must be declared in the list as program identifier.

The specific tasks of the semantic analysis and code generation will be explained in the next section.

We'll now expand our grammar so that we can write the program for the syntactic analysis.

```
02  subroutine          ::= 'subroutine' (subroutine identifier
                            definition) subroutine_identifier
                            'is' instruction_block 'srend'
                            (subroutine identifier check)
                            subroutine_identifier '.' subroutine
03                       :  E
04  identifier          ::= (save letter) letter
                            identifier_1
04a identifier_1        ::= (save letter) letter
                            identifier_1
05                       :  (output identifier) E
06  letter              ::= 'a', 'b', ..., 'z'
07  definition_block    ::= variable_definition
08                       :  E
09  variable_definition ::= 'float' variable_identifier
                               variable-definition_2
                               '.' definition_block
10  variable_definition_2 ::= ',' variable_identifier
                                 variable_definition_2
11                          :  E
```

```
12  instruction_block ::= instruction instruction_sequence
13  instruction_sequence ::= instruction
                             instruction_sequence
14                        :  E
15  instruction      ::= 'empty' '.'
16                    :  (output) 'write' write_1 '.'
17                    :  (output with linefeed) 'writeln'
                         write_1 '.'
18                    :  'linefeed' '.'
19                    :  (clearscreen) 'clearscreen' '.'
20                    :  (cursor column) 'curscol' '.'
21                    :  (cursor line) 'curslin' integer '.'
22                    :  (output ASCII) 'write as' integer '.'
23                    :  (device) 'outputdevice'
                         device_identifier '.'
24                    :  (border) 'border' color_identifier
                         '.'
25                    :  (background) 'background'
                         color_identifier '.'
26                     :  (type) 'type' color_identifier '.'
27                    :  (input) 'get' variable_identifier
                         '.'
28                    :  (assign) 'assign' assign_1 '.'
29                    :  (add) 'add' variable_identifier
                         'and' variable_identifier 'tovar'
                         variable_identifier '.'
30                    :  (subtract) 'subtract'
                         variable_identifier 'from'
                         variable_identifier '.'
31                    :  (multiply) 'multiply'
                         variable_identifier 'with'
                         variable_identifier 'tovar'
                         variable_identifier '.'
32                    :  (divide) 'divide'
```

```
                        variable_identifier 'by'
                        variable_identifier '.'
33                   :  (power) 'power' variable_identifier
                        'with' variable_identifier 'tovar'
                        variable_identifier '.'
34                   :   (generate) 'generate'
                        function_identifier 'from'
                        variable_identifier generate_1
                        (period) '.'
35                   :   (conditional) 'if' condition
                        'then' instruction_block (else)
                        instruction_block 'endif'
                        (conditional end) '.'
36                   :  'loop' (infinite loop) 'over'
                        instruction_block 'lend'
                        (infinite loop end)
                        loop_identifier '.'
37                   :   (exit) 'exit' loop_identifier
                        'if' condition '.'
38                   :   (index loop) 'for'
                        variable_identifier 'from'
                        variable_identifier 'to'
                        variable_identifier 'repeat'
                        (loop start)
40                     :  (jump) 'jump' jump_identifier '.'
41                   :   (call) 'call' subroutine_identifier
                        '.'
42   assign_1       ::= floating_point_number 'tovar'
                        variable_identifier
43                   :  (variable) variable_identifier
                        'tovar' variable_identifier
44   generate_1     ::=  (generate_1) 'tovar'
                        variable_identifier
45                   :  E
```

```
46  condition          ::= variable_identifier operator
                           variable_identifier
47  operator           ::= (operator=) '='
48                      :  (operator/=) '/='
49                      :  (operator<) '<'
50                      :  (operator<=) '<='
51                      :  (operator>) '>'
52                      :  (operator>=) '>='
53  string             ::= (save string) character string
54                      :  (output string) E
55  integer            ::= (save digit) digit integer_1
55a integer_1           ::= (save digit) digit integer-1
56                      :  (output integer) E
57  digit              ::= '0','1',...,'9'
58  floating_point_number ::= integer floating_point_num-
                              ber_1
59  floating_point_number_1 ::= (decimal places) '.'
                                integer floating_point_num-
                                ber_2
60                           :  exponent
61                           :  (no exponent) E
62  floating_point_number_2 ::= exponent
63                           :  (no exponent) E
64  exponent           ::= (exponent) 'e' exponent_1
65  exponent_1         ::= (plus) '+' integer
66                      :  (minus) '-' integer
67                      :  integer
68                     ::= '"' string '"'
69                      :  variable_identifier
```

We cannot give the parser the grammar in this form. We will first code the terminal symbols, the non-terminal symbols, and the semantic symbols into numbers.

We'll take the numbers for the terminal symbols from the section on the lexical analysis. For the non-terminal symbols, which include numbers from the index to the grammar and the numbers for the semantic symbols, we offer the following list:

Number	Symbol
1	program identifier definition
2	definition section start
3	definition section end
4	program identifier check
5	program end
6	subroutine identifier definition
7	subroutine identifier check
8	output
9	output with linefeed
10	linefeed
11	clearscreen
12	cursor column
13	cursor line
14	output ASCII
15	device
16	border
17	background
18	type
19	input
20	assign
21	add
22	subtract
23	multiply
24	divide
25	power
26	generate

```
27  period
28  conditional
29  else
30  conditional end
31  infinite loop
32  infinite loop end
33  exit
34  index loop
35  loop start
36  index loop end
37  jump label
38  jump
39  call
40  floating point number
41  variable
42  generate_1
43  operator=
44  operator/=
45  operator<
46  operator<=
47  operator>
48  operator>=
49  decimal places
50  no exponent
51  exponent
52  plus
53  minus
54  save letter
55  output identifier
56  save character
57  output string
58  save digit
59  output digit
```

We need a way for the parser to always be able to decide what kind of symbol is in the stack, so:

> Terminal symbols will be stored as numbers.
> Non-terminal symbols will be stored in two "levels:"
>
> > The lower level contains the number of the symbol.
> > The upper level contains the key number 253.
>
> Semantic symbols through 53 inclusive have the key number 252, and at number 54, the key number 251. Up to 53, it suffices to output the number; at 54 additional things must be done.

Rule 01 represented with these codes looks like this:

148, 252, 1, 253, 5, 141, 252, 2, 253, 8, 252, 3, 131, 253, 2, 146, 252, 4, 253, 5, 46, 253, 18, 252, 5, 254.

4.7 THE PARSER

Now we are far enough along so that we can actually write the parser. Because we explained each step in the lexical analysis, we'll deal with the following programs more quickly. We can do this because the program steps have already been discussed in detail.

All the rules of the grammar are placed in a string area so that when using a rule only the corresponding variable contents need be placed on the stack. Since we cannot directly undertake the value assignment for the individual string variables, we store the alternatives in DATA lines and then transfer the contents to variables.

The grammar is located in lines 49000 to 50000.

Lines 10040 to 10670:

The parser has found an error and reacts according to our strategy. The top stack symbol contains the information concerning which symbols are possible.

Lines 11000 to 14630:

The top stack symbol is a rule and the parser checks to see which alternative it must choose.

Try to follow the path through the parser for the smallest MINI program. You will then see how simple the program analysis has become.

4.8 THE PARSER LISTING

```
10   REM  PROGRAM FOR THE LEXICAL ANALYSIS
20   REM  AND SYNTANTICAL ANALYSIS
30   REM  OF MINI PROGRAMS
40   REM  --
50   REM  DECLARATIONS
60   REM  --
80   FL=8
90   BE=15
100  IN=8
110  OU=7
120  PR=3
130  I1$="" : I2$="" : I3$=""
140  I1=0   : I2=0   : I3=0
150  EN=0 : EM$="" : ET=0 : ES=0
160  DIM BA$(75) : REM TEST BASIC WORDS
170  DIM MI$(48) : REM TEXT MINI WORDS
180  T$="" : T=0
190  A=0 : B=0 : C=0 : D=0
200  DIM AL$(69) : REM GRAMMAR RULES
210  Z=0 : Z%=0 : REM LOCAL VARIABLES
220  SG=1000 : REM  STACK SIZE
230  DIM S%(SG) : REM  STACK
240  S=0 : REM STACK POINTER
250  AL=1 : REM POINTERCURRENT ALTERNATIVE
260  AL%=0 : REM LENGTH OF AL
270  T9=0 : REM LOCAL LOOP INDEX
280  TL=1 : G$="" : G=0
290  O=0 : REM TOP STACK SYMBOL
300  BU$="" : REM VARIABLE FOR  ACTION
310  FE=0 : REM ERROR  VARIABLE
320  ZN=0 : REM FLAG FOR START OF LINE
1000 REM --
1010 REM PREPARATIONS
1020 REM --
1030 :
1040 PRINTCHR$(147);CHR$(14)
1050 PRINT:PRINT"THE MINI-COMPILER :":PRINT
1060 PRINT"SEND TO PRINTER OR SCREEN? (P/S)?"
1070 GET I1$ : IF I1$="P" OR I1$="S" THEN 1090
1080 GOTO 1070
1090 IF I1$="P" THEN PR=4
1100 OPEN PR,PR,7 : REM OPEN OUTPUT CHANNEL
1110 PRINT:PRINT"PLEASE INSERT DATA DISK"
1120 PRINT"PRESS     RETURN !":PRINT
1130 GET I1$ : IF I1$<>CHR$(13) THEN 1130
1140 PRINT"NAME OF PROGRAM TO BE COMPILED"
1150 INPUT "NAME ";I2$
1160 OPEN BE,FL,BE
1170 PRINT#BE,"I"
```

```
1180 PRINT#BE,"S:MINI-SYN"
1190 INPUT#BE,EN,EM$,ET,ES
1210 OPEN IN,FL,IN,"0:"+I2$+",P,R" : GOSUB 60000
1220 GET#IN,I1$,I1$
1230 OPEN OU,FL,OU,"0:MINI-SYN,S,W" : GOSUB 60000
1240 GOSUB 59000
1245 GOSUB 58000
1250 GOSUB 51500 : GOSUB 51500 : GOSUB 51000
1260 GOSUB 49000 : REM GRAMMAR LOAD
1270 GOSUB 42000 : REM PLACE FIRST CHARACTER
1280 GOSUB 41000 : REM LOAD FIRST ALTERNATIVE
2000 REM PARSER LOOP
2010 O=S%(S)
2020 AL=255
2030 IF O=251 THEN GOTO 38000
2040 IF O=252 THEN GOTO 39000
2050 IF O=253 THEN GOTO 10000
2060 IF S%(S)=255 AND G=255 THEN GOTO 35000
2070 IF S%(S)=G THEN GOSUB 40000 : GOSUB42000 : GOTO 2000
2080 PRINT#PR,CHR$(13)"ERROR !"
2085 IF S%(S)=255 THEN PRINT#PR,"EOF EXPECTED."
2086 IF S%(S)=255 THEN PRINT#PR,"I AM ENDING THE PROGRAM." :
GOTO 35000
2090 IF S%(S)>127 THEN PRINT#PR,"THE WORD ";MI$(S%(S)-128);"
EXPECTED."
2100 IF S%(S)<128 THEN PRINT#PR,"THE WORD ";S;" EXPECTED."
2110 GOTO 10400
10000 REM SEARCH FOR ALTERNATIVE
10010 O=S%(S-1)
10020 GOSUB 11000
10025 IF AL=255 THEN GOTO 10040
10030 IF LEN(AL$(AL))<>0 THEN GOSUB 40000 :  GOSUB 41000  :
GOTO 2000
10035 GOSUB 41000 : GOTO 2000
10040 PRINT#PR,CHR$(13)
10050 PRINT#PR,"ERROR  !"
10060 PRINT#PR,"NO ALTERNATIVE POSSIBLE !"
10070 PRINT#PR,"ONE OF THE FOLLOWING SYMBOLS EXPECTED:"
10080 IF  O=1ORO=2ORO=3 THEN PRINT#PR,"EMPTY WRITE  WRITELN
LINEFEED BORDER   "
10090 IF O=1ORO=2ORO=3 THEN PRINT#PR,"CUSCOL CURSLIN WRITEAS
OUTPUTDEVICE    "
10100 IF O=1ORO=2ORO=3 THEN PRINT#PR,"CLEARSCREEN BACKGROUND
TYPE GET "
10110 IF O=1ORO=2ORO=3 THEN PRINT#PR,"ASSIGN ADD SUBTRACT"
10120 IF O=1ORO=2ORO=3 THEN PRINT#PR,"MULTIPLY DIVIDE POWER"
10130 IF O=1ORO=2ORO=3 THEN PRINT#PR,"GENERATE IF LOOP EXIT
FOR"
10140 IF O=1ORO=2ORO=3 THEN PRINT#PR,"JUMPLABEL JUMP CALL"
10150 IF  O=4 OR O=5 OR O=24 OR O=6 OR O=17  OR  O=25  THEN
PRINT#PR,"LETTER    ";
10160 IF O=24 OR O=15 THEN PRINT#PR,"= /= < <= > >= "
```

```
10170 IF O=24 THEN PRINT#PR,"IS AND OF WITH BY OVER IF TO"
10180 IF O=24 THEN PRINT#PR,"REPEAT THEN      "
10190 IF O=24 OR O=20 THEN PRINT#PR,", ";
10200 IF  O=24 OR O=20 OR O=3 OR O=6 OR O=23 OR  O=12  THEN
PRINT#PR,". ";
10210 IF O=16 THEN PRINT#PR,"PROGRAM  ";
10220 IF O=18 THEN PRINT#PR,"SUBROUTINE    EOF "
10230 IF O=7 OR O=19 THEN PRINT#PR,"FLOAT  ";
10240 IF O=8 THEN PRINT#PR,"BEGIN   "
10250 IF O=17 OR O=14 OR O=23 OR O=22 OR O=11 OR O=10  THEN
PRINT#PR,"A DIGIT  "
10260 IF O=6 OR O=23 OR O=12 OR O=13 THEN PRINT#PR,"TOVAR"
10270 IF O=3 THEN PRINT#PR,"PEND  SREND ELSE ENDIF LEND   "
10280 IF  O=21 THEN PRINT#PR," KEYBOARD  CHARACTER  WITHOUT
QUOTE MARKS"
10290 IF O=21 OR O=25 THEN PRINT#PR,"QUOTATION MARK      "
10300 IF O=23 OR O=12 OR O=13 OR O=9 THEN PRINT#PR,"E ";
10310 IF O=10 THEN PRINT#PR,"+ - ";
10400 FE=FE+1
10405 PRINT#PR,
10410 PRINT#PR,"THE FOLLOWING SYMBOL IS ALSO EXPECTED:"
10420 IF S%(S)>250 AND S>=2 THEN S=S-2 : GOTO 10420
10430 IF S<=0 THEN 35000
10440 IF S%(S)>127 THEN PRINT#PR,"  ";MI$(S%(S)-128)
10450 IF S%(S)<128 THEN PRINT#PR,"  ";S
10460 PRINT#PR,"PRESS   <RETURN>"
10470 GET EM$ : IF EM$<>CHR$(13) THEN 10470
10480 PRINT#PR,"SEARCHING PROGRAM";
10490 IF S%(S)>127 THEN PRINT#PR,MI$(S%(S)-128);
10500 IF S%(S)<128 THEN PRINT#PR,S;
10510 PRINT#PR," APPEARS."
10520 PRINT#PR,"SEARCHING HIGHEST TO NEXT PERIOD"
10530 PRINT#PR,"OR START OF LINE."
10540 IF  NZ<>1 THEN IF G<>46 THEN IF G<>S%(S)  THEN  GOSUB
42000 : GOTO 10540
10545 PRINT#PR,
10550 IF  NZ=1 THEN PRINT#PR,"I HAVE READ UP  TO  THE  NEXT
LINE!"
10560 IF   G=S%(S)  AND  S%(S)>127  THEN   PRINT#PR,"FOUND
";MI$(S%(S)-128)
10570 IF G=S%(S) AND S%(S)<128 THEN PRINT#PR,"FOUND ";S
10580 IF  G=S%(S) THEN PRINT#PR,"CONTINUING  THE  SYNTACTIC
CHECK."
10590 IF G=S%(S) THEN GOSUB 40000 : GOSUB 42000 : GOTO 2000
10600 IF  G=46  THEN PRINT#PR,"I HAVE READ UP TO  THE  NEXT
PERIOD."
10610 PRINT#PR,"TRYING TO RESTART AND      "
10620 PRINT#PR,"CONTINUE WITH SYNTACTIC      "
10630 PRINT#PR,"CHECK."
10640 IF S%(S)>250 AND S>=2 THEN S=S-2 : GOTO 10640
10650 IF S<=0 THEN 35000
10660 IF S%(S)<>46 THEN S=S-1  : GOTO 10640
```

```
10670 GOSUB 42000 : GOSUB 40000 : GOTO 2000
11000 REM BRANCHTO THE ALTERNATIVES
11010 ON INT(O/10) GOTO 11030,11040
11020 ON O GOTO12100,12200,12300,12400,12500,12600,12700,
12800,12900
11030 ON O-9 GOTO 13000,13100,13200,13300,13400,13500,13600,
13700,13800,13900
11040 ON O-19 GOTO 14000,14100,14200,14300,14400,14500,14600
11050 PRINT#PR,CHR$(13)"THIS RULE DOESN'T EXIST!"
11060 STOP
12000 REM SELECTION OF THE ALTERNATIVE
12100 REM /01/ INSTRUCTION
12105 IF G=142 THEN AL=15 : RETURN
12110 IF G=167 THEN AL=16 : RETURN
12115 IF G=168 THEN AL=17 : RETURN
12120 IF G=164 THEN AL=18 : RETURN
12125 IF G=152 THEN AL=19 : RETURN
12130 IF G=133 THEN AL=20 : RETURN
12135 IF G=134 THEN AL=21 : RETURN
12140 IF G=169 THEN AL=22 : RETURN
12145 IF G=129 THEN AL=23 : RETURN
12150 IF G=149 THEN AL=24 : RETURN
12155 IF G=139 THEN AL=25 : RETURN
12157 IF G=151 THEN AL=26 : RETURN
12160 IF G=140 THEN AL=27 : RETURN
12162 IF G=160 THEN AL=28 : RETURN
12164 IF G=128 THEN AL=29 : RETURN
12166 IF G=158 THEN AL=30 : RETURN
12168 IF G=144 THEN AL=31 : RETURN
12170 IF G=137 THEN AL=32 : RETURN
12172 IF G=147 THEN AL=33 : RETURN
12174 IF G=132 THEN AL=34 : RETURN
12176 IF G=166 THEN AL=35 : RETURN
12178 IF G=153 THEN AL=36 : RETURN
12180 IF G=130 THEN AL=37 : RETURN
12182 IF G=171 THEN AL=38 : RETURN
12184 IF G=157 THEN AL=39 : RETURN
12186 IF G=156 THEN AL=40 : RETURN
12188 IF G=150 THEN AL=41 : RETURN
12190 RETURN
12200 REM /02/ INSTRUCTION BLOCK
12210 GOSUB 12100
12220 IF AL<>255 THEN AL=12
12230 RETURN
12300 REM /03/ INSTRUCTION SET
12310 GOSUB 12100
12320 IF G=146 THEN AL=14 : RETURN
12330 IF G=161 THEN AL=14 : RETURN
12340 IF G=155 THEN AL=14 : RETURN
12350 IF G=165 THEN AL=14 : RETURN
12360 IF G=46  THEN AL=14 : RETURN
12370 IF G=154 THEN AL=14 : RETURN
```

```
12380 IF AL<>255 THEN AL=13 : RETURN
12390 RETURN
12400 REM /04/ CONDITION
12410 IF G>64 AND G<91 THEN AL=46
12420 RETURN
12500 REM /05/ IDENTIFIER
12510 IF G>64 AND G<91 THEN AL=4
12520 RETURN
12600 REM /06/ GENERATE-1
12610 IF G=145 THEN AL=44 : RETURN
12620 IF G=46  THEN AL=45 : RETURN
12630 RETURN
12700 REM /07/ LETTER
12710 AL=6
12720 RETURN
12800 REM /08/ DEFINITION BLOCK
12810 IF G=138 THEN AL=7
12820 IF G=131 THEN AL=8
12830 RETURN
12900 REM /09/ EXPONENT
12910 IF G=69 THEN AL=64
12920 RETURN
13000 REM /10/ EXPONENT-1
13010 IF G=43 THEN AL=65
13020 IF G=45 THEN AL=66
13030 IF G>47 AND G<58 THEN AL=67
13040 RETURN
13100 REM /11/ FLOATING-POINT  NUMBER
13110 IF G>47 AND G<58 THEN AL=58
13120 RETURN
13200 REM /12/ FLOATING-POINT  NUMBER 1
13210 IF G=46 THEN AL=59
13220 IF G=69 THEN AL=60
13230 IF G=145 THEN AL=61
13240 RETURN
13300 REM /13/ FLOATING-POINT  NUMBER 2
13310 IF G=69 THEN AL=62
13320 IF G=145 THEN AL=63
13330 RETURN
13400 REM /14/ INTEGER
13410 IF G>47 AND G<58 THEN AL=55
13420 RETURN
13500 REM /15/ OPERATOR
13510 IF G=61 THEN AL=47 : RETURN
13520 IF G=174 THEN AL=48 : RETURN
13530 IF G=60  THEN AL=49 : RETURN
13540 IF G=175 THEN AL=50 : RETURN
13550 IF G=62  THEN AL=51 : RETURN
13560 IF G=176 THEN AL=52 : RETURN
13570 RETURN
13600 REM /16/ PROGRAM
13610 IF G=148 THEN AL=1
```

```
13620 RETURN
13700 REM /17/ ASSIGN-1
13710 IF G>47 AND G<58 THEN AL=42 : RETURN
13720 IF G>64 AND G<91 THEN AL=43
13730 RETURN
13800 REM /18/ SUBROUTINE
13810 IF G=162 THEN AL=2
13820 IF G=255 THEN AL=3
13830 RETURN
13900 REM /19/ VARIABLE DEFINITION
13910 IF G=138 THEN AL=9
13920 RETURN
14000 REM /20/ VARIABLE DEFINITION-2
14010 IF G=44 THEN AL=10
14020 IF G=46 THEN AL=11
14030 RETURN
14100 REM /21/ STRING
14110 IF G<>34 THEN AL=53
14120 IF G=34 THEN AL=54
14130 RETURN
14200 REM /22/ DIGIT
14210 AL=57
14220 RETURN
14300 REM /23/ INTEGER-1
14310 IF G>47 AND G<58 THEN AL=55
14320 IF G=46 OR G=69 OR G=145 THEN AL=56
14330 RETURN
14400 REM /24/ IDENTIFIER-1
14410 IF G>64 AND G<91 THEN AL=4 : RETURN
14420 IF G=141 OR G=46 OR G=44 OR G=170 OR G=145 THEN AL=5 :
RETURN
14430 IF G=163 OR G=143 OR G=136 OR G=159 OR G=166 THEN AL=5
: RETURN
14440 IF G=172 OR G=173 OR G=135 OR G=174 OR G=175 THEN AL=5
: RETURN
14450 IF G=176 OR G=60 OR G=61 OR G=62 THEN AL=5 : RETURN
14460 RETURN
14500 REM /25/ WRITE-1
14510 IF G=34 THEN AL=68 : RETURN
14520 IF G>64 AND G<91 THEN AL=69 : RETURN
14530 RETURN
14600 REM /26/ KEYBOARD CHARACTER
14610 IF G<>34 THEN AL=57 : RETURN
14620 AL=6
14630 RETURN
35000 REM PROGRAM END
35010 PRINT#PR,CHR$(13)CHR$(13)"SYNTACTIC CHECK COMPLETED."
35020 IF FE<>0 THEN PRINT#PR,FE; " ERRORS FOUND."
35030 IF FE=0 THEN PRINT#PR,CHR$(13)"PROGRAM SYNTACTIC
CORRECT."
35040 PRINT#OU,CHR$(255)CHR$(255);
35050 CLOSE PR : CLOSE IN : CLOSE OU : CLOSE BE
```

```
35060 END
38000 REM OUTPUT FOOT SEMANTIC
38010 IF S%(S-1)=54 THEN BU$=BU$+G$ :  GOSUB 40000  :  GOSUB
42000 : GOTO 2000
38020  IF S%(S-1)=56 THEN BU$=BU$+G$ :  GOSUB 40000 :  GOSUB
42000 : GOTO 2000
38030 IF S%(S-1)=58 THEN BU$=BU$+G$ :  GOSUB 40000  :  GOSUB
42000 : GOTO 2000
38040  IF S%(S-1)=55 THEN PRINT#OU,CHR$(255)CHR$(55);BU$;  :
BU$="" : GOSUB 40000
38050 IF S%(S-1)=57 THEN  PRINT#OU,CHR$(255)CHR$(57);BU$;  :
BU$="" : GOSUB 40000
38060  IF S%(S-1)=59 THEN PRINT#OU,CHR$(255)CHR$(59);BU$;  :
BU$="" : GOSUB 40000
38070 GOTO 2000
39000 REM OUTPUT FGFOR SEMANTIC
39010 PRINT#OU,CHR$(255)CHR$(253)CHR$(S%(S-1));
39020 GOSUB 40000
39030 GOTO 2000
40000 REM REMOVING THE TOP SYMBOLS
40010 IF S%(S)=253 THEN S=S-2 : RETURN
40020 IF S%(S)=252 THEN S=S-2 : RETURN
40030 IF S%(S)=251 THEN S=S-2 : RETURN
40040 S=S-1 : RETURN
40050 :
41000 REM WRITE ALTERNATIVE ON THE STACK
41010 AL%=LEN(AL$(AL))
41020 IF AL%=0 THEN GOSUB 40000 : RETURN
41025 S=S+1
41030 FOR T9=AL% TO 1 STEP -1
41040 S%(S)=ASC(MID$(AL$(AL),T9,1)) : S=S+1
41050 NEXT T9
41060 S=S-1
41065 REM PRINT#PR,CHR$(13)CHR$(13):FORL=STO0STEP-
1:PRINT#PR,S%(L);:NEXT
41070 RETURN
41080 :
42000 REM PLACE A NEW CHARACTER
42010 IF TL>LEN(T$) THEN T$="" : GOSUB 50000 : TL=1
42020 G$=MID$(T$,TL,1) : G=ASC(G$)
42030 TL=TL+1
42035 REM PRINT#PR,CHR$(13)"THE CURRENT CHARACTER:";G
42037 REM PRINT#PR,CHR$(13)"T$=  ";T$
42040 RETURN
42050 :
49000 REM   THE GRAMMAR
49010 DATA 148,252,1,253,5,141,252,2,253,8,252,3
49012 DATA 131,253,2,146,252,4,253,5,46,253,18,252,5,255,250
49020 DATA 162,252,6,253,5,141,253,2,161,252,7,253,5,46,253,
18,250
49030 DATA 250
49040 DATA 251,54,253,7,253,24,250
```

```
49050 DATA 251,55,250
49060 DATA 250
49070 DATA 253,19,250
49080 DATA 250
49090 DATA 138,253,5,253,20,46,253,8,250
49100 DATA 44,253,5,253,20,250
49110 DATA 250
49120 DATA 253,1,253,3,250
49130 DATA 253,1,253,3,250
49140 DATA 250
49150 DATA 142,46,250
49160 DATA 252,8,167,253,25,46,250
49170 DATA 252,9,168,253,25,46,250
49180 DATA 252,10,164,46,250
49190 DATA 252,11,152,46,250
49200 DATA 252,12,133,253,14,46,250
49210 DATA 252,13,134,253,14,46,250
49220 DATA 252,14,169,253,14,46,250
49230 DATA 252,15,129,253,5,46,250
49240 DATA 252,16,149,253,5,46,250
49250 DATA 252,17,139,253,5,46,250
49260 DATA 252,18,151,253,5,46,250
49270 DATA 252,19,140,253,5,46,250
49280 DATA 252,20,160,253,17,46,250
49290 DATA 252,21,128,253,5,170,253,5,145,253,5,46,250
49300 DATA 252,22,158,253,5,163,253,5,145,253,5,46,250
49310 DATA 252,23,144,253,5,143,253,5,145,253,5,46,250
49320 DATA 252,24,137,253,5,136,253,5,145,253,5,46,250
49330 DATA 252,25,147,253,5,143,253,5,145,253,5,46,250
49340 DATA 252,26,132,253,5,163,253,5,253,6,252,27,46,250
49350 DATA 252,28,166,253,4,135,253,2,252,29,155,253,2
49355 DATA 165,252,30,46,250
49360 DATA 252,31,153,253,5,159,253,2,154,252,32,253,5,
46,250
49370 DATA 252,33,130,253,5,166,253,4,46,250
49380 DATA 252,34,171,253,5,163,253,5,172,253,5,173
49385 DATA 252,35,253,2,154,252,36,46,250
49390 DATA 252,37,157,253,5,46,250
49400 DATA 252,38,156,253,5,46,250
49410 DATA 252,39,150,253,5,46,250
49420 DATA 252,40,253,11,145,253,5,250
49430 DATA 252,41,253,5,145,253,5,250
49440 DATA 252,42,145,253,5,250
49450 DATA 250
49460 DATA 253,5,253,15,253,5,250
49470 DATA 252,43,61,250
49480 DATA 252,44,174,250
49490 DATA 252,45,60,250
49500 DATA 252,46,175,250
49510 DATA 252,47,62,250
49520 DATA 252,48,176,250
49530 DATA 251,56,253,26,253,21,250
```

```
49540 DATA 251,57,250
49550 DATA 251,58,253,22,253,23,250
49560 DATA 251,59,250
49570 DATA 250
49580 DATA 253,14,253,12,250
49590 DATA 252,49,46,253,14,253,13,250
49600 DATA 253,9,250
49610 DATA 252,50,250
49620 DATA 253,9,250
49630 DATA 252,50,250
49640 DATA 252,51,69,253,10,250
49650 DATA 252,53,43,253,14,250
49660 DATA 252,53,45,253,14,250
49670 DATA 253,14,250
49680 DATA 34,253,21,34,250
49690 DATA 253,5,250
49800 FOR Z=1 TO 69
49810 READ Z% : IF Z%<>250 THEN AL$(Z)=AL$(Z)+CHR$(Z%) :
GOTO 49810
49820 NEXT
49830 RETURN
50000 REM LEXICAL ANALYSIS
50005 ZN=0
50010 IF I1=0 THEN IF I2=0 THEN IF I3=0 THEN T$=CHR$(255) :
RETURN
50020 IF I1=0 THEN GOSUB 51000
50030 IF I1>127 AND I1<204 THEN I1$=BA$(I1-128)
50040 IF I1=171 THEN IF I2=171 THEN GOSUB 51700 : GOTO 50000
50050 IF I1$=" " THEN IF I2$=" " THEN PRINT#PR,I1$;  : GOSUB
51500 : GOTO 50000
50070 IF I1$="<" AND I2=178 THEN T$=CHR$(175) : GOTO 50195
50072 IF I1$=">" AND I2=178 THEN T$=CHR$(176) : GOTO 50195
50074 IF I1$="/" AND I2=178 THEN T$=CHR$(174) : GOTO 50195
50080 IF I1=34 THEN GOSUB 53000 : GOTO 50210
50085 IF I1=32 THEN IF I2=34 THEN GOTO 50130
50090 IF I2$="=" OR I2$="/" OR I2$="<" OR I2$=">" THEN GOSUB
52000 : GOTO 50200
50100 IF  I2$="." OR I2$=" " OR I2=0 OR  I2=34  THEN  GOSUB
52000 : GOTO 50200
50110 IF I2$="," THEN GOSUB 52000 : GOTO 50200
50126 IF I1$="." OR I1$="," OR I1$="=" THEN T$=I1$  :  GOTO
50200
50128 IF I1$="/" OR I1$="<" OR I1$=">" THEN T$=I1$  :  GOTO
50200
50130 IF I1$=" " THEN PRINT#PR," ";  :  GOSUB 51500 :  GOTO
50000
50190 T$=T$+I1$ : PRINT#PR,I1$; : GOSUB 51500 : GOTO 50000
50195 PRINT#PR,I1$; : GOSUB 51500
50197 IF I1>127 AND I1<204 THEN I1$=BA$(I1-128)
50200 PRINT#PR,I1$; : GOSUB 51500
50210 RETURN
51000 REM START OF LINE
```

```
51005 :
51007 ZN=1
51010 GOSUB 51500 : GOSUB 51500 : GOSUB 51500
51020 PRINT#OU,CHR$(255)CHR$(254)CHR$(I1)CHR$(I2);
51030 PRINT#PR,CHR$(13);STR$(I1+I2*256);" ";
51040 GOSUB 51500 : GOSUB 51500 : RETURN
51050 :
51500 REM READ CHARACTER
51510 I1$=I2$ : I2$=I3$ : GET#IN,I3$
51520 I1=I2 : I2=I3 : IF I3$="" THEN I3=0 : RETURN
51530 I3=ASC(I3$) : RETURN
51540 :
51700 REM SKIP LINE TO END
51710 PRINT#PR,I1$; : GOSUB 51500 : IF I1=0 THEN RETURN
51720 IF I1>127 AND I1<204 THEN I1$=BA$(I1-128)
51730 GOTO 51710
51740 :
52000 REM IS  T$ A   TOKEN?
52010 :
52020 T$=T$+I1$ : T=0 : A=ASC(LEFT$(T$,1))
52030 IF A<65 OR A>90 THEN RETURN
52035 D=0:C=45:GOSUB52700:RETURN
52700 FOR D=B TO C
52710 IF T$=MI$(D) THEN T=128+D
52720 NEXT D
52800 IF T=0 THEN RETURN
52810 T$=CHR$(T) : RETURN
52900 RETURN
53000 REM SKIP STRING
53010 T$=T$+I1$ :  PRINT#PR,I1$;  :  GOSUB 51500 : IF I1<>34
THEN GOTO 53010
53020 T$=T$+I1$ : PRINT#PR,I1$; : GOSUB 51500 : RETURN
53030 :
58000 REM MINI WORDS
58005 :
58008 MI$(0)="ADD"
58009 MI$(42)="AND"
58010 MI$(32)="ASSIGN"
58020 MI$(11)="BACKGROUND"
58030 MI$(3)="BEGIN"
58035 MI$(21)="BORDER"
58040 MI$(8)="BY"
58042 MI$(22)="CALL"
58045 MI$(24)="CLEARSCREEN"
58050 MI$(5)="CURSCOL"
58060 MI$(6)="CURSLIN"
58090 MI$(9)="DIVIDE"
58091 MI$(27)="ELSE"
58093 MI$(14)="EMPTY"
58094 MI$(37)="ENDIF"
58099 MI$(2)="EXIT"
58100 MI$(10)="FLOAT"
```

```
58112 MI$(4)="GENERATE"
58120 MI$(12)="GET"
58121 MI$(38)="IF"
58130 MI$(13)="IS"
58132 MI$(28)="JUMP"
58134 MI$(29)="JUMPLABEL"
58140 MI$(26)="LEND"
58145 MI$(36)="LINEFEED"
58150 MI$(25)="LOOP"
58160 MI$(16)="MULTIPLY"
58161 MI$(35)="OF"
58162 MI$(31)="OVER"
58163 MI$(1)="OUTPUTDEVICE"
58180 MI$(18)="PEND"
58190 MI$(19)="POWER"
58200 MI$(20)="PROGRAM"
58270 MI$(33)="SREND"
58280 MI$(34)="SUBROUTINE"
58290 MI$(30)="SUBTRACT"
58292 MI$(7)="THEN"
58296 MI$(17)="TOVAR"
58299 MI$(23)="TYPE"
58388 MI$(15)="WITH"
58390 MI$(39)="WRITE"
58400 MI$(40)="WRITELN"
58410 MI$(41)="WRITEAS"
58430 MI$(43)="FOR"
58440 MI$(44)="TO"
58450 MI$(45)="REPEAT"
58460 MI$(46)="/="
58470 MI$(47)="<="
58480 MI$(48)=">="
58490 RETURN
59000 REM BASIC-WORDS
59008 BA$(0)="END"
59010 BA$(1)="FOR"
59020 BA$(2)="NEXT"
59030 BA$(3)="DATA"
59040 BA$(4)="INPUT#"
59050 BA$(5)="INPUT"
59060 BA$(6)="DIM"
59070 BA$(7)="READ"
59080 BA$(8)="LET"
59090 BA$(9)="GOTO"
59100 BA$(10)="RUN"
59110 BA$(11)="IF"
59120 BA$(12)="RESTORE"
59130 BA$(13)="GOSUB"
59140 BA$(14)="RETURN"
59150 BA$(15)="REM"
59160 BA$(16)="STOP"
59170 BA$(17)="ON"
```

```
59180 BA$(18)="WAIT"
59190 BA$(19)="LOAD"
59200 BA$(20)="SAVE"
59210 BA$(21)="VERIFY"
59220 BA$(22)="DEF"
59230 BA$(23)="POKE"
59240 BA$(24)="PRINT#"
59250 BA$(25)="PRINT"
59260 BA$(26)="CONT"
59270 BA$(27)="LIST"
59280 BA$(28)="CLR"
59290 BA$(29)="CMD"
59300 BA$(30)="SYS"
59310 BA$(31)="OPEN"
59320 BA$(32)="CLOSE"
59330 BA$(33)="GET"
59340 BA$(34)="NEW"
59350 BA$(35)="TAB("
59360 BA$(36)="TO"
59370 BA$(37)="FN"
59380 BA$(38)="SPC("
59390 BA$(39)="THEN"
59400 BA$(40)="NOT"
59410 BA$(41)="STEP"
59420 BA$(42)="+"
59430 BA$(43)="-"
59440 BA$(44)="*"
59450 BA$(45)="/"
59460 BA$(46)="^"
59470 BA$(47)="AND"
59480 BA$(48)="OR"
59490 BA$(49)=">"
59500 BA$(50)="="
59510 BA$(51)="<"
59520 BA$(52)="SGN"
59530 BA$(53)="INT"
59540 BA$(54)="ABS"
59550 BA$(55)="USR"
59560 BA$(56)="FRE"
59570 BA$(57)="POS"
59580 BA$(58)="SQR"
59590 BA$(59)="RND"
59600 BA$(60)="LOG"
59610 BA$(61)="EXP"
59620 BA$(62)="COS"
59630 BA$(63)="SIN"
59640 BA$(64)="TAN"
59650 BA$(65)="ATN"
59660 BA$(66)="PEEK"
59670 BA$(67)="LEN"
59680 BA$(68)="STR$"
59690 BA$(69)="VAL"
```

```
59700 BA$(70)="ASC"
59710 BA$(71)="CHR$"
59720 BA$(72)="LEFT$"
59730 BA$(73)="RIGHT$"
59740 BA$(74)="MID$"
59750 BA$(75)="GO"
59760 RETURN
59770 :
60000 REM READ DISK DRIVE ERROR CHANNEL
60010 :
60020 INPUT#BE,EN,EM$,ET,ES : IF EN=0 THEN RETURN
60030 PRINT
60040 PRINT"****  ERROR ON  DISKETTE!"
60050 PRINT"****  ERROR NUMBER : ";EN
60060 PRINT"****  ERROR MESSAGE   :"
60070 PRINT"****  ";EM$
60080 PRINT"****  TRACK           : ";ET
60090 PRINT"****  SECTOR          : ";ES
60100 PRINT#BE,"I"
60110 CLOSE BE : CLOSE PR
60120 PRINT : PRINT"****  PROGRAM  STOPPED!!!!!"
60130 END
```

4.9 PRINTING THE PARSER OUTPUT

We can print the parser output required for the semantic analysis and code generation with the following program.

You then get a "condensed" or summarized version of your program which still contains all of the necessary information.

Example:

The output for the program "start":

```
program identifier definition
identifier : start
definition section start
definition section end
program identifier check
identifier : start
program end
program end
```

The first "program end" signifies the logical end of the program and the second indicates the physical end of the file.

```
1000 REM PROGRAM FOR OUTPUT OF
1010 REM MINI-SYN
1020 REM
1030 REM ------------------------
1040 PRINT CHR$(147)
1050 PRINT"OUTPUT TO PRINTER (P) OR SCREEN (S) ?"
1060 GET W$ : IF W$="P" THEN PR=4 : GOTO 1085
1070 IF W$="S" THEN PR=3 : GOTO 1085
1080 GOTO 1060
1085 PRINT"WITH LINE NUMBERS (Y) ? "
1087 GET W$ : IF W$="" THEN 1087
1089 IF W$="Y" THEN SS=1
1100 OPEN PR,PR
1110 IN=8
1120 OPEN IN,IN,IN,"MINI-SYN,S,R"
1130 GET#IN,W$
1140 GET#IN,W$ : IF W$="" THEN W=0 : GOTO 1160
1150 W=ASC(W$)
1160 IF W=253 THEN  PRINT#PR,CHR$(13);
1165 IF W<>253 THEN 5000
1166 GET#IN,W$ : W=ASC(W$)
1170 IF W=1 THEN PRINT#PR,"PROGRAM IDENTIFIER DEFINITION
          "; : GOTO 1130
1180 IF W=2  THEN PRINT#PR,"DEFINITIONS START
                    "; : GOTO 1130
1190 IF W=3  THEN PRINT#PR,"DEFINITIONS END
                      "; : GOTO 1130
1200 IF W=4  THEN PRINT#PR,"PROGRAM IDENTIFER CHECK
                "; : GOTO 1130
1210 IF W=5  THEN PRINT#PR,"PROGRAM END
                           "; : GOTO 1130
1220 IF W=6  THEN PRINT#PR,"SUBROUTINE IDENTIFIER
                  "; : GOTO 1130
1230 IF W=7  THEN PRINT#PR,"SUBROUTINE IDENTIFER CHECK
              "; : GOTO 1130
1240 IF W=8  THEN PRINT#PR,"OUTPUT
                                 "; : GOTO 1130
1250 IF W=9  THEN PRINT#PR,"OUTPUT WITH LINEFEED
                  "; : GOTO 1130
1260 IF W=10 THEN PRINT#PR,"LINEFEED
                                   "; : GOTO 1130
1270 IF W=11 THEN PRINT#PR,"CLEARSCREEN
                           "; : GOTO 1130
1280 IF W=12 THEN PRINT#PR,"CURSOR COLUMN
                         "; : GOTO 1130
1290 IF W=13 THEN PRINT#PR,"CURSOR LINE
                           "; : GOTO 1130
1300 IF W=14 THEN PRINT#PR,"OUTPUT  ASCII
                         "; : GOTO 1130
1310 IF W=15 THEN PRINT#PR,"DEVICE
                                 "; : GOTO 1130
```

```
1320 IF W=16 THEN PRINT#PR,"BORDER
                                  "; : GOTO 1130
1330 IF W=17 THEN PRINT#PR,"BACKGROUND
                                   "; : GOTO 1130
1340 IF W=18 THEN PRINT#PR,"TYPE
                                     "; : GOTO 1130
1350 IF W=19 THEN PRINT#PR,"INPUT
                                    "; : GOTO 1130
1360 IF W=20 THEN PRINT#PR,"ASSIGN
                                    "; : GOTO 1130
1400 IF W=21 THEN PRINT#PR,"ADD
                                      "; : GOTO 1130
1410 IF W=22 THEN PRINT#PR,"SUBTRACT
                                   "; : GOTO 1130
1420 IF W=23 THEN PRINT#PR,"MULTIPLY
                                   "; : GOTO 1130
1430 IF W=24 THEN PRINT#PR,"DIVIDE
                                    "; : GOTO 1130
1440 IF W=25 THEN PRINT#PR,"POWER
                                     "; : GOTO 1130
1450 IF W=26 THEN PRINT#PR,"GENERATE
                                   "; : GOTO 1130
1460 IF W=27 THEN PRINT#PR,"PERIOD
                                    "; : GOTO 1130
1500 IF W=28 THEN PRINT#PR,"DECISION
                                   "; : GOTO 1130
1510 IF W=29 THEN PRINT#PR,"ELSE
                                     "; : GOTO 1130
1520 IF W=30 THEN PRINT#PR,"DECISION END
                                "; : GOTO 1130
1530 IF W=31 THEN PRINT#PR,"INFINITE LOOP
                               "; : GOTO 1130
1540 IF W=32 THEN PRINT#PR,"INFINITE LOOP END
                            "; : GOTO 1130
1550 IF W=33 THEN PRINT#PR,"EXIT
                                     "; : GOTO 1130
1560 IF W=34 THEN PRINT#PR,"INDEX LOOP
                                 "; : GOTO 1130
1600 IF W=35 THEN PRINT#PR,"LOOP  START
                                "; : GOTO 1130
1610 IF W=36 THEN PRINT#PR,"INDEX LOOP END
                              "; : GOTO 1130
1620 IF W=37 THEN PRINT#PR,"JUMP LABEL
                                 "; : GOTO 1130
1630 IF W=38 THEN PRINT#PR,"JUMP
                                     "; : GOTO 1130
1640 IF W=39 THEN PRINT#PR,"CALL
                                     "; : GOTO 1130
1650 IF W=40 THEN PRINT#PR,"FLOATING-POINT NUMBER
                         "; : GOTO 1130
1660 IF W=41 THEN PRINT#PR,"VARIABLE
                                   "; : GOTO 1130
```

```
1700 IF W=42 THEN PRINT#PR,"GENERATE-1
                           "; : GOTO 1130
1710 IF W=43 THEN PRINT#PR,"OPERATOR =
                           "; : GOTO 1130
1720 IF W=44 THEN PRINT#PR,"OPERATOR /=
                          "; : GOTO 1130
1730 IF W=45 THEN PRINT#PR,"OPERATOR <
                           "; : GOTO 1130
1740 IF W=46 THEN PRINT#PR,"OPERATOR <=
                           "; : GOTO 1130
1750 IF W=47 THEN PRINT#PR,"OPERATOR >
                           "; : GOTO 1130
1760 IF W=48 THEN PRINT#PR,"OPERATOR >=
                           "; : GOTO 1130
1800 IF W=49 THEN PRINT#PR,"DECIMAL PLACES
                        "; : GOTO 1130
1810 IF W=50 THEN PRINT#PR,"NO   EXPONENT
                       "; : GOTO 1130
1820 IF W=51 THEN PRINT#PR,"EXPONENT
                          "; : GOTO 1130
1830 IF W=52 THEN PRINT#PR,"PLUS
                            "; : GOTO 1130
1840 IF W=53 THEN PRINT#PR,"MINUS
                              "; : GOTO 1130
1850 PRINT#PR,"ERROR  !!!!! ";
1860 STOP
5000 REM ROUTINE  W>54
5010 IF W=255 THEN PRINT#PR,CHR$(13)"PROGRAM  END ";  : GOTO
13000
5020 IF W=254 AND SS=1 THEN PRINT#PR,CHR$(13)"LINENUMBER   :
"; : GOTO 14000
5025 IF W=254 THEN GOTO 14000
5100 IF W=55 THEN PRINT#PR,CHR$(13)"IDENTIFIER :  ";  : GOTO
10000
5110 IF W=57 THEN PRINT#PR,CHR$(13)"STRING       : "; : GOTO
11000
5120 IF W=59 THEN PRINT#PR,CHR$(13)"INTEGER   :  ";  :  GOTO
12000
10000 REM READ IDENTIFER
10010 GET#IN,W$ : IF W$=CHR$(255) THEN 1140
10020 PRINT#PR,W$;
10030 GOTO 10010
11000 REM READ STRING
11010 GET#IN,W$ : IF W$=CHR$(255) THEN 1140
11020 PRINT#PR,W$;
11030 GOTO 10010
12000 REM READ INTEGER
12010 GET#IN,W$ : IF W$=CHR$(255) THEN 1140
12020 PRINT#PR,W$;
12030 GOTO 10010
13000 REM PROGRAM  END
13010 CLOSE PR
```

```
13020 CLOSE IN
13030 END
14000 REM LINE NUMBER
14010 GET#IN,W$ : IF W$="" THEN Z=0 : GOTO 14020
14015  Z=ASC(W$)
14020 GET#IN,W$ : IF W$="" THEN GOTO 14025
14022  Z=Z+ASC(W$)*256
14025 IF SS=1 THEN PRINT#PR,Z;
14030 GOTO 1130
READY.
```

Example program "START":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : START
DEFINITIONS START
DEFINITIONS END
PROGRAM IDENTIFER CHECK
IDENTIFIER : START
PROGRAM END
PROGRAM  END
```

Example program "OUTPUT":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : OUTPUT
DEFINITIONS START
DEFINITIONS END
OUTPUT WITH LINEFEED
STRING        : EXAMPLE OF
OUTPUT
STRING        : THE TEXT
OUTPUT
STRING        : OUTPUT
PROGRAM IDENTIFER CHECK
IDENTIFIER : OUTPUT
PROGRAM END
PROGRAM  END
```

Example program "OUTPUTEX":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : OUTPUTEX
DEFINITIONS START
DEFINITIONS END
CLEARSCREEN
CURSOR LINE
INTEGER  : 5
CURSOR COLUMN
INTEGER  : 5
OUTPUT  ASCII
INTEGER  : 18
OUTPUT
STRING        : OUTPUT ON THE SCREEN
OUTPUT  ASCII
INTEGER  : 146
LINEFEED
LINEFEED
LINEFEED
OUTPUT WITH LINEFEED
STRING        : OUTPUT ON THE SCREEN
```

```
PROGRAM IDENTIFER CHECK
IDENTIFIER : OUTPUTEX
PROGRAM END
PROGRAM  END
```

Example program "PRINTER":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : PRINTER
DEFINITIONS START
DEFINITIONS END
DEVICE
IDENTIFIER : PRINTER
OUTPUT WITH LINEFEED
STRING       : NOW THE PRINTER PRINTS!
DEVICE
IDENTIFIER : SCREEN
OUTPUT WITH LINEFEED
STRING       : NOW BACK TO THE SCREEN!
PROGRAM IDENTIFER CHECK
IDENTIFIER : PRINTER
PROGRAM END
PROGRAM  END
```

Example program "DECLARATION":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : DECLARATION
DEFINITIONS START
IDENTIFIER : TOM
IDENTIFIER : DICK
IDENTIFIER : HARRY
DEFINITIONS END
PROGRAM IDENTIFER CHECK
IDENTIFIER : DECLARATION
PROGRAM END
PROGRAM  END
```

Example program "COLOR":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : COLOR
DEFINITIONS START
DEFINITIONS END
CLEARSCREEN
BACKGROUND
IDENTIFIER : BLACK
BORDER
IDENTIFIER : WHITE
```

```
TYPE
IDENTIFIER : WHITE
CURSOR LINE
INTEGER  : 12
CURSOR COLUMN
INTEGER  : 17
OUTPUT  ASCII
INTEGER  : 18
OUTPUT WITH LINEFEED
STRING       : COLOR
OUTPUT  ASCII
INTEGER  : 146
BACKGROUND
IDENTIFIER : WHITE
BORDER
IDENTIFIER : BLACK
PROGRAM IDENTIFER CHECK
IDENTIFIER : COLOR
PROGRAM END
PROGRAM  END
```

Example program "IN":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : IN
DEFINITIONS START
IDENTIFIER : AGE
DEFINITIONS END
CLEARSCREEN
OUTPUT
STRING       : YOUR AGE:
INPUT
IDENTIFIER : AGE
LINEFEED
OUTPUT
STRING       : YOU ARE
OUTPUT
IDENTIFIER : AGE
OUTPUT WITH LINEFEED
STRING       :   YEARS OLD.
PROGRAM IDENTIFER CHECK
IDENTIFIER : IN
PROGRAM END
PROGRAM  END
```

Example program "ARITHMETIC":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : ARITHMETIC
DEFINITIONS START
IDENTIFIER : MONICA
IDENTIFIER : THOMAS
IDENTIFIER : CECILIA
IDENTIFIER : TOM
IDENTIFIER : DICK
IDENTIFIER : HARRY
DEFINITIONS END
CLEARSCREEN
LINEFEED
OUTPUT WITH LINEFEED
STRING       : DEMONSTRATION OF FLOATING-POINT ARITHMETIC
LINEFEED
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 2
NO   EXPONENT
IDENTIFIER : MONICA
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 5
NO   EXPONENT
IDENTIFIER : THOMAS
ASSIGN
VARIABLE
IDENTIFIER : THOMAS
IDENTIFIER : CECILIA
OUTPUT
STRING       : MONICA =
OUTPUT WITH LINEFEED
IDENTIFIER : MONICA
OUTPUT
STRING       : THOMAS =
OUTPUT WITH LINEFEED
IDENTIFIER : THOMAS
OUTPUT
STRING       : CECILIA =
OUTPUT WITH LINEFEED
IDENTIFIER : CECILIA
ADD
IDENTIFIER : MONICA
IDENTIFIER : THOMAS
IDENTIFIER : TOM
LINEFEED
OUTPUT
STRING       : 2 + 5 =
OUTPUT WITH LINEFEED
IDENTIFIER : TOM
```

```
SUBTRACT
IDENTIFIER : CECILIA
IDENTIFIER : TOM
IDENTIFIER : DICK
OUTPUT
STRING       : 7 - 5 =
OUTPUT WITH LINEFEED
IDENTIFIER : TOM
MULTIPLY
IDENTIFIER : THOMAS
IDENTIFIER : MONICA
IDENTIFIER : HARRY
OUTPUT
STRING       : 5 * 2 =
OUTPUT WITH LINEFEED
IDENTIFIER : HARRY
DIVIDE
IDENTIFIER : CECILIA
IDENTIFIER : MONICA
IDENTIFIER : DICK
OUTPUT
STRING       : 5 / 2 =
OUTPUT WITH LINEFEED
IDENTIFIER : DICK
POWER
IDENTIFIER : MONICA
IDENTIFIER : THOMAS
IDENTIFIER : TOM
OUTPUT
STRING       : 2 ^  5 =
OUTPUT WITH LINEFEED
IDENTIFIER : TOM
PROGRAM IDENTIFER CHECK
IDENTIFIER : ARITHMETIC
PROGRAM END
PROGRAM  END
```

Example program "FUNCTION":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : FUNCTION
DEFINITIONS START
IDENTIFIER : A
IDENTIFIER : B
IDENTIFIER : C
IDENTIFIER : D
IDENTIFIER : E
IDENTIFIER : F
IDENTIFIER : G
IDENTIFIER : PIQUARTER
DEFINITIONS END
```

```
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 4
NO   EXPONENT
IDENTIFIER : A
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 4
NO   EXPONENT
IDENTIFIER : B
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 3
DECIMAL PLACES
INTEGER  : 1415
NO   EXPONENT
IDENTIFIER : C
DIVIDE
IDENTIFIER : C
IDENTIFIER : B
IDENTIFIER : PIQUARTER
CLEARSCREEN
OUTPUT WITH LINEFEED
STRING       : DEMONSTRATION OF THE FUNCTIONS
LINEFEED
GENERATE
IDENTIFIER : INTEGER
IDENTIFIER : C
GENERATE-1
IDENTIFIER : D
PERIOD
OUTPUT
STRING       :  C =
OUTPUT WITH LINEFEED
IDENTIFIER : C
OUTPUT
STRING       :  D =
OUTPUT WITH LINEFEED
IDENTIFIER : D
GENERATE
IDENTIFIER : ABSOLUTE
IDENTIFIER : A
PERIOD
OUTPUT
STRING       :  ABSOLUTE VALUE OF 4 =
OUTPUT WITH LINEFEED
IDENTIFIER : A
GENERATE
IDENTIFIER : ARCTANGENT
IDENTIFIER : PIQUARTER
GENERATE-1
IDENTIFIER : E
```

```
PERIOD
OUTPUT
STRING       :  ARCTANGENT FOF PI/4     =
OUTPUT WITH LINEFEED
IDENTIFIER : E
GENERATE
IDENTIFIER : COSINE
IDENTIFIER : PIQUARTER
GENERATE-1
IDENTIFIER : E
PERIOD
OUTPUT
STRING       :  COSINE OF PI/4          =
OUTPUT WITH LINEFEED
IDENTIFIER : E
GENERATE
IDENTIFIER : EXPONENT
IDENTIFIER : B
GENERATE-1
IDENTIFIER : E
PERIOD
OUTPUT
STRING       :  EXPONENT OF  4 =
OUTPUT WITH LINEFEED
IDENTIFIER : E
GENERATE
IDENTIFIER : INTEGER
IDENTIFIER : PIQUARTER
GENERATE-1
IDENTIFIER : E
PERIOD
OUTPUT
STRING       :  INTEGER OF PI/4         =
OUTPUT WITH LINEFEED
IDENTIFIER : E
GENERATE
IDENTIFIER : LOGARITHM
IDENTIFIER : B
GENERATE-1
IDENTIFIER : E
PERIOD
OUTPUT
STRING       :  LOGARITHM OF  4 =
OUTPUT WITH LINEFEED
IDENTIFIER : E
GENERATE
IDENTIFER : MEMORYVALUE
IDENTIFER : B
GENERATE=1
IDENTIFER : F
PERIOD
OUTPUT
```

```
STRING       :  CONTENTS OF ADDRESS 4       =
OUTPUT WITH LINEFEED
IDENTIFIER : F
GENERATE
IDENTIFIER : RANDOM
IDENTIFIER : A
GENERATE-1
IDENTIFER : G
PERIOD
OUTPUT
STRING       :  RANDOM OF 4 =
OUTPUT WITH LINEFEED
IDENTIFIER : G
GENERATE
IDENTIFIER : SIGN
IDENTIFIER : B
GENERATE-1
IDENTIFIER : G
PERIOD
OUTPUT
STRING       :  SIGN OF 4 =
OUTPUT WITH LINEFEED
IDENTIFIER : G
GENERATE
IDENTIFIER :OSINE
IDENTIFIER : PIQUARTER
GENERATE-1
IDENTIFIER : E
PERIOD
OUTPUT
STRING       :  SINE OF PI/4        =
OUTPUT WITH LINEFEED
IDENTIFIER : G
GENERATE
IDENTIFIER : SQUAREROOT
IDENTIFIER : B
GENERATE-1
IDENTIFIER : G
PERIOD
OUTPUT
STRING       :  SQUAREROOT OF 4 =
OUTPUT WITH LINEFEED
IDENTIFIER : G
GENERATE
IDENTIFIER : TANGENT
IDENTIFIER : PIQUARTER
GENERATE-1
IDENTIFIER : F
PERIOD
OUTPUT
STRING       :  TANGENT OF PI/4       =
OUTPUT WITH LINEFEED
```

```
IDENTIFER : F
OUTPUT WITH LINE FEED
STRING      :  DID EVERYTHING WORK?
PROGRAM IDENTIFER CHECK
IDENTIFER : FUNCTION
PROGRAM END
PROGRAM  END
```

Example program "EVEN":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : EVEN
DEFINITIONS START
IDENTIFIER : NUMBER
IDENTIFIER : TEMP
IDENTIFIER : TWO
IDENTIFIER : ZERO
IDENTIFIER : TEMPV
DEFINITIONS END
CLEARSCREEN
LINEFEED
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 2
NO   EXPONENT
IDENTIFIER : TWO
OUTPUT
STRING       : PLEASE ENTER A POSITIVE INTEGER.
INPUT
IDENTIFIER : NUMBER
LINEFEED
ASSIGN
VARIABLE
IDENTIFIER : NUMBER
IDENTIFIER : TEMP
DIVIDE
IDENTIFIER : TEMP
IDENTIFIER : TWO
IDENTIFIER : TEMP
GENERATE
IDENTIFIER : INTEGER
IDENTIFIER : TEMP
PERIOD
DIVIDE
IDENTIFIER : NUMBER
IDENTIFIER : TWO
IDENTIFIER : TEMPV
SUBTRACT
IDENTIFIER : TEMP
IDENTIFIER : TEMPV
IDENTIFIER : TEMP
```

```
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 0
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : ZERO
DECISION
IDENTIFIER : TEMP
OPERATOR =
IDENTIFIER : ZERO
OUTPUT
IDENTIFIER : NUMBER
OUTPUT WITH LINEFEED
STRING       : THIS IS AN EVEN NUMBER.
ELSE
OUTPUT
IDENTIFIER : NUMBER
OUTPUT WITH LINEFEED
STRING       : THIS IS AN ODD NUMBER.
DECISION END
PROGRAM IDENTIFER CHECK
IDENTIFIER : EVEN
PROGRAM END
PROGRAM  END
```

Example program "SELECTION":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : SELECTION
DEFINITIONS START
IDENTIFIER : TEST
IDENTIFIER : ONE
DEFINITIONS END
CLEARSCREEN
LINEFEED
OUTPUT
STRING       : OUTPUT TO PRINTER (1) OR SCREEN(2) ?
INPUT
IDENTIFIER : TEST
LINEFEED
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 1
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : ONE
DECISION
IDENTIFIER : TEST
OPERATOR =
```

```
IDENTIFIER : ONE
DEVICE
IDENTIFIER : PRINTER
OUTPUT WITH LINEFEED
STRING       : OUTPUT TO THE PRINTER.
DEVICE
IDENTIFIER : SCREEN
ELSE
OUTPUT WITH LINEFEED
STRING       : OUTPUT ON THE SCREEN
DECISION END
PROGRAM IDENTIFER CHECK
IDENTIFIER : SELECTION
PROGRAM END
PROGRAM  END
```

Example program "CONDITIONALTEST":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : CONDITIONALTEST
DEFINITIONS START
IDENTIFIER : THREE
IDENTIFIER : FOUR
DEFINITIONS END
CLEARSCREEN
LINEFEED
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 3
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : THREE
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 4
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : FOUR
OUTPUT WITH LINEFEED
STRING       : CONDITIONS
DECISION
IDENTIFIER : THREE
OPERATOR =
IDENTIFIER : FOUR
OUTPUT WITH LINEFEED
STRING       : ERROR ON  =!
ELSE
OUTPUT WITH LINEFEED
STRING       : NO ERROR ON  =!
```

```
DECISION
IDENTIFIER : THREE
OPERATOR >
IDENTIFIER : FOUR
OUTPUT WITH LINEFEED
STRING       : ERROR ON > !
ELSE
OUTPUT WITH LINEFEED
STRING       : NO ERROR ON  > !
DECISION
IDENTIFIER : THREE
OPERATOR >=
IDENTIFIER : FOUR
OUTPUT WITH LINEFEED
STRING       : ERROR ON  >= !
ELSE
OUTPUT WITH LINEFEED
STRING       : NO ERROR ON  >= !
DECISION
IDENTIFIER : THREE
OPERATOR /=
IDENTIFIER : FOUR
OUTPUT WITH LINEFEED
STRING       : NO ERROR ON  /= !
DECISION
IDENTIFIER : THREE
OPERATOR <
IDENTIFIER : FOUR
OUTPUT WITH LINEFEED
STRING       : NO ERROR ON  < !
DECISION
IDENTIFIER : THREE
OPERATOR <=
IDENTIFIER : FOUR
OUTPUT WITH LINEFEED
STRING       : NO ERROR ON  <= !
ELSE
OUTPUT WITH LINEFEED
STRING       : ERROR ON  <= !
DECISION END
ELSE
OUTPUT WITH LINEFEED
STRING       : ERROR ON  < !
DECISION END
ELSE
OUTPUT WITH LINEFEED
STRING       : ERROR ON  /= !
DECISION END
DECISION END
DECISION END
DECISION END
PROGRAM IDENTIFER CHECK
```

```
IDENTIFIER : CONDITIONALTEST
PROGRAM END
PROGRAM  END
```

Program example "INFINITE":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : INFINITE
DEFINITIONS START
IDENTIFIER : NUMBER
IDENTIFIER : ONE
DEFINITIONS END
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 1
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : ONE
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 0
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : NUMBER
INFINITE LOOP
IDENTIFIER : INT
OUTPUT WITH LINEFEED
IDENTIFIER : NUMBER
ADD
IDENTIFIER : ONE
IDENTIFIER : NUMBER
IDENTIFIER : NUMBER
INFINITE LOOP END
IDENTIFIER : INT
PROGRAM IDENTIFER CHECK
IDENTIFIER : INFINITE
PROGRAM END
PROGRAM  END
```

Program example "HUNDRED":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : HUNDRED
DEFINITIONS START
IDENTIFIER : NUMBER
IDENTIFIER : ONE
IDENTIFIER : HUNDRED
DEFINITIONS END
```

```
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 1
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : ONE
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 0
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : NUMBER
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 101
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : HUNDRED
INFINITE LOOP
IDENTIFIER : INT
OUTPUT WITH LINEFEED
IDENTIFIER : NUMBER
ADD
IDENTIFIER : ONE
IDENTIFIER : NUMBER
IDENTIFIER : NUMBER
EXIT
IDENTIFIER : INT
IDENTIFIER : HUNDRED
OPERATOR =
IDENTIFIER : NUMBER
INFINITE LOOP END
IDENTIFIER : INT
PROGRAM IDENTIFER CHECK
IDENTIFIER : HUNDRED
PROGRAM END
PROGRAM  END
```

Program example "EXITLOOPTEST":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : EXITLOOPTEST
DEFINITIONS START
IDENTIFIER : SAM
IDENTIFIER : GEORGE
IDENTIFIER : ONE
DEFINITIONS END
ASSIGN
```

```
FLOATING-POINT NUMBER
INTEGER  : 1
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : ONE
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 0
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : GEORGE
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 100
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : SAM
INFINITE LOOP
IDENTIFIER : OUTER
INFINITE LOOP
IDENTIFIER : INNER
OUTPUT WITH LINEFEED
IDENTIFIER : GEORGE
EXIT
IDENTIFIER : OUTER
IDENTIFIER : GEORGE
OPERATOR =
IDENTIFIER : SAM
ADD
IDENTIFIER : ONE
IDENTIFIER : GEORGE
IDENTIFIER : GEORGE
INFINITE LOOP END
IDENTIFIER : INNER
INFINITE LOOP END
IDENTIFIER : OUTER
PROGRAM IDENTIFER CHECK
IDENTIFIER : EXITLOOPTEST
PROGRAM  END
PROGRAM  END
```

Program example "EXITTEST":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : EXITTEST
DEFINITIONS START
IDENTIFIER : SAM
IDENTIFIER : GEORGE
```

```
IDENTIFIER : HAROLD
DEFINITIONS END
CLEARSCREEN
LINEFEED
OUTPUT WITH LINEFEED
STRING       : TEST OF THE EXIT INSTRUCTION.
LINEFEED
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 2
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : SAM
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 2
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : HAROLD
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 7
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : GEORGE
INFINITE LOOP
IDENTIFIER : LOOPA
INFINITE LOOP
IDENTIFIER : LOOPB
INFINITE LOOP
IDENTIFIER : LOOPC
INFINITE LOOP
IDENTIFIER : LOOPD
INFINITE LOOP
IDENTIFIER : LOOPE
INFINITE LOOP
IDENTIFIER : LOOPF
INFINITE LOOP
IDENTIFIER : LOOPG
INFINITE LOOP
IDENTIFIER : LOOPH
EXIT
IDENTIFIER : LOOPH
IDENTIFIER : SAM
OPERATOR /=
IDENTIFIER : GEORGE
OUTPUT WITH LINEFEED
STRING       : ERROR IN EXIT LOOPH.
INFINITE LOOP END
```

```
IDENTIFIER : LOOPH
OUTPUT WITH LINEFEED
STRING       : EXIT LOOPH OK.
EXIT
IDENTIFIER : LOOPG
IDENTIFIER : SAM
OPERATOR <
IDENTIFIER : GEORGE
OUTPUT WITH LINEFEED
STRING       : ERROR IN EXIT LOOPG.
INFINITE LOOP END
IDENTIFIER : LOOPG
OUTPUT WITH LINEFEED
STRING       : EXIT LOOPG OK.
EXIT
IDENTIFIER : LOOPF
IDENTIFIER : SAM
OPERATOR <=
IDENTIFIER : GEORGE
OUTPUT WITH LINEFEED
STRING       : ERROR IN EXIT LOOPF.
INFINITE LOOP END
IDENTIFIER : LOOPF
OUTPUT WITH LINEFEED
STRING       : EXIT LOOPF OK.
EXIT
IDENTIFIER : LOOPE
IDENTIFIER : SAM
OPERATOR <=
IDENTIFIER : HAROLD
OUTPUT WITH LINEFEED
STRING       : ERROR IN EXIT LOOPE.
INFINITE LOOP END
IDENTIFIER : LOOPE
OUTPUT WITH LINEFEED
STRING       : EXIT LOOPE OK.
EXIT
IDENTIFIER : LOOPD
IDENTIFIER : GEORGE
OPERATOR >
IDENTIFIER : HAROLD
OUTPUT WITH LINEFEED
STRING       : ERROR IN EXIT LOOPD.
INFINITE LOOP END
IDENTIFIER : LOOPD
OUTPUT WITH LINEFEED
STRING       : EXIT LOOPD OK.
EXIT
IDENTIFIER : LOOPC
IDENTIFIER : GEORGE
OPERATOR >=
IDENTIFIER : HAROLD
```

```
OUTPUT WITH LINEFEED
STRING        : ERROR IN EXIT LOOPC.
INFINITE LOOP END
IDENTIFIER : LOOPC
OUTPUT WITH LINEFEED
STRING        : EXIT LOOPC OK.
EXIT
IDENTIFIER : LOOPB
IDENTIFIER : SAM
OPERATOR >=
IDENTIFIER : HAROLD
OUTPUT WITH LINEFEED
STRING        : ERROR IN EXIT LOOPB.
INFINITE LOOP END
IDENTIFIER : LOOPB
OUTPUT WITH LINEFEED
STRING        : EXIT LOOPB OK.
EXIT
IDENTIFIER : LOOPA
IDENTIFIER : SAM
OPERATOR =
IDENTIFIER : HAROLD
OUTPUT WITH LINEFEED
STRING        : ERROR IN EXIT LOOPA.
INFINITE LOOP END
IDENTIFIER : LOOPA
OUTPUT WITH LINEFEED
STRING        : EXIT LOOPA OK.
OUTPUT WITH LINEFEED
STRING        : TEST PROGRAM DONE.
PROGRAM IDENTIFER CHECK
IDENTIFIER : EXITTEST
PROGRAM END
PROGRAM  END
```

Program example "COUNT":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : COUNT
DEFINITIONS START
IDENTIFIER : INDEX
IDENTIFIER : LOWERBOUND
IDENTIFIER : UPPERBOUND
DEFINITIONS END
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 0
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : LOWERBOUND
```

```
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 9
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : UPPERBOUND
INDEX LOOP
IDENTIFIER : INDEX
IDENTIFIER : LOWERBOUND
IDENTIFIER : UPPERBOUND
LOOP  START
OUTPUT WITH LINEFEED
IDENTIFIER : INDEX
INDEX LOOP END
PROGRAM IDENTIFER CHECK
IDENTIFIER : COUNT
PROGRAM END
PROGRAM  END
```

Program example "PARTEST":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : PARTEST
DEFINITIONS START
IDENTIFIER : INDEXA
IDENTIFIER : INDEXB
IDENTIFIER : INDEXC
IDENTIFIER : LOWERBOUND
IDENTIFIER : UPPERBOUND
IDENTIFIER : NUMBER
IDENTIFIER : ONE
DEFINITIONS END
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 1
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : LOWERBOUND
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 10
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : UPPERBOUND
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 0
DECIMAL PLACES
```

```
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : NUMBER
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 1
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : ONE
INDEX LOOP
IDENTIFIER : INDEXA
IDENTIFIER : LOWERBOUND
IDENTIFIER : UPPERBOUND
LOOP  START
INDEX LOOP
IDENTIFIER : INDEXB
IDENTIFIER : LOWERBOUND
IDENTIFIER : UPPERBOUND
LOOP  START
INDEX LOOP
IDENTIFIER : INDEXC
IDENTIFIER : LOWERBOUND
IDENTIFIER : UPPERBOUND
LOOP  START
ADD
IDENTIFIER : ONE
IDENTIFIER : NUMBER
IDENTIFIER : NUMBER
INDEX LOOP END
INDEX LOOP END
INDEX LOOP END
OUTPUT WITH LINEFEED
STRING       : THE RESULT IS    :
OUTPUT WITH LINEFEED
IDENTIFIER : NUMBER
PROGRAM IDENTIFER CHECK
IDENTIFIER : PARTEST
PROGRAM END
PROGRAM  END
```

Program example "JUMPEX":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : JUMPEX
DEFINITIONS START
IDENTIFIER : ONE
IDENTIFIER : TEST
DEFINITIONS END
ASSIGN
FLOATING-POINT NUMBER
```

```
INTEGER  : 1
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : ONE
JUMP LABEL
IDENTIFIER : START
INPUT
IDENTIFIER : TEST
LINEFEED
DECISION
IDENTIFIER : TEST
OPERATOR =
IDENTIFIER : ONE
ELSE
JUMP
IDENTIFIER : START
DECISION END
PROGRAM IDENTIFER CHECK
IDENTIFIER : JUMPEX
PROGRAM END
PROGRAM  END
```

Program example "SUBRTEST":

```
PROGRAM IDENTIFER DEFINITION
IDENTIFIER : SUBRTEST
DEFINITIONS START
IDENTIFIER : TEST
IDENTIFIER : ONE
DEFINITIONS END
ASSIGN
FLOATING-POINT NUMBER
INTEGER  : 1
DECIMAL PLACES
INTEGER  : 0
NO   EXPONENT
IDENTIFIER : ONE
JUMP LABEL
IDENTIFIER : START
OUTPUT WITH LINEFEED
STRING       : SUBROUTINE A(1)/B(2)
INPUT
IDENTIFIER : TEST
LINEFEED
DECISION
IDENTIFIER : TEST
OPERATOR =
IDENTIFIER : ONE
CALL
IDENTIFIER : A
```

```
ELSE
CALL
IDENTIFIER : B
DECISION END
JUMP
IDENTIFIER : START
PROGRAM IDENTIFER CHECK
IDENTIFIER : SUBRTEST
SUBROUTINE IDENTIFIER
IDENTIFIER : A
OUTPUT WITH LINEFEED
STRING       : SUBROUTINE A.
SUBROUTINE IDENTIFER CHECK
IDENTIFIER : A
SUBROUTINE IDENTIFIER
IDENTIFIER : B
OUTPUT WITH LINEFEED
STRING       : SUBROUTINE B.
SUBROUTINE IDENTIFER CHECK
IDENTIFIER : B
PROGRAM END
PROGRAM  END
```

5. THE SEMANTIC ANALYSIS AND CODE GENERATION

This program reads the trace of the MINI program written to the file "MINI-SYN" by the syntactic analyzer. You saw the "traces" of the sample programs on the previous pages.

The following program contains a subroutine at line 61000 which allows the traces to be outputted in coded form. After loading the program, enter the line "RUN 61000". A row of numbers will appear on the printer which can be decoded with the help of table of the semantic symbols.

During the semantic analysis, an assembly language program will be generated from the MINI program. This program can be loaded into the computer with LOAD "MINI-SRC",8 and can be listed. You can then compare the MINI program with the assembly language program the compiler produces. You can best understand the operation of the semantic analysis and the code generation by following how the MINI programs are compiled.

You can assemble the assembly language program with the assembler. You must enter "MINI-SRC" as the name after starting the assembler.

5.1 THE SEMANTIC ANALYSIS PROGRAM

```
500 DIM B$(500) : DIM T%(500)
510 DIM I(40) : IZ=0 : LZ=0
1000 PRINT "COMPILE WITH TRACE (Y/N) ?"
1010 GET W$ : IF W$="" THEN 1010
1020 IF W$="Y" THEN SP=1
1030 PRINT "OUTPUT TO PRINTER (P) OR TO SCREEN (S) ?"
1040 GET W$ : IF W$="" THEN 1040
1050 IF W$="P" THEN PR=4 : GOTO 1080
1060 IF W$="S" THEN PR=3 : GOTO 1080
1070 GOTO 1040
1080 PRINT "PLEASE INSERT DATA DISK"
1090 PRINT "AND PRESS < RETURN > ."
1100 GET W$ : IF W$="" THEN 1100
1110 OPEN PR,PR
1120 OPEN 15,8,15,"I"
1130 PRINT#15,"S:MINI-SRC"
1140 INPUT#15,EN,EM$,ET,ES
1150 IF EN<>1 THEN GOTO 60020
1160 OU=9 : OPEN OU,8,OU,"0:MINI-SRC,P,W"
1170 GOSUB 60000
1180 IN=8 : OPEN IN,8,IN,"0:MINI-SYN,S,R"
1190 GOSUB 60000
1200 PC=2049 : REM BASIC START ADDRESS IN OF YOUR COMPUTER
1210 P1=INT(PC/256)
1220 P2=PC-P1*256
1230 PRINT#OU,CHR$(P2)CHR$(P1);
2000 REM OUTPUT OF THE SUBROUTINES
2010 OU$="LDA #14" : GOSUB 59000
2020 OU$="JSR $FFD2" : GOSUB 59000
2030 OU$="LDA #144" : GOSUB 59000
2040 OU$="JSR $FFD2" : GOSUB 59000
2050 OU$="LDA #6" : GOSUB 59000
2060 OU$="STA 53281" : GOSUB 59000
2070 OU$="STA 53280" : GOSUB 59000
2080 OU$="JMP DECLARATION" : GOSUB 59000
2090 OU$="ZVOR .M" : GOSUB 59000
2100 OU$="LDA #13" : GOSUB 59000
2110 OU$="JSR $FFD2" : GOSUB 59000
2120 OU$="RTS" : GOSUB 59000
2130 OU$="AUS .M" : GOSUB 59000
2140 OU$="JSR $BDDD" : GOSUB 59000
2150 OU$="LDX #0" : GOSUB 59000
2160 OU$="AUSP .M" : GOSUB 59000
2170 OU$="LDA $0100,X" : GOSUB 59000
2180 OU$="BEQ AUSE" : GOSUB 59000
2190 OU$="JSR $FFD2" : GOSUB 59000
2200 OU$="INX" : GOSUB 59000
2210 OU$="BNE AUSP" : GOSUB 59000
2220 OU$="AUSE .M" : GOSUB 59000
```

```
2230 OU$="RTS" : GOSUB 59000
2240 OU$="EIN .M" : GOSUB 59000
2250 OU$="LDA #$3F" : GOSUB 59000
2260 OU$="JSR $FFD2" : GOSUB 59000
2270 OU$="LDX #0" : GOSUB 59000
2280 OU$="EIN1 .M" : GOSUB 59000
2290 OU$="JSR $FFCF" : GOSUB 59000
2300 OU$="STA $0220,X" : GOSUB 59000
2310 OU$="INX" : GOSUB 59000
2320 OU$="CMP #$D" : GOSUB 59000
2330 OU$="BNE EIN1" : GOSUB 59000
2340 OU$="JSR $FFD2" : GOSUB 59000
2350 OU$="LDA #$02" : GOSUB 59000
2360 OU$="STA $23" : GOSUB 59000
2370 OU$="LDA #$20" : GOSUB 59000
2380 OU$="STA $22" : GOSUB 59000
2390 OU$="DEX" : GOSUB 59000
2400 OU$="TXA" : GOSUB 59000
2410 OU$="JSR $B7B5" : GOSUB 59000
2420 OU$="RTS" : GOSUB 59000
2430 OU$=".C" : GOSUB 59000
2440 OU$=".T "+CHR$(34)+"LINE : "+CHR$(34) : GOSUB 59000
2450 OU$="SPUR .M" : GOSUB 59000
2460 OU$="JSR ZVOR" : GOSUB 59000
2470 OU$="LDY # CH" : GOSUB 59000
2480 OU$="LDA # CL" : GOSUB 59000
2490 OU$="JSR $AB1E" : GOSUB 59000
2500 OU$="RTS" : GOSUB 59000
2990 OU$="DECLARATION .M" : GOSUB 59000
5000 REM SEMANTIC LOOP
5010 GET#IN,I$
5020 GET#IN,I$ : IF I$="" THEN I=0 : GOTO 5040
5030 I=ASC(I$)
5035 IF I=55 THEN 9000
5040 IF I<>253 THEN 6000
5050 GET#IN,I$ : I=ASC(I$)
5060 IF I=1 THEN 10000
5070 IF I=2 THEN 10500
5080 IF I=3 THEN 11000
5090 IF I=4 THEN 11500
5100 IF I=5 THEN 12000
5110 IF I=6 THEN 12500
5120 IF I=7 THEN 13000
5130 IF I=8 THEN 13500
5140 IF I=9 THEN 14000
5150 IF I=10 THEN 14500
5160 IF I=11 THEN 15000
5170 IF I=12 THEN 15500
5180 IF I=13 THEN 16000
5190 IF I=14 THEN 16500
5200 IF I=15 THEN 17000
5210 IF I=16 THEN 17500
```

```
5220 IF I=17 THEN 18000
5230 IF I=18 THEN 18500
5240 IF I=19 THEN 19000
5250 IF I=20 THEN 19500
5260 IF I=21 THEN 20000
5270 IF I=22 THEN 20500
5280 IF I=23 THEN 21000
5290 IF I=24 THEN 21500
5300 IF I=25 THEN 22000
5310 IF I=26 THEN 22500
5320 IF I=27 THEN 23000
5330 IF I=28 THEN 23500
5340 IF I=29 THEN 24000
5350 IF I=30 THEN 24500
5360 IF I=31 THEN 25000
5370 IF I=32 THEN 25500
5380 IF I=33 THEN 26000
5390 IF I=34 THEN 26500
5400 IF I=35 THEN 27000
5410 IF I=36 THEN 27500
5420 IF I=37 THEN 28000
5430 IF I=38 THEN 28500
5440 IF I=39 THEN 29000
5500 PRINT#PR,"SEMANTIC PROGRAM ERROR  !!!!!"
5510 STOP
6000 IF I=254 THEN 30000
6010 IF I=255 THEN 31000
6020 PRINT#PR,"SEMANTIC PROGRAM ERROR  !!!!!"
6030 STOP
9000 REM VARIABLE DEFINITION
9010 IF DF=0 THEN GOSUB 56040
9020 IF DF=0 THEN ME$="VARIABLES ARE DECLARED IN DEFINITION
SECTION!":GOTO58700
9030 GOSUB 56040
9040 BE$=I1$ : TY%=3 : GOSUB 57000
9050 IF SC=1 THEN ME$="IDENTIFER ALREADY DEFINED  !" : GOTO
58700
9060 GOSUB 57500
9070 OU$=BE$+" .BL 5" : GOSUB 59000
9080 GOTO 5020
9090 :
10000 REM PROGRAM IDENTIFER DEFINITION
10005 GOSUB 56000
10010 BE$=I1$ : TY%=1 : GOSUB 57000
10020 IF SC=1 THEN ME$="PROGRAM IDENTIFIER ALREADY DEFINED
!" : GOTO 58700
10030 GOSUB 57500
10040 GOTO 5020
10050 :
10500 REM DEFINITION SECTION START
10505 OU$="JMP DEFINITIONSENDE" : GOSUB 59000
10507 OU$="IMMERSYS .BL 5" : GOSUB 59000
```

```
10510 DF=1 : GOTO 5000
10520 :
11000 REM DEFINITION SECTION END
11005 OU$="DEFINITIONSENDE .M" : GOSUB 59000
11010 DF=0 : GOTO 5000
11020 :
11500 REM PROGRAM IDENTIFIER CHECK
11505 GOSUB 56000
11510 BE$=I1$ : TY%=1 : GOSUB 57000
11520 IF SC=0 THEN ME$="PROGRAM IDENTIFIER NOT DEFINED
!" : GOTO 58700
11540 GOTO 5020
11550 :
12000 REM PROGRAM  END
12010 OU$="LDA #$76" : GOSUB 59000
12020 OU$="LDY #$A3" : GOSUB 59000
12030 OU$="JSR $AB1E" : GOSUB 59000
12040 OU$="JMP $A480" : GOSUB 59000
12050 OU$=".END" : GOSUB 59000
12060 GOTO 5000
12070 :
12500 REM SUBROUTINE IDENTIFIER DEFINITION
12505 GOSUB 56000
12510 BE$=I1$ : TY%=2 : GOSUB 57000
12520 IF SC=1 THEN ME$="PROGRAM IDENTIFIER ALREADY DEFINED
!" : GOTO 58700
12530 GOSUB 57500
12535 OU$="U-"+BE$+" .M" : GOSUB 59000
12540 GOTO 5020
12550 :
13000 REM SUBROUTINE IDENTIFIER CHECK
13005 GOSUB 56000
13010 BE$=I1$ : TY%=2 : GOSUB 57000
13020 IF SC=0 THEN ME$="PROGRAM IDENTIFIER NOT DEFINED
!" : GOTO 58700
13030 OU$="RTS" : GOSUB 59000
13040 GOTO 5020
13050 :
13500 REM OUTPUT
13510 GOSUB 13600
13520 GOTO 5020
13530 :
13600 GOSUB 55000
13610 IF I<>255 THEN ME$="NEW SYMBOL EXPECTED   !" : GOSUB
58500 : RETURN
13630 GOSUB 55000
13640 IF I=55 THEN I1$="" : GOSUB 56040 : GOTO 13800
13650 I1$="" : GOSUB 56340
13652 D9=D9+1
13654 D9$=MID$(STR$(D9),2,LEN(STR$(D9))-1)
13656 OU$="JMP DE"+D9$ : GOSUB 59000
13660 OU$=".C" : GOSUB 59000
```

```
13670 OU$=".T "+CHR$(34)+I1$+CHR$(34) : GOSUB 59000
13675 OU$="DE"+D9$+" .M" : GOSUB 59000
13680 OU$="LDY # CH" : GOSUB 59000
13690 OU$="LDA # CL" : GOSUB 59000
13700 OU$="JSR $AB1E" : GOSUB 59000
13710 RETURN
13800 BE$=I1$ : TY%=3 : GOSUB 57000
13810 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED     !" : GOTO
58700
13820 GOSUB 54000
13830 OU$="JSR AUS" : GOSUB 59000
13840 RETURN
13850 :
14000 REM OUTPUT WITH LINEFEED
14010 GOSUB 13600
14020 OU$="JSR ZVOR" : GOSUB 59000
14030 GOTO 5020
14040 :
14500 REM LINEFEED
14510 OU$="JSR ZVOR" : GOSUB 59000
14520 GOTO 5000
14530 :
15000 REM SCLEARSCREEN
15010 OU$="JSR $E544" : GOSUB 59000
15020 GOTO 5000
15030 :
15500 REM CURSOR COLUMN
15510 GOSUB 56500
15520 IF VAL(I1$)>40 THEN ME$="COLUMN NUMBER GREATER THAN
40":GOTO   58700
15530 IF VAL(I1$)<1 THEN ME$="COLUMN NUMBER LESS THAN 1 " :
GOTO 58700
15540 OU$="SEC" : GOSUB 59000
15550 OU$="JSR $FFF0" : GOSUB 59000
15560 OU$="CLC" : GOSUB 59000
15570 OU$="LDY #"+STR$(VAL(I1$)-1) : GOSUB 59000
15580 OU$="JSR $FFF0" : GOSUB 59000
15590 GOTO 5020
15600 :
16000 REM CURSOR LINE
16010 GOSUB 56500
16020 IF VAL(I1$)>24 THEN ME$="LINE NUMBER GREATER THAN 24!"
: GOTO 58700
16030 IF VAL(I1$)<1 THEN ME$="LINE NUMBER LESS THAN  1 !" :
GOTO 58700
16040 OU$="SEC" : GOSUB 59000
16050 OU$="JSR $FFF0" : GOSUB 59000
16060 OU$="CLC" : GOSUB 59000
16070 OU$="LDX #"+STR$(VAL(I1$)-1) : GOSUB 59000
16080 OU$="JSR $FFF0" : GOSUB 59000
16090 GOTO 5020
16100 :
```

```
16500 REM OUTPUT  ASCII
16510 GOSUB 56500
16520 OU$="LDA #"+I1$ : GOSUB 59000
16530 OU$="JSR $FFD2" : GOSUB 59000
16540 GOTO 5020
16550 :
17000 REM DEVICE
17010 GOSUB 56000
17020 IF I1$="SCREEN" THEN 17200
17030 IF I1$="PRINTER" THEN 17300
17040 ME$="UNKNOWN DEVICE  !  "  : GOTO 58700
17200 REM DEVICE SCREEN
17210 OU$="JSR ZVOR" : GOSUB 59000
17220 OU$="JSR $FFCC" : GOSUB 59000
17230 OU$="LDA #4" : GOSUB 59000
17240 OU$="JSR $FFC3" : GOSUB 59000
17250 GOTO 5020
17300 REM DEVICE PRINTER
17310 OU$="LDA #4" : GOSUB 59000
17320 OU$="STA 184" : GOSUB 59000
17330 OU$="STA 186" : GOSUB 59000
17340 OU$="LDA #7" : GOSUB 59000
17350 OU$="STA 185" : GOSUB 59000
17360 OU$="LDA #0" : GOSUB 59000
17370 OU$="STA 183" : GOSUB 59000
17380 OU$="JSR $FFC0" : GOSUB 59000
17390 OU$="LDX #4" : GOSUB 59000
17400 OU$="JSR $FFC9" : GOSUB 59000
17410 GOTO 5020
17500 REM BORDER
17510 GOSUB 56000
17520 FF=-1
17530 IF I1$="BLACK"   THEN FF=0
17540 IF I1$="WHITE" THEN FF=1
17550 IF I1$="RED" THEN FF=2
17560 IF I1$="CYAN"    THEN FF=3
17570 IF I1$="PURPLE"  THEN FF=4
17580 IF I1$="GREEN" THEN FF=5
17590 IF I1$="BLUE" THEN FF=6
17600 IF I1$="YELLOW" THEN FF=7
17610 IF I1$="ORANGE" THEN FF=8
17620 IF I1$="BROWN" THEN FF=9
17630 IF I1$="LTRED" THEN FF=10
17640 IF I1$="GREYA" THEN FF=11
17650 IF I1$="GREYB" THEN FF=12
17660 IF I1$="GREYC" THEN FF=15
17670 IF I1$="LTGREEN" THEN FF=13
17680 IF I1$="LTBLUE" THEN FF=14
17690 IF FF=-1 THEN ME$="UNKNOWN COLOR !" : GOTO 58700
17700 OU$="LDA #"+STR$(FF) : GOSUB 59000
17710 OU$="STA 53280" : GOSUB 59000
17720 GOTO 5020
```

```
18000 REM BACKGROUND
18010 GOSUB 56000
18020 FF=-1
18030 IF I1$="BLACK" THEN FF=0
18040 IF I1$="WHITE" THEN FF=1
18050 IF I1$="RED" THEN FF=2
18060 IF I1$="CYAN" THEN FF=3
18070 IF I1$="PURPLE" THEN FF=4
18080 IF I1$="GREEN" THEN FF=5
18090 IF I1$="BLUE" THEN FF=6
18100 IF I1$="YELLOW" THEN FF=7
18110 IF I1$="ORANGE" THEN FF=8
18120 IF I1$="BROWN" THEN FF=9
18130 IF I1$="LTRED" THEN FF=10
18140 IF I1$="GREYA" THEN FF=11
18150 IF I1$="GREYB" THEN FF=12
18160 IF I1$="GREYC" THEN FF=15
18170 IF I1$="LTGREEN" THEN FF=13
18180 IF I1$="LTBLUE" THEN FF=14
18190 IF FF=-1 THEN ME$="UNKNOWN COLOR " : GOTO 58700
18200 OU$="LDA #"+STR$(FF) : GOSUB 59000
18210 OU$="STA 53281" : GOSUB 59000
18220 GOTO 5020
18500 REM TYPE
18510 GOSUB 56000
18520 FF=-1
18530 IF I1$="BLACK" THEN FF=144
18540 IF I1$="WHITE" THEN FF=5
18550 IF I1$="RED" THEN FF=28
18560 IF I1$="PURPLE" THEN FF=156
18580 IF I1$="GREEN" THEN FF=30
18590 IF I1$="BLUE" THEN FF=31
18600 IF I1$="YELLOW" THEN FF=158
18610 IF I1$="CYAN" THEN FF=159
18690 IF FF=-1 THEN ME$="UNKNOWN COLOR !" : GOTO 58700
18700 OU$="LDA #"+STR$(FF) : GOSUB 59000
18710 OU$="JSR $FFD2" : GOSUB 59000
18720 GOTO 5020
19000 REM INPUT
19010 GOSUB 56000
19020 BE$=I1$ : TY%=3 : GOSUB 57000
19030 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED     !" : GOTO
58700
19040 OU$="JSR EIN" : GOSUB 59000
19050 GOSUB 53000
19060 GOTO 5020
19400 REM ASSIGN     VARIABLE
19410 GOSUB 56000
19420 BE$=I1$ : TY%=3 : GOSUB 57000
19430 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
19440 GOSUB 54000
```

```
19450 GOSUB 56030
19460 BE$=I1$ : TY%=3 : GOSUB 57000
19470 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
19480 GOSUB 53000
19490 GOTO 5020
19500 REM ASSIGN
19510 VA$=""
19520 GOSUB 55000 : GOSUB 55000 : GOSUB 55000
19524 IF I=41 THEN 19400
19530 GOSUB 56500 : VA$=I1$
19535 GOSUB 55000 : GOSUB 55000
19540 IF I<>49 THEN 19600
19550 VA$=VA$+"." : GOSUB 56500 : VA$=VA$+I1$
19555 GOSUB 55000 : GOSUB 55000
19600 IF I<>51 THEN 19900
19610 VA$=VA$+"E"
19620 GOSUB 55000 : GOSUB 55000
19630 IF I=59 THEN 19700
19640 IF I=52 THEN VA$=VA$+"+"
19650 IF I=53 THEN VA$=VA$+"-"
19700 I1$="" : GOSUB 56540
19710 VA$=VA$+I1$
19900 GOSUB 51000
19910 GOSUB 56000
19920 BE$=I1$ : TY%=3 : GOSUB 57000
19930 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
19940 D9=D9+1
19942 D9$=MID$(STR$(D9),2,LEN(STR$(D9))-1)
19946 OU$="JMP DE"+D9$ : GOSUB 59000
19950 OU$="DD"+D9$+" .M" : GOSUB 59000
19952 OU$=" .B"+STR$(V0) : GOSUB 59000
19953 OU$=" .B"+STR$(V1) : GOSUB 59000
19954 OU$=" .B"+STR$(V2) : GOSUB 59000
19955 OU$=" .B"+STR$(V3) : GOSUB 59000
19956 OU$=" .B"+STR$(V4) : GOSUB 59000
19960 OU$="DE"+D9$+" .M" : GOSUB 59000
19962 HI$=BE$ : BE$="DD"+D9$ : GOSUB 54000
19970 BE$=HI$ : GOSUB 53000
19980 GOTO 5020
19990 :
20000 REM ADD
20010 GOSUB 56000
20020 BE$=I1$ : TY%=3 : GOSUB 57000
20030 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
20040 GOSUB 54000
20050 GOSUB 56030
20060 BE$=I1$ : TY%=3 : GOSUB 57000
20070 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
```

```
20080 OU$="LDY #HB-"+BE$ : GOSUB 59000
20090 OU$="LDA #LB-"+BE$ : GOSUB 59000
20100 OU$="JSR $B867" : GOSUB 59000
20110 GOSUB 56030
20120 BE$=I1$ : TY%=3 : GOSUB 57000
20130 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
20140 GOSUB 53000
20150 GOTO 5020
20500 REM SUBTRACT
20510 GOSUB 56000
20520 BE$=I1$ : TY%=3 : GOSUB 57000
20530 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
20540 GOSUB 54000
20550 GOSUB 56030
20560 BE$=I1$ : TY%=3 : GOSUB 57000
20570 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
20580 OU$="LDY #HB-"+BE$ : GOSUB 59000
20590 OU$="LDA #LB-"+BE$ : GOSUB 59000
20600 OU$="JSR $B850" : GOSUB 59000
20610 GOSUB 56030
20620 BE$=I1$ : TY%=3 : GOSUB 57000
20630 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
20640 GOSUB 53000
20650 GOTO 5020
20920 OU$="JSR U-"+I1$ : GOSUB 59000
21000 REM MULTIPLY
21010 GOSUB 56000
21020 BE$=I1$ : TY%=3 : GOSUB 57000
21030 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
21040 GOSUB 54000
21050 GOSUB 56030
21060 BE$=I1$ : TY%=3 : GOSUB 57000
21070 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
21080 OU$="LDY #HB-"+BE$ : GOSUB 59000
21090 OU$="LDA #LB-"+BE$ : GOSUB 59000
21100 OU$="JSR $BA28" : GOSUB 59000
21110 GOSUB 56030
21120 BE$=I1$ : TY%=3 : GOSUB 57000
21130 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
21140 GOSUB 53000
21150 GOTO 5020
21500 REM DIVIDE
21510 GOSUB 56000
21520 BE$=I1$ : TY%=3 : GOSUB 57000
21530 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
```

```
58700
21540 HI$=BE$
21550 GOSUB 56030
21560 BE$=I1$ : TY%=3 : GOSUB 57000
21570 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
21575 GOSUB 54000
21580 OU$="LDY #HB-"+HI$ : GOSUB 59000
21590 OU$="LDA #LB-"+HI$ : GOSUB 59000
21600 OU$="JSR $BB0F" : GOSUB 59000
21610 GOSUB 56030
21620 BE$=I1$ : TY%=3 : GOSUB 57000
21630 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
21640 GOSUB 53000
21650 GOTO 5020
22000 REM POWER
22010 GOSUB 56000
22020 BE$=I1$ : TY%=3 : GOSUB 57000
22030 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
22040 GOSUB 52000
22050 GOSUB 56030
22060 BE$=I1$ : TY%=3 : GOSUB 57000
22070 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
22080 GOSUB 54000
22090 OU$="JSR $BF7B" : GOSUB 59000
22110 GOSUB 56030
22120 BE$=I1$ : TY%=3 : GOSUB 57000
22130 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
22140 GOSUB 53000
22150 GOTO 5020
22500 REM GENERATE
22510 GOSUB 56000
22520 FF$=""
22530 IF I1$="ABSOLUTE" THEN FF$="$BC58"
22540 IF I1$="ARCTANGENT" THEN FF$="$E30E"
22550 IF I1$="COSINE" THEN FF$="$E264"
22560 IF I1$="EXPONENT" THEN FF$="$BFED"
22570 IF I1$="INTEGER" THEN FF$="$BCCC"
22580 IF I1$="LOGARITHM" THEN FF$="$B9EA"
22590 IF I1$="MEMORYVALUE" THEN FF$="$B80D"
22600 IF I1$="RANDOM" THEN FF$="$E097"
22610 IF I1$="SIGN" THEN FF$="$BC39"
22620 IF I1$="SINE" THEN FF$="$E26B"
22630 IF I1$="SQUAREROOT" THEN FF$="$BF71"
22640 IF I1$="TANGENT" THEN FF$="$E2B4"
22650 IF FF$="" THEN ME$="UNKNOWN FUNCTION  !" : GOTO 58700
22660 GOSUB 56030
22670 BE$=I1$ : TY%=3 : GOSUB 57000
```

```
22680 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
22690 GOSUB 54000
22700 OU$="JSR "+FF$ : GOSUB 59000
22705 GOSUB 55000 : GOSUB 55000
22710 IF I=27 THEN 22800
22720 GOSUB 56000 : GOSUB 55000 : GOSUB 55000
22800 BE$=I1$ : TY%=3 : GOSUB 57000
22810 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
22820 GOSUB 53000
22830 GOTO 5000
23000 REM PERIOD
23500 REM DECISION
23505 EF=EF+1 : IZ=IZ+1 : I(IZ)=EF
23506 EF$=MID$(STR$(I(IZ)),2,LEN(STR$(I(IZ))))
23510 GOSUB 56000
23520 BE$=I1$ : TY%=3 : GOSUB 57000
23530 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
23540 GOSUB 54000
23550 GOSUB 55000 : GOSUB 55000
23560 OP=I
23570 GOSUB 56000
23580 BE$=I1$ : TY%=3 : GOSUB 57000
23590 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
23600 OU$="LDY #HB-"+BE$ : GOSUB 59000
23610 OU$="LDA #LB-"+BE$ : GOSUB 59000
23620 OU$="JSR $BC5B" : GOSUB 59000
23630 IF OP=43 THEN OU$="CMP #0" : GOSUB 59000
23640 IF OP=43 THEN OU$="BEQ TH"+EF$ : GOSUB 59000
23650 IF OP=44 THEN OU$="CMP #0" : GOSUB 59000
23660 IF OP=44 THEN OU$="BNE TH"+EF$ : GOSUB 59000
23670 IF OP=45 THEN OU$="CMP #$FF" : GOSUB 59000
23680 IF OP=45 THEN OU$="BEQ TH"+EF$ : GOSUB 59000
23690 IF OP=46 THEN OU$="CMP #1" : GOSUB 59000
23700 IF OP=46 THEN OU$="BNE TH"+EF$ : GOSUB 59000
23710 IF OP=47 THEN OU$="CMP #1" : GOSUB 59000
23720 IF OP=47 THEN OU$="BEQ TH"+EF$ : GOSUB 59000
23730 IF OP=48 THEN OU$="CMP #$FF" : GOSUB 59000
23740 IF OP=48 THEN OU$="BNE TH"+EF$ : GOSUB 59000
23800 OU$="JMP EL"+EF$ : GOSUB 59000
23810 OU$="TH"+EF$+" .M" : GOSUB 59000
23820 GOTO 5020
24000 REM ELSE
24005 EF$=MID$(STR$(I(IZ)),2,LEN(STR$(I(IZ))))
24010 OU$="JMP EN"+EF$ : GOSUB 59000
24020 OU$="EL"+EF$+" .M" : GOSUB 59000
24030 GOTO 5000
24500 REM DECISION END
24510 EF$=MID$(STR$(I(IZ)),2,LEN(STR$(I(IZ))))
```

```
24520 OU$="EN"+EF$+" .M" : GOSUB 59000
24525 IZ=IZ-1
24530 GOTO 5000
25000 REM INFINITE LOOP
25005 GOSUB 56000
25010 BE$=I1$ : TY%=4 : GOSUB 57000
25020 IF SC=1 THEN ME$="LOOP IDENTIFIER ALREADY DEFINED !" :
GOTO 58700
25030 GOSUB 57500
25035 OU$="E-"+BE$+" .M" : GOSUB 59000
25040 GOTO 5020
25500 REM INFINITE LOOP END
25505 GOSUB 56000
25510 BE$=I1$ : TY%=4 : GOSUB 57000
25520 IF SC=0 THEN ME$="LOOP IDENTIFIER NOT DEFINED !" :
GOTO 58700
25530 OU$="JMP E-"+I1$ : GOSUB 59000
25535 OU$="EE-"+I1$+" .M" : GOSUB 59000
25540 GOTO 5020
26000 REM OUTPUT
26005 GOSUB 56000
26010 BE$=I1$ : TY%=4 : GOSUB 57000
26020 IF SC=0 THEN ME$="LOOP IDENTIFIER NOT DEFINED !" :
GOTO 58700
26030 SS$=BE$
26040 GOSUB 56030
26050 BE$=I1$ : TY%=3 : GOSUB 57000
26060 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
26070 GOSUB 54000
26080 GOSUB 55000 : GOSUB 55000
26090 OP=I
26100 GOSUB 56000
26110 BE$=I1$ : TY%=3 : GOSUB 57000
26120 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
26130 OU$="LDY #HB-"+BE$ : GOSUB 59000
26140 OU$="LDA #LB-"+BE$ : GOSUB 59000
26150 OU$="JSR $BC5B" : GOSUB 59000
26160 D9=D9+1
26170 D9$=MID$(STR$(D9),2,LEN(STR$(D9))-1)
26200 IF OP=43 THEN OU$="CMP #0" : GOSUB 59000
26210 IF OP=43 THEN OU$="BNE SZ-"+D9$ : GOSUB 59000
26220 IF OP=44 THEN OU$="CMP #0" : GOSUB 59000
26230 IF OP=44 THEN OU$="BEQ SZ-"+D9$ : GOSUB 59000
26240 IF OP=45 THEN OU$="CMP #255" : GOSUB 59000
26250 IF OP=45 THEN OU$="BNE SZ-"+D9$ : GOSUB 59000
26260 IF OP=46 THEN OU$="CMP #1" : GOSUB 59000
26270 IF OP=46 THEN OU$="BEQ SZ-"+D9$ : GOSUB 59000
26280 IF OP=47 THEN OU$="CMP #1" : GOSUB 59000
26290 IF OP=47 THEN OU$="BNE SZ-"+D9$ : GOSUB 59000
26300 IF OP=48 THEN OU$="CMP #255" : GOSUB 59000
```

```
26310 IF OP=48 THEN OU$="BEQ SZ-"+D9$ : GOSUB 59000
26440 OU$="JMP EE-"+SS$ : GOSUB 59000
26450 OU$="SZ-"+D9$+" .M" : GOSUB 59000
26480 GOTO 5020
26500 REM INDEX LOOP
26510 GOSUB 56000
26520 BE$=I1$ : TY%=3 : GOSUB 57000
26530 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
26540 DX$=BE$
26550 GOSUB 56030
26560 BE$=I1$ : TY%=3 : GOSUB 57000
26570 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
26580 UX$=BE$
26590 GOSUB 56030
26600 BE$=I1$ : TY%=3 : GOSUB 57000
26610 IF SC=0 THEN ME$="IDENTIFIER NOT DEFINED !" : GOTO
58700
26620 OX$=BE$
26630 S$(SL)=DX$ : SL=SL+1
26640 GOTO 5020
27000 REM LOOP START
27010 BE$=UX$ : GOSUB 54000
27020 BE$=DX$ : GOSUB 53000
27030 OU$="IA-"+S$(SL-1)+" .M" : GOSUB 59000
27040 BE$=DX$ : GOSUB 54000
27050 OU$="LDY #HB-"+OX$ : GOSUB 59000
27060 OU$="LDA #LB-"+OX$ : GOSUB 59000
27070 OU$="JSR $BC5B" : GOSUB 59000
27080 D9=D9+1
27090 D9$=MID$(STR$(D9),2,LEN(STR$(D9))-1)
27100 OU$="CMP #1" : GOSUB 59000
27110 OU$="BNE IA"+D9$ : GOSUB 59000
27120 OU$="JMP IE-"+S$(SL-1) : GOSUB 59000
27130 OU$="IA"+D9$+" .M" : GOSUB 59000
27140 OU$="LDA #$E8" : GOSUB 59000
27150 OU$="LDY #$BF" : GOSUB 59000
27160 OU$="JSR $B867" : GOSUB 59000
27170 BE$=DX$ : GOSUB 53000
27180 GOTO 5000
27500 REM INDEX LOOP END
27505 OU$="JMP IA-"+S$(SL-1) : GOSUB 59000
27510 OU$="IE-"+S$(SL-1)+" .M" : GOSUB 59000
27520 SL=SL-1
27530 GOTO 5000
28000 REM JUMPLABEL
28010 GOSUB 56000
28020 OU$="S-"+I1$+" .M" : GOSUB 59000
28030 GOTO 5020
28500 REM JUMP
28510 GOSUB 56000
```

```
28520 OU$="JMP S-"+I1$ : GOSUB 59000
28530 GOTO 5020
29000 REM CALL
29010 GOSUB 56000
29020 OU$="JSR U-"+I1$: GOSUB 59000
29030 GOTO 5020
30000 REM LINE NUMBER
30010 IF SP=1 THEN OU$="JSR SPUR" : GOSUB 59000
30020 GET#IN,I$ : IF I$="" THEN I$=CHR$(0)
30030 IF SP=1 THEN OU$="LDX #"+STR$(ASC(I$)) : GOSUB 59000
30040 GET#IN,I$ : IF I$="" THEN I$=CHR$(0)
30050 IF SP=1 THEN OU$="LDA #"+STR$(ASC(I$)) : GOSUB 59000
30060 IF SP=1 THEN OU$="JSR $BDCD" : GOSUB 59000
30070 GOTO 5000
30080 :
31000 REM PROGRAM  END
31010 PRINT#OU,CHR$(0)CHR$(0);
31020 CLOSE OU
31030 CLOSE IN
31040 CLOSE 15
31050 CLOSE PR
31055 PRINT : PRINT
31060 PRINT "END OF THE SEMANTIC  ANALYSIS."
31065 PRINT : PRINT
31070 PRINT "THE ASSEMBLER CAN BE LOADED WITH  :"
31075 PRINT : PRINT
31080 PRINT "LOAD"+CHR$(34)+"ASSEMBLER"+CHR$(34)+",8"
31085 PRINT : PRINT
31090 PRINT "THE ASSEMBLY LANGUAGE PROGRAM CAN BE LOADED
WITH :"
31095 PRINT : PRINT
31100 PRINT "LOAD"+CHR$(34)+"MINI-SRC"+CHR$(34)+",8"
31110 END
51000 REM CONVERT NUMBER IN VA$ TO INTERNAL REPRESENTATION
51005 IF VAL(VA$)=0 THEN GOTO 51200
51010 V1=0:V5=0:V6=129:V7=VAL(VA$):IFV7<0THENV1=128:V7=ABS
(V7)
51020 IF V7<1 AND V7<>0 THEN V6=V6-1 : V7=V7*2 : GOTO 51020
51030 V8=V7
51040 IF V8>=2 THEN V8=INT(V8/2) : V5=V5+1 : GOTO 51040
51050 V0=V5+V6
51060 V7=(V7-2^V5)/2^V5
51070 V7=V7*128:V1=V1+INT(V7):V7=V7-INT(V7)
51080 V7=V7*256:V2=INT(V7):V7=V7-V2
51090 V7=V7*256:V3=INT(V7):V7=V7-V3
51100 V7=V7*256:V4=INT(V7):V7=V7-V4
51110 RETURN
51200 V0=0:V1=0:V2=0:V3=0:V4=0: RETURN
52000 REM PUT VARIABLE IN ARG
52010 OU$="LDA #LB-"+BE$ : GOSUB 59000
52020 OU$="LDY #HB-"+BE$ : GOSUB 59000
52030 OU$="JSR $BA8C" : GOSUB 59000
```

```
52040 RETURN
53000 REM LOAD VARIABLE WITH FAC
53010 OU$="LDX #LB-"+BE$ : GOSUB 59000
53020 OU$="LDY #HB-"+BE$ : GOSUB 59000
53030 OU$="JSR $BBD4" : GOSUB 59000
53040 RETURN
54000 REM PUT VARIABLE IN FAC
54010 OU$="LDA #LB-"+BE$ : GOSUB 59000
54020 OU$="LDY #HB-"+BE$ : GOSUB 59000
54030 OU$="JSR $BBA2" : GOSUB 59000
54040 RETURN
54050 :
55000 REM PLACE NEXT SYMBOL
55010 GET#IN,I$ : IF I$="" THEN I=0 : RETURN
55020 I=ASC(I$) : RETURN
55030 :
56000 REM IDENTIFIER EXPECTED
56005 I1$=""
56010 GOSUB 55000
56020 IF I<>255 THEN ME$="NEW SYMBOL EXPECTED   !" : GOSUB
58500 : RETURN
56030 GOSUB 55000
56040 I1$="" : IF I<>55 THEN ME$="IDENTIFIER EXPECTED !" :
GOSUB 58500 : RETURN
56050 GOSUB 55000
56060 IF I=255 THEN RETURN
56070 I1$=I1$+I$ : GOTO 56050
56080 :
56300 REM STRING EXPECTED
56305 I1$=""
56310 GOSUB 55000
56320 IF I<>255 THEN ME$="NEW SYMBOL EXPECTED   !" : GOSUB
58500 : RETURN
56330 GOSUB 55000
56340 IF I<>57 THEN ME$="STRING EXPECTED!" : GOSUB 58500 :
RETURN
56350 GOSUB 55000
56360 IF I=255 THEN RETURN
56370 I1$=I1$+I$ : GOTO 56050
56380 :
56500 REM INTEGER  EXPECTED
56505 I1$=""
56510 GOSUB 55000
56520 IF I<>255 THEN ME$="NEW SYMBOL EXPECTED !" : GOSUB
58500 : RETURN
56530 GOSUB 55000
56540 IF I<>59 THEN ME$="INTEGER EXPECTED !" : GOSUB 58500 :
RETURN
56550 GOSUB 55000
56560 IF I=255 THEN RETURN
56570 I1$=I1$+I$ : GOTO 56050
56580 :
```

```
57000 REM IDENTIFIER NOT YET DEFINED ?
57005 SC=0
57010 FOR T=0 TO L
57020 IF B$(T)=BE$ THEN IF T%(T)=TY% THEN SC=1
57030 NEXT T
57040 RETURN
57050 :
57500 REM ENTER IDENTIFIER
57510 B$(L)=BE$ : T%(L)=TY%
57520 L=L+1
57530 RETURN
57540 :
58000 REM WAIT AFTER ERROR
58010 PRINT#PR,CHR$(13)
58020 PRINT#PR,ME$
58030 PRINT#PR,CHR$(13)
58040 PRINT#PR,"PRESS < RETURN >."
58050 GET W$ : IF W$<>CHR$(13) THEN 58050
58060 GOTO 5000
58070 :
58500 REM WAIT AFTER ERROR
58510 PRINT#PR,CHR$(13)
58520 PRINT#PR,ME$
58530 PRINT#PR,CHR$(13)
58540 PRINT#PR,"PRESS < RETURN >."
58550 GET W$ : IF W$<>CHR$(13) THEN 58550
58560 RETURN
58700 REM WAIT AFTER ERROR
58710 PRINT#PR,CHR$(13)
58720 PRINT#PR,ME$
58730 PRINT#PR,CHR$(13)
58740 PRINT#PR,"PRESS < RETURN >."
58750 GET W$ : IF W$<>CHR$(13) THEN 58750
58760 GOTO 5020
58770 :
59000 REM OUTPUT OF OU$ ON OU
59010 PC=PC+LEN(OU$)+5
59020 P1=INT(PC/256)
59030 P2=PC-P1*256
59040 ZL=ZL+10
59050 Z1=INT(ZL/256)
59060 Z2=ZL-Z1*256
59070 PRINT#OU,CHR$(P2)CHR$(P1)CHR$(Z2)CHR$(Z1);OU$;CHR$(0);
59080 PRINT#PR,STR$(ZL)+" "+OU$
59090 OU$=""
59100 RETURN
59120 :
60000 REM READ ERROR CHANNEL
60010 INPUT#15,EN,EM$,ET,ES : IF EN=0 THEN RETURN
60020 PRINT#PR,CHR$(13)CHR$(13)
60030 PRINT#PR,"ERROR ON  DISKETTE !"
60040 PRINT#PR,"ERROR REPORT : ";EM$
```

```
60050 PRINT#PR,"ERROR NUMBER   : ";EN
60060 PRINT#PR,"TRACK          : ";ET
60070 PRINT#PR,"SECTOR         : ";ES
60080 PRINT#PR,"PROGRAM STOPPED !"
60090 CLOSE PR
60100 CLOSE IN
60110 CLOSE OU
60120 CLOSE 15
60130 END
61000 OPEN8,8,8,"MINI-SYN,S,R"
61010 OPEN4,4
61020 GET#8,I$ : IF ST=64 THEN 61100
61025 IF I$="" THEN I=0 : GOTO 61040
61030 I=ASC(I$)
61040 PRINT#4,I;
61050 GOTO 61020
61100 CLOSE8 : CLOSE4
READY.
```

6. THE 6502 ASSEMBLY LANGUAGE

6.1 A BRIEF INTRODUCTION

This introduction is intended for those who have not done any programming in machine or assembly language. You should make use of this section in order to be able to understand the significance of the individual assembler commands produced in the code generated by the SEMANTIC ANALYSIS program.

Only the commands which we will actually use in the compilation will be discussed. The assembler, the listing of which is printed later, can naturally assemble all of the 6502/6510 assembly language commands.

Once you have become acquainted with the principles of assembly language programming by studying a few commands, you will not find it hard to build upon this knowledge. In our favor is the fact that the microprocessor in our computer is not very complicated and is well suited for learning machine language. This does not mean that the microprocessor is not efficient or effective.

We'll make no distinctions between the 6502 and 6510 (the processor in the C-64) microprocessors. For our purposes, both processors react to the corresponding commands in the same way.

We'll avoid the level of machine language since the assembler can translate assembly language commands directly

into machine language codes. We will restrict ourselves to assembly language because working with machine language directly does not open up any more possibilities for us.

We'll use the following simplified representation of the computer for this section:

The computer consists of memory with individual memory cells or locations and a machine (microprocessor) which can alter the contents of the individual memory cells. The memory cells are numbered from 0 to 65535. It is normal when programming in assembly language to write numbers not only in decimal, but also in hexadecimal (base sixteen).

The transition from base ten to base sixteen has advantages because the construction of computers is closer to the base sixteen system. This takes some getting used to for the beginner and since this introduction is intended for the beginner, we will also use the decimal notation in the following discussion.

Hexadecimal numbers are designated with a "$" prefix. The memory range of our computer therefore extends from $0000 to $FFFF. There is a routine in the disassembler program which can convert numbers from one number system to the other.

In the memory itself are programs which the microprocessor is to execute in order to transport data and change the data stored in other memory locations.

In this section we can concentrate almost exclusively on commands which transport data since we will make maximum use of the operating system programs found within the CBM-64. Because the data will be changed with the help of these

programs, it will be our task to transport the necessary information to the appropriate memory locations so that these programs can process it.

In order to understand what the microprocessor does, we'll take a look at a model of it.

The microprocessor contains some registers with which we can work. We will limit ourselves to the following registers:

- accumulator
- X-register
- Y-register
- status register

The accumulator is the working register of our processor. Most operations have something to do with the accumulator.

The X- and Y-registers are additional registers often used for counters.

We can get information about the condition of the processor from the status register.

Let's take a look at an example:

The contents of memory location 2055 are to be moved to memory location 7256.

To do this we must perform the following: Load the accumulator with the contents of memory location 2055 and transfer the contents of the accumulator to memory location 7256.

The pertaining assembly language instructions:

```
LDA 2055
STA 7256
```

The command abbreviation "LDA" stands for "LoaD the Accumulator with". "STA" means "STore the Accumulator in". These abbreviations are called mnemonics.

Behind the mnemonic is something called an operand. The operands tell the microprocessor the memory location which the command references. Operands can be decimal numbers, hexadecimal numbers, or symbols. Our assembler requires that all symbols begin with a letter, but may contain any characters and can be as long as desired. A symbol is assigned a numerical value within the assembly language program which is then inserted in place of the symbol when the program is translated into machine code. In order to assign a value to a symbol, we must give the assembler an instruction. The instructions which control the assembler are called pseudo-instructions, pseudo-operations, or just pseudo-ops. See the chapter on "THE ASSEMBLER COMMAND SET."

There are various ways in which the microprocessor can arrive at the address which a command references based on the operand.

We will become acquainted with the addressing modes using the LDA command as an example.

Note: The options for addressing are different from command to command. The LDA command has the widest variety available to it.

Direct addressing:

In direct addressing, the operand does not imply an address but a value.

Example:

```
LDA # 77
```

This instruction means: Load the accumulator directly with the number 77. We put a number sign (#) in front of the number so that the assembler can distinguish this type of addressing mode from the others.

Page-zero addressing:

What is the zero-page?

We can imagine the memory as 256 pages (numbered from 0 to 255) in which each page is composed of 256 memory locations (also numbered 0 through 255). The first page in memory is called the zero page. There are a number of commands which have special addressing modes for using zero page. This is because the corresponding machine language codes can be shorter and can also be executed faster by the microprocessor. Variables which will be used often during the course of a program are usually placed in page zero. One should keep the zero page in mind at all times when writing programs and plan accordingly.

In order to make zero-page addressing possible, the operand may only have a value between 0 and 255.

Example:

```
LDA 77
```

The value which is stored in memory location 77 will be loaded into the accumulator.

Page zero with X indexing:

Example:

```
LDX # 1
LDA 77,X
```

These two instructions operate as follows:

1) The X-register is directly loaded with the value 1.

2) The accumulator is loaded with the contents of memory location 77 + X = 78. The contents of the X-register are added to the address given in the operand, yielding the effective address for the load command.

The possibility to address with the help of the index registers gives us the freedom to decide at the time of program execution what memory locations will be referenced.

Page zero with Y indexing:

This addressing is only possible with two commands:

```
     LDX and STX
or
     LDX operand, Y
     STX operand, Y
```

STX is the counterpart of LDX:

While LDX implies that the X-register will be loaded with a value, STX means that the contents of the X-register will be placed into a memory location.

The calculation of an address results in the same manner as for the indexing with the X-register, except that these commands work with the contents of the Y-register.

Absolute addressing:

The operand may assume a value from 0 to 65535 when using absolute addressing, meaning that we can address every memory location.

Example:

```
     LDA 47000
```

This instruction loads the accumulator with the contents of memory location 47000.

Absolute addressing with X-indexing:

The same as zero-page with X-indexing, except that the operand may be in the range from 0 to 65535.

Absolute addressing with Y-indexing:

As for zero-page with Y-indexing, except that the operand may be in the range from 0 to 65535.

This addressing mode applies for a number of additional commands, however.

Absolute indirect addressing:

This addressing mode is valid for only one command, the JMP command, but it is the foundation for further possibilities of address generation. Often one requires the ability to process an address which is actually calculated through some numeric operations. This calculated address is then placed in two memory locations and the command is given the address of the first of the two memory locations which the instruction is to reference. The processor gets the actual address out of these two memory locations. Why do we need two memory locations? We can only store values between 0 and 255 in a single memory location. This makes it possible for us to reference a certain memory location within a page, but we also need to be able to specify the page as well. How do we arrive at the values which are to be placed in the two memory locations? If we have the address represented as a hexadecimal number, this is quite simple: The last two digits give the position within the page and the first two give the page.

Example:

The address required will reside at memory location $AAD2. The actual address is $FFD2. This means that we place the value $D2 in $AAD2 and the value $FF in $AAD3.

Address are always stored in this form (High-byte, Low-byte) in assembly language programming.

The JMP commands must still be explained.

The microprocessor gets its instructions from the memory and when it encounters JMP, it reads the next instruction from the memory location whose address is found after the JMP or is calculated through it.

Example:

```
JMP $FFD2
JMP ($FFD2)
```

The first command uses absolute addressing. The next command to be executed is the one residing in memory location $FFD2.

The second example uses indirect addressing. The next command which will be executed is located at the address stored in memory locations $FFD2 and $FFD3.

Indexed indirect addressing:

This mode of addressing is only possible with the help of the X register.

Example:

```
LDX # 10
LDA (20,X)
```

The address of the memory location whose contents will be loaded into the accumulator is determined as follows:

The value of the X-register is added to the value of the operand, 20 in our case:

```
20 + 10 = 30
```

Now the microprocessor assumes that an address is found in memory locations 30 and 31.

For example: contents of memory location 30 is 10 and the contents of memory location 31 is 1.

This yields the address 10 + 1 * 256 = 266.

The accumulator is then loaded with the contents of memory location 266.

The operand may have a value between 0 and 255.

Indirect indexed addressing:

The index register this time is the Y register.

Example:

```
LDY # 10
LDA (20),Y
```

Here the microprocessor expects an address in memory locations 20 and 21.

For example, the contents of 20 are 10 and the contents of 21 are 1.

This address points to memory location 10 + 1 * 256 = 266.

The contents of the Y register, in our case 10, are added to this address: 266 + 10 = 276. Now the accumulator is loaded with the contents of memory location 276. The operand may only have values from 0 to 255. It must be a zero-page address.

Relative addressing:

This mode of addressing applies only to a certain group of instructions, namely the instructions for conditional jumps. What is a conditional jump?

A conditional jump is a jump which is executed depending on the contents of the status register. This allows us to construct various forms of conditional execution structures.

These conditions are met with the help of the status register. The status register consists of 8 positions (bits) which can be read individually. We want to look at this in the following example:

A bit can contain the information 0 or 1. There is a set of jump instructions, from which we want to examine two:

BEQ and BNE

Example

```
BEQ operand
BNE operand
```

Example 1: A jump is made relative to the operand if Z contains the information 1. If Z is 0, the command immediately following the jump instruction will be executed. A jump can be made to a destination 128 memory locations back or 127 memory locations forward from the memory location in which the jump command is located. This is the reason for the term "relative" addressing. The jump is made relative to the position of the command.

Example 2: When Z = 0 the jump is executed, otherwise the command immediately following the jump instructions will be executed.

The question remains: How does the information get in the status register?

Many commands influence individual positions in the status register and there are also commands with which one can directly influence the positions, and finally, there are commands which can set or clear the positions in the status register depending on the contents of memory locations. We will clarify this a bit more shortly.

Now to the last addressing mode which we want to look at:

Implied addressing:

The addresses are an assumed part of the command in this addressing mode. Here are some of them:

```
DEX   ; decrement the X-register by one

DEY   ; decrement the Y-register by one

INX   ; increment the X-register by one

INY   ; increment the Y-register by one

NOP   ; no operation

RTS   ; return from subroutine

TAX   ; transfer accumulator to X-register

TAY   ; transfer accumulator to Y-register

TYA   ; transfer Y-register to accumulator

TXA   ; transfer X-register to accumulator
```

If the result of one of these commands is zero or if one of them moves a zero, the zero bit in the status register is set to one, while if no zero comes into play, the zero bit is cleared to zero.

The following list of commands is just to familiarize you with the mnemonics and a bit of the functions of various commands, some of which we will encounter later in the section on code generation. More information about all of the assembly language commands can be obtained from a variety of books available on machine language.

```
CMP operand
  ; compare accumulator with memory

CPX operand
  ; compare X-register and memory

CPY operand
  ; compare Y-register and memory

JMP operand
  ; jump to new location

JSR operand
  ; jump to subroutine

LDA operand
  ; load accumulator

LDX operand
  ; load X-register

LDY operand
  ; load Y-register

STA operand
  ; store accumulator

STX operand
  ; store X-register

STY operand
  ; store Y-register
```

This should satisfy us for now, although the 6502 possess still more commands. A comprehensive explanation of all them and their various uses requires a separate book in itself, such as the Machine Language Book for the Commodore 64 from Abacus Software.

This section was intended as a brief introduction on programming in assembly language. This will give you an idea of what happens when the computer executes the commands which we give it through the code generation.

In the following section I would like to explain the command set of the assembler and give a listing of the assembler itself. After this the various commands of the disassembler will be explained and a listing of the disassembler provided.

6.2 THE ASSEMBLER COMMAND SET

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 the assembler presented in this book.

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 pseudo-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 MINI compiler produces. This is perhaps the greatest aid to you. I left this interface to the MINI compiler open, even though there are

faster ways of compiling a program. This allows you to see how the compiler goes about analyzing a MINI 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.

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
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
```

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 MINI and assembly language, and are familiar with how the compiler works, you can try to optimize the assembly source code. This MINI 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!

6.3 THE ASSEMBLER PROGRAM

```
45000 OPEN15,8,15 : PRINT#15,"I" : PRINT CHR$(14)
45010 PRINT"{CLR}{DWN}{DWN}  ****    PLEASE INSERT DATA
****"
45020 PRINT"  ****          DISK!           ****"
45030 PRINT"  ****   PRESS      < RETURN >  ****"
45040 GET F$:IFF$<>CHR$(13) THEN 45040
45050 PRINT#15,"I"
45070 INPUT#15,EN,EM$,ET,ES
45100 IF EN<>0 THEN GOTO 45010
45110 CLOSE15
47000 GOTO 49999
48000 REM
48010 PRINT"{DWN}{DWN}  ****  PRESS      < RETURN > ! ****
"
48020 GET W$:IFW$<>CHR$(13) THEN 48020
48030 SYS 2076
49999 PRINTCHR$(14)
50000 REM************       ASSEMBLE
50001 : OPEN 15,8,15 : PRINT#15,"I"
50002 DIMF$(150),L$(500),L(500),N$(200),N(200),S$(400),
S(400)
50003 L=2 : F=0 : PC=2129 : PM=PC : N=0 : S=0 : L$(0)="CH" :
L$(1)="CL" : PO=0
50004 TI$="000000"
50005 PRINT"{HOME}{19 DWN}{3 LEFT}  NAME OF SOURCE PROGRAM
:"
50006 INPUT"{12 LEFT}";NA$
50007 FOR T=2 TO LEN(NA$) : IF MID$(NA$,T,1)<>" " THEN NEXT
50008 NA$=LEFT$(NA$,T-1) : PRINT"{CLR}"
50010 GOSUB 60000
50011 OPEN 8,8,8,"0:"+NA$+",P,R" : GOSUB 50800 : IF EN <>0
THEN 48000
50012 GET#8,G$,H$ : IF ST<>0 THEN 48000
50014 G=ASC(G$) : H=ASC(H$)
50016 AD=G+256*H : A1=AD-2 : GOTO 50500
50020 GOSUB 50400
50030 IF P=0 THEN GOTO 50500
50050 IF P=59 THEN PRINT"COMMENT  ";: GOTO 50500
50060 IF P=32 THEN GOTO 50700
50065 IF P=34 THEN 50420
50240 IF P>127 AND P<204 THEN GOTO 51000
50250 IF P<128 AND P>32 THEN B$=B$+CHR$(P)
50260 GOTO 50020
50400 G=H:P=G:A1=A1+1:GET#8,H$:IFST<>0THENPRINT:PRINT".END
FORGOTTEN?":GOTO62300
50405 IF H$="" THEN H=0 : GOTO 50415
50410 H=ASC(H$)
50415 RETURN
50420 B$=B$+CHR$(P)
```

```
50430 GOSUB 50400 : IF P=34 THEN B$=B$+CHR$(P) : GOTO 50020
50440 IF P=0 THEN 50030
50450 GOTO 50420
50499 :
50500 REM*************       START OF NEW LINE
50501 :
50503 IF B$<>"" THEN GOSUB 51200
50504 IF T0$="1" THEN F=F+1 : F$(F)=ZN$+"  PSEUDO
INSTRUCTION EXPECTED" :PRINTF$(F)
50505 T0$="N" : F$="N" : EX=0
50507 IF A1=AD THEN 50512
50510 A3=A1 : FOR A2=A3 TO AD-1 : GOSUB 50400 : NEXT A2
50512 AD=G+256*H
50515 GOSUB 50400 : GOSUB 50400
50527 ZN=G+256*H : ZN$="LINE : "+STR$(ZN)
50530 PRINT : PRINT"LINE  :";ZN;
50535 GOSUB 50400
50540 GOTO 50020
50541 :
50700 REM*************       BLANK FOUND
50701 :
50710 IF B$="" THEN 50020
50712 IF LEFT$(B$,1)=CHR$(34) THEN B$=B$+CHR$(P) : GOTO
50020
50715 IF B$="#" OR B$="(" THEN 50020
50717 IF H=32 THEN 50020
50718 IF EX=1 THEN 50020
50719 IF T0$<>"N" AND H>169 AND H<175 THEN GOSUB 50400 :
EX=1 : GOTO 50070
50720 GOSUB 51200
50730 GOTO 50020
50731 :
50750 :
50800 REM ************  ERROR CHANNEL
50810 INPUT#15,EN,EM$,ET,ES : IF EN=0 THEN RETURN
50820 PRINT"{CLR}{DWN}{DWN}  ****    ERROR ON DISKETTE
****" : CLOSE 8
50830 PRINT"{DWN}{DWN}{DWN}{DWN}  ****    ERROR NUMBER
: ";EN
50840 PRINT"{DWN}{DWN}{DWN}  ****    ERROR MESSAGE       :"
: PRINT"{DWN}  ****    ";EM$
50850 PRINT"{DWN}{DWN}{DWN}  ****    TRACK               :
";ET
50860 PRINT"{DWN}{DWN}{DWN}  ****    SECTOR              :
";ES
50870 GOTO 62300
51000 REM*************       CONVERT INTERPRETER CODE
51001 :
51010 H$=""
51012 IF P<140 THEN P9=41116 : P6=P-127 : GOTO 51020
51014 IF P<160 THEN P9=41160 : P6=P-139 : GOTO 51020
51016 IF P<180 THEN P9=41244 : P6=P-159 : GOTO 51020
```

```
51018 P9=49483-8192   : P6=P-179
51020 FOR T1=1 TO P6
51030 P9=P9+1 : IF PEEK(P9)<128 THEN 51030
51040 NEXT T1
51050 P9=P9+1 : IF PEEK(P9)<128 THEN H$=H$+CHR$(PEEK(P9)) :
GOTO 51050
51060 H$=H$+CHR$(PEEK(P9)-128)
51090 B$=B$+H$
51095 GOTO 50020
51096 :
51200 REM*************      TOKEN FOUND    / EVALUATE
51201 :
51203 PRINTB$+" ";
51205 MN$="N"
51207 IF T0$="I" THEN GOSUB 55400 : GOTO 51290
51208 IF B$<>".VAREQU" AND F$="D" THEN GOSUB 54580
51210 IF LEFT$(B$,1)="." THEN GOSUB 51400 : GOTO 51290
51220 IF T0$="P" THEN GOSUB 54000 : GOTO 51290
51222 IF T0$="R" THEN GOSUB 56500 : GOTO 51290
51225 IF T0$="M" THEN GOSUB 54800 : GOTO 51290
51230 IF LEN(B$)=3 AND T0$="N" THEN GOSUB 53000
51240 IF MN$="J" THEN GOTO 51290
51270 IF T0$="1" THEN F=F+1 : F$(F)=ZN$+"  PSEUDO
INSTRUCTION EXPECTED" :PRINTF$(F)
51280 IF T0$="N" THEN LA$=B$ : T0$="1"
51290 B$=""
51300 RETURN
51310 :
51400 REM*************      PSEUDO-OPERATION FOUND
51401 :
51405 PS=0
51410 IF B$=".END" THEN PS=1
51420 IF B$=".EQU" OR B$=".E" THEN PS=2
51430 IF B$=".START" THEN PS=3
51440 IF B$=".BLOCK" OR B$=".BL" THEN PS=4
51450 IF B$=".BYTE" OR B$=".B" THEN PS=5
51460 IF B$=".DBYTE" OR B$=".DB" THEN PS=6
51470 IF B$=".TEXT" OR B$=".T" THEN PS=7
51480 IF B$=".WORD" OR B$=".W" THEN PS=8
51490 IF B$=".VAREQU" OR B$=".V" THEN PS=9
51495 IF B$=".MARKE" OR B$=".M" THEN PS=10
51497 IF B$=".COUNT" OR B$=".C" THEN PS=11
51500 ONPSGOSUB51600,52000,52100,52200,52300,52400,52500,
52600,52700,52800,52900
51510 IF PS=0 THEN PRINT"{DWN}{DWN}**** NO PSEUDO ****"
51515 IF PS<>0 THEN T0$="P"
51520 RETURN
51521 :
51600 REM*************      PSEUDO-OPERATION .END
51601 : CLOSE 8 : GOSUB 50800
51602 PRINT : PRINT"{DWN}{DWN}  ****     PROGRAMM-LENGTH
****"
```

```
51603 IF PO=0 THEN PRINT"  ****          ";PC-PM+2129-2047
51604 IF PO=1 THEN PRINT"  ****          ";PC-PM
51605 PRINT"  ****          BYTES               ****"
51606 GOSUB 61000
51610 PRINT"{DWN}{DWN}  ****          END OF
****"
51640 PRINT"  ****          ERROR CHECK         ****"
51643 PRINT"  ****  PRESS       < RETURN > !    ****"
51645 GET W$ : IF W$<>CHR$(13) THEN 51645
51650 FOR T=1 TO F
51660 PRINT F$(T)
51670 FOR G=1 TO 1000 : NEXT
51680 NEXT
51681 PRINT"  ****  THE MACHINE LANGUAGE PRG.":PRINT"  ****
HAS THE NAME "+NA$+"-OBJ"
51685 GOTO 62300
51691 :
51781 :
52000 REM*************      PSEUDO-OPERATION .EQU
52001 :
52010 PS$="E"
52020 RETURN
52021 :
52100 REM*************      PSEUDO-OPERATION .START
52101 :
52110 PS$="S"
52120 RETURN
52121 :
52200 REM*************      PSEUDO-OPERATION .BLOCK
52201 :
52210 PS$="K"
52220 IF LA$<>"" THEN B$=LA$ : GOSUB 54500
52221 :
52230 IF F$="D" THEN GOSUB 54580
52240 RETURN
52300 REM*************      PSEUDO-OPERATION .BYTE
52301 :
52310 PS$="B"
52320 IF LA$<>"" THEN B$=LA$ : GOSUB 54500
52321 :
52330 IF F$="D" THEN GOSUB 54580
52340 RETURN
52400 REM*************      PSEUDO-OPERATION .DBYTE
52401 :
52410 PS$="D"
52420 IF LA$<>"" THEN B$=LA$ : GOSUB 54500
52421 :
52430 IF F$="D" THEN GOSUB 54580
52440 RETURN
52500 REM*************      PSEUDO-OPERATION .TEXT
52501 :
52510 PS$="T"
```

```
52520 IF LA$<>"" THEN B$=LA$ : GOSUB 54500
52521 :
52530 IF F$="D" THEN GOSUB 54580
52540 RETURN
52600 REM*************      PSEUDO-OPERATION .WORD
52601 :
52610 PS$="W"
52620 IF LA$<>"" THEN B$=LA$ : GOSUB 54500
52621 :
52630 IF F$="D" THEN GOSUB 54580
52640 RETURN
52700 REM*************      PSEUDO-OPERATION .VAREQU
52701 :
52710 PS$="V" : F$="N"
52720 RETURN
52721 :
52800 REM*************      PSEUDO-OPERATION .MARKE
52801 :
52810 T0$="I" : B$=LA$ : GOSUB 54500
52820 IF F$="D" THEN GOSUB 54580
52830 RETURN
52831 :
52900 REM*************      PSEUDO-OPERATION .COUNT
52901 :
52910 T0$="I"
52920 L(0)=INT(PC/256) : L(1)=PC-L(0)*256
52930 RETURN
53000 REM*************      MNEMONIC ASSUMED
53001 :
53005 W=0
53010 T=62
53020 GOSUB 53700
53030 TE$=LEFT$(T$,3)
53035 REM     PRINTTE$,B$
53040 IF B$=TE$ THEN W=VAL(MID$(T$,6,1)) : TS=T : GOTO 53570
53050 IF B$<TE$ THEN T=ASC(MID$(T$,4,1)) : GOTO 53210
53060 T=ASC(MID$(T$,5,1))
53210 IF T=91 THEN W=0 : GOTO 53570
53220 GOTO 53020
53570 IF W=0 THEN MN$="N" : RETURN
53580 IF W=1 THEN GOSUB 55500 : RETURN
53590 IF W=2 THEN GOSUB 55700 : RETURN
53600 GOSUB 56000 : RETURN
53601 :
53700 REM*************      LOAD TT WITH ADDRESS T
53701 :
53705 ON INT(T/10)-2 GOTO 53710,53720,53730,53740,53750,
53760,53770
53710 ON T-34 GOSUB 53801,53802,53803,53804,53805 : RETURN
53720 ON T-39 GOSUB 53806,53807,53808,53809,53810,53811,
53812,53813,53814,53815 : RETURN
53730 ON T-49 GOSUB 53816,53817,53818,53819,53820,53821,
```

```
53822,53823,53824,53825 : RETURN
53740 ON T-59 GOSUB 53826,53827,53828,53829,53830,53831,
53832,53833,53834,53835 : RETURN
53750 ON T-69 GOSUB 53836,53837,53838,53839,53840,53841,
53842,53843,53844,53845 : RETURN
53760 ON T-79 GOSUB 53846,53847,53848,53849,53850,53851,
53852,53853,53854,53855 : RETURN
53770 GOSUB 53856 : RETURN
53771 :
53801 T$="ADC[[369[[[[1265#7K23275#8K3626D[[[[237D[[[[8379[
[#59361[[[[4271[[[[52":RETURN
53802 T$="AND#%329[[[[1225[[J23235$8J3622D[[J4233D[[[[8339[
[$59321?1[[4231[[[[52":RETURN
53803 T$="ASL[[30A[[[[:106[[[[3216[[[[620E[[[[231E,1*183":
RETURN
53804 T$="BCC$'290Y1W1;2":RETURN
53805 T$="BCS[(2B0U1@6;2":RETURN
53806 T$="BEQ[[2F0D1P1;2":RETURN
53807 T$="BIT&-324$7$2322CJ1$423":RETURN
53808 T$="BMI[[230I1O1;2":RETURN
53809 T$="BNE*,2D08111;2":RETURN
53810 T$="BPL[[210G101;2":RETURN
53811 T$="BRK+.100[[[[01":RETURN
53812 T$="BVC[/250F121;2":RETURN
53813 T$="BVS[[270H1Q1;2":RETURN
53814 T$="CLC)8118E3E601":RETURN
53815 T$="CLD[[1D8434501":RETURN
53816 T$="CLI13158:3:501":RETURN
53817 T$="CLV[[1B8[[[[01":RETURN
53818 T$="CMP263C9[[[[12C547=132D5487262CD637323DD467483D9[
[[[93C1616242D1[[[[52":RETURN
53819 T$="CPX[[3E0+1(112E4N7N232EC[[[[23":RETURN
53820 T$="CPY573C0[[[[12C4[[[[32CC[[[[23":RETURN
53821 T$="DEC[[3C6[[[[32D6[[[[62CE[[[[23DE[[[[83":RETURN
53822 T$="DEX4;1CA424401":RETURN
53823 T$="DEY[:188[[[[01":RETURN
53824 T$="EOR[[349[[[[1245:7C23255:8C3624D[[C4235D:6C58359[
[[[9341L1[[4251[[[[52":RETURN
53825 T$="INC9<3E652<132F6[[[[62EE[[[[23FE[[[[83":RETURN
53826 T$="INX[=1E8[[N101":RETURN
53827 T$="INY[[1C8714101":RETURN
53828 T$="JMP0M34CC1:4236CK1K4<3":RETURN
53829 T$="JSR[[320[[[[23":RETURN
53830 T$="LDA?A3A9[[[[12A5A1V132B5B3A362ADB4A423BD[[[[83B9@
3B593A1[[[[42B1[[[[52":RETURN
53831 T$="LDX[[3A2@7B212A6[[[[32B631[[72AE[[[[23BEB15193":
RETURN
53832 T$="LDY@D3A0&1'112A4[[[[32B4@8[[62AC[[[[23BCX1@583":
RETURN
53833 T$="LSR[[34A:1[[:146[[[[3256[[[[624E[[[[235E[[[[83":
RETURN
53834 T$="NOPCE1EA;1N401":RETURN
```

```
53835 T$="ORA[[309[[%11205E7%23215E8%3620DE1%4231D[[[[8319[
[E59301-1[[4211[[[[52":RETURN
53836 T$="PHABI148:2>101":RETURN
53837 T$="PHP[[108E2E401":RETURN
53838 T$="PLAG[168#2>201":RETURN
53839 T$="PLPHK128)1)201":RETURN
53840 T$="ROL[[32A$1[[:126[[[[3236[[[[622E[[[[233E%5M183":
RETURN
53841 T$="RORJL36A#1[[:166[[[[3276[[[[626E#4[[237EJ5A583":
RETURN
53842 T$="RTI[[140[[[[01":RETURN
53843 T$="RTSFS160.1/101":RETURN
53844 T$="SBC[[3E9[[[[12E5[[[[32F5N8;262ED53;323FDN6;483F9[
[[[93E1[[[[42F1[[[[52":RETURN
53845 T$="SECN[138$3$601":RETURN
53846 T$="SEDOR1F8N3N501":RETURN
53847 T$="SEI[[178#3#601":RETURN
53848 T$="STAQ[385T1S13295T2S2628DT3S3239D[[[[8399[[[[9381[
[[[4291[[[[52":RETURN
53849 T$="STXPW38691[[3296[[[[728E[[[[23":RETURN
53850 T$="STY[[384R6[[3294R7[[628C[[[[23":RETURN
53851 T$="TAXTV1AA@2@401":RETURN
53852 T$="TAY[Y1A8A2@101":RETURN
53853 T$="TYAU[198R2Z101":RETURN
53854 T$="TSX[[1BA[[[[01":RETURN
53855 T$="TXAXZ18AR1R301":RETURN
53856 T$="TXS[[19AR5R401":RETURN
53860 :
54000 REM*************      PSEUDO-OPERATION OPERAND
54010 GOSUB 55200
54015 IF PS$="T" THEN B$=B$+CHR$(0)
54020 IF PS$="S" THEN PC=B : PM=PC : PO=1
54030 IF PS$="E" THEN GOTO 54350
54040 IF PS$="K" THEN PC=PC+B : FOR T=1 TO B : C0=0 : GOSUB
60100 : NEXT T
54050 IF PS$="B" AND B<256 THEN C0=B : GOSUB 60100 : PC=PC+1
54060 IF PS$="B" AND B>=256 THEN GOSUB 55400
54070 IF PS$="D" THEN GOSUB 57620 : C0=B2 : GOSUB 60100 :
C0=B1 : GOSUB 60100 : PC=PC+2
54080 IF PS$="W" THEN GOSUB 57620 : GOSUB 57060
54090 IF PS$="T" THEN FORT=2TOLEN(B$)-1:C0=ASC(MID$(B$,T,1
)):GOSUB60100:PC=PC+1 : NEXT T
54100 IF PS$="V" THEN L(VQ)=B
54340 T0$="I" : RETURN
54350 B$=LA$ : GOSUB 54500
54360 IF F$="D" THEN GOSUB 54580 : GOTO 54340
54370 L(L-1)=B : GOTO 54340
54500 REM*************      LABEL FOUND    & ENTERED
54501 :
54510 FOR T=0 TO L
54520 IF L$(T)<>B$ THEN NEXT
54540 IF T<>L+1 THEN F$="D" : VQ=L : RETURN
```

```
54550 L$(L)=B$ : L(L)=PC : VQ=L : L=L+1 : LA$=""
54560 RETURN
54570 :
54580 F=F+1 : F$(F)=ZN$+"  "+B$+"  DOUBLY  DEFINED." :
PRINTF$(F) : RETURN
54800 REM*************       LABEL TO MNEMONIC SEARCH
VARIABLE ADDRESS
54801 :
54805 Y=LEN(B$)
54810 IF B$="ACCU" THEN OP%=1 : CE%=10 : BY=1 : GOSUB 57000
: RETURN
54820 IF LEFT$(B$,1)="#" THEN 58000
54830 IF LEFT$(B$,1)="(" AND RIGHT$(B$,3)=",X)" THEN 58400
54840 IF LEFT$(B$,1)="(" AND RIGHT$(B$,1)=")" THEN 58200
54850 IF LEFT$(B$,1)="(" AND RIGHT$(B$,3)="),Y" THEN 58600
54860 IF RIGHT$(B$,2)=",X" THEN 58800
54870 IF RIGHT$(B$,2)=",Y" THEN 59000
54880 GOTO 59200
54881 :
55200 REM*************  OPERAND B$ - B  DEZ
55201 :
55205 IF LEFT$(B$,1)=CHR$(34) THEN B=0 : RETURN
55210 IF LEFT$(B$,1)="$" THEN GOSUB 57800 : RETURN
55220 IF ASC(LEFT$(B$,1))>47 AND ASC(LEFT$(B$,1))<58 THEN
GOSUB 57900 : RETURN
55230 FOR T=0 TO L : IF L$(T)<>B$ THEN NEXT
55240 IF T=L+1 THEN GOSUB 62400
55245 IF T=L+1 THEN PRINT CHR$(13);B$;"  NOT YET DEFINED."
55250 IF T=L+1AND(CE%=1 ORCE%=4ORCE%=5) THEN B=255 :
S$(S)=B$ : S(S)=PC : S=S+1 : RETURN
55254 REM       PRINT"55254 B=256*256"
55255 IF T=L+1 THEN B=256*256-1 : S$(S)=B$ : S(S)=PC : S=S+1
: RETURN
55257 IF K7=1 THEN K7=0 : RETURN
55260 B=L(T) : RETURN
55261 :
55300 REM*************       HEX TO DEC   B$-B
55301 :
55310 HE$=MID$(B$,2,LEN(B$)-1)
55320 GOSUB 59700
55330 B=DE : RETURN
55331 :
55400 REM*************       NO OPERAND POSSIBLE   - ERROR
55401 :
55410 F=F+1
55420 F$(F)=ZN$+"  "+B$+"  NOT POSSIBLE AS OPERAND." :
PRINTF$(F)
55430 RETURN
55431 :
55500 REM*************       ONE-BYTE COMMAND
55501 :
55510 ME$=B$ : OP%=1 : CE%=0
```

```
55520 GOSUB 55600
55530 C0=CO : GOSUB 60100
55540 PC=PC+1
55550 T0$="I" : MN$="J"
55560 RETURN
55561 :
55600 REM************   ESTABLISH MACHINE CODE CO
55601 :
55610 T1=LEN(T$)-6
55620 FOR T=0 TO T1/8-1
55630 IF CE%=ASC(MID$(T$,13+T*8,1))-48 THEN 55685
55640 NEXT T
55660 F=F+1
55670 F$(F)=ZN$+"  ADDRESSING NOT ALLOWED." : PRINTF$(F)
55680 CO=234 : RETURN
55681 :
55685 IF OP%<>VAL(MID$(T$,14+T*8,1)) THEN 55660
55690 HE$=MID$(T$,7+T*8,2) : GOSUB 59700
55695 CO=DE : RETURN
55699 :
55700 REM************  COMMAND WITH RELATIVE ADDRESSING
55701 :
55710 ME$=B$ : OP%=2 : CE%=11
55720 GOSUB 55600
55730 C0=CO : GOSUB 60100
55740 PC=PC+1
55750 T0$="R" : MN$="J"
55760 RETURN
55761 :
56000 REM************  COMMAND WITH VARIABLE ADDRESSING
56001 :
56010 ME$=B$ : T0$="M" : MN$="J" : RETURN
56011 :
56500 REM************      OPERAND WITH RELATIVE ADDRESS
56501 :
56510 IF LEFT$(B$,1)<>"$" THEN GOTO 56600
56520 GOSUB 57800 : GOSUB 55300
56530 IF B>255 THEN 55670
56540 C0=B : GOSUB 60100
56550 PC=PC+1
56560 T0$="I"
56570 RETURN
56571 :
56600 IF ASC(LEFT$(B$,1))>47 AND ASC(LEFT$(B$,1))<58 THEN
GOSUB 57900 : GOTO 56530
56601 :
56610 FOR T=0 TO L : IF L$(T)<>B$ THEN NEXT
56620 IF T=L+1 THEN N$(N)=B$ : N(N)=PC : N=N+1 : B=255 :
GOTO 56530
56630 B=255-PC+L(T) : GOTO 56530
56631 :
57000 REM************      SEARCH FOR CO & POKE
```

```
57001 :
57010 IF F$="J" THEN T0$="I" : RETURN
57020 GOSUB 55600
57030 C0=CO : GOSUB 60100 : PC=PC+1
57040 IF BY=1 THEN T0$="I" : RETURN
57050 IF BY=2 THEN C0=B1 : GOSUB 60100 : PC=PC+1 : T0$="I" :
RETURN
57060 C0=B1:GOSUB 60100 : C0=B2 : GOSUB 60100 : PC=PC+2 :
T0$="I" : RETURN
57061 :
57590 REM************* TEST OPERAND AND SET B1/B2
57591 :
57600 IF B<0 THEN 57650
57610 IF B<256 THEN BY=2 : B1=B : B2=0 : RETURN
57620 IF B>65536 THEN 57650
57630 BY=3 : B2=INT(B/256) : B1=B-B2*256 : RETURN
57650 F=F+1 : F$(F)=ZN$+"  "+STR$(B)+"  NOT LEGAL." :
PRINTF$(F) : F$="J" : RETURN
57699 :
57700 REM*************       TEST IF HEX NUMBER
57701 :
57800 FOR T=2 TO LEN(B$) : TE=ASC(MID$(B$,T,1))
57810 IF TE>47 AND TE<58 OR TE>64 AND TE<71 THEN NEXT :
GOSUB 55300 : RETURN
57830 F=F+1 : F$(F)=ZN$+"  "+B$+"  NOT A HEX NUMBER." :
PRINTF$(F) : F$="J"
57840 B=0 : RETURN
57889 :
57890 REM*************       TEST IF DECIMAL NUMBER
57891 :
57900 FOR T=1 TO LEN(B$)-1 : TE=ASC(MID$(B$,T,1))
57910 IF TE>47 AND TE<58 THEN NEXT : B=VAL(B$) : RETURN
57930 F=F+1 : F$(F)=ZN$+"  "+B$+"  NOT A DECIMAL NUMBER." :
PRINTF$(F) : F$="J"
57950 B=0 : RETURN
57979 :
57980 REM************* SEPARATING OPERANDS BY MNEMONIC
57981 :
58000 B$=MID$(B$,2,Y-1) : CE%=1 : GOSUB 59400 : OP%=BY :
GOTO 57000
58200 B$=MID$(B$,2,Y-2) : GOSUB 59400 : OP%=3 : CE%=12 :
GOTO 57000
58400 B$=MID$(B$,2,Y-4) : CE%=4 : GOSUB 59400 : OP%=BY :
GOTO 57000
58600 B$=MID$(B$,2,Y-4) : CE%=5 : GOSUB 59400 : OP%=BY :
GOTO 57000
58800 B$=MID$(B$,1,Y-2) : GOSUB 59400 : OP%=BY : CE%=6
58810 IF BY=3 THEN CE%=8
58820 GOTO 57000
59000 B$=MID$(B$,1,Y-2) : CE%=7 : GOSUB 59400 : OP%=BY
59010 IF BY=3 THEN CE%=9
59020 GOTO 57000
```

```
59200 B$=MID$(B$,1,Y) : CE%=3 : GOSUB 59400 : OP%=BY
59210 IF BY=3 THEN CE%=2
59220 GOTO 57000
59299 :
59300 REM*************          B$ TO B
59301 :
59400 GOSUB 55200 : GOTO 57600
59599 :
59700 REM*************   CONVERT HEX TO DEC
59701 :
59710 DE=0
59720 FOR T=LEN(HE$) TO 1 STEP -1
59730 H2$=MID$(HE$,T,1)
59740 FOR T1=1 TO 16
59750 IF MID$("0123456789ABCDEF",T1,1)<>H2$ THEN NEXT T1
59760 DE=DE+(T1-1)*16^(LEN(HE$)-T)
59770 NEXT T : RETURN
59780 :
60000 REM************* OPEN THE OUTPUT FILE
60010 PRINT#15,"S:"+NA$+"ZW"
60020 INPUT#15,EN,EM$,ET,ES : REM    PRINT "
**";LEFT$(EM$,13);"**"
60030 OPEN 9,8,9,"0:"+NA$+"ZW"+",P,W" : GOSUB 50800 : IF
EN<>0 THEN STOP
60035 PRINT#9,CHR$(0)CHR$(0);
60040 RETURN
60100 REM************* OUTPUT TO DISK
60110 CO%=CO%+1
60120 IF CO%<254 THEN BL$=BL$+CHR$(C0) : RETURN
60130 PRINT#9,BL$;
60135 REM          PRINT"BL$ = ";BL$
60140 BL$=CHR$(C0) : CO%=1
60150 RETURN
60200 REM************* ASSIGN THE LAST BLOCK
60205 REM       PRINT"BL$ =";BL$,LEN(BL$)
60210 PRINT#9,BL$; : RETURN
61000 REM ************      PASS II
61005 IF ST <>64 THEN 50800
61010 GOSUB 60200
61020 CLOSE 9 : GOSUB 50800 : IF EN<>0 THEN 48000
61022 REM     OPEN 9,8,9,NA$+"ZW,P,R"
61024 REM     GET#9,B$ : IF ST<>0 THEN CLOSE 9 : GOTO 61030
61026 REM     IF B$="" THEN PRINT0; : GOTO 61024
61028 REM     PRINT ASC(B$); : GOTO 61024
61030 REM     PRINT ASC(B$) : CLOSE 9 : GOSUB 50800 : IF
EN<>0 THEN STOP
61040 PRINT#15,"S:"+NA$+"-OBJ"
61050 INPUT#15,EN,EM$,ET,ES : REM   PRINT "
**";LEFT$(EM$,13);"**"
61060 OPEN9,8,9,"0:"+NA$+"-OBJ"+",P,W" : GOSUB 50800 : IF
EN<>0 THEN GOTO 48000
61062 P7=PC : PC=PM : T2=INT(PM/256) : T3=PM-T2*256
```

```
61063 IF PO=1 THEN PRINT#9,CHR$(T3)CHR$(T2);
61064 IF PO=0 THEN PRINT#9,CHR$(1 )CHR$(8 );
61065 GOSUB 62100
61066 REM     PRINT"ST=";ST
61070 OPEN 8,8,8,"0:"+NA$+"ZW"+",P,R"
61072 GET#8,B$
61073 GET#8,B$
61090 S(S)=65000 : N(N)=65000 : T2=0 : T3=0
61100 IF S(T2)=65000 AND N(T3)=65000 THEN 61600
61110 IF S(T2)=65000 THEN T4=N(T3) : T5$=N$(T3) : T3=T3+1 :
T6$="R" : GOTO61150
61120 IF N(T3)=65000 THEN
T4=S(T2)+1:T5$=S$(T2):T2=T2+1:T6$="A":GOTO61150
61140 IF S(T2)<N(T3) THEN T4=S(T2)+1 : T5$=S$(T2) :T2=T2+1
:T6$="A" : GOTO 61150
61145 T4=N(T3) : T5$=N$(T3) : T3=T3+1: T6$="R"
61150 FOR T=0 TO L
61160 IF L$(T)<>T5$ THEN NEXT T
61170 IF T=L+1 THEN F=F+1:F$(F)=B$+" NOT DEFINED." : GOTO
61100
61180 IF T6$="R" THEN 61500
61190 B=L(T) : GOSUB 57630
61200 GET#8,G$ : IF G$="" THEN G$=CHR$(0)
61205 IF PC<>T4 THEN PRINT#9,G$; : PC=PC+1 : GOTO 61200
61210 GET#8,H$ : IF H$="" THEN H$=CHR$(0)
61220 IFASC(G$)=255ANDASC(H$)=255THENPRINT#9,CHR$(B1)CHR$
(B2);:PC=PC+2:GOTO61100
61230 IFASC(G$)=255ANDB<256THENPRINT#9,CHR$(B);H$;:PC=PC+2:
GOTO61100
61240 F=F+1:F$(F)="LABEL"+STR$(PC)+" ILLEGAL OR DECLARED
EARLIER."
61250 GOTO 61100
61500 B=L(T)-N(T3-1)-1
61510 IF PC<>T4 THEN GET#8,G$ : IF G$="" THEN G$=CHR$(0)
61515 IF PC<>T4 THEN PRINT#9,G$; : PC=PC+1 : GOTO 61510
61520 IF B>127 THEN F=F+1:F$(F)="NO RELATIVE JUMP
"+STR$(PC)+" POSSIBLE."
61530 IF B>127 THEN PC=PC+1 : GOTO 61100
61540 PRINT#9,CHR$(B); : PC=PC+1 : GET#8,G$ : GOTO 61100
61600 GET#8,G$ : IF G$="" THEN G$=CHR$(0)
61601 IF ST=0 THEN PRINT#9,G$; : GOTO 61600
61603 PRINT#9,G$;
61605 PRINT#9,CHR$(0)CHR$(0)CHR$(0);
61610 CLOSE 9 : GOSUB 50800 : IF EN<>0 THEN 48000
61620 CLOSE 8 : GOSUB 50800 : IF EN<>0 THEN 48000
61630 PRINT#15,"S:"+NA$+"ZW"
61640 INPUT#15,EN,EM$,ET,ES : REM    PRINT EM$
61645 CLOSE
```

6.4 THE DISASSEMBLER

The disassembler is required 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 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. 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 a 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 big the program which you have in memory is, 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.

R : 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 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.

6.5 THE DISASSEMBLER PROGRAM

```
9995 REM**************       DISSASSEMBLER
9996 :
10032 :
30020 POKE650,128 : EN=49151 : AN=40960 : LA=AN : A1=0
30027 REM*************       MENU
30029 :
30040 PRINT"{CLR}{DWN} I : INSERT BYTE"
30045 PRINT" E : DELETE BYTE"
30050 PRINT" F : POINTER FORWARD"
30055 PRINT" B : POINTER BACKWARD"
30056 PRINT" Z : DEC TO HEX"
30057 PRINT" X : HEX TO DEC"
30060 PRINT" Y : SYS(";AN;")"
30065 PRINT" A : ADDRESS SET"
30068 PRINT" S : SAVE DISK"
30070 PRINT" L : LOAD DISK"
30071 PRINT" R : FREE SPACE"
30072 PRINT" P : POKE"
30073 PRINT" M : MENU "
30074 PRINT" D : DIS & PRINT"
30076 PRINT" F5: DIS & DISPLAY DEC"
30077 PRINT" F7: DIS & DISPLAY HEX"
30080 GET W$ : IF W$="" THEN GOTO 30080
30085 IF W$="F" THEN LA=LA+1 : PRINT LA;PEEK(LA) : GOTO
30080
30090 IF W$="B" THEN LA=LA-1 : PRINT LA;PEEK(LA):GOTO 30080
30100 IF W$="I" THEN GOSUB 31000 : GOTO 30040
30102 IF W$="Z" THEN GOSUB 32200
30104 IF W$="X" THEN GOSUB 32400
30110 IF W$="E" THEN GOSUB 32000 : GOTO 30040
30120 IF W$=CHR$(135)  THEN GOSUB 44300
30130 IF W$=CHR$(136)  THEN GOSUB 44300
30131 IF W$="R" THEN PRINT"{CLR}{DWN}{DWN}FREE SPACE
:";FRE(8)
30132 IF W$="S" THEN GOSUB 40000
30133 IF W$="L" THEN GOSUB 41000
30135 IF W$="A" THEN INPUT"{CLR}{DWN}{DWN}START ADDRESS ";AN
30136 IF W$="P" THEN PRINT LA; : INPUT PO : POKE LA,PO :
LA=LA+1
30137 IF W$="Y" THEN PRINT"  SYS(";AN;")"; : SYS(AN) :
PRINT"  END"
30138 IF W$="M" THEN 30040
30139 IF W$="D" THEN 44000
30140 IF W$="A" THEN INPUT"{DWN}{DWN} END ADDRESS";EN :LA=AN
30141 :
30150 GOTO 30080
30997 REM*************       INSERT BYTES
30999 :
31000 PRINT"{CLR}{DWN}{DWN}   NUMBER OF BYTES" : INPUT" TO
```

```
INSERT       ";BY
31010 FOR T=EN TO LA+BY STEP-1 : POKE T,PEEK(T-BY) : NEXT
31020 FOR T=LA TO LA+BY-1 : POKE T,234 : NEXT
31030 RETURN
31032 :
31997 REM*************        DELETE BYTES
31999 :
32000 PRINT"{CLR}{DWN}{DWN}  NUMBER OF BYTES" : INPUT" TO
DELETE       ";BY
32010 FOR T=LA TO LA+BY : POKE T,PEEK(T+BY) : NEXT
32020 RETURN
32022 :
32200 REM*************       DEC TO HEX
32202 :
32210 INPUT"{CLR}{DWN}{DWN}{LEFT}{LEFT}DECIMAL NUMBER
{DWN}{DWN}{6 LEFT}";DE
32220 GOSUB 45000
32230 PRINT"{DWN}{DWN}{LEFT}{LEFT}THE HEX EQUIVALENT IS:
{DWN}{DWN} ";HE$
32240 RETURN
32241 :
32400 REM*************       HEX TO DEC
32402 :
32410 INPUT"{CLR}{DWN}{DWN}{LEFT}{LEFT}HEX NUMBER
{DWN}{DWN}{10 LEFT}";HE$
32420 GOSUB 32600
32430 PRINT"{DWN}{DWN}{LEFT}{LEFT}THE DECIMAL NUMBER IS:
{DWN}{DWN} ";DE
32440 RETURN
32441 :
32600 REM*************       CONVERT HEX TO DEC
32602 :
32610 DE=0
32620 FOR T=LEN(HE$) TO 1 STEP-1
32630 H2$=MID$(HE$,T,1)
32640 FOR T1=1 TO 16
32650 IF MID$("0123456789ABCDEF",T1,1)<>H2$ THEN NEXT T1
32660 DE=DE+(T1-1)*16^(LEN(HE$)-T)
32670 NEXT T
32680 RETURN
32681 :
39997 REM*************       SAVE TO DISK
39998 :
40000 PRINT"{CLR}{DWN}":INPUT"   FILE NAME      ";W$
40005 A1=0 : A2=0
40007 PRINT"{DWN}{DWN}  SAVE      "
40010 INPUT"{DWN}{DWN}  FROM ";A1
40020 INPUT"{DWN}{DWN}  TO   ";A2
40025 AC=8 : RX=8 : RY=1 : GOSUB 42200 : SYS 65466
40030 GOSUB 42500
40040 AC=LEN(W$) : RX=175 : RY=2 : GOSUB 42200 : SYS 65469
40050 AC=251 : RX=PEEK(253) : RY=PEEK(254) : GOSUB 42200 :
```

```
SYS 65496
40060 GOSUB 42900
40070 RETURN
40071 :
40997 REM*************      LOAD FROM DISK
40998 :
41000 PRINT"{CLR}{DWN}" : INPUT"   FILE NAME     ";W$
41005 A1=0 : A2=0
41007 PRINT"{DWN}{DWN}  LOAD "
41010 INPUT"{DWN}{DWN}  FROM ";A1 : A2=0
41025 AC=8 : RX=8 : RY=1 : GOSUB 42200 : SYS 65466
41030 GOSUB 42500
41040 AC=LEN(W$) : RX=175 : RY=2 : GOSUB 42200 : SYS 65469
41050 AC=0 : RX=251 : RY=252 : GOSUB 42200 : SYS 65493
41060 GOSUB 42900
41070 RETURN
41071 :
42200 REM*************      SYS
42201 :
42210 POKE 780,AC
42220 POKE 781,RX
42230 POKE 782,RY
42280 RETURN
42281 :
42500 REM*************  SET THE REGISTER 680 FF
42580 REM**    251 := L.B.  START-ADDRESS FOR LOADING /
SAVING
42590 REM**    252 := H.B.  START-ADDRESS FOR LOADING /
SAVING
42600 REM**    253 := L.B.  END-ADDRESS  +1 FOR LOADING
/SAVING
42610 REM**    254 := H.B.  END-ADDRESS  +1 FOR LOADING /
SAVING
42640 REM**    687-700 := NAME
42760 IF A1=0 AND A2=0 THEN A1=AN : A2=EN
42770 POKE 251,A1-INT(A1/256)*256
42780 POKE 252,INT(A1/256)
42790 POKE 253,A2+1-INT((A2+1)/256)*256
42800 POKE 254,INT((A2+1)/256)
42810 FOR T=687 TO 686+LEN(W$)
42820 POKE T,ASC(MID$(W$,T-686,1))
42830 NEXT
42840 RETURN
42841 :
42900 REM*************      READ ERROR CHANNEL
42901 :
42910 OPEN 1,8,15
42920 INPUT#1,EF,EM$,ET,ES
42930 IF EF=0 THEN CLOSE 1 : RETURN
42940 PRINT"{CLR}{DWN}** ERROR ON  DISK **"
42950 PRINT"** ERROR NUMBER  :";EF
42960 PRINT"** ERROR MESSAGE  ";EM$
```

```
42970 PRINT"** TRACK           ";ET
42980 PRINT"** SECTOR            ";ES
42990 CLOSE 1 : RETURN
42991 :
42992 :
44000 REM*************     DIS & PRINT
44002 :
44010 PRINT"{CLR}{DWN}{DWN}  PRINT OUT LIST   "
44030 OPEN4,4
44033 INPUT"{DWN}{DWN}  INPUT IN HEX  (Y/N)";EF$
44035 IF EF$<>"Y" THEN 44050
44040 INPUT"{DWN}{DWN}  FROM ADDRESS";HE$ : GOSUB 32600 :
A1=DE
44045 INPUT"{DWN}{DWN}  TO ADDRESS  ";HE$ : GOSUB 32600 :
A2=DE : GOTO 44065
44050 INPUT"{DWN}{DWN}FROM ADDRESS";A1
44060 INPUT"{DWN}{DWN}TO ADDRESS  ";A2
44065 PRINT"{DWN}{DWN} PRESS <RETURN> TO STOP
{DWN}{DWN}{DWN}{DWN}"
44080 IF A1>A2 THEN CLOSE 4 : GOTO 30027
44085 GET EF$ : IF EF$=CHR$(13) THEN CLOSE 4 : GOTO 30027
44090 GOSUB 48500
44100 PRINT#4,PR$
44110 A1=A1+OE
44120 GOTO 44080
44121 :
44299 :
44300 REM*************     DIS & PRINT HEX / DEC
44302 :
44320 A9=LA : A1=A9
44330 GOSUB 48500
44335 IF W$=CHR$(135) THEN PR$=MID$(PR$,38,LEN(PR$)-35) :
GOTO 44350
44340 PR$=LEFT$(PR$,35)
44350 PRINT PR$
44360 GET EF$ : IF EF$=CHR$(13) THEN LA=A1 : RETURN
44370 A1=A1+OE : GOTO 44330
44371 :
45000 REM*************      CONVERT DEC- HEX 4 - DIGIT
45002 EL=3
45005 HE$=""
45010 FOR D=EL TO 0 STEP -1
45020 H1=16^D : H2=INT(DE/H1) :
HE$=HE$+MID$("0123456789ABCDEF",H2+1,1) : DE=DE-H2*H1
45030 NEXT
45040 RETURN
45050 :
45060 REM*************      CONVERT  DEC - HEX 2 - DIGIT
45062 :
45065 EL=1 : GOSUB 45005 : RETURN
45099 :
45100 REM*************                OE=0 / ERRORR
```

```
45102 :
45110 PR$=PR$+" *****"
45120 HI$=HI$+"  *******"
45130 A1=A1+1
45140 RETURN
45199 :
45200 REM************                  OE=1 / 1 BYTE
45202 :
45210 PR$=PR$+"      "
45220 HI$=HI$+"        "
45230 RETURN
45299 :
45300 REM************                  OE=2 / 2 BYTE
45302 :
45310 DE=PEEK(A1+1) : GOSUB 45060
45320 PR$=PR$+" "+HE$+"   "
45330 DE=PEEK(A1+1) : HI$=HI$+" "+RIGHT$("  "+STR$(DE),3)+
"    "
45340 RETURN
45398 :
45400 REM************                  OE=3 / 3 BYTE
45402 :
45410 FOR T=1 TO 2
45420 DE=PEEK(A1+T) : GOSUB 45060
45430 PR$=PR$+" "+HE$
45440 DE=PEEK(A1+T) : HI$=HI$+" "+RIGHT$("  "+STR$(DE),3)
45450 NEXT
45460 RETURN
45498 :
45500 REM************           ONE BYTE COMMAND
45502 :
45510 PR$=PR$+"          "
45520 RETURN
45698 :
45700 REM************           IMMEDIATE
45702 :
45710 GOSUB 48300
45720 PR$=PR$+" #"+HE$+"   "
45730 HI$=HI$+" #"+DE$+"   "
45740 RETURN
45898 :
45900 REM************           ABSOLUTE
45902 :
45910 GOSUB 48400 : GOSUB 45960
45920 PR$=PR$+"   "
45930 HI$=HI$+"   "
45940 RETURN
45960 PR$=PR$+"  "+HE$ : HI$=HI$+"  "+DE$ : RETURN
46098 :
46100 REM************           ZERO-PAGE
46102 :
46110 GOSUB 48300 : GOSUB 45960
```

```
46120 PR$=PR$+"    "
46130 HI$=HI$+"    "
46140 RETURN
46298 :
46300 REM*************           (IND,X)
46302 :
46310 GOSUB 48300
46320 PR$=PR$+" ("+HE$+",X)"
46330 HI$=HI$+" ("+DE$+",X)"
46340 RETURN
46499 :
46500 REM*************           (IND),Y
46502 :
46510 GOSUB 48300
46520 PR$=PR$+" ("+HE$+"),Y"
46530 HI$=HI$+" ("+DE$+"),Y"
46540 RETURN
46699 :
46700 REM*************           ZERO PAGE   ,X
46702 :
46710 GOSUB 48300 : GOSUB 45960
46720 PR$=PR$+" ,X"
46730 HI$=HI$+" ,X"
46740 RETURN
46898 :
46900 REM*************           ZERO PAGE   ,Y
46902 :
46910 GOSUB 48300 : GOSUB 45960
46920 PR$=PR$+" ,Y"
46930 HI$=HI$+" ,Y"
46940 RETURN
47098 :
47100 REM*************           ABSOLUTE X
47102 :
47110 GOSUB 48400 : GOSUB 45960
47120 PR$=PR$+" ,X"
47130 HI$=HI$+" ,X"
47140 RETURN
47298 :
47300 REM*************           ABSOLUTE Y
47302 :
47310 GOSUB 48400 : GOSUB 45960
47320 PR$=PR$+" ,Y"
47330 HI$=HI$+" ,Y"
47340 RETURN
47499 :
47500 REM*************           ACCUMULATOR
47502 :
47510 PR$=PR$+"  ACCU    "
47520 HI$=HI$+"  ACCU    "
47530 RETURN
47698 :
```

```
47700 REM*************          RELATIVE
47702 :
47710 IF PEEK(A1+1)>127 THEN 47750
47720 SP=A1+2+PEEK(A1+1)
47730 GOTO 47800
47740 :
47750 SP=A1-254+PEEK(A1+1)
47760 :
47800 DE=SP : GOSUB 45000
47810 PR$=PR$+"  "+RIGHT$("    $"+HE$,5)+"    "
47820 HI$=HI$+"  "+RIGHT$("     "+STR$(SP),5)+"    "
47830 RETURN
47898 :
47900 REM*************           INDIRECT
47902 :
47910 GOSUB 48400
47920 PR$=PR$+" ("+HE$+")   "
47930 HI$=HI$+" ("+DE$+")   "
47940 RETURN
48098 :
48100 REM*************           ERROR
48102 :
48110 PR$=LEFT$(PR$,16)+"                "
48120 HI$=LEFT$(HI$,23)+"   "
48130 IF OP<32 OR OP>95 THEN HI$=HI$+"NOT ASCII CHARACTER ":
 RETURN
48140 HI$=HI$+"ASCII - CHARACTER : "+CHR$(OP)
48150 RETURN
48299 :
48300 REM*************      RETURN 1 BYTE IN DEC AND HEX
48302 :
48310 DE=PEEK(A1+1) : D9=DE
48320 GOSUB 45060
48340 HE$="  $"+HE$
48350 DE$=RIGHT$("     "+STR$(D9),5)
48360 RETURN
48398 :
48399 :
48400 REM*************      RETURN 2 BYTES IN DEC AND HEX
48402 :
48410 DE=PEEK(A1+1)+PEEK(A1+2)*256 : D9=DE
48420 GOSUB 45000
48440 HE$="$"+HE$
48450 DE$=RIGHT$("   "+STR$(D9),5)
48460 RETURN
48498 :
48500 REM*************     DISASSEMBLER LOOP
48502 :
48510 DE=A1
48520 GOSUB 45000
48530 PR$=HE$ : HI$="    "+STR$(A1)
48540 OP=PEEK(A1)
```

```
48550 GOSUB 49000
48560 DE=OP
48570 GOSUB 45060
48580 PR$=PR$+"  "+HE$
48590 HI$=HI$+"  "+RIGHT$(" "+STR$(OP),3)
48600 ON OE+1 GOSUB 45100,45200,45300,45400
48610 PR$=PR$+"   "+ME$
48615 HI$=HI$+"   "+ME$
48620 IF CE%>7 GOTO 48627
48622 ON CE%+1 GOSUB 45500,45700,45900,46100,46300,46500,
46700,46900
48625 GOTO 48630
48627 ON CE%-7 GOSUB 47100,47300,47500,47700,47900,48100
48630 PR$=PR$+"   "+HI$
48640 RETURN
48641 :
49000 REM************  SEARCH FOR  ME$,OE,CE%
49010 TE$="K" : SE=4
49020 T=ASC(TE$)
49030 GOSUB 53700
49040 HE$=MID$(T$,7+SE*8,2) : GOSUB 32600
49050 IF OP=DE THEN 49500
49060 IF OP<DE THEN TE$=MID$(T$,9+SE*8,1) : SE=VAL(MID$(T$,
10+SE*8,1))-1 : GOTO 49200
49070 TE$=MID$(T$,11+SE*8,1) : SE=VAL(MID$(T$,12+SE*8,1))-1:
 GOTO 49200
49200 IF SE=-1 THEN ME$="*" : OE=0 : CE%=13 : RETURN
49210 GOTO 49020
49500 ME$=LEFT$(T$,3)
49510 OE=VAL(MID$(T$,14+SE*8,1))
49520 CE%=ASC(MID$(T$,13+SE*8,1))-48
49530 RETURN
53700 REM************      LOAD T$ WITH ADDRESS T
53705 ON INT(T/10)-2 GOTO
53710,53720,53730,53740,53750,53760,53770
53710 ON T-34 GOSUB 53801,53802,53803,53804,53805 : RETURN
53720 ON T-39 GOSUB 53806,53807,53808,53809,53810,53811,
53812,53813,53814,53815 : RETURN
53730 ON T-49 GOSUB 53816,53817,53818,53819,53820,53821,
53822,53823,53824,53825 : RETURN
53740 ON T-59 GOSUB 53826,53827,53828,53829,53830,53831,
53832,53833,53834,53835 : RETURN
53750 ON T-69 GOSUB 53836,53837,53838,53839,53840,53841,
53842,53843,53844,53845 : RETURN
53760 ON T-79 GOSUB 53846,53847,53848,53849,53850,53851,
53852,53853,53854,53855 : RETURN
53770 GOSUB 53856 : RETURN
53771 :
53801 T$="ADC[[369[[[[1265#7K23275#8K3626D[[[[237D[[[[8379[
[#59361[[[[4271[[[[52":RETURN
53802 T$="AND#%329[[[[1225[[J23235$8J3622D[[J4233D[[[[8339[
[$59321?1[[4231[[[[52":RETURN
```

```
53803 T$="ASL[[30A[[[[:106[[[[3216[[[[620E[[[[231E,1*183":
RETURN
53804 T$="BCC$'290Y1W1;2":RETURN
53805 T$="BCS[(2B0U1@6;2":RETURN
53806 T$="BEQ[[2F0D1P1;2":RETURN
53807 T$="BIT&-324$7$2322CJ1$423":RETURN
53808 T$="BMI[[230I1O1;2":RETURN
53809 T$="BNE*,2D08111;2":RETURN
53810 T$="BPL[[210G101;2":RETURN

53811 T$="BRK+.100[[[[01":RETURN
53812 T$="BVC[/250F121;2":RETURN
53813 T$="BVS[[270H1Q1;2":RETURN
53814 T$="CLC)8118E3E601":RETURN
53815 T$="CLD[[1D8434501":RETURN
53816 T$="CLI13158:3:501":RETURN
53817 T$="CLV[[1B8[[[[01":RETURN
53818 T$="CMP263C9[[[[12C547=132D5487262CD637323DD467483D9[
[[[93C1616242D1[[[[52":RETURN
53819 T$="CPX[[3E0+1(112E4N7N232EC[[[[23":RETURN
53820 T$="CPY573C0[[[[12C4[[[[32CC[[[[23":RETURN
53821 T$="DEC[[3C6[[[[32D6[[[[62CE[[[[23DE[[[[83":RETURN
53822 T$="DEX4;1CA424401":RETURN
53823 T$="DEY[:188[[[[01":RETURN
53824 T$="EOR[[349[[[[1245:7C23255:8C3624D[[C4235D:6C58359[
[[[9341L1[[4251[[[[52":RETURN
53825 T$="INC9<3E652<132F6[[[[62EE[[[[23FE[[[[83":RETURN
53826 T$="INX[=1E8[[N101":RETURN
53827 T$="INY[[1C8714101":RETURN
53828 T$="JMP0M34CC1:4236CK1K4<3":RETURN
53829 T$="JSR[[320[[[[23":RETURN
53830 T$="LDA?A3A9[[[[12A5A1V132B5B3A362ADB4A423BD[[[[83B9@
3B593A1[[[[42B1[[[[52":RETURN
53831 T$="LDX[[3A2@7B212A6[[[[32B631[[72AE[[[[23BEB15193":
RETURN
53832 T$="LDY@D3A0&1'112A4[[[[32B4@8[[62AC[[[[23BCX1@583":
RETURN
53833 T$="LSR[[34A:1[[:146[[[[3256[[[[624E[[[[235E[[[[83":
RETURN
53834 T$="NOPCE1EA;1N401":RETURN
53835 T$="ORA[[309[[%11205E7%23215E8%3620DE1%4231D[[[[8319[
[E59301-1[[4211[[[[52":RETURN
53836 T$="PHABI148:2>101":RETURN
53837 T$="PHP[[108E2E401":RETURN
53838 T$="PLAG[168#2>201":RETURN
53839 T$="PLPHK128)1)201":RETURN
53840 T$="ROL[[32A$1[[:126[[[[3236[[[[622E[[[[233E%5M183":
RETURN
53841 T$="RORJL36A#1[[:166[[[[3276[[[[626E#4[[237EJ5A583":
RETURN
53842 T$="RTI[[140[[[[01":RETURN
53843 T$="RTSFS160.1/101":RETURN
```

```
53844 T$="SBC[[3E9[[[[12E5[[[[32F5N8;262ED53;323FDN6;483F9[
[[[93E1[[[[42F1[[[[52":RETURN
53845 T$="SECN[138$3$601":RETURN
53846 T$="SEDOR1F8N3N501":RETURN
53847 T$="SEI[[178#3#601":RETURN
53848 T$="STAQ[385T1S13295T2S2628DT3S3239D[[[[8399[[[[9381[
[[[4291[[[[52":RETURN
53849 T$="STXPW38691[[3296[[[[728E[[[[23":RETURN
53850 T$="STY[[384R6[[3294R7[[628C[[[[23":RETURN
53851 T$="TAXTV1AA@2@401":RETURN
53852 T$="TAY[[1A8A2@101":RETURN
53853 T$="TYAUY198R2Z101":RETURN
53854 T$="TSX[[1BA[[[[01":RETURN
53855 T$="TXAXZ18AR1R301":RETURN
53856 T$="TXS[[19AR5R401":RETURN
READY.
```

7. THE C-64 & C-128 OPERATING SYSTEM AND INTERPRETER

A computer does not only consist of a microprocessor and memory, but includes a variety of other devices such as a display screen and printer, and devices for data storage such as disk drives or tape machines. The microprocessor needs special programs so that it can use these devices. These programs are already contained within the computer as subroutines and this package of programs is called the operating system.

In addition, the C-64 and C-128 also contains the programming language BASIC 2.0. In order to be able to understand BASIC instructions, the computer has something called a BASIC interpreter at its disposal. This interpreter is nothing more than a machine language program which analyzes BASIC instructions and can then initiate the appropriate actions.

In this chapter we want to learn how these programs built into our computer can be used for our own purposes. The programs are written as subroutines to which we need only pass the necessary information. This is usually done with the help of the X and Y registers and the accumulator.

The operating system subroutines are used in order to work with external devices, but why would we want to concern ourselves with the interpreter subroutines? There are certain tasks which must be performed in every programming language, such as floating-point arithmetic, or processing conditions, so we can make use of the subroutines to do these sort things already present in the BASIC interpreter.

Since it is very time-consuming to write fast and above all functional machine language programs, one should have quite important reasons for wanting to write or rewrite routines to perform these tasks.

On one hand this chapter is intended specifically for the C-64 and C-128 in C-64 mode, while on the other hand, most computers have an operating system and an interpreter at their disposal which are fairly similar to those in the Commodore 64. Literature is available for many computers which contains the information regarding where the corresponding subroutines can be found in that particular computer and how the parameters must be passed to the operating system subroutines.

First we'll look at the screen output.

Setting the cursor:

For this task we need a subroutine which can be called with different parameters. In order to set the cursor to a specific screen position, we pass the line number in the X-register and the column number in the Y-register. Then we clear the carry bit with "CLC" and jump to the subroutine at $FFF0. In one instruction we will change either the column or the line, but never both. We do not know, however, the line the cursor is in when we change the column. Therefore we must first get the cursor position. This is done with the same subroutine, but we must set the carry bit before we call it. The carry bit is set with "SEC".

The C-64 starts counting the lines and columns with zero instead of one. This means that column 17 would have the number 16 internally.

Example: The cursor is to be set to column 22.

```
SEC
JSR $FFF0
CLC
LDY #21
JSR $FFF0
```

Clearing the screen:

There is a separate subroutine for clearing the screen:

```
JSR $E544
```

Outputting a character on the screen:

To output a character to the screen, we must place its ASCII value in the accumulator and jump to the subroutine at location $FFD2.

Example: Output the character A on the screen.

```
LDA #65
JSR $FFD2
```

Outputting strings on the screen:

There is a routine which outputs an entire string, enclosed in quotation marks, on the screen. To do this we must pass the address of the first character (the quotation mark) in the Y-register and accumulator and jump to the routine at $AB1E.

The page number is given in the accumulator and the position within the page in the Y-register. The page number is also referred to as the most-significant byte of the address and the position within the page as the least-significant byte. In assembly language, the most-significant byte (high byte) is abbreviated to HB address and the least-significant byte to LB address.

Example: Output the string at $1000.

```
ADDRESS .EQU  $1000
LDY LB-ADDRESS
LDA HB-ADDRESS
JSR $AB1E
```

Line feed:

Here the same program used to output a character on the screen is called into play. The ASCII value of line feed is 13 (only a carriage return is required on the screen).

```
LDA #13
JSR $FFD2
```

We will define this function as a subroutine with the name LFEED:

```
LFEED .LABEL
LDA #13
JSR $FFD2
RTS
```

Redirecting output to the printer:

This is a somewhat complicated task. First we must open an output channel for the printer. To do this we must save:

the length of the filename in 183
the logical file number in 184
the secondary address in 185 and
the device number in 186

Since we do not want to give a filename, the length will be zero.

```
LDA #0
STA 183
```

The logical file number and the device number should have the value four:

```
LDA #4
STA 184
STA 186
```

The secondary address has the value 7 in order to put the printer in upper/lower case mode.

```
LDA #7
STA 185
```

Now we jump to the subroutine to open a file:

```
JSR $FFC0
```

The output will be sent to this channel. The "Set output device" subroutine will do this if the channel number is passed in the X-register.

```
LDX #4
JSR $FFC9
```

Now all the output will be sent to the printer.

Directing output back to the screen:

In order to empty the print buffer in the printer, we send a line feed.

```
JSR LFEED
```

After this we must set the output device back to the screen:

```
JSR $FFCC
```

Channel 4 must be closed. We pass the channel number in the accumulator:

```
LDA #4
JSR $FFC3
```

Floating-point arithmetic:

First a few basic considerations:

By operation we mean addition, subtraction, multiplication, division, or exponentiation of two floating-point numbers.

In addition, we will assume that the floating-point numbers are already stored in their internal representations. There are several possibilities for performing an operation. Two will be explained:

Whenever an operation is to be performed, we write in the program text the code for the operation in question. A subroutine cannot be written without something additional since our variables are scattered in memory and each operation must operate on different memory locations. It is, however, a great waste of memory to use the same code (except for the addresses) so many times in a program.

It is for this reason that the next method will be taken:

We choose two memory areas for each floating-point number, place the corresponding variables in these memory locations at the start of the operation and then jump to the subroutine for the appropriate operation which then operates on the two areas. The result can be placed in an area in memory and moved to the memory area for the destination variable. This takes somewhat longer to execute because we must save the contents of memory areas, but it saves an enormous amount of space. In addition, the interpreter contains routines for all of the necessary operations.

There are two memory areas available with which we can perform computations: In abbreviated form the first is called FAC and the second ARG. This is in agreement with the ROM listing in The Anatomy of the Commodore 64. The result of an operation is placed in the FAC (Floating-point ACcumulator).

We will use the following subroutines:

Transfer a variable to the FAC
Transfer the FAC to a variable
Addition of a variable to the FAC
Subtraction of the FAC from a variable
Multiplication of a variable by the FAC
Division of a variable by the FAC
Exponentiation of a variable by the FAC

All operations run according to the same pattern:
Example:

variable 1 operation variable 2 to variable 3

Variable 2 is brought into the FAC.

The operation is called with variable 1.
The FAC is placed in variable 3.

Let's take a look at what we must do in each case:

1) Transferring a variable to the FAC:

The address of the variable is stored in the accumulator and Y-register as follows: The Y-register contains the most-significant byte of the address and the accumulator the least-significant. Then control is passed to the subroutine at $BBA2.

Example: Transfer the variable TOM to the FAC:

```
LDY #HB-TOM
LDA #LB-TOM
JSR $BBA2
```

2) Transfer the FAC to a variable:

Here the destination address is placed in the X and Y registers where the most-significant byte is stored in the Y-register and the least-significant in the X-register. The subroutine begins at $BBD4.

Example: Transfer the FAC to a variable called HARRY.

```
LDX #LB-HARRY
LDY #HB-HARRY
JSR $BBD4
```

3) Addition of a variable to the FAC:

To do this operation we pass the address of the variable in the accumulator and Y-register and jump to the subroutine at $B867. This subroutine first transfers the variable to the memory area ARG and then adds ARG and FAC, placing the result in FAC.

The following operations follow the same pattern; only the addresses of the routines are different.

Example: Add the variable with the name "SALLY" to the FAC.

```
LDY #HB-SALLY
LDX #LB-SALLY
JSR $B867
```

4) Subtraction of the FAC from a variable:

As number 3) with the address $B850.

5) Multiplication of a variable by the FAC:

As number 3) with address $BA28.

6) Division of a variable by the FAC:

As number 3) with address $BB0F.

7) Exponentiation of a variable by the FAC:

For this operation the operating system offers us only the routine ARG to the FAC power, which means we must place the variable in ARG ourselves. This is done the same way as for the FAC, except that a different subroutine is used.

Example: Exponentiation of the variable "TOM" by the FAC.

```
LDY #HB-TOM
LDA #LB-TOM
JSR $BA8C ; TOM TO ARG
JSR $BF7B ; ARG ^ FAC
```

Functions:

To calculate the values of functions we proceed as follows: We place the value, the variable, in the FAC, jump to the subroutine for the appropriate function and then transfer the FAC to the destination variable.

First a list of the functions and the addresses of the corresponding subroutines:

Function	Address
absolute value	$BC58
arctangent	$E30E
cosine	$E264
exponent	$BFED
integer	$BCCC
logarithm	$B9EA
memory value	$B80D
random	$E097
sine	$E26B
square root	$BF71
tangent	$E2B4

Example: Convert the instruction:
"generate absolute of TOM tovar HARRY."
to an assembly-language program

```
;Transfer TOM to the FAC
LDY #HB-TOM
LDA #LB-TOM
JSR $BBA2
;Call the function "absolute"
JSR $BC58
;Transfer the FAC to HARRY:
LDY #HB-HARRY
LDX #LB-HARRY
JSR $BBD4
```

Isn't this a truly elegant way of programming?

7.1 DATA INPUT AND OUTPUT

The interpreter and operating system also place routines for data input and output at our disposal.

We start with output:

Floating-point numbers are stored in an internal representation in the computer. We must convert this form into a form consisting of ASCII characters so that we humans can read it directly.

The conversion routine again works on the FAC, in which we must place our floating-point numbers. This will then be converted and placed in a section of memory which begins at $0100. We will call this section the input/output buffer or just "buffer" for short. This buffer holds 12 characters. The routine writes a binary zero as the last character of the current conversion. This way we always know exactly how long our string is. We can perform the output with the afore-mentioned routine, but we can write a small subroutine ourselves. The first solution is clear, so we will proceed to the second. This routine will be written as a subroutine so that we can call it from different places in the program.

1) Place contents of the variable in the FAC:

```
LDA #LB-name
LDY #HB-name
JSR $BBA2
JSR OUT ; call output routine
```

2) Output routine:

```
OUT .LABEL
JSR $BDDD       ; FAC to $0100
LDX #0          ; clear X reg
OUTP .M         ; output loop
LDA $0100,X     ; load acc with character
BEQ OUTE        ; acc=0 then end
JSR $FFD2       ; output acc
INX             ; increment X reg for next char
BNE OUTP        ; to output loop
OUTE .M         ; loop end
RTS             ; task done
```

Is it difficult to write a small assembly language program?

Some comments:

JSR $BDDD converts the contents of the FAC to the corresponding ASCII character string.

LDA $0100,X loads the character in $0100+X into the accumulator.

BEQ OUTE if a binary zero is loaded into the accumulator, the zero flag will be set and a jump will be made to OUTE.

JSR $FFD2 Otherwise control passes to the output routine which outputs a character (in the accumulator).

BNE OUTP is not clear at first glance. The BNE command is executed faster than the JMP command. Since INX has as result a number other than zero, the zero flag is cleared as a result of the INX. This has the result that the jump is

always executed. Only when the INX functions so that the X-register is changed from 255 to zero is the zero flag set. But since a maximum of 11 characters will be before the binary zero in the buffer, this case never occurs.

Data input:

For input of data we again write a small subroutine which we will always call when we want to initiate input.

```
IN .LABEL
```

A question mark should appear on the screen when input is expected. The question mark has the ASCII value of $3F.

```
LDA #$3F
JSR $FFD2 ; output "?"
```

We want to store the input at $0220 temporarily. To do this we use the X-register as an index register. We perform the input of a character through a call to the routine $FFCF. If the RETURN key was not the last key pressed (ASCII value $0D), we request additional characters.

```
LDX #0
IN1 .LABEL
JSR $FFCF
STA $220,X
INX
CMP #$D
BNE IN1
```

If the RETURN key was pressed, we print a line feed with

```
JSR $FFD2
```

to output the RETURN character. Now we pass the pointer to our temporary storage and the length of the input field to a routine which converts the input to internal form and places it in the FAC.

```
LDA #$02
STA $23
LDA #$20
STA $22
DEX
TXA
JSR $B7B5
RTS
```

Now we need only to write the following lines in our program:

```
JSR IN
LDX #LB-name
LDY #HB-name
JSR $BBD4
```

and the user can enter the value of a variable.

Output of line numbers:

To output line numbers when the trace is turned on we use the interpreter routine which outputs the line number for a "break in ..." message. The line number is divided into most- and least-significant bytes. These two parts are passed in the X-register and the accumulator. The subroutine starts at $BDCD.

Example: Output the line number 259.

```
LDX #3
LDA #1
JSR $BDCD
```

Conditionals:

The language construction which we need to transform is:

```
if A  operator B
then
--
--       block 1
--
else
--
--       block 2
--
endif
```

If the condition "A operator B" is fulfilled, then block 1 will be executed. If the condition is not fulfilled, a jump will be made to the else and block 2 will be executed. We will take a look at how this is done for the possible

conditions. The interpreter has a routine which can compare two floating-point numbers with each other. It starts at $BC5B. The first floating-point number must be in the FAC and the address of the second must be passed in the Y-register and accumulator in the usual manner. After execution of the subroutine, we find information about the result of the comparison in the accumulator:

```
accumulator =   0 means that the values are equal
accumulator =   1 means that variable 1 is greater
                  than variable 2
accumulator = 255 means that variable 1 is less than
                  variable 2
```

The following code results in the individual operators:

Operator: =

```
; 1) transfer variable 1 to the FAC

     LDA #LB-variable1
     LDY #HB-variable1
     JSR $BBA2

; 2) pass the address of variable 2

     LDA #LB-variable2
     LDY #HB-variable2

; 3) Jump to the comparison routine

     JSR $BC5B
```

```
; 4) Evaluate the result for '='

     CMP #0     ; compare the acc with zero
     BNE ELSE
;
; The following instructions of the program
;

     JMP ENDIF ; block 1 is executed
     ELSE .LABEL ; start of block 2

;
; Instructions of the second block
;
     ENDIF .LABEL ; end of conditional
```

We test for equality with CMP #0. If the values are equal, no jump is made. We make a jump on inequality. The principal construction always stays the same, differences occur only in point 4. Only these are given in the following descriptions.

Operator: /=

```
; 4) Evaluate the result for "/="

     CMP #0
     BEQ ELSE
```

Equality of the variable values is again tested with CMP #0. Execution is continued in case of inequality and a jump made to "ELSE" in case of equality.

```
Operator: >

; 4) Evaluate the result for ">"

     CMP #1
     BNE ELSE
```

Execution will be continued if variable 1 > variable 2, else a jump is made.

```
Operator: <=

; 4) Evaluate the result for "<="

     CMP #1
     BEQ ELSE
```

For <= we do the opposite of >.

```
Operator: <

; 4) Evaluate the result for "<"

     CMP #255
     BNE ELSE

Operator: >=

; 4) Evaluate the result for ">="

     CMP #255
     BEQ ELSE
```

We now know how individual conditionals are realized. If conditionals are to be nested within each other, selection of the jump labels becomes a bit more involved. See the chapter on code generation for more information.

Loops:

The instruction to be transformed is formulated:

```
for variable1 from variable2 to variable3 repeat
--
-- Block
--
lend.
```

What do we have to do?

First we must assign the start value, the value of variable2, to variable1. Then a jump label must be set which represents the start of the loop. Now we check if the value of variable1 is strictly greater than the value of variable3. If it is, we jump to the end of the loop. If it is not, we add one to the value of variable1 and execute the instructions in the block. At the end of the block we jump to the label which we designated at the start of the loop.

Example:

```
for otto from tom to harry repeat
--
--
--
lend.
```

Tom has the value 1 and harry the value 3.

The following happens:

1) Otto contains the value 1.
2) Is otto greater than harry?
3) No! Therefore otto = otto + 1.
4) Execute the instructions.
5) Jump to the test with otto = 2.
6) Is otto greater than harry?
7) No! Therefore otto = otto + 1.
8) Execute the instructions.
9) Jump to the test with otto = 3.
10) Is otto greater than harry?
11) No! Therefore otto = otto + 1.
12) Execute the instructions.
13) Jump to the test with otto = 4
14) Is otto greater than harry?
15) Yes! Jump to the end of the loop.

The instructions in the block were executed a total of three times.

We will create a corresponding assembly-language program for this loop and see how we can use the operating system:

1) Otto contains the value of tom:

```
; Transfer tom to the FAC
LDY #HB-tom
LDA #LB-tom
JSR $BBA2
; Transfer the FAC to otto
LDY #HB-otto
```

```
LDA #LB-otto
JSR $BBD4
```

2) Set the label for the start of the loop:

```
LOOP .LABEL
```

3) See if otto > harry

```
; Transfer otto to the FAC
LDY #HB-otto
LDA #LB-otto
JSR $BBA2
; Transfer address of harry for the comparison
LDY #HB-harry
LDA #LB-harry
JSR $BC5B
; Compare if greater
CMP #1
BEQ LOOPEND
; No, then continue
; Add 1 to the FAC
LDA #$E8
LDY #$BF
JSR $B867
; Save the FAC to otto
LDY #HB-otto
LDA #LB-otto
JSR $BBD4
---instructions in block---
;
; end of loop
JMP LOOP ; back to the beginning
LOOPEND .LABEL ; end of the loop
```

A few comments:

You are already acquainted with

"CMP #1" and "BEQ LABEL"

from the last section. The comparison routine does not alter the FAC so we can add one to the FAC directly. We must put the value of otto in the FAC before the comparison since we don't know if the FAC is changed inside the loop itself.

The interpreter has a one stored at address $BFE8 for its own purposes. We use this for the addition. Afterwards we save the value in the FAC in otto for the next pass through the loop.

If we have several loops nested inside each other, we must choose different names for each loop according to some system. More information can be found in the section on code generation.

Loops with an exit are in principle handled like conditionals. Again, see the chapter on code generation for more information.

What is still missing?

You can find out how numbers are converted to the internal number format in the section on code generation.

To set the screen colors, the corresponding numbers need only be stored in special memory locations and the computer takes care of the rest. The table of colors and numbers can be found in the user's guide.

One comment in closing:

There is much yet to say about the interpreter and operating system, information that could easily fill a book. The information in this book is sufficient to allow you to understand the material presented here and should serve to whet your appetite for more.

8. HOW DO WE MAKE OUR PROGRAMS SMALLER?

We have developed programs in this book which require a large portion of the memory in our computer for the program text. This does not leave much space to extend the program.

If you have a BASIC compiler, you can make the programs smaller by compiling them, assuming that your compiler does that.

If you do not have a compiler, you can reduce your programs by hand, but that requires a lot of effort. There is another possibility, however, one which I would like to propose now.

Who among us has not wished for a program which, at the press of a button, made a BASIC program as small and fast as possible? When one reaches the limits of the available memory, one begins to consider how space might be saved.

First, the comment lines must be removed, since they are not necessary for execution of the program. But this backfires when you later go back and try to change the program. Removing excess spaces in a program is also another method, but it becomes very tiresome in longer programs.

Multiple statements can also be combined onto a single line, but the editor only processes lines up to a maximum length of 80 characters, not the 255 characters that BASIC can process. If you combine all of these in a program, you can save some memory and shorten execution time a bit, but the program then becomes very unreadable and is very difficult to change. In addition, the work is very time-intensive and requires concentration, especially in combining lines, since

one of them may be the destination of a jump or an IF...THEN construction may be altered.

But we didn't buy a computer in order to have is sit there for hours and remove spaces from programs. With the program "BASIC COMPRESSOR" your computer can make your programs up to 55% shorter and up to 40% faster. This is naturally dependent on your programming style. My own programs become on the average 45% shorter and 35% faster. And to see that I don't make such extravagant use of the memory, see the following listing of the BASIC compressor.

Operation of the program:

Turn the computer off and then back on in order to make sure it is in the power-up state. Then load the program which you wish to compress and enter the following two lines in the command mode:

```
POKE 43,(PEEK(45)+256*PEEK(46)-2)AND255
POKE 44,(PEEK(45)+256*PEEK(46)-2) / 256
```

This moves the start of the BASIC program area to the end of the program to be compressed. Now load the BASIC compressor and start it with RUN. The BASIC compressor is now the program which you will execute.

It first asks you how long the lines may be in the program to be compressed. Note that the internal line length is meant here. Lines are typically longer in the representation on the screen than they are internally (for example, GOSUB is stored as a single character internally). The LIST command can only process lines which have a maximum of 256 in the LIST format, however.

This means that your computer can execute lines which are longer than it can list!

The computer makes a proposal for the answer to each question, which you can change by simply overwriting. Inputs are ended by pressing the RETURN key.

Then it asks for the number of the first line which the compressed program should have, followed by the increment of the numbers.

Finally, the computer would like to know approximately how many lines the program to be compressed has. If you enter a number which is too large, an "OUT OF MEMORY" error may appear if memory space is tight. If you get a "BAD SUBSCRIPT" error, you entered a number that was too small.

The program makes three passes through your program and outputs the number of the line it is currently working on. After the compressor has completed its task, only your compressed program is left in memory. But don't be surprised if you don't recognize it anymore!

How do we test the BASIC compressor? On itself, naturally!

Load the BASIC compressor, enter the two lines, and load the compressor again. After starting it with RUN, answer all the questions by simply pressing RETURN. After the program is done, save the compressed BASIC compressor. Turn the computer off and then back on. Then load the original BASIC compressor, enter the two lines, and load the compressed BASIC compressor. RUN it and again answer all the questions with RETURN. When the program is done, use the VERIFY command to test that the program in memory is the same as

the saved version of the compressed compressor. If so, then you probably did not make any errors in entering or in operating the program. Before you try out the BASIC compressor, you should in any event make a back-up copy just be safe, since the program destroys itself at its conclusion.

Study the original compressor and compare it with the compressed version. Such a study should convince you of the benefits of such a program and allow you to recognize the individual possibilities. Use the possibility to write extremely well documented programs and then compress them during your well-earned coffee break.

8.1 BASIC COMPRESSOR LISTING

```
53000 REM ============================
53010 REM =   BASIC - COMPRESSOR     =
53030 REM =   BY    VOLKER SASSE     =
53050 REM =   FOR LIST20 & CBM64     =
53070 REM =   BASIC  VERSION 2.0     =
53080 REM ============================
53090 :
54010 PRINT"{CLR}{DWN}{DWN}{DWN}    BASIC - COMPRESSOR"
54020 PRINT"{DWN}{DWN}{DWN}    MAXIMUM LINE LENGTH "
54021 INPUT"      60{LEFT}{LEFT}{LEFT}{LEFT}";ZM
54022 PRINT"{DWN}    FIRST LINE NUMBER "
54023 INPUT"      10{LEFT}{LEFT}{LEFT}{LEFT}";SA
54024 PRINT"{DWN}    LINE NUMBER INCREMENT"
54025 INPUT"      10{LEFT}{LEFT}{LEFT}{LEFT}";AB
54027 PRINT"{DWN}    NUMBER OF LINES"
54028 INPUT"      300{LEFT}{LEFT}{LEFT}{LEFT}{LEFT}";A9
54030 DIM A(A9),M%(A9),VS(A9)
54040 L=4608
54045 IF PEEK(64787)=56 AND PEEK(64788)=48 THEN L=2048
54050 H=PEEK(44)*256+PEEK(43):Z=L-1:N1=L
54055 PRINT"{DWN}{LEFT}{LEFT}PASS 1        LINE  : ";
54060 Z=Z+1 : IF Z>H THEN 54175
54070 T=PEEK(Z)
54080 IF T=34 THEN GOSUB 57500
54090 IF T=32 THEN POKE Z,7 : GOTO 54060
54100 IF T=139 THEN GOSUB 57300 : GOTO 54060
54110 IF T=167 THEN GOSUB 56200 : GOTO 54060
54120 IF T=137 THEN GOSUB 56400 : GOTO 54060
54130 IF T=141 THEN GOSUB 56400 : GOTO 54060
54140 IF T=0 THEN Z=Z+3 : GOSUB 57700  : GOTO 54060
54150 IF T=58 THEN GOSUB 57650 : GOTO 54060
54160 IF T=143 THEN GOSUB 57600 : GOTO 54060
54170 GOTO 54060
54175 REM ============= PASS 1 END
54176 M%(1)=1
54177 GOSUB 57100
54178 REM ============= PASS 2 END
54180 FOR T=1 TO VO : ZN=VS(T) : GOSUB 57000 : NEXT
54200 T1=1
54210 FOR T=1 TO ZL
54220 IF M%(T)=1 THEN A(T1)=A(T) : T1=T1+1
54230 NEXT
54235 FOR T=T1 TO ZL : A(T)=0 : NEXT
54240 ZL=T1-1
54300 Z=L-1 : PZ=L-1 : Z4=1
54305 PRINT:PRINT"{DWN}{LEFT}{LEFT}PASS      3   LINE  : ";
54310 Z=Z+1 : IF Z>H THEN 54400
54320 T=PEEK(Z)
54340 IF T=7 THEN 54310
```

```
54345 IF T=34 THEN GOSUB 58000 : IF T=34 THEN 54310
54350 IF T=167 THEN PZ=PZ+1:POKE PZ,167:GOSUB56100:GOTO54310
54360 IF T=137 THEN PZ=PZ+1:POKE PZ,137:GOSUB55800:GOTO54310
54370 IF T=141 THEN PZ=PZ+1:POKE PZ,141:GOSUB55800:GOTO54310
54380 IF T=0 THEN GOSUB 56600 : GOTO 54310
54390 PZ=PZ+1 : POKE PZ,T : GOTO 54310
54400 PZ=PZ-1:POKE PZ-2,0:POKE PZ-1,0
54410 H=INT(PZ/256) : I=PZ-INT(PZ/256)*256
54412 POKE 828,I : POKE 829,H
54413 REM ============= PASS 3 END
54414 POKE L,0
54415 PRINT:PRINT"{DWN}    ";Z-PZ;"BYTES LESS!"
54420 POKE 43,1 : POKE 44,18 : IF L=2048 THEN POKE 44,8
54430 POKE 45,PEEK(828) : POKE46,PEEK(829) : CLR : END
55398 :
55400 REM ============ ADDITIONAL LINE NUMBERS
55401 :
55410 Z=Z+1 : T=PEEK(Z)
55420 IF T=7 THEN 55410
55430 IF T=44 THEN PZ=PZ+1 : POKE PZ,44 : GOTO 55410
55440 IF T>47 AND T<58 THEN GOSUB 55600 : GOTO 55400
55450 Z=Z-1 : RETURN
55598 :
55600 REM ============  RECOGNIZE AND REPLACE LINE NUMBERS
55602 :
55610 Z=Z-1 : ZN$=""
55620 Z=Z+1
55622 IF PEEK(Z)>47ANDPEEK(Z)<58THENGOSUB55700:GOTO55620
55630 ZN=VAL(ZN$) : Z=Z-1
55640 FOR T1=1 TO ZL : IF A(T1)<ZN THEN NEXT
55650 REM ============ T1 NEW LINE NUMBER
55660 ZN$=STR$((T1-1)*AB+SA)
55670 FOR T1=2 TO LEN(ZN$)
55672 PZ=PZ+1 : POKE PZ,ASC(MID$(ZN$,T1,1))
55674 NEXT
55680 RETURN
55681 :
55700 ZN$=ZN$+CHR$(PEEK(Z)) : RETURN
55702 :
55800 REM ============ LINE NUMBERS AFTER GOTO / GOSUB
55802 :
55810 Z=Z+1 : T=PEEK(Z)
55820 IF T=7 THEN 55810
55830 IF T>47ANDT<58THENGOSUB55600:GOSUB55400:RETURN
55840 Z=Z-1 : RETURN
55841 :
56000 REM ============ SAVE LINE NUMBER  THEN/GOTO/GOSUB
56001 :
56010 Z=Z-1 : ZN$=""
56020 Z=Z+1
56022 IF PEEK(Z)>47ANDPEEK(Z)<58THENGOSUB55700:GOTO56020
56030 ZN=VAL(ZN$)
```

```
56040 IF ZN<=A(ZL) THEN GOSUB 57000 : GOTO 56055
56050 VO=VO+1 : VS(VO)=ZN
56055 Z=Z-1
56060 RETURN
56061 :
56100 REM ============ LINE NUMBER AFTER THEN
56102 :
56110 Z=Z+1 : T=PEEK(Z)
56120 IF T=7 THEN 56110
56130 IF T=137 THEN PZ=PZ+1:POKE PZ,137:GOSUB55800:RETURN
56140 IF T=141 THEN PZ=PZ+1:POKE PZ,141:GOSUB55800:RETURN
56150 IF T>47 AND T<58 THEN GOSUB 55600 : RETURN
56160 Z=Z-1 : RETURN
56161 :
56200 REM ============ LINE NUMBER AFTER THEN
56202 :
56210 Z=Z+1 : T=PEEK(Z)
56220 IF T=32 THEN POKE Z,7 : GOTO 56210
56230 IF T=137 THEN GOSUB 56400 : RETURN
56235 IF T=141 THEN GOSUB 56400 : RETURN
56240 IF T>47 AND T<58 THEN GOSUB 56000 : RETURN
56250 Z=Z-1 : RETURN
56252 :
56400 REM ============ LINE NUMBER AFTER GOTO / GOSUB
56402 :
56410 Z=Z+1 : T=PEEK(Z)
56420 IF T=32 THEN POKE Z,7 : GOTO 56410
56430 IF T>47ANDT<58THENGOSUB56000:GOSUB56800:RETURN
56440 Z=Z-1 : RETURN
56599 :
56600 REM ============ CAN LINE BE RESOLVED?
56610 REM ============ CHANGE THE LINE NUMBER
56620 ZN=PEEK(Z+3)+PEEK(Z+4)*256
56622 PRINT SPC(8-LEN(STR$(ZN)))STR$(ZN);
56625 IF Z=H-1 THEN 56650
56630 IF A(Z4)=ZN THEN 56650
56640 PZ=PZ+1 : POKE PZ,58 : Z=Z+4 : RETURN
56650 PZ=PZ+1 : POKE PZ,0
56655 POKE N1,PZ+1-INT((PZ+1)/256)*256
56657 POKE N1+1,INT((PZ+1)/256)
56660 N1=PZ+1 : Z5=(Z4-1)*AB+SA
56665 POKE PZ+3,Z5-INT(Z5/256)*256 : POKE PZ+4,INT(Z5/256)
56670 PZ=PZ+4 : Z4=Z4+1 : Z=Z+4 : RETURN
56671 :
56800 REM ============ ADDITIONAL LINE NUMBERS
56802 :
56810 Z=Z+1 : T=PEEK(Z)
56820 IF T=32 THEN POKE Z,7 : GOTO 56810
56830 IF T=44 THEN 56810
56840 IF T>47 AND T<58 THEN GOSUB56000:GOSUB57300:GOTO56800
56850 Z=Z-1 : RETURN
56852 :
```

```
57000 REM ============ SEARCH FOR ZN IN A() AND SAVE IN M%()
57001 :
57010 FOR Z4=1 TO ZL : IF A(Z4)<ZN THEN NEXT
57020 M%(Z4)=1
57030 RETURN
57032 :
57100 REM ============ SET LINE LENGTH
57102 :
57105 PRINT:PRINT"{DWN}{LEFT}{LEFT}PASS      2   LINE  : ";
57110 Z=L-1 : T=0
57120 Z=Z+1 : Q=PEEK(Z)
57130 IF Q=0 THEN 57160
57140 IF Q<>7 THEN ZC=ZC+1
57150 GOTO 57120
57160 T=T+1 : IF ZZ+ZC>ZM THEN M%(T-1)=1 : ZZ=ZC : ZC=0
57163 ZN=PEEK(Z+3)+PEEK(Z+4)*256
57165 PRINT SPC(8-LEN(STR$(ZN)))STR$(ZN);
57170 IF M%(T)=1 THEN ZZ=0 : ZC=0 : GOTO 57200
57180 ZZ=ZZ+ZC
57190 ZC=0
57200 IF Z=H-1 THEN RETURN
57201 Z=Z+4 : GOTO 57120
57202 :
57300 REM ============ SAVE NEXT LINE FOR IF/ON
57302 :
57310 M%(ZL+1)=1 : RETURN
57312 :
57500 REM ============ SKIP STRINGS
57520 :
57530 Z=Z+1 : T=PEEK(Z) : IF T<>34 AND T<>0 THEN 57530
57540 RETURN
57560 :
57600 REM ============ DELETE COMMENT LINES
57602 :
57605 IF PEEK(Z-1)=58 THEN POKE Z-1,7 :POKE Z,7 : GOTO 57620
57606 IF PEEK(Z-1)=7 AND PEEK(Z-2)=58 THEN POKE Z-2,7 :POKE
Z,7 : GOTO 57620
57610 ZL=ZL-1 : FOR Z3=Z-5 TO Z : POKE Z3,7 : NEXT
57620 Z=Z+1 : IF PEEK(Z)<>0 THEN POKE Z,7 : GOTO 57620
57630 Z=Z+3 : GOSUB 57700 : RETURN
57632 :
57650 REM ============ DELETE COLON LINES
57652 :
57655 IF PEEK(Z-5)=0 AND PEEK(Z+1)=0 THEN 57660
57657 RETURN
57660 ZL=ZL-1 : FOR Z3=Z-5 TO Z :  POKE Z3,7 : NEXT
57670 Z=Z+4 : GOSUB 57700 : RETURN
57671 :
57700 REM ============ SAVE LINE NUMBERS
57720 :
57730 ZL=ZL+1 : A(ZL)=PEEK(Z)+PEEK(Z+1)*256
57735 PRINT SPC(8-LEN(STR$(A(ZL)))STR$(A(ZL));
```

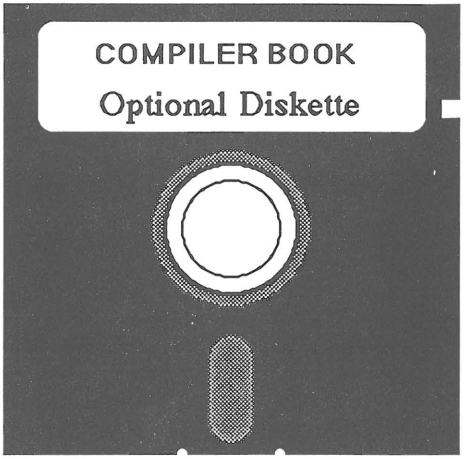
```
57740 Z=Z+1 : RETURN
58000 REM========== PASS STRINGS
58010 PZ=PZ+1 : POKE PZ,34
58020 Z=Z+1 : T=PEEK(Z)
58030 IF T=0 OR T=34 THEN PZ=PZ+1 : POKE PZ,34 : RETURN
58040 PZ=PZ+1 : POKE PZ,T : GOTO 58020
READY.
```

8.2 COMPRESSED BASIC COMPRESSOR

```
1 PRINT"{CLR}{DWN}{DWN}{DWN}     BASIC - COMPRESSOR"
:PRINT"{DWN}{DWN}{DWN}     MAXIMUM LINE LENGTH "
:INPUT"       60{LEFT}{LEFT}{LEFT}{LEFT}";ZM
:PRINT"{DWN}     FIRST LINE NUMBER "
:INPUT"       10{LEFT}{LEFT}{LEFT}{LEFT}";SA
:PRINT"{DWN}     LINE NUMBER INCREMENT"
:INPUT"       10{LEFT}{LEFT}{LEFT}{LEFT}";AB
:PRINT"{DWN}     NUMBER OF LINES"
:INPUT"       300{LEFT}{LEFT}{LEFT}{LEFT}{LEFT}";A9
:DIMA(A9),M%(A9),VS(A9):L=4608
2 IFPEEK(64787)=56ANDPEEK(64788)=48THENL=2048
3 H=PEEK(44)*256+PEEK(43):Z=L-1:N1=L
:PRINT"{DWN}{LEFT}{LEFT}PASS 1         LINE   : ";
4 Z=Z+1:IFZ>HTHEN15
5 T=PEEK(Z):IFT=34THENGOSUB77
6 IFT=32THENPOKEZ,7:GOTO4
7 IFT=139THENGOSUB76:GOTO4
8 IFT=167THENGOSUB49:GOTO4
9 IFT=137THENGOSUB54:GOTO4
10 IFT=141THENGOSUB54:GOTO4
11 IFT=0THENZ=Z+3:GOSUB87:GOTO4
12 IFT=58THENGOSUB84:GOTO4
13 IFT=143THENGOSUB79:GOTO4
14 GOTO4
15 M%(1)=1:GOSUB67:FORT=1TOVO:ZN=VS(T):GOSUB65:NEXT:T1=1
:FORT=1TOZL:IFM%(T)=1THENA(T1)=A(T):T1=T1+1
16 NEXT:FORT=T1TOZL:A(T)=0:NEXT:ZL=T1-1:Z=L-1:PZ=L-1:Z4=1
:PRINT:PRINT"{DWN}{LEFT}{LEFT}PASS       3    LINE   : ";
17 Z=Z+1:IFZ>HTHEN25
18 T=PEEK(Z):IFT=7THEN17
19 IFT=34THENGOSUB88:IFT=34THEN17
20 IFT=167THENPZ=PZ+1:POKEPZ,167:GOSUB44:GOTO17
21 IFT=137THENPZ=PZ+1:POKEPZ,137:GOSUB36:GOTO17
22 IFT=141THENPZ=PZ+1:POKEPZ,141:GOSUB36:GOTO17
23 IFT=0THENGOSUB57:GOTO17
24 PZ=PZ+1:POKEPZ,T:GOTO17
25 PZ=PZ-1:POKEPZ-2,0:POKEPZ-1,0:H=INT(PZ/256)
:I=PZ-INT(PZ/256)*256:POKE828,I:POKE829,H:POKEL,0:PRINT
:PRINT"{DWN}     ";Z-PZ;"BYTES LESS!":POKE43,1:POKE44,18
:IFL=2048THENPOKE44,8
26 POKE45,PEEK(828):POKE46,PEEK(829):CLR:END
27 Z=Z+1:T=PEEK(Z):IFT=7THEN27
28 IFT=44THENPZ=PZ+1:POKEPZ,44:GOTO27
29 IFT>47ANDT<58THENGOSUB31:GOTO27
30 Z=Z-1:RETURN
31 Z=Z-1:ZN$=""
32 Z=Z+1:IFPEEK(Z)>47ANDPEEK(Z)<58THENGOSUB35:GOTO32
33 ZN=VAL(ZN$):Z=Z-1:FORT1=1TOZL:IFA(T1)<ZNTHENNEXT
34 ZN$=STR$((T1-1)*AB+SA):FORT1=2TOLEN(ZN$):PZ=PZ+1
```

```
:POKEPZ,ASC(MID$(ZN$,T1,1)):NEXT:RETURN
35 ZN$=ZN$+CHR$(PEEK(Z)):RETURN
36 Z=Z+1:T=PEEK(Z):IFT=7THEN36
37 IFT>47ANDT<58THENGOSUB31:GOSUB27:RETURN
38 Z=Z-1:RETURN
39 Z=Z-1:ZN$=""
40 Z=Z+1:IFPEEK(Z)>47ANDPEEK(Z)<58THENGOSUB35:GOTO40
41 ZN=VAL(ZN$):IFZN<=A(ZL)THENGOSUB65:GOTO43
42 VO=VO+1:VS(VO)=ZN
43 Z=Z-1:RETURN
44 Z=Z+1:T=PEEK(Z):IFT=7THEN44
45 IFT=137THENPZ=PZ+1:POKEPZ,137:GOSUB36:RETURN
46 IFT=141THENPZ=PZ+1:POKEPZ,141:GOSUB36:RETURN
47 IFT>47ANDT<58THENGOSUB31:RETURN
48 Z=Z-1:RETURN
49 Z=Z+1:T=PEEK(Z):IFT=32THENPOKEZ,7:GOTO49
50 IFT=137THENGOSUB54:RETURN
51 IFT=141THENGOSUB54:RETURN
52 IFT>47ANDT<58THENGOSUB39:RETURN
53 Z=Z-1:RETURN
54 Z=Z+1:T=PEEK(Z):IFT=32THENPOKEZ,7:GOTO54
55 IFT>47ANDT<58THENGOSUB39:GOSUB61:RETURN
56 Z=Z-1:RETURN
57 ZN=PEEK(Z+3)+PEEK(Z+4)*256
:PRINTSPC(8-LEN(STR$(ZN)))STR$(ZN);:IFZ=H-1THEN60
58 IFA(Z4)=ZNTHEN60
59 PZ=PZ+1:POKEPZ,58:Z=Z+4:RETURN
60 PZ=PZ+1:POKEPZ,0:POKEN1,PZ+1-INT((PZ+1)/256)*256
:POKEN1+1,INT((PZ+1)/256):N1=PZ+1:Z5=(Z4-1)*AB+SA
:POKEPZ+3,Z5-INT(Z5/256)*256:POKEPZ+4,INT(Z5/256)
:PZ=PZ+4:Z4=Z4+1:Z=Z+4:RETURN
61 Z=Z+1:T=PEEK(Z):IFT=32THENPOKEZ,7:GOTO61
62 IFT=44THEN61
63 IFT>47ANDT<58THENGOSUB39:GOSUB76:GOTO61
64 Z=Z-1:RETURN
65 FORZ4=1TOZL:IFA(Z4)<ZNTHENNEXT
66 M%(Z4)=1:RETURN
67 PRINT:PRINT"{DWN}{LEFT}{LEFT}PASS      2   LINE  : ";
:Z=L-1:T=0
68 Z=Z+1:Q=PEEK(Z):IFQ=0THEN71
69 IFQ<>7THENZC=ZC+1
70 GOTO68
71 T=T+1:IFZZ+ZC>ZMTHENM%(T-1)=1:ZZ=ZC:ZC=0
72 ZN=PEEK(Z+3)+PEEK(Z+4)*256
:PRINTSPC(8-LEN(STR$(ZN)))STR$(ZN);
:IFM%(T)=1THENZZ=0:ZC=0:GOTO74
73 ZZ=ZZ+ZC:ZC=0
74 IFZ=H-1THENRETURN
75 Z=Z+4:GOTO68
76 M%(ZL+1)=1:RETURN
77 Z=Z+1:T=PEEK(Z):IFT<>34ANDT<>0THEN77
78 RETURN
```

```
79 IFPEEK(Z-1)=58THENPOKEZ-1,7:POKEZ,7:GOTO82
80 IFPEEK(Z-1)=7ANDPEEK(Z-2)=58THENPOKEZ-2,7:POKEZ,7:GOTO82
81 ZL=ZL-1:FORZ3=Z-5TOZ:POKEZ3,7:NEXT
82 Z=Z+1:IFPEEK(Z)<>0THENPOKEZ,7:GOTO82
83 Z=Z+3:GOSUB87:RETURN
84 IFPEEK(Z-5)=0ANDPEEK(Z+1)=0THEN86
85 RETURN
86 ZL=ZL-1:FORZ3=Z-5TOZ:POKEZ3,7:NEXT:Z=Z+4:GOSUB87:RETURN
87 ZL=ZL+1:A(ZL)=PEEK(Z)+PEEK(Z+1)*256
:PRINTSPC(8-LEN(STR$(A(ZL))))STR$(A(ZL));:Z=Z+1:RETURN
88 PZ=PZ+1:POKEPZ,34
89 Z=Z+1:T=PEEK(Z):IFT=0ORT=34THENPZ=PZ+1:POKEPZ,34:RETURN
90 PZ=PZ+1:POKEPZ,T:GOTO89
READY.
```



For your convenience, the programs that are listed in this book are available on a 1541 formatted diskette. If you want to use the programs, without typing them in from the listings in the book, you may want to order this diskette.

All programs on the diskette have been fully tested. The diskette is available for $14.95 + $2.00 ($5.00 foreign) for postage and handling charges.

When ordering, Please specify the title of the diskette, your name and shipping address and enclose a check, money order or credit card information. Mail your order to:

ABACUS Software
P.O. Box 7211
Grand Rapids, MI 49510

Call today for the name of your nearest local dealer
Phone (616) 241-5510

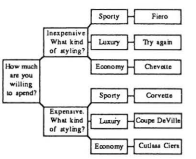
SUPER PRODUCTIVITY

POWER PLAN

Names	Wage per /hr	Mo	Tu
Andew	6.00 $	7.50	8.10
Bosen	7.50 $	7.70	3.50
Carter	4.70 $	6.80	3.50
Damien	5.90 $	1.90	10.60
Gerrit	13.00 $	11.50	10.00
Higgins	9.10 $	6.50	7.80
Mc Donald	7.20 $	9.00	10.40
Nimitz	8.99 $	9.20	4.40
Smith 1	15.90 $	4.40	13.10
Smith 2	15.00 $	10.10	4.40
Wimpy	6.00 $	9.20	11.60
Minimum		1.90	3.50
Average		7.62	7.95
Maximum		11.50	13.10

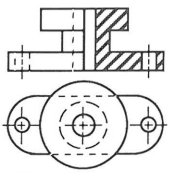
Powerful spreadsheet *plus* builtin graphics - display your important data visually as well as numerically. You'll learn fast with the 90+ HELP screens. Advanced users can use the short-cut commands. For complex spreadsheets, you can use POWER PLAN's impressive features: cell formatting, text formatting, cell protection, windowing, math functions, row and column sort, more. Then quickly display your results in graphics format in a variety of 2D and 3D charts. Includes system diskette and user's handbook. **$49.95**

XPER - expert system



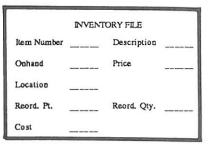
XPER is the first *expert system* - a new breed of intelligent software for the C-64 & C-128. While ordinary data base systems are good at reproducing facts, **XPER** can help you make decisions. Using its simple entry editor, you build the information into a *knowledge base.* XPER's very efficient searching techniques then guide you through even the most complex decision making criteria. Full reporting and data editing. Currently used by doctors, scientists and research professionals. **$59.95**

CADPAK Revised Version



CADPAK is a superb design and drawing tool. You can draw directly on the screen from keyboard or using optional lightpen. POINTs, LINEs, BOXes, CIRCLEs, and ELLIPSEs; fill with solids or patterns; free-hand DRAW; ZOOM-in for intricate design of small section. Mesuring and scaling aids. Exact positioning using our *AccuPoint* cursor positioning. Using the powerful **OBJECT EDITOR** you can define new fonts, furniture, circuitry, etc. Hardcopy to most printers. **$39.95**
McPen lightpen, optional **$49.95**

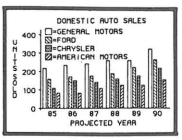
DATAMAT - data management



"Best data base manager under $50"
RUN Magazine

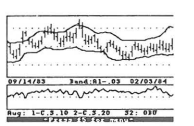
Easy-to-use, yet versatile and powerful features. Clear menus guide you from function to function. Free-form design of data base with up to 50 fields and 2000 records per diskette (space dependent). Simple data base design. Convenient and quick data entry. Full data editing capabilities. Complete reporting: sort on multiple fields and select records for printing in your specific format. **$39.95**

CHARTPAK



Make professional quality charts from your data in minutes. Quickly enter, edit, save and recall your data. Then interactively build pie, bar, line or scatter graph. You can specify scaling, labeling and positioning and watch **CHARTPAK** instantly draw the chart in any of 8 different formats. Change the format immediately and draw another chart. Incudes statistical routines for average, deviation, least squares and forecasting. Hardcopy to most printers. **$39.95**
CHARTPLOT-64 for 1520 plotter **$39.95**

TAS - technical analysis



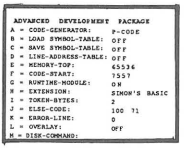
Technical analysis charting package to help the serious investor. Enter your data at keyboard or capture it through **DJN/RS** or Warner Services. Track high, low, close, volume, bid and ask. Place up to 300 periods of information for 10 different stocks on each data diskette. Build a variety of charts on the split screen combining information from 7 types of moving averages, 3 types of oscillators, trading bands, least squares, 5 different volume indicators, relative charts, much more. Hardcopy to most printers. **$84.95**

SUPER LANGUAGES

The most advanced **C** development package available for the C-64 or C-128 with very complete source editor; full K&R compiler (w/o bit fields); linker (binds up to 7 separate modules); and set of disk utilities. Very complete editor handles search/replace, 80 column display with horizontal scrolling and 41K source files. The I/O library supports standard functions like printf and fprintf. Free runtime package included. For C-64/C-128 with 1541/1571 drive. Includes system diskette and user's handbook. **$79.95**

Not just a compiler, but a complete development system. Rivals Turbo Pascal© in both speed and features. Produces fast 6510 machine code. Includes advanced source file editor; full Jensen & Wirth compiler with system programming extensions, new high speed DOS (3 times faster); builtin assembler for specialized requirements. Overlays, 11-digit arithmetic, debugging tools, graphics routines, much more. Free runtime package. Includes system diskette and complete user's handbook. **$59.95**

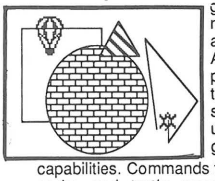
BASIC-64 full compiler



```
ADVANCED  DEVELOPMENT  PACKAGE
A = CODE-GENERATOR:        P-CODE
B = LOAD SYMBOL-TABLE:     OFF
C = SAVE SYMBOL-TABLE:     OFF
D = LINE-ADDRESS-TABLE:    OFF
E = MEMORY-TOP:            65536
F = CODE-START:            7557
G = RUNTIME-MODULE:        ON
H = EXTENSION:             SIMON'S BASIC
I = TOKEN-BYTES:           2
J = ELSE-CODE:             100 71
K = ERROR-LINE:            0
L = OVERLAY:               OFF
M = DISK-COMMAND:
```

The most advanced **BASIC** compiler available for the C-64. Our bestselling software product. Compiles to super-fast 6510 machine code or very compact speed-code. You can even mix the two in one program. Compiles the complete BASIC language. Flexible memory management and overlay options make it perfect for all program development needs. **BASIC 64** increases the speed of your programs from 3 to 20 times. Free runtime package. Includes system diskette and user's handbook. **$39.95**

VIDEO BASIC development



The most advanced graphics development package available for the C-64. Adds dozens of powerful commands to standard BASIC so that you can use the hidden graphics and sound capabilities. Commands for hires, multicolor, sprite and turtle graphics, simple and complex music and sound, hardcopy to most printers, memory management, more. Used by professional programmers for commerical software development. Free runtime package. Includes system diskette and user's handbook. **$39.95**

FORTH Language

```
SCR # 20
 0 P ( RANDOM NUMBER TESTER 7RND )
 1 P FORTH DEFINITIONS DECIMAL
 2 P : 7RND
 3 P      ( INITIALIZE FIRST SCREEN)
 4 P      1024 1000 ASCII 8 FILL
 5 P      BEGIN
 6 P           1000 RND  ( RANDOM 0,999)
 7 P           40 /MOD   ( COLUMN, LINE)
 8 P           SWAP      ( EXCHANGE)
 9 P           ZOUP SE   ( CHARACTER)
10 P           1+ -ROT   ( ADD 1)
11 P           SI        ( SAVE)
12 P      ?TERMINAL UNTIL
13 P
```

Our **FORTH** language is based on the Forth 79 standard, but also includes much of the 83 level to give you 3 times vocabulary of fig-Forth. Includes full-screen editor, complete Forth-style assembler, set of programming tools and numerous sample programs to get you deeply involved in the FORTH language. Our enhanced vocabulary supports both hires and lores graphics and the sound synthesizer. Includes system diskette with sample programs and user's handbook. **$39.95**

Other software also available!
Call now for free catalog and the name of your nearest dealer. Phone: **616/241-5510.**

Abacus Software

P.O. Box 7211 Grand Rapids, MI 49510 616/241-5510

For fast service call **616/241-5510.** For postage and handling, include $4.00 per order. Foreign orders include $8.00 per item. Money orders and checks in U.S. dollars only. Mastercard, Visa and Amex accepted.

Dealer Inquiries Welcome
More than 1200 dealers nationwide

COMPILER DESIGN AND IMPLEMENTATION FOR C-64 & C-128

This book is a practical introduction to writing compilers. Author Volker Sasse draws upon is experience as an implementor of several compilers and discusses:

- Compiler fundamentals
- Language design
- Lexical analysis
- Syntactical analysis
- Semantic analysis
- Generating 6502 code
- Interpacing to the operating system

ISBN 0-916439-35-6

P.O. BOX 7211 GRAND RAPIDS, MICH. 49510 PHONE 616-241-5510

C-64
C-128

COMPILER DESIGN & IMPLEMENTATION