Introduction to Minicomputers

Programming Languages

digital

1st Printing, June 1976
2nd Printing (Rev), October 1977
3rd Printing, August 1979

Printed in U.S.A.

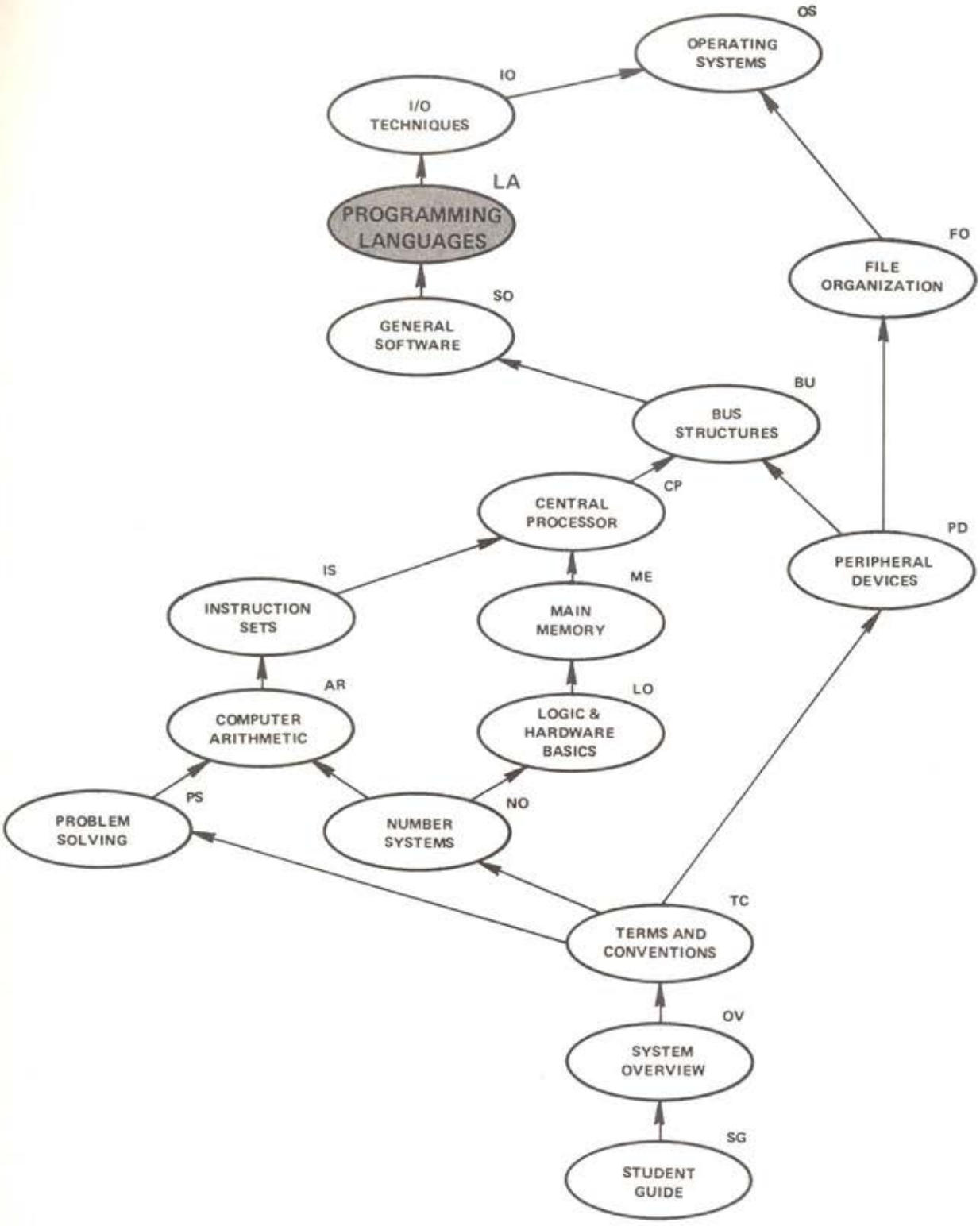
INTRODUCTION TO MINICOMPUTERS

Programming Languages

Student Workbook

Audio-Visual Course by Digital Equipment Corporation

COURSE MAP


OS
OPERATING SYSTEMS
IO
I/O TECHNIQUES
LA
PROGRAMMING LANGUAGES
FO
FILE ORGANIZATION
SO
GENERAL SOFTWARE
BU
BUS STRUCTURES
CP
CENTRAL PROCESSOR
PD
PERIPHERAL DEVICES
IS
INSTRUCTION SETS
ME
MAIN MEMORY
AR
COMPUTER ARITHMETIC
LO
LOGIC & HARDWARE BASICS
PS
PROBLEM SOLVING
NO
NUMBER SYSTEMS
TC
TERMS AND CONVENTIONS
OV
SYSTEM OVERVIEW
SG
STUDENT GUIDE

CONTENTS

Programming Languages

Introduction

Now that you have learned the distinctions between low- and high-level programming languages and some of the general properties of software, we will discuss further aspects of software and the programming process. Software is a crucial part of any computer system. In many areas the cost of the software now frequently exceeds the cost of the hardware. As a result, the central focus of these areas is now the improvement of programmer productivity and of program reliability. The increasing importance of quality, correctness, and maintainability of programs makes the selection of programming languages a procedure that requires careful consideration. These major considerations are the focus of this module.

Lesson One discusses subroutines and relocatable assemblers. These software features allow modular programming – the breaking up of large problems into small segments (or modules) that are easier to code and to correct. Subroutines can also be catalogued, filed in libraries, and exchanged.

Lesson Two discusses macroinstructions and macroassemblers, two software features that allow low-level language programmers to approximate the productivity of high-level language programmers. The improvement in productivity makes macroinstructions effective weapons against the spiraling costs of software development.

The third lesson discusses the two major types of high-level language translators: compilers and interpreters. These two types perform complementary tasks, and each type has its own advantages and special characteristics. To a significant extent, the choice of which type of translator to use is closely related to the important question of which language to use. Hence, knowledge of the translation processes is highly useful.

The final lesson of this module compares and contrasts three of the most common high-level languages – FORTRAN, COBOL, and BASIC. Each language is described in terms of its history, basic design philosophy, and general features. The relative advantages and disadvantages of each language are discussed with reference to their effect on software development costs.

Subroutines, Absolute Assemblers, and Relocatable Assemblers

OBJECTIVES

1. Given statements that describe assemblers, be able to label those statements that refer to absolute assemblers and those that refer to relocatable assemblers.

2. Given statements describing source code, object code, and machine code, be able to identify the code described by each of the statements.

3. Given statements describing software features, be able to label those statements that describe a translator, a subroutine, or an argument.

SAMPLE TEST ITEMS

1. Indicate that each of the following statements refers to an absolute (A) or a relocatable (R) assembler by writing the correct letter in the space provided.

Statement	Type of Assembler
Converts the source assembly language into an intermediate form called object code.	_____
To be executed, the program must be translated into machine code by a linker.	_____
Faster than other assemblers and requires fewer resources.	_____
.	.
.	.
.	.

SAMPLE TEST ITEMS

2. Indicate that each of the following statements refers to source (S), object (O), or machine (M) code by writing the correct letter in the space provided.

Statement	Code Type
All addresses are absolute.	_____
Form of a program that a computer can execute.	_____
Must be translated before it can be executed.	_____
Code produced by linkers.	_____
.	.
.	.
.	.

3. Indicate that each of the following statements describes a translator (T), a subroutine (S), or an argument (A) by writing the correct letter in the space provided.

Statement	Software Feature
Sequence of instructions designed to be solved by more than one program.	_____
May require the use of a linker to complete code conversions.	_____
.	.
.	.
.	.

Mark your place in this workbook and view Lesson 1 in the A/V program, "Programming Languages."

We know that a conversion process, which we call *assembly,* is needed to convert source code to machine code. We also know that source code is the form of a program actually written by the programmer, while machine code is the form used by the computer for execution. The conversion process is performed by a *translator* program that regards the source program as input data and produces machine code as output. The term *assembler* may also be used for a symbolic language translator.

In the audio-visual program, we have seen that development of a symbolic language program can be a long, tedious process and that the use of subroutines facilitates the development process in two cases:

1. When prewritten programs exist in a subroutine library.

2. When an identical sequence of instructions is required many times within a single program.

Thus, a subroutine is a segment of code which may be shared among programs or used many times within a single program. Prewritten subroutines frequently exist as packages or libraries to perform mathematical functions or input/output operations.

Associated with a subroutine are a name and possibly a list of one or more *arguments.* An argument is an operand for a subroutine and may be either a *source of data* or a *destination for data.* For example, a programmer may have available a subroutine which prints a variable on the line printer. To be useful, the subroutine should be able to print any one of a large number of variables by the user specifying which one is to be printed. In a high-level language, the use of such a subroutine might look like:

.
.
.

CALL PRINTER (A)

.
.
.

where A is the label (or name) of the variable to be printed. At some other point, the subroutine PRINTER might be invoked using a different variable as the argument. For the symbolic language programmer, the process of calling a subroutine is not as simple. A typical sequence of instructions for calling and using a subroutine includes:

1. The creation of a list of arguments

2. The establishment and saving of a return address (usually the next instruction after the call)

3. A jump to the starting address of the subroutine

4. The execution of the subroutine instructions

5. A jump from the subroutine to the return address in the calling program

A detailed description of this procedure is not within the scope of this course. It also is highly dependent on a particular computer's architecture and instruction set.

Care must be taken to ensure that subroutines and calling programs are compatible in two ways:

1. The type (numbers or letters) and number of arguments must agree.

2. There must be no conflict in names and labels if the subroutine and mainline program are assembled at the same time.

The agreement of arguments is essential for the correct operation of the subroutine; incompatibility frequently leads to unpredictable results. The requirement of non-conflicting labels is a result of the assembly process – conflicting labels will cause a symbol table error during the initial pass through the program.

The problem of conflicting labels may be solved by assembling each routine separately. Remember that the assembly process removes all symbolic addresses. This action, however, causes another problem. When several programs are assembled separately and are then loaded together for execution, there can be no absolute assurance that two or more routines may not attempt to occupy the same memory locations. This is a situation which, generally, results in a loss of data or program instructions, and it is clearly an error.

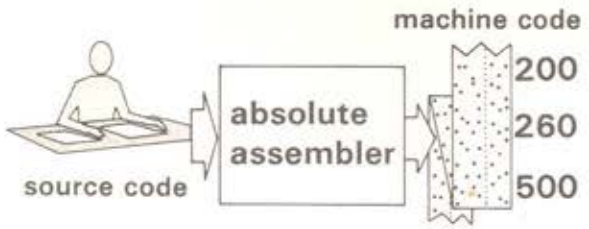


Figure 1 Absolute Assembler

To avoid both the problem of conflicting labels and the problem of conflicting memory assignments, a special type of assembler, called a *relocatable assembler,* is used. Until this point, all references to assemblers have been to the type called absolute assemblers (Figure 1). Absolute assemblers have the following characteristics:

- *Assembly language is directly translated into machine code.*
- *All addresses are resolved into "absolute" machine locations.*
- *All addresses are fixed.*

Relocatable assemblers, on the other hand, have these characteristics:

- Assembly language is translated into an intermediate form called object code.
- All addresses are relative to a reference address that is assigned an "absolute" machine location during the linking process.
- All addresses can be reassigned by simply changing the value of the reference address.

The object code is an intermediate form between the source code and machine code. It cannot be directly executed by a computer because all memory references are in a special form that does not include either the specific addresses of machine code or the symbolic labels of source code. An object code may be combined with an object code produced at another time by using a second special program called a linker or linkage editor. The linker performs the following operations:

- Combines the mainline and subroutine object code.

- Sets up the communications (links) between calling programs and called subroutines.

- *Converts the object code into machine code by resolving all relative addresses into specific (absolute) addresses.*

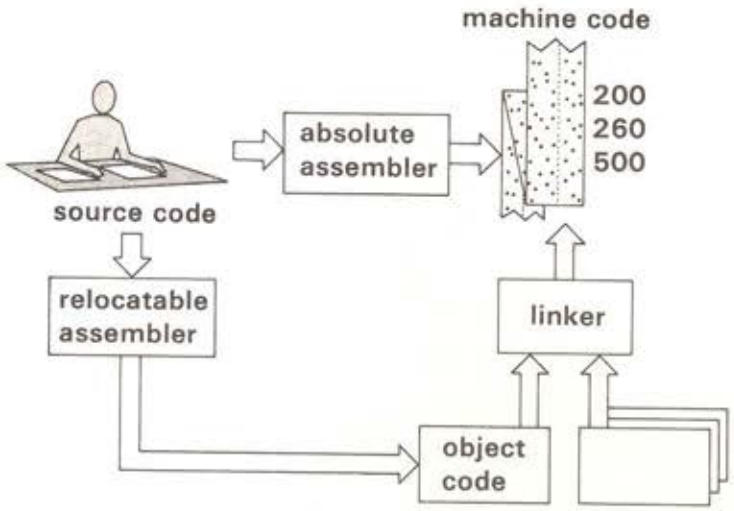


Figure 2 Absolute Assembler, Relocatable Assembler, and Linker

In summary, absolute assemblers translate source code to machine code in *one step*. Thus, while less flexible than relocatable assemblers, absolute assemblers require *less* assembly time. On the other hand, relocatable assemblers translate source code into object code, which must then be further translated into machine code by a linker. This *two-step* process sacrifices some assembly speed for greatly enhanced flexibility. Subroutine libraries of prewritten programs may be included in object code form, thereby reducing the development effort. Frequently, only one of the two types of assembler is present on any single computer, and the choice is determined by the computer resources (primarily memory) available. Where the programmer does have a choice between an absolute and a relocatable assembler, the choice should be determined by the trade-off between assembly time and flexibility.

EXERCISES

1. Explain the differences between "source code," "object code," and "machine code."

2. Define:

   a. Translator

   b. Subroutine

   c. Argument

SOLUTIONS

1. Explain the differences between "source code," "object code," and "machine code."

   a. **Source Code** is the form of the program written by the programmer with symbolic addresses and labels. It must be translated before it can be executed.

   b. **Object Code** is an intermediate form produced by a relocatable assembler with no symbolic labels and relative addresses. It must be linked before it can be executed.

   c. **Machine Code** is the form which is executable by the computer, is produced by absolute assemblers and linkers, and has absolute addresses.

2. Define:

   a. **Translator** – A program which converts a program, written in "source code," into machine code for computer execution. It may require the supplementary use of a linker to achieve this goal.

   b. **Subroutine** – A sequence of instructions, of a semi-independent nature, which is designed to be shared by more than one program or called more than once by a single program.

   c. **Argument** – An operand of a subroutine which is the address of a memory location to be used as a source or destination of data manipulated in the subroutine.

EXERCISES

3. Briefly explain the differences between absolute and relocatable assemblers, and state one advantage each has over the other.

3. Briefly explain the differences between absolute and relocatable assemblers, and state one advantage each has over the other.

Absolute assembler – Directly produces machine code from source code in one step.

Advantages:

- Faster assembly time than relocatable ones
- requires less space, so used on smaller computers.

Relocatable assembler –

- Produces object code from source code
- Requires use of the linker to produce machine code
- Permits flexibility in assembly process, and allows inclusion of subroutine libraries.

Advantages:

- Greater flexibility
- Program development time is less through the use of pre-written subroutines.

Macroinstructions and Macroassemblers

OBJECTIVES

1. Given six descriptive statements, be able to select those statements that describe macroassemblers.

2. Given eight statements of characteristics, be able to label those statements that describe macroassemblers and those that describe subroutines.

SAMPLE TEST ITEMS

1. For each of the following statements, write a T in the space provided if the statement correctly describes macroassemblers. Write an F if it does not correctly describe macroassemblers.

Statement	T or F
Greatly increased operational speed slightly lessens the chance of error-free programs.	_____
Can significantly raise the daily output of assembly language programs.	_____
.	.
.	.
.	.

SAMPLE TEST ITEMS

2. Indicate that each of the following characteristics describes a macroassembler (M) or a subroutine (S) by marking the correct letter in the space provided.

Characteristic	M or S
Expanded once for each special assembly language instruction in the program.	_____
Frequently consumes more memory.	_____
.	.
.	.
.	.

Mark your place in this workbook and view Lesson 2 in the A/V program, "Programming Languages."

Macroinstructions

Macroinstructions are special assembly language instructions that represent frequently used sequences of assembly language instructions. Like normal assembly language instructions, macroinstructions must be translated into machine code before a computer can execute them. Unlike normal assembly language instructions, however, the translation process is not as direct.

A *macrodefinition* is the specification of the sequence of instructions represented by the macroinstruction. It is therefore a pattern or template for the *expansion* of the macroinstruction. Macrodefinitions must be collected at the beginning of the program and have the following parts:

1. A key word, such as DEFINE, to indicate the start of a macrodefinition.

2. The name of the macro and the specification of its operands. The operands are frequently preceded by a special character, such as the ampersand, to indicate that they will be replaced by actual symbolic names during the expansion phase.

3. The body of the macro in the form of normal assembly language instructions using the operands. The body of the macro is frequently enclosed by delimiting characters such as "<" and ">."

Figure 3 shows a complete macrodefinition for the macroinstruction SWAP. Notice that macrodefinitions must appear at the beginning of the program and are defined only once.

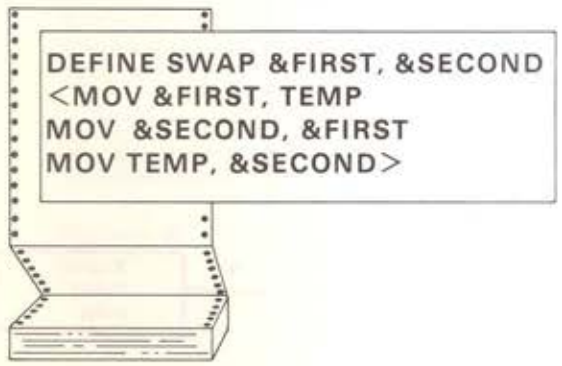


Figure 3 Complete Macrodefinition

Macroassemblers

The process of translating macroinstructions into machine code is performed by a class of assemblers called *macroassemblers*. The ability to process macros is independent of whether an assembler is absolute or relocatable. Consequently, assemblers with either of these properties can in fact translate macros into machine code. A macroassembler performs several steps. This process is summarized in Figure 4.

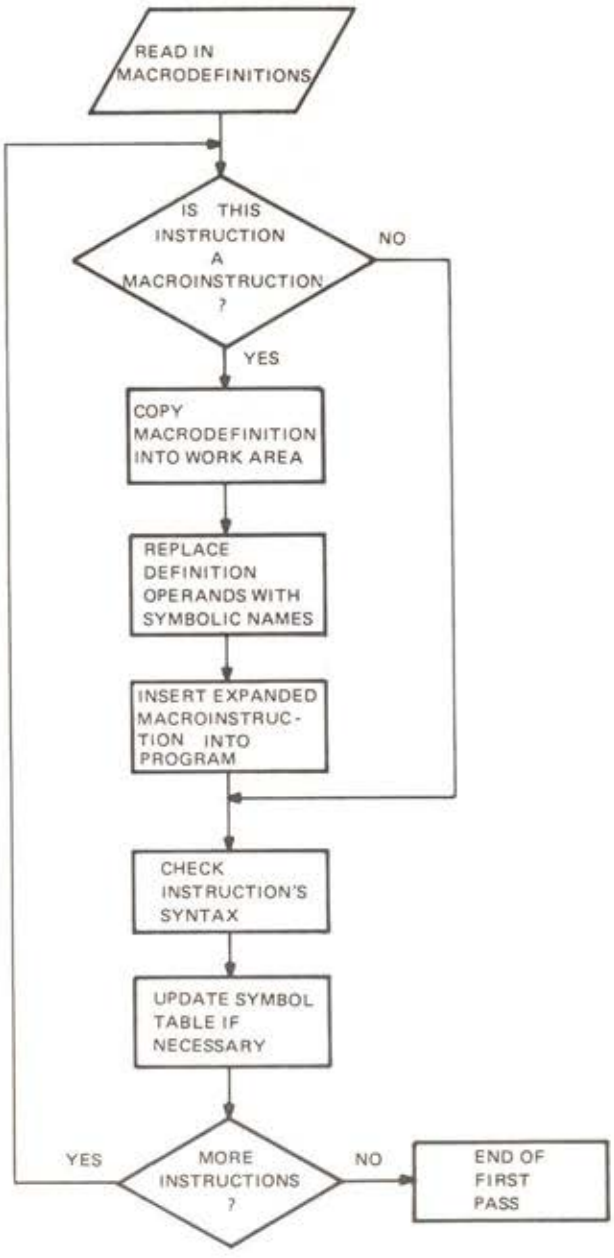


Figure 4 The Macroassembly Process

The first step for the macroassembler is to read in all the macrodefinitions. Remember that these are placed at the beginning of the program. The definitions are stored in memory for use during the expansion process of the first pass.

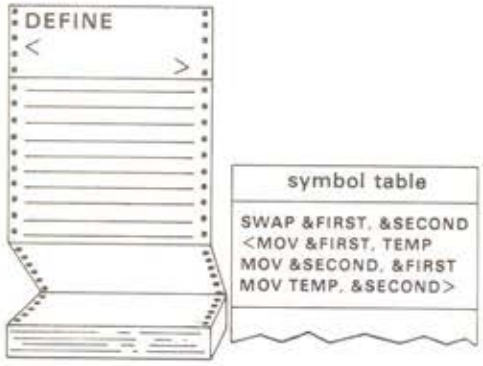


Figure 5 Symbol Table and Definition

The macroassembler then performs the first pass through the source program. Each instruction is checked to determine if it is a macroinstruction. If it is not, it is checked for instruction syntax and a possible label in the normal manner. If the instruction is, in fact, a macroinstruction, the macrodefinition is copied into a work area, and the temporary operands are replaced by the specified symbolic names.

In our example (Figure 6), all occurrences of &FIRST in the definition are replaced by X; all occurrences of &SECOND are replaced by Y.

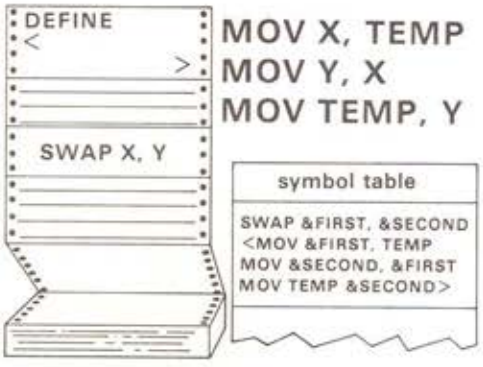


Figure 6 Completed Expansion

As a final step, the completed expansion (Figure 7) is inserted sequentially in the program where the macroinstruction had been.

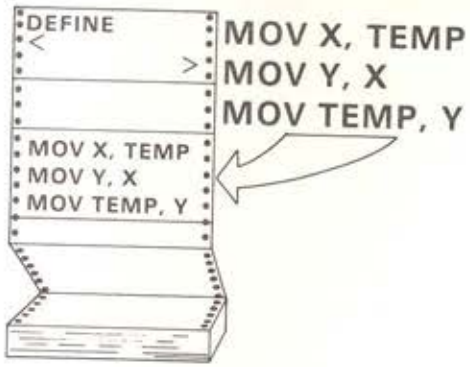


Figure 7 Insertion of Expanded Macro

Assembler processing continues with the next instruction. Notice that the insertion of additional instructions in the program, through the expansion of macroinstructions, has the effect of separating sections of user-generated code (Figure 7). This shifting of the relative addresses is the reason why macroexpansion must occur before or during creation of the symbol table. (If expansion were deferred until the second pass, many of the associated addresses in the symbol table would be incorrect.) Once the macroinstructions have been expanded, the macroassembler functions in the same manner as an assembler without macrocapability.

In short, macros assist the development of a program in several ways. First, the creation of a library of pre-written macrodefinitions for commonly used operations eliminates duplication of effort from program to program. Frequently, such a library includes manufacturer-supplied macrodefinitions to enable the assembly language programmer to easily request various input/output operations and system operations and services. Secondly, a collection of user-written definitions oriented toward a specific application can be greatly helpful. Macro libraries and user-written macros together offer the following advantages:

- Macroinstructions enable the user to enhance the instruction set to meet a particular application. Thus, statement of the problem in assembly language will be more concise and easier from the point of view of the programmer.

- Macroinstructions, whose expansions have been *thoroughly* debugged, generally offer better chances for error-free programs.

- Programmers who use macroinstructions can be far more productive than assembly language programmers who do not use them. Remember that the average programmer produces approximately 20 lines of finished, documented source code per day. Macroinstructions, like high-level language statements, exhibit a many-to-one relationship to the generated machine code instructions. Therefore, the use of macroinstructions can boost the effective daily output of assembly language programmers to ten or twenty times the normal rate.

In conclusion, the use of macroinstructions and definitions enables the source program to be more concise and easier to write, to be more dependable, and to be developed in significantly less time.

Macroinstructions versus Subroutines

At this point, it may be useful to compare and contrast macroinstructions and subroutines.

- A subroutine is a *semi-independent sequence of instructions,* while a macroinstruction is a *single, special instruction* that represents a sequence of other instructions.

- Although both macros and subroutines are defined only once in a program, many expansions of the same macrodefinition may exist within a program while only one copy of a subroutine may exist.

- Subroutines are executed by "calling" or "jumping" to them, while macroinstructions are executed as sequences of "in-line" code.

- Thus macroinstructions generally consume *more assembly time* (because of the expansion step) and *more memory* (because of the existence of multiple copies), but consume *less execution time* (because the complicated calling procedure is not required).

For the most part, the programmer must choose between execution speed and memory economy when deciding whether to use macros or subroutines for a specific application. This decision varies with each application, and neither macros nor subroutines possess a clear advantage over the other. Rather, they are complementary programming aids, which may be used together or separately to assist in the development of a program.

EXERCISES

1. For each of the questions below, answer with one of these words: MACROS, SUBROUTINES, BOTH, or NEITHER.

   a. Which technique produces "in-line" code? _____

   b. Which technique requires a special assembler? _____

   c. Which technique allows a sequence of instructions to be defined once, but used many times? _____

   d. Which technique produces only one copy of a defined sequence of instructions, regardless of the number of times it is used? _____

   e. Which technique generally requires less execution time? _____

   f. Which technique generally requires less assembly time? _____

   g. Which technique generally requires less memory? _____

   h. Which technique has a clear advantage over the other? _____

SOLUTIONS

1. For each of the questions below, answer with one of these words: MACROS, SUBROUTINES, BOTH, or NEITHER.

   a. Which technique produces "in-line" code? MACROS

   b. Which technique requires a special assembler? MACROS

   c. Which technique allows a sequence of instructions to be defined once, but used many times? BOTH

   d. Which technique produces only one copy of a defined sequence of instructions, regardless of the number of times it is used? SUBROUTINES

   e. Which technique generally requires less execution time? MACROS

   f. Which technique generally requires less assembly time? SUBROUTINES

   g. Which technique generally requires less memory? SUBROUTINES

   h. Which technique possesses a clear advantage over the other? NEITHER

EXERCISES

2. Given the source program and the macrodefinition shown below, show the assembly language program as it would be after the first pass had been completed.

Source Program

```
.
.
.
                Define    ABS   &ANS,   &INPUT
ABS X,A                   CLA
                          ADD   &INPUT
                          SPA
                          CMA
                          IAC
                          STR   &ANS
```

Absolute Value Macrodefinition

Expanded Assembly Language Program

2. Given the source program and the macrodefinition shown below, show the assembly language program as it would be after the first pass had been completed.

Source Program	Absolute Value Macrodefinition		
.			
.			
.	Define	ABS	&ANS, &INPUT
ABS X,A		CLA	
		ADD	&INPUT
		SPA	
		CMA	
		IAC	
		STR	&ANS

Expanded Assembly Language Program

```
.
.
.
CLA
ADD A
SPA
CMA
IAC
STR X
```

NOTE

The mnemonic SPA is for the operation "skip next instruction if the accumulator is positive." Thus, the complement operation will be executed only if the accumulator is negative.

EXERCISES

3. Given the source program and the macrodefinition shown below, show the assembly language program as it would be after the first pass had been completed.

Source Program

```
        .
        .
        .          Define   SUB   &ANS,   &ONE,   &TWO

SUB X,A,B                   CLA
ADD C                       ADD   &TWO
STR TEMP                    CMA
SUB Y,E,B                   IAC
ADD TEMP                    ADD   &ONE
STR ANSWER                  STR   &ANSI
```

Subtraction Macrodefinition

Expanded Assembly Language Program

3. Given the source program and the macrodefinition shown below, show the assembly language program as it would be after the first pass had been completed.

Source Program

```
        .
        .
        .
        Define  SUB   &ANS,  &ONE,  &TWO

SUB X,A,B       CLA
ADD C           ADD   &TWO
STR TEMP        CMA
SUB Y,E,B       IAC
ADD TEMP        ADD   &ONE
STR ANSWER      STR   &ANS
```

Subtraction Macrodefinition

Expanded Assembly Language Program

```
+ CLA
+ ADD B
+ CMA
+ IAC
+ ADD A
+ STR X
  ADD C
  STR TEMP
+ CLA
+ ADD B
+ CMA
+ IAC
+ ADD E
+ STR Y
  ADD TEMP
  STR ANS
```

Instructions indicated by a plus have been generated from a macroinstruction.

EXERCISES

4. Explain at least one advantage of using macroinstructions and a macroassembler over an assembler without macro capability.

4. Explain at least one advantage of using macroinstructions and an macroassembler over an assembler without macro capability.

   a. Macro libraries eliminate duplicated effort from program to program.

   b. Macroinstructions permit the instruction set to be enhanced to fit a particular application. This makes writing the program easier and allows the source code to be more concise.

   c. Thoroughly debugged macros assure a better chance of error-free programs.

   d. Macroinstructions can significantly raise the daily output of assembly language programmers as a result of their many-to-one relationship to normal assembler language instructions.

High-Level Language Translators

OBJECTIVE

Given statements of characteristics and advantages of high-level language translators, be able to label those statements that refer to compilers and those that refer to interpreters.

SAMPLE TEST ITEM

Statements of characteristics and advantages of two types of translators are given below. Indicate that each statement refers to compilers (C) or interpreters (I) by writing the correct letter in the space provided.

Characteristic	Type of Translator
Execution of single statement operations. Process continues with next logical statement, not necessarily next sequential one.	_____
Assembly language instructions generated one source statement at a time.	_____
Linking into machine code.	_____
.	.
.	.
.	.

Mark your place in this workbook and view Lesson 3 in the A/V program, "Programming Languages."

A translator is a program that converts a source program into some other form such as object code or machine code. You have learned that assemblers are translators for low-level languages. There are two major classes of translators for high-level languages: *compilers* and *interpreters.* Although their final result – execution of a high-level language source program – is the same, their methods of performing this task are quite different. Let's examine these differences.

Compilers

A compiler translates a high-level language source program into *object code.* This object code can then be linked into machine code. The complete translation process, using a compiler, includes five phases.

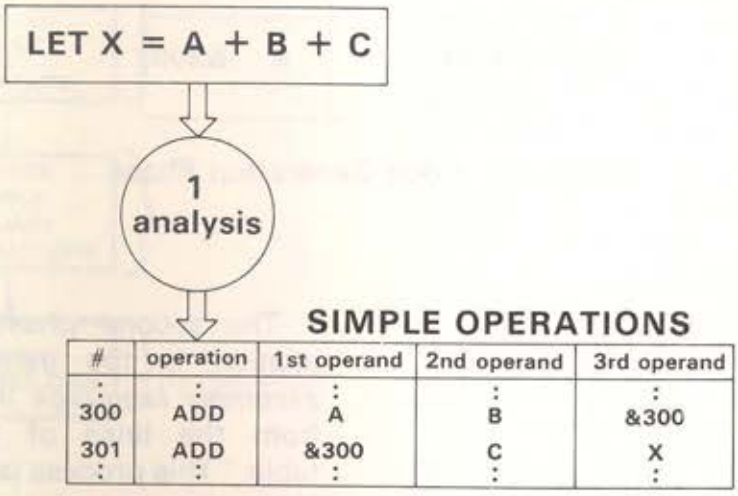
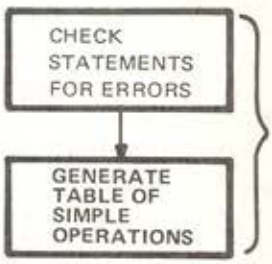


Figure 8 Analysis Phase



The first phase involves *analysis* of the individual statments (Figure 8). Each statement is checked for errors in obeying the rules of the language (syntax checking). A table that contains the simple machine operations corresponding to each statement is then generated. This table is frequently called a "1-op table" because each line of the table contains *one* and only one operation.

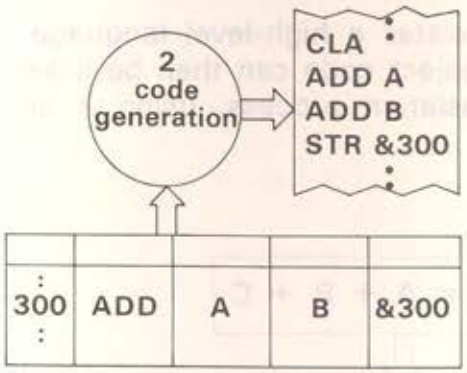
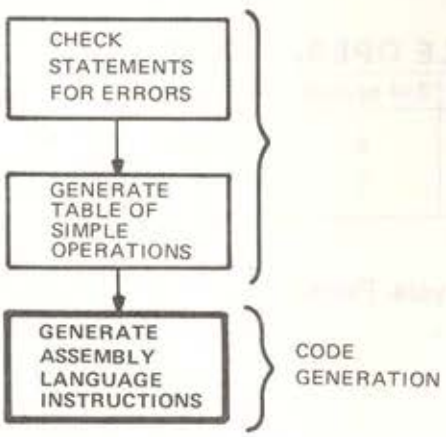


Figure 9 Code Generation Phase



The second phase of compilation is the *generation of assembly language* instructions from the lines of the "1-op table." This process is a relatively simple one as can be seen from Figure 9. Notice the close similarities between the tabular "1-op" lines and the generated assembly language instructions.

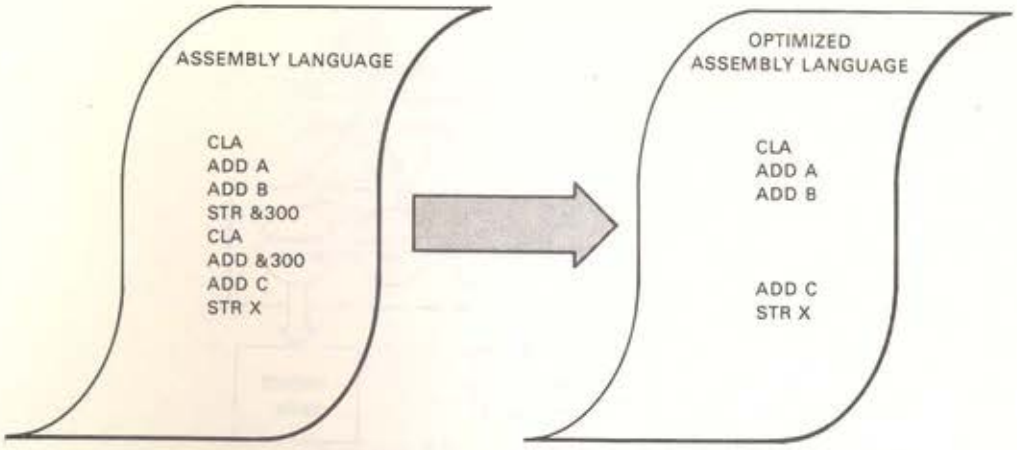
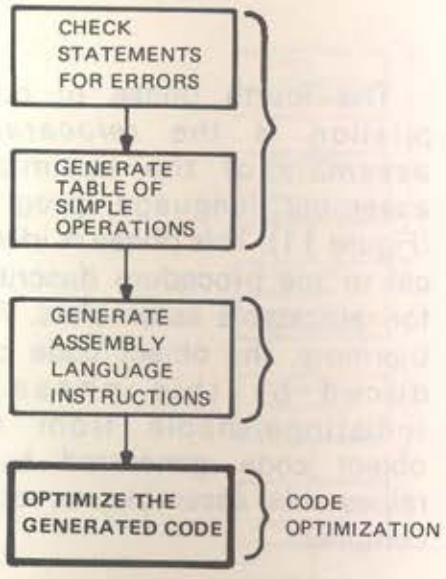


Figure 10 Code Optimization Phase



The third phase involves the *optimization* of the generated assembly language instructions. Because the generated code was produced from the "1-op table," unnecessary store-fetch operations may be created. These "wasted" instructions are removed during the optimization phase as shown in Figure 10. Depending on the sophistication of the compiler, additional, more subtle optimizations may be performed. The optimization process is an important one, as it enables high-level language programs to have execution speeds comparable to those of assembly language programs.

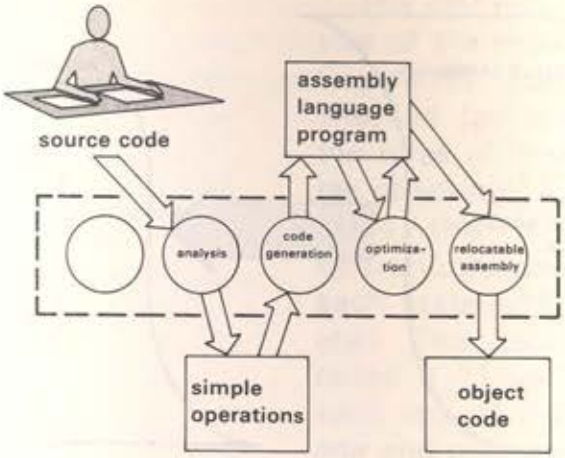
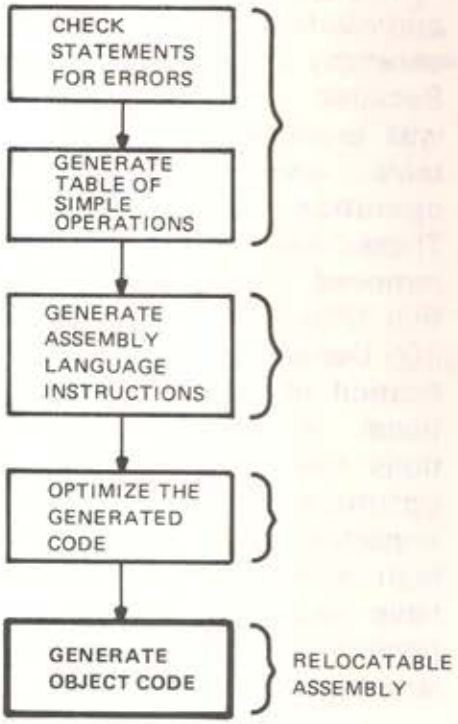


Figure 11 Relocatable Assembly Phase



The fourth phase of compilation is the *relocatable assembly* of the optimized assembly language program (Figure 11). This phase is identical to the procedure described for relocatable assemblers. Furthermore, the object code produced by this phase is indistinguishable from the object code generated by a relocatable assembler or other compilers.

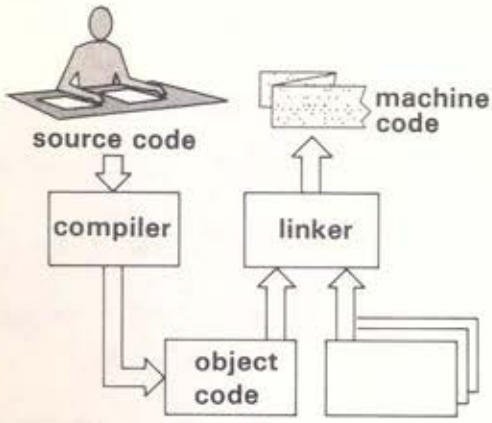
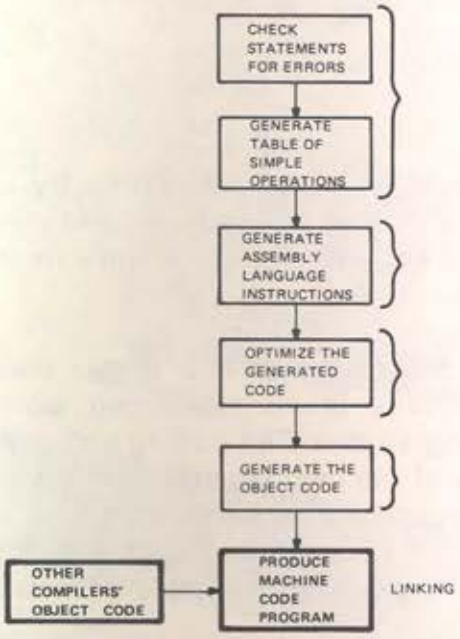


Figure 12 Linking Phase



The final phase in the translation of a high-level language to machine code is the *linking* of the object code. Because this phase (Figure 12) is performed by the linker, it is not strictly part of the compilation process. Remember that the object code produced by a compiler is compatible with the object code produced by other compilers and a relocatable assembler. This feature allows the programmer to write each program and subroutine in the language which best suits the particular application. The programmer then links the various object programs into a single machine code program that can be executed. Hence, the programmer is given a choice in how to develop a system of programs. Additionally, the development phase of programming can be considerably shortened by astute choices of the languages.

Several important points remain to be made about compilers:

- The output of a compiler is the object code.
- Compilation and optimization can take a great deal of time.
- Execution speed is relatively high because of code optimization.
- Development time may be reduced by mixing suitable languages on a subroutine-by-subroutine basis.
- Development time is increased by the fact that logical errors in the program are not detected *until* the lengthy compilation linking process is complete. Hence it takes longer to find and correct a logical mistake.
- The object code output of a compiler can be saved for linking and execution at a later date. Therefore, libraries of already compiled programs are easily created.

Interpreters

An interpreter performs high-level language translation by a very different method – *the program is executed as it is being analyzed and translated.* An interpreter, therefore, produces the program's *results* as its output, rather than object code.

The first step in interpreting a program is *on-line syntax checking and editing (Figure 13).* This activity is an interactive exchange between the interpreter and the programmer. The editing and checking concerns the correctness (in terms of the language rules, or syntax) of each statement. Because most interpreters are configured with terminals for users, this man-machine dialogue occurs as each statement is originally typed. Thus, language errors are immediately detected and can be immediately corrected.

Figure 13 On-line Editing Phase

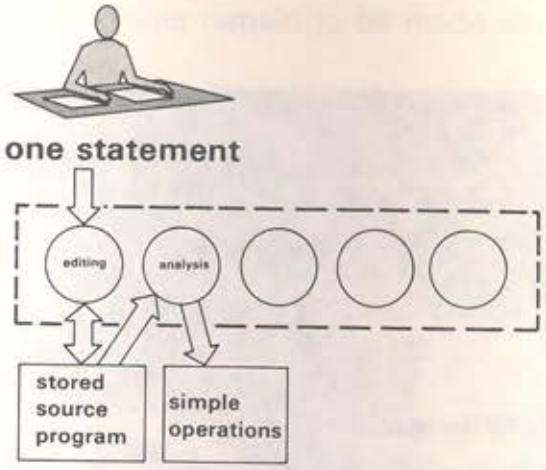


Figure 14 Analysis Phase

After a program has been correctly entered, it may be run *immediately.* The interpreter begins execution of a statement by analyzing it in a process similar to that of a compiler (Figure 14). Note, however, that only one statement at a time is translated into a "1-op table."

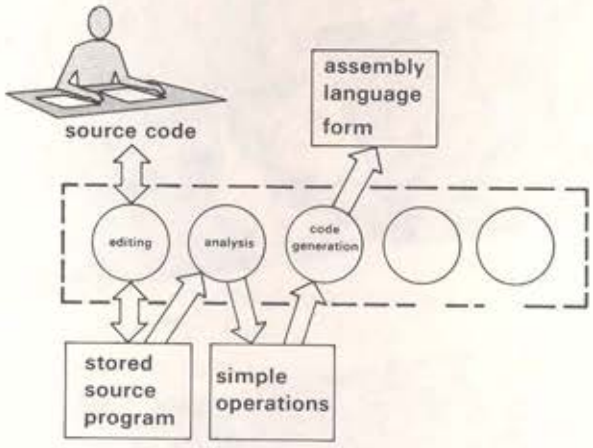


Figure 15 Code Generation Phase

The "1-op" entries for a single statement are then immediately translated into the corresponding assembly language instructions (Figure 15). Again, only the operations of one statement are translated.

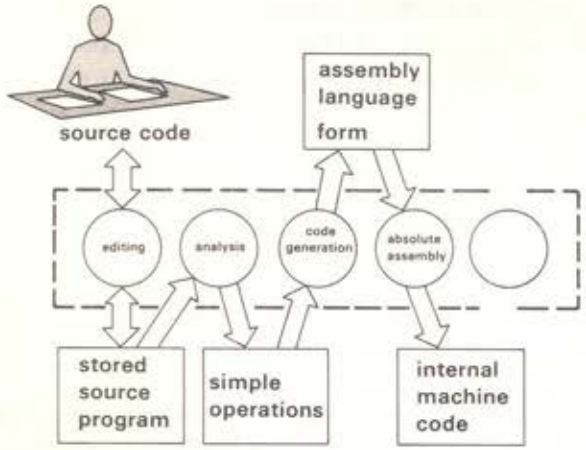


Figure 16 Absolute Assembly

The assembly language instructions are then immediately assembled into machine code. Remember that only one statement is executed at a time. Because of this, there is no reason to generate relocatable object code. The assembly instructions are therefore translated into machine code directly by absolute assembly (Figure 16). For the same reason, there is no code optimization phase as with a compiler. Therefore, optimization is simply not practical for interpreted programs.

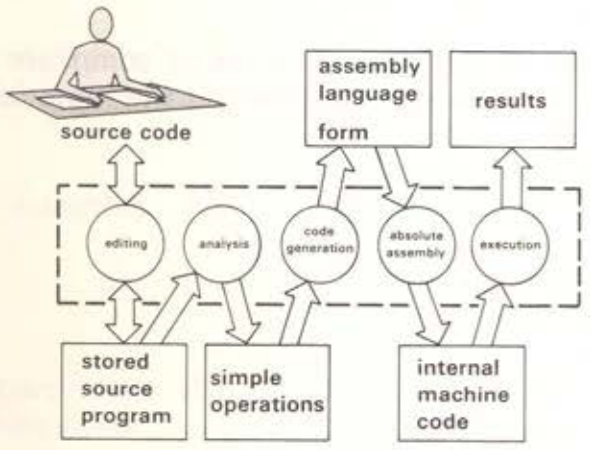


Figure 17 Execution Phase

The machine code for a single source statement is then executed (Figure 17). Once all operations for a statement have been executed, the interpreter begins the translation process on the *next logical* (not necessarily the next sequential) *statement.* In fact, each statement is

translated only as it is encountered, and it may be retranslated many times during the execution of the program. Execution continues until the program is finished, an error or "bug" is found, or the programmer stops the execution.

To conclude, interpreters possess significantly different characteristics from compilers:

- The *outputs* of an interpreter are the executed results of the program.
- *Execution speed* is relatively slow because each statement must be translated as it is encountered. This problem can be particularly time consuming in program loops.
- *Development time* is reduced as a result of the highly interactive nature of the editing process and the immediate production of results.
- *Interpreters* are not usually capable of *interfacing* with other interpreters, compilers, or assemblers. Hence programmers cannot mix languages within an application.
- *Programs* may only be *saved in source form,* as the various intermediate forms of the translation process only exist temporarily for any given statement.

As a summary of the characteristics of compilers and interpreters, the following is a tabular presentation of the material in this lesson.

Comparison of Compilers and Interpreters

Compilers	Interpreter
• Output is object program	• Output is executed result
• Execution time is faster	• Execution time is slower
• Development time is generally slower	• Development time is generally faster
• Flexible; may be interfaced with other languages	• Inflexible; may not usually be interfaced
• Programs may be saved in source or object form	• Programs may be saved in source form only

Compilation Process	Interpretive Process
1. *Compilation:* • Analysis and creation of table • Code generation • Optimization of code 2. *Assembly:* • Relocatable assembly into object code 3. *Linking:* • Linking of object code into internal machine code	1. *Editing:* • On-line development and editing 2. *Run:* • Analysis and creation of table for one statement • Code generation for one statement • Absolute assembly of one statement into machine code • Execution of the operations for one statement and start of process for next logical statement

EXERCISES

1. For both compilers and interpreters, list at least one advantage that each has over the other.

2. Describe the five steps of the translation process using a compiler.

SOLUTIONS

1. Compiler advantages over interpreters:

   a. Better execution speed.

   b. Flexible; language may be interfaced to other languages.

   c. Programs may be saved in either source or object form.

   Interpreter advantages over compilers:

   a. Shorter development time.

   b. Immediate results.

   c. Better programmer control over execution.

2. A compiler translates a high-level language to machine language in five steps:

   a. **Analysis** – Statements are checked for errors, and a table of simple operations is created which corresponds to all the statements.

   b. **Code generation** – The table of simple operations is translated on a 1:1 basis into assembly language instructions.

   c. **Code optimization** – The assembly language instructions are edited for unnecessary or useless instructions.

   d. **Assembly** – The optimized assembly code is relocatably assembled into object form.

   e. **Linking** – When the program is to be executed, the linker translates the object code into executable machine code.

EXERCISES

3. List three major differences between compilers and interpreters.

3. An interpreter differs from a compiler in several major points:

   a. The program is *entered interactively.* This allows errors to be found and corrected as they are made.

   b. Analysis and code-generation steps proceed as with a compiler *except that only one statement is worked upon at a time.*

   c. There is *no* code optimization performed.

   d. The assembly codes for a *single* instruction are assembled absolutely and *executed immediately.*

   e. The next statement to be interpreted is determined by the *program execution,* not by the sequence of statements in the source program.

Common High-Level Languages

OBJECTIVE

Given statements of applications and differences for the common high-level languages, be able to label those statements that refer to the FORTRAN, COBOL, or BASIC computer language, respectively.

SAMPLE TEST ITEM

Check the appropriate box or boxes next to each statement to indicate which language or languages best answers the statement.

	FORTRAN	COBOL	BASIC
1. Originally designed for business applications.	☐	☐	☐
2. Originally designed for educational applications.	☐	☐	☐
3. Originally designed for scientific/engineering applications.	☐	☐	☐
. . .	. . .	. . .	. . .

Please mark your place in this workbook and view Lesson 4 in the A/V program, "Programming Languages."

There are literally hundreds of high-level languages, differing from each other in the statements and expressions that are accepted by their translators and in the rules that are used to form statements. Many of these languages are special-purpose, locally used languages that have limited acceptance elsewhere. Of the dozen or so major languages, this lesson briefly examines three of the most popular.

FORTRAN

FORTRAN was the first high-level language to be widely accepted. Developed in 1954 and released to the general public in 1957, FORTRAN has subsequently been made available for almost every computer ever made. FORTRAN was the first programming language to have an industry-wide standard, and hundreds of thousands of programs have been written in FORTRAN.

FORTRAN was originally intended for the scientific and engineering fields, and its primary feature is the easy conversion of complex formula and equations into FORTRAN statements. As an example

$$S_p = \sqrt{\frac{P(10.7 - P)}{N^2}}$$

would appear in FORTRAN as

```
SP = SQRT (P * (10.7 - P)/N**2)
```

Thus standard algebraic notation is expressed in a similar fashion within FORTRAN. Indeed, the name FORTRAN is a contraction for FORmula TRANslator.

Another feature of FORTRAN is the ability to specify formats for input and output operations. This capability permits easy, readable output of the tables of numerical values that are characteristic of scientific and engineering problems.

Most FORTRAN translators are *compilers* rather than interpreters for three reasons:

1. Scientific and engineering programs are often run many times without changes, once they have been developed.

2. Problems in FORTRAN are frequently lengthy, and the faster execution time with a compiler is an important advantage.

3. In cases of critical operations requiring high efficiency, assembly language subroutines may be assembled into object code and then linked together with the output of the FORTRAN compiler to form a single machine code program. This procedure was described in the previous lesson on interpreters and compilers.

A severe deficiency of FORTRAN is the absence of an alphabetic variable type. This has eliminated FORTRAN from being used effectively in many non-numerical applications such as text processing, many business operations, and other activities where the manipulation of letters is more important than the manipulation of numbers.

A strong advantage of FORTRAN is a powerful subroutine capability. This is especially important as many scientific programs are both long and complex. By using subroutines, the programmer can break such a program into several smaller subroutines (or modules). The modular approach to programming accelerates the program development process because smaller modules are easier to code and easier to correct. Thus, program development costs can be significantly reduced.

COBOL

The mathematical background assumed by FORTRAN is neither common nor necessary in the business world. Accordingly, a language was developed, in 1959, which better suits the needs and backgrounds of commercial applications programmers. COBOL, the acronym for COmmon Business Oriented Language, has been standardized like FORTRAN, but unlike FORTRAN, COBOL was *designed* as a standardized language. Therefore, COBOL programs are more readily exchanged between different computers than are FORTRAN programs.

COBOL statements are very much like everyday English language sentences with all variable names and operations spelled out. For example,

MULTIPLY PRINCIPAL BY RATE GIVING INTEREST

is a legal COBOL statement. COBOL programs, therefore, are highly self-documenting.

A major feature of COBOL is the ease with which data can be manipulated and organized. Most commercial applications involve a great amount of data movement and the use of data files organized in various ways. COBOL was therefore designed to answer this need, while subordinating the ability to perform involved numerical calculations. There are also character handling facilities available in COBOL. Hence, COBOL is used for processing data involving letters (as are many business operations).

COBOL, like FORTRAN, is generally *compiled* rather than interpreted, because business programs are executed many times without changes. However, more computer time is generally used with COBOL programs than with FORTRAN programs.

The widespread acceptance and use of COBOL by the business world makes it one of the most commonly known and used programming languages.

BASIC

BASIC, the acromyn for Beginner's All Purpose Symbolic Instruction Code, was developed at Dartmouth College in 1965. It is intended to be a very simple language to learn, and is primarily used to train beginning programmers.

BASIC is very similar to FORTRAN in the expression of mathematical equations. Thus, the equation

$$c = \sqrt{a^2 + b^2}$$

becomes

```
LET C = SQR (A ↑ 2 + B ↑ 2)
```

Unlike FORTRAN, however, BASIC uses very simple input and output statements. There are no complicated format specifications required. Thus, to print the value of the variable X at the user's terminal, the programmer can merely write

```
PRINT X
```

BASIC translators are normally *interpreters* because the language's emphasis is on training, and the error detection-correction feature of interpreters is an important characteristic. BASIC compilers, however, are starting to appear as former student users of BASIC enter jobs in the scientific, engineering, and business worlds. Thus, some applications presently written in FORTRAN and COBOL are also beginning to appear in extended versions of BASIC. In particular, BASIC is now being used for business applications on several minicomputers where COBOL cannot be adequately implemented.

BASIC has good capabilities for manipulating letters as data. Applications such as text processing and report generation can be easily written in BASIC. On the other hand, BASIC has weak subroutine features. Therefore, modular programming is not easily accomplished.

Although a standard for BASIC has not yet been completed, a proposed standard does exist and is under review.

Remember that although FORTRAN, COBOL, and BASIC were each designed to benefit different application areas, there is some overlap. In general, however, some sacrifice in overall efficiency is required when these languages are used outside of their primary areas.

The following chart summarizes this lesson on the three high-level languages, FORTRAN, COBOL, and BASIC.

Characteristic	FORTRAN	COBOL	BASIC
Originally intended application area	Science and engineering	Business	Education
Primary advantages to intended application areas	Easy translation of mathematical equations Efficient execution of numerical calculations	English-like for self-documentation rather than mathematical Good data manipulation and definition abilities	Easy to learn Usually interactive
Usual Translator	Compiler	Compiler	Interpreter
Standard	Yes	Yes	In process

EXERCISES

1. Describe the fundamental differences between FORTRAN, COBOL, and BASIC by giving at least two primary advantages each offers to their respective areas of application.

   a. FORTRAN

   b. COBOL

   c. BASIC

2. Identify the application areas for which BASIC, FORTRAN, and COBOL were originally intended.

SOLUTIONS

1. Describe the fundamental differences between FORTRAN, COBOL, and BASIC by giving at least two primary advantages each offers to their respective areas of application.

   a. FORTRAN

      - Easy translation of mathematical equations into source code.
      - Efficient execution of numerical calculations.

   b. COBOL

      - English-like language which is self-documenting.
      - Good data manipulation and definition facilities for commercial programmers.

   c. BASIC

      - Easy to learn, so students start to program quickly.
      - Usually interactive to assist in correcting programs.

2. Identify the application areas for which BASIC, FORTRAN, and COBOL were originally intended.

   a. **FORTRAN** – Scientific and engineering applications, particularly when involving formulas and equations.

   b. **COBOL** – Business applications, particularly where little mathematical background or capability is assumed or required.

   c. **BASIC** – Education, particularly in the instruction of beginning programmers.

Take the test for this module and evaluate your answers before studying another module.