Name : أحمد عصام سعيد حسانين

ID : 104

Group : 1

Section : 5

* The programming language

1. Home

 <div id="home">

        <p>A programming language is a formal language comprising a set of strings that produce various kinds of machine

            code output.

            Programming languages are one kind of computer language, and are used in computer programming to implement

            algorithms.

            Most programming languages consist of instructions for computers. There are programmable machines that use a

            set of specific instructions, rather than general programming languages.

            The description of a programming language is usually split into the two components of 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) while other languages (such as Perl) have a dominant

            implementation that is treated as a reference. Some languages have both, with the basic language defined by

            a standard and extensions taken from the dominant implementation being common.

        </p>

    </div>

1. Classification

<div id="classification">

        <table>

            <caption>classification of programming language</caption>

            <tr>

                <th scope="col"> </th>

                <th>statically checked</th>

                <th>dynamically checked</th>

            </tr>

            <tr>

                <th scope="row">strong type</th>

                <td> Hskel, java, pascl </td>

                <td> Lips, Sceme </td>

            </tr>

            <tr>

                <th scope="row"> weakly type </th>

                <td>C</td>

                <td>c++ </td>

                <td>perl</td>

            </tr>

        </table>

    </div>

1. Semantics

 <div class="Semantics">

        <p>

            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 static 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.

            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 system involves a trade-off: while it rejects many incorrect programs, it can also

            prohibit some correct, albeit unusual programs. In order to bypass this downside, a number of 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 a number of languages, usually functional ones, infer types, relieving the programmer

            from the need to write type annotations. The formal design and study of type systems is known as type

            theory.

        </p>

        <!-- img  -->

        <img src="https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcT6kd2g8aY1vnkktn9CdSQIJSwEK\_TeRmZ6hA&usqp=CAU">

    </div>

1. Design

 <div id="Design ">

        <p>

            Programming languages share properties with natural languages related to their purpose as vehicles for

            communication, having a syntactic form separate from its semantics, and showing language families of related

            languages branching one from another.[50][51] But as artificial constructs, they also differ in fundamental

            ways from languages that have evolved through usage. A significant difference is that a programming language

            can be fully described and studied in its entirety since it has a precise and finite definition.[52] By

            contrast, natural languages have changing meanings given by their users in different communities. While

            constructed languages are also artificial languages designed from the ground up with a specific purpose,

            they lack the precise and complete semantic definition that a programming language has.

            Many programming languages have been designed from scratch, altered to meet new needs, and combined with

            other languages. Many have eventually fallen into disuse. Although there have been attempts to design one

            "universal" programming language that serves all purposes, all of them have failed to be generally accepted

            as filling this role.[53] The need for diverse programming languages arises from the diversity of contexts

            in which languages are used:

            Programs range from tiny scripts written by individual hobbyists to huge systems written by hundreds of

            programmers.

            Programmers range in expertise from novices who need simplicity above all else to experts who may be

            comfortable with considerable complexity.

            Programs must balance speed, size, and simplicity on systems ranging from microcontrollers to

            supercomputers.

            Programs may be written once and not change for generations, or they may undergo continual modification.

            Programmers may simply differ in their tastes: they may be accustomed to discussing problems and expressing

            them in a particular language.

            One common trend in the development of programming languages has been to add more ability to solve problems

            using a higher level of abstraction. The earliest programming languages were tied very closely to the

            underlying hardware of the computer. As new programming languages have developed, features have been added

            that let programmers express ideas that are more remote from simple translation into underlying hardware

            instructions. Because programmers are less tied to the complexity of the computer, their programs can do

            more computing with less effort from the programmer. This lets them write more functionality per time

            unit.[54]

            Natural language programming has been proposed as a way to eliminate the need for a specialized language for

            programming. However, this goal remains distant and its benefits are open to debate. Edsger W. Dijkstra took

            the position that the use of a formal language is essential to prevent the introduction of meaningless

            constructs, and dismissed natural language programming as "foolish".[55] Alan Perlis was similarly

            dismissive of the idea.[56] Hybrid approaches have been taken in Structured English and SQL.

            A language's designers and users must construct a number of artifacts that govern and enable the practice of

            programming. The most important of these artifacts are the language specification and implementation.

        </p>

    </div>

1. specification

  <div id="Specification">

        <p>

            Main article: Programming language specification

            The specification of a programming language is an artifact that the language users and the implementors can

            use to agree upon whether a piece of source code is a valid program in that language, and if so what its

            behavior shall be.

            A programming language specification can take several forms, including the following:

            An explicit definition of the syntax, static semantics, and execution semantics of the language. While

            syntax is commonly specified using a formal grammar, semantic definitions may be written in natural language

            (e.g., as in the C language), or a formal semantics (e.g., as in Standard ML[57] and Scheme[58]

            specifications).

            A description of the behavior of a translator for the language (e.g., the C++ and Fortran specifications).

            The syntax and semantics of the language have to be inferred from this description, which may be written in

            natural or a formal language.

            A reference or model implementation, sometimes written in the language being specified (e.g., Prolog or ANSI

            REXX[59]). The syntax and semantics of the language are explicit in the behavior of the reference

            implementation.

        </p>

    </div>

* Document Html

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>programming language</title>

</head>

<body>

    <p class="logo">programming language</p>

    <!-- list of anchor -->

    <ul>

        <li><a href="#home">home</a> </li>

        <li><a href="#classification">classification</a></li>

        <li><a href="#Semantics">Semantics</a></li>

        <li><a href="#Design ">Design </a></li>

        <li><a href="#Specification">Specification</a></li>

    </ul>

    <div id="home">

        <p>

            Programming languages are one kind of computer language, and are used in computer programming to implement

            algorithms.

            Most programming languages consist of instructions for computers. There are programmable machines that use a

            set of specific instructions, rather than general programming languages.

            The description of a programming language is usually split into the two components of 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) while other languages (such as Perl) have a dominant

            implementation that is treated as a reference. Some languages have both, with the basic language defined by

            a standard and extensions taken from the dominant implementation being common.

            The description of a programming language is usually split into the two components of 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) while other languages (such as Perl) have a dominant

            implementation that is treated as a reference. Some languages have both, with the basic language defined by

            a standard and extensions taken from the dominant implementation being common.

        </p>

    </div>

    <!-- table for classification of programming language-->

    <div id="classification">

        <table>

            <caption>classification of programming language</caption>

            <tr>

                <th scope="col"> </th>

                <th>statically checked</th>

                <th>dynamically checked</th>

            </tr>

            <tr>

                <th scope="row">strong type</th>

                <td> Hskel, java, pascl </td>

                <td> Lips, Sceme </td>

            </tr>

            <tr>

                <th scope="row"> weakly type </th>

                <td>C</td>

                <td>c++ </td>

                <td>perl</td>

            </tr>

        </table>

    </div>

    <div class="Semantics">

        <p>

            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 static 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.

            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 system involves a trade-off: while it rejects many incorrect programs, it can also

            prohibit some correct, albeit unusual programs. In order to bypass this downside, a number of 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 a number of languages, usually functional ones, infer types, relieving the programmer

            from the need to write type annotations. The formal design and study of type systems is known as type

            theory.

        </p>

        <!-- img  -->

        <img src="https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcT6kd2g8aY1vnkktn9CdSQIJSwEK\_TeRmZ6hA&usqp=CAU">

    </div>

    <div id="Design ">

        <p>

            Programming languages share properties with natural languages related to their purpose as vehicles for

            communication, having a syntactic form separate from its semantics, and showing language families of related

            languages branching one from another.[50][51] But as artificial constructs, they also differ in fundamental

            ways from languages that have evolved through usage. A significant difference is that a programming language

            can be fully described and studied in its entirety since it has a precise and finite definition.[52] By

            contrast, natural languages have changing meanings given by their users in different communities. While

            constructed languages are also artificial languages designed from the ground up with a specific purpose,

            they lack the precise and complete semantic definition that a programming language has.

            Many programming languages have been designed from scratch, altered to meet new needs, and combined with

            other languages. Many have eventually fallen into disuse. Although there have been attempts to design one

            "universal" programming language that serves all purposes, all of them have failed to be generally accepted

            as filling this role.[53] The need for diverse programming languages arises from the diversity of contexts

            in which languages are used:

            Programs range from tiny scripts written by individual hobbyists to huge systems written by hundreds of

            programmers.

            Programmers range in expertise from novices who need simplicity above all else to experts who may be

            comfortable with considerable complexity.

            Programs must balance speed, size, and simplicity on systems ranging from microcontrollers to

            supercomputers.

            Programs may be written once and not change for generations, or they may undergo continual modification.

            Programmers may simply differ in their tastes: they may be accustomed to discussing problems and expressing

            them in a particular language.

            One common trend in the development of programming languages has been to add more ability to solve problems

            using a higher level of abstraction. The earliest programming languages were tied very closely to the

            underlying hardware of the computer. As new programming languages have developed, features have been added

            that let programmers express ideas that are more remote from simple translation into underlying hardware

            instructions. Because programmers are less tied to the complexity of the computer, their programs can do

            more computing with less effort from the programmer. This lets them write more functionality per time

            unit.[54]

            Natural language programming has been proposed as a way to eliminate the need for a specialized language for

            programming. However, this goal remains distant and its benefits are open to debate. Edsger W. Dijkstra took

            the position that the use of a formal language is essential to prevent the introduction of meaningless

            constructs, and dismissed natural language programming as "foolish".[55] Alan Perlis was similarly

            dismissive of the idea.[56] Hybrid approaches have been taken in Structured English and SQL.

            A language's designers and users must construct a number of artifacts that govern and enable the practice of

            programming. The most important of these artifacts are the language specification and implementation.

        </p>

    </div>

    <div id="Specification">

        <p>

            Main article: Programming language specification

            The specification of a programming language is an artifact that the language users and the implementors can

            use to agree upon whether a piece of source code is a valid program in that language, and if so what its

            behavior shall be.

            A programming language specification can take several forms, including the following:

            An explicit definition of the syntax, static semantics, and execution semantics of the language. While

            syntax is commonly specified using a formal grammar, semantic definitions may be written in natural language

            (e.g., as in the C language), or a formal semantics (e.g., as in Standard ML[57] and Scheme[58]

            specifications).

            A description of the behavior of a translator for the language (e.g., the C++ and Fortran specifications).

            The syntax and semantics of the language have to be inferred from this description, which may be written in

            natural or a formal language.

            A reference or model implementation, sometimes written in the language being specified (e.g., Prolog or ANSI

            REXX[59]). The syntax and semantics of the language are explicit in the behavior of the reference

            implementation.

        </p>

    </div>

</body>

</html>