**INTRODUCTION**

A language is a “conceptual universe” providing a framework for problem-solving and useful concepts and programming methods.

Conceptual universe should have something to help me solve problems like sorting etc.

Özne, yüklem vs. bizim natural dilimizin konseptleridir.

Imperative 🡪 C  
OOP 🡪 C++ / Java  
Functional 🡪 Lisp - Pascal  
Declerative/Logic 🡪 Prolog

**LANGUAGE EVALUATION CRITERIA**

* Readability
  + Most important
  + You must read the code to debug, to change, to learn how someone solved a problem
  + Too many features is bad
  + Multiplicity of features is bad
  + ORTHOGONALITY: We have unique features. I do one thing for one thing. Makes the language easy to learn and read.
    - A language is orthogonal if its features are built upon a small, mutually independent set of primitive operations
    - Fewer exceptional rules = conceptual simplicity
      * E.g., restricting types of arguments to a function
    - Assembly is very orthogonal
* Writability
  + You need simplicity and orthogonality
  + You need support for abstraction
* Simplicity, orthogonality
* High expressive power, flexibility
* Reliability
  + Program behavior is the same on different platforms with same data
  + Type errors are detected
  + Semantic errors are properly trapped
  + Memory leaks are prevented
* Safety
* Cost (influenced by above)
  + Creation
  + Execution
  + Maintenance

A picture containing text

Description automatically generated

Table

Description automatically generated with low confidence

Cost:

* Programmer training (If training a programmer takes long time, then it’s a bad situation)
* Software creation
* Compilation
* Execution
* Compiler cost
* Poor reliability
* Maintenance
* Portability
* Generality
* Well-definedness

**PROGRAMMING PARADIGMS**

Computation: Solving a problem

If logic is wrong (nontermination of a function) or there is undefined operation (like 3/0) computation may not result in expected value.

These two cases are mathematically equivalent (we call these types of functions that may not give us an answer as partial functions, these functions generate result and terminate only for a set of input values), but operationally different.

Total functions: Terminate and give a value.

There is no halting program that decides if a given function/program will halt or not.

Paradigms: Model for helping us understand concepts accepted by most people in a field

It describes an overall concept accepted by most people in an intellectual community, as those in one of the natural sciences, because of its effectiveness in explaining a complex process, idea, or set of data

A scientific paradigm is a framework containing all the commonly accepted views about a subject, conventions about what direction research should take and how it should be performed

In programming we have several paradigms. 2 major ones:

* Imperative (instruct the machine how to change its states)
  + Procedural (C, Fortran)
  + Object-Oriented (SmallTalk)
* Declarative (we give logic of computation but not the control flow, we tell what we want to get)
  + Functional / Applicative (LISP, Pascal, Scheme)
  + Logic (Prolog)
  + Mathematical

Hardware implementation of all computers are imperative. This paradigm can break problems into much smaller form.

In reality, very few languages are “purely” based on one of these paradigms. Most combine features of different paradigms.

For example C++ is mainly OO but it has a lot of procedural concepts as well  
 Most of the functional languages are incorporating some imperative concepts (both OO and procedural)  
 Some Python programs are completely procedural, some are OO, some are functional

Execution model (the way that data and operators are interact with each other) of the language:

* Allowing side effects (there is no change in state of the machine during execution of the function starts and ends)
* Defining sequence of operations

Code organization

* Grouping code into units along with the state modified by the code

Style of syntax and grammar

**IMPERATIVE**

Imperative (procedural) programs consists of actions to effect state change (does it do anything to memory or output devices), principally through assignment operations or side effects

* Fortran, Algol, Cobol, PL/I, Pascal, Modula-2, Ada, C

Imperative programming paradigm dominates the field bc imperative things are very easy to understand for us. Usually we think algorithm in terms of steps and each step does sth that I can track of. For example, give me the height of the rectangle, give me the width of the rectangle and then multiply them and print the result. Reads the data, does sth, puts the data back.

In OOP, we don’t think as give the data and do sth. We will consantrate on processes, not the data itself. Not think passing data to a function and return sth in OOP but think passing a message to an object to understand what my intention is.

FEATURES:

* states the order in which operations occur, with constructs that explicitly control that order,
* allow side effects, in which state can be modified at one point in time, within one unit of code, and then later read at a different point in time inside a different unit of code,
* communication between the units of code is not explicit. Everybody does their parts and interaction achieves what I want to do. In OOP, if I want to do sth, I will ask him as a message saying do this one for me and communication will be explicit.

Functions are implicitly coded at every step necessary to solve a problem in imperative programming, hence pre-coded models aren't used. In contrast to declarative programming, which tells the computer "what" the program should do, imperative programming tells the machine "how" to do it.

**OBJECT-ORIENTED**

Most OO languages have been imperative

* Simula, Smalltalk, C++, Modula-3, Java
* Notable exception: CLOS (Common List Object System)

FEATURES:

* Organizes code into objects containing state which is only modified by the code that is part of the object

**FUNCTIONAL AND LOGIC**

Focuses on function evaluation; avoids updates, assignment, mutable state, side effects

Not all functional languages are “pure”

* In practice, rely on non-pure functions for input/output and some permit assignment-like operators
* E.g., (set! x 1) in Scheme

Logic programming is based on predicate logic

* Targeted at theorem-proving languages, automated reasoning, database applications
* Declarative programming (Express the logic of a computation without describing its control flow)

It is a declarative type of programming style. Its main focus is on “what to solve” in contrast to an imperative style where the main focus is “how to solve”.

**UNIFYING CONCEPTS**

Unifying language concepts

* Types (both built-in and user-defined)
  + Specify constraints on functions and data
  + Static vs. dynamic typing
* Expressions (e.g., arithmetic, boolean, strings)
* Functions/procedures
* Commands

**DESIGN CHOICES**

|  |  |
| --- | --- |
| C | Efficient imperative programming with static types |
| C++ | OOP with static types and ad hoc, subtype and parametric polymorphism |
| Java | Imperative, OO, and concurrent programming with static types and garbage collection |
| Scheme | Lexically scoped, applicative-style recursive programming with dynamic types |
| Standard ML | Practical functional programming with strict (eager) evaluation and polymorphic type inference |
| Haskell | Pure functional programming with non-strict (lazy) evaluation |

Launch of Ariana 5 rocket is failed

* Cause: software error in inertial reference system
* Re-used Ariana 4 code, but flight path was different
* 64-bit floating point number related to horizontal velocity converted to 16-bit signed integer; the number was larger than 32767; inertial guidance crashed

**VON NEUMANN MODEL**

Diagram

Description automatically generated

**Language Translation**

* Native-code compiler: produces machine code, total translation
  + Compiled languages: Fortran, C, C++, SML…
* Interpreter: translates into internal form and immediately executes (read-eval-print loop), internal form can be automatically converted to whatever the target platform is. Interpreter is program itself and it executes at the same time during the translation
  + Interpreted languages: Scheme, Haskell, Python…
* Byte-code compiler: produces portable bytecode, which is executed on virtual machine (e.g., Java)
* Hybrid approaches
  + Source-to-source translation (early C++ to C to compile)
  + Just-in-time Java compilers convert bytecode into native machine code when first executed

Compiler: Program that translates a source language into a target language. Target language is often but not always, the assembly language for a particular machine.

Diagram

Description automatically generated

**Checks During Compilation/Interpretation**

Syntatically invalid constructs

* Invalid type conversions
* A value is used in the “wrong” context, e.g., assigning a float to an int

Static determination of type information is also used to generate more efficient code

* Know what kind of values will be stored in a given memory region during program execution

Some programmer logic errors

* Can be subtle: if (a=b) … instead of if (a==b)

Compilation Process

Compilation: source code 🡪 relocatable object code (binaries)

relocatable: there are some parametrization of the whatever the state of that machine is. You can relocate your binary code to fit in the form of the state that you are currently running into, especially with regards to the memory.

Linking: many relocatable binaries (modules plus libraries) 🡪 one relocatable binary (with all external references satisfied)

Loading: relocatable 🡪 absolute binary (with all code and data references bound to the addresses occupied in memory)

Execution: control is transferred to the first instruction of the program

At compile time, absolute addresses of variables and statement labels are not known

In static languages (such as Fortran), absolute addresses are bound at load time

In block-structured languages, bindings can change at run time

Phases of Compilation

Preprocessing: conditional macro text substitution

Lexical analysis: convert keywords, identifiers, constants into a sequence of tokens

Syntatic analysis: check that token sequence is syntactically correct

Generate abstract syntax trees (AST): check types

Intermediate code generation: “walk” the ASTs and generate intermediate code

* Apply optimizations to produce efficient code

Final code generation: produce machine code

Language Interpretation

Read-eval-print loop

* Read in an expression, translate into internal form
* Evaluate internal form
  + This requires an abstract machine and a “run-time” component (usually a compiled program that runs on the native machine)
* Print the result of evaluation
* Loop back to read the next expression