

THE UNIVERSITY *of York*

High Performance Computing - Evolution of Computer Languages and Programming Paradigms

Prof Matt Probert

<http://www-users.york.ac.uk/~mijp1>

Overview

- History of some early languages
- Structured Programming
- More recent languages
- Object Orientated Programming
- Current status and relevance to HPC

Why study history?

- There are by some estimates over 2500 computer languages in existence
 - Most never used outside of the authors group
 - Why were they all created?
 - What do they have in common?
 - Why do we need any more than 1 language?

Pre-History

- Analogue computers were “programmed” by changing gears, etc.
- Earliest digital computers followed a similar paradigm, with manual setting of switches etc
- 1945: John Von Neumann developed two key concepts:
 - “Shared-program” technique – use complex instructions to control simple hardware rather than use complex hardware so can re-program
 - “conditional control transfer” – i.e. no longer just sequential operation
 - can branch or loop or use subroutines etc

Assembly

- Human readable notation for binary instructions understood by the CPU.
- Translated into machine code by a program called an *assembler*.
- Need to know the instruction set for the specific CPU in question.
- Imagine a world with no compilers.....

Assembly (P4)

```
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
movl    $0, -4(%ebp)
flds    a.449
flds    b.448
fmulp   %st, %st(1)
fstps   a.449
movl    -4(%ebp), %eax
leave
ret
```



Sets up
space for a
and b

←Load a into register st

←Load b into register st(1)

←Multiply st by st(1) and store in st

←Store contents of st into memory

**This is roughly 50% of the
code needed for $a := a*b$!**

See <https://godbolt.org/> for compiler output ...

In the Beginning ...

- 1949: first computer language – “Short Code” – created. Did not have a compiler – the programmer had to convert statements into binary by hand
- 1951: first compiler written by Grace Hopper – “A-0” – but still everything was very “low level”
- Computers were starting to become more popular but take-up hampered by difficulty of programming, and so ...

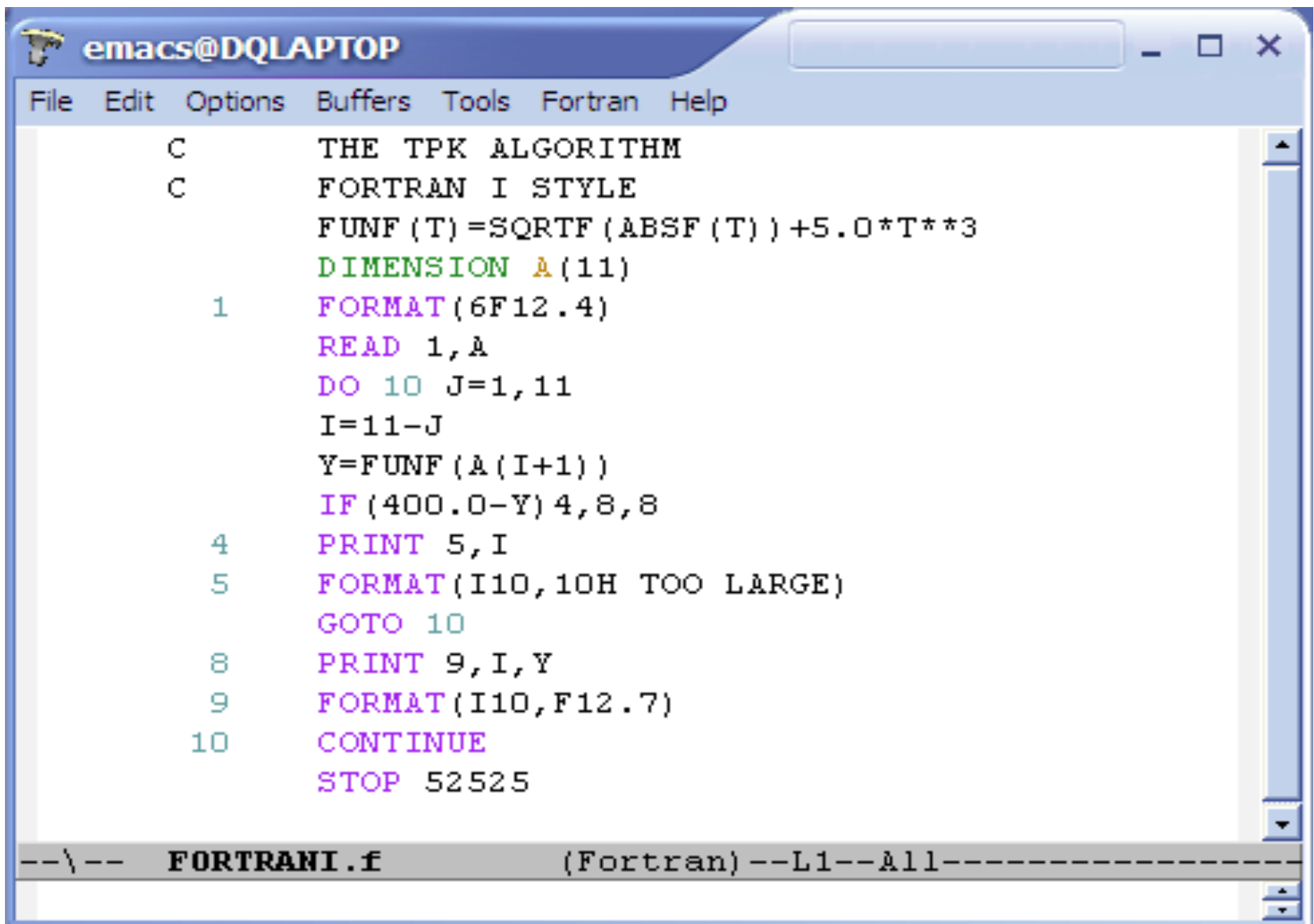
... was the Granddad of them all ...

- FORTRAN was created by IBM for scientific programming (John Backus was team leader)
 - First manual in 1956, first compiler in 1958
 - 18 man-years to write the first ever high-level compiler
- Why was it needed?
 - Programs were getting more complex and more error-prone as the uptake of computers increased
 - Programmers needed to focus on the problem they were trying to solve and not the tool they were using
 - Efficiency matters! One of the design goals was that the language should have at least 90% performance of hand-crafted machine code – often better!

... FORTRAN features ...

- A very simple language by modern standards – but revolutionary at the time – machine independent code!
 - IF, DO and GOTO statements (key von Neumann concepts)
 - Assignments looked like normal maths
 - Also introduced the idea of data types – logical, integer, real and double-precision numbers
 - FORTRAN I (1957), II (1958), IV (1961)
 - 66 (first ever language to be standardised)
 - 77 (significant changes)
 - 90 (major development), 95 (minor tweaks), 2003 (major tweaks), 2008 (minor tweaks), 2018 (major tweaks), etc.

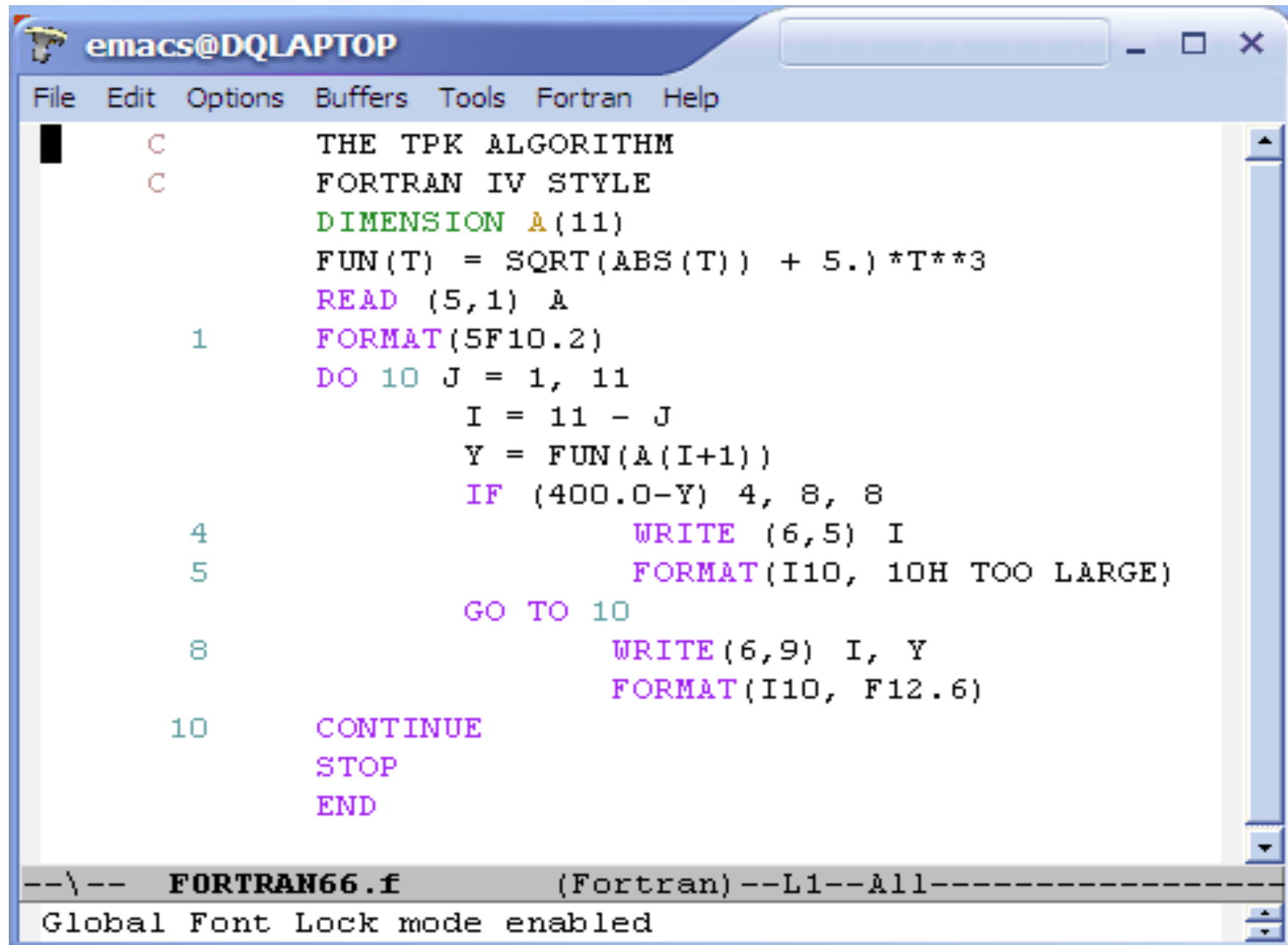
FORTRAN I



The image shows a screenshot of an Emacs editor window titled "emacs@DQLAPTOP". The window contains Fortran code for a program named "TPK ALGORITHM". The code is displayed in a monospaced font with syntax highlighting: comments are in black, function definitions in blue, dimension statements in green, and other code in purple. The code includes a function "FUNF", a loop from J=1 to 11, and various control statements like "FORMAT", "READ", "DO", "IF", "PRINT", "GOTO", and "STOP". The status bar at the bottom shows the file name "FORTRAN1.f" and the current line and column "L1--A11".

```
emacs@DQLAPTOP
File Edit Options Buffers Tools Fortran Help
C      THE TPK ALGORITHM
C      FORTRAN I STYLE
      FUNF (T) = SQRTF ( ABSF ( T ) ) + 5.0 * T ** 3
      DIMENSION A ( 11 )
1      FORMAT ( 6F12.4 )
      READ 1, A
      DO 10 J = 1, 11
      I = 11 - J
      Y = FUNF ( A ( I + 1 ) )
      IF ( 400.0 - Y ) 4, 8, 8
4      PRINT 5, I
5      FORMAT ( I10, 10H TOO LARGE )
      GOTO 10
8      PRINT 9, I, Y
9      FORMAT ( I10, F12.7 )
10     CONTINUE
      STOP 52525
--\--  FORTRAN1.f      (Fortran) --L1--A11-----
```

FORTRAN IV/66

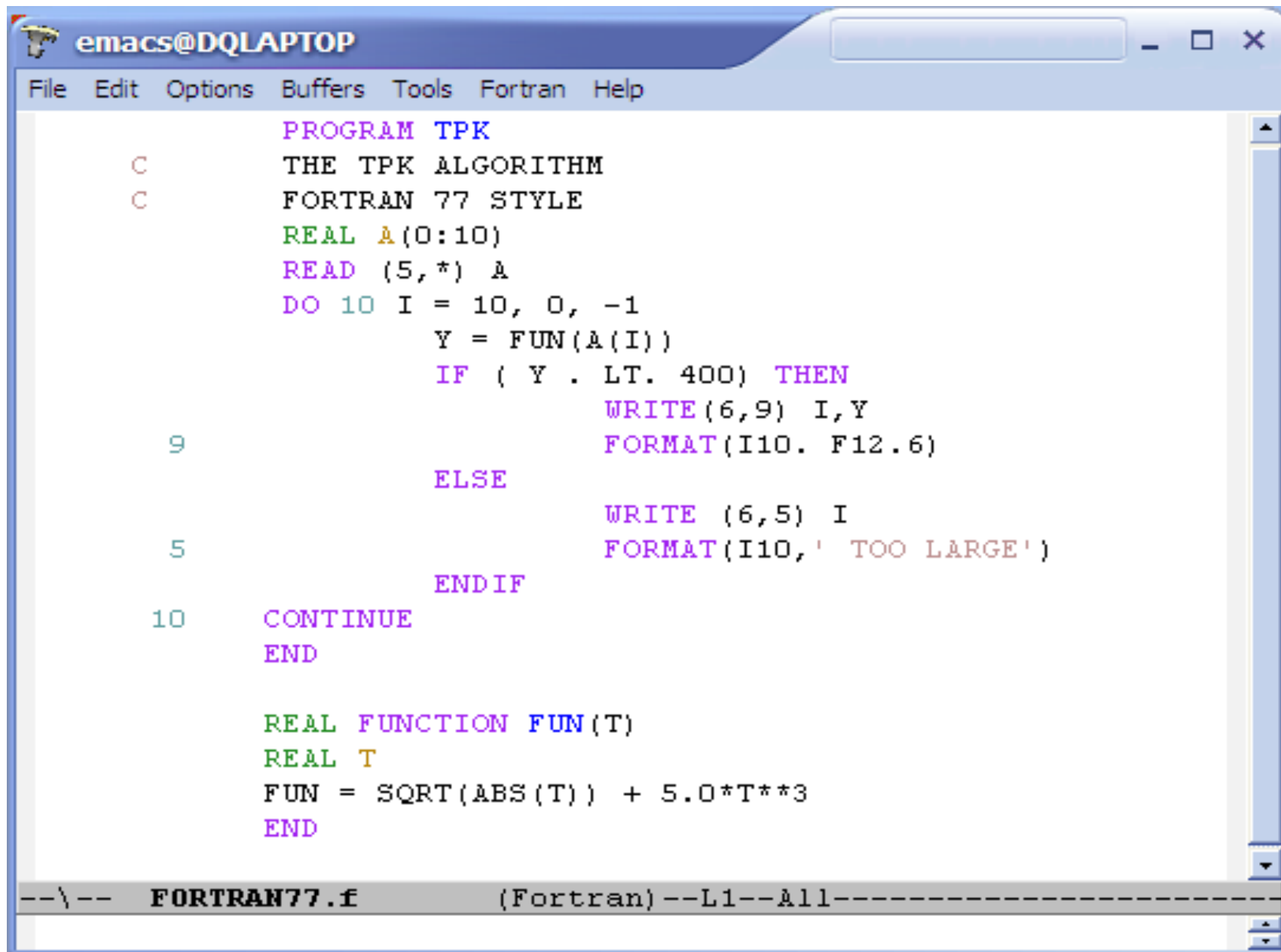


The image shows a screenshot of an Emacs editor window titled "emacs@DQLAPTOP". The window contains Fortran code for a program named "FORTRAN66.f". The code is displayed in a monospaced font with syntax highlighting: comments are in red, keywords like DIMENSION, READ, FORMAT, DO, GO TO, WRITE, and STOP are in purple, and the function definition FUN(T) is in green. The code implements a loop that calculates the square root of the absolute value of a number and then cubes it, with a check for values greater than 400. The status bar at the bottom shows the file name "FORTRAN66.f", the mode "(Fortran)", and the window configuration "--L1--All--". A message "Global Font Lock mode enabled" is also visible at the bottom.

```
emacs@DQLAPTOP
File Edit Options Buffers Tools Fortran Help
C THE TPK ALGORITHM
C FORTRAN IV STYLE
  DIMENSION A(11)
  FUN(T) = SQRT(ABS(T)) + 5.) * T**3
  READ (5,1) A
1  FORMAT(5F10.2)
  DO 10 J = 1, 11
      I = 11 - J
      Y = FUN(A(I+1))
      IF (400.0-Y) 4, 8, 8
4         WRITE (6,5) I
5         FORMAT(I10, 10H TOO LARGE)
      GO TO 10
8         WRITE (6,9) I, Y
          FORMAT(I10, F12.6)
10        CONTINUE
      STOP
      END

--\-- FORTRAN66.f (Fortran) --L1--All--
Global Font Lock mode enabled
```

FORTRAN 77



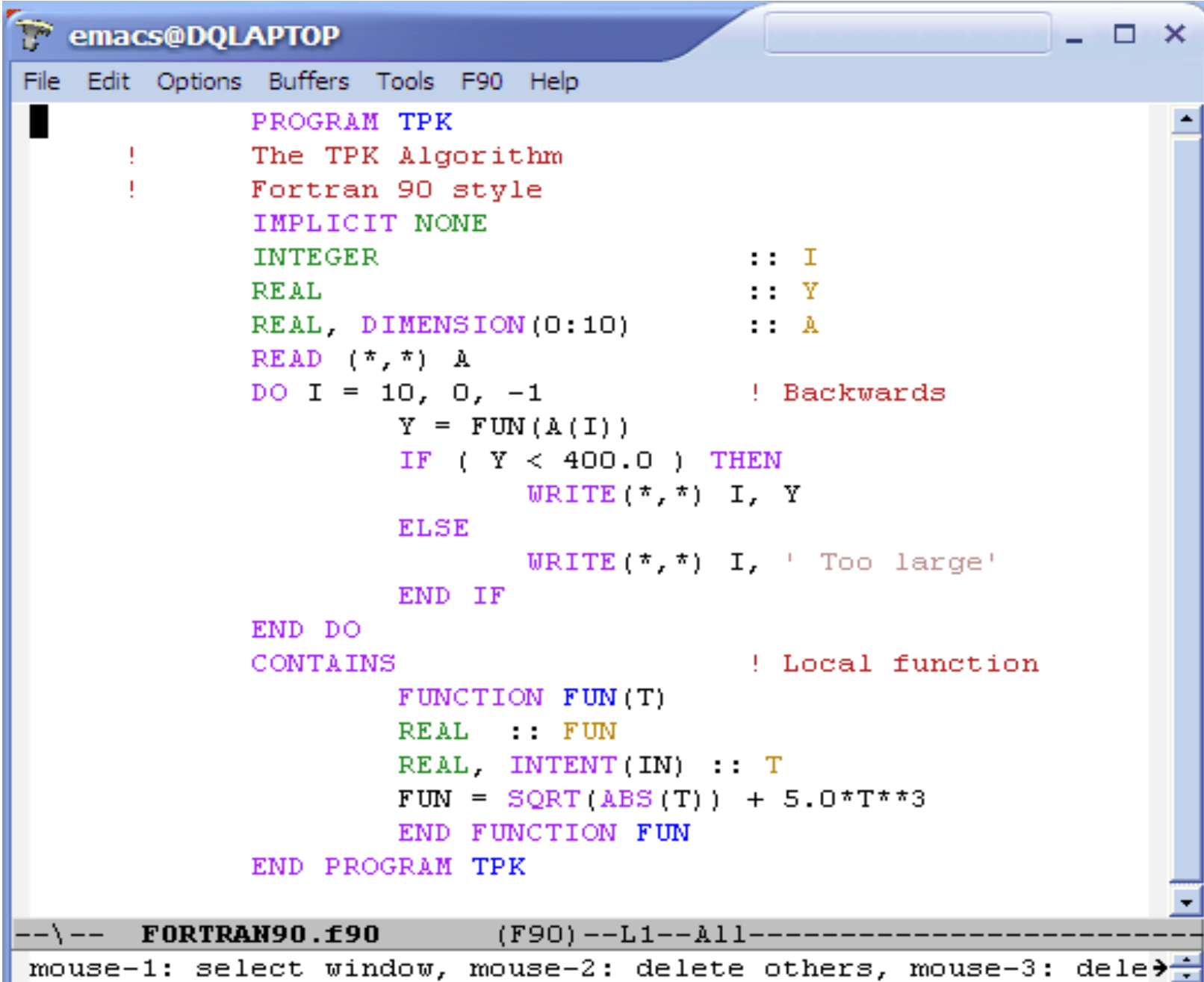
```
emacs@DQLAPTOP
File Edit Options Buffers Tools Fortran Help

PROGRAM TPK
C THE TPK ALGORITHM
C FORTRAN 77 STYLE
REAL A(0:10)
READ (5,*) A
DO 10 I = 10, 0, -1
    Y = FUN(A(I))
    IF ( Y . LT. 400) THEN
        WRITE(6,9) I,Y
        FORMAT(I10, F12.6)
9
    ELSE
        WRITE (6,5) I
        FORMAT(I10,' TOO LARGE')
5
    ENDIF
10 CONTINUE
END

REAL FUNCTION FUN(T)
REAL T
FUN = SQRT(ABS(T)) + 5.0*T**3
END

--\-- FORTRAN77.f (Fortran)--L1--All-----
```

Fortran 90

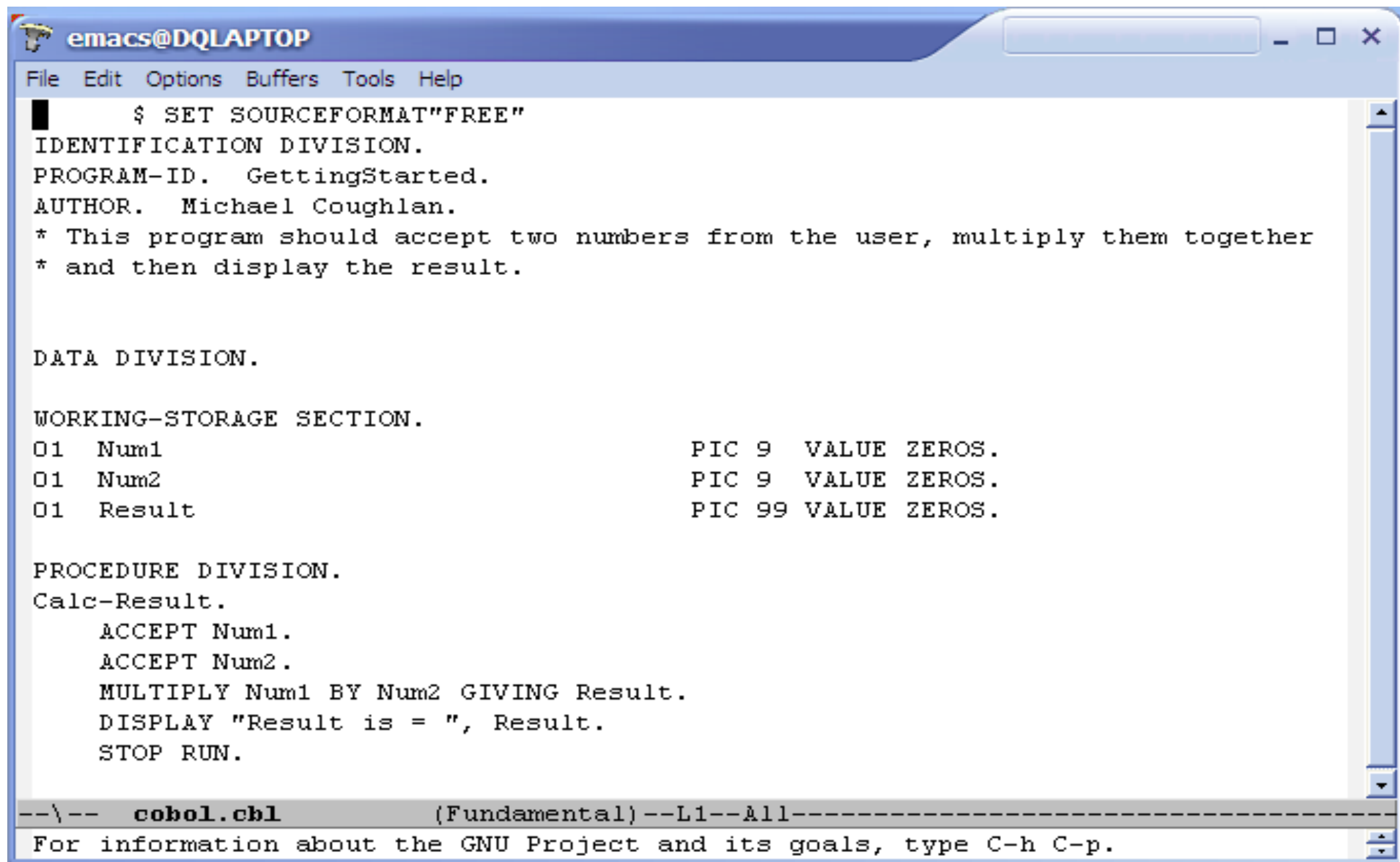


```
emacs@DQLAPTOP
File Edit Options Buffers Tools F90 Help
PROGRAM TPK
! The TPK Algorithm
! Fortran 90 style
IMPLICIT NONE
INTEGER :: I
REAL :: Y
REAL, DIMENSION(0:10) :: A
READ (*,*) A
DO I = 10, 0, -1 ! Backwards
    Y = FUN(A(I))
    IF ( Y < 400.0 ) THEN
        WRITE(*,*) I, Y
    ELSE
        WRITE(*,*) I, ' Too large'
    END IF
END DO
CONTAINS ! Local function
FUNCTION FUN(T)
REAL :: FUN
REAL, INTENT(IN) :: T
FUN = SQRT(ABS(T)) + 5.0*T**3
END FUNCTION FUN
END PROGRAM TPK
--\-- FORTRAN90.f90 (F90)--L1--All-----
mouse-1: select window, mouse-2: delete others, mouse-3: dele→
```

... rapidly followed by...

- COBOL in 1959
 - Common Business Oriented Language
 - Grace Hopper as team leader in creation (DoD)
 - FORTRAN was good at handling numbers but not at I/O which mattered for business
 - Only data types were numbers and strings
 - but these could be grouped into arrays and records for better data handling
 - Elegant, English-like syntax (long-winded!)
 - Still a major language in use today, but primarily for legacy code – Y2k bug

COBOL



```
emacs@DQLAPTOP
File Edit Options Buffers Tools Help
█      $ SET SOURCEFORMAT"FREE"
IDENTIFICATION DIVISION.
PROGRAM-ID.  GettingStarted.
AUTHOR.  Michael Coughlan.
* This program should accept two numbers from the user, multiply them together
* and then display the result.

DATA DIVISION.

WORKING-STORAGE SECTION.
01  Num1          PIC 9  VALUE ZEROS.
01  Num2          PIC 9  VALUE ZEROS.
01  Result        PIC 99 VALUE ZEROS.

PROCEDURE DIVISION.
Calc-Result.
  ACCEPT Num1.
  ACCEPT Num2.
  MULTIPLY Num1 BY Num2 GIVING Result.
  DISPLAY "Result is = ", Result.
  STOP RUN.

--\--  cobol.cbl          (Fundamental)--L1--All-----
For information about the GNU Project and its goals, type C-h C-p.
```

... and then there was ...

- **ALGOL**

- Created by a European-American committee in 1958
 - need perceived for standardisation and company independence
- First language to have a formal grammar
 - First “syntax-directed” compiler
- First actual ALGOL compiler in 1960
- Designed for publication (journals) as well as programming – self-documenting code
- Introduced block-structured programming, variable-length arrays, if-then-else, while-loops, and recursion!

... ALGOL in action ...

An Algol 60 program

The following Algol 60 program solves the problem of finding the mean of a list of numbers and how many numbers are greater than the mean.

```
begin
  comment this program finds the mean of n numbers
    and the number of values greater than the mean;
  integer n;
  read(n);
  begin
    real array a[1:n];
    integer i, number;
    real sum, mean;
    for i := 1 step 1 until n do
      read (a[i]);
    sum := 0.0;
    for i := 1 step 1 until n do
      sum := sum + a[i];
    mean := sum / n;
    number := 0;
    for i := 1 step 1 until n do
      if a[i] > mean then
        number := number + 1;
    write("MEAN = ", mean, "NUMBER OVER MEAN = ", number)
  end
end
```

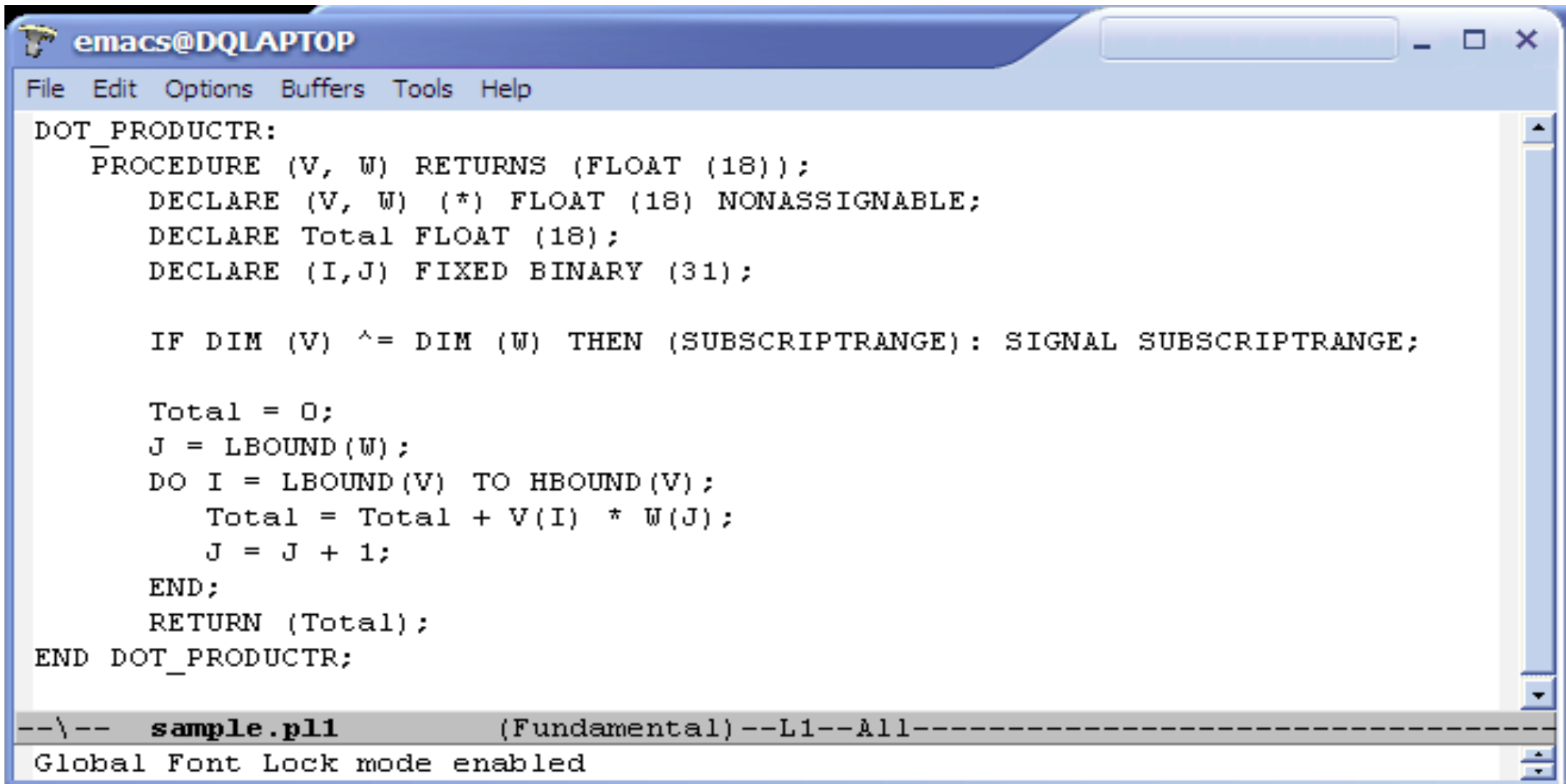
... but ALGOL dies ...

- So what happened to ALGOL?
 - Originally, IBM were involved, but as their business (and hence FORTRAN) grew, they sidelined ALGOL
 - FORTRAN was simpler and hence easier to compile and hence produced faster code and hence attracted more users
 - Nicklaus Wirth & others revised the language in 1966 – added case structures, records, pointers, complex and bitwise data types – resulting in ALGOL-W
 - Major language revision in 1968 but new version generally viewed as bloated and difficult to use – hence dies
 - ALGOL now seen as the root of many modern languages, including Pascal and C (and hence C++, Java, etc)

... PL/I ...

- Created by IBM in 1960
- A merger of ALGOL, FORTRAN and COBOL with some ideas from LISP!
- Designed as a general purpose language, to bring together business and science
- IBM wanted to drop FORTRAN and COBOL support and switch to PL/I but users protested and PL/I died.

PL/1



The image shows a screenshot of an Emacs window titled "emacs@DQLAPTOP". The window contains PL/1 code for a procedure named "DOT_PRODUCTR". The code declares variables V and W as FLOAT (18) NONASSIGNABLE, Total as FLOAT (18), and I and J as FIXED BINARY (31). It includes a check for dimension mismatch, a loop to calculate the dot product, and a return statement. The status bar at the bottom indicates the file is "sample.pl1" and "Global Font Lock mode enabled".

```
DOT_PRODUCTR:
  PROCEDURE (V, W) RETURNS (FLOAT (18));
  DECLARE (V, W) (*) FLOAT (18) NONASSIGNABLE;
  DECLARE Total FLOAT (18);
  DECLARE (I,J) FIXED BINARY (31);

  IF DIM (V) ^= DIM (W) THEN (SUBSCRIPTRANGE): SIGNAL SUBSCRIPTRANGE;

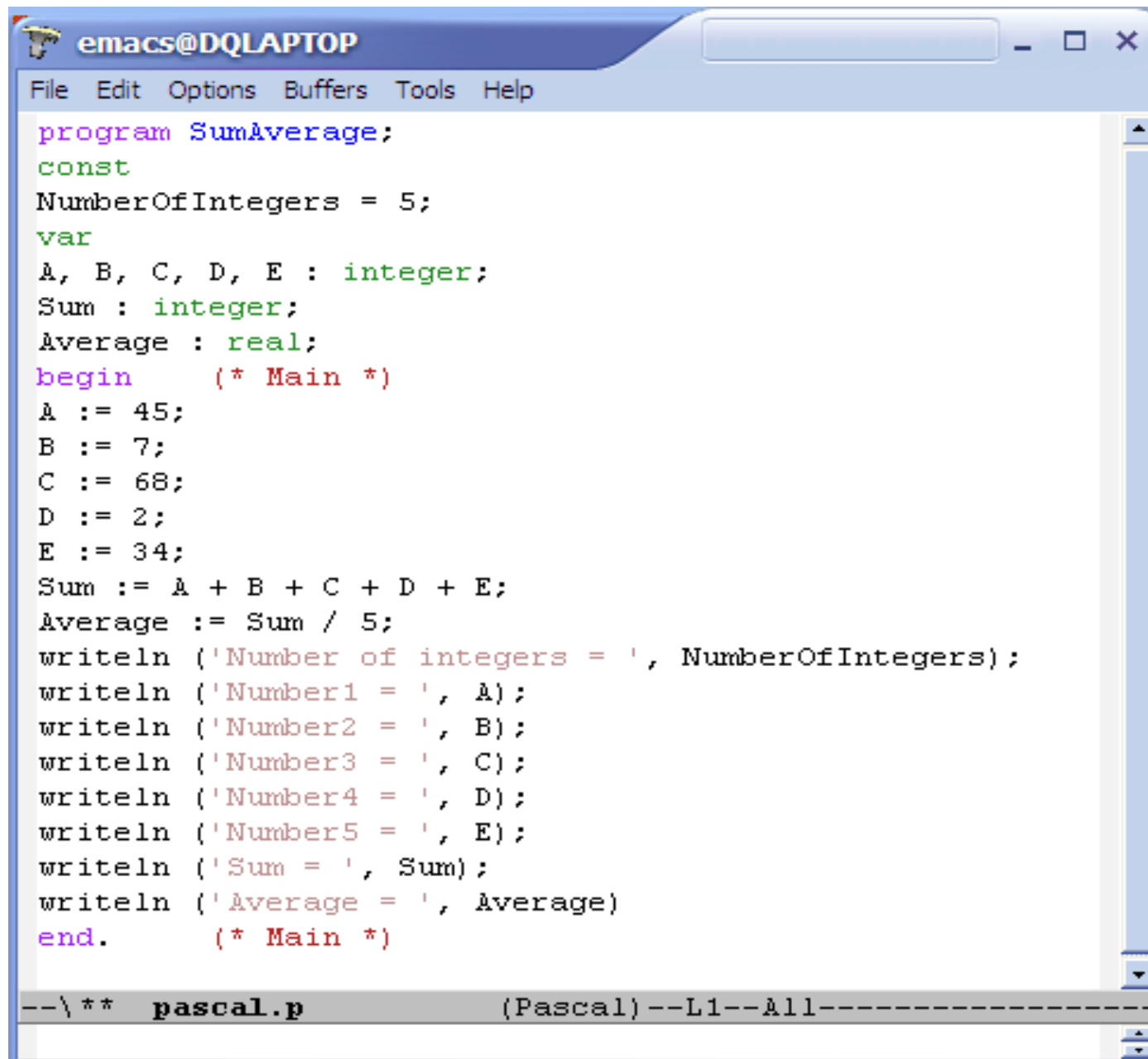
  Total = 0;
  J = LBOUND (W);
  DO I = LBOUND (V) TO HBOUND (V);
    Total = Total + V(I) * W(J);
    J = J + 1;
  END;
  RETURN (Total);
END DOT_PRODUCTR;
```

--\-- **sample.pl1** (Fundamental) --L1--All-----
Global Font Lock mode enabled

... and the end of the beginning

- 1968: Edsger Dijkstra wrote a article called "GOTO Statement Considered Harmful"
 - <https://doi.org/10.1145/362929.362947>
 - Controversial at the time – still argued in some circles!
 - Either way it created a debate
 - Started the move to “structured programming” as a means to develop reliable software
- Wirth creates Pascal in 1970 (based on ALGOL-W) as a teaching language
 - Smaller and more compact than ALGOL 68
 - Most popular language in University teaching in mid-80s
 - Very strict rules designed to teach good (structured) programming practice

PASCAL



The image shows a screenshot of an Emacs editor window titled 'emacs@DQLAPTOP'. The window contains a Pascal program named 'SumAverage.p'. The code is as follows:

```
program SumAverage;
const
NumberOfIntegers = 5;
var
A, B, C, D, E : integer;
Sum : integer;
Average : real;
begin      (* Main *)
A := 45;
B := 7;
C := 68;
D := 2;
E := 34;
Sum := A + B + C + D + E;
Average := Sum / 5;
writeln ('Number of integers = ', NumberOfIntegers);
writeln ('Number1 = ', A);
writeln ('Number2 = ', B);
writeln ('Number3 = ', C);
writeln ('Number4 = ', D);
writeln ('Number5 = ', E);
writeln ('Sum = ', Sum);
writeln ('Average = ', Average)
end.      (* Main *)
```

The status bar at the bottom of the window displays: --\ ** pascal.p (Pascal) --L1--All-----

Structured Programming I

- *“A study of program structure has revealed that programs can differ tremendously in their intellectual manageability. A number of rules have been discovered, violations of which will either seriously impair or totally destroy the intellectual manageability of the program....I now suggest that we confine ourselves to the design and implementation of intellectually manageable programs”* .-----E. Dijkstra (1972)
- First serious attempt to improve the standard of code construction – more than just “no GOTOs”

Structured Programming II

- Emphasis on “readability”
- Top-down design and construction
 - Break code into manageable subprograms
 - Create a hierarchy of modules/subroutines with each having a single point of entry and exit
- Limited scope of data and control structures
 - Use sensible data names and limit scope
 - No jumping in/out of control-structure/ module/ subroutine at anything other than single entry/exit point

Structured Programming III

Goal is to enable programs that are:

- Easy to write
 - Modular design, multiple programmers, subroutine reuse
- Easy to debug
 - Single process per procedure and no spaghetti!
- Easy to understand
 - Modular design and meaningful variable names
- Easy to change
 - Should be self-documenting if well written!

Middle-Ages

- Pascal was designed for teaching
 - Good combination of features, I/O and maths
 - Improved pointers
 - Added 'case' statements and dynamic variables via NEW and DISPOSE commands
 - But no dynamic arrays or structures
- And so the stage was set for C ...

Birth of C

- Developed in 1972 by Dennis Ritchie at Bell Labs in order to create the UNIX operating system (O/S)
 - Hence “lower level” language than FORTRAN, etc
 - Access to hardware and internal structures
 - Close links to UNIX – hence dynamic variables, multitasking, interrupt handling, forking, low-level I/O
 - Popular with system programmers and commercial software manufacturers
 - Many variants – not ANSI standardised until 1990

C Features

- Concept of O/S developed alongside hardware and computer languages – particularly to remove the need to write machine-specific I/O
- C + O/S enabled portability and hence C's success
 - most O/S now written in C or variants
- A small language (unlike Algol-68 etc) but with many libraries
 - Libraries contain machine-specific code to perform low-level tasks – hence only the library needs to be ported to a new machine – not the rest of the code
 - Enabled source code reuse and portability

C Problems

- Optimisation
 - Pointers make it very hard for compiler writers to generate efficient numerical code
- Lack of support for large projects
 - Everything is either globally visible or private to just one procedure
 - Similarly, data is either static or automatic – programmer has to manage memory and no automatic garbage collection

... software engineering fails ...

- Even though C was developed with support for structured programming it is easy to create unintelligible code
 - see <http://www.ioccc.org> for examples!
- Many large software projects were suffering from lack of code re-use and very high maintenance/ bug-fixing costs
- So a new programming paradigm was created ...

Any Guesses?

1988 Winner by Ian Phillipps ...

```
main(t,_,a ) char * a; { return! 0<t? t<3? main(-79,-13,a+
main(-87,1-_, main(-86, 0, a+1 ) +a)): 1, t<_? main( t+1, _, a
) :3, main ( -94, -27+t, a ) &&t == 2 ?_ <13 ? main ( 2, _+1,
"%s %d %d\n" ) :9:16: t<0? t<-72? main( _, t,
"@n'+,#'/*{ }w+/w#cdnr/+,{ }r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{1,+,/n
{n+,/+#n+,/#;#q#n+,/+k#;*+,/'r : 'd*'3,}{w+K w'K:'+}e#' ;dq#'l
q#' +d'K#!/+k#;q#'r}eKK#}w'r}eKK{n1]'/#;#q#n') { )#}w') { ) {n1]' /+#n
';d}rw' i;# ) {n1]!/n{n#'; r{#w'r nc{n1]' /#{1,+'K {rw'
iK{;[{n1]' /w#q#n'wk nw' iwk{KK{n1]!/w{% 'l##w#' i;
:{n1]' /*{q#'ld;r'}{n1wb!/*de}'c ;;{n1'-
}{rw]' /+,}##' * }#nc, ',#nw]' /+kd'+e}+;#'rdq#w! nr' / ' )
}+}{r1#'{n' ' )# }'+}##(!!/" ) : t<-50? _==*a ? putchar(31[a]):
main(-65,_,a+1) : main(*a == '/' ) + t, _, a + 1 ) : 0<t? main
( 2, 2 , "%s" ) :*a=='/' || main(0, main(-61,*a, "!ek;dc
i@bK' (q) - [w]*%n+r3#1, { }:\nuwloca-0;m .vpbks,fxntdCeghiry")
,a+1); }
```

Object Oriented Programming (OOP)

- A *class encapsulates* both the *data* types of a data structure and also the types of operations (*methods*) that can be performed with it:

```
class Point {  
    int _x, _y;  
    public:  
    void setX(const int val);  
    void setY(const int val);  
    int getX() { return _x; }  
    int getY() { return _y; }  
};
```

} data

} methods

- We can now create and manipulate objects which are instances of this class:

```
Point :: my_point;  
my_point::setX(1.0);  
my_point::setY(2.3);
```


OOP II

- OOP designed to enable code re-use
 - Once a class has been carefully created, it can be **reused** in many different projects – hence class libraries
 - If a new type of object is required, it can be descended from a parent class via inheritance

```
class Rectangle inherits from Point {
  attributes:
    int _width, // Width of rectangle
        _height; // Height of rectangle
  methods:
    setWidth(int newWidth)
    getWidth()
    setHeight(int newHeight)
    getHeight()
}
```

OOP III

- Data within a class can only be manipulated through a controlled interface – safe with predictable results.
- Some languages also support *polymorphism* where one piece of code works with many different classes of objects.
 - E.g. could extend point to circle class and have common interface to calculate area of circle and rectangle.
- OOP comes into its own in big projects with many developers. Not so useful for small codes we will write in this course!

OOP IV

- Many modern languages support OOP
 - Original language was Smalltalk (1979) but C begat C++ (1983), Basic begat Visual Basic (1991), Pascal begat Delphi (1995), etc. Even FORTRAN!
- Re-usability is not the same as portability, hence Sun begat Java (1995) for “write once, run anywhere” portability across the Net
 - MS begat C# (2000) for ??? reasons
- Fortran 90 has some OOP-like features, F2003 has more, F2018 even more ...
- OOP is a “bottom-up” not “top-down” style

Implementation

- **Compiled**
 - Most languages are compiled
 - Best for speed of resulting code
 - Time for compilation can be excessive
- **Interpreted**
 - No compilation time but slower running time
 - Interactive environments
- **Blurred boundaries**
 - Java is compiled into machine-independent byte code, which is then interpreted
 - PERL is compiled into a syntax tree, which is then interpreted

Language Generations

- 1st = Machine code
- 2nd = Assembler
- 3rd = Traditional languages (e.g. Fortran, C, Java, etc)
 - Language contains simple instructions
 - Can be very flexible but time consuming to code
- 4th = “Higher level” languages (e.g. application generators such as IDL, or Maple etc)
 - Highly complex & powerful instructions
 - Loss of flexibility and steep learning curve

Relevance to HPC?

- Hardware
 - Traditionally, optimal performance came from direct access to hardware, requiring proprietary language extensions and/or compiler directives, i.e. non-portable
- Modern approach = Standard languages + libraries
 - Use a familiar programming language (typically Fortran or C) with an open standard library
 - Standard API with custom implementations on different platforms i.e. a portable solution with small learning curve
 - *This is the approach we use in this course*

Requirements for HPC

- Speed – need compiled language.
- Portability – need standardised language.
- Libraries – need interfaces to MPI, BLAS LAPACK etc.
- Maturity – need a knowledge base (think – why did PL/1 fail?) and well developed optimising compilers.

Boils down to C/C++ or Fortran!

Fortran vs C / C++

Fortran

- Specifically designed for number crunching!
 - Useful intrinsic functions
- Limited object oriented
- Ignored outside HPC community
 - Uncommon outside academia.
- Restrictive
 - This is a good thing!

C / C++

- General purpose
 - Need libraries for nearly all maths.
- Object oriented
- Widely used
 - Transferable skill
 - Many free tools available
- Unrestrictive
 - Bad for HPC!

Why restrictive can be good

Fortran

```
integer,parameter :: N = 3
integer :: i

do i = 1,N
  <Insert some highly important
    maths>
end do
```

This loop will *always* execute exactly 3 times regardless of its contents – this information helps the compiler – see optimisation lectures.

C / C++

```
int const N = 3;
int i;

for (i = 1; i<4 ; i++) {
  <Insert some highly important
    maths>
}
```

The variable `i` can be modified within the loop. Compiler may not be able to figure out how many times loop executes – cannot optimise!

Further Reading

- “Computer Languages History” at <http://www.levenez.com/lang>
- Leslie B. Wilson and Robert G. Clark, “Comparative Programming Languages”, Addison-Wesley, 3rd ed. 2001
- “The History of Computing” http://www.thocp.net/software/languages/languages_index.htm
- The compiler explorer at <https://godbolt.org/>