```
x : integer;
y, z : real;
...
x := y + z;   ⇐
...
```

- Interpreter must know the types of x, y and z while executing the statement marked with '⇐'. Existence of the symbol table helps to fulfill this requirement.

4.1 Merits and demerits of interpreters

To compare the execution characteristics of compilers and interpreters, consider the following notation which represents the average CPU time cost for different kinds of processing of a statement :

t_i : Interpretation time per statement
t_c : Compilation time per statement
t_e : Execution time of a compiled statement

There are two reasons to assume $t_c \cong t_i$. First, compilers and interpreters both involve lexical, syntax and semantic analysis of the source statement. Second, the code generation effort for a statement performed by the compiler is of the same order of magnitude as the effort involved in interpretation of the statement. We can further observe that $t_i = n \cdot t_e$, $n \geq 10$ because of the following reasons.

1. Execution of a compiled statement involves execution of a few machine instructions, while its interpretation would involve execution of hundreds of instructions. For example

   $a := b + c;$

   may be compiled into only 3 machine instructions.

2. The value of n would depend on the features of the source language. For a language 'rich' in data types, n would be larger due to the need for type conversions, whereas for a language with a single numerical type (e.g. Dartmouth Basic), n would be smaller.

Using the above information, we can argue that interpreters are superior to compilers in a program development environment. We start by characterizing a program development environment as follows:

~~22 Dec~~ 17 Dec
24 Dec

1. During program development, a user is typically testing and debugging his/her program, i.e.

   (a) running the program on known data to evaluate its correctness

   (b) editing the program to remove logical errors

2. Test data is selected so as to restrict the amount of execution of the program. Thus loops in the program are made to execute only a few times (rather than a few hundred times). This is done to curtail the cost of debugging the program. Hence if a program consists of, say, 300 statements, the number of statements actually visited during the test run would be typically fewer than 300.

3. Every test run is typically followed by some modifications to the program. Hence if a compiler is being used, the program will have to be compiled for every test run.

Now consider a 300 statement program being executed on a test data such that only 75 statements are visited during the test run. This can happen because

1. Depending on the values in the test data, only selected portions of the program may be visited during the test run.

2. Program loops may be executed only a few times.

Comparative performance of a compiler and an interpreter in this case would be as follows : Total CPU time in compilation followed by execution of the program is $300 \cdot t_c + 75 \cdot t_e \cong 307.5 \cdot t_c$, while total CPU time in interpretation of the program is $75 \cdot t_i \cong 75 \cdot t_c$.

Thus, interpretation would be cheaper in this case. This advantage would persist if up to 300 statements are executed during a test run. If more than 300 statements are to be executed, compilation followed by execution would be cheaper. Hence, from the point of view of the CPU time cost, ① interpreters are well suited for a program development environment, but not for a production environment.

Apart from the speed of execution, we would also like to consider the ease of developing an interpreter and porting it from one system to another. Here we find that ② Interpreters are simpler to develop than compilers because compilers have to generate target machine code, which is a complex task, while interpreters do not have to perform code generation. ③ Interpreters can also be made portable by carefully coding them in a higher level language, whereas compilers are 'bound' to a specific target machine and can not be ported.

4 Interpreters also permit *dynamic specification* of program elements during the execution of the program. For example, in SNOBOL the type of a variable is determined by the type of the value assigned to it. Hence, types can change during program execution. Consider the following program

$$\begin{array}{ll} X = 23.5 & \\ \ldots & \\ X = \text{'356'} & \\ Z = Y + X & \qquad \Leftarrow \end{array}$$

The statement marked with $\Leftarrow$ cannot be compiled, since the type of X (whether a real, or a string) depends on which of the execution paths is taken during a run (hence, the compiler can not generate code for it). However, the program can be interpreted. The interpreter can keep track of the type of X as different statements are interpreted. So it will know the correct type while interpreting the statement.

5 Interpreters also have another significant advantage over compilers in that self-modifying programs, i.e. programs that modify themselves or 'evolve' during execution, can be interpreted. However the same is not true about compilers.

From the above discussion, we can summarize the use of interpreters as follows : Interpreters can be used in program development environments, and in environments where only one execution is typically desired, e.g. data base or operating system commands. Interpreters are also useful in environments where interpreter may be smaller in size than a compiler, and in environments where dynamic program modification is an advantage

4.2 An example of interpretation

In this example, we consider different steps in the interpretation of a basic statement to illustrate how type conversion, etc. are carried out in a machine independent manner. This in turn illustrates how the writing of an interpreter is simpler than writing of a compiler, and how writing an interpreter in a higher level language makes it machine independent.

Consider the following source program in Basic

```
INTEGER I
REAL A, B
LET I=7 B=1.2
LET A=B+I
```

While interpreting this program, the interpreter needs to know the types of the variables. It makes use of the symbol table to keep this information. The interpreter also