Introduction

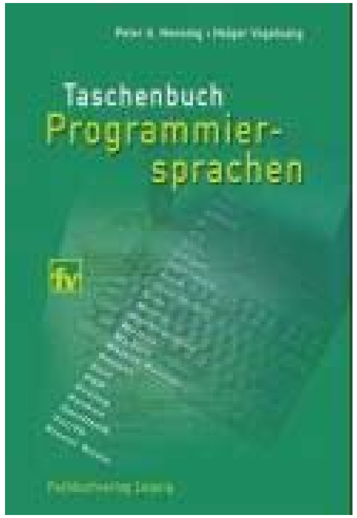
- History of Programming Languages
- The very first Language: FORTRAN

Organisation des Kurses

- 2 Lektionen Vorlesung
- 2 Lektionen Praktikum

- Unterlagen unter
  - www.zhaw.ch/~rege
- und in OLAT unter
  - https://olat.zhaw.ch/

- Leistungsnachweis
  - Praktika
    - *Abgabe 10%*
  - Zwischenprüfung
    - *Note 10%*
  - Semesterendprüfung 80%



Taschenbuch Programmiersprachen
von Peter Henning

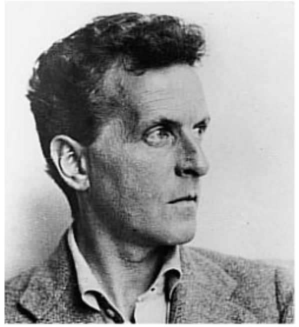
Semesterplan

Nr	Woche	Vorlesung	Praktikum	Script/Material
1 /	8	Geschichte/Einführung	Fortran	OLAT
2 /	9	R:Compiler 1	Ausdrücke Token.java Scanner.java Calculator.java	JVM Bytecodes bcel JavaDoc
3 /	10	R:Compiler 2	Eigene Programmiersprache Teil 1 CodeGen.java ICodeGenStart.java ILSample.java ILSample2.java	
4 /	11	R:Logische Progr.	Eigene Programmiersprache Teil 2 Token.java Scanner.java Program.java Musterlösung: ProgramSolution.java	Einführung in Logic Aussagenlogik
5 /	12	R: Prolog	Prolog Praktikum stammbaum.pl faecher.pl eliza.pl Musterlösung: faecherSolution.pl	Programme: SWI Prolog Mac Versions SWI Prolog Windows Version SWI Prolog Editor Windows Version Tutorials: Einführung in Prolog Prolog Introduction Adventure in Prolog Prolog by Examples
6 /	13	R: Smalltalk	Smalltalk Praktikum CanvasMorph.st	Smalltalk QuickRef Pharo Doku
7 /	14	R: Pascal Familie Modulkonzept	Modula Praktikum	Modula2 Reference Modula2 Handbook
8 /	15	Lisp 1	Lisp 1	
9 /	16	Lisp 2	Lisp 2	
10 /	17	Funktionale Programmierung 1	Lisp 3	
11 /	18	Funktionale Programmierung 2	Funktionale Programmierung	
12 /	19	Python 1	Python 1	
13 /	20	Python 2	Python 2	
14 /	21	Reserve		
	22,23	Prüfungsvorbereitung		
	24	Probeprüfung Prüfung		

Why Study (Programming-) Languages

- The purpose of language is simply that it must convey meaning. (Confucius, 551 - 479 BC)

- The limits of my language means the limits of my world. (Wittgenstein,1889-1951)



- Programming languages are important for students in all disciplines of computer science because they are the primary tools of the central activity of computer science : programming.

Why Study Programming Languages?

- To improve your ability to develop effective algorithms and to improve your use of your existing programming language.
  - e.g. O-O features, recursion
  - e.g. call by value, call by reference

- To increase your vocabulary of useful programming constructs.
- To allow a better choice of programming languages.
- To make it easier to learn a new language.

- very rarely to make it easier to design a new language.

Marjan Sirjani

The Early Days

Algorithm

- Abu Ja’far Muhammad ibn Musa al-Khorezmi
  - Lived in Baghdad around 780 – 850 AD
  - Chief mathematician in Khalif Al Mamun’s “House of Wisdom”
  - Adopted 1..9 from India and introduced 0
  - Author of “A Compact Introduction To Calculation Using Rules Of Completion And Reduction”

Removing negative units from the equation by adding the same quantity on the other side (“al-gabr” in Arabic)

Vitaly Shmatikov

Calculus of Thought

- Gottfried Wilhelm Leibniz
  - 1646 - 1716
  - Inventor of calculus and binary system
  - “Calculus Ratiocinator”: **human reasoning can be reduced to a formal symbolic language**, in which all arguments would be settled by mechanical manipulation of logical concepts

- Invented a mechanical calculator

Vitaly Shmatikov

Formalisms for Computation (1)

- Predicate logic
  - Gottlöb Frege (1848-1925)
  - Formal basis for proof theory and automated theorem proving
  - Logic programming
    - *Computation as logical deduction*

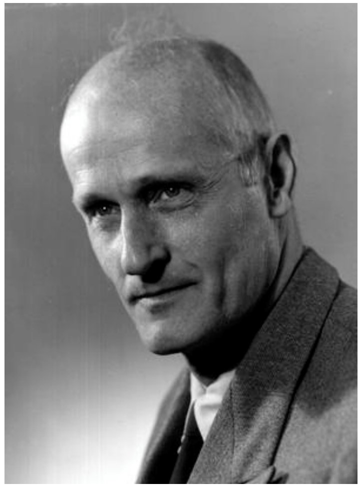
- Turing machines
  - Alan Turing (1912-1954)
  - Imperative programming
    - *Sequences of commands, explicit state transitions, update via assignment*

Vitaly Shmatikov

Formalisms for Computation (2)

- Lambda calculus
  - Alonzo Church (1903-1995)
  - Formal basis for all functional languages, semantics, type theory
  - Functional programming
    - *Pure expression evaluation, no assignment operator i.e. states*

- Recursive functions & automata
  - Stephen Kleene (1909-1994)
  - Regular expressions, finite-state machines



Vitaly Shmatikov

The Early Days

- The very first programs were written in pure **binary notation**
  - Data and instructions had to be encoded in strings of 1s and 0s, octal or hex values
  - It was up to the programmer to keep track of where everything was stored in the machine's memory.
    - *Binary representation of Op Codes and addresses had to be determined by hand*
    - *e.g. before you could call a subroutine, you had to calculate its address.*

- Technology that lifted these burdens from the programmer was **assembly language**
  - Binary codes were replaced by symbols such as *load, store, add, sub*.
  - The symbols were translated into binary by a program called an assembler
    - *also calculated addresses of subroutines and calls*

This was the very first time in which the computer was used to help with its own programming

http://www.cs.iastate.edu/~leavens/ComS541Fall97/hw-pages/history/

Assembly Languages

- Invented by machine designers the early 1950s
- Mnemonics instead of binary opcodes

```
push ebp
mov ebp, esp
sub esp, 4
push edi
```

- Reusable macros and subroutines

Assembly Languages Drawback

- Assembly language drawbacks
  - The programmer had to keep in mind all the minutiae in the instruction set of a specific computer.
  - Programs had to be rewritten for every hardware platform
    - *C intention was to have a portable assembler*
  - Mathematical expression such as x^2+y^2 might require dozens of assembly-language instructions.
- First higher-level language: FORTRAN
  - The programmer thinks in terms of variables and equations
    - *Rather than registers and addresses.*
    - *e.g.in FORTRAN x^2+y^2 would be written simply as X**2+Y**2.*

Expressions of this kind are translated into binary form by a program called a compiler.

http://www.voidspace.org.uk/technology/programming_history.shtml

The Essence of a Programming Language

- Formal notation for specifying computations
  - Virtual all programming languages are Turing Complete
  - Can only perform algorithms that are executed by a Turing maschine

- Syntax (usually specified by a context-free grammar)
- Semantics for each syntactic construct

- Practical implementation on a real or virtual machine
  - Translation vs. compilation vs. interpretation
    - *C++ was originally* ***translated*** *into C by Stroustrup's Cfront*
    - *Java originally used a* ***bytecode*** *interpreter, now native code compilers are commonly used for greater efficiency*
    - *Lisp, Scheme and other functional languages are* ***interpreted*** *by a virtual machine, but code is often precompiled to an internal executable for efficiency*

- Efficiency vs. portability

All About Efficiency

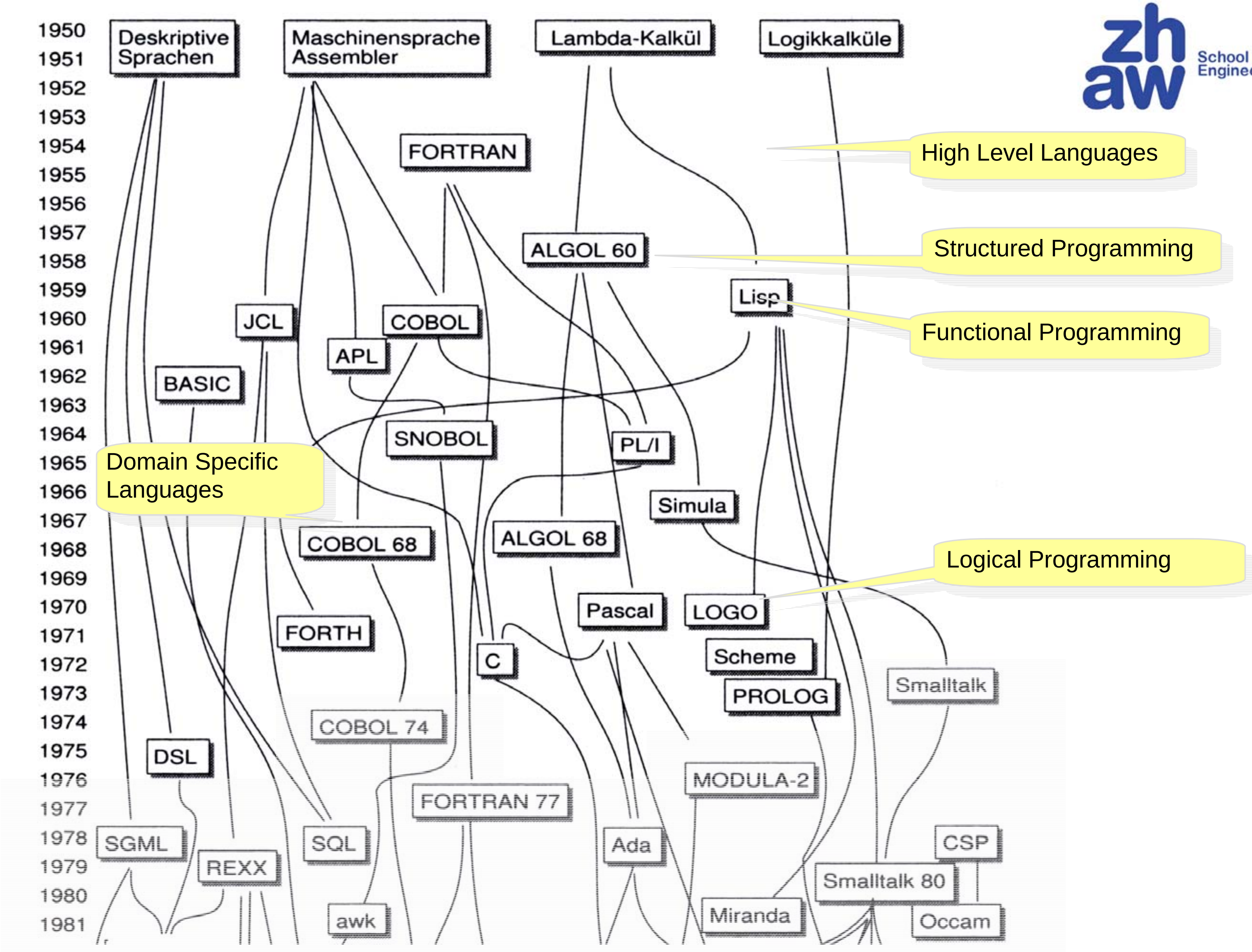
- Efficiency was key for the early programming languages
  - Compilation took minutes/hours

Performance Tier	FLOPS equivalent	Key Platforms
kiloflops (KFLOPS)	1,000 FLOPS	IBM 701 (1953) IBM 704 (1955) Apple II (1977)
megaflops (MFLOPS)	1,000,000 FLOPS	CDC 6600 (1966) Cray 1 (1976) Intel Pentium (1993)
gigaflops (GFLOPS)	1,000,000,000 FLOPS	Cray 2 (1985) Thinking Machines CM-2 (1987) Microsoft Xbox (2001)
teraflops (TFLOPS)	1,000,000,000,000 FLOPS	Intel ASCI Red (1996) IBM ASCI Blue Pacific (1998) IBM ASCI White (2000) NEC Earth Simulator (2002)
petaflops (PFLOPS)	1,000,000,000,000,000 FLOPS	IBM Blue Gene (2005-2010)

Apollo 11 (1969)
Lunar Module
Mission Computer
40 KFlops

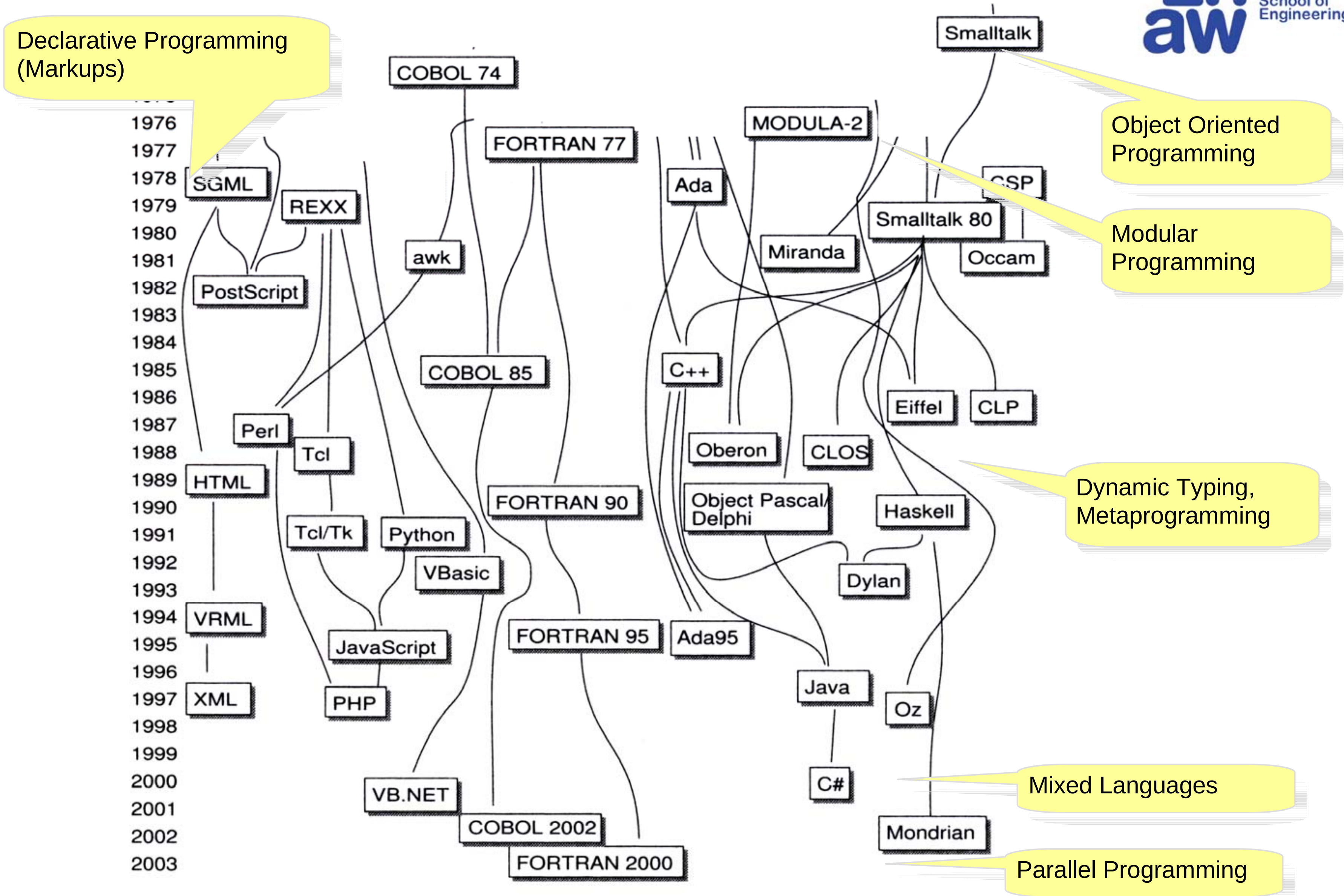
iPhone 5s
76.8GFlops

- Intel Core i750 can have 7 GFlops
- GPUs can have much as 2.5 TFlops
- A PC, all in all, to be capable of 2 to 2600 GFlops



1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
Deskriptive Sprachen
Maschinensprache Assembler
Lambda-Kalkül
Logikkalküle
High Level Languages
FORTRAN
Structured Programming
ALGOL 60
Lisp
Functional Programming
JCL
COBOL
APL
BASIC
SNOBOL
PL/I
Domain Specific Languages
Simula
COBOL 68
ALGOL 68
Logical Programming
Pascal
LOGO
FORTH
C
Scheme
Smalltalk
PROLOG
COBOL 74
DSL
MODULA-2
FORTRAN 77
SGML
SQL
Ada
CSP
REXX
Smalltalk 80
Miranda
awk
Occam

zh aw School of Engineering



Popularity of Programming Languages 2015

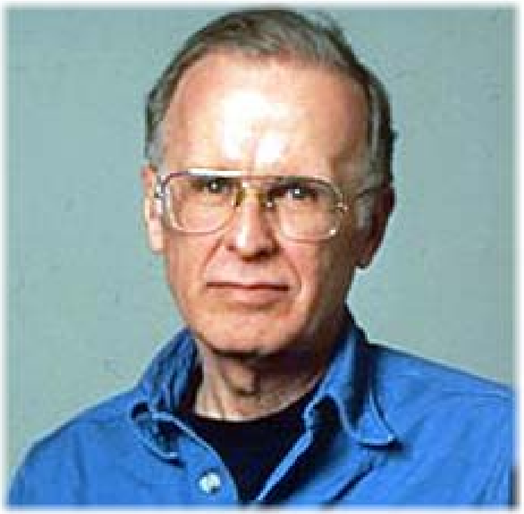
Feb 2015	Feb 2014	Change	Programming Language		Ratings	Change
1	1		C	1974	16.488%	-1.85%
2	2		Java	1994	15.345%	-1.97%
3	4		C++	1979	6.612%	-0.28%
4	3		Objective-C	1981	6.024%	-5.32%
5	5		C#	2000	5.738%	-0.71%
6	9		JavaScript	1995	3.514%	+1.58%
7	6		PHP		3.170%	-1.05%
8	8		Python		2.882%	+0.72%
9	10		Visual Basic .NET		2.026%	+0.23%
10	-		Visual Basic		1.718%	+1.72%
11	20		Delphi/Object Pascal		1.574%	+1.05%
12	13		Perl		1.390%	+0.50%
13	15		PL/SQL		1.263%	+0.66%

http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

FORTRAN

FORTRAN (1954-57)

- Stands for **FOR**mula **TRAN**slation
- Developed at IBM under the guidance of John Backus primarily for scientific programming
- Dramatically changed forever the way computers used
- Has continued to evolve
- Always among the most efficient compilers, producing fast code
- Still in use e.g. for supercomputers



John Backus

FORTRAN History

- First high level programming language
- FORTRAN originally began as a digital code interpreter for the IBM 701
  - with < 10 Kflops
- Originally only three control structures:
  - DO
  - IF
  - GOTO
- FORTRAN has undergone many modifications. The newest version is FORTRAN 2008
- FORTRAN is still used for numeric computations and scientific computing

FORTRAN History

- The design of FORTRAN made it easier to translate mathematical formulas into code.
- At that time it was called *Speedcoding*
- The point of FORTRAN was to make programming easier.
- At the beginning of the 60ies over 50% of the software was in FORTRAN

developers at work

Destry Diefenbach

FORTRAN I Features

- Names could have up to six characters
- Post-test counting loop (DO)
- Formatted I/O
- User-defined subprograms
- Three-way selection statement (arithmetic IF)
  - IF (ICOUNT-1) 100, 200, 300

    negative, zero, positive
- No data typing statements
  - variables beginning with i, j, k, l, m or n were integers, all else floating point
- No separate compilation
- Programs larger than 400 lines rarely compiled correctly, mainly due to IBM 704's poor hardware reliability
- Code was very fast - for that time

FORTRAN Evolution

- Version history
  - FORTRAN 1957
  - FORTRAN II
  - FORTRAN IV
  - FORTRAN 66 (released as ANSI standard in 1966)
  - FORTRAN 77 (ANSI standard in 1977)
  - FORTRAN 90 (ANSI standard in 1990)
  - FORTRAN 95 (ANSI standard version)
  - FORTRAN 2000 (ANSI standard version)
  - FORTRAN 2008 (ISO/IEC 1539-1:2010)

- Many different “dialects” produced by computer vendors
  - e.g. Digital VAX FORTRAN, now Intel FORTRAN
  - Fortran.NET by Fujitsu

Statement Format

- FORTRAN before 90 requires a fixed format

```
      PROGRAM MAIN
C     COMMENTS ARE ALLOWED IF A “C” IS PLACED IN COLUMN #1
      DIMENSION X(10)
      READ(5,*) (X(I),I=1,10)
      WRITE(6,1000) X
 1000 FORMAT(1X,’THIS IS A VERY LONG LINE OF TEXT TO SHOW HOW TO CONTINUE ’
     *  ‘THE STATEMENT TO A SECOND LINE’,/,10F12.4)
```

1-5 Label

6

7-72 Statements

73-80 Optional Line #s

Any character: continuation line

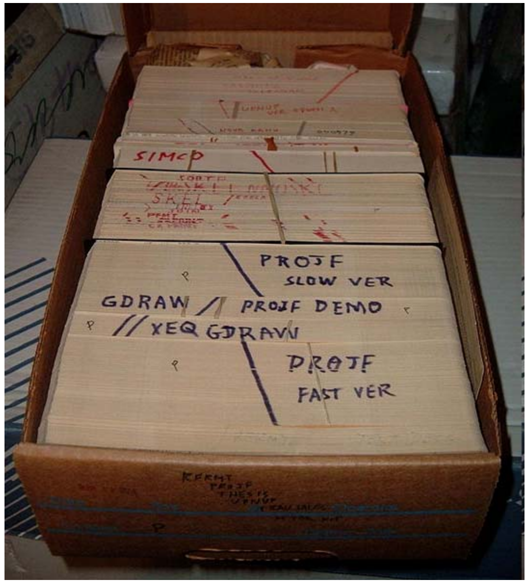
- Based on the punch card in use when FORTRAN was created

Statement Format - Fixed Format

- “C” in column 1 indicates that line is a comment
- Columns 1-5 are reserved for statement labels
  - Statement labels are not required unless the statement is the target of a goto
  - Labels are numeric values only
- Column 6 is the continuation flag
  - Any character in column 6, other than space or “0”, indicates that this line is a continuation of the previous line
  - There is usually a limit of 19 on the number of continuations
- Columns 7-72 are contain FORTRAN statements
- Columns 73-80 is for sequence information
  - Only of any use when using punch cards

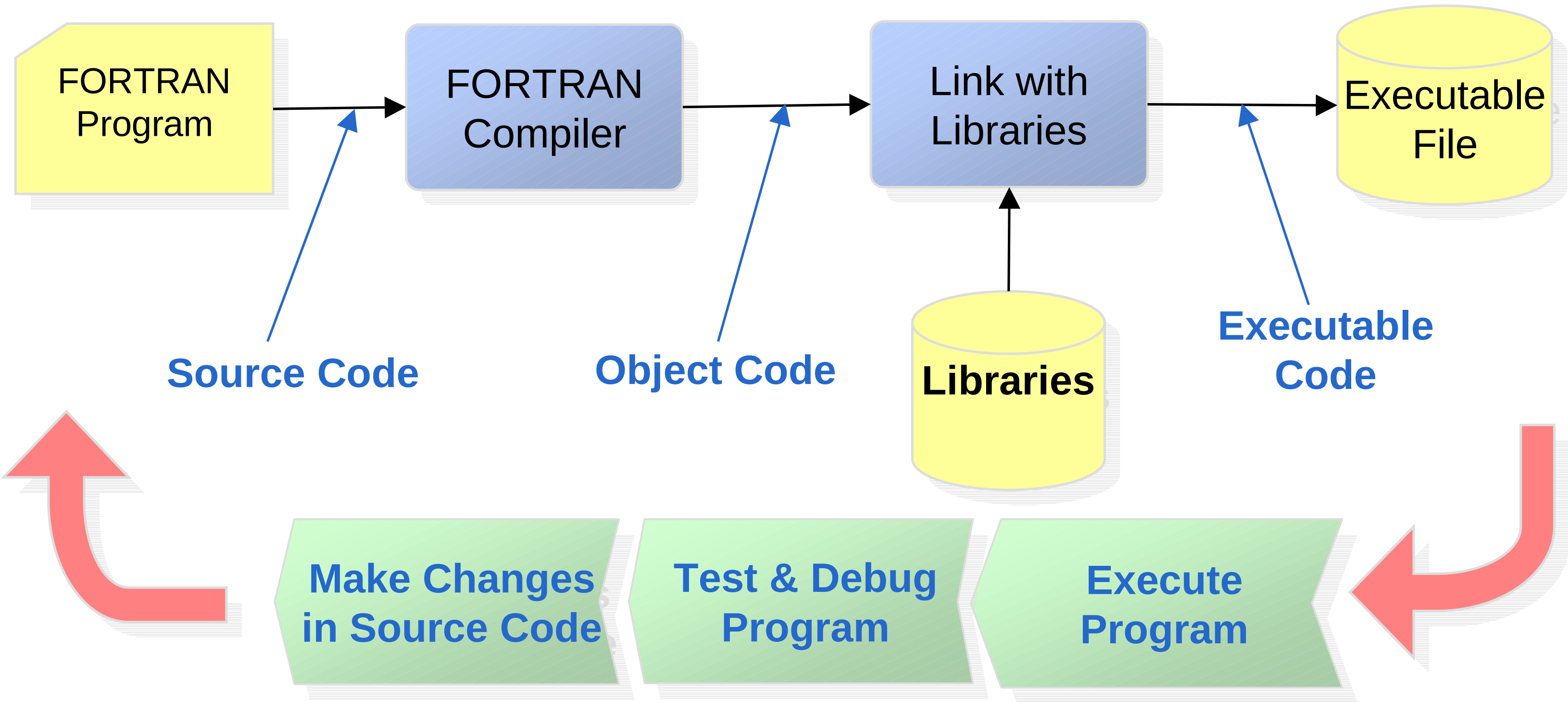
Statement Format

- IBM punch card
- 1 Card = 1 Line of Code



Building a FORTRAN Program

- FORTRAN is a complied language (like C) so the source code (what you write) must be converted into machine code before it can be executed (e.g. Make command)



Structure of a FORTRAN Program

- FORTRAN is a compiled language
- All memory is allocated statically at compile time
  - There is no standard method for dynamically allocating memory in a FORTRAN program before FORTRAN 90
  - Memory is allocated in a predictable manner, a fact which can be used by the programmer to his advantage or distress
  - FORTRAN does not guarantee values of un-initialized memory
- There is no official recursion support before FORTRAN 90
  - *some vendor implementations had recursive capabilities*
  - *static memory allocation is at odds with the use of a stack which is needed for recursion*

Structure of a FORTRAN Program

- FORTRAN consists of program units
  - Program
  - Function
  - Subroutine
  - Block Data

- The program unit contains the main code and the point where execution starts
  - Earlier versions of FORTRAN did not have a `program` statement
  - Since FORTRAN 77 a program begins with the `program` statement
  - The `end` statement terminates the program unit

- A program unit may contain internal sub-programs
  - Internal functions
  - Internal subroutines

Original Style of FORTRAN Program Sample

- All in Capitals

```
      PROGRAM FUNDEM
C     DECLARATIONS FOR MAIN PROGRAM
      REAL A,B,C
      REAL AV, AVSQ1, AVSQ2
      REAL AVRAGE
C     ENTER THE DATA
      DATA A,B,C/5.0,2.0,3.0/

C     CALCULATE THE AVERAGE OF THE NUMBERS
      AV = AVRAGE(A,B,C)
      AVSQ1 = AVRAGE(A,B,C) **2
            AVSQ2 = AVRAGE(A**2,B**2,C**2)

      WRITE  (6,100) 'THE AVERAGE OF THE SQUARES IS: ', AVSQ2
100   FORMAT (A32, F5.3)
      END

      REAL FUNCTION AVRAGE(X,Y,Z)
      REAL X,Y,Z,SUM
      SUM = X + Y + Z
      AVRAGE = SUM /3.0
      RETURN
      END
```

Hello FORTRAN

- FORTRAN prints "Hello World" the default output device
- Fixed Format

```
      WRITE (6,100) "Hello World"
100   FORMAT (A11,//)
      END
```

6 blanks

- Key Words are all capitalized by convention- in earlier times
- Statements start at position 6
- Since FORTRAN 90 free format allowed, ! indicate comment
- File Extension e.g. f95

```
! hello world program
program hello
print *, "hello world!"
end program hello
```

Sample FORTRAN Programm

- ! indicate comment

```
program Convert
implicit none
! -----------------------------------------------Declare
real*4 tempC, tempF, FACTOR
integer*2 ZERO_SHIFT
parameter (ZERO_SHIFT = 32, FACTOR = 5./9.)
! -----------------------------------------------Input
print*, "Enter the temperature in Fahrenheit ..."
read*, tempF
! -----------------------------------------------Compute
tempC = FACTOR * (tempF - ZERO_SHIFT)
! -----------------------------------------------Output
print*, "The corresponding Centigrade temperature is "
print*, tempC, " degrees."
end
```

FORTRAN Variable

- Variables represent the memory of the program
- FORTRAN variables
  - FORTRAN IV numbers and letters, at least 6 significant characters
  - FORTRAN 77 numbers and letters and "_", at least 16 characters
  - must start with a letter
- Up through 77, **spaces in a FORTRAN program are ignored**
  - IVALUE and I VAL UE are the same
  - using strange spacing, while acceptable, is bad practice
- FORTRAN variables are typed
- FORTRAN is case insensitive
  - **ivar** is the same as **IVAR** or **IvAr**

FORTRAN Variable Typing

- All FORTRAN variables are typed nowadays

- INTEGER
  - ordinal number
- REAL/DOUBLE PRECISION
  - floating point values
- COMPLEX
  - complex values

- CHARACTER (77+)
  - strings
- LOGICAL
  - boolean values

FORTRAN Variable Typing

- A unique feature of FORTRAN – implicit typing
  - When a variable appears that has not been declared previously it is created (at compile time)
  - It is assigned a type based on the first character of the name
    - *A-H,O-Z is type REAL*
    - *I-N is type INTEGER*
  - A typo can cause the creation of a new variable – not an error
  - Old FORTRAN joke: Good is REAL if not defined otherwise

- Starting with 77 the **`implicit`** statement was added
  - Allowed changing the first letter assignments
  - Most 77 compilers include the **`implicit none`** statement that requires that all variables be explicitly declared and typed – prevents the typo problem

- Today, it is regarded as good style to use **`implicit none`**

FORTRAN Variable Typing

- Disable implicit typing altogether

```
program test
implicit none
```

- In the declarations section enter a type identifier followed by :: and a list of variable names

```
integer :: a,value,istart
real initial_value
```

- In the declarations section enter a type identifier followed by a list of variable names
- The first letter implicit typing is over-ridden when explicit typing is used

FORTRAN Variable Typing

- The types presented earlier are the default types
- The range of both INTEGER and REAL had dependent on the computer architecture
  - One computer may have a 32 bit integer while another may use 16 bit as its default
- A first attempt to deal with this lead to types such as
  - **`real*8, integer*4`**
  - The number after the * indicates the number of bytes used
  - Most computers have 8 bit bytes
  - Not every architecture will have every combination
    - *Not an actual problem*
  - But knowledge of the architecture of the system where a legacy FORTRAN program was developed is needed to be converted
- Today use of IEEE Types (without size)
  - **`real :: test`**

FORTRAN Variable Typing

- The COMPLEX type
  - A built in data type

```
complex :: a,b,c
a = (3.0,-1.5)
b = (1, -1)
c = a * b
```

FORTRAN Variable Typing

- The CHARACTER type was introduced in 77
- The * notation is used to specify the maximum number of characters the variable can hold

```
character*20 :: string1
string1 = 'abcd'
character*8 :: string2
```

FORTRAN Variable Typing : LOGICAL

- The LOGICAL type

```
logical :: error
error = .false.
```

- May be result of a comparison

```
error =  b**2 - 4*a*c .lt. 0.0
```

FORTRAN Arrays

- The array is the only data structure supported in 77 and before
- An array is a linear allocation of memory
- An array can contain up to 7 dimensions
- Arrays are indexed starting a 1 !

```
integer :: a
dimension a(10)
integer :: b
dimension b(10,10)
!shortcut
real :: c(10,10,10)
```

FORTRAN Subroutine

- The subroutine unit contains FORTRAN code that can be called from other FORTRAN code
- A subroutine begins with a **`subroutine`** statement
  - Contains a name for the subroutine
  - A list of formal arguments
- Subroutines may be internal or external
  - An internal subroutine is included in the code of program unit and is only callable by the program
  - An external subroutine is created outside of a program unit and is callable from everywhere
- Has no return value

```
      subroutine mult(a,b,c)
      real :: a,b,c
      c = a * b
      return
      end
```

```
      call mult(5.0,x,value)
```

FORTRAN Function

- The function unit contains FORTRAN code that can be called from other FORTRAN code
- It differs from a subroutine in that it **returns a value**
- A subroutine begins with a **`function`** statement
  - Contains a name for the function
  - A list of formal arguments
  - Specifies a return type
- Functions may be internal or external
  - An internal function is included in the code of program unit and is only callable by the program
  - An external function is created outside of a program unit and is callable from everywhere

```
      real function mult(a,b)
      real :: a,b
      mult = a * b
      return
      end
```

```
    value = mult(5.0,x)
```

FORTRAN Variables and Subroutines

- All **arguments** to a FORTRAN subroutine are **passed by reference**
  - The subroutine receives the address of the variable
  - Any changes made by the subroutine are seen by the caller
  - Most other languages pass by value (the subroutine receives a copy)
  - Passing an array as an argument with just the name will pass the address of the first element

- **On entry to a subroutine its local variables are not guaranteed to have any known value**
  - The **save** statement introduced in F 77 will ensure
    that a variable will have on entry the value that it had on its last exit from the subroutine

FORTRAN Block Data

- Normally variables in a FORTRAN program are local to the unit in which they are declared
  - variables may be made known to subroutines using the arguments
  - variables may be created in a common block
- Common blocks are named shared memory areas
  - each program unit that declares the common block has access to it
  - each program unit that declares access to a common block defines it’s own view
    - *type of each variable in the block*
    - *size of each array in the block*

```
programm a
common /xmach/ a,b(250),c
common /fx/ nt,ntd,nfr(5),ec,el,gzero

programm b
common /xmach/ a,b(50,5),c
```

name of common block

FORTRAN Assignment

- The simple assignment statement stores the result of computations into a variable

```
integer :: a


a = a + 1
```

```
dimension a(10,10)
    …
a(i,10) = 2.0 * pi * r**2
```

FORTRAN Literals

- Literals are constants that appear in a FORTRAN program
- Number
  - integers - 1, -34
  - real - 1.0, 4.3E10, 5.1D-5
  - complex – (5.2,.8)
- Other
  - logical - .true., .false.
  - character – ‘title line’

FORTRAN Literals

```
integer :: a
a = 34
```

```
real :: a(20)
a(1) = 31.4159e-1
```

```
iterm = -10.3
```

```
complex :: z
z = (10,-10.5)
real_part = real(z)
aimag_part = aimag(z)
z = cmplx(real_part * 2,aimag_part)
```

FORTRAN Expressions

- Expressions are the heart of FORTRAN (Formula Translator)
- There are two types of expressions
  - numeric
    - *2 * 3.14159 * RADIUS**2*
    - *SIN(PI)*
  - logical
    - *LOGICAL IBOOL = .TRUE.*
    - *I .EQ. 10 .AND. ISTOP*

```
integer :: a
a = 34
```

```
real :: a(20)
a(1) = 31.4159e-1
```

```
complex :: z
z = (10,-10.5)
real_part = real(z)
aimag_part = aimag(z)
z = cmplx(real_part * 2,aimag_part)
```

FORTRAN Parameter Statement

- The `PARAMETER` statement is used to define constants
- Old syntax untyped

```
      PARAMETER (MAX=20)
```

- New syntax typed

```
      integer, parameter :: max=20
```

- A parameter can be used wherever a variable is expected – but cannot be overwritten
- Can be used in declarations

```
      integer, parameter :: max=20
      integer a(max)
```

FORTRAN Numerical Operators

- The numerical operators
  - **(exponentiation)
  - * /
  - unary + -
  - binary + -
- Parentheses are used to alter the order of evaluation
- For binary operators, if the types do not match an implicit conversion is performed to the most general type
  - *integer -> real -> double precision*
  - *anything -> complex*

- WARNING: division of an integer by an integer will produce a truncated result
  - *5 / 2 => 2 not 2.5*
  - *float(5)/2 => 2.5*
- The type-conversion intrinsic functions can be used to get the desired results

Intrinsic (built-in) Functions

- FORTRAN includes an extensive set of built-in functions
- FORTRAN 66 has different names for these functions depending on the return type and argument type
  - One letter prefix to define type of function I->int; D->double; C -> complex
- FORTRAN 77 introduced generic names for intrinsic functions
- e.g.
  - *log(*real or double*)* *the generic version*
  - *dlog(*double*)*
  - *clog(*complex*)*

Type Conversion

- The intrinsic functions have two forms
  - generic available only in 77 and above
  - argument specific
- Square root
  - SQRT(*real or double*) the generic version
  - SQRT(*real*)
  - DSQRT(*double*)
  - CSQRT(*complex*)
- Conversion to integer
  - INT(*any*) the generic version
  - IFIX(*real*)
  - IDINT(*double*)
- Conversion to double
  - DBLE(*any*) the generic version
- Conversion to complex
  - COMPLX(*any*) the generic version
- Conversion to real
  - REAL(*any*) the generic version
  - FLOAT(*integer*)
  - REAL(*integer*)
  - SNGL(*double*)

Math Functions (subset)

- Sine and Cosine (radians)
  - SIN(*real or double*) the generic version
  - SIN(*real*)
  - DSIN(*double*)
  - CSIN(*complex*)

- Exponential
  - EXP(*real or double*) the generic version
  - EXP(*real*)
  - DEXP(*double*)
  - CEXP(*complex*)

- Natural logarithm
  - LOG(*real or double*) the generic version
  - DLOG(*double*)
  - CLOG(*complex*)

FORTRAN Control Statements

- Branching (GOTO)
- Comparison (IF)
- Looping (DO)
- Subroutine invocation (CALL)

FORTRAN Branching

- FORTRAN includes a `GOTO` statement
- In modern languages this is considered very bad
  - its use was essential in FORTRAN 66 its predecessors
  - FORTRAN 77 introduced control statements that lessened the need for the GOTO

```
      if (i .eq. 0) go to 100
```

```
      a = 4.0 * ainit
      goto 200
  100 b = 52.0
      …
  200 c = b * a
```

FORTRAN Branching

- The FORTRAN `GOTO` always branched to a FORTRAN statement that contained a label in columns 1-5
- The labels varied from 1 to 99999
- Variations of the go to statement are
  - assigned goto
  - computed goto
- Spaces are ignored in FORTRAN code before 90
  - `GOTO` and `GO  TO` are equivalent
- Excessive use of the goto (required in 66 and before) leads to difficult to understand code

FORTRAN Branching

- Computed goto
- Operates much like a case or switch statement in other languages

```
      goto (100,200,300,400),igo
      …
  100 continue
      …
      goto 500
  200 continue
      …
      goto 500
      …
  500 continue
```

FORTRAN Continue

- The **`CONTINUE`** statement is a do-nothing statement and is frequently used as a marker for labels
- It is used most frequently with DO loops

FORTRAN IF

- The `IF` statement is used to perform logical decisions
- The oldest form is the 3-way if (also called arithmetic if)
- The logical if appeared in FORTRAN IV/66
- The more modern if-then-else appeared in FORTRAN 77

FORTRAN 3-way If - Original Construct

- The 3-way if statement tested a numerical value against zero
- It branched to one of three labels depending on the result; < 0, 0, >0

```
      if (radius) 10,20,30
   10 continue
      …
      goto 100
   20 continue
      …
      goto 100
   30 continue
      …
      goto 100
  100 continue
```

```
      if (abs(radius-eps)) 10,10,20
   10 continue
      …
      goto 100
   20 continue
      …
      goto 100
  100 continue
```

FORTRAN Logical If

- The logical if statement performed a test using the logical operators
  - .EQ., .NE., .LT., .LE., .GT., .GE.
  - .AND., .OR., .NOT.
- If result is true then **a single statement** is executed, e.g. goto

```
      if (istart .eq. 50) goto 100
      …
  100 continue
```

```
      if (imode .eq. 2) a = sqrt(cvalue)
      …
```

```
      logical :: quick
      quick = .true.
      if (quick) step=0.5
      if (.not. quick) step = 0.01
```

FORTRAN Comparison

- Comparison Oparators

```
.lt.   less than
.le.   less than or equal to
.eq.   equal to
.ne.   not equal to
.ge.   greater than or equal to
.gt.   greater than
```

FORTRAN Modern If

- FORTRAN 77 introduced the modern if statement (so-called structured programming)
- The test operated the same as the logical if
- Greatly reduced the need for using the goto statement
- Includes
  - **`then`** clause
  - **`else`** clause
  - **`else if`** clause

FORTRAN Modern If

- This form eliminates the goto statements from the previous example

```
logical :: quick
quick = .true.
if (quick) then
  step=0.5
else
  step = 0.01
endif
…
if (quick .and. (abs(xvalue – eps) .lt. 0.005)) then
…
end if
```

FORTRAN Looping

- The DO statement is the mechanism for looping in FORTRAN
- The do loop is the only “official” looping mechanism in FORTRAN through 77

```
      do 100 i=1,10,2
      …
  100 continue
```

- Here I is the control variable
  - it is normally an integer but can be real
  - 1 is the start value
  - 10 is the end value
  - 2 is the increment value, may be omitted -> 1
  - everything to the 100 label is part of the loop

FORTRAN Looping

- The labeled statement can be any statement not just continue
- Loop may be nested
  - nested loops can share the same label – very bad form

```
      do 100 i=1,10,2
      do 100 j=1,5,1
      …
  100 a(i,j) = value
```

```
      do 200 i=1,10,2
      do 100 j=1,5,1
      …
      a(i,j) = value
  100 continue
  200 continue
```

FORTRAN Looping

- FORTRAN 77 introduced a form of the do loop that does not require labels

```
do i=1,100
…
enddo
```

```
do i=1,100
  do j=1,50
    a(i,j) = i*j
  end do
enddo
```

- The indented spacing is not required

Miscellaneous Statements

- `RETURN` will cause a sub program to return to the caller at that point – the `END` statement contains an implied `RETURN`
- A number on a RETURN statement indicates that an alternate return be taken
- `STOP` will cause a program to terminate immediately – a number may be included to indicate where the stop occurred, `STOP 2`
- PAUSE will cause the program to stop with a short message – the message is the number on the statement, PAUSE 5

FORTRAN I/O Statements

- FORTRAN contains an extensive input/output capability

- FORTRAN I/O is based on the concept of a unit number
  - 5 oder * is generally input – stdin on Unix
  - 6 oder * is usually output – stdout on Unix

- Files are can be created as needed

```
open (unit = 4, file = 'genetories1.dat', form='formatted')
open (unit = 11, file="ustream.demo", status="new", access="stream")
```

FORTRAN I/O Statements

- There are two types of I/O in FORTRAN
  - formatted
  - unformatted or binary
- There are two modes of operation
  - sequential
  - random
- Formatted I/O uses a format statement to prepare the data for output or interpret for input
- Unformatted I/O does not use a format statement
  - the form of the data is generally system dependent
  - usually faster and is generally used to store intermediate results

FORTRAN I/O Statements

- Unformatted output I/O does not use a format statement

```
print *, a,b,c,d,e
write (6,*) a,b,c,d,e
write (*,*) a,b,c,d,e
```

- The input are

```
read *, a,b,c,d,e
read (5,*) a,b,c,d,e
```

FORTRAN I/O Statements

- The `FORMAT` statement is the heart of the FORTRAN formatted I/O system
- The format statement instructs the computer on the details of both input and output
  - size of the field to use for the value
  - number of decimal places
- The format is identified by a statement label
  - A format can be used any number of times
  - The label number must not conflict with goto labels

Warning: first column might be directive for printer:
0 linefeed, 1 new page

```
      write(6,9000) a,b,c,d,e
9000 format(1x,4f8.5,2x,e14.6,//)
```

1 space

4 floats 8 wide and 5 digits

typical print output, if you mistakenly used 1 in first column

FORTRAN Parallel Programming

- FORTRAN has support for parallel programming via OMP
- Good performance because of the mostly static data of FORTRAN
- Still popular for supercomputers e.g. for weather prediction

```
program main
use omp_lib
double precision :: a,h,pi,sum,x,sum_local
...
h = 1.0d0 / n
sum = 0.0d0
!$omp parallel private(i,x,sum_local) num_threads(2)
sum_local = 0.0d0
do i = 1,n
  x = h * (DBLE(i)-0.5d0)
  sum_local = sum_local + f(x)
end do
!$omp critical
sum = sum + sum_local
!$omp end critical
!$omp end parallel
pi = h * sum
...
end program main
```

a new variable instance for each thread

sum up in a shared variable (Mutex)

Summary

- History and evolution of programming languages
- FORTRAN as the first but still used programming language
  - Efficiency was everything
  - Card oriented, with information in fixed columns
  - First language to catch on in a big way
  - Because it was first, FORTRAN has much room for improvement

Appendix Format Specifiers

- X format code
  - Syntax: **nX**
  - Specifies n spaces to be included at this point
- I format code
  - Syntax: **Iw**
  - Specifies format for an integer using a field width of w spaces. If integer value exceeds this space, output will consist of ****
- F format code
  - Syntax: **Fw.d**
  - Specifies format for a REAL number using a field width of w spaces and printing d digits to the right of the decimal point.
- A format code
  - Syntax: **A** or **Aw**
  - Specifies format for a CHARACTER using a field width equal to the number of characters, or using exactly w spaces (padded with blanks to the right if characters are less than w.

… Format Specifiers

- T format code
  - Syntax: **Tn**
  - Skip (tab) to column number n
- Literal format code
  - Syntax: **‘quoted_string’**
  - Print the quoted string in the output (not used in input)
- L format code
  - Syntax: **Lw**
  - Print value of logical variable as T or F, right-justified in field of width, w.