# UNIVERSITÀ DELLA SVIZZERA ITALIANA

# FACULTY OF COMPUTER SCIENCE

# Master in Artificial Intelligence

# Topic: Programming language

By Heider Jeffer

Instructor: Prof. Mehdi Jazayeri

14 December - 20 December 2014

# Summary

A programming language is an artificial language designed to express computations that can be performed by a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine, to express algorithms precisely, or as a mode of human communication. Many programming languages have some form of written specification of their syntax (form) and semantics (meaning). Some languages are defined by a specification document. For example, the C programming language is specified by an ISO Standard. Other languages, such as Perl, have a dominant implementation that is used as a reference.

The earliest programming languages predate the invention of the computer and were used to direct the behavior of machines such as Jacquard looms and player pianos. Thousands of different programming languages have been created, mainly in the computer field, with many more being created every year. Most programming languages describe computation in an imperative style, i.e., as a sequence of commands, although some languages, such as those that support functional programming or logic programming, use alternative forms of description. A programming language is a notation for writing programs, which are specifications of a computation or algorithm. Some, but not all, authors restrict the term ”programming language” to those languages that can express all possible algorithms.

# Function and target

A computer programming language is a language used to write computer programs, which involve a computer performing some kind of computation or algorithm and possibly controlling external devices such as printers, disk drives, robots, and so on. For example, PostScript programs are frequently created by another program to control a computer printer or display. More generally, a pro-programming language may describe computation on some, possibly abstract, machine. It is generally accepted that a complete specification for a programming language includes a description, possibly idealized, of a machine or processor for that language. In most practical contexts, a programming language involves a computer; consequently, programming languages are usually defined and studied this way. Programming languages differ from natural languages in that natural languages are only used for interaction between people, while programming languages also allow humans to communicate instructions to machines.

# Abstractions

Programming languages usually contain abstractions for defining and manipulating data structures or controlling the flow of execution. The practical necessity that a programming language supports adequate abstractions is expressed by the abstraction principle; this principle is sometimes formulated as a recommendation to the programmer to make proper use of such abstractions.

# Expressive power

The theory of computation classifies languages by the computations they are capable of expressing. All Turing complete languages can implement the same set of algorithms. ANSI/ISO SQL and Charity are examples of languages that are not Turing complete, yet often called programming languages.

# Elements

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, some programming languages are more graphical, 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.

# Static Semantics

Static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms. 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.

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 static semantics.

# Type system

A type system defines how a programming language classifies values and expressions into types, how it can manipulate those types, and how they interact. The goal of a type system is to verify and usually enforce a certain level of correctness in programs written in that language by detecting certain incorrect operations. Any decidable type of system involves a trade-off: while it rejects many incorrect programs, it can also prohibit some correct, albeit unusual programs. To bypass this downside, several languages have type loopholes, usually unchecked casts that may be used by the programmer to explicitly allow a normally disallowed operation between different types. In most typed languages, the type system is used only to type-check programs, but several languages, usually functional ones, perform type inference, which relieves the programmer from writing type annotations. The formal design and study of type systems are known as type theory.

# Typed versus untyped languages

A language is typed if the specification of every operation defines the types of data to which the operation is applicable, with the implication that it does not apply to other types. For example, ”this text between the quotes” is a string. In most programming languages, dividing a number by a string has no meaning. Most modern programming languages will therefore reject any program attempting to perform such an operation. In some languages, the meaningless operation will be detected when the program is compiled (”static” type checking), and rejected by the compiler, while in others, it will be detected when the program is run (”dynamic” type checking), resulting in a runtime exception.