There are many programming languages (High Level) that one can choose from but the computer can understand only the language of 1’s and 0’s (Low level language) so therefore we need something that converts the high level programming language to make the computer understand these languages.

Who does this work? A compiler!

Shape

Description automatically generated with medium confidenceIn simplest terms, a compiler is a program that converts a source language into a target language. For now, we can think of a compiler as a magic black box which takes in a program written by us (the programmer) and translates it into a form which can be understood by the computer.

Not all text files can be converted into an equivalent machine code. The source program must conform to all the specifications of the language in order to be transformed into an equivalent machine code.

The compiler is designed into two parts. The first phase is the **analysis** phase while the second phase is called **synthesis.**

The main objective of the **analysis phase** is to break the source code into parts. It then arranges these pieces into a meaningful structure (or grammar of the language). **The analysis phase takes in** Lexical analysis, syntax analysis and semantic analysis constitute the **analysis** phase

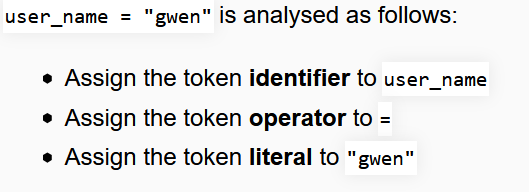
**Lexical Analysis:**

Stream of characters in the source program is grouped into meaningful sequences called lexemes. Tokens are produced for each lexeme. A token is an abstract symbol generated during lexical analysis.

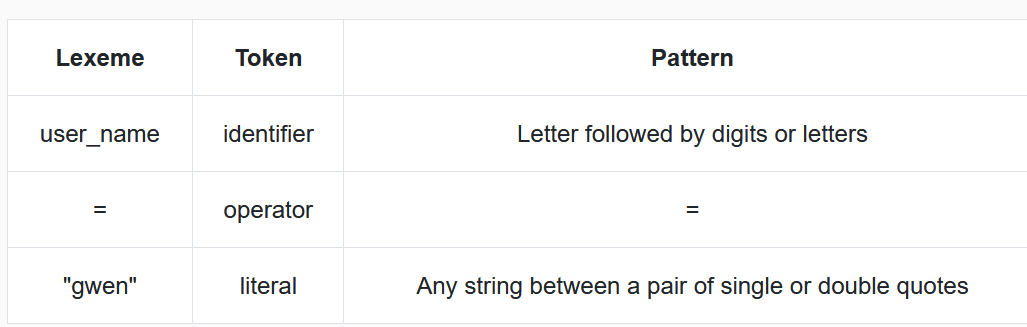
Lexical analysis is the first stage of the compilation process, where the source code created by the programmer is **tokenised** for translation into executable code.

The first part of the lexical analysis operation is removal of any non-program elements. The source code is a text document that contains a lot of characters, which is why it's good programming practice to indent code blocks and use white space to improve a program’s readability. Moreover, comments may be used to help explain complex parts of the code. These constructs are of benefit to the human reader but are not necessary for executable code, so the compiler removes them during lexical analysis.

Next, the characters are read and each string is analysed. A line of code such as



The process can be represented diagrammatically as follows (the individual items are referred to as **lexemes**):



dentifiers are checked against sets of rules, e.g. they may not be allowed to start with a number or contain certain characters. Reserved words (e.g. print) can only be used as keyword tokens.

**Syntax Analysis:**

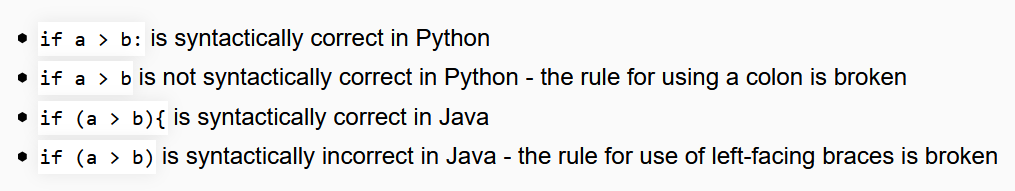
The syntax analyzer checks each line of the code and spots every tiny mistake that the programmer has committed while typing the code. In fact, it even suggests possible ways to rectify our errors. Ever wondered how that happens?  Well, of course it’s not magic! The compiler follows a detailed procedure using the tokens creates by the lexical analyzer and creates a tree-like structure called the syntax tree. However, at this point it is sufficient to understand exactly what type or errors are detected during syntax analysis.

The syntax analyzer checks whether the order of tokens conform to the rules of the programming language. Unmatched parenthesis, missing semicolons are some of the errors detected in this phase.

his is alternatively known as **parsing**. This stage analyses the syntax of the statements to ensure they conform to the rules of grammar for the computer language in question.  
It is roughly the equivalent of checking that some ordinary text written in a natural language (e.g. English) is grammatically correct (without worrying about meaning).  
The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens. Tokens are a valid sequence of symbols, keywords, identifiers etc.

Syntax analysis is the compilation stage immediately following lexical analysis. Once tokens have been assigned to the code elements, the compiler checks that the tokens are in the correct order and follow the rules of the language. For example, in Python the command print(user\_name) is syntactically correct, in that it follows the rules for a print statement: print in lowercase immediately followed by a bracket followed by an identifier and closed by a righthand bracket. In abstraction, this is no different from natural languages such as English; the sentence “he Moved wearily” fails on 3 syntax points: the sentence does not start with a capital letter; a capital letter is used incorrectly in the second word; there is no full stop to end the sentence. The syntax rules for programming languages are finite but must be followed.

Syntax rules differ between languages:



During this stage, an **abstract syntax tree (AST)** is created. This maps the structure of the program, first dropping the brackets, semicolons, etc, that were used by the programmer. If required tokens are missing from the tree, or in the wrong place, the compiler will report an error.

As an example, the AST to compare 2 variables would be generated as follows:

**Semantic Analysis**

Semantic” by definition is concerned with meanings. A semantic analyser is mainly concerned with what the program means and how it executes. Type checking is an important aspect of semantic analysis where each operator should be compatible with its operands.

Code generation

This stage follows the stages of lexical and syntax analysis. A separate program is created that is distinct from the original source code. The code generated is the object code, which is the binary equivalent of the source code. This is the executable version of the code, before linked libraries are included.

Code generation is a major distinguishing feature between compilation and interpretation; interpreters do not produce a separate executable file.

Code optimization

Code optimisation is carried out throughout the compilation process and in particular as part of the code generation stage. The optimiser may identify redundant or repeated code, and remove or rearrange the code as necessary. Examples are removing procedures that are never called, or moving an assignment statement that had incorrectly been placed inside a loop, potentially causing it to be inefficiently executed multiple times.

**Symbol Table**

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

The symbol table  
There are a few activities that interact with various phases across both stages. One is symbol table management; a symbol table contains information about all the identifiers in the program along with important attributes such as type and scope. Identifiers can be found in the lexical analysis phase and added to the symbol table. During the two   
phases that follow (syntax and semantic analysis), the compiler updates the identifier entry in the table to include information about its type and scope. When generating intermediate code, the type of the variable is used to determine which instructions to   
emit. During optimization, the "live range" of each variable may be placed in the table to aid in register allocation. The memory location determined in the code generation phase might also be kept in the symbol table.

Error handling  
Another activity that occurs across several phases is error handling. Most error handling occurs in the first three phases of the analysis stage. The scanner keeps an eye out for stray tokens, the syntax analysis phase reports invalid combinations of tokens, and the   
semantic analysis phase reports type errors and the like. Sometimes these are fatal errors that stop the entire process, while others are less serious and can be circumvented so the compiler can continue