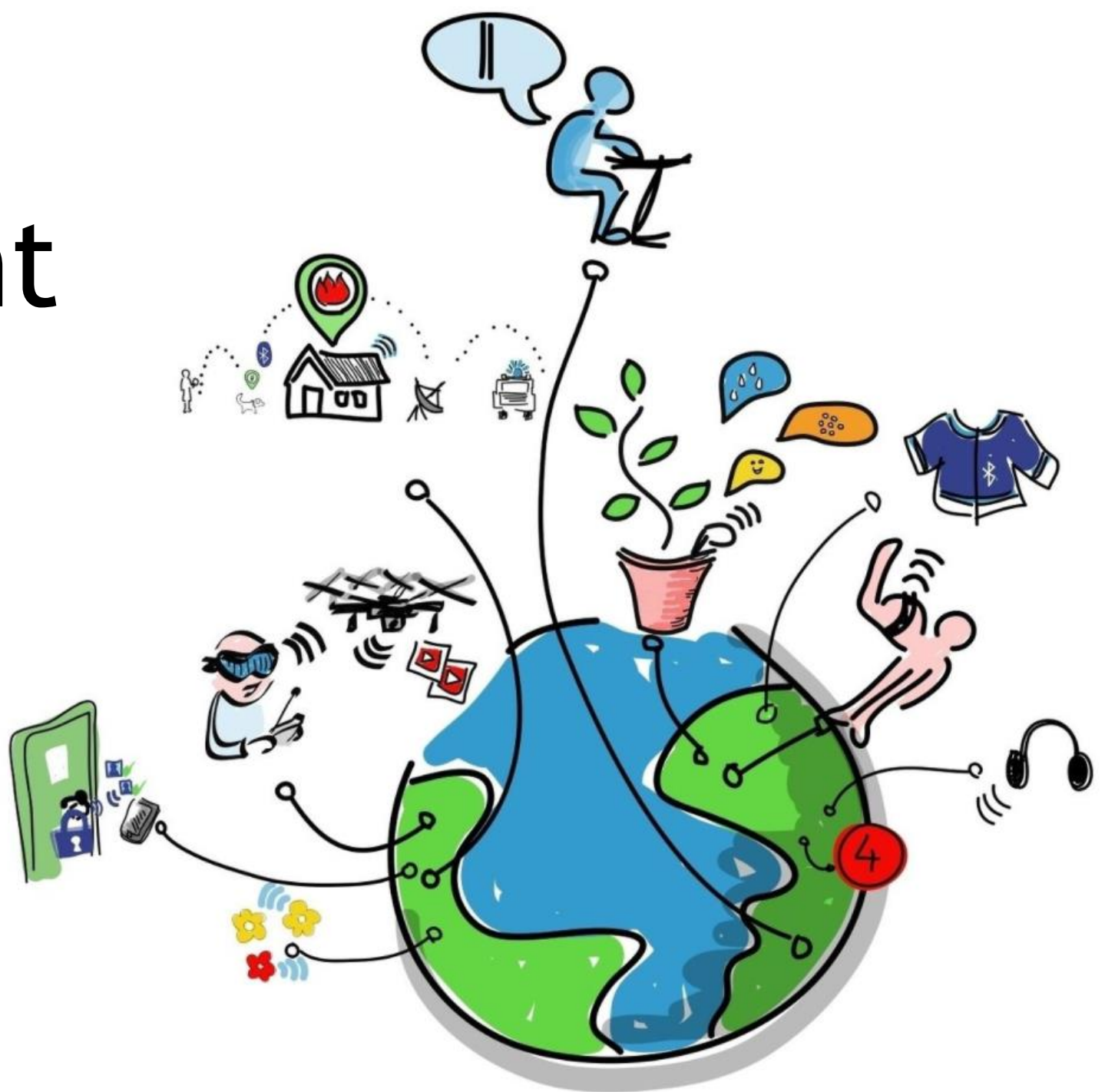


# Program development Cycle

- **Concept of Program Development Cycle**
- **Apply the concept on our daily lives**



Process to develop various sets of instruction is known as programming.



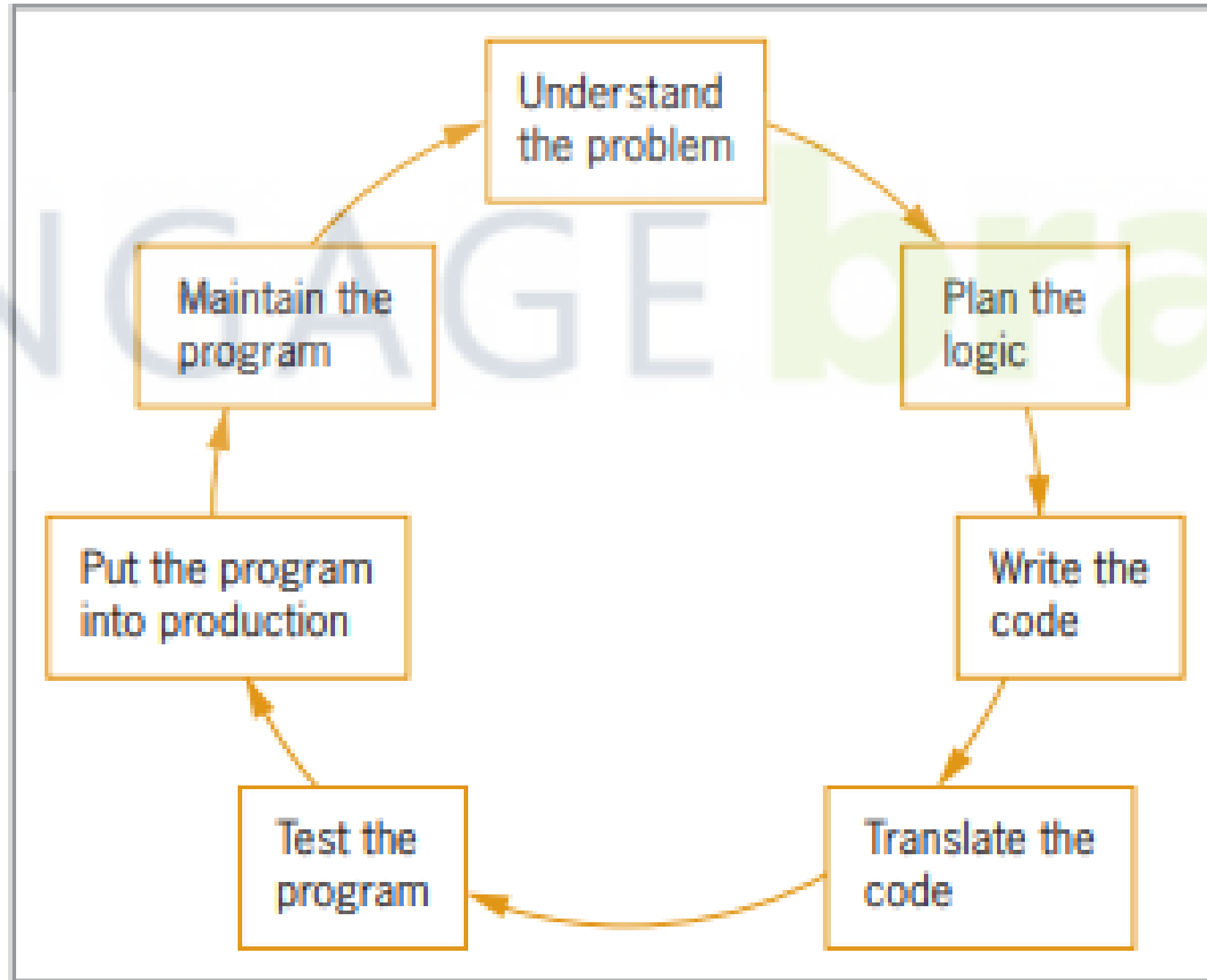
Various sets of instruction is known as program.



# *Program Development Cycle*

# Program Development Cycle

1. Understand the problem
2. Plan the logic
3. Code the program
4. Use software (a compiler or interpreter) to translate the program into machine language
5. Test the program
6. Put the program into production
7. Maintain the program





Professional  
computer  
programmers  
write programs to  
satisfy the needs  
of others, called  
users or end  
users.



# USER or END USER

The term end user distinguishes those who actually use and benefit from a software product from others in an organization who might purchase, install, or have other contact with the software.



*Understand the  
problem*

# *Understand the problem*

Because programmers are providing a service to these users, programmers must first understand what the users want. Although when a program runs, you usually think of the logic as a cycle of input-processing-output operations; when you plan a program, you think of the output first. After you understand what the desired result is, you can plan what to input and process to achieve it.

# *Understand the problem*

Example:

Suppose the director of Human Resources says to a programmer, “Our department needs a list of all employees who have been here over five years, because we want to invite them to a special thank-you dinner.” On the surface, this seems like a simple request. An experienced programmer, however, will know that the request is incomplete.

# *Understand the problem*

Example:

For example, you might not know the answers to the following questions about which employees to include:

- Does the director want a list of full-time employees only, or a list of full- and part-time employees together?

# *Understand the problem*

Example:

- Does she want people who have worked for the company on a month-to-month contractual basis over the past five years, or only regular, permanent employees?
- Do the listed employees need to have worked for the organization for five years as of today, as of the date of the dinner, or as of some other cutoff date?

# *Understand the problem*

Example:

- What about an employee who, for example, worked three years, took a two-year leave of absence, and has been back for three years?

The programmer cannot make any of these decisions; the user (in this case, the Human Resources director) must address these questions.



# *Understand the problem*

Example:

More decisions still might be required. For example:

- What data should be included for each listed employee? Should the list contain both first and last names? Social Security numbers? Phone numbers? Addresses?
- Should the list be in alphabetical order? Employee ID number order? Length-of-service order? Some other order?

# *Understand the problem*

Example:

- Should the employees be grouped by any criteria, such as department number or years of service?

Several pieces of documentation are often provided to help the programmer understand the problem.

# Documentation

- consists of all the supporting paperwork for a program; it might include items such as original requests for the program from users, sample output, and descriptions of the data items available for input.

**NOTE:** Really understanding the problem may be one of the most difficult aspects of programming. On any job, the description of what the user needs may be vague—worse yet, users may not really know what they want, and users who think they know frequently change their minds after seeing sample output.

*A good programmer is often part counselor, part detective!*

You may hear  
programmers  
refer to  
planning a  
program as  
“developing an  
algorithm.”



# *Algorithm*

The sequence of steps necessary to solve any problem.



# *Planning the Logic*

# *Planning the Logic*

The heart of the programming process lies in planning the program's logic. During this phase of the process, the programmer plans the steps of the program, deciding what steps to include and how to order them.

You can plan the solution to a problem in many ways.

# Planning the Logic

The two most common planning tools are **flowcharts** and **pseudocode**.

Both tools involve writing the steps of the program in English, much as you would plan a trip on paper before getting into the car or plan a party theme before shopping for food and favors.

# Planning the Logic

The programmer shouldn't worry about the syntax of any particular language at this point, but should focus on figuring out what sequence of events will lead from the available input to the desired output.

Planning the logic includes thinking carefully about all the possible data values a program might encounter and how you want the program to handle each scenario.

# *desk-checking*

The process of walking through a program's logic on paper before you actually write the program.

# IPO *chart*

which delineates input, processing, and output tasks



# TOE *charts*

which list tasks, objects, and events.

After the logic is developed, only then can the programmer write the program. Hundreds of programming languages are available.



# *Coding the Program*

# *Coding the Program*

Programmers choose particular languages because some have built-in capabilities that make them more efficient than others at handling certain types of operations.

# Coding the Program

Despite their differences, programming languages are quite alike in their basic capabilities—each can handle:

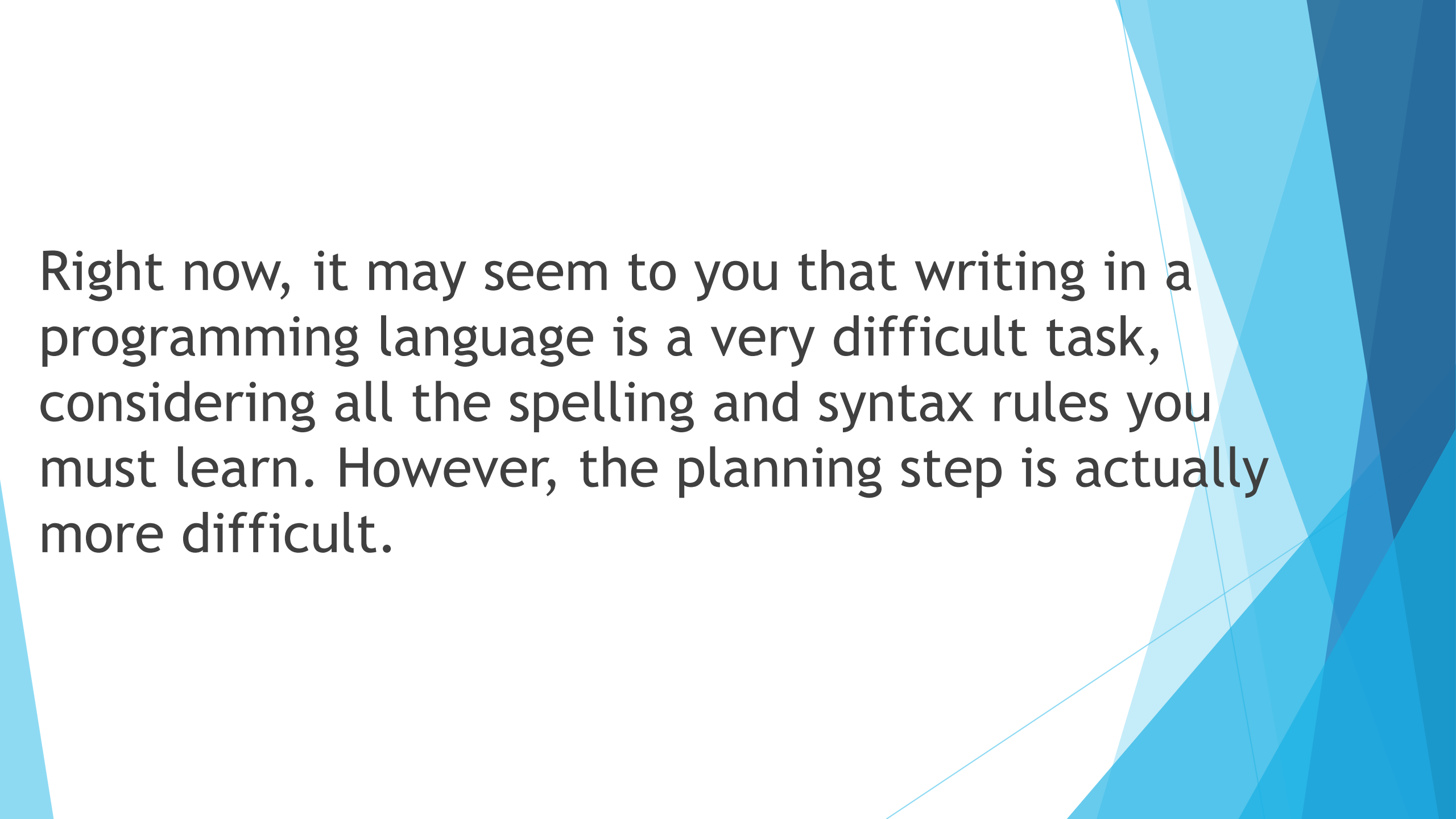
- input operations
- arithmetic processing
- output operations
- and other standard functions

# Coding the Program

The logic developed to solve a programming problem can be executed using any number of languages. Only after choosing a language must the programmer be concerned with proper punctuation and the correct spelling of commands—in other words, using the correct ***syntax***.



*Which step is harder:  
planning the logic or  
coding the program?*

The background of the slide features an abstract design composed of various overlapping triangles and polygons in different shades of blue, ranging from light sky blue to deep navy blue. These shapes are primarily located on the right side of the slide, creating a modern, geometric aesthetic.

Right now, it may seem to you that writing in a programming language is a very difficult task, considering all the spelling and syntax rules you must learn. However, the planning step is actually more difficult.

Which is more difficult: thinking up the twists and turns to the plot of a best-selling mystery novel, or writing a translation of an existing novel from English to Spanish?

Even though there are many programming languages, each computer knows only one language: its machine language, which consists of 1s and 0s.



Computers understand machine language because they are made up of thousands of tiny electrical switches, each of which can be set in either the on or off state, which is represented by a 1 or 0, respectively.



Using Software to  
Translate the Program  
into Machine Language

# Using Software to Translate the Program into Machine Language

Languages like Java or Visual Basic are available for programmers because someone has written a translator program (a compiler or interpreter) that changes the programmer's English-like **high-level programming language** into the **low-level machine language** that the computer understands.



# Using Software to Translate the Program into Machine Language

If you write a programming language statement incorrectly (for example, by misspelling a word, using a word that doesn't exist in the language, or using “illegal” grammar), the translator program doesn't know how to proceed and issues an error message identifying a **syntax error**, which is a misuse of a language's grammar rules.



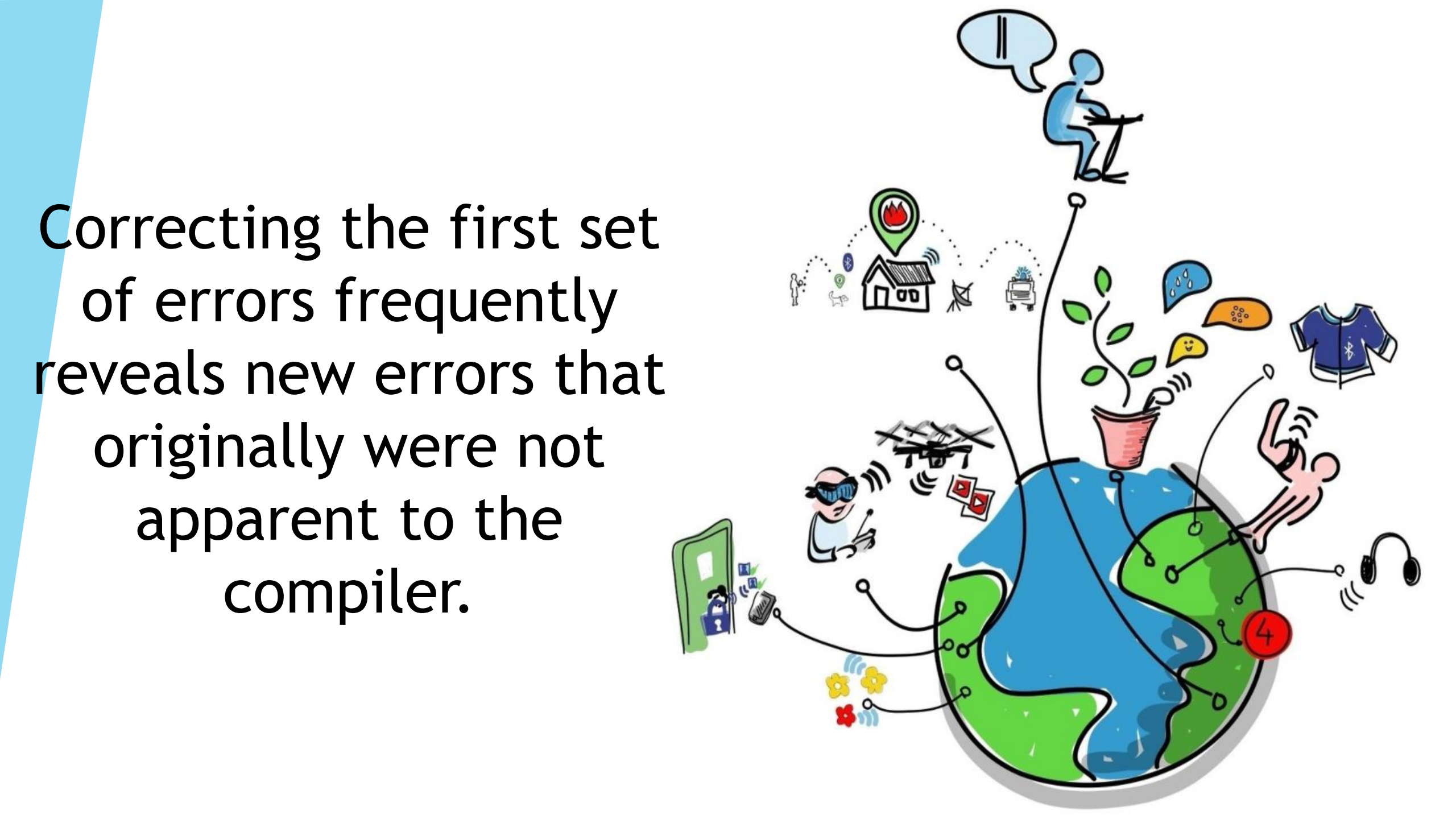
# Using Software to Translate the Program into Machine Language

Although making errors is never desirable, syntax errors are not a major concern to programmers, because the compiler or interpreter catches every syntax error and displays a message that notifies you of the problem. The computer will not execute a program that contains even one syntax error.

*Typically, a programmer :*

- *develops* a program's logic
- writes the code
- compiles the program
- receiving a list of syntax errors
- corrects the syntax errors
- compiles the program again

Correcting the first set of errors frequently reveals new errors that originally were not apparent to the compiler.



For example, if you could use an English compiler and submit the sentence

“The dg chase the cat,”

the compiler at first might point out only one syntax error.

The second word, “dg,” is illegal because it is not part of the English language.

Only after you corrected the word to “dog” would the compiler find another syntax error on the third word, “chase,” because it is the wrong verb form for the subject “dog.”

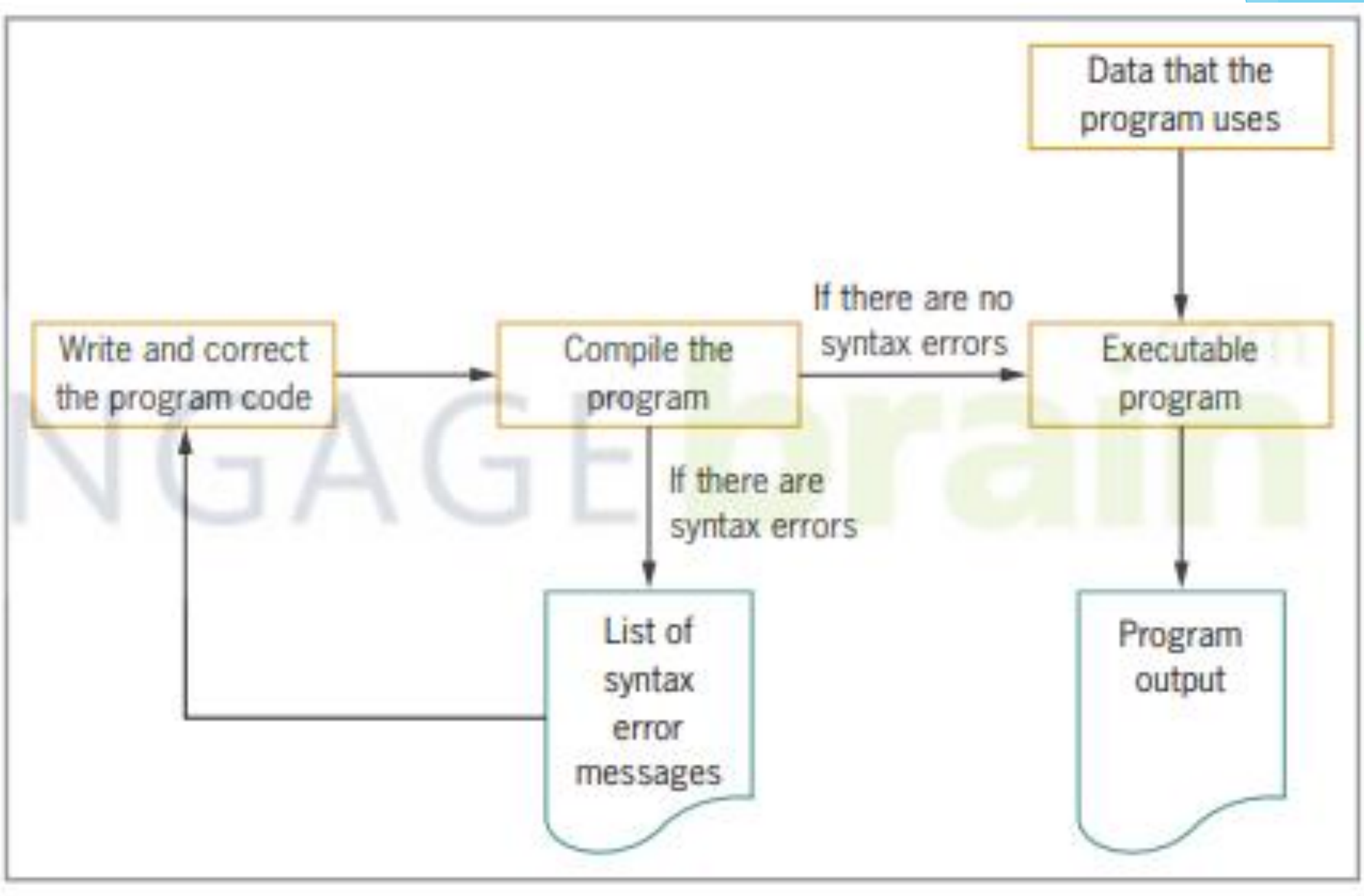
This doesn't mean “chase” is necessarily the wrong word. Maybe “dog” is wrong; perhaps the subject should be “dogs,” in which case “chase” is right.

Compilers  
don't always know  
exactly what you mean,  
nor do they know what  
the  
proper correction should  
be, but they do know  
when something is  
wrong with your syntax.











A program that is free of syntax errors is not necessarily free of logical errors. A **logical error** results when you use a syntactically correct statement but use the wrong one for the current context.



# Testing the Program

# Testing the Program

For example, the English sentence “The dog chases the cat,” although syntactically perfect, is not logically correct if the dog chases a ball or the cat is the aggressor.

Once a program is free of syntax errors, the programmer can test it—that is, execute it with some sample data to see whether the results are logically correct.



# Testing the Program

Recall the number-doubling program:  
input myNumber  
set myAnswer = myNumber \* 2  
output myAnswer

If you execute the program, provide the value 2 as input to the program, and the answer 4 is displayed, you have executed one successful test run of the program.

# Testing the Program

However, if the answer 40 is displayed, maybe the program contains a logical error. Maybe the second line of code was mistyped with an extra zero, so that the program reads:

```
input myNumber  
set myAnswer = myNumber * 20  
output myAnswer
```

***Don't Do It***

The programmer typed "20" instead of "2".

# Testing the Program

Placing 20 instead of 2 in the multiplication statement caused a logical error. Notice that nothing is syntactically wrong with this second program—it is just as reasonable to multiply a number by 20 as by 2—but if the programmer intends only to **double myNumber**, then a logical error has occurred.

# *Debugging*

The process of finding and correcting  
program errors



# Testing the Program

Programs should be tested with many sets of data. For example, if you write the program to double a number, then enter 2 and get an output value of 4, that doesn't necessarily mean you have a correct program.

# Testing the Program

Perhaps you have typed this program by mistake:

```
input myNumber  
set myAnswer = myNumber + 2  
output myAnswer
```

**Don't Do It**

The programmer typed  
"+" instead of "\*".

# Testing the Program

An input of 2 results in an answer of 4, but that doesn't mean your program doubles numbers—it actually only adds 2 to them. If you test your program with additional data and get the wrong answer—for example, if you enter 7 and get an answer of 9—you know there is a problem with your code.

Se  
son  
it  
b



# *Putting the Program into Production*

Once the program is tested adequately, it is ready for the organization to use. Putting the program into production might mean simply running the program once, if it was written to satisfy a user's request for a special list.



# Putting the Program into Production

However, the process might take months if the program will be run on a regular basis, or if it is one of a large system of programs being developed. Perhaps data-entry people must be trained to prepare the input for the new program; users must be trained to understand the output; or existing data in the company must be changed to an entirely new format to accommodate this program.

# Conversion

The entire set of actions an organization must take to switch over to using a new program or set of programs, can sometimes take months or years to accomplish.



# *Maintaining the Program*

# *Maintaining the Program*

After programs are put into production, making necessary changes is called **maintenance**.

**Maintenance** can be required for many reasons:

- new tax rates are legislated
- the format of an input file is altered
- the end user requires additional information not included in the original output specifications

When you maintain the programs others have written, you will appreciate the effort the original programmer put into writing clear code, using reasonable variable names, and documenting his or her work.



# *Maintaining the Program*

When you make changes to existing programs, you repeat the development cycle. That is, you must:

- understand the changes
- then plan
- Code
- Translate
- test them before putting them into production

# *Maintaining the Program*

If a substantial number of program changes are required, the original program might be retired, and the program development cycle might be started for a new program.

Any questions???

# Applications???

## REFERENCES:

- Farrell, J. (2011). *Programming Logic and Design Comprehensive. Sixth Edition.*  
[https://drive.uqu.edu.sa/\\_/fbshareef/files/farrell23936\\_1111823936\\_02\\_01\\_chapter01.pdf](https://drive.uqu.edu.sa/_/fbshareef/files/farrell23936_1111823936_02_01_chapter01.pdf)
- *Programming Logic and Design Comprehensive. Sixth Edition.*  
[https://websites.delta.edu/donaldsouthwell/cst170/ch01\\_ppt.pdf](https://websites.delta.edu/donaldsouthwell/cst170/ch01_ppt.pdf)
- Computer Programming. (n.d.).  
<https://homepage.cs.uri.edu/faculty/wolfe/book/Readings/Reading13.htm>