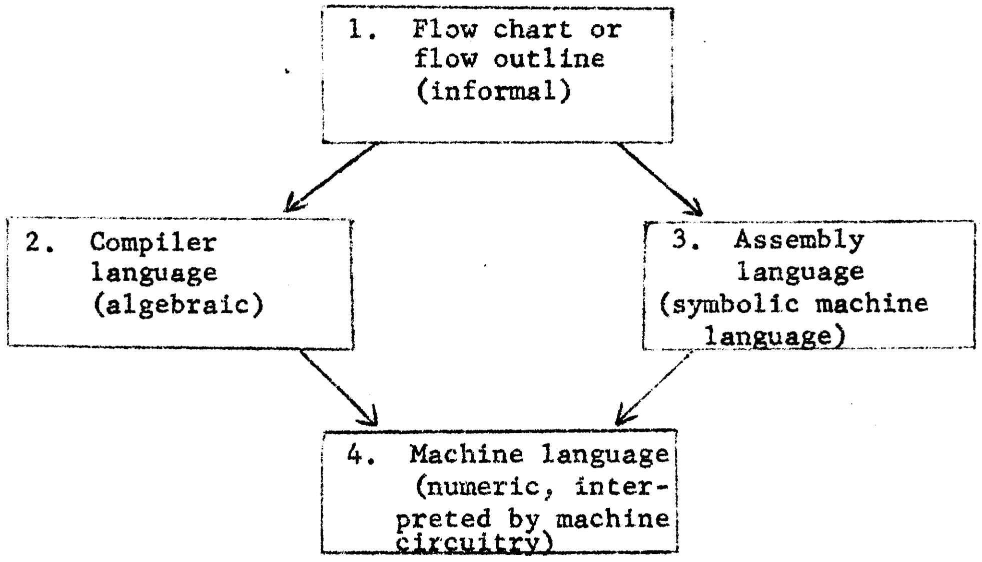
COMPUTER LANGUAGES

D. E. Knuth -- December, 1964

The purpose of this lecture is to present, by means of examples, various levels of language used with digital computer programs, and also to give an introduction to programming.

The four principal types of language used with computers are depicted in this chart:


1. Flow chart or flow outline (informal)
2. Compiler language (algebraic)
3. Assembly language (symbolic machine language)
4. Machine language (numeric, interpreted by machine circuitry)

1. The flow chart or flow outline is a step-by-step description of the procedure to be followed. A computational procedure must be spelled out in detail, much like a recipe in a cookbook. The word algorithm is used to denote a finite computational process which is well-defined and which produces specified output when given specified input. A description of an algorithm in a computer language is called a program.

Flow charts and flow outlines serve as an aid to the person preparing a program; most programmers like to get their thoughts in order by first preparing a flow chart which shows how to carry out the desired solution. Flow charts have a wide range of levels, depending on the programmer's taste; he may wish to give a brief overall picture of the "flow" of an algorithm, or he may wish to make an extremely detailed flow description. Language used in flow charts is usually a combination of English and mathematics; in general, the language is sufficiently precise for the programmer to understand, but is far too vague for a computer to understand (at least until we build a computer that thinks). In other words, it takes common sense to determine the meaning of a flow chart description of a program. This is not true for the other languages we describe; they will conform to specific rules, and every statement will have a precise meaning, understandable by the computer.

2. Compiler language is a precise rendering of the flow chart into a well-defined language, which still remains sufficiently close to English and mathematical language to make it easily readable for a non-specialist (after a short briefing). There are dozens of different compiler languages, of which the most commonly used are ALGOL and FORTRAN. (Computer programmers and manufacturers are fond of using "acronyms" for naming their programs; ALGOL stands for ALGOrithmic Language, FORTRAN stands for FORmula TRANslator.) At CalTech, the ALGOL compiler language is presently used for programs on the Burroughs 220 computer. FORTRAN, which was actually the first compiler language ever designed in this country still survives today and is presently used for programs on the IBM 7094 computer. The example compiler language to be given in this lecture will be FORTRAN.

3,4. The machine language of a computer is the code in which instructions are stored inside the machine. The computer automatically analyzes this code and carries out the instructions mechanically. People find it harder to analyze such code, but machine circuitry naturally finds numerically coded information easy to recognize. Assembly language is directly related to machine language, except it is designed for human comprehension. The programmer does not have to remember many of the details of machine language code; these are abbreviated into a symbolic language, closely related to the way people actually think about the machine instructions. This makes the creation of machine language programs in the equivalent assembly language rather easy. Another important advantage is the programmer's freedom from worrying about clerical details and, consequently, he makes less errors. The programmer who uses an assembly language must still, of course, be intimately familiar with the machine characteristics, but a trained programmer can often dash off an assembly language program as quickly as a flow chart. There will not be time in this lecture to discuss assembly language and machine language in great detail; only the flavor of those languages will be demonstrated.

I. Flow Diagrams

The problem we will investigate here is a simple everyday problem of making exact change for a given amount of money, using the fewest possible coins in the process. We will assume that the amount to be paid is less than $10.00, and that we have enough coins of each kind to make up any such amount. (The actual assumption is that we have at least one $5 bill, four

$1 bills, one half-dollar, one quarter, two dimes, a nickel, four pennies, although we will never have to use all of these coins at once.) A simple method to use for this problem is to find the largest coin less than or equal to the amount, to take one of this coin, and repeat the process with the remaining amount, until the total is reached.

The most important thing to know about computers is that they understand very little about what they are told. Every part of a process must be expressed clearly and unambiguously in simple basic steps. Merely to say "find the largest coin less than or equal to the amount, etc." is quite insufficient; it is necessary to say <u>what</u> the coins are, <u>how</u> to search for the largest one less than or equal to the amount, <u>when</u> it has been found, etc.

A more precise way to state this algorithm is in terms of the flow chart below. The flow chart itself is not very explicit about what is to be done; it is accompanied by a <u>flow outline</u>, a set of steps explaining not only what is to be done in the algorithm but why it is done. It is a good idea to prepare such a detailed description so that one can remember several months later what procedure was used.

<u>Algorithm for making change</u>

S1. An AMOUNT is input for which we are to make up change.

S2. Set J = 1. [J will later specify which coin is currently being tried. The coins are (in terms of cents) 500, 100, 50, 25, 10, 5, 1 in that order.] Also set K = 0. [K equals the number of coins accumulated so far as change.]

S3. If AMOUNT is less than the J-th coin, the J-th coin is too big so we go to step S4. If AMOUNT is greater than or equal to the J-th coin, go to step S5.

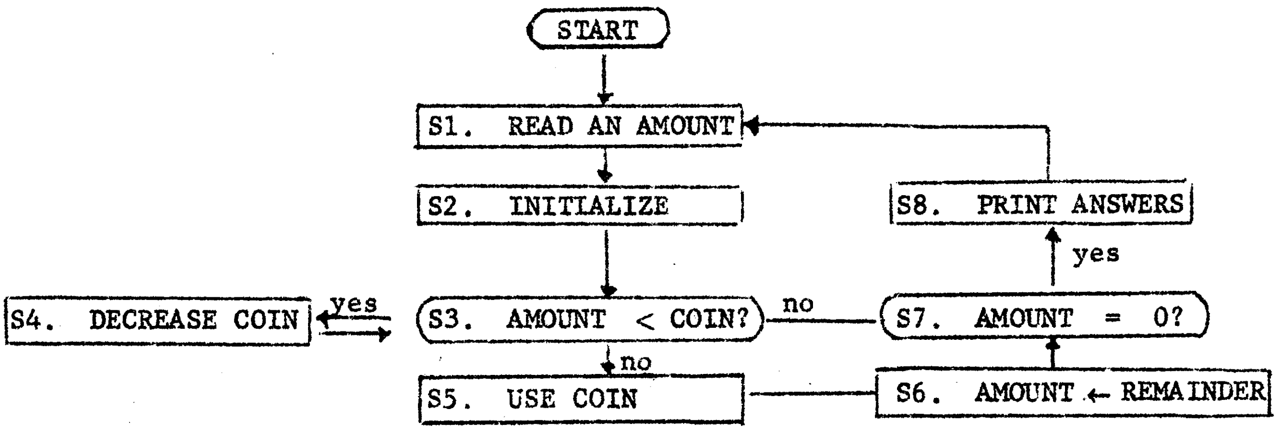
S4. Increase J by 1 (thus specifying the next smaller coin) and return to step S3.

S5. We take one of the J-th coins as part of the change. This means increase K by one, and set the K-th item of change to the J-th coin.

S6. Decrease AMOUNT by the value of the J-th coin; this represents the amount we must still make up.

S7. If AMOUNT is zero, we are done, so we go on to step S8; otherwise go back to step S3.

S8. Print out the answers for this case, then return to step S1.



II. Compiler Language

Compiler language is quite similar to a flow description; the main differences are that compiler language has a specified form for each kind of operation, and that things are usually adjusted so they are all on one line. Instead of writing $\frac{A}{B}$ one writes A/B; instead of writing X_2 one writes X(2); instead of writing X^2 one writes X**2; instead of writing $\sqrt{X}$ one writes SORT(X). Further explanations of the meaning of each line of the FORTRAN program given below (each line is a so-called "FORTRAN statement") appears together with the statements.

FORTRAN Statements	Explanatory Notes
C MAKING CHANGE	The "C" means this is a comment only.
DIMENSION COIN (8) x CHANGE (13)	This says there are eight coins called COIN (1), COIN (2),...COIN (8), and that there are at most thirteen items of change, called CHANGE (1), CHANGE (2),... CHANGE (13). Note: The "x" here means that the information on that line is a continuation of the preceding line.
COIN (1) = 500 COIN (2) = 100 COIN (3) = 50 COIN (4) = 25 COIN (5) = 10 COIN (6) = 5 COIN (7) = 1 COIN (8) = 0	This series of statements sets up the appropriate values of the coins.

1	READ (5,100) AMOUNT	This causes a unit of information from a punched card to enter the computer; this record is prepared in format number 100 (see format statements below); the data item is stored in AMOUNT. If no more data is present, the computer terminates the program.
	TOTAL = AMOUNT	Set the variable TOTAL equal to the value of the variable AMOUNT. This is done so that when later printing out the answers we remember the original amount.

2	J = 1
	K = 0
		The variables J and K are set respectively to 1 and 0.

3	IF (AMOUNT - COIN (J)) x 4,5,5	The quantity AMOUNT-COIN(J) is computed. If it is negative, go to statement number 4; if it is zero, go to statement number 5; if it is positive, go to statement number 5. (Compare with step S3 of the flow outline.)
4	J = J + 1	Increase J by 1. Notice that an "=" sign in FORTRAN has a different meaning than its conventional mathematical usage; it denotes the action of taking the value on the right and using it as the new value of the variable on the left.
	GO TO 3	Return to step 3
5	K = K + 1 CHANGE (K) = COIN (J)	Increase K by 1 Set CHANGE(K) equal to COIN(J) (Compare with step S5)

6 AMOUNT = AMOUNT - x COIN (J)	Decrease AMOUNT by COIN(J)
7 IF (AMOUNT) 3,8,3	If AMOUNT is now negative or positive (i.e. non-zero), return to step 3; if AMOUNT is zero, go to step 8.
8 WRITE (6,200) x TOTAL, (CHANGE(I), x I = 1, K)	Record onto the output page a unit of information containing the original TOTAL amount, and the values of CHANGE(1) through CHANGE(K), using format number 200.
GO TO 1	
100 FORMAT (F8.0)	This is a special notation for the format in which the input cards are punched. The code F8.0 means eight card columns are used, and the numbers are given without any decimal places.
200 FORMAT(9HOAMOUNT = x -2P F7.2, 7H, COINS, x 7F7.2/30X, 6F7.2)	This format for the answers says to double space, to print titles in the form "AMOUNT = x.xx, COINS", to convert from cents to dollars for readability, and to give up to 13 answers with at most seven on the first line.
END	This indicates the end of the program.

III. Assembly Language and Machine Language

It would take one or two further lectures to explain the details of assembly language and machine language so we will be content here to merely present an example of each. The important thing to learn at this time is merely the fact that these languages exist, and to get some idea what they look like.

The following assembly language program (written in a language called MAP) corresponds to that part of the FORTRAN program above, following statement number 1 up to and including statement number 7.

```
Name    Op     Address, Index
        STO    TOTAL
        AXT    1,1
        AXT    0,2
COMP    CAS    COIN,1
        TXI    USE,2,1
        TXI    USE,2,1
        TXI    COMP,1,1
USE     LDQ    COIN,1
        STQ    CHANGE,2
        FSB    COIN,1
        TNZ    COMP
```

The reader is not expected to understand this language at this time, but we can give some idea of the correspondence: the two lines "LDQ COIN,1" and "STQ CHANGE,2", which are the 3-rd and 4-th last lines here, correspond to the FORTRAN statement "CHANGE(K) = COIN(J)."

The machine itself works in quite another kind of language: it deals with signed 35-bit numbers in the binary number system. Here is an example of machine language:

Location	Machine language instruction
01000	+001100000010000000000000010000000000
01001	+001111111000000000100000000000000001
01002	+001111111000000000100000000000000000
01003	+000111000000000000010000110000000000
01004	+010000000000000010100000010000000111
01005	+010000000000000010100000010000000111
01006	+010000000000000010010000010000000011
01007	+001011100000000010000000110000000000
01010	-001100000000000000100001000000000000
01011	+000110000000000000010000110000000000
01012	-000010000000000000000000001000000000

This is in a form that the computer circuitry understands. The "location" here means the place within the computer's memory in which the instruction is stored.

This machine language program is the exact equivalent of the assembly language given above; i.e., the first instruction is the equivalent of "STO TOTAL", the next is the equivalent of "AXT 1,1", and so on. The point here is that assembly language and machine language are very closely related; the former is a symbolic form of the latter.

III. Discussion

Flow charts and compiler languages are customarily called problem-oriented languages, and assembly or machine languages are known as processor-oriented or machine-oriented languages. The big advantage of problem-oriented languages, as far as most scientists are concerned, is that it is for the most part unnecessary for the programmer to learn the details about any specific computer, he can write up programs in a fairly familiar way with little extra training. Specially written computer programs (called compilers) take anything written in a compiler language and automatically translate it into machine language; then the machine executes the resulting machine language program.

Assembly languages are valuable to programmers who are trained in the intricacies of machine language, as mentioned earlier; other specially written computer programs (called <u>assemblers</u>) translate assembly language into machine language. In fact many compilers translate from the compiler language into assembly language (as indicated by the broken line in the diagram), and an assembler finishes the job.

There is bound to be some loss of efficiency resulting from the automatic translation of compiler language into machine language. However, the programs produced run 90-95% as fast as those written in assembly language by a trained programmer. In fact, compiler-produced programs tend to beat hand-coded problems written by an average programmer when the problem is complex and lengthy.

There are, on the other hand, classes of problems for which compiler output compares poorly with hand-written programs; this is due to a number of features of the machine which are never utilized by the output of a compiler, because of the nature of compiler language. Problems which make heavy use of such facilities are much better when hand-written. (For some combinatorial problems, for example, the compiler programs are less than one percent efficient! It is to be emphasized, however, that such problems are definitely in the minority.)

<u>Exercises</u>

1. Rewrite the FORTRAN algorithm so as to allow $2 bills in the change.
2. Would our algorithm always produce the least number of coins, if we were not allowed to give nickels? (Hint: consider the case of 30¢.)
3. What would happen in our program if the original amount were zero?

```
C      MAKING CHANGE
       DIMENSION COIN(8), CHANGE(13)
       COIN(1) = 500
       COIN(2) = 100
       COIN(3) = 50
       COIN(4) = 25
       COIN(5) = 10
       COIN(6) = 5
       COIN(7) = 1
       COIN(8) = 0
     1 READ(5,100) AMOUNT
       TOTAL = AMOUNT
     2 J = 1
       K = 0
     3 IF(AMOUNT - COIN(J)) 4,5,5
     4 J = J + 1
       GO TO 3
     5 K = K + 1
       CHANGE(K) = COIN(J)
     6 AMOUNT = AMOUNT - COIN(J)
     7 IF(AMOUNT) 3,8,3
     8 WRITE(6,200) TOTAL, (CHANGE(I), I = 1,K)

       GO TO 1
   100 FORMAT (F8.0)
   200 FORMAT (9H0AMOUNT =, -2P  F7.2, 7H, COINS,
      X         7F7.2 / 30X, 6F7.2)
       END
```

```
AMOUNT =   0.25, COINS   0.25

AMOUNT =   0.26, COINS   0.25   0.01

AMOUNT =   0.40, COINS   0.25   0.10   0.05

AMOUNT =   0.93, COINS   0.50   0.25   0.10   0.05   0.01   0.01   0.01

AMOUNT =   0.95, COINS   0.50   0.25   0.10   0.10

AMOUNT =   9.99, COINS   5.00   1.00   1.00   1.00   1.00   0.50   0.25
                                0.10   0.10   0.01   0.01   0.01   0.01
```