Hi Everyone!

I am happy and honored to have this opportunity to talk about the design of programming languages.

I will focus on what we may expect from the programming languages that will be designed in the near future as a consequence of new developments in our field.

WHAT WILL BE THE NEXT PROGRAMMING LANGUAGE?

SBLP 2021

Roberto S. Bigonha

UFMG

October, 1st, 2021

Need for New Languages

- **Continuous demands** - There is a continuous demand for new languages
  - ◇ **new platforms demand new languages** - At any time, new programming principles and new platforms may be considered of the highest value for particular applications, and therefore new languages may have to be developed.
- **Special constructions**
  - ◇ **oportunity for simplification** - One motivation for new features is that, frequently, programming can be simplified if special constructions are available.
- **Mapping real-world to program environment**
  - ◇ **semantic gap** - For example, programming involves defining mappings from real-world entities to program objects.

  The larger the semantic gap between the real-world entities and those in the program environment, the more difficult is the programming process.
- **Adequate programming notation**

  Clearly, an adequate programming notation is necessary to reduce this gap.

Program Correctness

- **Program correctness** - Another important point is that correctness is an essential attribute of any software. It is more relevant than program efficiency or programmer's productivity.
- **Correctness X Efficiency** - After all, exaggerating a bit, a very efficient program with bugs is only good for producing errors at high speed!
- **Correctness X Productivity** - And a very productive programmer whose programs are mostly wrongly encoded is a complete disaster!
- **Correctness enhancement** - Therefore, correctness of the implemented code must be pursued by all means. This demands the adoption of strong static type-checking disciplines and special programming methodologies, such as programming by contract, that is, a systematic association of preconditions and postconditions to the operations encapsulated in modules.
- Good programming languages should enhance program correctness

Less Execution Errors

- **Software maintenance is expensive and unavoidable**
  - ◇ **programs must be readable** - Another important point is that the source code of computer programs may have to be modified over its lifetime, so good programming languages must stimulate the production of readable programs.
- **Trade-offs between easy writing and easy reading**
  - ◇ **easy reading should prevail** - For instance, the use of some language constructs may allow writing programs quickly, but they may make it difficult for other programmers to change the program code, probably jeopardizing correctness.
- **Strong static type-checking is mandatory**
  - ◇ **less execution errors** - In addition, type error detection mechanisms should be always incorporated to the compilers, and the type checking rules for large-scale programs should always be statically performed. When this type discipline is adopted, if a program is successfully compiled, then only specification or logical errors can occur. This imposes a need for a robust type discipline.

Fundamental Data Structures

- **Primitive types** - The most common primitive types are:
  - ◇ basic types: **integer, long, real, double, decimal, complex, char, boolean, string, pointers**
  - ◇ special types: **labels, semaphores, exception errors, files**
- **Data aggregations** - The most common data aggregations are:
  - ◇ arrays, records, structures and unions
  - ◇ lists and sets and tables
- **Types are a fundamental concept**

  Types became a fundamental concept in high-level programming languages. They deeply pervade all languages as primitive types or data aggregations. All new imperative languages must also consider addressing this concept properly!
- **New imperative languages must provide these types!**

Types

- Well, what is a type?
  - ◇ **set of operations** - In this context, a type in a computer program is defined by the operations that characterize the behavior of objects of the type.
  - ◇ **primitive or user-defined** - Extensibility of types is an important feature.
- **Hierarchy of types**
  - ◇ **subtypes and supertypes** - And the set of operations defined for each type may be used to create hierarchies of types in which each subtype has at least all the operations of its supertype. Hierarchy of types is the basis to implement type-checking disciplines.
- General type-checking rule

  An entity of type `T1` may be used in a context where one of type `T2` is expected only when type `T1` is in the type hierarchy of `T2`, that is, `T1` is a subtype of `T2`.

Type-Checking Disciplines

- **Strong type-checking**

  In strongly-typed languages, all type errors are reported either by the compiler or by the execution system. In weakly typed languages not all errors are detected.

  ◇ **static-type checking** - With static type-checking, all type errors are detectable by the compiler.

  ◇ **dynamic-type checking** - With dynamic type-checking, some type errors can only be detected during execution.

- **Weak type-checking**

  ◇ **not all errors detected** - On the other hand, in weakly-typed languages, some type errors may not be detected, and this is too bad because programs with type errors may produce wrong results! Consequently, in this case, if the program compiles, all the programmer could do is starting praying hoping that nothing bad will happen during execution!

Alternative Type-Checking Discipline

- **Programmer-defined type hierarchy**
  - ◇ **explicit is-a relationship** - The hierarchies of types are defined by the relationship between classes and subclasses specified by the programmer. There are, however, languages that adopt different methods for building hierarchies.
- **Compiler-inferred type hierarchy**
  - ◇ **implicit is-a relationship** - For example, hierarchies of types in a program may be deduced by the compiler from the operations defined for each type.
  - ◇ **accidental relationships** - Hierarchies constructed in this way may establish accidental relationships among conceptually unrelated types. Consequently, the suitability of an object for specific uses is determined by its possession of the required operations rather than its actual type. This is the duck-typing discipline.
- The idea behind duck-typing discipline is:

  If an object walks like a duck and it quacks like a duck, it must be a duck
- Needless to say that program correctness demands **strong static typing**.

The Compiler Role

- **Redundancy in type declaration**
  - ◇ **type of variables may be inferred** - The explicit association of types to variables is redundant, because the type of variables can be inferred from their use.
- **Static-type checking discipline** - However, this redundancy is very useful because it turns the compiler into an ally of the programmer, since, in a well-defined static-type system, the compiler warns the programmer whenever he makes any mistake regarding types, and thus forcing him to improve the code quality before releasing the software.
- **Dynamic-type checking discipline**

  On the other hand, in a dynamically-typed environment, the compiler generates code to inform to the user of the software the programmer's mistakes! So, in this case, the compiler works for the users!

Need for Clear Notations

- **Motivation** - Another point is that we need clear programming notations.
  - ◇ **clear notation to favor readability** - Simplicity is another very important design principle. The search for simplicity comes from the fact that we need programming notations to express the engineering solutions clearly and directly. Indeed, in software development processes, there is no substitute for simplicity.
- Fundamental idea behind this design principle is the Ockham's razor:
  - ◇ **original:** *Entia non sunt multiplicanda praeter necessitatem*
  - ◇ **literally:** *Entities must not be multiplied beyond necessity*
  - ◇ **free translation:** *Never complicate entity's descriptions unless it is absolutely necessary*.

    The Ockham's razor is also called law of parcimony.
- **Benefits**
  - ◇ A benefit of the law of parcimony is simplicity in the design results.

Simplicity as a Goal

- **Simplicity is complex** - However, to achieve simplicity is not a simple task!
  - ◇ Leonardo da Vinci said:

  Simplicity is the ultimate sophistication

  This definition suggests that the achievement of simplicity can be a very complex endeavor, which was brilliantly and precisely resumed by Blaise Pascal, who said:
  - ◇ **Blaise Pascal:**

  I wrote this long letter because I did not have the time to make it shorter

- How to achieve simplicity?
  - ◇ **orthogonal design** - One way to achieve simplicity is instead of designing a special language construct for each kind of abstraction that may be required by some applications, one should design collections of simple and orthogonal constructions that can be combined to produce all the complex abstractions that may be desired. By orthogonal constructions, I mean independent constructions and flexible combination rules. Algol 68 is a good example.

Taming the Complexity

- **Software component management**
  - ◇ **complex endeavor** - And, in large-scale programs, programmers must manage the interaction among software components, and this can be a complex endeavor.
- **Right modular structure** - And this demands a flexible module structure
  - ◇ **separation of concerns** - In fact, to keep complexity under control, separation of concerns must be enforced all the time, and different kinds of abstraction must be easily implementable in direct and clear ways.
  - ◇ **separate compilation** - For example, module definitions should be separated from their corresponding module implementations to allow separate compilation instead of independent compilation. Separate compilation enforces type checking across modules. In independent compilation, this may not be possible.
- **Adequate methodologies**
  - ◇ **specific advances** - Therefore, programming languages should be designed to provide means to exercise separation of concerns.

Adequate Methodology

- **Power of a software**
  - ◇ **ability to complexity management**

  Another relevant point is that the only thing that limits the power of a software is the ability of the programmer to manage complexity, and this requires well-defined methodologies, such as Divide-and-Conquer, and adequate notations.

- **Divide-and-Conquer**
  - ◇ **large-scale programs**

  The Divide-and-Conquer methodology is still the most effective way of taming programming complexity. Therefore, it must be part of the development process of large systems.

- **Means**
  - ◇ **separation of concerns, abstractions and modules**

  A good programming language must provide clear means to put this methodology into practice.

Divide and Conquer Tools

- **Human mind has limitation**
  - ◇ **separation of concerns** - Separation of concerns is a recognition that human beings work better within limited contexts. It is generally accepted that the human mind is limited to dealing with small units of data at a time. Software components are easier to be used if different concerns are separated into independent modules.
- **Abstraction in action**
  - ◇ **separating perception of behavior** - And abstraction is the act of separating the perception of the behavior of software components from their implementation details. It requires learning to look at software components from two points of view: what it does, and how it does it. Abstraction is then the ability to focus on what a specific solution does, without worring on how it does it.
- **Adequate constructions required**
  - ◇ **many types of modules** - Implementation of separation of concerns and abstraction demands appropriate language constructions.

Types of Modules

- **Implementation demands**

  Several types of modules are always required. And there must be a type of module for each group of semantically relevant constructions in the language. In a modern imperative language, the types of modules needed should at least be:

- **Procedure abstraction - Procedure abstraction is the abstraction of a sequence of commands and the associated data they handle**
- **Data abstraction - Data abstraction is the abstraction of variable declarations and the declarations of the procedures that handle them**
- **Type abstraction - Type abstraction is the abstraction of type definitions and the declarations of the procedures that handle variables of these types**

THE INITIAL LANDMARK

1954 $\Rightarrow$ 1961

FORTRAN - [1954–1958]

- FORTRAN, the FORmula TRANslation language, was created by IBM under the leadership of John Backus in 1958
- Data types: INTEGER, REAL, DOUBLE, COMPLEX, LOGICAL, and arrays
- Type discipline: statically typed
- Memory allocation: statically allocated when the program is loaded
- Scope of names: local and global (common)
- Modules: subroutines that encapsulate data declarations and code
- Parameter mechanism: by reference
- **Key contributions:** variable = address abstraction, commands = machine instruction abstractions (assignment, IF, DO, and GOTO), basic types, arrays, subroutines, parameter passing by reference, two level of scope for names

Fortran Parameter by Reference

- **Houston, we have a problem!**

  In 1970, NASA reported that Apolo 13's routine maintenance task went awry and caused the spacecraft's oxygen tanks to explode!

  The rumor was that some misuse of Fortran call by reference caused it!

```
REAL FUNCTION F(X,Y)
F = X**2 + Y**2
RETURN
END
...
SUBROUTINE S(I)
I = I + 1
RETURN
END
```

```
    DIMENSION A(10)
    R = F(10.0,20.0)
    K = 6
    CALL S(K)
    CALL S(2)
    WRITE(1,5) K, 2
5   FORMAT(2I3)
    STOP
    END
Saída:  7  3
```

A Safer Fortran Program

- In Fortran, a constant should never be passed as parameter to functions or subroutines in Fortran
- It is safer to pass an expression that produces the constant value

```
      REAL FUNCTION F(X,Y)
      F = X**2 + Y**2
      RETURN
      END
      ...
      SUBROUTINE S(I)
      I = I + 1
      RETURN
      END
```

```
      DIMENSION A(10)
      R = F((10.0),(20.0))
      K = 6
      CALL S(K)
      CALL S((2))
      WRITE(1,5) K, 2
5     FORMAT(2I3)
      STOP
      END
Saída:  7  2
```

Algol 60 - [1957–1960]

- ALGOL 60, the ALGOrithm Language, was created by a committee led by Peter Naur in 1960
- Data types: integer, real, boolean, arrays, labels
- Type discipline: statically typed
- Scope of names: multiple levels of enclosing blocks
- Memory allocation: dynamic by means of a stack
- Modules: external procedures
- Parameter mechanism: by value and by name
- Key contributions: nested blocks, scope of names, dynamic memory allocation in the form of stacks of activation records, commands if-then-else, while, for, new parameter passing mechanisms, and recursive procedures

An Algol 60 Program

- **The program**

```
begin
  integer r;
  integer procedure factorial(n); value n; integer n;
    factorial := if n > 0 then n X factorial(n - 1) else 1;
  r := factorial(4);
  begin
    integer r;
    r := factorial(3);
    outreal(r);
  end;
  outreal(r);
end.
```

- **prints 24 and 6** - **also a very familiar notation.**

COBOL - [1959–1961]

- COBOL, the COmmon-Bussiness Oriented Language, was created by a committee led by Grace Hopper from DOD in 1961
- Application: Data processing
- Data types: integer, real, decimal, pictures, arrays, and records
- Type discipline: statically typed
- Commands: assignment, IF, GO, PERFORM, DATA DIVISION and PROCEDURE DIVISION
- Memory allocation: static
- Modules: independent routines
- Parameter mechanism: by value and by reference
- Key contributions: sophisticated file manipulation structure, precise decimal arithmetic, and a new data aggregation: the record

A COBOL Program

```
DATA DIVISION
FD FILE1;
   DATA RECORD IS S.

01 S.
   02 A PICTURE IS X(80)
   02 B
      03 DAY PICTURE IS 99.
      03 MONTH PICTURE IS 99.
      03 YEAR PICTURE IS 99.
   02 C PICTURE IS X(40).
```

```
PROCEDURE DIVISION
   OPEN OUTPUT FILE1.
L.
   ...
   MOVE somevalue TO S.A.
   ...
   MOVE somevalue to S.B.
   ...
   WRITE S.
   ...
   GOTO L
```

Record introduced the idea of data aggregations of elements of different types. All important languages follow suit!

The Immediate Consequence

- The second phase in language development started in the first years of the 1960s, following the success of Fortran, Algol 60 and Cobol
- The objectives were the development of languages with:
  - ◇ a rich collection of commands and data types
  - ◇ extensible flow of control structures
  - ◇ constructions to perform special operations, e.g., pattern matching
  - ◇ special constructions to cope with treatment of errors
  - ◇ constructions for modelling concurrent programs
  - ◇ flexible mechanisms of memory allocation
- An important implementation instance of this thought is PL/I

PL/I - [1964]

- PL/I, the Programming Language One, was created by IBM in 1964
- Influenced by: Fortran, Cobol e Algol 60
- Data types: CHARACTER, PICTURE, BINARY, FIXED, DECIMAL, FLOAT, COMPLEX, POINTER, CONTROLLED, arrays, structures and semaphores
- **Type discipline: strong typing**
- Commands: blocks, IF, SELECT, WHEN, OTHERWISE, DO, GOTO, ALLOCATE, FREE, I/O commands, exception handling, multitasking execution, and a sophisticated file system.
- **Memória: stack and heap**
- **Modules: external procedures**
- Parameter mechanism: by value and by reference
- **Key contributions:** extensible general purpose language.

PL/I Command Extensibility

```
%uwrite:
 procedure keys (File, From, Count);
 dcl (File, From, Count, Number, Size) char;
 if parmset(File) & parmset(From) then; else do;
    note ('FILE and FROM must be specified!', 12);
    return;
    end;
 if parmset(Count)
    then Size = 'min(length(' || From || '), ' || Count || ')';
    else Size = 'length(' || From || ')';
 Number = Counter();
 ans ('do;');
 ans ('dcl Count' || Number || ' fixed bin (15);' ) skip;
 ans ('Count' || Number || ' = filewrite(' || File || ', ptradd(addr('
                      || From || '), 2)' || ', ' || Size || ');') skip;
 ans ('end;') skip;
 %end;
 %act uwrite;
```

Use of Macros

- **The non-PL/I statement**

  ```
  uwrite file(file_name) from(var_str) count(64);
  ```

  is pre-processed by the compiler to generate the following PL/I code:

  ```
  do;
  dcl Count00001 fixed bin (15);
  Count00001 = filewrite(file_name, ptradd(addr(var_str), 2),
                                    min(length(var_str), 64));
  end;
  ```

- **Code may be unreadable!**

  ◇ **no user-manual!** - **Each macro has its own syntax and semantics. Note that to understand the user-defined uwrite statement, the definition of macro uwrite must be studied directly from the source code! And, of course, this is not practical with large-scale programs. A great merit of PL/I was that it provoked a scientific discussion on what would be a good programming language.**

The Reaction

- **Reaction to PL/I approach**
  - ◇ **PL/I too complex** - The complexity of the language PL/I provoked a healthy reaction from the scientific community.
  - ◇ **parcimony law to the rescue** - The parcimony law has been invoked.
- **Structured Programming**
  - ◇ **simplicity** - The result of this reaction was the development of Structured Programming, which was a computer science view of simplicity implementation.
- **Type extensibility**
  - ◇ **not command extensibility** - In addition, at that time, it had formed a consensus that only types should be extensible, not commands.
- **Type-checking mechanisms** - And that a suitable type checking mechanism is fundamental to support the development of correct programs.
- Important new languages: Algol 68, Pascal, and C

Algol 68 - [1963 – 1968]

- The language Algol 68 was created in 1968 by a special committee: Fritz Bauer, E. Dijsktra, P. Naur, C.A.R. Hoare, N. Wirth, P. Landin
- Influenced by: Algol 60
- Data types: int, real, char, bool, string, compl, bits, bytes, sema, format, file, references, structures, arrays, union
- Type discipline: statically typed
- Modules: still only external procedures
- Parameter mechanism: by value
- Key contributions:
  - ◇ user-defined types: `mode A = struct (int x, real y, char z)`

    The `mode` declaration defines a new non-abstract type.
  - ◇ orthogonality principle

Algol 68 Programs

```
begin
  int i;
  real x, y;
  [1:3] proc(real) real f :=
                    (sin,cos,tan);
  to 100 do
     read((i,x));
     y := f[i](x);
     print(y)
  od
end
```

```
begin
  int n; read(n);
  mode vector = [1:n] real;
  vector v,v1,v2,v3,v4;
  op + (vector x,y) vector:
  begin vector sum;
    for i to upb x
      do sum[i]:=x[i]+y[i] od;
    sum
  end;
  read ((v1,v2,v3,v4));
  v := v1 + v2 + v3 + v4;
  print(v5)
end
```

Disjoint Union in Algol 68

- **union (int, real) x**
  in which `x` may hold int or real values.

- **Incorrect use of disjoint union:**

```
union(int, real) x;
int k;
real r;
...
x := 3; ... ; x := 2.0; ... ;
...
k := x;  <== invalid operation
...
r := x;  <== invalid operation
...
```

- **The correct use via conformity clause:**

```
union(int, real) x;
int k;
real r;
...
x := 3; ... ; x = 2.0; ... ;
...
case x in
   (int intval)   : k := intval,
   (real realval) : r := realval
esac
```

- **Algol 68 showed how to implement a strongly statically typed `union`!**
  No need for compiler generated code for dynamic-type checking!

Pascal - [1964–1970]

- The language Pascal was created by Niklaus Wirth (ETH) in 1970
- Influenced by: Algol 60 and Algol 68
- Data types: integer, subrange, enumeration, real, boolean, char, array, record, union, set, pointer, and files
- user-defined non-abstract data types:

  type A = record x: integer; y: real; end
- Commands: assignment, if-then-else, while, repeat, goto, call
- Memory allocation: stack and heap
- Modules: external procedure
- Parameter mechanism: by reference and by value
- Type discipline: weakly typed because of `variant record`, a kind of type union
- Key contributions: enumeration, subrange, set, a reduced collection of commands and a simple useful language.

C - 1973

- The language C was created by Dennis Richie at ATT in 1973
- Influenced by: Algol 68
- Data types: char, short, int, long, float, double, structures, union, arrays, pointers
- Type discipline: weakly typed
- User-defined non-abstract types: `struct A {int x; real y;};`
- Commands: similar to Pascal
- Variable scope: two levels only, global or local to a function
- Memory allocation: static, stack and heap
- Modules: data abstraction, although not ADT.
- Parameter mechanism: by value
- Key contributions: efficiency, flexibility, data abstraction implementation, and an alternative to assembly languages

Data Abstraction in C

- **File mystack.h:**

```
void push(int);
int pop();
```

- **File mystack.c**

```
static int stack[1000];
void push(int x) f {...}
int empty() {...}
int pop() {...}
```

- **File main.c**

```
#include mystack.h;
main (argc, argv)
int x;
{ ...
  push(10);
  ...
  x = pop();
  ...
  stack[5] = 5; /* unknown */
  ...
}
```

File mystack.h provides the module interface, while file mystack.c, the module implementation $\Rightarrow$ primitive separate compilation!

ABSTRACT DATA TYPE TRACK

$1970 \Rightarrow 1980$

The Invention of ADT - [1974]

- In 1974, Barbara Liskov at MIT developed the language CLU whose main construction is the `cluster`, a built-in mechanism for creating abstract data types
- The cluster construction of CLU allows:
  - ◇ **information hiding**: so the representation of the type defined by the cluster can be completely encapsulated and protected from outside access
  - ◇ **type extensibility**: so that programmers can freely create new types
- **Greatest invention** - We believe that ADT was the greatest invention of the 1970 decade!

CLU ADT

```
stack : cluster is push, pop, top, empty;
  rep(t: type) =
     rep (tp: integer; etype: type; stk: array[1...] of t;);
  create
     s : rep(elemtype);
     s.tp := 0; s.etype := elemtype;
     return s;
  end
  push: operation(s:rep, v:s.etype);
    s.tp := s.tp + 1; s.stk[s.tp] := v;
    return;
  end
  ...
end stack;
```

Modula-2 - [1977–1980]

- Modula-2 was created by Niklaus Wirth at ETH in 1980
- Influenced by: Pascal, Algol 68, C and CLU
- Data types: char, short, int, long, float, double, record, union, arrays, and pointers
- Extensibility of types: allows implementation of Liskov's ADT
- Type discipline: mostly statically typed, except for unions
- Commands: similar to Pascal + concurrent processes, and coroutines
- Memory allocation: stack, static e dynamic
- Modules: definition and implementation module
- Parameter mechanism: by reference and by value
- Key contributions: type extensibility via ADT, separate (not independent) compilation via module definition and module implementation

Modula-2 Abstract Data Type Module

```
DEFINITION MODULE CharStackModule;
    TYPE CharStack;
    PROCEDURE Push(VAR s: CharStack, ch: CHAR);
    PROCEDURE Pop(VAR s: CharStack, ch: CHAR):CHAR;
    PROCEDURE IsEmpty(s: CharStack):BOOLEAN;
END CharStackModule.
IMPLEMENTATION MODULE CharStackModule;
    FROM IO IMPORT  Wistr, WrLn;
    CONST stackSize = 100;
    TYPE StackRecord = RECORD
         stackArray:  [1..stackSize] OF CHAR;
         topOfstack: [0..stackSize + 1];
    END
    CharStack = POINTER TO StackRecord;
    PROCEDURE Push(VAR CharStack,ch: CHAR); BEGIN ... END Push;
    ...
    BEGIN
       topOfstack := 0
END CharStackModule.
```

ADA - [1975-1980]

- The language ADA was created by J. Ichbiah et alii at DoD in 1980
- Influenced by: Algol 68, Pascal and CLU
- Data types: char, short, int, long, float, double, record, union, arrays, pointers
- Type extensibility: type definition + package produce new parametrized ADTs
- Type discipline: statically typed like Algol 68
- Commands: like Pascal + exceptions + concurrent programming
- Scope of names: external and local to functions
- Memory allocation: stack, static, and dynamic
- Modules: package interface and package body

  This package components support separate compilation.
- Parameter mechanism: in, out and in out
- Key contributions: type extensibility via ADT, separate compilation via module definition and module implementation

ADA ADT Module

```
package RATIONAL_NUMBERS is
    type RATIONAL is private;
    function equal (X,Y : RATIONAL) return BOOLEAN;
    function add (X,Y : RATIONAL) return RATIONAL;
    function times (X,Y : RATIONAL) return RATIONAL;
    private
       type RATIONAL is record
              NUMERATOR : INTEGER;
              DENOMINATOR: INTEGER range 1..INTEGER'LAST;
       end record;
end;
package body RATIONAL_NUMBERS is
    ...
    function equal (X,Y : RATIONAL) return BOOLEAN is begin ... end;
    function add (X,Y : RATIONAL) return RATIONAL is begin ... end;
    function times (X,Y : RATIONAL) return RATIONAL is begin ... end;
    BEGIN
    ...
end RATIONAL_NUMBERS;
```

OBJECT-ORIENTED PROGRAMMING TRACK

1967 $\Rightarrow$ 1995

Simula 67 - [1964–1967]

- Simula 67 created by Ole-Johan Dahl and Kristen Nygaard in 1967
- Data types: Algol 60 types + primitive class
- **Type discipline: statically typed**
- Modules: classes and external procedures
- Parameter mechanism: by value, by name and by reference
- **Key contributions:**
  - ◇ Enrich Algol 60 with classes, subclasses, objects, single inheritance, polymorphism, coroutines, and type extensibility
  - ◇ Define the initial notions of encapsulation of data and associated operations, objects, methods, type hierarchy, inheritance and polymorphism
  - ◇ Establish of new programming paradigm - Simula 67 class and object concepts were the basis for the development of Smalltalk by Alan Kay and the creation of abstract data type by Liskov and Ziller in 1974.

Simula 67 Class Structure

- **A program**

```
begin
   class P(a,b); integer a,b;
   begin
      a :- a * 10;
      b :- b * 10;
   end;
   ref (P) x, y;
   x :- new P(1,2);
   y :- new P(3,5);
   y.b :- 10;
   x.a :- y.b + 6;
end
```

- **A class hierarchy structure**

```
class A(PA); SA;
begin DA;
      IA;
      inner;
      FA
end
A class B(PB); SB;
begin DB;
      IB;
      inner;
      FB
end
```

Smalltalk - [1969-1980]

- The language Smalltalk was created by Alan Kay at Xerox in 1980
- Influenced by: Simula 67, Algol 68
- Data types: all variables and constants are objects, including those of basic types
- Type extensibility: class + visibility control + single inheritance
- Type discipline: dynamic typing, including duck typing
- **Commands: message senders**

  Smalltalk's syntax is based on sending messages to objects. This emphasizes that operations are done by the objects, not to them! An object is a private data and a set of operations that can access that data.
- **Modules: classes**
- Parameter mechanism: by reference and by value
- **Key contributions:** pure object-oriented programming, information hiding, and message-oriented programming

Smalltalk Class Definition

```
Object subclass: #Stack
  instanceVariableNames: 'top'
  classVariableNames: ''
  poolDictionaries: ''  !

!Stack class methods !

new
  | s |
  s := super new.
  s setsize: 10.
  ^s! !
```

```
!Stack methods !
pop
  | item |
  item := anArray at: top.
  top := top - 1.
  ^item!

push: item
  top := top + 1.
  anArray at: top put: item!

setsize: n
  anArray := Array new: n.
  top := 0! !
```

Smalltalk is Message-Oriented

- In Smalltalk, all objects, including basic constants, are able to react to messages, which trigger specific behavior on them
- For example, the Smalltalk assignment statement

$$\texttt{r} := \underbrace{\underbrace{\overbrace{\texttt{3 factorial}}^{6} + \overbrace{\texttt{4 factorial}}^{24}}_{30}\ \texttt{between:10 and:100}}_{true}$$

is evaluated as follows:

1. `3` receives the message `factorial` and answers `6`
2. `4` receives the message `factorial` and answers `24`
3. `6` receives the message `+` with argument `24` and answers `30`
4. `30` receives the message `between:and:` with `10` and `100` as arguments and answers `true`
5. message `:=` with argument `true` is sent to object `r`

Uniform View of All Values

- **Uniform view of values**

  In Smalltalk, all objects, including basic constants, are able to react to messages, which trigger specific behavior on them

- **Is this a good idea?**
  - ◇ **in favor: orthogonality principle**

    – Yes, the number of primitive concepts in language should be minimum, independent of each other, and allowed to be combined without restrictions.
  - ◇ **against: efficiency consideration**

    – However, basic constants are special structures that pervade all program code, and thus deserve to receive an exceptional treatment in order to enhance operation efficiency.

- **Practice** – Most languages treat constants as special structures, not objects

Objective-C - [1980-1984]

- Designed by Tom Love and Brad Cox in 1984
- Influenced by: Smalltalk-80 messaging style and language C
- **Typing discipline: dynamic typing** - It is possible to send messages to object that does not have these messages in its interface. It works by means of a message forwarding mechanism.
- Data types: types of C + classes with single inheritance
- Visibility control: internal data of classes are private
- Commands: similar to C + Smalltalk messaging notation
- **Memory allocation: automatic, static e dynamic**
- **Modules: external files, which export classes and other entities**
- Parameter mechanism: by value and by reference
- **Key contributions:** protocol-oriented programming

  Protocol is similar to Java's interfaces with default operations.

C++ - [1983-1985]

- The language C++ was created by B. Stroustrup at At&T in 1985
- **Data types: types of C + classes and templates**
- Visibility control: public, protected e private attributes
- **Extensibility of types:** classes allow implementation of generic ADTs
- **Type discipline: strongly typed**
- Commands: similar to C + exception handling mechanism
- **Memory allocation: dynamic**
- **Modules: external files, which export classes and other entities**
- Parameter mechanism: by constant and by value
- **Key contributions:** classes with flexible visibility control, single and multiple inheritance, and templates, which are parameterized types
- C++ is a very rich language, may be too rich!

A C++ ADT

```
#include <iostream.h>
template <class T> class Stack{
private: ...
public:
   Stack() { ... }
   ~Stack() { ... }
   void push(T x);
   T pop(); ...
}
template <class T>
  void Stack<T>::push(T x){...}
template <class T>
  T Stack::pop() {...}
...
```

```
#include ...
int main() {
   int z;
   Stack<int> stk;
   ...
   stk.push(4);
   ...
   stk.push(1);
   ...
   z = stk.pop();
   ...
}
```

Eiffel - [1986–1988]

- The language Eiffel was created by Bertrand Meyer in 1988
- Influenced by: CLU, Smalltalk, Pascal, C++
- **Data types: similar to Pascal, user-defined types**
- **Extensibility of types:** classes allow implementation of new ADT
- **Type discipline: statically typed**
- **Commands: similar to C++**
- **Memory allocation: stack and heap** (access type)
- **Modules: classes** - whose members may be public, protected, or private
- Parameter mechanism: by constant
- **Key contributions:**
  - ◇ programming by contract: which consists in viewing the relationship between a class and its clients as a formal agreement, expressing each party's rights and obligations.
  - ◇ clean syntax and semantics

An Eiffel ADT

```
class STACK[T] export
  empty, full, push, pop, top
feature
  size: INTEGER;
  ...
  empty: BOOLEAN is do ... end;
  push(x : T) is
    require ...
    do ...
    ensure ...
  end;
  pop : T is do ... end;
  top : T is do ... end;
end -- class STACK
```

```
class C export
  ...
feature
  i : INTEGER;
  s : STACK[INTEGER]
  member_function is do
     s.push(4);
     i := s.pop;
     s.push(2);
     i := s.top;
  end;
  ...
end -- class C
```

Python - 1991

- The language Python was created by Guido van Rossum in 1991
- Influenced by: Algol 68, CLU, Smalltalk and C++
- Data types: basic types, arrays, dictionaries, lists, tuples, and sets
- Type discipline: dynamically typed, variables do not have types. Python popularized this idea borrowed from Smalltalk, and many languages follow suit.
- Extensibility of types: classes with public and private properties, multiple inheritance and virtual functions.
- Commands: try-except and raise, multithreading, synchronization primitives such as locks, events, condition variables and semaphores
- Module: a file containing definitions and statements
- Parameter mechanism: by value
- Key contributions: compact code, list comprehension, tuples, sets, and dictionaries, which are lists of key, value pairs

A Python Class Definition

```
class Mapping:
  def __init__(self, iterable):
      self.items_list = []
      self.__update(iterable)
  def update(self, iterable):
      for item in iterable:
          self.items_list.append(item)
  __update = update

class MappingSubclass(Mapping):
  def update(self, keys, values):
      for item in zip(keys, values):
      self.items_list.append(item)
```

Python Duck Typing

```
class A:
   def f(self):
        print("f of A")

   def g(self):
        print("g of A")

class B:
   def f(self):
        print("f of B")

for animal in [A(), B()]:
    animal.f()
    animal.g()
```

Output:

```
f of A
g of A
f of B

AttributeError:
'B' object has no attribute 'g'
```

Python Traits

- Programs in Python is typically compact because:
  - ◇ its high-level data types allow expressing complex operations in a single statement
  - ◇ statement grouping is done by indentation instead of beginning and ending brackets
  - ◇ no variable or argument declaration is necessary
- It is considered an ideal language for scripting and rapid application development in many areas on most platforms
- Would the Python's emphasis on flexibility turned it into the PL/I of modern times?

  This is an open question.

Lua - 1993

- The language Lua was created by Roberto Yerusalimschy, Waldemar Celes and Luiz Henrique de Figueiredo at PUC-Rio in 1993
- Influenced by: Scheme, CLU, Modula-2 and C++
- Applications: Lua is an extensible extension language, designed to implement embedded systems and games.
- **Data types:** No classes, but the table data type allow implementing objects, and there are mechanisms to simulate inheritance.
- Type discipline: dynamically typed, variables do not have types
- Extensibility: tables allow implemention of arrays, records, and sets
- **Commands: similar to C**
- **Modules: functions and coroutines**
- Parameter mechanism: by value
- **Key contributions:** tables and metatables, and the elegant design of a powerful and simple extensible extension programming language

A Lua Code

```
function circular(n)
  list = {} -- an empty table
  current = list
  i = 0
  while i < n do
    current.value = i
    current.next = {}
    current = current.next
    i = i+1
  end
  current.value = i
  current.next = list
  return current
end
```

```
function clone (o)
  local new_o = {}
  local i, v = next(o,nil)
  while i do
    new_o[i] = v
    i, v = next(o,i)
  end
  return new_o
end
```

Java - [1995]

- Java was created by James Gosling at Sun Microsystems in 1995
- Influenced by: C++ and Eiffel
- Data types: classes + int, byte, short, int, long, float, double
- **Type discipline: strongly typed**
- Extensibility of types: ADT specification and implementation classes
- Visibility: member of classes can be public, protected e private
- **Commands: similar to C++, including exception handling**
- **Memory allocation: stack, static and heap**
- **Modules: classes e packages**
- Parameter mechanism: by value, but originally parameters could not be methods!
- **Key contributions:** modularity with flexible control of visibility, single inheritance of classes, multiple inheritance of interfaces and independence of platforms: write once and run anywhere property.

A Java ADT

```
public class S<T> {
  private int last;
  private T[] itens;
  private int top = -1;
  public S(int max) {
    this.last = max -1 ;
    itens=(T[])new Object[max];
  }
  public void push(T v)
         throws ExMax {...}
  public T pop( )
         throws ExMin {...}
  public boolean empty() {...}
}
```

```
public class TestOfS {
 public ... main(String[] args)
  S<Integer>p=new S<Integer>(3);
  S<Double> q=new S<Double>(4);
  int a, x[] = 1,2,3,4,5;
  double b,y[] = {1.0,2.0,3.0};
  for(int i : x) p.push(i);
  for(double i : y) q.push(i);
  while(!p.empty()) {
    a = p.pop();
    System.out.print(" " + a);
  }
 }
}
```

AspectJ - [2001]

- Created by Gregor Kiczales in 2001
- Objective: modularization of crosscutting concerns
- **Data types**
  - ◇ similar to Java
- **Commands**
  - ◇ similar to Java
- **Modules**
  - ◇ classes creating new types and the aspect mechanism that encapsulates crosscutting concerns
- **Key contributions:**
  - ◇ extension of Java to incorporate the aspect mechanism that eases the separation of crosscutting concerns across modules

Crosscutting Concerns

- **Crosscutting concerns are shown in red:**

```
public class C  {
    "Private members of module C";
    public void operation(OperationInformation info) {
       "Authenticate the user for the operation";
       "Block current object to assure synchronism";
       "Make sure that cache is up-to-date";
       "Do login to initiate operation";
       "Perform the objective operation";
       "Do logout to finalize operation";
       "Liberate access to current object";'
    }
}
```

- **Code in red should be moved to different modules**

ASPECTJ Modules

- **Classe with the crosscutting concerns removed:**

```
public class C  {
   "Private members of module C";
   public void operation(OperationInformation info) {
      "Perform the objective operation";
   }
}
```

- **Aspects with the code to insert the concerns removed back into `C` :**

```
public aspect Autentication { "authentication code" }
public aspect Synchronism { "synchronization code"}
public aspect Cache { "cache administrator code" }
public aspect Logging { "logging in and out codes" }
```

JavaScript - 2005

- JavaScript has been designed by Brendan Eich at Netscape in 2005
- Influenced by: Java, Scheme, Self
- Application: Scripting language
- Data types: Number, BigInt, String, boolean, Array, Map, Set and objects
- Type discipline: dynamically typed, variables are announced without a type and have the type of the value they currently hold
- Extensibility of types: classes with visibility control, inheritance
- Commands: usual assignment, for, while, if statements
- Modules: classes
- Parameter mechanism: by value
- Key contributions: a implementation of prototypal inheritance

JavaScript Class Inheritance

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed = speed;
  }
  stop() {
    this.speed = 0;
  }
}
```

```
class Rabbit extends Animal {
 constructor(name,earlength){
   super(name);
   this.earlength = earlength;
 }
 hide() {
  alert(`${this.name} hides!`);
 }
}
```

```
let rabbit = new Rabbit("while rabbit",10);
rabbit.run(5); rabbit.hide();
```

JavaScript Prototypal Inheritance

- An object can inherit properties from other objects via a mechanism called **Prototypal Inheritance**
- For example, object `y` inherits the property `a` from object `x`:

```
let x = { a: ..., b: ..., }
let y = { c: ..., b: ...,  }
...
Object.setPrototype(y,x);
// from now y has a pointer to x
...
...y.c ...; ... y.b; ... y.a ... ;
```

- Since `a` is not declared in `y`, `y.a` is obtained by following the prototypal pointer that connects `y` to `x` to get the value of `x.a`
  This mechanism makes this world a dangerous land to live.!!!

Swift - 2014

- The language Swift was created by Apple Inc in 2014
- Influenced by: Objective-C, Python
- Data types: basic types, arrays, structures, sets, dictionaries
- Type discipline: strong statically typed, variables do not have declared types, their types are inferred from the first value assigned to them
- Extensibility of types: classes, inheritance, protocols and extensions
- Visibility control: no restrictions on accessing class properties
- Commands: usual assignment, for, while, if, etc statements
- Memory allocation: stack, static and dynamic
- Parameter mechanism: by value
- Modules: structures, classes and protocols
- Key contributions: protocol-oriented programming.

Example of Swift Protocols

```
protocol P {
   init()
   func f()
   func g()
   func h()
}
```

```
extension P {
   func h() { ... }
}

class A {
   init() { ... }
   func f() { ... }
}
```

```
class B: A, P {
   override init() { ... }
   func g() { ... } // definitions of f and h are inherited
}
```

- **Swift Protocol resembles Java default-interface or vice-versa**

CONCLUSIONS

2021

Mandadory Design Guidelines

1. The design of constructs for the next language should respect the law of parcimony
2. The orthogonality principle should be always considered to ensure good quality design choices
3. Readability should always prevail over writability
4. Correctness is a non-negotiable requirement
5. Strong static type checking discipline should be a priority
6. The next language should consider incorporate consolidated achievements on language expressive power.
7. Languages for large-scale programming should have at least the indispensable constructions proved as valuable so far
8. Focus on innovation is a basic requirement for a new language!

Indispensable Constructs

The next imperative language for large-scale programming should provide mechanism to implement:

- strongly static-type checking facilities
- abstract data type, abstractions, separation of concerns,
- object-oriented programming,
- single-class inheritance, multiple-interface inheritance,
- object encapsulation, information hiding, polymorphism,
- programming by contract,
- exception handling facilities,
- adequate module structure, separate compilation
- and, most importantly, real worthwhile innovations

Examples of Consolidated Achievements

- **Modularity**

  In the 1970s, it became clear that the most effective mechanism to face the complexity of large-scale programs is modularity.

- **Types of modules**

  ◇ **relevant abstractions** - In fact, it became widely accepted that there must be a type of module for each semantically relevant construction in the language.

- **Programming practice**

  As a consequence, abstract data type, object-oriented programming, and protocol-oriented programming have been invented, and languages like C++, Java and Eiffel were developed. And those concepts and associated methodologies became universally accepted and put to practice.

- The next language should honor those important scientific and methodological achievements!

An Example of Possible Innovation

- **Imagination gap**

  Another important point is a missing support for the development of large programs by reducing the gap from the coding act to the results that will be produced when the program is executed.

  As an analogy, a painter sees the effect of what he is doing all the time during the painting process. If he sees a mistake, he may correct it on fly. With computer programming is quite different: the programmer can only see the effect of what he is writing much later when the program is executed.

- Reducing imagination gap is a priority

  ◇ **new constructions to see the results** - we need facilities and new constructions to reduce this gap, allowing the programmer to see immediately the implication of what he is encoding, and hence improving program correctness.

- Innovation should always be the main reason for designing new languages.

Not Recommended Design Choices

- Abdication of strong static type checking
- Liberation of variables from being explictly declared, even when it is clearly redundant
- Adoption of indentation for delimiting statement and function body instead of the traditional beginning and ending brackets
  Although indentation encourage compactness it seems to be error prone in large-programs!
- Omission of fundamental and traditional data types even when they could be simulated by means of other constructions
- Inclusion of subreptitious mechanisms like JavaScript Prototypal Inheritance

Talk Structure

- **Fundamental background**
  - ◇ **review of basic concepts** - We started reviewing fundamental concepts on programming language development to set up the basis on which everything should be built.
- **Historical facts**
  - ◇ **breakthrough languages** - In the sequel, details of the design of a few breakthrough languages were presented
  - ◇ **important constructions** - And some important language constructions that cannot be forgotten were discussed and contextualized.
- **Conclusions**
  - ◇ **show the way** - And the conclusions present suggestions for what should be the next imperative programming language for large-scale programs.

An Interesting Citation

- **The profile of the next language**

  Now that we know what should be the main characteristics of what should be the next imperative programming language for the development of large-scale programs, I would like to mention that this talk accidentally honors the following Albert Einstein's citation.

- **Albert Einstein, Evolution of Physics, 1938:**

  "The formulation of a problem is often more essential than its solution, which may be merely a matter of mathematical or experimental skill. To raise new questions, new possibilities, to regard old problems from a new angle requires creative imagination and marks real advances in science."

- **Fortunately, the road to what will be the next programming language has been well paved by those who designed the languages we have discussed in this talk!**

Thank You!