Element In The Programming Language

All programming languages have some primitive building blocks for the description of data and the processes or transformations applied to them (like the addition of two numbers or the selection of an item from a collection). These primitives are defined by syntactic and semantic rules which describe their structure and meaning respectively.

SYNTAX
A programming language's surface form is known as its syntax. Most programming languages are purely textual; they use sequences of text including words, numbers, and punctuation, much like written natural languages. On the other hand, there are some programming languages which are more graphical in nature, using visual relationships between symbols to specify a program.

The syntax of a language describes the possible combinations of symbols that form a syntactically correct program. The meaning given to a combination of symbols is handled by semantics (either formal or hard-coded in a reference implementation). Since most languages are textual, this article discusses textual syntax.

Programming language syntax is usually defined using a combination of regular expressions (for lexical structure) and Backus–Naur form (for grammatical structure).

Not all syntactically correct programs are semantically correct. Many syntactically correct programs are nonetheless ill-formed, per the language's rules; and may (depending on the language specification and the soundness of the implementation) result in an error on translation or execution. In some cases, such programs may exhibit undefined behavior. Even when a program is well-defined within a language, it may still have a meaning that is not intended by the person who wrote it.

SEMANTICS
- Static semantics
The static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms.[2] For compiled languages, static semantics essentially include those semantic rules that can be checked at compile time. Examples include checking that every identifier is declared before it is used (in languages that require such declarations) or that the labels on the arms of a case statement are distinct.[45] Many important restrictions of this type, like checking that identifiers are used in the appropriate context (e.g. not adding an integer to a function name), or that subroutine calls have the appropriate number and type of arguments, can be enforced by defining them as rules in a logic called a type system. Other forms of static analyses like data flow analysis may also be part of static semantics. Newer programming languages like Java and C# have definite assignment analysis, a form of data flow analysis, as part of their.

- Dynamic semantics
Once data has been specified, the machine must be instructed to perform operations on the data. For example, the semantics may define the strategy by which expressions are evaluated to values, or the manner in which control structures conditionally execute

statements. The dynamic semantics (also known as execution semantics) of a language defines how and when the various constructs of a language should produce a program behavior. There are many ways of defining execution semantics. Natural language is often used to specify the execution semantics of languages commonly used in practice. A significant amount of academic research went into formal semantics of programming languages, which allow execution semantics to be specified in a formal manner. Results from this field of research have seen limited application to programming language design and implementation outside academia.

WEAK AND STRONG TYPING

Weak typing allows a value of one type to be treated as another, for example treating a string as a number.[46] This can occasionally be useful, but it can also allow some kinds of program faults to go undetected at compile time and even at run time.

Strong typing prevents these program faults. An attempt to perform an operation on the wrong type of value raises an error.[46] Strongly typed languages are often termed type-safe or safe.

An alternative definition for "weakly typed" refers to languages, such as Perl and JavaScript, which permit a large number of implicit type conversions. In JavaScript, for example, the expression 2 * x implicitly converts x to a number, and this conversion succeeds even if x is null, undefined, an Array, or a string of letters. Such implicit conversions are often useful, but they can mask programming errors. Strong and static are now generally considered orthogonal concepts, but usage in the literature differs. Some use the term strongly typed to mean strongly, statically typed, or, even more confusingly, to mean simply statically typed. Thus C has been called both strongly typed and weakly, statically typed.

It may seem odd to some professional programmers that C could be "weakly, statically typed". However, notice that the use of the generic pointer, the void* pointer, does allow for casting of pointers to other pointers without needing to do an explicit cast