Modern Programming Languages

Introduction and Historical Background

Lecture 1-8

Course Objectives

Thousands of different programming languages have been designed by different people over the last 60 years. The questions that come to mind are:

- Why are there so many different programming languages?
- How and why they are developed?
- What is the intended purpose of a language?
- In what ways are they similar?
- What are the differences among them?
- Why wouldn't we simply continue to use what we have today?
- What kinds of programming languages may be developed in future?

We will try to answer some of these questions in this course.

In addition, we will discuss the design issues of various languages, design choices and alternatives available, historical context and specific needs, and specific implementation issues.

Text Book

The main text book for this course is:
Concepts of Programming Languages, 6th Ed. by Robert Sebesta.

Reasons to study concepts of Programming Languages

The first question is: why should we study programming languages. There are many reasons for that and some of them are enumerated in the following paragraphs.

1. Increased capacity to express programming concepts

Study of programming languages helps in increasing the capacity of express programming concepts.

Dijkstra has put it as follows:

> *The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.*

That is, one is limited in his/her thinking by the tools used to express his/her ideas.

Depth at which we can think is influenced by the expressive power of the language. It includes the kind of algorithms you can develop. The range of software development thought process can be increased by learning new languages as those constructs can be simulated.

2. Improved background for choosing appropriate languages

Study of programming languages also helps one in choosing the right language for the given task. Abraham Maslow says, "To the man who only has a hammer in the toolkit, every problem looks like a nail."

That is, if the only tool you have is a hammer, then you will treat every problem like a nail. Sometimes, some programming languages are more suitable for a specific task. There are many special purpose languages. In this course we will study one such language by the name of Snobol.

3. Increased ability to learn new languages

Study of different programming languages also helps one in learning new languages by learning the syntax and semantics of different languages and understanding different design methodologies.

4. Understanding the significance of implementation

In some cases, an understanding of implementation issues leads to an understanding of why languages are designed the way they are. This ultimately leads to efficient use of the language. One such example is Row vs. column major. If a programmer knows that two-dimensional arrays are stored column-wise (column major) in FORTRAN (where in most other languages it is row major) then he will be careful to process it column-wise, hence making it more efficient.

Same is the case with recursion. If the programmer knows how recursion is implemented and the associated cost of recursive programs, he can use this knowledge to come-up with more efficient programs if needed.

Also, certain bugs can only be found and fixed if the programmer knows some related implementation details.

5. Increased ability to design new languages

By learning a number of programming languages, one gets to know the pros and cons of different language features and issues related to these features. This knowledge will therefore help if one has to design a new language for any purpose.

Language Evaluation Criterion

In order to evaluate and compare different languages, we need some mechanism for their evaluation. The first criterion that comes to mind is: how long it takes to develop a program in a given programming language.

Capers Jones has developed the following Programming Languages Table which relates languages with the productivity.

Language Level Relationship to Productivity

LANGUAGE LEVEL	PRODUCTIVITY AVERAGE PER STAFF MONTH
1 - 3	5 to 10 Function Points
4 - 8	10 to 20 Function Points
9 - 15	16 to 23 Function Points
16 - 23	15 to 30 Function Points
24 - 55	30 to 50 Function Points
Above 55	40 to 100 Function Points

As can be seen, a higher-level language will yield more productivity as compared to a lower level language. Some of the common languages with their levels are listed below.

Assembly(1), C(2.5), Pascal(3.5), LISP(5), BASIC(5), C++(6)

As can be seen, C++ has the highest level in this list.

Let us now try to use this criterion to evaluate different languages.

When a programmer starts to learn a new language, a typical first exercise is to program the computer to display the message "Hello World". A compilation of “Hello World programs” designed by various categories of “developer” follows.

Hello World Programs – adapted from: Infiltec Humor Page
www.infiltec.com

A compilation of *Hello World programs* designed by various categories of *developer* follows.

High School/Jr.High - BASIC Language

```
10 PRINT "HELLO WORLD"
20 END
```

First year in College - Pascal

```
program Hello(input, output)
  begin
    writeln('Hello World')
  end.
```

Senior year in College - LISP

```
(defun hello
  (print
    (cons 'Hello (list 'World))))
```

New professional - C

```
#include
void main(void)
{
  char *message[] = {"Hello ", "World"};
  int i;

  for(i = 0; i < 2; ++i)
    printf("%s", message[i]);
  printf("\n");
}
```

Seasoned professional – C++

```
==========================
#include
#include

class string
{
private:
  int size;
  char *ptr;

public:
  string() : size(0), ptr(new char('\0')) {}

  string(const string &s) : size(s.size)
  {
    ptr = new char[size + 1];
    strcpy(ptr, s.ptr);
  }

  ~string()
  {
    delete [] ptr;
  }

  friend ostream &operator <<(ostream &, const string &);
  string &operator=(const char *);
};

ostream &operator<<(ostream &stream, const string &s)
{
  return(stream << s.ptr);
}

string &string::operator=(const char *chrs)
{
  if (this != &chrs)
  {
    delete [] ptr;
   size = strlen(chrs);
    ptr = new char[size + 1];
    strcpy(ptr, chrs);
  }
  return(*this);
}

int main()
{
  string str;

  str = "Hello World";
  cout << str << endl;

  return(0);
}
```

```
[
uuid(2573F8F4-CFEE-101A-9A9F-00AA00342820)
]
library LHello
{
    // bring in the master library
    importlib("actimp.tlb");
    importlib("actexp.tlb");

    // bring in my interfaces
    #include "pshlo.idl"

    [
    uuid(2573F8F5-CFEE-101A-9A9F-00AA00342820)
    ]
    cotype THello
 {
 interface IHello;
 interface IPersistFile;
 };
};

[
exe,
uuid(2573F890-CFEE-101A-9A9F-00AA00342820)
]
module CHelloLib
{

    // some code related header files
    importheader();
    importheader();
    importheader();
    importheader("pshlo.h");
    importheader("shlo.hxx");
    importheader("mycls.hxx");

    // needed typelibs
    importlib("actimp.tlb");
    importlib("actexp.tlb");
    importlib("thlo.tlb");

    [
    uuid(2573F891-CFEE-101A-9A9F-00AA00342820),
    aggregatable
    ]
    coclass CHello
 {
 cotype THello;
 };
};

#include "ipfix.hxx"

extern HANDLE hEvent;
```

```
class CHello : public CHelloBase
{
public:
    IPFIX(CLSID_CHello);

    CHello(IUnknown *pUnk);
    ~CHello();

    HRESULT  __stdcall PrintSz(LPWSTR pwszString);

private:
    static int cObjRef;
};

#include
#include
#include
#include
#include "thlo.h"
#include "pshlo.h"
#include "shlo.hxx"
#include "mycls.hxx"

int CHello::cObjRef = 0;

CHello::CHello(IUnknown *pUnk) : CHelloBase(pUnk)
{
    cObjRef++;
    return;
}

HRESULT  __stdcall  CHello::PrintSz(LPWSTR pwszString)
{
    printf("%ws\n", pwszString);
    return(ResultFromScode(S_OK));
}

CHello::~CHello(void)
{

// when the object count goes to zero, stop the server
cObjRef--;
if( cObjRef == 0 )
    PulseEvent(hEvent);

return;
}

#include
#include
#include "pshlo.h"
#include "shlo.hxx"
#include "mycls.hxx"

HANDLE hEvent;
```

```
 int _cdecl main(
int argc,
char * argv[]
) {
ULONG ulRef;
DWORD dwRegistration;
CHelloCF *pCF = new CHelloCF();

hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

// Initialize the OLE libraries
CoInitializeEx(NULL, COINIT_MULTITHREADED);

CoRegisterClassObject(CLSID_CHello, pCF, CLSCTX_LOCAL_SERVER,
    REGCLS_MULTIPLEUSE, &dwRegistration);

// wait on an event to stop
WaitForSingleObject(hEvent, INFINITE);

// revoke and release the class object
CoRevokeClassObject(dwRegistration);
ulRef = pCF->Release();

// Tell OLE we are going away.
CoUninitialize();

return(0); }

extern CLSID CLSID_CHello;
extern UUID LIBID_CHelloLib;

CLSID CLSID_CHello = { /* 2573F891-CFEE-101A-9A9F-00AA00342820 */
    0x2573F891,
    0xCFEE,
    0x101A,
    { 0x9A, 0x9F, 0x00, 0xAA, 0x00, 0x34, 0x28, 0x20 }
};

UUID LIBID_CHelloLib = { /* 2573F890-CFEE-101A-9A9F-00AA00342820 */
    0x2573F890,
    0xCFEE,
    0x101A,
    { 0x9A, 0x9F, 0x00, 0xAA, 0x00, 0x34, 0x28, 0x20 }
};

#include
#include
#include
#include
#include
#include "pshlo.h"
#include "shlo.hxx"
#include "clsid.h"

int _cdecl main(
int argc,
char * argv[]
```

```
) {
HRESULT  hRslt;
IHello        *pHello;
ULONG  ulCnt;
IMoniker * pmk;
WCHAR  wcsT[_MAX_PATH];
WCHAR  wcsPath[2 * _MAX_PATH];

// get object path
wcsPath[0] = '\0';
wcsT[0] = '\0';
if( argc  1) {
    mbstowcs(wcsPath, argv[1], strlen(argv[1]) + 1);
    wcsupr(wcsPath);
    }
else {
    fprintf(stderr, "Object path must be specified\n");
    return(1);
    }

// get print string
if(argc  2)
    mbstowcs(wcsT, argv[2], strlen(argv[2]) + 1);
else
    wcscpy(wcsT, L"Hello World");

printf("Linking to object %ws\n", wcsPath);
printf("Text String %ws\n", wcsT);

// Initialize the OLE libraries
hRslt = CoInitializeEx(NULL, COINIT_MULTITHREADED);

if(SUCCEEDED(hRslt)) {

    hRslt = CreateFileMoniker(wcsPath, &pmk);
    if(SUCCEEDED(hRslt))
 hRslt = BindMoniker(pmk, 0, IID_IHello, (void **)&pHello);

    if(SUCCEEDED(hRslt)) {

 // print a string out
 pHello->PrintSz(wcsT);

 Sleep(2000);
 ulCnt = pHello->Release();
 }
    else
 printf("Failure to connect, status: %lx", hRslt);

    // Tell OLE we are going away.
    CoUninitialize();
    }

return(0);
}
```

Apprentice Hacker
==========

```
#!/usr/local/bin/perl
$msg="Hello, world.\n";
if ($#ARGV >= 0) {
  while(defined($arg=shift(@ARGV))) {
    $outfilename = $arg;
    open(FILE, ">" . $outfilename) || die "Can't write $arg: $!\n";
    print (FILE $msg);
    close(FILE) || die "Can't close $arg: $!\n";
  }
} else {
  print ($msg);
}
1;
```

Experienced Hacker
==========

```
#include
#define S "Hello, World\n"
main(){exit(printf(S) == strlen(S) ? 0 : 1);}
```

Seasoned Hacker
==========

```
% cc -o a.out ~/src/misc/hw/hw.c
% a.out
```

Guru Hacker
==========

```
% cat
Hello, world.
^D
```

New Manager – Back to BASIC
==========

```
10 PRINT "HELLO WORLD"
20 END
```

Middle Manager
==========

```
mail -s "Hello, world." bob@b12
Bob, could you please write me a program that prints "Hello,
world."?
I need it by tomorrow.
^D
```

Senior Manager
==========

```
% zmail jim
I need a "Hello, world." program by this afternoon.
```

Chief Executive
==========

```
% letter
letter: Command not found.
% mail
To: ^X ^F ^C
% help mail
help: Command not found.

% damn!
!: Event unrecognized
% logout
```

Language Evaluation Criterion

Jokes aside, this set of programs gives the readers some initial ideas about the different types of languages. Anyway, the question is: what happened to the programming language levels? Why is there so much variation for a simple program like "Hello World". The answer to this question lies in the question itself – "Hello World" is not the program that should be analyzed to compare different languages. It is so simple that the important features of different programming languages are not highlighted and wrong conclusions can be drawn.

A fair comparison needs much deeper analysis than this type of programs. Bjarne Stroustrup – the designer of C++ language – makes the following observation in this regards:

> *I have reaffirmed a long-standing and strongly held view: Language comparisons are rarely meaningful and even less often fair. A good comparison of major programming languages requires more effort than most people are willing to spend, experience in a wide range of application areas, a rigid maintenance of a detached and impartial point of view, and a sense of fairness.*

The question remains then how to compare and examine languages?

For this purpose we will use Readability, Writability, Reliability as the base criteria.

Readability

Readability is directly related to the cost of maintenance. The different factors that impact readability are:

- Simplicity
- Control Statements
- Data types and data structures
- Syntax Considerations
- Orthogonality

Simplicity

A simple language will be easy to learn and hence understand. There are a number of factors that account for simplicity. These are:

- Number of basic components

Learning curve is directly proportional to the number of basic components. If a language has more basic components, it will be difficult to learn and vice-versa. It is important to note that one can learn a subset of a programming language for writing but for reading you must know everything.

- Feature multiplicity
  If a language has more than one way to accomplish the same task, then it can cause confusion and complexity. For example, a number can be incremented in C in any of the following manners.

  ```
  i = i + 1;
  i++;
  ++i;
  i += 1;
  ```

  If a programmer has used all these different constructs to increment a variable at different locations, the reader may get confused. It is simpler to adopt one structure and use it consistently.

- Operator overloading

  Operator overloading can be problematic if its use is inconsistent or unconventional.

- How much simple should it be?

  Simplicity has another dimension as well. If a language is very simple then its level may also be decreased as it may loose abstraction. In that case, it will be difficult to write and read algorithms in that language.

Control Statements

Control statements also play an important role in readability. We are all aware of the hazards of goto statement. If a language relies on goto statements for control flow, the logic becomes difficult to follows and hence understand and maintain. Restricted use of goto in extreme was needed and useful as well but with the emergence of new language constructs to handle those situations, it probably not an issue any more.

Data types and data structures

Data types and data structures also play an important role in improving the abstraction level of a programming language, hence making it more readable. For example, a simple omission of Boolean type in C resulted in many problems as integers were instead of Boolean which made the program difficult to follow and debug in many cases.

Similarly, in FORTRAN there was no record (or struct) type. So related data had to be put in different data structures, making it very difficult to read the logic and maintain.

Syntax

Syntax of a language plays a very important role in programming languages. There are many different considerations that we will discuss throughout this course. Here we give a brief overview of some of these considerations.

- Variable names
  The first consideration is the restriction of variable names. In the beginning it was a huge issue as some languages restricted the length of a variable name up to 6 or 8 characters making it difficult to choose meaningful names for the variable. Other considerations include the type of special characters allowed in the variable names to improve readability.

- Special keywords for signaling the start and end of key words.

  Some languages allow special keywords for signaling the start and end of a construct, making it more readable and less prone to errors. This can be elaborated with the help of following examples:

  Let us first consider the following C if statement.

```
if (some condition)
        // do this
        // now do this
```

  It may be noted that the second part "// now do this" is not part of the body of the if statement and it will be executed regardless of the truth value of the condition. This may cause problems as it is easy for the reader to miss this point because of indentation.

  Ada programming language solves this problem by using the keywords "then" and "end if" in the if statement. The resulting code is shown below:

```
if (some condition) then
        -- do this
end if
        -- now do this
```

  Now it can be easily seen that a reader of this program will not be confused by the indentation.

Another similar example is when the if has an else part also. Let us first consider the following C code.

```
if (cond1)
        if (cond2)
          // do 1
    else if (cond3)
      // do 2
else
    // do 3
```

Is the last else part of third if or the first one? Once again the indentation has caused this confusion and the reader can easily miss this point. Ada has solved this problem by introducing an elsif keyword. The Ada code is shown below and it can be seen very easily that the confusion has been removed.

```
if (cond1) then
        if (cond2) then
           -- do 1
        elsif (cond3) then
           -- do 2
        end if;
else
    -- do 3
end if;
```

Writability

Writability is a measure of support for abstraction provided by a language. It is the ability to define and use complicated structures without bothering about the details. It may be noted that the degree of abstraction is directly related to the expressiveness. For example, if you are asked to implement a tree, it will be a lot difficult in FORTRAN as compared to C++. It also is a measure of expressivity: constructs that make it convenient to write programs. For example, there is a set data type in Pascal which makes it easier to handle set operations as compared to a language such as C++ where this support is not available.

Orthogonality

Orthogonality is a very important concept. It addresses how relatively small number of components that can be combined in a relatively small number of ways to get the desired results. It is closely associated with simplicity: the more orthogonal the design the fewer exceptions and it makes it easier to learn, read, and write programs in a programming language. The meaning of an orthogonal feature is independent of the context. The key parameters are symmetry and consistency. For example pointers is an orthogonal concept.

Here is one example from IBM Mainframe and VAX that highlights this concept. In IBM main frame has two different instructions for adding the contents of a register to a memory cell or another register. These statements are shown below:

```
A       Reg1, memory_cell
AR      Reg1, Reg2
```

In the first case, contents of Reg1 are added to the contents of a memory cell and the result is stored in Reg1; in the second case contents of Reg1 are added to the contents of another register (Reg2) and the result is stored in Reg1.

In contrast to the above set of statement, VAX has only one statement for addition as shown below:

```
ADDL operand1, operand2
```

In this case the two operands, operand1 and operand2, can be registers, memory cells, or a combination of both and the instruction simply adds the contents of operand1 to the contents of operand2 and store the result in operand1.

It can be easily seen that the VAX's instruction for addition is more orthogonal than the instructions provided by IBM and hence it is easier for the programmer to remember and use it as compared to the one provided by IBM.

Let us now study the design of C language from the perspective of orthogonality. We can easily see that C language is not very consistent in its treatment of different concepts and language structure and hence makes it difficult for the user to learn and use the language. Here are a few examples of exceptions:

- You can return structures but not arrays from a function
- An array can be returned if it is inside a structure
- A member of a structure can be any data type except void or the structure of the same type
- An array element can be any data type except void
- Everything is passed by value except arrays
- void can be used as type in a structure but you cannot declare a variable of this type in a function.

Orthogonality The Other side

Too much orthogonality is also troublesome. Let us again have a look at C language and try to understand this concept:

- In C
  - All statements (including assignment, if, while, etc) return some value and hence can be used in expressions.
  - Logic and arithmetic expressions can be intermixed.

  This can cause side effects and cryptic code.

  Since languages need large number of components, too much orthogonality can cause problems. From a language designer's point of view, the most difficult task is to strike a balance which obviously is not trivial.

Reliability

Reliability is yet another very important factor. A programming language should enable the programmers to write reliable code. The important attributes that play an important role in this respect are listed below:

- Type Checking

  Type checking is related with checking the type of the variables used as operands in different. The question that needs to be addressed is whether this type checking is done at compile time or at run-time. This also includes checking types of parameters and array bounds. Traditionally, these two have been the major sources of errors in programs which are extremely difficult to debug. A language that does type checking at compile time generates more reliable code than the one that does it at run-time.

- Exception handling

  Exception handling enables a programmer to intercept run-time errors and take corrective measure if possible and hence making it possible to write more reliable code.

Cost

Cost is also a very important criterion for comparing and evaluating programming language. In order to understand the cost involved, one has to consider the following:

- Training cost – how much does it cost to train a new programmer in this language.
- What is the cost of writing programs in the language – this is a measure of productivity
- What is the cost of the development environment and tools?
- What is the compilation cost? That is, how long does it take to compile a program. This is related to productivity as the more time it takes to compile a program, the more time a programmer will be sitting idle and hence there will be a reduction in productivity.
- Execution cost is also an important factor. If the program written in a language takes more execution time then the overall cost will be more.
- A related is whether to have a more optimized code or to increase the compilation speed
- Cost of language implementation deals with the level of difficulty in terms of writing a compiler and development environment for the language.
- If the program written in a particular language is less reliable than the cost of failure of the system may be significant.
- The most important of all of these factors is the maintenance cost. It is a function of readability.

Portability

Portability deals with how easy it is to port a system written in a given programming language to a different environment which may include the hardware and the operating system. In today's heterogeneous environment, portability is a huge issue. Portability has to do a lot with the standardization of a language. One of the reasons the programs written in COBOL were less portable than C was because C was standardized very early whereas there was not universal standard available for COBOL.

Generality

Generality is also an important factor and deals with the applicability of the language to a range of different domains. For example, C is more general purpose than LISP or FORTRAN and hence can be used in more domains than these two languages.

Issues and trade-offs

Like all design problems, in the case of programming language design, one has to deal with competing criterion such as execution versus safety, readability versus writability, and execution versus compilation. It would be nice if one could assign weights to different criteria and then compare the different options. Unfortunately, this kind of help is not available and hence the balancing act, as usual, is a very difficult job.

Influence of computer architecture on language design

Over the years, development in the computer architecture has had a major impact on programming language design. With the decreasing cost and increasing speed of hardware we simply can afford the programming languages which are more complex and tolerate the programming language to be less efficient. We shall now study the impact of the hardware architecture on the evolution of the programming languages.

Babbage's Analytical Engine

Charles Babbage is considered to be the inventor of the first computer. This machine, known as the Analytical Engine (or Difference Engine), was invented in 1820's. In the beginning, it could only be made to execute tasks by changing the gears which executed the calculations. Thus, the earliest form of a computer language was physical motion.
The key features of this machine included Memory (called store), jump, loop, and the concept of subroutines. It was motivated by the success of power looms. It had limited capability and because of the technological limitations this design could not be fully implemented.

ENIAC (Electronic Numerical Integrator and Calculator)

Physical motion was eventually replaced by electrical signals when the US Government built the ENIAC (Electronic Numerical Integrator and Calculator) in 1942. It followed many of the same principles of Babbage's engine and hence, could only be "programmed" by presetting switches and rewiring the entire system for each new "program" or calculation. This process proved to be very tedious.

Von Neumann Architecture - 1945

In 1945, John Von Neumann was working at the Institute for Advanced Study. He developed two important concepts that directly affected the path of computer programming languages. The first was known as "shared-program technique". This technique stated that the actual computer hardware should be simple and not need to be hand-wired for each program. Instead, complex instructions should be used to control the simple hardware, allowing it to be reprogrammed much faster.

The second concept was also extremely important to the development of programming languages. Von Neumann called it "conditional control transfer". This idea gave rise to the notion of subroutines, or small blocks of code that could be jumped to in any order, instead of a single set of chronologically ordered steps for the computer to take. The second part of the idea stated that computer code should be able to branch based on logical statements such as IF (expression) THEN, and looped such as with a FOR statement. "Conditional control transfer" gave rise to the idea of "libraries," which are blocks of code that can be reused over and over.

In March 1949, Popular Mechanics made the following prediction:

"Where a calculator on the ENIAC is equipped with 18 000 vacuum tubes and weighs 30 tons, computers of the future may have only 1 000 vacuum tubes and perhaps weigh 1½ tons."

In 1949, a few years after Von Neumann's work, the language Short Code appeared. It was the first computer language for electronic devices and it required the programmer to change its statements into 0's and 1's by hand. Still, it was the first step towards the complex languages of today. In 1951, Grace Hopper wrote the first compiler, A-0. A compiler is a program that turns the language's statements into 0's and 1's for the computer to understand. This lead to faster programming, as the programmer no longer had to do the work by hand.

A most important class of programming languages, known as the imperative languages, is based upon the von Neumann Architecture. This includes languages like FORTAN, COBOL, Pascal, Ada, C, and many more.

Other major influences on programming language design have been mentioned as below:

Programming Methodologies

- 1950s and early 1960s:
  - *Simple applications; worry about machine efficiency*
- Late 1960s:
  - *People efficiency became important*
  - *readability, better control structures*
- Late 1970s:
  - *Data abstraction*
- Middle 1980s:
  - *Domain and data complexity - Object-oriented programming*
- Today
  - Web and networked environment; distributed computing

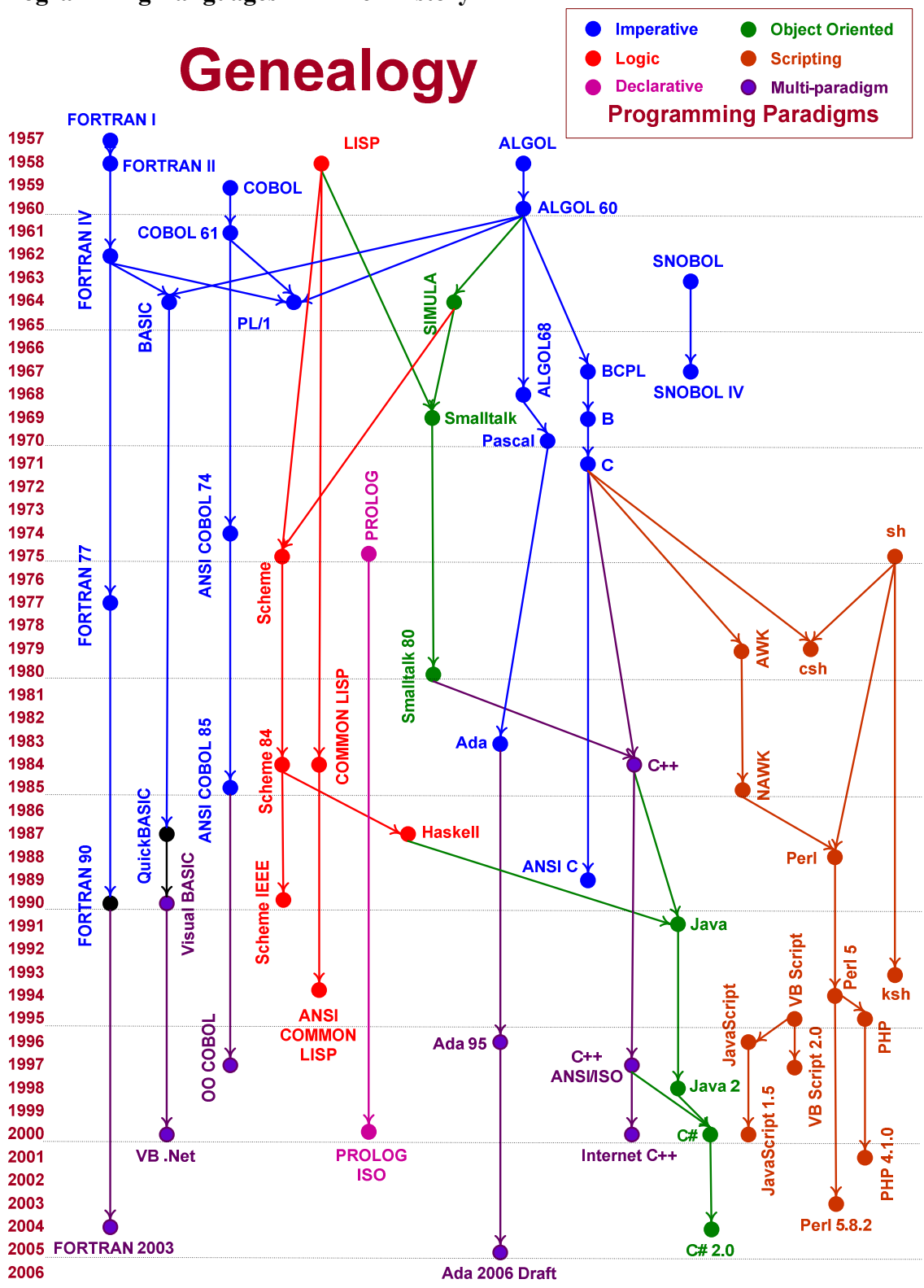
Language Categories

- Imperative
  - variables, assignment, iteration
- Functional (applicative)
  - set of functions
- Logic (Declarative)
  - rule based
  - Order of execution in not defined
- Object Oriented
  - close relatives of imperative

Application Domains

1. Scientific applications
    - simple data structures, large FP operations - FORTRAN
2. Business applications
    - reports, decimal arithmetic, character operations - COBOL
3. Artificial intelligence
    - Symbol processing, logic programming – Lisp, Prolog
4. Embedded systems
    - Concurrent programming - Ada
5. Systems programming
    - execution efficiency, low-level features – PL/1, BLISS, C
6. Scripting languages
    - list of commands – batch files, ksh, Perl
7. Special purpose languages
    - Hundreds of languages

Programming Languages - A Brief History



The First Programmer was a Lady

Charles Babbage and Ada Lovelace, lived in London of Dickens and Prince Albert (and knew them both). A hundred years before some of the best minds in the world used the resources of a nation to build a digital computer, these two eccentric inventor-mathematicians dreamed of building their "Analytical Engine". He constructed a practical prototype and she used it, with notorious lack of success, in a scheme to win a fortune at the horse races. Despite their apparent failures, Babbage was the first true computer designer, and Ada was history's first programmer.

Zuse's Plankalkül – 1945

The language was designed by the German scientist in isolation during the Second World War and was discovered and Published in 1972.

It was never implemented but included the following interesting features:

- The data type included floating point, arrays, records, nesting in records
- It had the notion of advanced data types and structures
- There was no explicit goto
- The concept of iteration was there
- It had selection without else part
- It also had the notion of invariants and assertions

Zuse used this language to solve many problems. The list of programs include sorting, graph connectivity, integer and floating point arithmetic, expressions with operator precedence, and chess playing. It may be noted that many of these problems remained a challenge for quite some time.

The language had a terse notation and hence was difficult to use. However, a number of programming language designer think that had this language be known at an earlier stage, we may never have seen languages like FORTRAN and COBOL.

Assemblers, Assembly Language, and Speed Coding

Initially, programs used to be written in machine code. It suffered from poor readability and poor modifiability – addition and deletion of instructions. Coding of expression was tedious. On top of that, machines did not support floating point and indexing and these things had to be coded by hand.

In 1954, John Backus developed what is known as Speedcoding for IBM 701. It had Pseudo operations for arithmetic and math functions, conditional and unconditional branching, and auto-increment registers for array access. It was slow as it was interpreted and after loading the interpreter, only 700 words left for user program. However, it made the job of programmer much easier as compared to the earlier situation.

The First Compiler - Laning and Zierler System - 1953

The first compiler was implemented on the MIT Whirlwind computer. It was the first "algebraic" compiler system which supported subscripted variables, function calls, and expression translation. It was never ported to any other machine.

FORTAN
The First High Level Language - FORTRAN I – John Backus 1957

FORTRAN stands FORmula TRANslating system. It was the first implemented version of FORTRAN as FORTRAN 0 (1954) was never implemented. It was the first compiled high-level language designed for the new IBM 704, which had index registers and floating point hardware. At that time computers were small and unreliable, applications were scientific, and no programming methodology or tools were available. Machine efficiency was most important and no need was felt for dynamic storage. So the language needed good array handling and counting loops. No string handling or decimal arithmetic was supported. Names could have up to six characters. It had support for formatted I/O and user-defined subprograms. There was no data typing statements and no separate compilation. The compiler was released in April 1957, after 18 man/years of effort. Unfortunately, programs larger than 400 lines rarely compiled correctly, mainly due to poor reliability of the 704 but code was very fast. Within one year, 50% of the code written for 704 was being developed in FORTRAN.

All statements of FORTRAN I were based on 704's instruction set. It included 3 way branch and computed if in the form of If (EXPRESSION) L1, L2, L3. It also had posttest counting loop as shown below:

```
DO L1 I = N, M
```

Fortran II (1958) and IV (1960)

Fortran II came after one year in 1958. It added independent compilation and had some bug fixes.

Fortran IV was released in 1960 and became the most popular language of its time. It had support for explicit type declarations and logical IF statement. Subprograms could also be passed as parameters. ANSI standard of FORTRAN IV was release in 1966 and remained mostly unchanged for the next 10 years.

FORTRAN 77 and 90

FORTRAN 77, released in 1977, was the next major release of FORTRAN after FORTRAN IV. It added support for structured Programming, character string handling, logical loop control statement, and IF-THEN-ELSE statement.

FORTRAN 90, released in 1990, added modules, dynamic arrays, pointers, recursion – stack frames, case statement, and parameter type checking. However, because of the

backward compatibility constraints, it could never gained the same popularity as its predecessors.

Functional Programming – LISP – McCarthy 1959

LISP stands for LISt Processing language. It was designed by John McCarthy in 1959 as part of AI research at MIT to deal with problems of linguistic, psychology, and mathematics. They needed a language to process data in dynamically growing lists (rather than arrays) and handle symbolic computation (rather than numeric). LISP has only two data types: atoms and lists and syntax is based on lambda calculus. It pioneered functional programming where there is no need for assignment and control flow is achieved via recursion and conditional expressions. It is still the dominant language for AI. COMMON LISP and Scheme are contemporary dialects of LISP and ML, Miranda, and Haskell are related languages.

ALGOL

ALGOL 58 – 1958 – Search for a "Universal Language"

ALGOL stands for ALGOrithmic Language. It was designed in 1958. At that time FORTRAN had (barely) arrived for IBM 70x and was owned by IBM. Many other languages were being developed, all for specific machines. There was no portable language as all were machine-dependent. Also, there was no universal language for communicating algorithms.

ALGOL was thus designed to be a language that was close to mathematical notation, good for describing algorithms, and was machine independent. That is, it was an algorithmic language for use on all kinds of computers.

Salient features of the language are:

- Concept of type was formalized
- Names could have any length
- Arrays could have any number of subscripts
- Lower bound of an array could be defined
- Parameters were separated by mode (in & out)
- Subscripts were placed in brackets
- Compound statements were introduced (begin ... end)
- Semicolon as a statement separator was used
- Assignment operator was :=
- if had an else-if clause

It was actually not meant to be implemented, but variations of it were (MAD, JOVIAL) implemented. Although IBM was initially enthusiastic but vested interest in FORTRAN resulted in taking back all support by mid-1959.

Algol 60 - 1960

ALGOL 60 was a modified version of ALGOL 58. It added new features to the language. These included block structure (local scope), two parameter passing methods – value and name, subprogram recursion, and stack-dynamic arrays – run time size definition and space allocation. It did not have built-in I/O facility.

It was quite successful as it was the standard way to publish algorithms for over 20 years. All subsequent imperative languages are based on it. It was the first machine-independent language and the first language whose syntax was formally defined in BNF. It also impacted the hardware design.

It was never widely used, especially in U.S because there was no i/o support and the character set made programs non-portable. Also, it was too flexible and hence was hard to understand and implement. On top of that, IBM never supported it on their machines because of their interest in FORTRAN. In addition, BNF was considered strange and complicated!

Algol 68 - 1968

ALGOL 68 was from the continued development of ALGOL 60, but it is not a superset of that language. Its design is based on the concept of orthogonality. Major contributions of this language include user-defined data structures, reference types, and dynamic arrays (called flex arrays).

From a practical point of view it had even less usage than ALGOL 60 but had strong influence on subsequent languages, especially Pascal, C, and Ada.

COBOL - 1960

COBOL was designed in 1960 to support business-oriented computation which requires fixed point arithmetic. It was designed to look like simple English to broaden the base of computer users. It was thus required to be easy to use, even if that means it will be less powerful. Another design consideration was that it must not be biased by current compiler.

It encountered problems because the design committee members were all from computer manufacturers and DoD branches and there were fights among manufacturers.

It was the first language to add macro facility in a high-level language. It also introduced hierarchical data structures (records) and nested selection statements. The language supported long variable names (up to 30 characters) with hyphens, data division, and fixed-point arithmetic.

It was the first language required by DoD and would probably have failed without support from DoD. It is still a very widely used business applications language and is very popular in business and government, much less at universities.

Basic - 1964

BASIC was designed by Kemeny & Kurtz at Dartmouth College with the following goals in mind: Easy to learn and use for non-science students; Must be "pleasant and friendly"; Fast turnaround for homework; Free and private access; User time is more important than computer time.

Current popular dialects include QuickBASIC and Visual BASIC.

PL/1 – 1965 – Everything for Everybody

From IBM's point of view there were two computing groups: scientific computing and business computing. The scientific computing was supported by IBM 1620 and 7090 computers and FORTRAN. Business computing was done on IBM 1401, 7080 computers and COBOL programming language.

By 1963 scientific users began to need more elaborate I/O, like COBOL had and business users began to need fl. pt. and arrays (MIS). It looked like many shops would begin to need two kinds of computers, languages, and support staff. This proposition was obviously too costly and the obvious solution was to build a new computer to do both kinds of applications and design a new language to do both kinds of applications. Hence PL/1 came into being.

PL/I was the first language to introduce unit-level concurrency, exception handling, pointer data type, and array cross sections.

The language was not a huge success because many new features were poorly designed and it was too large and too complex.

Early Dynamic Languages -Characterized by dynamic typing and dynamic storage allocation

APL (A Programming Language) 1962

It was designed as a hardware description language (at IBM by Ken Iverson). It was highly expressive (many operators, for both scalars and arrays of various dimensions) but programs are very difficult to read - commonly known as "write-only" language.

SNOBOL (1964)

It was designed as a string manipulation language (at Bell Labs by Farber, Griswold, and Polensky). It had powerful operators for string pattern matching but suffered from poot readability and maintainability.

Simula 67 – 1967 –The first Object-oriented language

It was designed in Norway by Nygaard and Dahl, primarily for system simulation. It was based on ALGOL 60 and SIMULA I. Its primary contributions include the concept of a class which was the basis for data abstraction. Classes are structures that include both local data and functionality.

Pascal – 1971 – Simplicity by Design

Pascal was designed by Niklaus Wirth, who quit the ALGOL 68 committee because he didn't like the direction of that work. It was designed for teaching structured programming. It was small and simple with nothing really new. Because of its simplicity and size it was, for almost two decades, the most widely used language for teaching programming in colleges.

C – 1972 – High-level system programming language

C was designed for systems programming at Bell Labs by Dennis Richie. It evolved primarily from B, but was also influenced by ALGOL 68. It had powerful set of operators, but poor type checking. It initially spread through UNIX but became very popular in the academic circles because of its easy portability. Many modern programming languages including C++, Java, and C# are direct descendents of C.

Prolog – 1972 – Logic Programming

It was developed at the University of Aix-Marseille, by Comerauer and Roussel, with some help from Kowalski at the University of Edinburgh. It is based on formal logic. It is a non-procedural declarative programming language with built-in backtracking mechanism. It has support for associative memory and pattern directed procedure invocation. After LISP, it is the second most widely used language in AI community, especially in Europe.

Ada – 1983 – History's largest design effort

It involved a huge design effort, involving hundreds of people, much money, and about eight years. It introduced Packages - support for data abstraction, elaborate exception handling, generic program units, and concurrency through the tasking model.

It was the outcome of a competitive design effort. It included all that was then known about software engineering and language design. Because of its size and complexity, first

compilers were very difficult and the first really usable compiler came nearly five years after the language design was completed

Smalltalk - 1972-1980 – The Purest Object-Oriented Langauge

It was developed at Xerox PARC, initially by Alan Kay and then later by Adele Goldberg. It is the first full implementation of an object-oriented language (data abstraction, inheritance, and dynamic type binding) and the purest object-oriented language yet! It pioneered the graphical user interface everyone now uses.

C++ - 1985

It was developed at Bell Labs by Stroustrup. It evolved from C and SIMULA 67. Facilities for object-oriented programming, taken partially from SIMULA 67, were added to C. It also has exception handling. It is a large and complex language, in part because it supports both procedural and OO programming. Rapidly grew in popularity, along with OOP and ANSI standard was approved in November, 1997.

Java - 1995

Java was developed at Sun in the early 1990s and is based on C++. It is significantly simplified as compared to C++ and supports only OOP. It eliminated multiple inheritance, pointers, structs, enum types, operator overloading, and goto statement and added support for applets and a form of concurrency.

C# - 2002

It is part of the .NET framework by Microsoft. It is based upon C++ and Java and support component-based software development. It has taken some ideas from Visual Basic. It brought back pointers, structs, enum types, operator overloading, and goto statement. It has safer enum types, more useful struct types, and modified switch statement.

Scripting Languages for Web

These languages were designed to address the need for computations associated with HTML documents. JavaScript and PHP are two representative languages in this domain.

JavaScript – client side scripting

Primary objective of JavaScript is to create dynamic HTML documents and check validity of input forms. It is usually embedded in an HTML document. It is not really related to Java

PHP (Personal Home Page) – server-side scripting
It is interpreted on the Web Server when the HTML document in which embedded is requested by the browser. It often produces HTML code as an output and is very similar

to JavaScript. It allows simple access to HTML form data and makes form processing easy. It also provides support for many different database management systems and hence provides Web access to databases.

Programming Language Evolution

Over the last five decades, programming languages have evolved after going through several phases. These phases are briefly described below:

- **1950's – Discovery and description of programming language concepts**

  Programming language design in this period took an empirical approach. Programming languages were regarded solely as tools for facilitating the specification of programs rather than as interesting objects of study in their own right. In this era we saw development of symbolic assembly language, macro-assembly, FORTRAN, Algol 60, COBOL, and LISP. Many of the basic implementation techniques were discovered which include symbol table construction and look-up, stack algorithms for evaluating arithmetic expressions, activation record stack, and marking algorithms for garbage collection.

- **1960's – Analysis and elaboration**

  This was the era when programming language design took a mathematical approach. Here we saw theoretical research as an end in itself and a lot of analysis was carried out for the purpose of constructing models and theories of programming languages. Representative languages of this era include PL/1, Simula, Algol 68, and Snobol. These languages elaborated the earlier languages and attempted to achieve greater richness by synthesis of existing features and generalization. This resulted in greater complexity. This was the time when formal languages and automata theory with application to parsing and compiler theory as well as theory of operational and mathematical semantics were defined. There was a lot of emphasis on program correctness and verification.

- **1970's – Effective software technology**

  Decreasing hardware cost and increasing software cost resulted in more complex software requiring support for software engineering in the programming languages. It also required development of tools and methodologies for controlling the complexities, cost and reliability of large programs Therefore, languages designed in this period had software engineering support in the form of structured design in the form of structured programming, modular design, and verification. It saw a transition from pure research to practical management of the environment. Verifiable languages such as Pascal and Modula were developed as a result of this effort.

- **1980's – Support for SE continued**
- **1990's and 2000's**
  - Support for OOP and Internet
- **2010's – Aspect-Oriented Programming**

Language Design Perspectives

Programming Language design is not a simple task. A designer has to consider the need for many different stake holders and balance their requirements. For example, software developers ask for more features and theoreticians ask for features with clean semantics. On the other hand developers of mission-critical software want features that are easier to verify formally. Compiler writers want features that are orthogonal, so that their implementation is modular. These goals are often in conflict and hence it is difficult to satisfy all of them.

From a language definition perspective, once again different users have different needs. Programmers need tutorials, reference manuals, and programming guides (idioms). Implementers demand precise operational semantics and verifiers require rigorous axiomatic or natural semantics. Language designers and language advocates want all of the above. This means that a language needs to be defined at different levels of detail and precision but none of them can be sloppy!