A SUGGESTED COURSE IN INTRODUCTORY COMPUTER PROGRAMMING

Warren A. Harrison
Kenneth I. Magel
University of Missouri, Rolla

Introduction

Introductory programming courses have long been a popular topic of discussion. Often it is either the only computer course a student takes or it is the foundation upon which all further training in computer science is built. The usual goal of such a course is to introduce the student to the use of a computer to solve simple problems in his or her particular discipline.

Generally the method of presenting the material may be separated into two distinct schools of thought, which we have termed the "Black Box School of Thought" and the "White Box School of Thought". In this paper, we discuss the benefits and drawbacks associated with the two alternative approaches. Additionally, we present a suggested course outline using the "White Box" method.

The Black Box and White Box Methods

The Black Box School of Thought seeks to teach the student how to use the computer while treating it as a black box. The student is taught how to insert the instructions at one end of the "box" and retrieve the results from the other end. Little time, if any, is spent on the internal workings of the machine. This approach is usually characterized by teaching only a high level language such as BASIC, COBOL or FORTRAN.

The White Box approach however, attempts to have the student view the computer as an assembly of various parts, such as memory, registers, etc. which interact according to the instructions in the program. As we define it, this approach is usually implemented by teaching a real or fictitious low level language before teaching the high level language.

The Students

Before determining which method would prove most effective, one must identify the characteristics of the students enrolled in the course. We have identified three major classes of students who may be attracted to an introductory course in computer programming.

The first grouping of students may be characterized, from the standpoint of computer science, as "dilettantes". In most cases, it is highly unlikely that the students in this group will make direct use of computers later in their careers. The largest part of this group is made up of students from fields such as Art, Music and History.

The second grouping of students is made up of individuals who are taking the course to develop skills which would help them use the computer in the solution of problems throughout their careers. Like the dilettantes, this course will probably be the only computer course they ever take. The members of this group include students from engineering, science and other quantitatively-oriented disciplines such as economics, business and certain areas of the other social sciences (e.g., sociology and psychology).

The third grouping includes both computer scientists and others who wish to use the computer extensively in their areas. The introductory course for these people is simply a foundation which they hope to build upon with additional courses in computer science.

Obviously, the objectives of the introductory course would vary greatly among the groups. The dilettantes, as their name would imply, would be better served by receiving a "sampler" of computer science, and a survey of the

field's effects upon society. The main concern with this group is "computer literacy". The typical "computers in society" course would be adequate to serve the needs of these students. Because of this, we will exclude the needs of this group of students from our following discussion of an introductory computer programming course.

The group of students viewing the computer as a tool for use in their fields must develop a certain level of programming ability. However, just as important is a background of sufficient depth to allow them to expand their ability through a reasonable amount of self-study after they leave school and begin to apply their programming skills.

The needs of the third group of students are actually quite similar to the needs of the previous group. Like the members of the second group, these people must develop skill at programming. Also like the second group, this group of students must have a broad background of general computer knowledge (i.e., language independent). Because most subsequent computer science courses will build upon the first course, a good general knowledge of computers, not just FORTRAN, BASIC, etc. should prove invaluable to the student.

Advantages of the Black Box Method

There are several reasons why the Black Box Method is often picked over the White Box Method. In most cases it is easy to rationalize because a high level language is probably the only type the student will ever have to use. With assembly language programming becoming less and less prevalent in industry, even the third group will seldom have to program in assembly language. Therefore, in some respects, a high level language is all the student will ever really have to use. In addition, by devoting the entire course to a single high level language, more time is available to pursue the intricacies of the language, and hence, more indepth coverage of the language's advanced features is possible.

If one also takes into account the fact that the high level language is machine independent, and therefore the specific knowledge the student has gained such as syntax, etc. is widely applicable, the Black Box Method becomes quite attractive.

Besides the reasons just mentioned, there are two more very important reasons why only a high level language is commonly taught. First, because of the complexity associated with most low level instruction sets, high level languages are usually much easier to learn. The second reason is that by teaching a single high level language the instructor's knowledge of the language is also widely applicable. The mobility afforded computer scientists today makes this especially important. It is possible that an instructor may teach at several different institutions over a short period of time. If each school uses a different low level language, the skills gained by the instructor would be of little use as he goes from school to school. On the other hand, high level language skills would transfer easily since BASIC, COBOL, FORTRAN, etc. remain very similar from machine to machine.

Advantages of the White Box Method

The White Box Method possesses some advantages which we feel compenates for the lack of some of the Black Box Method's advantages. To begin with, high level languages tend to hide the general functions of the computer. On the other hand, low level languages illustrate these functions. Mayer (2) has pointed out the importance of teaching at the transaction level. Mayer defines a transaction as a unit of programming knowledge which consists of an operation, an object and a location. While Mayer states that transactions do not require an understanding of machine-level operations, we feel that low level languages offer a "natural" method to present material at this level.

Since the White Box approach provides a model the student can examine if he desires, he can observe the effects of transactions upon various parts of the machine, and the interactions of those parts through the use of "selective" core dumps. This can help alleviate some of the abstraction inherent in most high level languages. A significant group of students have a great deal of difficulty dealing with abstraction. They tend to perform much better and understand much more when given something concrete to relate to. In addition to helping minimize abstraction, the model can give the student a good idea of the general organization of a computer.

Knowledge of the general organization of a computer can help build the foundation needed if the student is to pursue advanced programming topics on his own. Aspects of programming which were not covered in class can be anticipated and understood on the basis of the model.

Familiarity with a low level language can also help the student understand the concept of source language translation. This can provide an appreciation and better understanding of compilers and the compilation process.

Actual or Simulated Boxes?

While it is possible to present the

internal organization of a computer by other methods, there appears to be a general concensus among the adherents of the White Box Method that using a low level language to illustrate the organization and operation of the computer reaps the greatest benefits. The supporters of this method are split however upon the vehicle used to support such a language.

One camp maintains that a fictitious machine, designed for this purpose be implemented upon a host computer. For example, Sebesta and Kraushaar (4,5) and Wainwright (6) have developed such psuedo-machines.

The rationale for using a fictitious machine is that the complexity of a real computer precludes its use with beginning students. The use of a fictitious model allows extremely simple assembly and/or machine languages to be developed for purely pedagogical use.

Fictitious machines have certain problems associated with them however. The primary problem is that they give the student an unrealistic, over-simplified view of a computer. While it is certainly desirable for the language to be simple, it should not be unrealistically so. This tends to isolate the student from a number of features that will no doubt have an effect upon his programming later in the course, such as internal representation of floating point and integer values, two's complement, word lengths, etc.

The use of a fictitious machine is also a "dead end" approach to introducing beginners to programming. After the student is finished with the simulation, the knowledge gained of the particular "machine" cannot be used again except in a strictly general sense.

Yet another problem associated with using a fictitious machine arises from its implementation upon the host machine. It is very possible that the implementation may have errors in it. When the student encounters these errors, he will be exposed to the underlying host machine through error messages, etc. which are not related to his view of the problem. This can confuse and discourage the student.

The second camp feels that the added complexity of using a real machine is more than sufficiently compensated for by the realism inherent in this method. The actual machine, with all its limitations and "rough edges" provides a more useful model than the "perfect" mythical machine for explaining errors and other phenomenon in an actual operating environment.

The main complaint against actual low level languages is their complexity. The complexity may be minimized greatly by using an extremely basic subset of the full instruction set available for a particular machine.

The instruction subset learned in the introductory course may be expanded upon for use in a more advanced course. Courses in operating systems, language translators, etc. could all make use of an expanded instruction set, built up from the foundation supplied by the first course. For this reason, the use of a real machine is not the "dead end" approach to learning as is the fictitious machine.

An additional virtue of using a real machine as opposed to a fictitious machine is that the student has a start on a marketable skill. With additional study, the student could easily improve upon his skills in low level programming for a particular machine. Likewise, it would prepare the student to transfer from machine to machine more easily than a fictitious simulation would.

The Role of the Microcomputer

A microcomputer is a natural choice for the vehicle to be used to present this knowledge to the student. Microcomputers provide a most favorable interactive environment. The student can enter his program, either through the front panel or a CRT keyboard, and have complete control over the machine's resources. More importantly, he is not insulated from the machine by a card reader or miles of telephone line.

Microcomputers also tend to be quite a bit simpler than their larger counterparts. This facilitates reduced complexity. In addition to being simpler, microcomputers are also much smaller physically than large computers. This can aid in reducing the intimidation felt by many beginning programmers when faced with the experience of being one of many users working on an anonymous machine which requires a special air-conditioned facility larger than his classroom.

Other aspects of microcomputers that make them attractive is the fact that they are becoming widely available. They are relatively inexpensive and require no special operating environment (e.g., air conditioning).. In the near future, microcomputers will start to be widely available in student homes.

Instruction Subset

It is desirable to limit the subset of instructions to be used to an absolute minimum in order to reduce complexity. The minimum subset should contain all those instructions which are usually associated with a classical single accumulator machine. These instructions are generally

agreed to include:

- Load accumulator from memory
- Store contents of accumulator in memory
- Add contents of memory to contents of accumulator
- Subtract contents of memory from contents of accumulator
- Unconditional transfer of control
- Conditional transfer of control
- Input and Output

and may be easily implemented on most microcomputers with a subset of a dozen or so instructions. As an illustration of how such a subset may be developed, Appendix A lista a subset for the popular 8080/8085 based microcomputer.

Pragmatic Considerations

The main idea behind the use of a low level instruction subset is to present the general ideas of computer organization and operation. Because of this, the low level language should be used only until the student has an adequate understanding of the simplified computer.

The primary goal of an introductory programming class is to introduce students to the computer as a tool for solving problems. It is almost universally recognized that problem-oriented languages such as BASIC and FORTRAN are more suitable for actually solving problems than are assembly languages. Fot this reason, it is incumbent upon an introductory course to provide students with an acquaintance with such a language.

The introduction and use of a problem-oriented language should follow after the low level language programming has been completed. This will allow the student to build on the knowledge, understanding and techniques gained during the first part of the course.

As the student is introduced to new statements in the high level language, the general relation of the statement to corresponding groups of low level instructions may be pointed out. This allows the student to grasp the relationship between the individual low level instructions, the various parts of the machine, and the actual high level language instructions.

Course Content

Because the goal of the introductory programming course is usually to introduce students to the use of a computer to solve problems, the first part of the course should deal with how to solve problems, regardless of whether a computer is being used or not. Generally the process of problem solving is dealt with in introductory text books, for example, Moore and Makela (3) and Feingold(1). The process, in general, is characterized as consisting of the following steps:

- Problem definition
- Formulation of a procedure to produce the solution
- Coding the procedure in a programming language
- Entering and running the job
- Locating and correcting errors.

The problem solving section of the course should deal exclusively with the first two items of the above list. The instructor should present several non-trivial examples in class, and then guide the students through several more. General techniques of problem definition and formulation of solutions should be described and illustrated in the examples. Tricks and subtleness should be avoided if possible. This is not to suggest, of course, that problem definition and solution formulation be neglected during the remainder of the course. On the contrary, the process of problem definition and formulation of solutions should be present throughout the entire course.

Following the problem solving material, the second part of the course should deal with the concepts of machine organization and low level language programming. By the time this portion of the course is completed, the student should have an adequate grasp of the relationship of the different parts of the machine to each other, and the way these parts are affected by the program during execution.

The third and final part of the course should introduce the use of a high level programming language. This part of the course should, in addition to teaching the student how to use a high level language in an effective manner, teach them a disciplined approach to programming by way of the concepts of structured programming.

A course outline for a typical fifteen week session using WATFIV-S as the high level language is shown in Figure 1.

Summary

The course described in this paper has been developed to address several problems common to most introductory programming courses. Primarily, through the use of

the White Box Method, we have provided an alternative to the "cookbook" programming inherent in the traditional Black Box Method. The approach presented here not only teaches the student how to instruct the computer to solve problems, but it also provides some insight into what those instructions cause to happen inside the computer.

WEEK	CONTENT
1	Problem Solving: Problem definition
2	Problem Solving: Development of problem solutions
3	Problem Solving: Role of systems thinking in problem solving
4	Machine Organization: CPU; Memory; I/O; Accumulator; Number systems; etc.
5	Machine Level Programming: Load; Store; Arithmetic and Output instructions; Program testing
6	Machine Level Programming: Input; Conditional and Unconditional branch instructions; looping; counters; Program testing
7	Machine Level Programming: Array concepts via machine language
8	Machine Level Programming: Simple subprogram concepts
9	Concepts of code translation: Symbolic memory locations; idea of compilers
10	FORTRAN: Character set; constants; variables; operators; expressions; assignment statements; simple I/O
11	FORTRAN: Unconditional GOTO; Computed GOTO; DO CASE; Arithmetic and Logical IF; IF-THEN-ELSE
12	FORTRAN: DO-LOOPS; WHILE DO; Looping
13	FORTRAN: Arrays; Array I/O; Additional specification statements
14	FORTRAN: Formatted I/O; Subprograms
15	FORTRAN: Subprograms

Figure 1. Sample outline for introductory programming course using the White Box Method and FORTRAN (WATFIV-S).

References

1. Carl Feingold, Fundamentals of Structured COBOL Programming, Wm. C. Brown Company, Dubuque Iowa, 1978.

2. Richard E. Mayer, "A Psychology of Learning BASIC", Communications of the ACM, 22, 11 (November 1979), 589-593.

3. John B. Moore and Leo J. Makela, Structured FORTRAN with WATFIV, Reston Publishing Co., Inc., Reston, Virginia, 1978.

4. Robert W. Sebesta and James M. Kraushaar, "TOYCOM - A Tool for Teaching Elementary Computer Concepts", Proc. SIGCSE 11th Technical Symposium, February 1980, 58-62.

5. R.W. Sebesta and J.M. Kraushaar, "A Simple Computer Model for Teaching Introductory Computing", Proc. ACM Computer Science Conf., February 1980, 57.

6. Roger L. Wainwright, "An Introductory Computer Science Course for Non-Majors", Proc. SIGCSE 11th Technical Symposium, February 1980, 154-160.

APPENDIX A

The instruction subset should consist of the absolute minimum number of instructions in order to keep complexity within acceptable bounds. The instructions chosen to be included within this subset should correspond somewhat to those instructions associated with a classical Von Neumann computer. Additional instructions may also be included for the sake of pedagogy. By carefully selecting the instructions to be used in the subset, minimum complexity may be achieved with little loss of generality. It should be noted that it is important for the subset to maintain consistency wherever possible regarding addressing modes.

In order to maintain the characteristics of a single accumulator machine only one register, used as an accumulator, need be introduced in some cases. However, due to the fact that register indirect addressing is inescapable with many machines, additional registers may need to be introduced. If this is the case, the register(s) used for indirect addressing should be used only for that purpose (i.e., they should not be used as additional general purpose registers...this confuses the student and adds to the complexity of the model.

A suggested subset for the popular 8080/8085 based microcomputers follow. The reader should note that both the numeric opcode and mnemonic instruction is given. This allows the instruction to be first introduced in numeric form, and then during the discussion of the translation process, the mnemonic form may be introduced.

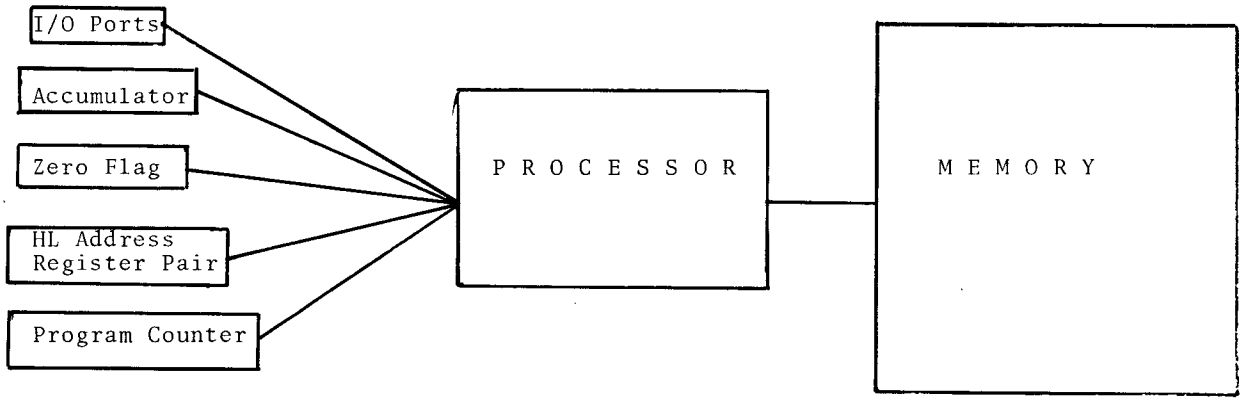
The numeric opcodes of the 8080/8085 based micros are similar to those of the Z-80. The mnemonics however, differ. Therefore, the numeric opcodes listed in this appendix may also be used as a Z-80 instruction subset. The mnemonics must however, be supplied by the reader.

This machine requires the use of indirect addressing. For this reason, a set of three, 8 bit registers must be introduced to the student. These three registers are: the A register, used as the accumulator and referenced in the instruction summaries as 'A', and the HL register pair used for indirect memory reference and referenced as 'M' in the instruction summaries.

In addition to being used for the arithmetic operations, the accumulator is also used to hold data for I/O transfer and in operations affecting the condition flags.

Five single bit condition flags are provided with the architecture. Only one of these, the 'zero flag' is used with the selected instruction subset. Operations involving this flag are discussed later.

Each time an instruction byte is fetched from memory, the processor increments the 16 bit program counter by one. When the sequential flow of execution is altered by a jump instruction, the processor replaces the contents of the program counter with the address of the new instruction. The next byte fetched is from the location specified by the new address. The selected subset provides the following model for the student to work on.



A description of the instructions included in the subset follow. Note that the Z-80 uses different mnemonics, but has the same numeric opcodes. Many of the instructions, due to complexity considerations, are not used to their full capacity.

- Load Accumulator from Memory
  Opcode: 7E
  Mnemonic: MOV
  Form: MOV A,M
  Action: A ← c(c(M))

- Store contents of Accumulator in Memory
  Opcode: 77
  Mnemonic: MOV
  Form: MOV M,A
  Action: c(M) ← c(A)

- Load HL Register pair Immediate*
  Opcode 21
  Mnemonic: LXI
  Form: LXI H,address
  Action: M ← address

- Load HL Register pair Direct
  Opcode: 2A
  Mnemonic: LHLD
  Form: LHLD address
  Action: L ← c(address),
  H ← c(address+1)

- Add contents of Memory to Accumulator
  Opcode: 86
  Mnemonic: ADD
  Form: ADD M
  Action: A ← c(A)+c(c(M))

- Subtract contents of Memory from Accumulator
  Opcode: 96
  Mnemonic: SUB
  Form: SUB M
  Action: A ← c(A)-c(c(M))

- Unconditional Transfer of Control*
  Opcode: C3
  Mnemonic: JMP
  Form: JMP address
  Action: Continue execution at specified address

- Compare contents of Memory with contents of Accumulator
  Opcode: BE
  Mnemonic: CMP
  Form: CMP M
  Action: If c(A)=c(c(M)) 'Zero Flag' ← 1, otherwise 'Zero Flag' ← 0 (i.e., checks if the difference is zero)

- Transfer Control if 'Zero Flag' is zero* (transfer if difference ≠ 0)
  Opcode: C2
  Mnemonic: JNZ
  Form: JNZ address
  Action: If 'Zero Flag' is 0, then continue execution at the specified address

- Transfer Control if 'Zero Flag' is one* (transfer if difference = 0)
  Opcode: CA
  Mnemonic: JZ
  Form: JZ address
  Action: If 'Zero Flag' is 1, then continue execution at the specified address

- Increment Register pair**
  Opcode: 23
  Mnemonic: INX
  Form: INX H
  Action: Adds 1 to contents of HL Register pair (M)

- Input***
  Opcode: DB
  Mnemonic: IN
  Form: IN port #
  Action: Loads Accumulator with 8 bits of data from the specified port

- Output***
  Opcode: D3
  Mnemonic: OUT
  Form: OUT port #
  Action: Sends contents of the Accumulator to specified port

- Terminate Execution
  Opcode: 76
  Mnemonic: HLT
  Form: HLT
  Action: Halts processor

---

*3 byte instruction, byte 1 is opcode, bytes 2 and 3 is address-specified in hex form.

**To be used only on HL Register pair when illustrating array concepts.

***Port number may be constant 00H thru 0FFH