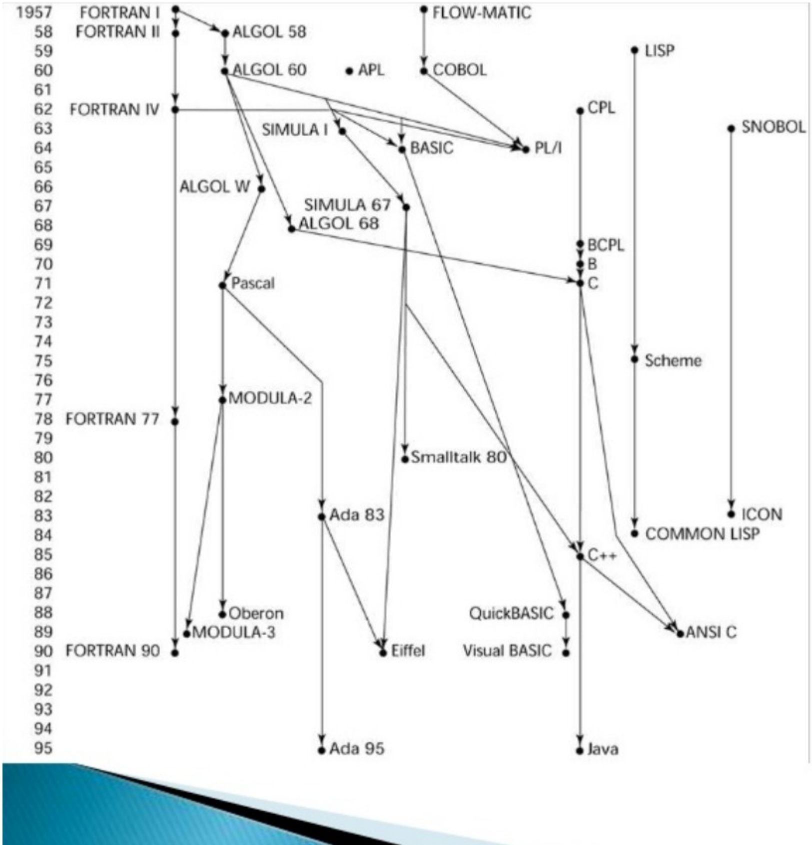
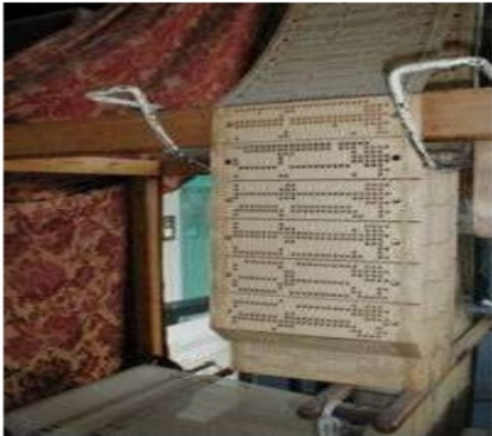
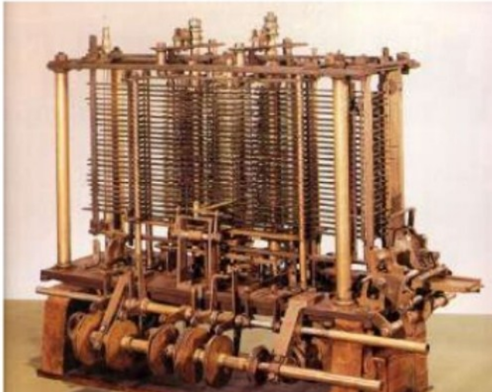
PROGRAMMING LANGUAGES THEORY

İBRAHİM ATLI

Evolution of Programming Languages

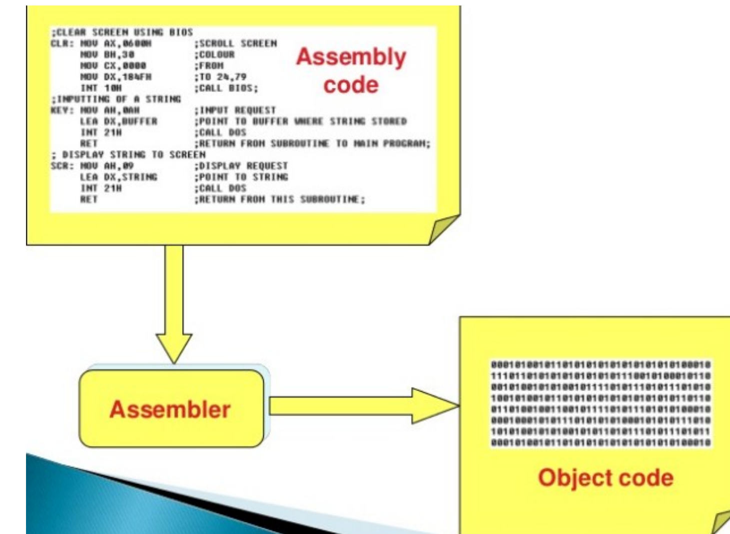




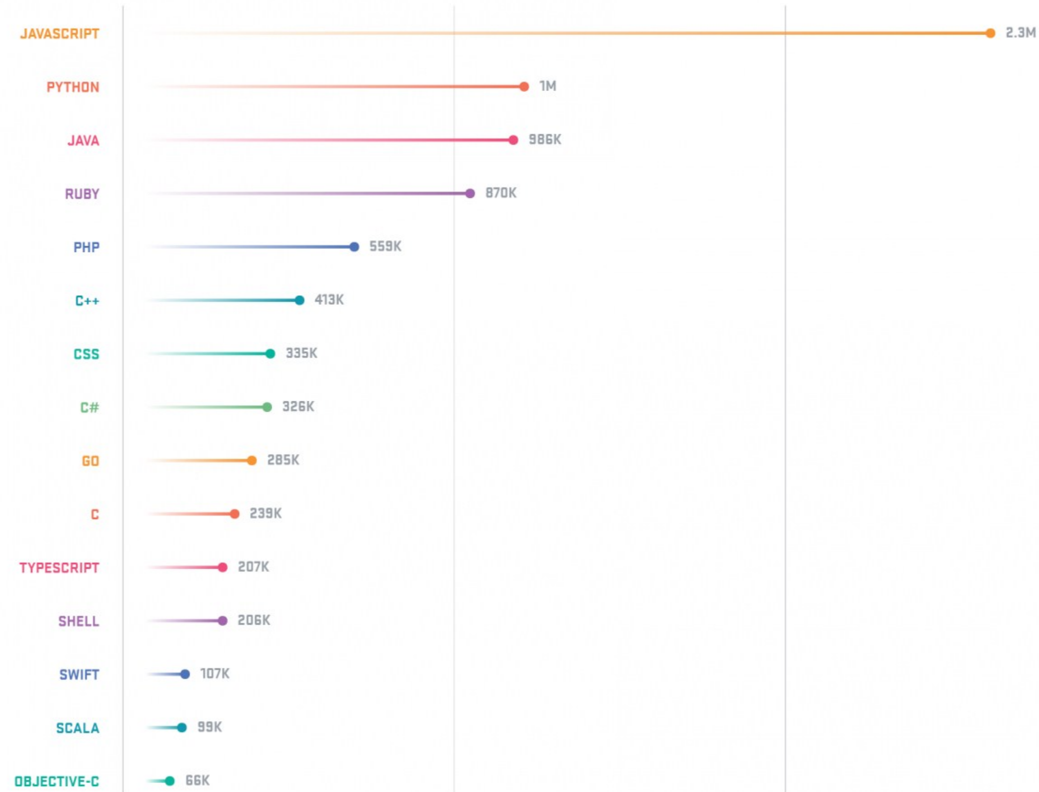


Generation of PL

- Machine Language-First Generations. (1’s and 0’s)
- Assemble Language-Second Generation.
- Procedural Language-Third Generation.
- Problem Oriented-Fourth Language.
- Natural Language-Fifth Language.



Current Trends in PL



http://www.businessinsider.com/the-9-most-popular-programming-languages-according-to-the-facebook-for-programmers-2017-10/#1-javascript-15

Programming Languages vs Natural Languages

- Formal vs Informal
- Strict rules of well-formedness vs Error tolerant
- Restricted vs Unrestricted domain

What makes a language programming language?

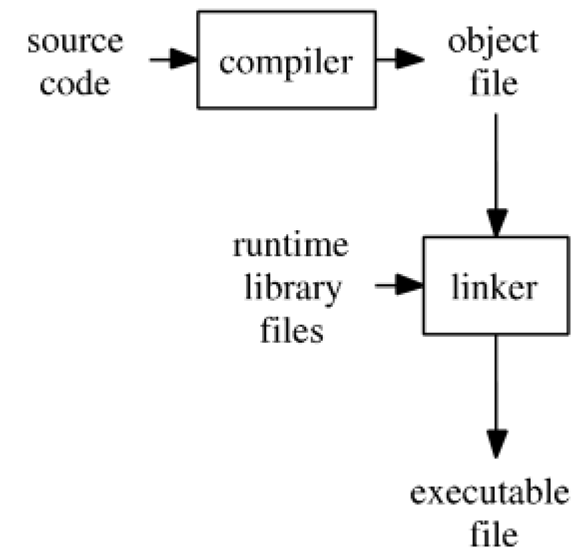
- Is any formally defined language a programming language?

(HTML?)

  - Universal: All computation problems should be expressable. condition+(loop and/or recursion)
  - Natural: All features required for the application domain.
    - Fortran: numerical computation,
    - COBOL: file processing,
    - LISP tree and list operations.
  - Implementable: It is possible to write a compiler or interpreter working on a computer. Mathematics, natural language?
  - Efficient: Works with acceptable amount of CPU and memory.

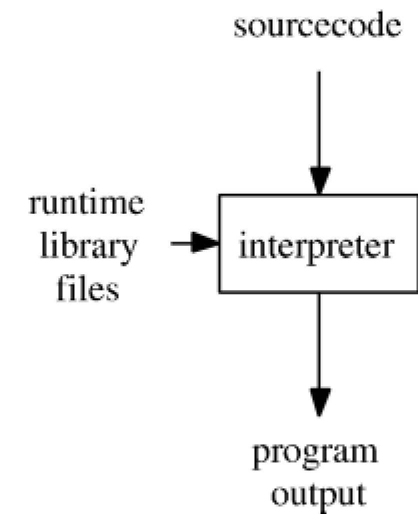
Compiled Programming Languages

- A compiled language is a programming language
  - implementations are typically **compilers** (translators that generate machine code from source code)
  - not interpreters (step-by-step executors of source code)
- A combination of both solutions is also common:
  - a compiler can translate the source code into some intermediate form (often called p-code or bytecode),
  - which is then passed to an interpreter which executes it.
- Examples
  - C, C++
  - COBOL
  - Haskell
  - Objective-C
  - Ada
  - Fortran…



Script Languages

- No need to compile
- Interpreted language
  - most of its implementations execute instructions directly and freely
  - The interpreter executes the program directly
- Interpreted languages use an intermediate representation which combines
  - compiling
  - intepreting.
- Examples
  - Javascript
  - Lisp
  - Python
  - Ruby
  - Bash
  - Matlab ...



Advantages Disadvantages of Compiled Languages

- Programs compiled into native code at compile time tend to be faster than those translated at run time
  - Due to overhead of the translation process
  - Compiled prog. makes it easier for programmers to control use of central processing unit (CPU) and memory in fine detail.
- Disadvantages
  - Needs compilation in every platform
  - Needs more effort…
- Mixed solutions (JAVA) using bytecode tend toward intermediate efficiency

Advantages of Interpreted Languages

- Interpreting a language gives implementations some additional flexibility over compiled implementations.
- platform independence (Python, Java's byte code, for example)
- reflection and reflective use of the evaluator (e.g. a first-order eval function)
- dynamic typing
- smaller executable program size (since implementations have flexibility to choose the instruction code)
- dynamic scoping

- Furthermore, source code can be read and copied, giving users more freedom.

Disadvantages of Interpreted Languages

- Without static type-checking, which is usually performed by a compiler, programs can be less reliable, because type checking eliminates a class of programming errors.
- Interpreters can be susceptible to Code injection attacks.
- Slower execution compared to direct native machine code execution on the host CPU.
- Source code can be read and copied (e.g. JavaScript in web pages), or more easily reverse engineered through reflection in applications where intellectual property has a commercial advantage.
  - In some cases, obfuscation is used as a partial defense against this.

Examples

C++

```
#include <iostream>
int main(){
  std::cout << ‘’Hello’’<<std::endl;
  return 0;
}
```

Bash

```
echo hello
```

Examples

C++

```
#include <iostream>
int main(){

  for(int i=0;i<100;i+=10){
        std::cout << i << std::endl;
  }
  return 0;
}
```

Bash

```
#!/bin/bash
for number in {0..100..10}
do
        echo "$number "
Done
```

For more examples: https://www.cyberciti.biz/faq/bash-for-loop/

Paradigms

- Paradigms : Theoretical or model frame. A model forming a basis for all similar approaches.
- Imperative
  - Fortran, Cobol, Algol, Pascal, C, C++, Java, Basic.
- Functional
  - Lisp, Scheme, ML, Haskell, Miranda
- Object Oriented
  - Smalltalk, C++, Object Pascal, Eifell, Java, C#.
- Concurrent
  - Ada, Occam, Par-C, Pict, Oz Logic Prolog, Icon
- Constraint Programming
  - CLP, CHR
- Mixed
  - Parallel Prolog, Oz (A functional, logic, object oriented, concurrent language: http://www.mozart-oz.org)

Syntax and Semantics

- Syntax Form: How language is structured, how it is expressed.
- Syntax is represented by Context Free Grammars expressed in BNF (Bacus Naur Form) notation (Automata Theory)
- Semantics What does a program mean? How it works.

Language Processors

- Compilers
  - gcc, javac, f77
- Interpreters
  - scheme, hugs, sml, bash
- Beautifiers, pretty printers
  - a2ps, ident
- Syntax directed editors
  - vim, anjuta, eclipse, visual studio
- Validators
  - vgrind, lint
- Verifiers

Values & Types

- Value anything that exist, that can be computed, stored, take part in data structure.
  - Constants,
  - variable content,
  - parameters,
  - function return values,
  - operator results...
- Type set of values of same kind.
  - C Types
    - int,
    - char,
    - long,...
    - float,
    - double pointers structures: struct, union arrays

Values & Types

- Each type represents a set of values.
- Is that enough?
- What about the following set? Is it a type?
  - {"ahmet", 1 , 4 , 23.453, 2.32, ’b’}
- Values should exhibit a similar behavior. The same group of operations should be defined on them.

Primitive vs Composite Types

- Primitive Types: Values that cannot be decomposed into other sub values.
  - C: int, float, double, char, long, short, pointers
  - Haskell: Bool, Int, Float, function values
- Cardinality of a type: The number of distinct values that a datatype has. Denoted as:
  - ”#Type”.
  - #Bool = 2
  - #char = 256
  - #short = 216
  - #int = 232
  - #double = 232 , ...
- What does cardinality mean?
  - How many bits required to store the datatype?

Static Type Checking

- Compile time type information is used to do type checking.
- All incompatibilities are resolved at compile time.
- Variables have a fixed time during their lifetime.
- Most languages do static type checking
- User defined constants, variable and function types:
  - Strict type checking. User has to declare all types (C, C++, Fortran,...)
  - Languages with type inference (Haskell, ML, Scheme...)
- No type operations after compilation. All issues are resolved. Direct machine code instructions.

Dynamic Type Checking

- Run-time type checking.
- No checking until the operation is to be executed.
- Interpreted languages like Lisp, Prolog, PHP, Perl, Python.
- A hypothetical language:

```
int whichmonth(input) {
        if (isinteger(input))    return input;
        else  if (isstring(input))
                switch(input) {
                case "January": return 1;
                case "February": return 2;
                ...
                case "December": return 12;}
}
...
read(input)      /* user input at run time? */
ay=whichmonth(input)
```

Dynamic Type Checking

- Run time decision based on users choice is possible.
- Has to carry type information along with variable at run time.
- Type of a variable can change at run-time (depends on the language).

Static vs Dynamic Type Checking

- Static type checking is faster. Dynamic type checking does
  - Type checking before each operation at run time.
  - Uses extra memory to keep run-time type information.
- Static type checking is more restrictive meaning safer.
  - Bugs avoided at compile time, earlier is better.
- Dynamic type checking is less restrictive meaning more flexible.
  - Operations working on dynamic run-time type information can be defined.

Type Equality

- S ? ≡ T
- How to decide?
  - Name Equivalence: Types should be defined at the same exact place.
  - Structural Equivalence: Types should have same value set. (mathematical set equality).
- Most languages use name equivalence.
  - C example

```c
typedef struct Comp { double x, y;}  Complex;
struct COMP { double x,y; };

struct Comp a;
Complex b;
struct COMP c;

/* ... */
a=b;   /* Valid, equal types */
a=c;   /* Compile error, incompatible types */
```

Variable Lifetime

- Variable lifetime:
  - The period between allocation of a variable and deallocation of a variable.
- 4 kinds of variable lifetime.

1. Global lifetime (while program is running)

   Life of global variables start at program startup and finishes when program terminates.

2. Local lifetime (while declaring block is active)

   a variable defined in a function or statement block, is the time between the declaring block is activated and the block finishes.

3. Heap lifetime (arbitrary)

   Allocation and deallocation is not automatic but explicitly requested by programmer via function calls.

   C: malloc(), free(), C++: new, delete

4. Persistent lifetime (continues after program terminates)

file, database, web service object

Garbage collection

- A solution to dangling reference and garbage problem:
  - PL does management of heap variable deallocation automatically. This is called garbage collection.
    - Java, Lisp, ML, Haskell, most functional languages
    - no call like free() or delete exists.
- How?
  - Count of all possible references is kept for each heap variable.
  - When reference count gets to 0 garbage collector deallocates the heap variable.
  - Garbage collector usually works in a separate thread when CPU is idle.

Memory Representation

- Global variables are kept in fixed region of data segment in memory
  They are directly accessible
- Heap variables are kept in dynamic region of data segment in memory

  In a data structure. A memory manager required.
- Local variables are usually kept in run-time stack (Why?)
  - recursion, each call needs its own set of local variables

Thanks For Your Attention

Questions?