# Programare procedurală – M2 Fundamentele limbajelor de programare

Grigore Albeanu
http://www.researcherid.com/rid/H-4522-2011

# Agenda

- Evolutia si clasificarea limbajelor de programare
- Specificarea sintaxei unui limbaj de programare.
- Unitati de program
- Compilarea unitatilor de program
- Editarea de legaturi
- Executarea programelor
- Interpretoare
- Medii integrate de dezvoltare

# Clasificarea si evolutia limbajelor de programare

- Limbaje de programare
- Limbaje native / limbaje masina
- Limbaje de asamblare
- Limbaje de nivel inalt
- Generatii de limbaje si paradigme de programare
- Evolutia limbajelor de programare

# Limbaje de programare

- Un **limbaj de programare** este o *notatie sistematica* prin care este descris un proces de calcul.
- Un **proces de calcul** este reprezentat de succesiunea de comenzi elementare (pe care un calculator le poate executa) asociate algoritmului de rezolvare a unei probleme. Un proces de calcul utilizeaza setul de comenzi (instructiuni) ale calculatorului (procesorului, microprocesorului).
- Limbajele de programare sunt limbaje artificiale.

# Limbaje native / limbaje masina

- Limbajul nativ al unui calculator este constituit din multimea codurilor instructiunilor acceptate pentru executare (recunoscute).
- Se foloseste si termenul de “limbaj masina”.
- Limbaje masina actuale: X86, MIPS, MMIX, etc.
- Tipuri de masini: RISC (Reduced Instruction Set Computer), CISC (Complex Instruction Set Computer)
- Puterea masinii: Clasele de instructiuni si mecanismele de procesare implementate, formatul si lungimea instructiunilor, timpul pentru executarea fiecarei instructiuni.
- Undocumented OpCodes: http://www.rcollins.org/secrets/IntelSecrets.html
- Masini virtuale: http://en.wikipedia.org/wiki/Virtual_machine
- The Java Virtual Machine Instruction Set http://java.sun.com/docs/books/jvms/second_edition/html/Instructions.doc.html

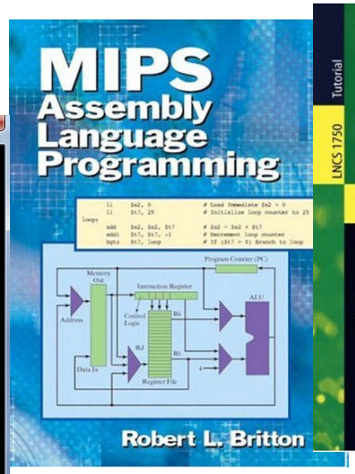
# Limbaje de asamblare

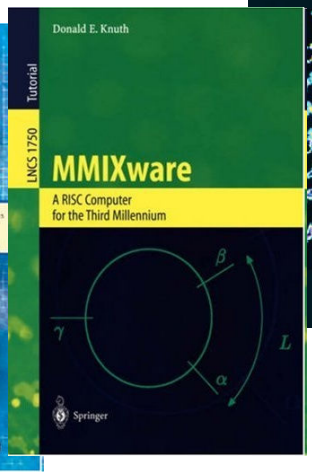
- Un limbaj de asamblare este un limbaj artificial care atribuie fiecarui OpCode al unei masini reale, un nume simbolic (mnemonica), fiecarui registru al masini de calcul, un mod de referire, iar pentru organizarea procesului de calcul descompune entitatile necesare in sectiuni (cod, date, extra, etc.) In plus are un mecanism de referire prin adrese simbolice (etichete).
- Un limbaj de macroasamblare adauga limbajului de asamblare posibilitatea de a descrie bucati de cod parametrizate, care in timpul asamblarii se extind (expandeaza) pentru a genera instructiunile corespunzatoare in limbaj de asamblare. In plus, limbajul de asamblare include *directive* de compilare conditionata si alte modalitati care usureaza munca programatorilor la nivel

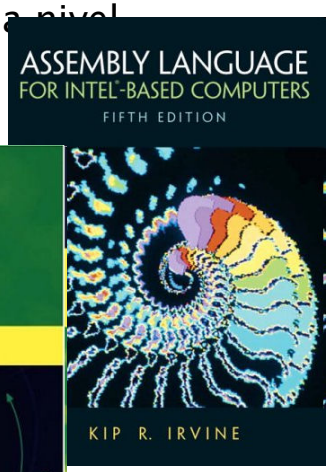
```
1796:0100  B8 79 0F 8E D8 B4 09 B8-34 12 BB 78 56 50 53 58   .y......4..xVPSX
1796:0110  5B 00 00 00 00 00 00 00-00 00 00 00 34 00 85 17   [...........4...
```

```
1796:0100 B8790F          MOV     AX,0F79
1796:0103 8ED8            MOV     DS,AX
1796:0105 B409            MOV     AH,09
1796:0107 B83412          MOV     AX,1234
1796:010A BB7856          MOV     BX,5678
```

```
Command Prompt - debug
-?
assemble    A [address]
compare     C range address
dump        D [range]
enter       E address [list]
fill        F range list
go          G [=address] [addresses]
hex         H value1 value2
input       I port
load        L [address] [drive] [firstsector] [number]
move        M range address
name        N [pathname] [arglist]
output      O port byte
proceed     P [=address] [number]
quit        Q
register    R [register]
search      S range list
trace       T [=address] [value]
unassemble  U [range]
write       W [address] [drive] [firstsector] [number]
allocate expanded memory        XA [#pages]
deallocate expanded memory      XD [handle]
map expanded memory pages       XM [Lpage] [Ppage] [handle]
display expanded memory status  XS
-_
```



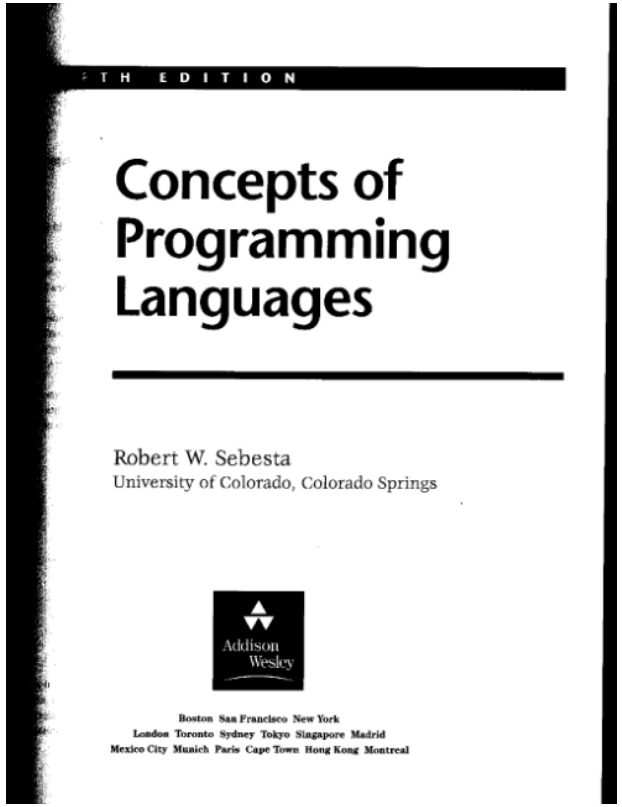
# Limbaje de nivel inalt

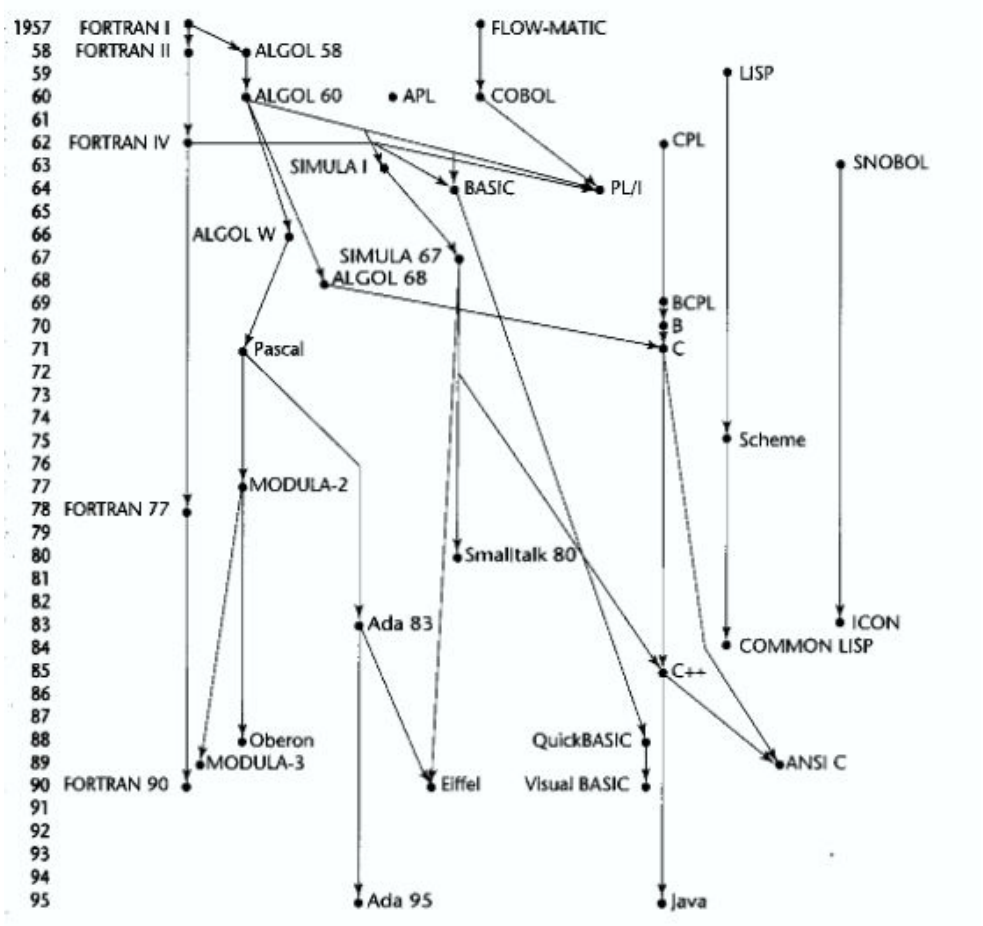
- Un limbaj de nivel inalt cuprinde mecanisme de exprimare apropiate de limbajul natural. Foloseste verbe pentru a desemna actiuni (do-executa, repeat-repeta, read-citeste, write-scrie, continue, switch-comuta, call, goto, etc.), conjunctii (if-daca, while – cât timp, etc.), adverbe (then-atunci, else-altfel, etc.), mecanisme de declare si definire, dar ofera suport si pentru importul/exportul de module ( pachete, sub-proiecte, etc.).
- Un limbaj de programare promoveaza un model de calcul (masina von Neumann, calcul paralel, etc.), permite definirea de tipuri de date (type – Pascal, class –C++/Java) si operatii (supraincarcarea operatorilor in C++), ofera facilitati de abstractizare (tipuri abstracte de date – templates: STL, etc.), etc.
- Un limbaj de programare trebuie sa aiba o descriere sintactica si semantica bine definita (exista modele formale care permit analiza sintactica si semantica a programelor scrise in limbaje de programare), este fiabil (descurajeaza greselile de programare – nu este cazul limbajelor C/C++; Java este un limbaj fiabil), permite traducerea rapida (viteza ridicata in faza de compilare si generarea codului – limbajul D special creat pentru compilare rapida), trebuie sa fie independent de procesor (pentru asigurarea portabilitatii codului), trebuie sa fie independent de sistemul de operare (pentru a permite realizarea de software multiplatforma), trebuie sa fie demonstrabil (sa se poata verifica corectitudinea programelor) etc.

# Generatii de limbaje si paradigme de programare

- 1954-1958: prima generatie (FORTRAN I, ALGOL58)
- 1959-1961: a doua generatie (ALGOL60, FORTRAN II, COBOL, LISP)
- 1962-1971: a treia generatie (PL/1, ALGOL68, Pascal, C, Simula)
- 1972-1979: a patra generatie (Ada, Smalltalk)
- 1980-in prezent: paradigme ale limbajelor de programare (limbaje “functionale”, “logice”, orientate obiect si distribuite).

# Genealogia limbajelor de programare de nivel inalt



# FORTRAN – ALGOL

The following is an example of a FORTRAN 90 program:

```
C  FORTRAN 90 EXAMPLE PROGRAM
C  INPUT:  AN INTEGER, LIST_LEN, WHERE LIST_LEN IS LESS
C         THAN 100, FOLLOWED BY LIST_LEN-INTEGER VALUES
C  OUTPUT: THE NUMBER OF INPUT VALUES THAT ARE GREATER
C         THAN THE AVERAGE OF ALL INPUT VALUES
      INTEGER INTLIST(99)
      INTEGER LIST_LEN, COUNTER, SUM, AVERAGE, RESULT
      RESULT = 0
      SUM = 0
      READ *, LIST_LEN
      IF ((LIST_LEN .GT. 0) .AND.
          (LIST_LEN .LT. 100)) THEN
C  READ INPUT DATA INTO AN ARRAY AND COMPUTE ITS SUM
        DO 10 COUNTER = 1, LIST_LEN
          READ *, INTLIST(COUNTER)
          SUM = SUM + INTLIST(COUNTER)
10      CONTINUE
C  COMPUTE THE AVERAGE
        AVERAGE = SUM / LIST_LEN
C  COUNT THE VALUES THAT ARE GREATER THAN THE AVERAGE
        DO 20 COUNTER = 1, LIST_LEN
          IF (INTLIST(COUNTER) .GT. AVERAGE) THEN
            RESULT = RESULT + 1
          END IF
20      CONTINUE
C  PRINT THE RESULT
        PRINT *, 'NUMBER OF VALUES > AVERAGE IS:', RESULT
      ELSE
        PRINT *, 'ERROR-LIST LENGTH VALUE IS NOT LEGAL'
      END IF
      STOP
      END
```

The following is an example of an ALGOL 60 program:

```
comment ALGOL 60 Example Program
 Input:  An integer, listlen, where listlen is less than
         100, followed by listlen-integer values
 Output: The number of input values that are greater than
         the average of all the input values  ;
begin
  integer array intlist [1:99];
  integer listlen, counter, sum, average, result;
  sum := 0;
  result := 0;
  readint (listlen);
  if (listlen > 0) ∧ (listlen < 100) then
    begin
comment Read input into an array and compute the average;
    for counter := 1 step 1 until listlen do
      begin
      readint (intlist[counter]);
      sum := sum + intlist[counter]
      end;
comment Compute the average;
    average := sum / listlen;
comment Count the input values that are > average;
    for counter := 1 step 1 until listlen do
      if intlist[counter] > average
        then result := result + 1;
comment Print result;
    printstring("The number of values > average is:");
    printint (result)
    end
  else
    printstring ("Error-input list length is not legal");
end
```

# COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PRODUCE-REORDER-LISTING.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. DEC-VAX.
OBJECT-COMPUTER. DEC-VAX.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT BAL-FWD-FILE    ASSIGN TO READER.
    SELECT REORDER-LISTING  ASSIGN TO LOCAL-PRINTER.

DATA DIVISION.
FILE SECTION.
FD  BAL-FWD-FILE
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 80 CHARACTERS.

01  BAL-FWD-CARD.
    02 BAL-ITEM-NO          PICTURE IS 9(5).
    02 BAL-ITEM-DESC        PICTURE IS X(20).
    02 FILLER               PICTURE IS X(5).
    02 BAL-UNIT-PRICE       PICTURE IS 999V99.
    02 BAL-REORDER-POINT    PICTURE IS 9(5).
    02 BAL-ON-HAND          PICTURE IS 9(5).
    02 BAL-ON-ORDER         PICTURE IS 9(5).
    02 FILLER               PICTURE IS X(30).

FD  REORDER-LISTING
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 132 CHARACTERS.

01  REORDER-LINE.
    02 RL-ITEM-NO           PICTURE IS Z(5).
    02 FILLER               PICTURE IS X(5).
    02 RL-ITEM-DESC         PICTURE IS X(20).
    02 FILLER               PICTURE IS X(5).
    02 RL-UNIT-PRICE        PICTURE IS ZZZ.99.
    02 FILLER               PICTURE IS X(5).
    02 RL-AVAILABLE-STOCK   PICTURE IS Z(5).
    02 FILLER               PICTURE IS X(5).
    02 RL-REORDER-POINT     PICTURE IS Z(5).
    02 FILLER               PICTURE IS X(71).

WORKING-STORAGE SECTION.
01  SWITCHES.
    02 CARD-EOF-SWITCH      PICTURE IS X.
01  WORK-FIELDS.
    02 AVAILABLE-STOCK      PICTURE IS 9(5).

PROCEDURE DIVISION.
000-PRODUCE-REORDER-LISTING.
    OPEN INPUT BAL-FWD-FILE.
    OPEN OUTPUT REORDER-LISTING.
    MOVE "N" TO CARD-EOF-SWITCH.
    PERFORM 100-PRODUCE-REORDER-LINE
        UNTIL CARD-EOF-SWITCH IS EQUAL TO "Y".
    CLOSE BAL-FWD-FILE.
    CLOSE REORDER-LISTING.
    STOP RUN.

100-PRODUCE-REORDER-LINE.
    PERFORM 110-READ-INVENTORY-RECORD.
    IF CARD-EOF-SWITCH IS NOT EQUAL TO "Y"
        PERFORM 120-CALCULATE-AVAILABLE-STOCK
        IF AVAILABLE-STOCK IS LESS THAN BAL-REORDER-POINT
            PERFORM 130-PRINT-REORDER-LINE.

110-READ-INVENTORY-RECORD.
    READ BAL-FWD-FILE RECORD
        AT END
            MOVE "Y" TO CARD-EOF-SWITCH.

120-CALCULATE-AVAILABLE-STOCK.
    ADD BAL-ON-HAND BAL-ON-ORDER
        GIVING AVAILABLE-STOCK.

130-PRINT-REORDER-LINE.
    MOVE SPACE             TO REORDER-LINE.
    MOVE BAL-ITEM-NO       TO RL-ITEM-NO.
    MOVE BAL-ITEM-DESC     TO RL-ITEM-DESC.
    MOVE BAL-UNIT-PRICE    TO RL-UNIT-PRICE.
    MOVE AVAILABLE-STOCK   TO RL-AVAILABLE-STOCK.
    MOVE BAL-REORDER-POINT TO RL-REORDER-POINT.
    WRITE REORDER-LINE.
```

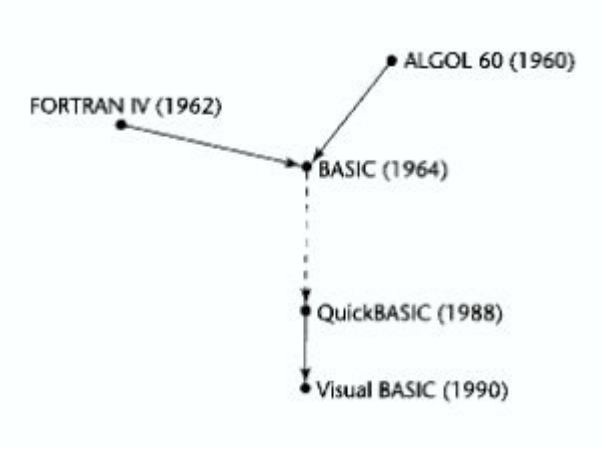
# BASIC (1964)

The most probable reasons for BASIC's success are the ease with which it can be learned and the ease with which it can be implemented, even on very small computers.

Two of the contemporary versions of BASIC that are now being widely used are QuickBASIC (Bradley, 1989) and Visual BASIC. Both of these run on PCs. Visual BASIC is based on QuickBASIC but is designed for developing software systems that have windowed user interfaces. Visual BASIC has also been used as a scripting language for CGI programming.

The following is an example of a QuickBASIC program:

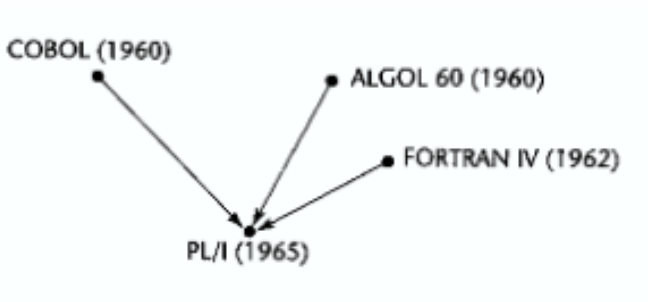
```
REM  QuickBASIC Example Program
REM  Input:  An integer, listlen, where listlen is less
REM          than 100, followed by listlen-integer values
REM  Output: The number of input values that are greater
REM          than the average of all input values
  DIM intlist(99)
  result = 0
  sum = 0
  INPUT listlen
  IF listlen > 0 AND listlen < 100 THEN
REM  Read input into an array and compute the sum
    FOR counter = 1 TO listlen
      INPUT intlist(counter)
      sum = sum + intlist(counter)
    NEXT counter
REM  Compute the average
    average = sum / listlen
REM  Count the number of input values that are > average
    FOR counter = 1 TO listlen
      IF intlist(counter) > average
        THEN result = result + 1
    NEXT counter
REM  Print the result
    PRINT "The number of values that are > average is:";
          result
  ELSE
    PRINT "Error-input list length is not legal"
  END IF
END
```



# PL/I (1965)

The following is an example of a PL/I program:

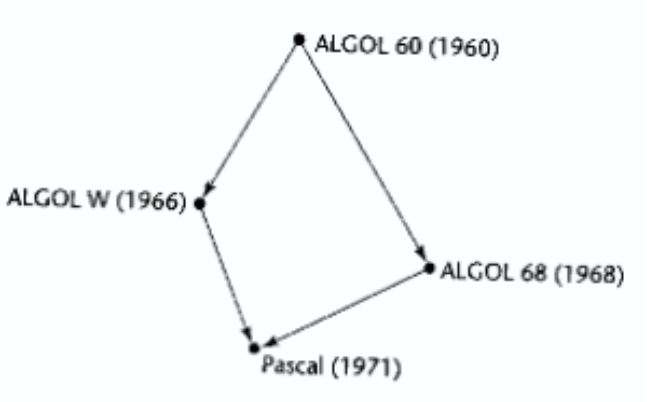
```
/* PL/I PROGRAM EXAMPLE
 INPUT:  AN INTEGER, LISTLEN, WHERE LISTLEN IS LESS THAN
         100, FOLLOWED BY LISTLEN-INTEGER VALUES
 OUTPUT: THE NUMBER OF INPUT VALUES THAT ARE GREATER THAN
         THE AVERAGE OF ALL INPUT VALUES   */
PLIEX: PROCEDURE OPTIONS (MAIN);
  DECLARE INTLIST (1:99) FIXED.
  DECLARE (LISTLEN, COUNTER, SUM, AVERAGE, RESULT) FIXED;
  SUM = 0;
  RESULT = 0;
  GET LIST (LISTLEN);
  IF (LISTLEN > 0) & (LISTLEN < 100) THEN
    DO;
/* READ INPUT DATA INTO AN ARRAY AND COMPUTE THE SUM */
    DO COUNTER = 1 TO LISTLEN;
      GET LIST (INTLIST (COUNTER));
      SUM = SUM + INTLIST (COUNTER);
    END;
/* COMPUTE THE AVERAGE */
    AVERAGE = SUM / LISTLEN;
/* COUNT THE NUMBER OF VALUES THAT ARE > AVERAGE */
    DO COUNTER = 1 TO LISTLEN;
      IF INTLIST (COUNTER) > AVERAGE THEN
        RESULT = RESULT + 1;
    END;
/* PRINT RESULT */
    PUT SKIP LIST ('THE NUMBER OF VALUES > AVERAGE IS:');
    PUT LIST (RESULT);
    END;
  ELSE
    PUT SKIP LIST ('ERROR-INPUT LIST LENGTH IS ILLEGAL');
  END PLIEX;
```



# Pascal (1971)

The following is an example of a Pascal program:

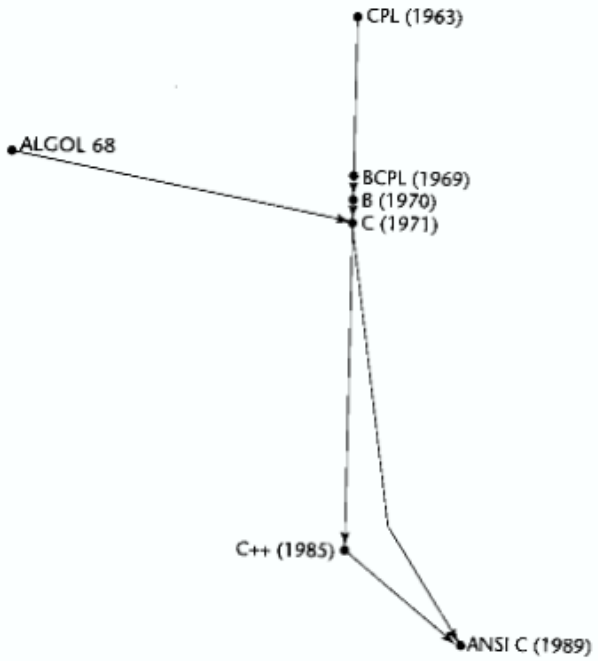
```
{Pascal Example Program
 Input:  An integer, listlen, where listlen is less than
         100, followed by listlen-integer values
 Output: The number of input values that are greater than
         the average of all input values }
program pasex (input, output);
  type intlisttype = array [1..99] of integer;
  var
    intlist : intlisttype;
    listlen, counter, sum, average, result : integer;
  begin
  result := 0;
  sum := 0;
  readln (listlen);
  if ((listlen > 0) and (listlen < 100)) then
    begin
{ Read input into an array and compute the sum }
    for counter := 1 to listlen do
      begin
      readln (intlist[counter]);
      sum := sum + intlist[counter]
      end;
{ Compute the average }
    average := sum / listlen;
{ Count the number of input values that are > average }
    for counter := 1 to listlen do
      if (intlist[counter] > average) then
        result := result + 1;
{ Print the result }
    writeln ('The number of values > average is:',
             result)
    end { of the then clause of if (( listlen > 0 ... }
  else
    writeln ('Error-input list length is not legal')
end.
```



# C (1971)

The following is an example of a C program:

```
/* C Example Program
 Input:  An integer, listlen, where listlen is less than
         100, followed by listlen-integer values
 Output: The number of input values that are greater than
         the average of all input values */
void main (){
  int intlist[98], listlen, counter, sum, average, result;
  sum = 0;
  result = 0;
  scanf("%d", &listlen);
  if ((listlen > 0) && (listlen < 100)) {
/* Read input into an array and compute the sum */
    for (counter = 0; counter < listlen; counter++) {
      scanf("%d", &intlist[counter]);
      sum = sum + intlist[counter];
     }
/* Compute the average */
    average = sum / listlen;
/* Count the input values that are > average */
    for (counter = 0; counter < listlen; counter++)
      if (intlist[counter] > average) result++;
/* Print result */
    printf("Number of values > average is:%d\n", result);
   }
  else
    printf("Error—input list length is not legal\n");
 }
```

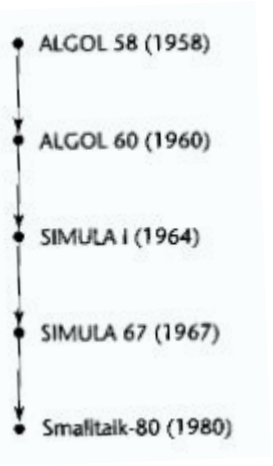


# Ada (1983)

```
-- Ada Example Program
-- Input:  An integer, LIST_LEN, where LIST_LEN is less
--         than 100, followed by LIST_LEN-integer values
-- Output: The number of input values that are greater
--         than the average of all input values
with TEXT_IO; use TEXT_IO;
procedure ADA_EX is
  package INT_IO is new INTEGER_IO (INTEGER);
  use INT_IO;
  type INT_LIST_TYPE is array (1..99) of INTEGER;
  INT_LIST : INT_LIST_TYPE;
  LIST_LEN, SUM, AVERAGE, RESULT : INTEGER;
  begin
  RESULT := 0;
  SUM := 0;
  GET (LIST_LEN);
  if (LIST_LEN > 0) and (LIST_LEN < 100) then
-- Read input data into an array and compute the sum
    for COUNTER := 1 .. LIST_LEN loop
      GET (INT_LIST(COUNTER));
      SUM := SUM + INT_LIST(COUNTER);
    end loop;
-- Compute the average
    AVERAGE := SUM / LIST_LEN;
-- Count the number of values that are > average
    for COUNTER := 1 .. LIST_LEN loop
      if INT_LIST(COUNTER) > AVERAGE then.
        RESULT := RESULT + 1;
      end if;
    end loop;
-- Print result
    PUT ("The number of values > average is:");
    PUT (RESULT);
    NEW_LINE;
  else
    PUT_LINE ("Error-input list length is not legal");
  end if;
end ADA_EX;
```

# SmallTalk (1980)

```
"Smalltalk Example Program"
"The following is a class definition, instantiations of which
can draw equilateral polygons of any number of sides"
class name                      Polygon
superclass                      Object
instance variable names         ourPen
                                numSides
                                sideLength

"Class methods"
  "Create an instance"
  new
     ^ super new getPen

  "Get a pen for drawing polygons"
  getPen
     ourPen <- Pen new defaultNib: 2

  "Instance methods"
  "Draw a polygon"
  draw
     numSides timesRepeat: [ourPen go: sideLength;
                            turn: 360 // numSides]

  "Set length of sides"
  length: len
     sideLength <- len

  "Set number of sides"
  sides: num
     numSides <- num
```



# Paradigme de programare

A: PARADIGMA PROGRAMARII PROCEDURALE SI STRUCTURATE :
Un program este privit ca o multime ierarhica de blocuri si proceduri;
B: PARADIGMA PROGRAMARII ORIENTATE SPRE OBIECT: Un program este constituit dintr-o colectie de obiecte care interactioneaza;
C: PARADIGMA PROGRAMARII CONCURENTE SI DISTRIBUITE:
Executia unui program este constituita din actiuni multiple posibil a fi executate in paralel pe una sau mai multe masini;
D: PARADIGMA PROGRAMARII FUNCTIONALE: Un program este descris pe baza unor functii de tip matematic (fara efecte secundare), utilizate de obicei recursiv;
E: PARADIGMA PROGRAMARII LOGICE: Un program este descris printr-un set de relatii intre obiecte precum si de restrictii ce definesc cadrul in care functioneaza acele obiecte. Executia inseamna activarea unui proces deductiv.
F: PARADIGMA PROGRAMARII LA NIVELUL BAZELOR DE DATE:
Actiunile programului sunt dictate de cerintele unei gestiuni corecte si consistente a bazelor de date asupra carora actioneaza programul.

# Evolutia limbajelor de programare

- FORTRAN – primul limbaj de programare de nivel inalt (IBM, FORTRAN IV–1966, FORTRAN77, FORTAN90, FORTAN95, PGI–Visual FORTRAN – suport pentru calcul paralel – High Performance Computing –2008: http://www.pgroup.com/)
- ALGOL (primul limbaj pentru care sintaxa a fost descrisa "matematic" – notatia (E)BNF) , ultima versiune fiind ALGOL68.
- COBOL (Departamentul Apararii al SUA, Common Business Oriented Language – 1959, COBOL81, Visual COBOL: http://visualcobol.net/)
- LISP (MIT–1959, LISt Processing, notatia Lambda, programare functionala) – Aplicatii ale inteligentei artificiale – sisteme expert
- PROLOG (Univ. Marsilia, 1972, Programare logica) – Aplicatii ale inteligentei artificiale – sisteme expert
- C++ (B. Stroustrup, 1984/5)
- SuperPascal (Pascal paralel), Pascal concurent, Modula, Oberon – limbaje care au introdus "multitasking" la nivelul limbajului.
- Java (SUN, 1996): complet orientat obiect, multithreading, networking, distributed programming, web programming (applets, servlets, …)
- C# (Microsoft–1999): http://download.microsoft.com/download/a/9/e/a9e229b9-fee5-4c3e-8476-917dee385062/CSharp%20Language%20Specification%20v1.0.doc
- Limbajul D (2007: http://en.wikipedia.org/wiki/D_%28programming_language%29; http://www.digitalmars.com/d/)
- Python (1999 – http://www.python.org/) Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI) in the Netherlands as a successor of a language called ABC.

In cadrul cursului nostru ne vom referi la C, cu toate detaliile (pointeri, vulnerabilitati software, functii cu numar variabil de argumente, etc.)

- C (1972): C is a general-purpose computer programming language developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories for use with the Unix operating system

# Specificarea sintaxei unui limbaj de programare (1)

## Notatia Backus-Naur (BNF si EBNF- exemplificare)

S := '-' FN | FN

FN := DL | DL '.' DL

DL := D | D DL

D := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Notatia EBNF:

S := '-'? D+ ('.' D+)?

D := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

? – optional; * - repetabilitate de cel putin zero ori; + - repetabilitate cel putin odata.

## Sintaxa C in descriere BNF (acoperire partiala - 1)

```
<storage-class-specifier> ::= auto
                            | register
                            | static
                            | extern
                            | typedef
```

```
<type-specifier> ::= void
                   | char
                   | short
                   | int
                   | long
                   | float
                   | double
                   | signed
                   | unsigned
                   | <struct-or-union-specifier>
                   | <enum-specifier>
                   | <typedef-name>
```

```
<struct-or-union> ::= struct
                    | union
```

```
<struct-or-union-specifier> ::= <struct-or-union> <identifier> { (<struct-declaration>)+ }
                              | <struct-or-union> { (<struct-declaration>)+ }
                              | <struct-or-union> <identifier>
```

```
<additive-expression> ::= <multiplicative-expression>
                        | <additive-expression> + <multiplicative-expression>
                        | <additive-expression> - <multiplicative-expression>

<multiplicative-expression> ::= <cast-expression>
                              | <multiplicative-expression> * <cast-expression>
                              | <multiplicative-expression> / <cast-expression>
                              | <multiplicative-expression> % <cast-expression>
```

http://www.cs.grinnell.edu/~stone/courses/languages/C-syntax.xhtml

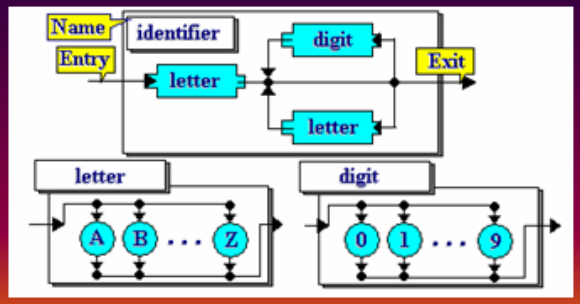
# Specificarea sintaxei unui limbaj de programare (2)

## Sintaxa C in descriere BNF (acoperire partiala - 2)

```
<compound-statement> ::= { (<declaration>)* (<statement>)* }

<statement> ::= <labeled-statement>
              | <expression-statement>
              | <compound-statement>
              | <selection-statement>
              | <iteration-statement>
              | <jump-statement>

<labeled-statement> ::= <identifier> : <statement>
                      | case <constant-expression> : <statement>
                      | default : <statement>

<expression-statement> ::= (<expression>)? ;

<selection-statement> ::= if ( <expression> ) <statement>
                        | if ( <expression> ) <statement> else <statement>
                        | switch ( <expression> ) <statement>

<iteration-statement> ::= while ( <expression> ) <statement>
                        | do <statement> while ( <expression> ) ;
                        | for ( (<expression>)? ; (<expression>)? ; (<expression>)? ) <statement>

<jump-statement> ::= goto <identifier> ;
                   | continue ;
                   | break ;
                   | return (<expression>)? ;
```

## Diagrame sintactice

Reprezentari grafice ale entitatilor sintactice (folosite, de obicei, in descrierea limbajului Pascal)




Specificarea identificatorilor Pascal

# Specificarea sintaxei unui limbaj de programare (3)

Gramatici formale (ierarhia Chomsky, expresii regulate, parsing)

## Gramatica instructiunilor C

statement -> labeled-statement | expression-statement | compound-statement | selection-statement | iteration-statement | jump-statement | declaration-statement

labeled-statement -> identifier : statement | **case** constant-expression : statement | **default** : statement

expression-statement -> expressionopt ;

compound-statement -> **{** statement-seqopt **}**

statement-seq -> statement | statement-seq statement

selection-statement -> **if (** condition **)** statement | **if (** condition **)** statement **else** statement | **switch (** condition **)** statement

condition -> expression | type-specifier-seq declarator = assignment-expression

iteration-statement -> **while (** condition **)** statement | **do** statement **while (** expression **) ;** | **for (** for-init-statement conditionopt ; expressionopt ) statement

for-init-statement -> expression-statement | simple-declaration

jump-statement -> **break ;** | **continue ;** | **return** expressionopt ; | **goto** identifier ;

declaration-statement -> block-declaration

# Unitati de program

- Structura generala a unui program scris intr-un limbaj de programare procedurala presupune existenta unui modul principal (functia main, clasa aplicatie cu metoda main) si existenta a zero, unul sau mai multe module (functii/proceduri, metode ale clasei aplicatie) care comunica intre ele si/sau cu modulul principal prin intermediul parametrilor si/sau a unor variabile globale (sau campuri ale clasei!)
- Orice unitate de program sau unitate de translatare apare intr-un fisier separat si poate conduce in urma compilarii la unul sau mai multe fisiere in cod intermediar (obiect, bytecode, etc.)
- In programarea procedurala, unitatea de program cea mai mica si care contine cod este functia/procedura. Aceasta contine partea de declaratii/definitii si partea imperativa (comenzile care se vor executa)
- Avantaje: compilare separata, reutilizarea codului, lucrul in echipa, lucrul la distanta, testarea/verificarea codului – Unit testing

# Compilarea unitatilor de program

- Analiza textului sursa (analiza lexicala – produce sir de atomi lexicali, analiza sintactica – produce arbori sintactici, analiza semantica – produce codul intermediar)
- Sinteza codului obiect (optimizarea codului – produce cod intermediar optimizat, generarea de cod – produce codul obiect final)
- In toate fazele se gestioneaza tabele (structuri de date specifice – functii hash pentru cautare rapida) si se utilizeaza mecanisme de raportarea erorilor.
- Codul intermediar poate fi: absolut (direct executabil), relocabil (editare de legaturi, translatarea adreselor), in limbaj de asamblare, un alt limbaj de programare (cross-compilers)

# Editarea de legaturi

- Mai multe module obiect (bucati de cod generate separat) se asambleaza impreuna cu module din bibliotecile standard pentru crearea aplicatiei finale.
- Aplificatia finala este obtinuta prin segmentare (overlay) sau cu ajutorul bibliotecilor dinamice (dll)
- Build = Compilare + Editare de legaturi

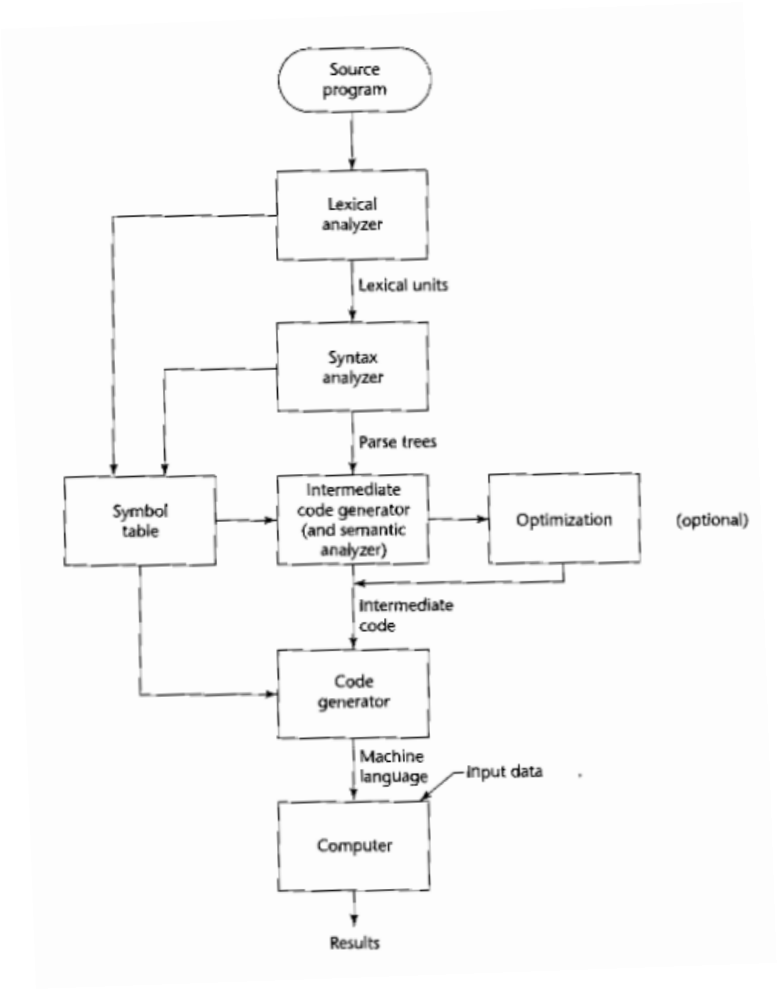
# Executarea programelor

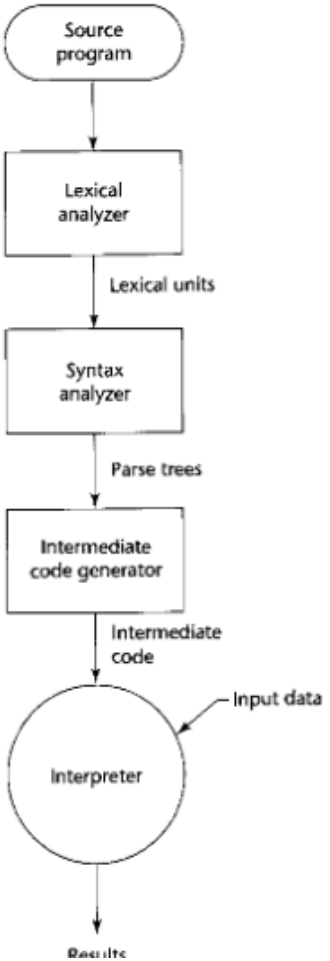
- Executarea programelor se realizeaza in urma pregatirii pentru executare de catre sistemul de operare.
- Executarea poate fi intrerupta (cazul sistemelor multitasking primitiv) de catre utilizator sau de catre sistemul de operare (sisteme multitasking pentru mai multi utilizatori) si poate fi reluata pe baza unei strategii de planificarea lucrarilor (prioritati, round robin, etc.)
- Deoarece functia main (in C/C++) returneaza un rezultat aceasta face ca mai multe programe sa poata fi apelate dintr-un program “panou de comanda” pentru operare.

# Interpretare

- Interpretarea programelor inseamna executarea pe calculator direct a programului sursa (LISP, Prolog, Basic, MMIX, Python) sau a codului intermediar (Java).
- Interpretarea scade viteza de procesare deoarece intr-o prima faza se realizeaza analiza lexicala, analiza sintactica si cea semantica si numai apoi se trece la executarea instructiunii.
- Avantajul codului intermediar consta in eliminarea fazelor de analiza, dar este necesara verificarea consistentei codului (sa nu fi fost deteriorat – corrupted code)
- Interpretorul Python: http://docs.securityorg.net/python.pdf

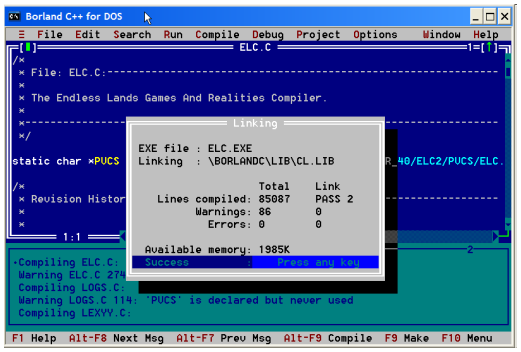
# Compilare – Interpretare

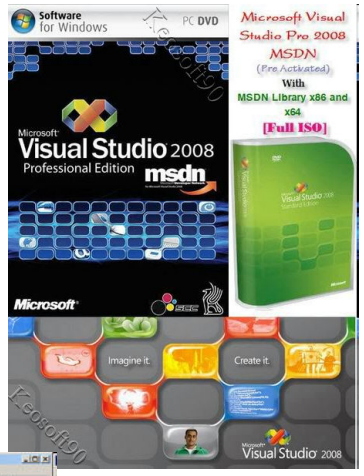


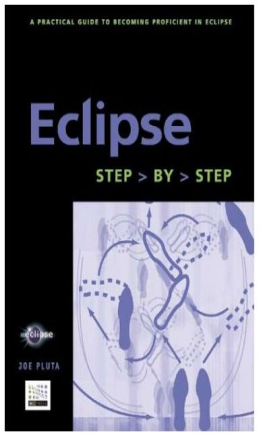



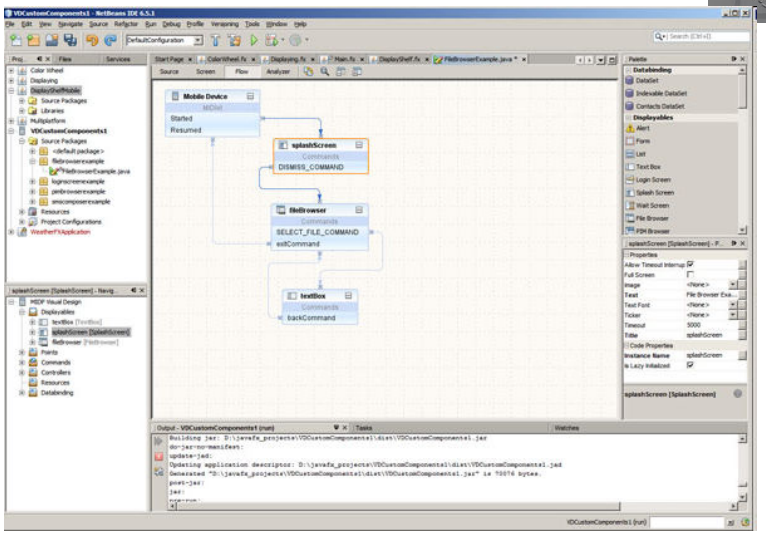
# Medii integrate de dezvoltare

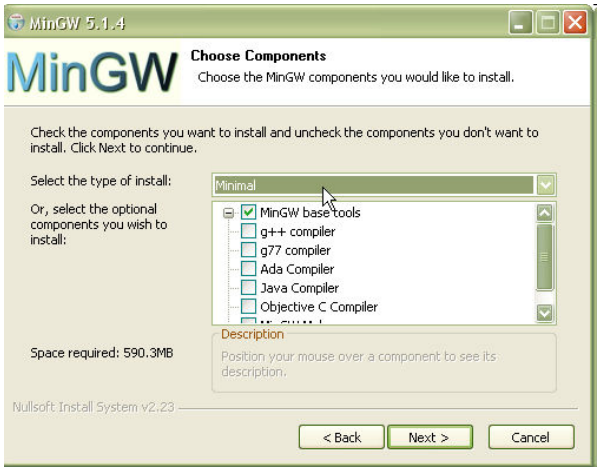
- Borland
- Microsoft
- IBM
- Sun
- Alte



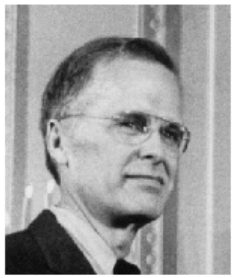



# Who's Who – Konrad Zuse


http://user.cs.tu-berlin.de/~zuse/Konrad_Zuse/index.html

# John Backus

- http://www.betanews.com/article/John-W-Backus-1924-2007/1174424012
- http://www-03.ibm.com/ibm/history/exhibits/builders/builders_backus.html



John Backus

**Achievement**

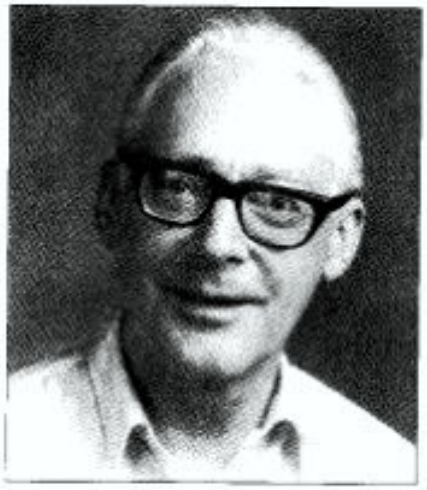
As projectleader with IBM John Backus developed in the early 1950's with his team: Fortran - Formula Translator. Fortran was released in 1954. The first high level programming language. This language is most widely used in physics and engineering.

He was also responsible for the Backus-Naur Form (or BNF), a standard notation which can be used to decribe the sytanx of a computer language in a formal and unambiguous way.

# Grace Hopper (1906 – 1992)

Grace M. Hopper, a naval officer and formerly employed by UNIVAC, designed a series of "compiler" systems in the early to mid-50s that were used for business applications programming. By 1958, these systems evolved to the first high-level programming language for business applications, FLOW-MATIC, on which COBOL was, to a large degree, based. She was also involved with the COBOL design effort, serving as an advisor to the executive committee of CODASYL.

# Ole-Johan Dahl (1931 – 2002)
# Adele Goldberg (1945 – )
# Bjarne Stroustrup (1950 – )



**Ole-Johan Dahl**

Ole-Johan Dahl and Kristen Nygaard, then both at the Norwegian Computing Center, were primarily interested in computer simulation. Driven by their needs in simulation, they designed and implemented the simulation languages SIMULA in 1964, and SIMULA 67 in 1967.

**Adele Goldberg**

Adele Goldberg spent 14 years at Xerox's Palo Alto Research Center, leading the design team for Smalltalk and its implementation. She was instrumental in the development not only of Smalltalk, but of the window- and icon-based user interface paradigm.

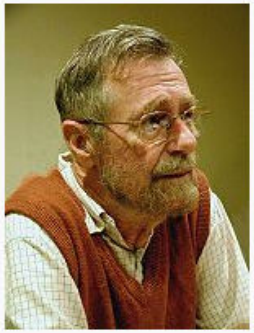
**Bjarne Stroustrup**

Bjarne Stroustrup, a member of the Technical Staff at AT&T Bell Labs since 1979, designed C++. C++ made object-oriented programming accessible and affordable to real-world software developers. Largely due to the popularity of C++, object-oriented programming became the pervasive development paradigm by the early 1990s. Stroustrup is the author of *The C++ Programming Language* and *The Design and Evolution of C++*.

http://www.research.att.com/~bs/homepage.html

# Edsger Dijkstra (1930 – 2002)
# Niklaus Wirth (1934 – )

**Edsger Wybe Dijkstra**



| | |
|---|---|
| **Born** | May 11, 1930<br>Rotterdam, Netherlands |
| **Died** | August 6, 2002 (aged 72)<br>Nuenen, Netherlands |
| **Fields** | Computer science |
| **Institutions** | Mathematisch Centrum<br>Eindhoven University of Technology<br>The University of Texas at Austin |
| **Doctoral advisor** | Adriaan van Wijngaarden |
| **Doctoral students** | Nico Habermann<br>Martin Rem<br>David Naumann<br>Cornelis Hemerik<br>Jan Tijmen Udding<br>Johannes van de Snepscheut<br>Antonetta van Gasteren |
| **Known for** | Dijkstra's algorithm<br>Structured programming<br>THE multiprogramming system<br>Semaphore |
| **Notable awards** | Turing Award<br>Association for Computing |

**Niklaus E. Wirth**

| | |
|---|---|
| **Born** | February 15, 1934 (age 75)<br>Winterthur, Switzerland |
| **Citizenship** | Switzerland |
| **Fields** | Computer Science |
| **Institutions** | ETH Zurich,<br>Stanford University,<br>University of Zurich<br>Xerox PARC |
| **Alma mater** | ETH Zurich,<br>Université Laval,<br>University of California, Berkeley |
| **Known for** | Algol W, Euler, Modula, Modula-2,<br>Oberon, Pascal |
| **Notable awards** | Turing Award |

- http://en.wikipedia.org/wiki/Niklaus_Wirth
- http://en.wikipedia.org/wiki/Edsger_W._Dijkstra

# Dennis Ritchie (1941–)
# Ken Thomson (1943 –)

**Dennis Ritchie**

Dennis Ritchie of Bell Laboratories was on of the principals involved with the development of UNIX. He was the designer of the first version of C, which was then used to rewrite UNIX for PDP-11 computers.

- http://en.wikipedia.org/wiki/Dennis_Ritchie
- http://en.wikipedia.org/wiki/Ken_Thompson_%28programmer%29

# Richard Stallman (1953 – )

- GNU
- Free Software Foundation
- GNU Public Licence
- Copyleft
- http://www.fsf.org/blogs/rms/
- Go open: http://www.youtube.com/watch?v=P8BhOH9g_QE

MinGW
Minimalist GNU for Windows

http://sourceforge.net/projects/mingw/files/