# Introduction to Fortran

Atmospheric Modelling

Zhou Putian
putian.zhou@helsinki.fi

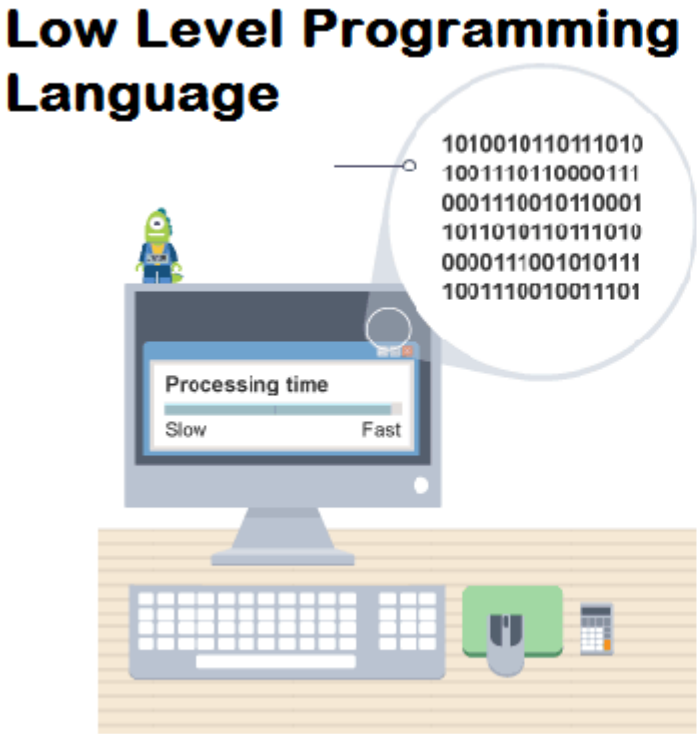
# Outline

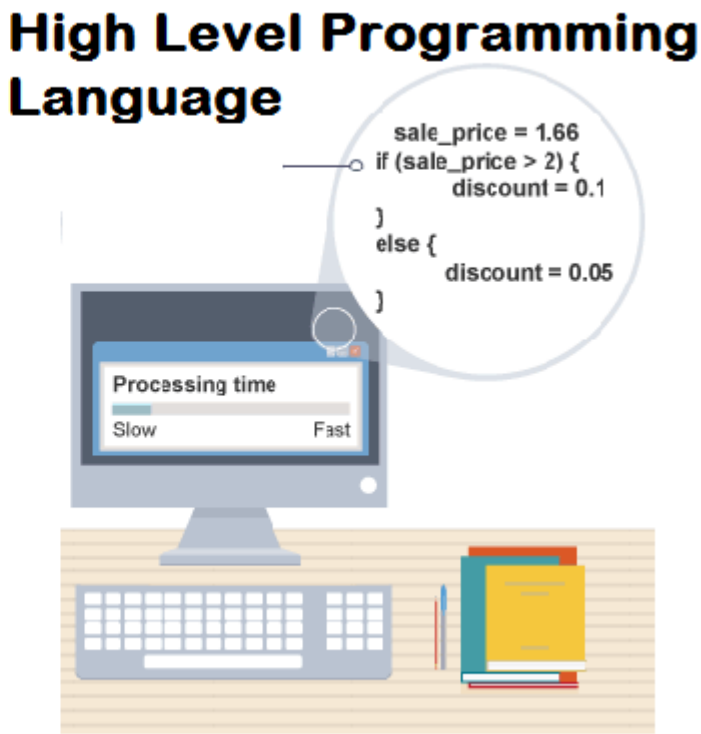
- Goals
- Introduction
- Fortran History
- Basic syntax
- Makefile

# Goals

- Write simple Fortran programs
- Understand and modify existing Fortran code
- Manage Fortran projects with makefiles

# Introduction




https://www.techdotmatrix.com/2018/01/high-level-programming-language-low-level-programming-language/

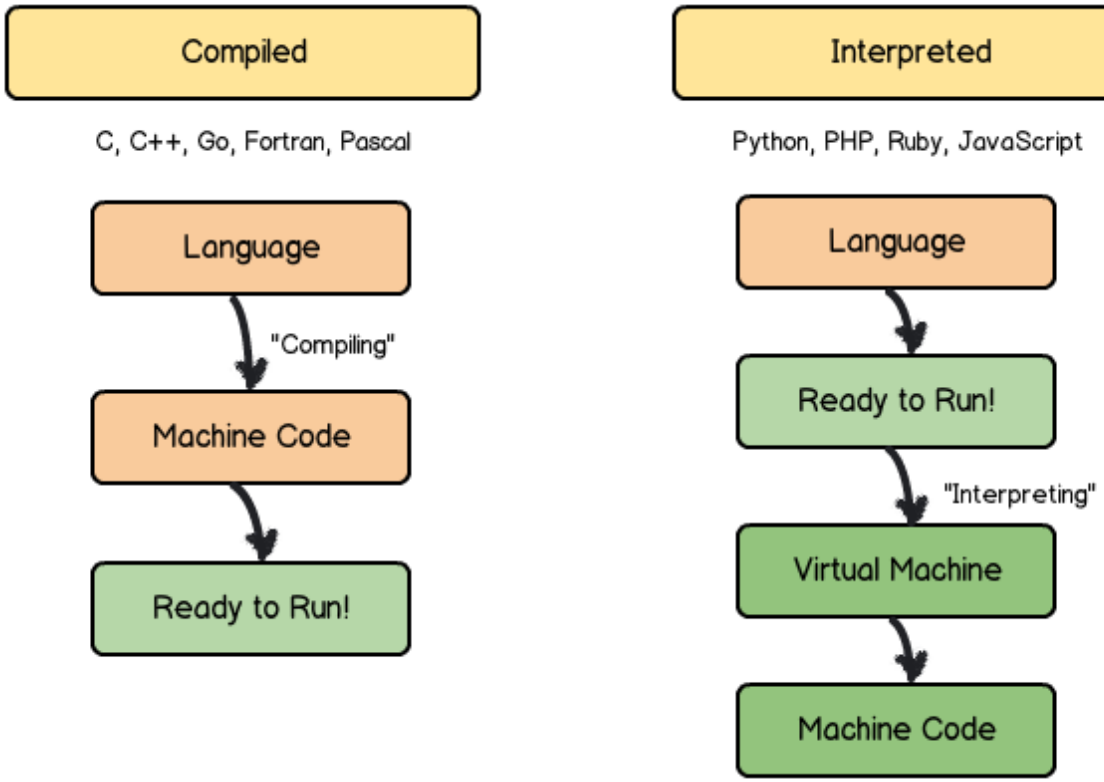
# Introduction

Two different types of high-level languages:

- Interpreted language (MATLAB, Python, ...)
  - Translation to machine-language is performed at run time by an interpreter
  - More convenient but slower (no need to declare variables; realize your idea quickly ...)
- Compiled language (Fortran, C, C++, ...)
  - Translation is performed once
  - Run faster (suitable for large-scale computing ...)

However, the border between them for some languages are not clear.

# Introduction



# Introduction

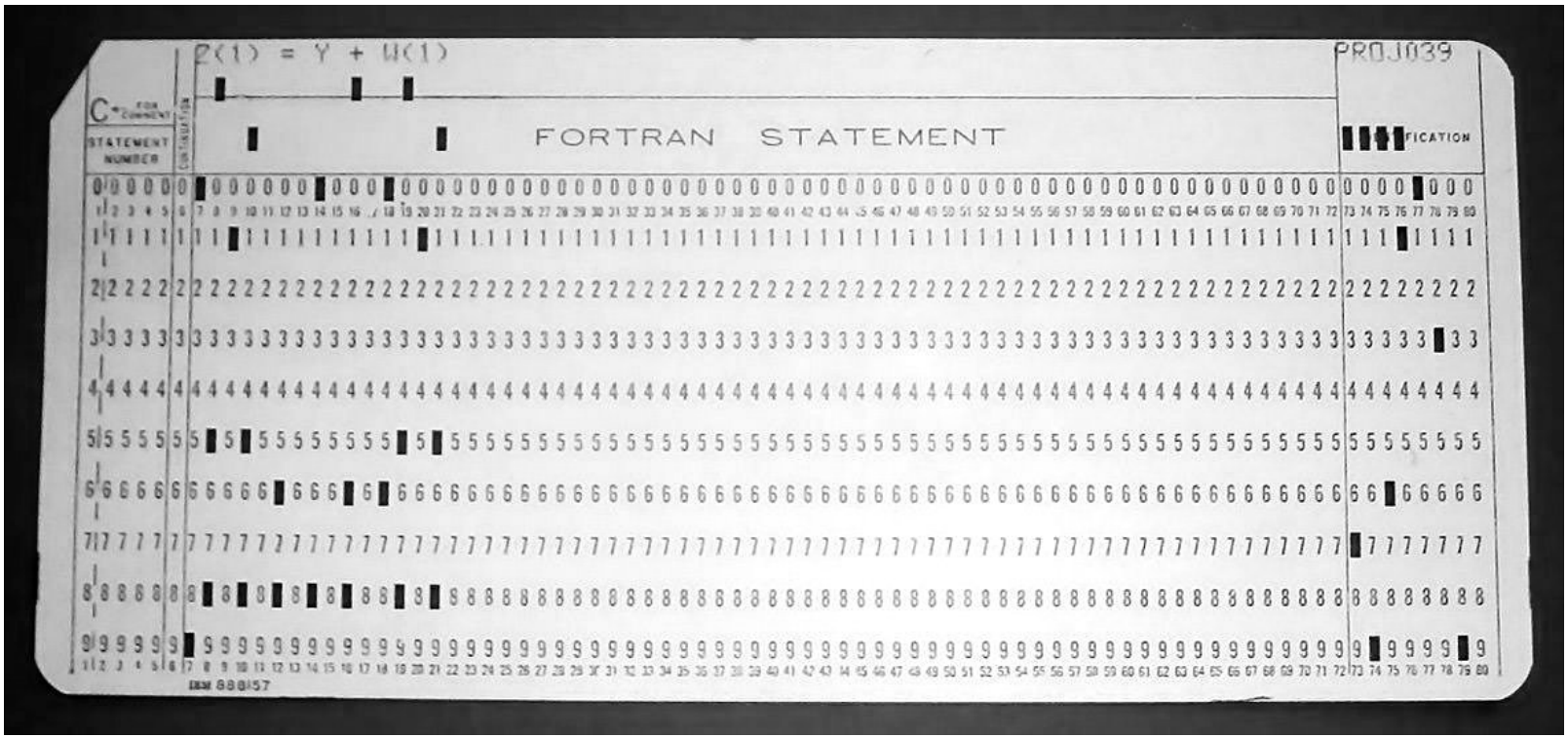
## What Language Should I Use?

- Generally, use the language you know best
- Interpreted languages are great for
  - Interactive applications
  - Code development and debugging
  - Algorithm development
- For major number crunching, compiled languages are preferred (Fortran, C, C++)

# Fortran History

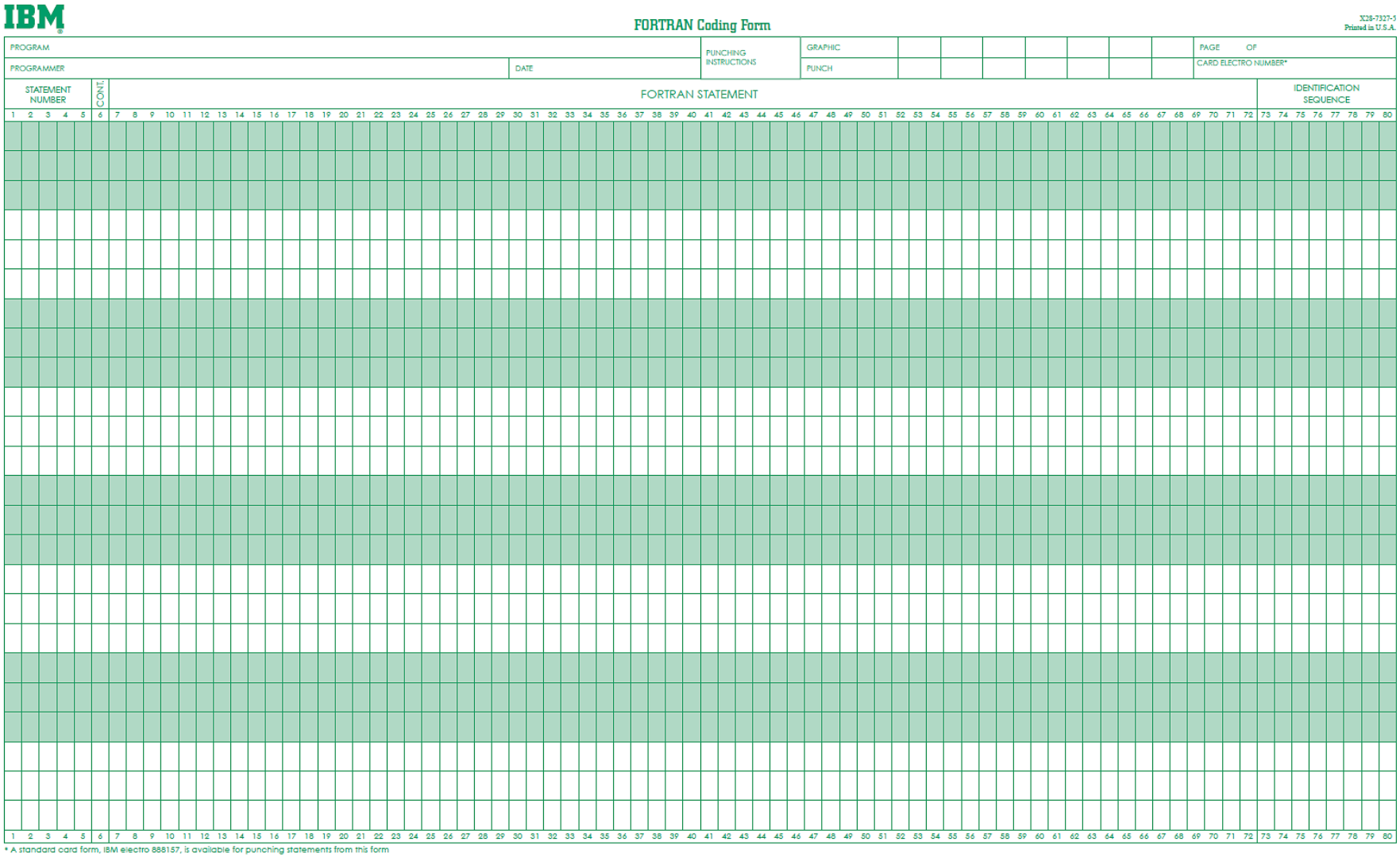
- Before Fortran, programs were written in assembly language (very tedious to say the least)
  - low-level commands such as “load x from memory into register 7” or “add values in registers 10 and 11 and write result to register 4”
- Fortran was the first widely-used high-level computer language
  - Developed by John Backus’ team in IBM as an alternative of assembly language in 1957, short for **Formula Translation**
  - Program written on a specially formatted green sheet, then entered as punched cards

# Fortran History




https://craftofcoding.wordpress.com/2017/01/28/read-your-own-punch-cards/

# Fortran History


IBM

FORTRAN Coding Form

X28-7327-5
Printed in U.S.A.

PROGRAM | PUNCHING INSTRUCTIONS | GRAPHIC | PAGE OF

PROGRAMMER | DATE | PUNCH | CARD ELECTRO NUMBER*

STATEMENT NUMBER | CONT. | FORTRAN STATEMENT | IDENTIFICATION SEQUENCE

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80

* A standard card form, IBM electro 888157, is available for punching statements from this form

# Fortran History

- Fortran 66 (1966), Fortran 77 (1978)
- Fortran 90 (1991)
  - "fairly" modern (structures, etc.)
  - Current “workhorse” Fortran
- Fortran 95 (minor tweaks to Fortran 90)
- Fortran 2003
  - Gradually being implemented by compiler companies
  - Object-oriented support
  - Interoperability with C is in the standard
- Fortran 2008 (submodules, ...)
- Fortran 2018 (formerly Fortran 2015, released in 2018)

# Basic Syntax

- Program is written in a text file called source code or source file
- Source code must be processed by a compiler to create an executable file
- Source file suffix can vary, e.g., .for, .f, .F, .f90, .F90, but we will always use ".f90"
- Since source file is simply text, it can be written with any text editor
  - emacs, vi, gedit, Notepad++, ...

# Basic Syntax

## Program organization

Most Fortran programs consist of a main program and one or more subprograms (subroutines, functions). The usual structure of a Fortran code is:

```
1  PROGRAM program_name
2    [USE module1]
3    [USE module2]
4    ...
5    IMPLICIT NONE
6    [specification part]
7    [execution part]
8  CONTAINS
9    [subprograms]
10 END PROGRAM program_name
```

- Suggestion: keep the source file the same name as the program: program_name
- Case insensitive
- Blank spaces serve as delimiter
- Line break is the statement separator

# Basic Syntax

## Program organization

- Comment character is !. Anything from ! to end of line is ignored by the compiler. Use comments liberally to document source code.
- Ampersand, &, at end of line tells compiler that statement is continued on next source line.
- Spaces don’t matter except within literal character strings, use them liberally to make code easy to read, e.g., before and after equal signs.
- Note that source lines do not end with semicolons as in C or Matlab.

# Basic Syntax

## Variable types

- **Intrinsic** variable types
  - real, integer, complex, logical, character
- **Real** variables have decimals
  - Real can be a whole number, but decimal places are stored internally
  - Even when a real is a whole number, it's a good practice to write one decimal place, e.g., write 3.0 rather than 3
- **Integer** variables do not have decimals
- **Logical** variables only have two values: .TRUE. and .FALSE.

# Basic Syntax

- Integer arithmetic is truncated, not rounded

$$3/2 = 1$$
$$2/3 = 0$$
$$5/(-2) = -2$$

- If at least one of them is real, results would be also real

$$3.0/2.0 = 1.5$$
$$2.0/3.0 = 0.6666667$$
$$5.0/(-2) = -2.50000000$$

# Basic Syntax

- **Logical** variables only have two values: .TRUE. and .FALSE.
- **Character** variables contain literal text enclosed in single or double quotes
  - e.g., "A", 'Hello', "Fortran is a computer language."
  - Blank spaces within quotes are significant, they are part of the string (contains more than one characters).

# Basic Syntax

## Operators

```
1  ! Arithmetic operators
2  x = 2.0**(-i)  ! power function and negation precedence: first
3  x = x*real(i)  ! multiplication and type change precedence: second
4  x = x / 2.0    ! division precedence: second
5  i = i + 1      ! addition precedence: third
6  i = i - 1      ! subtraction precedence: third
7  ! Relational operators
8  a < b   ! or (f77) a.lt.b -- less than
9  a <= b  ! or (f77) a.le.b -- less than or equal to
10 a == b  ! or (f77) a.eq.b -- equal to
11 a /= b  ! or (f77) a.ne.b -- not equal to
12 a > b   ! or (f77) a.gt.b -- greater than
13 a >= b  ! or (f77) a.ge.b -- greater than or equal to
14 ! Logical operators
15 .not.b ! logical negation precedence: first
16 a.and.b ! logical conjunction precedence: second
17 a.or.b  ! logical inclusive disjunction precedence: third
```

# Basic Syntax

## Variable declaration

- We need to declare the type for every variable
- Variable name
  - must start with a letter (a-z)
  - can mix with digits (0-9) and underscores ( _ ) but no blanks
  - name length <= 31
- Strongly recommend to adopt the practice of declaring with "implicit none" (MUST DURING OUR MODELING COURSE)
  - this promises the compiler that you will declare all variables
  - this goes before any type declaration statements

# Basic Syntax

## Variable declaration

- **Parameter variable**
  - If a variable has known fixed value, it can be declared as parameter and initialized when declaration.
  - The compiler substitutes values wherever variables appear in code.
  - Efficient, since there are no memory accesses
  - Example: `real, parameter :: pi = 3.14`
- Avoid using "l" because it could be mistaken for "1" or "i" (You could use "L")
- Good idea to establish your own naming conventions and follow through with them

# Basic Syntax

## Variable declaration

### Example

```
1 implicit none
2
3 real :: velocity, mass, pi
4 integer :: imax, jdim
5 character :: p
6 character(len=10) :: name  ! string
7
8 real, parameter :: pi = 3.14
9 integer, parameter :: one = 1
```

# Basic Syntax

## Kind

- Declarations of variables can be modified using "kind" parameter
- Often used for precision of reals
- Intrinsic function `selected_real_kind(n)` returns kind that will have at least n significant digits
  - n = 6 will give you "single precision"
  - n = 12 will give you "double precision"
- If you want to change precision, can easily be done by changing one line of code
- Example:

```
1 integer, parameter :: rk = selected_real_kind(12)
2 real(rk) :: x, z
```

# Basic Syntax

## Simple output

The simplest way to print out messages on the screen is to use 'list-directed' output

- `print *, a, b, ...` or `write(*,*) a, b, ...`
- Examples

```
1 print *, ra, "This is my character string."
2 write(*,*) "I am at bottom.", ib
```

# Exercise 1

Write a "hello world" program with your editor

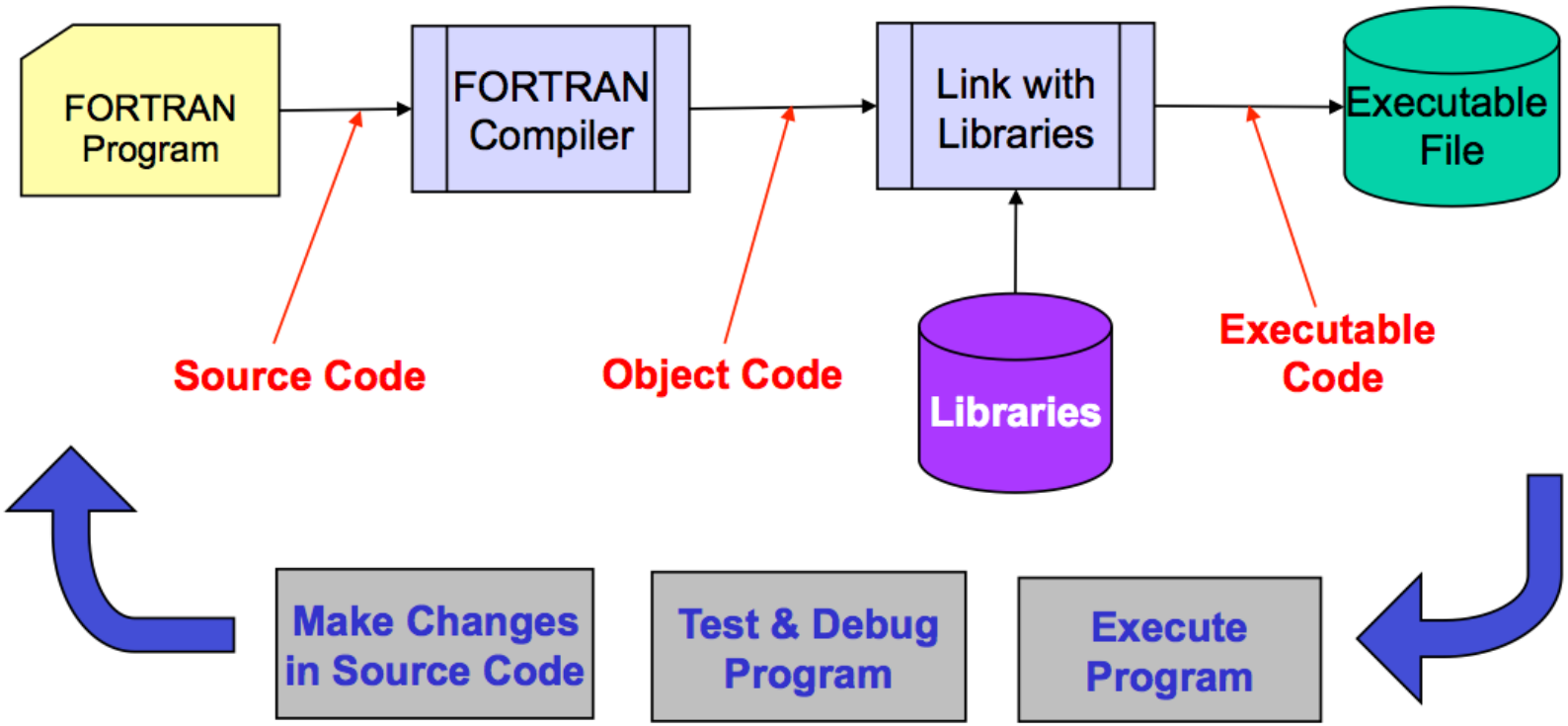
- Program should print a character string
- Save it to a file with a .f90 suffix in the name

# Solution 1

```
1 !=======================
2 ! hello world
3 !=======================
4 program hello
5   implicit none
6   write(*,*) &
7     "Hello world."
8 end program hello
```

# Compilation

Fortran is a compiled language as C, so the source code must be converted into machine code before it can be executed. This process is called compilation.



# Compilation

- A compiler is a program that reads source code and converts it to a form usable by the computer
- Internally, these steps are performed
  - preprocess source code
  - check source code for syntax errors
  - compiler translates source code to assembly language
  - assembler translates assembly language to machine language
  - linker gathers machine-language modules and libraries
- All these steps sometimes loosely referred to as compilation or compiling

# Compilation

- Code compiled for a given processor architecture will not generally run on other processors
- However, this problem will come to you later, when you start to run your code on different machines

# Compilation

- Compilers have huge numbers of options
- Compile hello.f90 on your laptop
  - If it simply returns a Unix prompt, it worked
  - If you get error messages, read them carefully and see if you can fix the source code and re-compile
- Once it compiles correctly, type the executable name at the Unix prompt, and it will print your string

```
$ gfortran hello.f90
$ ./a.out
```

# Arithmetic

- Notice: ** is power operator
  - $2.5^{1.5}$: 2.5**1.5
- Built-in math functions (sin, acos, exp, log, log10, ...), the arguments are in parentheses
  - sin(0.6), cos(pi), exp(x)
- Exponential notation indicated by letter "e" (single precision) or "d" (double precision)
  - $5.3 \times 10^4$: 5.3e4, 5.3d4

# More List-Directed I/O

- `read *, variables` is list-directed read, analogous to `print *, variables`
- `read(*,*) variables` VS `write(*,*) variables`
- Examples:

```
1 print *, "Enter a float and an integer:"
2 read *, x, j
3 print *, "float = ",  x, "   integer = ", j
4
5 write(*,*) "Enter a float and an integer"
6 read(*,*) x, j
7 write(*,*) "float = ", x, "   integer = ", j
```

# Exercise 2

Write a program to ask for a Celsius temperature (C), convert it to Fahrenheit (F), and print the result.

- make sure you declare all variables
- use decimal points with all reals, even if they're whole numbers
- the math equation is

$$F = (9/5) * C + 32$$
$$C = (5/9) * (F - 32)$$

# Solution 2

```
!=====================================
! ctof.f90
! prompt for Celcius temperature
! print Fahrenheit value
!=====================================
program ctof
  implicit none
  real :: c, f

  write(*,*) "Enter temperature in Celcius."
  read(*,*) c
  f = (9.0/5.0)*c + 32.0
  write(*,*) "T = ", f, "degrees Fahrenheit"
end program ctof
```

# Array

- Specify static dimensions in declaration

```
1 real, dimension(10,3,5) :: x
2 real :: m(2,3), n(100)
3 integer, dimension(10) :: I
```

- Can also specify ranges of dimension indices

```
1 integer, dimension(3:11, -15:-2) :: ival, jval
```

- Access array elements using parenthesis

```
1 a = y(3) + y(4)
```

- Fortran: column-major array

# Array

- Dynamic allocation
  - Useful when size is not known at compile time, e.g., input value
  - Need to specify number of dimensions in declaration
  - Need to specify that it's an allocatable array

```
1 real, dimension(:,:,:), allocatable :: x, y
```

  - allocate function performs allocation

```
1 allocate(  x(ni,nj,nk),  y(ldim,mdim,ndim)  )
```

  - When you're done with the variables, deallocate with `deallocate(x, y)`. But it is not necessary at very end of code; Fortran will clean them up for you

# Array

- Fortran can perform operations on entire arrays like MATLAB, unlike C.
- To add two arrays, simply use

```
1 c = a + b  ! a, b, c are arrays of the same shape and size
```

- Can also operate on array sections

```
1 c(-5:10) = a(0:15) + b(0:30:2)  ! must have same shape
```

  - Here b(0:30:2) represents b(0), b(2), b(4), etc., due to increment specification
  - Numbers of elements must be consistent

- Don’t assume that all Matlab matrix rules apply

```
1 c = a * b  ! * is elemental multiply, not matrix multiply
```

# Array

- There are intrinsic functions to perform some operations on entire arrays
  - sum(x): sum up all the elements in x
  - product(x): multiply all the elements in x
  - minval(x): minimum value in x
  - maxval(x): maximum value in x
  - matmul(x, y): matrix multiplication of x and y

# Exercise 3

Write a program to ask for 2 floating-point vectors of length 3, calculate the dot product and print the result

- Don’t name the code "dot_product" or "dot". Fortran has a “dot_product” intrinsic function, there is a Unix command called "dot".
- Can use array name in list-directed read, and it will expect the appropriate number of values (dimension) separated by spaces or commas

$$c = \sum_{i=1}^{3} a_i b_i = a_1 * b_1 + a_2 * b_2 + a_3 * b_3$$

# Solution 3

```
!=============================
! dotprod.f90
! prompt for two real vectors
! of length 3
! calculate dot product
! print result
!=============================
program dotprod
  implicit none
  real :: c
  real, dimension(3) :: a, b
  !-------------------------
  ! enter data
  !-------------------------
  print *, "Enter first vector"
  read *, a
  print *, "Enter second vector
     "
  read *, b
```

```
!-------------------------
! calculate dot product
!-------------------------
  c = a(1)*b(1) + a(2)*b(2) + a
     (3)*b(3)
  ! c = sum(a*b)

  !-------------------------
  ! print result
  !-------------------------
  print *, "Dot product = ", c

end program dotprod
```

# Control

## Conditional

Execute different code based on some condition(s)

- if-else

```
1 if (condition) then
2   ! do something
3 else if (condition2) then
4   ! ... or maybe alternative something else
5 else
6   ! ... or at least this
7 end if
```

- Conditional execution of block of source code based on relational operators <, >, == (equal to), <=, >=, /= (not equal to), .not., .and., .or.

# Control

## Conditional

```
1  integer           :: x
2  character(len=9)  :: grade
3
4  if (x < 60) then
5     grade = 'fail'        ! < 60
6  else if (x < 70) then
7     grade = 'pass'        ! 60 to 69
8  else if (x < 80) then
9     grade = 'good'        ! 70 to 79
10 else if (x < 90) then
11    grade = 'very good'   ! 80 to 89
12 else
13    grade = 'excellent'   ! >= 90
14 end if
```

# Control

## Conditional

- switch-case

```
1  select case (selector)
2    case (label-list-1)
3      statements-1
4    case (label-list-2)
5      statements-2
6      ...
7    case (label-list-n)
8      statements-n
9    case default
10     statements-default
11 end select
```

- selector is an expression of type INTEGER, CHARACTER or LOGICAL (no REAL type can be used)

# Control

## Conditional

```
1  integer :: n, range
2
3  select case (n)
4    case ( :-10, 10: )
5      range = 1   ! <= -10 or >= 10
6    case (-5:-3, 6:9)
7      range = 2   ! -5 to -3, 6 to 9
8    case (-2:2)
9      range = 3   ! -2 to 2
10   case (3, 5)
11     range = 4   ! 3 or 5
12   case (4)
13     range = 5   ! 4
14   case default
15     range = 6   ! other conditions
16 end select
```

# Control

## Loop

### Three loop formats

```
1  ! integer counter
2  do i = start_index, end_index, step
3    statement
4  end do
5
6  ! condition controlled
7  do while (condition)
8    statement
9    cycle  ! start directly next loop
10 end do
11
12 ! explicit exit
13 do
14   statement
15   exit   ! exit the loop
16 end do
```

# Control

## Loop

### Examples

```
1  do i = 10, -10, -2
2    write(*,*) i
3  end do
4
5  do while (x > 0)
6    read(*,*) x
7    totalsum = totalsum + x
8  end do
9
10 do
11   read(*,*) x
12   if (x < 0) then
13     exit
14   else
15     totalsum = totalsum + x
16   end if
17 end do
```

# Exercise 4

Calculate the dot product in Exercise 3 with loop and print the result.

# Solution 4

```
!=============================
! dotprod.f90
! prompt for two real vectors
! of length 3
! calculate dot product
! print result
!=============================
program dotprod
  implicit none
  real :: c
  real, dimension(3) :: a, b
  !-------------------------
  ! enter data
  !-------------------------
  print *, "Enter first vector"
  read *, a
  print *, "Enter second vector
    "
  read *, b
```

```
!-------------------------
! calculate dot product
!-------------------------
  c = 0.0
  do i = 1, 3
    c = c + a(i)*b(i)
  end do

  !-------------------------
  ! print result
  !-------------------------
  print *, "Dot product = ", c

end program dotprod
```

# Procedures

- Calculations may be grouped into subroutines and functions to perform specific tasks such as:
  - read or write data
  - initialize data
  - solve a system of equations
- Function returns a single object (number, array, etc.), and usually does not alter the arguments
- Subroutine transfers calculated values (if any) through arguments
- Fortran uses pass-by-reference: change of variables' values passed into procedures will be changed after returning
- Names of dummy arguments don't have to match actual names (input variable names)

# Procedures

## Function

### Function: Convert Celsius degree to Fahrenheit degree

```
1  real function fahrenheit(c)
2    real :: c
3    fahrenheit = (9.0/5.0)*c + 32.0 ! Convert Celsius to fahrenheit
4  end function fahrenheit
5
6  function fahrenheit(c) result(f)
7    real :: c
8    real :: f
9    f = (9.0/5.0)*c + 32.0 ! Convert Celsius to fahrenheit
10 end function fahrenheit
```

### Use functions as

```
1  degF = fahrenheit(0.0)
2  degF = fahrenheit(degC)
```

# Procedures

## Subroutine

### Subroutine: converting Celsius degree to Fahrenheit degree

```
1 subroutine temp_conversion(celsius, fahrenheit)
2   real :: celsius, fahrenheit
3   fahrenheit = (9.0/5.0)*celsius + 32.0
4 end subroutine temp_conversion
```

### Use subroutine as

```
1 call temp_conversion(degC, degF)
```

# Procedures

Functions and subroutines can be contained in the program

```
1  program main
2    statements
3  contains
4
5  ! Can not write statements outside procedures within "contains"
6
7  subroutine sub(a, b)
8    ...
9  end subroutine sub
10
11 function fun(a, b)
12   ...
13 end function fun
14
15 end program main
```

They can also be put before or after the main program in source code, but using "contains" is recommended.

# Exercise 5

Modify dot-product program to use a subroutine to compute the dot product.

- Don’t forget to declare arguments
- Give the subroutine a name different than the program

# Solution 5

```
!=============================
! dotprod.f90
! prompt for two real vectors
! of length 3
! calculate dot product
! print result
!=============================
program dotprod
  implicit none
  real :: c
  real, dimension(3) :: a, b
  !------------------------
  ! enter data
  !------------------------
  print *, "Enter 1st vector"
  read *, a
  print *, "Enter 2nd vector"
  read *, b
  !------------------------
  ! Calculate dot product
  !------------------------
  call dp(a,b,c)

  !------------------------
  ! print result
  !------------------------
  print *, "Dot product = ", c

contains

  subroutine dp(x,y,d)
    implicit none
    real :: d
    real, dimension(3) :: x, y
    d = sum(x*y)
  end subroutine dp

end program dotprod
```

# Exercise 6

1. Modify dot-product program to use a function to compute the dot product
2. Modify the Fahrenheit function into a function, converts, such that if input is in Fahrenheit, it returns the Celsius equivalence. If input is in Celsius, it returns Fahrenheit. (Hint: extra input parameter)

# Solution 6

## 1 Dot product function

```
function dotp(x,y)
  implicit none
  real :: dotp
  real, dimension(3) :: x, y
  dotp = sum(x*y)
end function dotp
```

## 2 Convert function

```
real function converts(temp, mode)
  real :: temp
  integer :: mode

  if (mode == 0) then  ! Celcius to Fahrenheit
    converts = (9.0/5.0)*temp + 32.0
  else  ! Fahrenheit to Celcius
    converts = (temp-  32.0) * (5.0/9.0)
  end if
end function converts
```

# Module

- A Fortran module can contain procedures, variables and data structure definitions
  - Grouping variables and procedures
  - Declaring "global" variables
  - Use "contains" to contain procedures

```
1 module module_name
2   implicit none
3   variable declarations
4 contains
5   procedure definitions
6 end module module_name
```

# Module

- In a program unit that needs to access the components of a module we need to write:

```
use module_name
```

- Use statement must be before implicit none
- use statement may specify specific components to access by using "only" qualifier:

```
use solvers_mod, only: nvals, x
```

# Module

## main.f90:

```
1  program main
2
3    use geometry_mod, only : dist
4
5    implicit none
6
7    real :: d  ! I am not the
       guys in geometry_mod
8
9    call dist(2.0, 3.4, d)
10
11   write(*,*) d
12
13 end program testprog
```

## geometry_mod.f90:

```
1  module geometry_mod
2
3    implicit none
4
5    real :: d  ! I am not the guy
       below
6
7  contains
8
9    subroutine dist(x, y, d)
10     real :: x, y
11     real :: d  ! I am not the
       guy above
12     d = sqrt(x**2 + y**2)
13   end subroutine dist
14
15 end module geometry_mod
```

# Module

- Fortran style suggestions
  - Group global variables in modules based on the module goal
  - Name modules (and associated files) with "_mod" in the name, e.g., solvers_mod, solvers_mod.f90
  - Employ "use only" for all variables required in program unit
  - All variables then appear at top of program unit in declarations or "use" statements

# Module

Compile the module files and the program file: compile module files first, then compile the program file, link them to generate executable file finally.

```
$ gfortran -c geometry_mod.f90  # get geometry_mod.o and
  ↪  geometry_mod.mod
$ gfortran -c main.f90  # get main.o
$ gfortran -o main.exe geometry_mod.o main.o
$ ./main.exe  # run the code
```

# Exercise 7

Put the dot product subroutine or function to a module, then use the module in the main program to access the procedure.

- save the module to a separate file
- compile the module file and the main program file, link them and run the code

# Solution 7

## main.f90

```
1  program dotprod
2
3    use math_mod
4
5    implicit none
6
7    ...
8
9    call dp(a,b,c)
10
11   ...
12
13 end program dotprod
```

## math_mod.f90

```
1  module math_mod
2
3  contains
4
5    subroutine dp(x,y,d)
6      implicit none
7      real :: d
8      real, dimension(3) :: x, y
9      d = sum(x*y)
10   end subroutine dp
11
12 end module math_mod
```

## compile and run

```
$ gfortran -c math_mod.f90
$ gfortran -c main.f90
$ gfortran -o main.exe math_mod.o main.o
$ ./main.exe
```

# Make

- Large projects
  - consist of multiple files, including the main program, module files and maybe procedure files
  - bad practice to put everything in the same file
  - recommended to create a separate file for each module, and group all the procedures to different modules
  - easier to read, edit, understand, as well as more efficient to compile, debug, maintain and develop.
- We can compile the project in the way as shown above, but it is not flexible and needs to write the commands every time. The command make is used to manage projects and make the compilation easier.

# Make

- make is a Unix utility to help manage codes
- When you type "make" in the command line, it will look for a file called "makefile" or "Makefile", or the specified name
- Makefile is a file that tells the make utility what to do
- Usage:

```
$ make
(or) $ make -f <makefile_name>
```

- Check the full manual of 'GNU make' at `http://www.gnu.org/software/make/manual/make.html`

# Make

## makefile

- Makefile contains different sections with different functions, the sections are not executed in order!
- Comment character is #, line continuation is \
- There are defaults for some values, but we recommend to define everything explicitly
- Variables
  - define variables like: `F90 = gfortran`
  - no quotes are required, and string may contain spaces
  - use variable as `$(F90)` or `${F90}`

# Make

- In makefile, you define the rules

```
target: prerequisites
<tab>recipe
```

  - The target is any name you choose, often use the name of executable
  - Prerequisites are files that are required by target, e.g., executable requires object files
  - Recipe tells what you want the makefile to do
- make will
  - search for the first target in the makefile
  - checks the time stamps on the prerequisites
  - if anyone is newer, make will update it
  - once all the prerequisites are updated as required, it performs the recipe

# Make

## Example

```
# compiler
F90 = gfortran

# objects
OBJ = geometry_mod.o  main.o

# compile and link
main.exe: $(OBJ)
<tab>$(F90) $(OBJ) -o main.exe

geometry_mod.o: geometry_mod.f90
<tab>$(F90) -c geometry_mod.f90 -o geometry_mod.o

main.o: main.f90
<tab>$(F90) -c main.f90 -o main.o

# Clean object files and executable file
clean:
<tab>rm -f *.o *.exe

# When there are multiple targets, specify desired target as argument to make command,
# otherwise the first target will be used
# $ make
# $ make clean
```

# Exercise 8

Write a makefile for any of the previous exercises. Test your makefile with the make command.

# Input/Output

- List-directed output, `print *` or `write(*,*)`, gives little control
- Use formatted output (`read` for input):

  `write(unit, format) variables`
- Here unit is an integer number indicating where you want to write data, some units are usually reserved and you should not use for your files:
  - stdin (read from screen): 5
  - stdout (write to screen): 6
  - stderr: 0
- An example of writing to a file

```
1 open(11, file="mydata.dat")  ! open a file named mydata.dat
2                              ! 11 is file unit number
3 write(11, *) 123  ! write to the file with the same unit
4 close(11)         ! close the file when you finished writing
```

# Input/Output

## Format

- Definitions
  - w: output width
  - d: number of digits after the decimal point
  - m: minimum number of characters
  - e: exponential digits
- format are included between "( and )".
- the default is right-justified
- all of them also fit to "read"

# Input/Output

- example table

| Data type | Format | Example | Output | Comments |
|---|---|---|---|---|
| integer | Iw, Iw.m | `write(*, "(i5.3)") 12` | `␣␣012` | padding with zeros |
| | | `write(*, "(i5.3)") 1234` | `␣1234` | |
| | | `write(*, "(i5.3)") 123456` | `*****` | showing * when exceeding w |
| string | A, Aw | `write(*, "(a)") "hello"` | `hello` | output any length |
| | | `write(*, "(a3)") "hello"` | `hel` | take first w |
| | | `write(*, "(a3)") "hi"` | `␣hi` | |
| decimal | Fw.d | `write(*, "(f5.2)") 1.2` | `␣1.20` | tailing zeros |
| | | `write(*, "(f5.2)") 1.226` | `␣1.23` | round the number |
| | | `write(*, "(f5.2)") 123.2` | `*****` | too long |
| exponential | Ew.d | `write(*, "(e8.1)") 123.2` | `␣0.1E+03` | w >= 6 + d |
| scientific | ESw.d | `write(*, "(es8.1)") 123.2` | `␣1.2E+02` | w >= 6 + d |
| | ESw.dEe | `write(*, "(es8.1e3)") 0.1232` | `1.2E-001` | w >= 4 + d + e |

- The formats can be combined with commas

  write(*, "(a, f6.2, i5, es15.3)") "answers are ", x, j, y

- You can check a full list of formats here:

  https://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/chap05/format.html

# Exercise 9

Write the result of dot product to a file.

# Solution 9

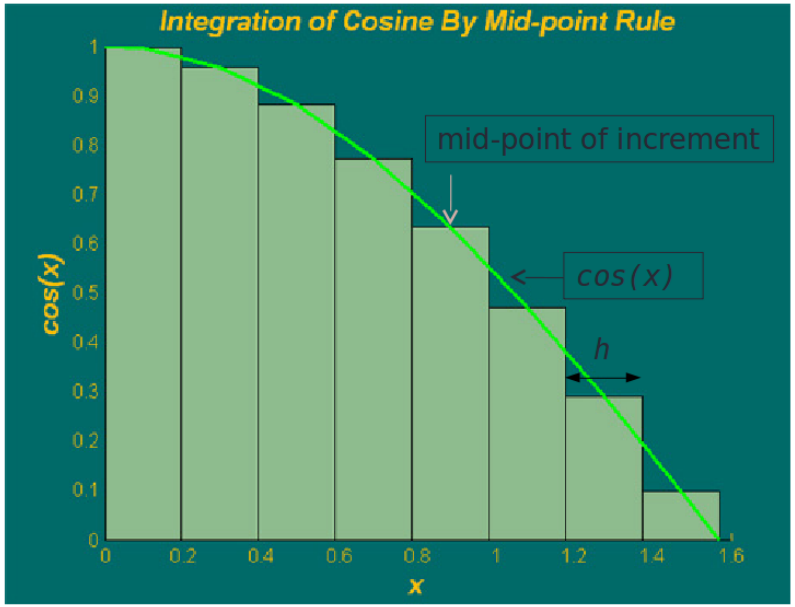
```
program main

...

!----------------------------
! write result to file
!----------------------------
open(21, file="dot_product.dat")
write(21, "(a, f6.3)") "dot product = ", c
close(21)
write(*,*)  ! an empty line
write(*,*) "Output written to file dot_product.dat"

...

end program main
```

# Exercise 10

## Integration of cosine

- Integration of cosine from 0 to $\pi/2$ with mid-point rule
- Integral ≈ sum of rectangles (height * width)
  $\int_a^b \cos(x)\mathrm{d}x \approx \sum_{i=1}^{m} \cos[a+(i-0.5)h]\cdot h$
- parameters as an example:

```
1 a = 0; b = pi/2   ! range
2 m = 8             ! # of increments
3 h = (b-a)/m       ! increment
```



# Exercise 10

Write a program to perform the integration of cosine using the mid-point rule

- Write a function integral to perform integration, $m$, $a$, $h$ are input to integral
- The main program calls integral a few times (using do loop), each time with a larger m than the previous time. The purpose is to study the convergence trend. (hint: you can use `m=25*2**n`; $n$ is the loop index)

# Solution 10

```
1  program integration_with_doloop
2
3  ! This program computes the numerical
       integration of cosine function
4  !
5  ! Written by: Kadin Tseng
6  ! Date written: September 19, 2012
7
8  implicit none
9  real, parameter :: pi=3.141593
10 real :: a, b, h, integ, integral
11 integer :: n, m
12
13 a = 0.0   ! lower limit of integration
14 b = pi/2  ! upper limit of integration
15 do n=1, 4  ! number of cases to study
16   m = 25*2**n    ! number of increments
17   h = (b - a)/m  ! increment length
18   integ = integral(a, h, m)
19   write(*,*) "No. of increments = ", m, &
20     "  Integral value is ", integ
21 end do
```

```
1  contains
2
3  real function integral(a, h, m)
4  ! performs midpoint integration
5
6  implicit none
7
8  real :: a, h, x
9  integer :: m, i
10
11 integral = 0.0  ! initialize integral
12
13 do i=1, m
14   ! mid-point of increment i
15   x = a+(i-0.5)*h
16   integral = integral + cos(x)*h
17 end do
18
19 end function integral
20
21 end program integration_with_doloop
```

# References

- Lots of books available, e.g., "Fortran 95/2003 Explained" by Metcalf, Reid, and Cohen is good
- Gfortran: `http://gcc.gnu.org/wiki/Gfortran`
- Fortran wiki: `http://fortranwiki.org/fortran/show/HomePage`
- CSC training course: `https://github.com/csc-training/fortran-introduction`
- Use google