Brief introduction to Fortran90

By

Nicole Mölders

Motivation

- Various scientific problems are too large to do them by “hand” on a calculator or napkin
- Super-computers can one thing better than we do, namely adding more quickly
  - Otherwise they are very stupid (Danger!!!!)
- Computers offer chance to get long and complicated time-consuming calculations done in an affordable time frame

Motivation (cont.)

Challenge:

- They do not understand English
- They cannot think or interpret what we want them to do

Consequences:

- We need a language to tell them
- You get what you code (which may not be what you want the computer to do)

Flow diagrams

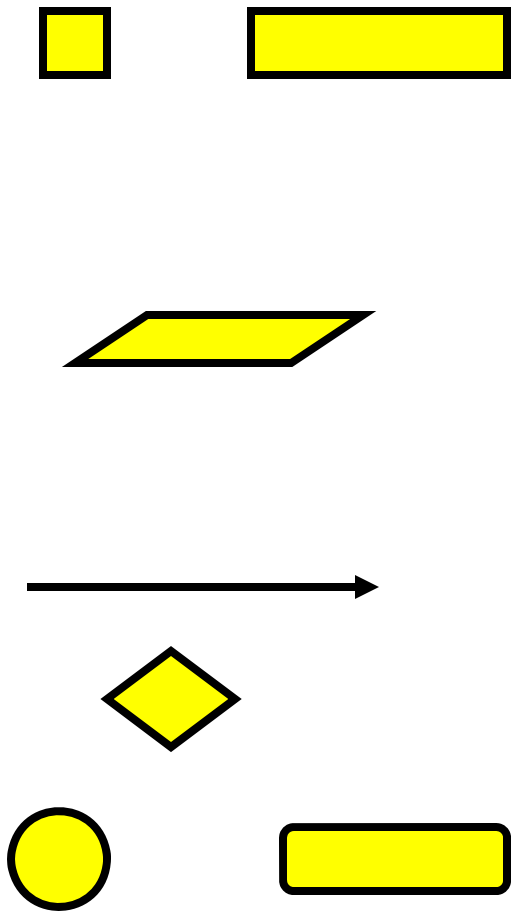
- *Flow diagrams (FDs)* help to structure, analyze and design programs

- FDs illustrate the paths, possibilities, decision trees, data flow and its logical input, storage and output

Flow diagrams (cont.)

There are several symbols:

- Squares or rectangular boxes represent *processes*, i.e. what to do
- Routs represent *input/output*, i.e. which data to take as input, where to write the output (data sources or destinations of data)
- Arrows represent the *data flows*
- Diamonds indicate decisions
- Circles or rounded rectangles indicate beginning/end of program/subroutine



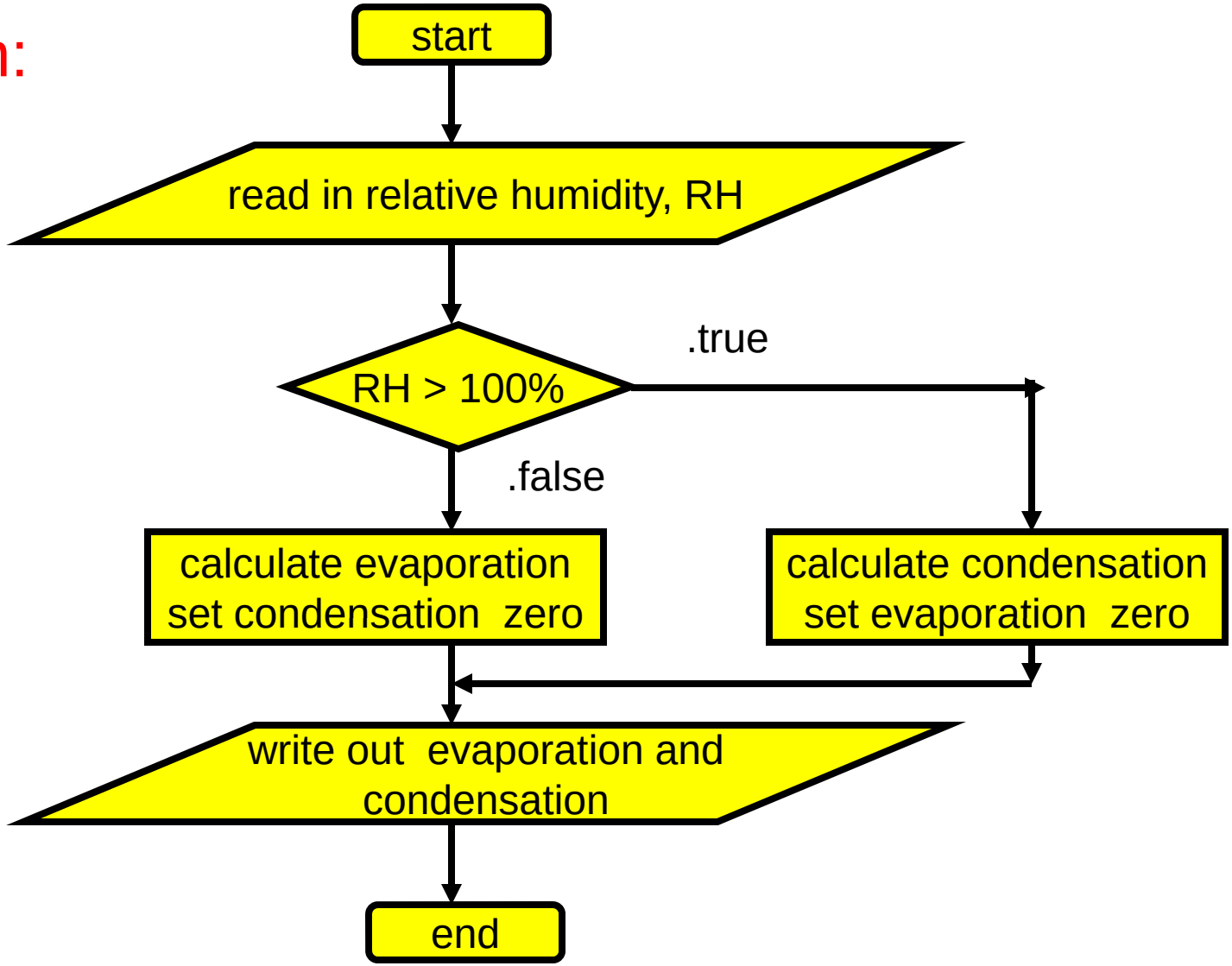
Flow diagrams (cont.)

Example:

Design a flow chart for condensation and evaporation

Flow diagrams (cont.)

Solution:



From the FD to the code

- To transfer the flow diagram into a program we need a computer language
- Like English or other languages they have a “grammar” (structure) and vocabulary (commands, types, variables, etc.)
- In addition, there are “names” (fortran identifiers = you name variables/quantities) that do not belong to the FORTRAN language
- An identifier never can be taken out of the group of “legal” fortran vocabulary!

Example:

do, if, and, program, read, end, subroutine, dimension, real, integer, character, endif, enddo cannot be used as identifiers

Program structure

PROGRAM program-name
[purpose, author, version, changes/bug fixes]
IMPLICIT NONE
[specification part]
[execution part]
[subprogram part]
END PROGRAM program-name

- Contents in [ ] are optional, but recommended
- Keyword IMPLICIT NONE is optional, but recommended
- A program starts with PROGRAM followed by the program's name,
- Next optional line is the IMPLICIT NONE statement (mandatory for this class and in my lab)
- *specification statements* (e.g. dimensions, real, integer, etc.)
- *execution part* (what it is to do)
- set of *internal subprograms*,
- Ends with the keywords END PROGRAM and the program's name
- For improving readability, add comment lines (starting with !)

Program structure (cont.)

- A program consists at least of the line stating its name, a statement of action to be undertaken and the end statement

Example:

```
program filename      ! name of the program
a=5+8                 ! statement of action
end filename          ! indicator of program's end
```

Program structure (cont.)

- A good program should also contain info on the
  - Author
  - Version
  - History
  - Purpose
  - data needed to run the program
  - Output
  - subroutines called by the program (or if the program is a subroutine the calling program)

Reading

In FORTRAN one can read from the keyboard, or a defined file.

Note that logical unit 5 is reserved for the keyboard. Sometimes units to 12 are assigned to mass storage or magnetic tape machines. Thus, better use higher unit values than 12.

Example:

```
read(5,*)word  (read value of word from keyboard)

open(26,file='filename.dat',form='formatted',status='old')
read(26,*)word  (read values of "word" from filename.dat
    defined by the open statement)
```

Fortran Comments

- Fortran compilers ignore all characters following an exclamation mark, **!**, except in a character string
- A blanc line is a comment line (you should use this to make your program more readable)

Examples:

```
PROGRAM CommentTest1
READ(*,*) year    ! read in the value of year
! This is a comment line in a program
year=1
year = year + 1    ! add 1 to year
END PROGRAM CommentTest1
```

```
PROGRAM CommentTest2
READ(*,*) count     ! The above blank line
                    ! is a comment line
WRITE(*,*) Count + 2
END PROGRAM CommentTest2
```

Fortran Continuation Lines

- A line ending with an ampersand, **&**, is continued on the next line
- Continuation is to the first character of the next non-comment line
- You can start anywhere in the next line for continuation

Example:

```
A = 55.5 * B &
    + C/ 100
```

The above is equivalent to

```
A = 55.5 *B + C / 100
```

Note that **&** is not part of the statement

```
A = 55.5 * B &
! this is a comment line, a blank line would be also a comment
    + C / 100
```

The above is equivalent to

```
A = 55.5 * B + C / 100
```

because the compiler ignores all comments

Fortran Alphabets

- **Letters:**

  a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- **Digits:**

  0 1 2 3 4 5 6 7 8 9

- **Special Characters:**

  space ( ) * + - / < > : = ' " _ ! & $; % ? , .

Fortran Identifiers

A fortran identifier must satisfy the following rules:

- no more than 31 characters
- the first character must be a *letter*,
- all remaining characters, if any, may be *letters, digits, or underscores*

Except for strings fortran identifiers are not case sensitive, i.e., **Schmit**, **schmit**, **schMiT**, **SCHMiT**, **SCHMIT** are all identical

Fortran Identifiers (cont.)

- **Examples:**

  MIT, MAC, Tango, Year, I, X, Y0815, a1b2c3, XYZ99a, X2_Y2, R2D2_, A___

Use meaningful names

field, forest, meadow, river or field1, field2, field3, field4 are not good names!

temp, rain, rho, pres are good names for variables that hold temperature, liquid precipitation, density, and pressure, respectively

Fortran Variables and Types

- **INTEGER**: variable holding an integer
- **REAL**: variable holding a real number
  - Note that if you do not program with IMPLICIT NONE fortran uses the first letter of a variable as indicator for whether the variable is and integer or real
  - The result of that may not be what you want!!!!!
- **COMPLEX**: variable holding a complex number
- **LOGICAL**: variable holding a logical value (i.e., **true** or **false**)
- **CHARACTER**: variable holding a character string of certain length

Fortran Constants

Integer Constants: a string of digits with a preceding optional sign

Examples: 0, 8762, -9746, +13892

Examples for what is not an integer:

23,1234 (comma is not allowed)
67.0 (no decimal point)
--1 or ++5 (too many optional signs)

Fortran Constants (cont.)

Real Constants: decimal representation or exponential representation with preceding optional sign. A decimal point must be presented, but no commas are allowed.

Example for decimal representation: 24.5, .312, 678., -0.67, -.12

Exponential Representation: consists of an integer or a real number in decimal representation (the mantissa or fractional part) followed by **E** or **e** and an integer (the exponent)

Examples

0.E0 or 0.e0	(equivalent to 0.0)
-5.6E3 or -5.6e3	(equivalent to -5600.)
3.14E0 or 3.14e0	(equivalent to 3.14)
-1.2E-1 or -1.2e-1	(equivalent to -0.0012)
67.E1 or 67.e1	(equivalent to 670.)

Examples for what is not a real:

32,345.45	(no comma is allowed)
57	(a decimal point is a must)
3.4E5.6	(exponent must be an integer)
34.12-5	(exponential sign **E** or **e is missing)**

Fortran Constants (cont.)

- Note that there are prescribed INTEGER and REAL constants unless you set the command **implicit none**
- **The letters i,j,k,l,m,n would be integer, all other would be real**

Why you should use “IMPLICIT NONE”

- You can use memorable variable names that start with i,j,k,l,m,n, e.g. Karman_const
- It is safer as you can easily find typos

Example

```
…
! Case without implicit none …
Test=tset*6.       ! tset is typo and not defined => Result of test=0.
…

If you set implicit none

Implicit none
Real :: test
Test=tset*6.     ! This would give you a compilation error and
!                    you would find your typo
….
```

Fortran Constants (cont.)

Character String: The content consists of all characters, spaces included, between single or double quotes, while the *length* of the string is the number of characters of its content. If the content is zero it will be called an *empty string*

Examples:

‘Bert' and **“Bert“** (content = **Bert,** length = 4)
' ' and **" “** (content = a single space, length = 1)
‘Ernie & Bert ' and **"Ernie & Bert “** (content = **Ernie & Bert**, length = 12)
'' and **““** (content = nothing, length = 0, (empty string))

Incorrect Examples:

‘Ernie and Bert (closing apostrophe is missing)
Hello, Mr. Sandman‘ (opening apostrophe is missing)
'Hi" and **"Hi‘** (mismatch of opening and closing quotes)

Note: If single quote is used in a string, then double quotes should be used to enclose the string, e.g. “Bert's car“ (content= **Bert's car**, length 10) Alternatively, ‘Bert”s car’. Compilers treat a pair of single quotes in the content of a string as one, i.e. the content is still **Bert's car**.

Fortran Constants (cont.)

Complex: not covered in this course as usually not used in Atmospheric Sciences problems

Logical: True or False

Fortran Variable Declarations

Declaring the type of a Fortran variable is done with **type statements** of the form: type-specifier :: list. Here the *type-specifier* is either integer, real, complex, logical or character and *list* is a list of variable names separated with commas

Examples:

Assume variables **ABC**, **Mean** and **stdev** are of type **INTEGER**.

```
INTEGER :: ABC, Mean, stdev
```

Assume variables **mean**, **error**, and **bias** are of type **REAL**.

```
REAL :: mean, error, bias
```

Or

```
real*8 :: mean, error, bias
```

Fortran Variable Declarations (cont.)

- Fortran has vectors and matrices

Example:

```
real :: vector(6)
real :: matrix(3,3)
integer :: vector(5)
integer :: matrix(4,4)
```

Fortran Variable Declarations (cont.)

In type **CHARACTER** a string has a length attribute, and hence a length value must be attached to character variable declarations. There are two ways to do this:

CHARACTER(LEN=*i*) declares character variables of length *I*

CHARACTER(*i*) declares character variables of length *I*, i.e. there is no **LEN=** in the parenthesis

Examples:

CHARACTER(LEN=20) :: City, Street

CHARACTER(20) :: City, Street

(City and **Street** are character variables that can hold a string of no more than 20 characters)

CHARACTER(LEN=15) :: FirstName, MiddleName, LastName

CHARACTER(5) :: FirstName, MiddleName, LastName

(FirstName , MiddleName , LastName are character variables that can hold a string of no more than 15 characters)

Fortran Variable Declarations (cont.)

To declare character variables of different length with a single statement, a length specification, ***i*, has to be attached to the right of a variable. Thus, only the corresponding variable has the indicated length, all other variables are *not* affected

Example:

CHARACTER(LEN=10) :: City, state*2, BOX, country*20

Here, variables **City** and **BOX** can hold a string of no more than 10 characters, **state** can hold a string of no more than 2 characters, and **country** can hold only 20 characters

There is one more way of specifying the length of a character variable (see good FORTRAN books)

PARAMETER Attribute

Often one just wants to assign a name to a particular value that is used several times and remains constant.

For example, 3.1415926 so that one could use **PI** rather than 3.1415926. Assigning a name to a value works as follows:

Add **PARAMETER** in front of the double colon (**::**) and use a comma to separate the type name (i.e. **REAL**) and the word **PARAMETER**. Following each name add an equal sign (**=**) followed by an expression. The value is then assigned the indicated name.

Note that the name assigned to a value is an alias of the value, *not* a variable. After assigning a name to a value, the name can be used in a program, even in subsequent type statements.

PARAMETER Attribute (cont.)

Examples:

```
INTEGER, PARAMETER :: ISTOP = 10, Min_Count = 80
```

(**ISTOP** is a name for the integer value 10, while **Min_Count** is a name for the integer value 80)

```
REAL, PARAMETER :: BB = 2.71828, PI = 3.141592
```

```
INTEGER, PARAMETER :: all = 10, iiter = 5, prod = all*iiter
```

```
CHARACTER(LEN=5), PARAMETER :: Name1 = ‘Ernie', &
                               Name2 = “Bert "
```

PARAMETER Attribute (cont.)

When to assign a name to a string?

Fortran allows the length of character name to be determined by the length of a string

CHARACTER(LEN=4), PARAMETER :: Name = ‘Roger'

If the string is longer, truncation to the right will happen Since the length of the string **“Roger"** is 5 while the length of **Name** is 4, the string is truncated to the right and the content of **Name** is **“Roge“**

CHARACTER(LEN=4), PARAMETER :: Name = “Jo"

If the string is shorter, spaces will be added to the right. Since the string **“Jo"** is of length 2 while **Name** is of length 4, two spaces will be padded to the right and the content of **Name** becomes **“Jo “**

CHARACTER(LEN=*), PARAMETER :: Name1 = ‘Bert', &
Name2 = "Li"

i.e. an *assumed length specifier* comes in. In the examples above, names **Name1** and **Name2** are declared to have an assumed length. Since the lengths of **‘Bert'** and **“Li"** are 4 and 2, the length of the names **Name1** and **Name2** are 4 and 2, respectively.

Data statement

- The data statement allows you to assign values, or characters

Example

```
CHARACTER(LEN=14) :: station(4)
DATA station /”Ho_Chi_Min.dat”,”Duisburg.dat  “,&
                “Fairbanks.dat “,”North_Pole.dat”/
```

Variable Initialization

A *variable* can be considered as a box holding a single value. Initially the content of a variable is empty. Thus, before use, the variable **must** receive a value.

Warning: Do not assume the compiler or computer will put some value, say 0, into a variable.

Variable Initialization (cont.)

There are three ways to assign a value into a variable:

1. *initializing* it when running the program
2. using an *assignment statement*
3. reading a value from keyboard or other device by using a **READ** statement

The initializing procedure is similar to the use of the **PARAMETER** attribute:

add an equal sign (**=**) to the right of a variable name to the right of the equal sign, write an expression. Note that all names in the expression *must* be constants or names of constants.

Initializing a variable is only done exactly *once* when the computer loads your program into memory for execution, i.e. all initializations are done *before* the program starts execution.

Variable Initialization (cont.)

Warning: Using non-initialized variables may cause unexpected result!

Examples:

REAL :: diff = 0.5, accuracy = 1.E-4

(Initializes variables **diff** to 0.5, and **accuracy** to 1.E-4)

CHARACTER(LEN=4) :: Name1 = “John", Name2 = “Gerd", Name3 = “Mary"

(Initializes variables **Name1** to “John", **Name2** to “Gerd", and **Name3** to "Mary“)

INTEGER, PARAMETER :: Q = 10, A = 435, B = 3 INTEGER :: P=Q*A, R=B+5

(defines 3 named integer constants with **PARAMETER** and uses these values to initialize two integer variables, i.e. variables **P** and **R** are initialized to have values 4350 (=10*435) and 8 (3+5), respectively)

Exercise: Find the mistake

INTEGER, PARAMETER :: Q=10, A=435 INTEGER :: P=Q*A R=P+5

INTEGER, PARAMETER :: P=C

INTEGER, PARAMETER :: Q=10, A=435,

INTEGER :: P=Q*A, R=P+5

INTEGER, PARAMETER :: P=C →?C

Arithmetic Operators

- In addition to addition **+**, subtraction **-**, multiplication ***** and division ***/***, Fortran has an *exponential operator* ******. Raising **X** to the **Y**th power is written as **X**Y**. The exponential operator has highest priority.

- Operators **+** and **-** can also be used as *unary* operators, i.e. they only need one operand, e.g. **-X** and **+Y**. The former means change the sign of **X**, while the latter is equivalent to **Y**.

Arithmetic Operators (cont.)

- Unary operators **+** and **-** have the same priority as their binary counterparts (*i.e.*, addition **+** and subtraction **-**). As a result, since ****** is higher than the negative sign **-**, -6**4 is equivalent to -(6**4), which is -1296

- For arithmetic operators, the exponential operator ****** is evaluated from right to left, i **2**3**4** is equal to **2**(3**4)** (=2.4e24) ratl than **(2**3)**4** (=4096)

Arithmetic Operators (cont.)

Remember: the operator on the top-most row (**) has the highest priority (*i.e.*, it is evaluated first) while the operators on the bottom-most row (*i.e.*, **.EQV.** and **.NEQV.**) have the lowest priority. The operators on the same row have the same priority and the order of evaluation is based on their associativity law

Type	***Operator***	***Associativity***
Ar**ithmetic**	**	right to left
	* / + -	left to right
Relational	< <= > >= == /=	none
Logical	**.NOT.**	right to left
	.AND.	left to right
	.OR.	left to right
	.EQV. **.NEQV.**	left to right

Single Mode Arithmetic Expressions

Operator	INTEGER	REAL
INTEGER	**INTEGER**	*mixed mode*
REAL	*mixed mode*	**REAL**

Examples:

2 + 3 is **5**

4 * 8 is **32**

-42** is **-16**

16/4 is **4**

21/4 is **5** rather than **5.25** because **21/4** is a single mode arithmetic expression and all of its operands are of **INTEGER** type. Thus, the result must also be of **INTEGER** type. The computer will *truncate* the mathematical result (3.25) making it an integer.

1.23 - 0.45 is **0.78**

3.6/2.4 is **1.5**

4.4/2.2 is **2.0** rather than **2**, because the result must be of **REAL** type

6/2.5=3 mixed mode

6./2.5=2.4

Attention: Truncation is handled differently by various compilers!

3/5 is **0** (compilers that just cut off)

3/5 is **1** (compilers that round for fractions greater equal 0.5)

The first is pretty standard, but better check than being sorry

Single Mode Arithmetic Expressions

Rules for evaluating expressions

- Expressions are always evaluated from *left to right*
- If an operator is encountered in the process of evaluation, its *priority* is compared with that of the next one

Single Mode Arithmetic Expressions (cont.)

1. if the next one is lower, evaluate the current operator with its operands 3 * 4 – 5. Here, in the left to right scan, operator ***** is encountered first. Since the operator **-** is lower, **3 * 4** is evaluated first transforming the expression to **12 – 5**, and the result is **7**

Single Mode Arithmetic Expressions (cont.)

2. if the next one is equal to the current, the *associativity rules* are used to determine which one should be evaluated. For example, if both the current and the next operators are *****, then **3 * 4 * 5** will be evaluated as **(3 * 4) * 5**. On the other hand, if the operator is ******, **A ** B ** C** will be evaluated as **A ** (B ** C)**

Single Mode Arithmetic Expressions (cont.)

3. if the next one is higher than the current, the scan should continue with the next operator. For example, consider the following expression: 4 + 5 * 6 ** 7 if the current operator is **+**, since the next operator ***** has higher priority, the scan continues to *****. Once the scan arrives at *, since the next operator ****** is higher, **6 ** 7** is evaluated first, transforming the expression to 4 + 5 * 279936. The next operator to be evaluated is *****, followed by **+**. Thus, the original expression is evaluated as **4 + (5 * (6 ** 7))** with the result 1399684

Single Mode Arithmetic Expressions

Exercises:

In the following examples, brackets are used to indicated the order of evaluation

- 2 * 4 * 5 / 3 ** 2

  --> [2 * 4] * 5 / 3 ** 2 --> 8 * 5 / 3 ** 2 --> [8 * 5] / 3 ** 2 --> 40 / 3 ** 2 --> 40 / [3 ** 2] --> 40 / 9 --> 4 (The result is 4 rather than 4.444444 since the operands are all integers)

- 100 + (1 + 250 / 100) ** 3

  --> 100 + (1 + [250 / 100]) ** 3 --> 100 + (1 + 2) ** 3 --> 100 + ([1 + 2]) ** 3 --> 100 + 3 ** 3 --> 100 + [3 ** 3] --> 100 + 27 --> 127 (sub-expressions in parenthesis must be evaluated first, values are all integers)

- 1.0 + 2.0 * 3.0 / (6.0*6.0 + 5.0*44.0) ** 0.25

  --> 1.0 + [2.0 * 3.0] / (6.0*6.0 + 5.0*44.0) ** 0.25 --> 1.0 + 6.0 / (6.0*6.0 + 5.0*55.0) ** 0.25 --> 1.0 + 6.0 / ([6.0*6.0] + 5.0*44.0) ** 0.25 --> 1.0 + 6.0 / (36.0 + 5.0*44.0) ** 0.25 --> 1.0 + 6.0 / (36.0 + [5.0*44.0]) ** 0.25 --> 1.0 + 6.0 / (36.0 + 220.0) ** 0.25 --> 1.0 + 6.0 / ([36.0 + 220.0]) ** 0.25 --> 1.0 + 6.0 / 256.0 ** 0.25 --> 1.0 + 6.0 / [256.0 ** 0.25] --> 1.0 + 6.0 / 4.0 --> 1.0 + [6.0 / 4.0] --> 1.0 + 1.5 --> 2.5 (**x**0.25** is equivalent to computing the fourth root of **x**. In general, taking the *k-th* root of **x** is equivalent to **x**(1.0/k)** in FORTRAN, where *k* is a real number)

Printing

In FORTRAN you can write on screen, in an unassigned file having a logical unit, or a defined file.

The logical unit 6 is reserved for the screen

Example:

write(6,*)word (prints values of word on screen)

write(66,*)word (prints values of word in fort.66)

open(26,file='filename.dat',form='formatted',status='new')

write(26,*)word (prints values of word on filename.dat defined by the open statement)

Note that the status can be new, old, unknown, and the format can be formatted or unformatted

Formatted printing/reading

1. Real values can be formatted by the F-format of form Fa.b where the a gives the total number to be printed/read and the b stands for the number of values behind the point
2. Note that the “point” in a real number also counts
3. Integer values are given by Ia where a give how long the integer to be printed may be and I indicates that integers are done
4. Characters are printed either in quotas or by the H-format Ha where a lists how many characters are to be printed and H indicates that characters are done
5. In either case, if the number/character is too long you will receive ******* to indicate that the number exceeds the slots assigned
6. Spaces a given by aX where a indicates how many spaces are to be done and X indicates that spaces are done

Formatted printing/reading (cont.)

Example:

temp=273.15

pres=1013.25

precip_event=0

Rain="rain"

write(6,120)temp,pres,precip_event,Rain

120 format(1x,f7.3,1x,f6.1,1x,I1,1x,H4)

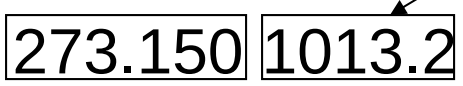
Result:

273.150 1013.2 0 rain

Or alternatively

Write(6, (1x,f7.3,1x,f6.1,1x,I1,1x,H4))temp,pres,precip_event

Result:



Loops

- If actions have to be repeated for a certain number of times loops are the way to go
- Loops have an open statement (do) that tells how often the action is to be repeated and an end statement (enddo) indicating where the loop ends

Example: sum up the numbers from 1 to 100

```
real :: sum100
sum100=0.                    ! initializes the scalar with 0
do i=0,99                    ! run loop from 0 to 99
sum100=sum100+i+1
enddo
write(6,*)'sum100=',sum100   ! Print result
```

Result: one number

Loops (cont.)

- Loops can be nested

Example: sum up numbers on a vector field

```
real :: sum100(0:99,0:99)                ! set dimensions
do i=0,99
do j=0,99
sum100(i,j)=0.                           ! initializes the field with 0
enddo
enddo
do i=0,99
do j=0,99
sum100(i,j)=sum100(i,j)+j+1   ! do the addition
enddo
enddo
print,'sum100=',sum100
```

Result: prints hundred times hundred numbers. Explain how they differ from the previous example and why.

Conditions

- Some problems require different actions for different conditions
- FORTRAN provides the if statement for this
- FORTRAN permits to assign different actions for more than just one case

Example: calculate how many water freezes at a certain temperature below freezing

```
if(t < 273.15) then
tdif=t-273.15
dice=tdif*cice/Lf
endif
```

or alternatively

```
if(t < 273.15)dice=(t-273.15)*cice/Lf
```

Conditions (cont.)

Example for different actions

```
if(t < 273.15) then
dice=(t-273.15)*cice/Lf     !freezing
else
dice=(273.15-t)*cice/Lf     ! Melting
endif
```

Conditions (cont.)

Example for different actions when several things have to be fulfilled

```
if(t > 273.15 .and. ice > 0.) then
dice=(273.15-t)*cice/Lf            ! melting
hlp(0)=0.
hlp(1)=ice-dice
ice=max(hlp)                       ! not more than ice available melts
else
  if(t > 273.15)then
  dice=0.                          ! Nothing happens, there is no ice
  else
  dice=(t-273.15)*cice/Lf          ! freezing
  ice=ice+dice                     ! new ice amount
  endif
endif
```

Program structure (cont.)

Example 1:

```
program filename                          ! name of the program
! purpose: sum numbers from 1 to 100
! author: Donald Duck
! version: 11-11-2002
! data requirements: none
! output: sum of numbers from 1 to 100 on file OutPut.dat
! calls: none
! main program
open(14,file='OutPut.dat',form='formatted,status='unknown')
 …..

write(14,*)'sum100=',sum100
close(14)      ! close the file on which the results in written
end            ! indicator of program's end
```

Here 14 is the logical unit that identifies your file. You can chose any number as logical unit > 12 and <100.

Program structure (cont.)

Example 2:

```
program filename                                    ! name of the program
! purpose: sum numbers from 1 to 100
! author: TTT
! version: 3-2-2009
! data requirements: none
! output: sum of numbers from 1 to 100 on file OutPut.dat
! calls: none
! main program
real :: sum100(50,45)
open(14,file='InPut.dat',form='formatted,status='old')
open(15,file='OutPut.dat',form='formatted,status='unknown')
read(14,*) sum100
…..                              ! Here some calculations are made that modify sum100

write(15,*)'sum100=',sum100
close(14)          ! close the file from which data were read
close(15)          ! close the file on which the results in written
end                ! indicator of program's end
```

Program structure (cont.)

Example 3:

```
program filename                                    ! name of the program
! purpose: sum numbers from 1 to 100
! author: TTT
! version: 3-2-2009
! data requirements: none
! output: sum of numbers from 1 to 100 on file OutPut.dat
! calls: none
! main program
real :: sum100(50,45)
open(14,file='InPut.dat',form='formatted,status='old')
read(14,*) sum100
close(14)          ! close the file from which data were read
…..                          ! Here some calculations are made that modify sum100

open(14,file='OutPut.dat',form='formatted,status='unknown')
write(14,*)'sum100=',sum100
close(14)          ! close the file on which the results in written
end                ! indicator of program's end
```

You can recycle the logical unit if you first close it before opening it again. Doing so is not recommended. You are in danger to accidently overwrite your data if the close statement gets deleted.

Program structure (cont.)

Example 4:

```
program filename                          ! name of the program
! purpose: sum numbers from 1 to 100
! author: Donald Duck
! version: 11-11-2002
! data requirements: none
! output: sum of numbers from 1 to 100 on file OutPut.dat
! calls: none
! main program
open(14,file='OutPut.dat',form='formatted,status='unknown')
 …..

write(44,*)'sum100=',sum100          ! Logical unit is unassigned
close(14)      ! close the file on which the results in written
end            ! indicator of program's end
```

The data would be written to fort.44 and nothing would happen to OutPut.dat.

Program structure (cont.)

- Example:

  Design a program that reads in temperature in Fahrenheit and converts it to an absolute temperature

Program structure (cont.)

- Solution:

```
program F_to_K
! Purpose: convert temperature in F to K
! Version: 7-8-2006
! Author: Nicole Moelders

! Declare variables
implicit none
real  :: t_f                    ! Temperature in Fahrenheit
real  :: t_k                    ! Temperature in Kelvin

! Ask the user which temperature value is to be converted
write(6,*)"Which temperature is to be converted from F to K?"
read(5,*)t_f

t_k=(5./9.)*(t_f-32.)+273.15                    ! do the conversion

write(6,*)"T=",t_f," F = ",t_k," K"             ! deliver the result

end program
```

Useful operators

- FORTRAN has many build-in operators, functions and subroutines to use
- min and max: finds minimum/maximum of two values

Example:

```
v1=5
v2=6
max1=max(v1,v2)
min1=min(v1,v2)
```

Results: max1=6, min1=5

Note: min, max in FORTRAN90 search for the minimum and maximum of 2 values, i.e. the min and max commands of FORTRAN and IDL (that we will learn later) are different!

Useful operators (cont.)

- Examples of other built-in operators are

abs(x)	absolute value of X
acos(x)	arcus cosine of X
asin(x)	arcus sine of X
atan(x)	arcus tangent of X
cos(x)	cosine of X
dble(x)	converts X to double precision
exp(x)	returns e^x
int(x)	returns integer of X
log(x)	returns $\log_e(x)$
log10(x)	returns $\log_{10}(x)$
sign(A,B)	returns A with the sign of B
sin(x)	sine of X
sqrt(x)	takes square root of X
tan(x)	tangent of X
tanh(x)	returns hyperbolic tangent of X

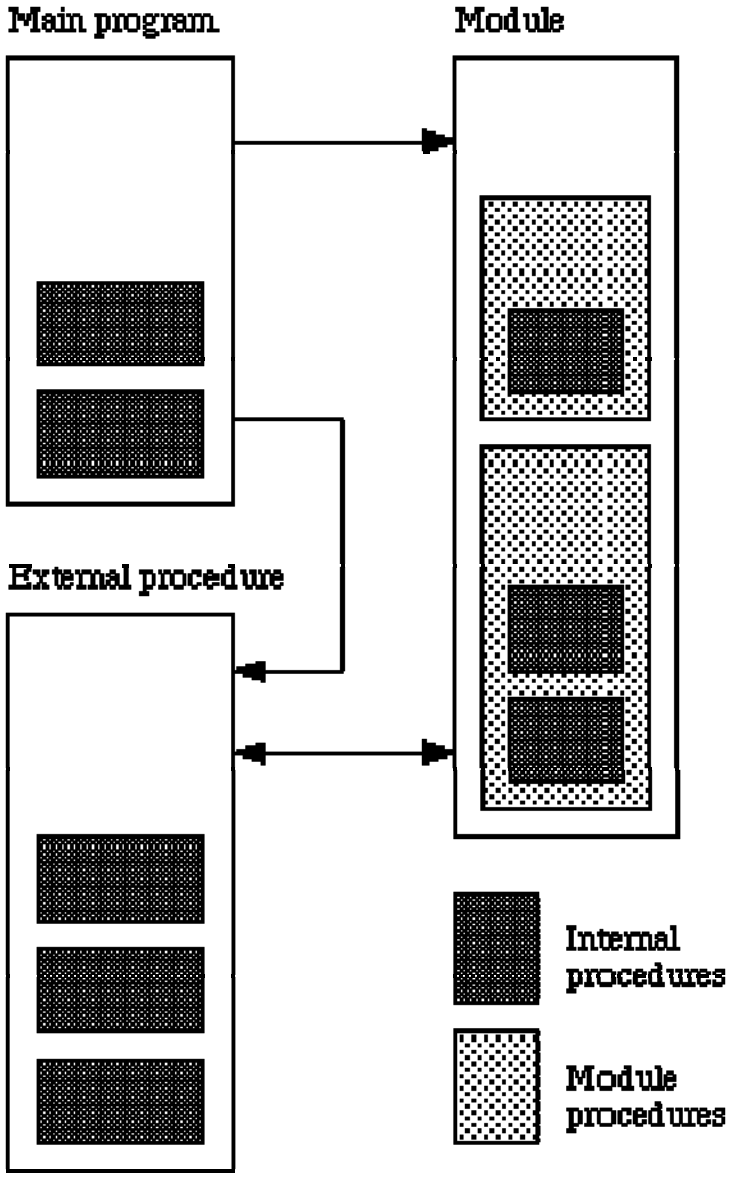
Useful operators (cont.)

- Important to know: You cannot give a variable a name that is reserved for an operator!

  i.e. a variable cannot have the name acos, but acos1 will work

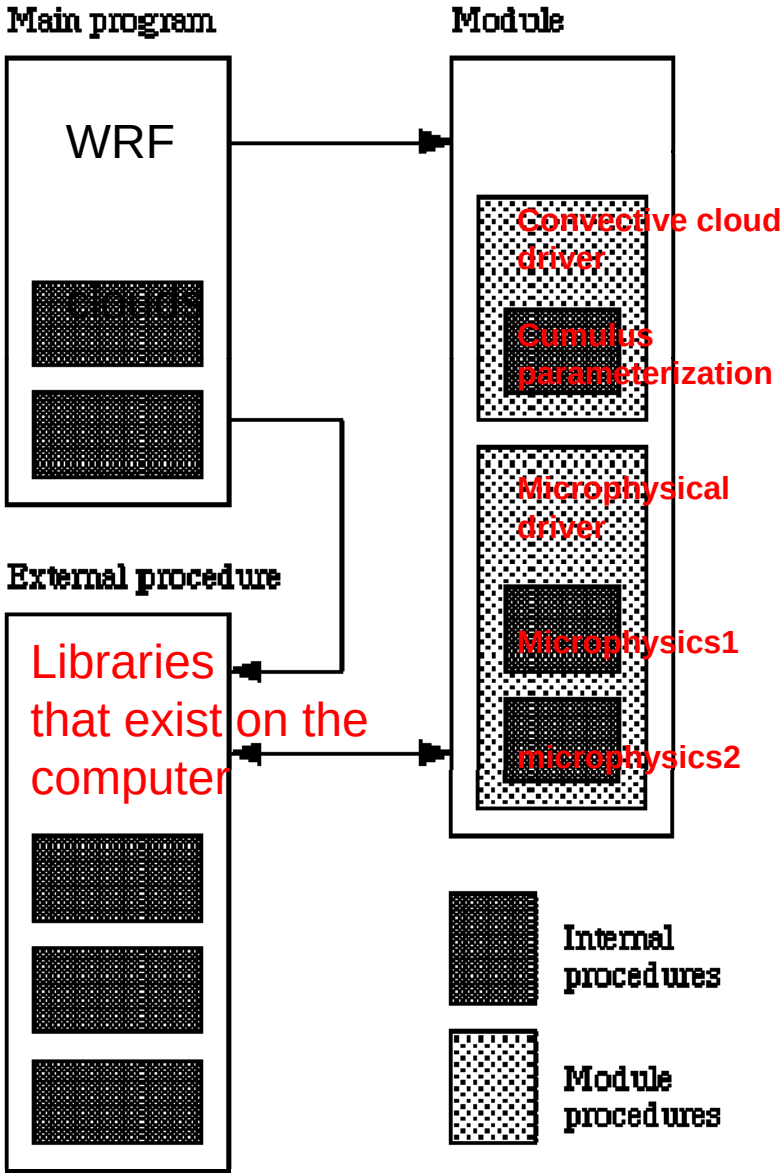
Program structure (cont.)

- A Fortran 90 program can consist of a number of distinct program units, namely procedures (internal, external and module), subroutines and modules
- An executable program consists of one main program and any number (including zero) of other program units
- Internal detail of each program unit is separate from other units
- The only link between units is the interface, i.e. where the program is called (example)
- Interfaces must be consistent



From http://www.qmw.ac.uk/~cgaa260/BUILDING/INTR_F90/OUTLINE.HTM

Program structure (cont.)



Internal procedures can be also functions, e.g. the Magnus formula

Program structure (continued)

Advantages of dividing a program into units:

- Units can be written and tested independently
- Program units of well defined task are easier to understand and maintain
- Modules and external procedures can be re-used in other programs (your personal library or exchange with colleagues)
- Most compilers better optimize modular code

Subroutines

- A **subroutine** is a FORTRAN procedure invoked by naming it in a **call** statement that has all input and return values

Example:

```
subroutine subroutine_name (agrument_list)  ! Argument list = input
                                            ! into subroutine and also
                                            ! holds the return values

…
(declaration section)
…
(execution section)
…
return
end subroutine subroutine_name
```

In declaration statement for dimensions input values are denoted by intent(in), return values by intent(out)

Subroutine (continued)

Interface is the call and the subroutine statement

Example 1

```
program main
…
real :: qv(10,123),t(10,153),qs(100,123)
…..
call cloud(t,qv,qs)                ! Here cloud is the subroutine, not a variable
…
end program main
subroutine cloud(t,qv,qs)     ! The variables must be in the same
                                   ! order as in call of the subroutine
real :: qv(10,123),t(10,153),qs(100,123) ! Must have same dimension, type
…
end subroutine cloud
```

Subroutine (continued)

Example 2

```
program main
…
real :: qv(10,123),t(10,153),qs(100,123)
…..
call cloud(qv,t,qs)
…
end program main
subroutine cloud(ttt,abc,xyz)    ! The variables must be in the same
                                 ! order as in call of the subroutine
real :: ttt(10,123),abc(10,153),xyz(100,123) ! Same dimension, type
…
end subroutine cloud
```

Subroutines (cont.)

- To avoid handing over variables that are calculated in the subroutine and are not defined yet when entering the subroutine and/or handing over variables that are not modified in the program when returning the **intent** of the variables has to be defined (data exchange is bottleneck in parallel computers!)

Example:

```
real, intent (in) :: noreturn    ! Variable enters from main
                                 ! program, but will not be returned
real, intent(out) :: leave       ! Variable will only leave subroutine
real, intent(inout) :: twoway ! Variable enters and leaves
real :: local                    ! Variable only used in subroutine
```

Note this works also with integer, real*8, double precision, complex, etc.

Subroutines (cont.)

Example:

```
subroutine hypotenuse (side1,side2,hypo)
! Purpose: calculate the hypotenuse
! Version: 8-31-2006
! Author: Nicole Moelders

implicit none
real, intent (in) :: side1, side2      ! Length of sides
real, intent(out) :: hypo              ! Length of hypotenuse
real :: hlp                            ! Auxiliary variable

hlp=side1**2+side2**2
hypo=hlp**0.5                          ! Calculate hypotenuse

return
end subroutine hypotenuse
```

Note that hypo=sqrt(hlp) is quicker than hypo=hlp**0.5

Subroutines (cont.)

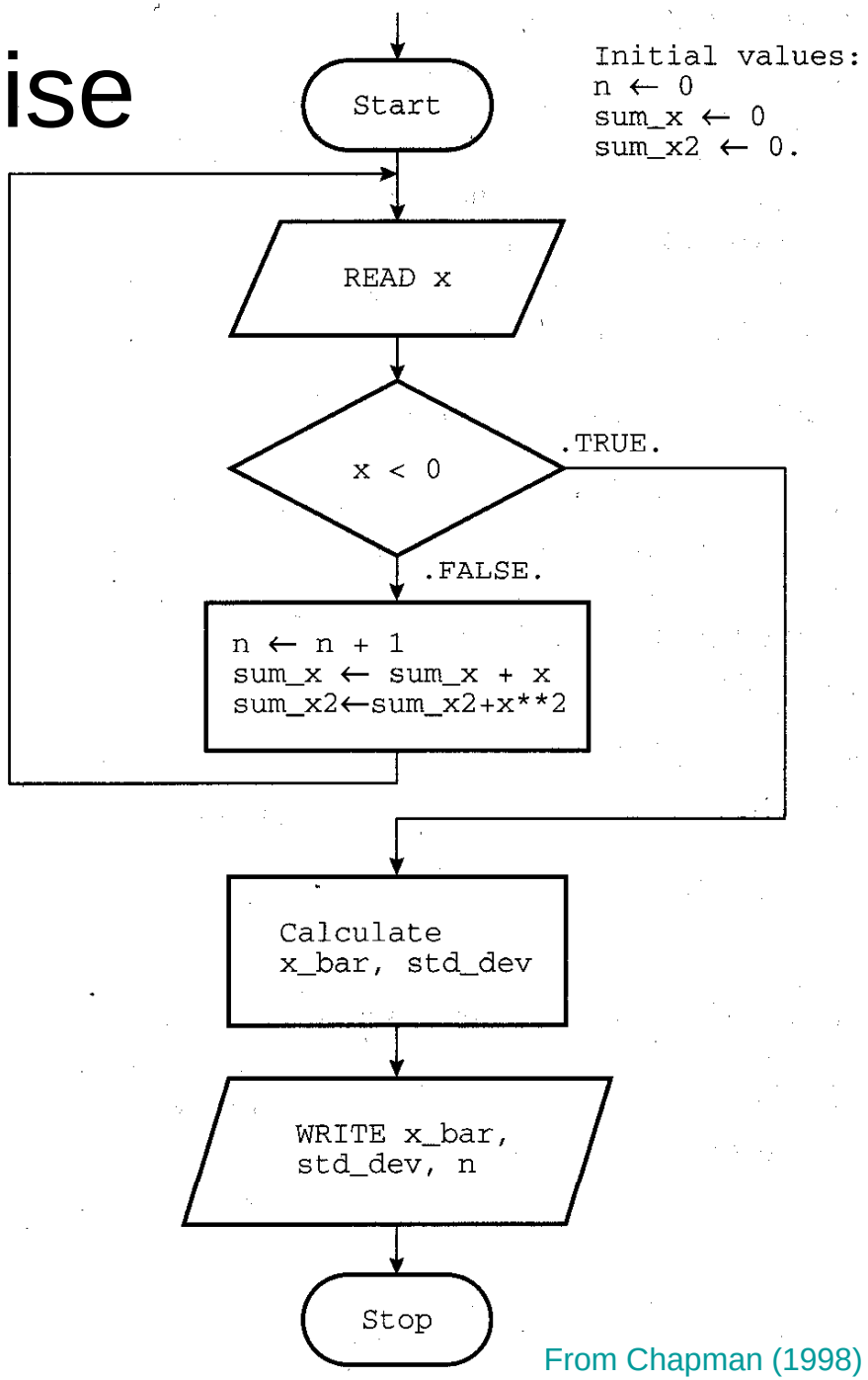
- Driver for a subroutine can be another subroutine or the main program
- Values in the call **must** be the same order than in the subroutine, but may have different names

Example:

```
program
! Purpose: test the subroutine hypotenuse
! Version: 8-31-2006
! Author: Nicole Moelders

implicit none
real :: a, b! Length of sides
real :: c                  ! Length of hypotenuse

! Get the lengths of the sides
write(6,*)”type in lengths of side 1 and 2. Thanks”
read(5,*)a, b

call hypotenuse(a, b, c)                    ! Call subroutine

write(6,*)”the length of the hypotenuse is “,c
end program
```

Exercise

What does the program designed by the flow diagram on the right do?



From Chapman (1998)

Exercise (cont.)

Turn the algorithm into fortran statements

```
program meanstddev
! Purpose: ……
! Author: Chapman
! History: modified by Nicole
implicit none
integer :: n=0.
real :: std_dev=0.
real :: sum_x=0.
real :: sum_x2=0.
real :: x_bar=0.

do
 write(6,*)’type in number’
 read(5,*)x
 if(x <0)exit
 n=n+1
 sum_x=sum_x+x
 sumx2=sumx2+x*x
 enddo
```

Exercise (cont.)

```
x_bar=sum_x/real(n)
std_dev=sqrt((real(n)*sum_x2                    &
          -sum_x*sum_x)/(real(n)*real(n-1)))
write(6,*)"mean=",x_bar
write(6,*)"standard deviation=",std_dev
write(6,*)"sample size=",n

end program meanstddev
```

To test the program calculate answers by hand and compare with results of the computer

Compilation

- A Fortran 90 program is usually has the suffix .f90
- To run the source code it has to be compiled
  - The programmer initiate compilation by typing and operating system command or submitting a shell script
  - Compilation translates the high-level statements (source code) into intermediate assembly code and finally to machine (object) code (usually filename.o on UNIX systems)
  - The compiler checks the syntax
  - The compiler may also initiate any linking of external files, e.g. include any code from a library or other pre-compiled file
  - If no errors are found and all files are present the compile generates the executable code (usually a.out on a UNIX system is the default name)

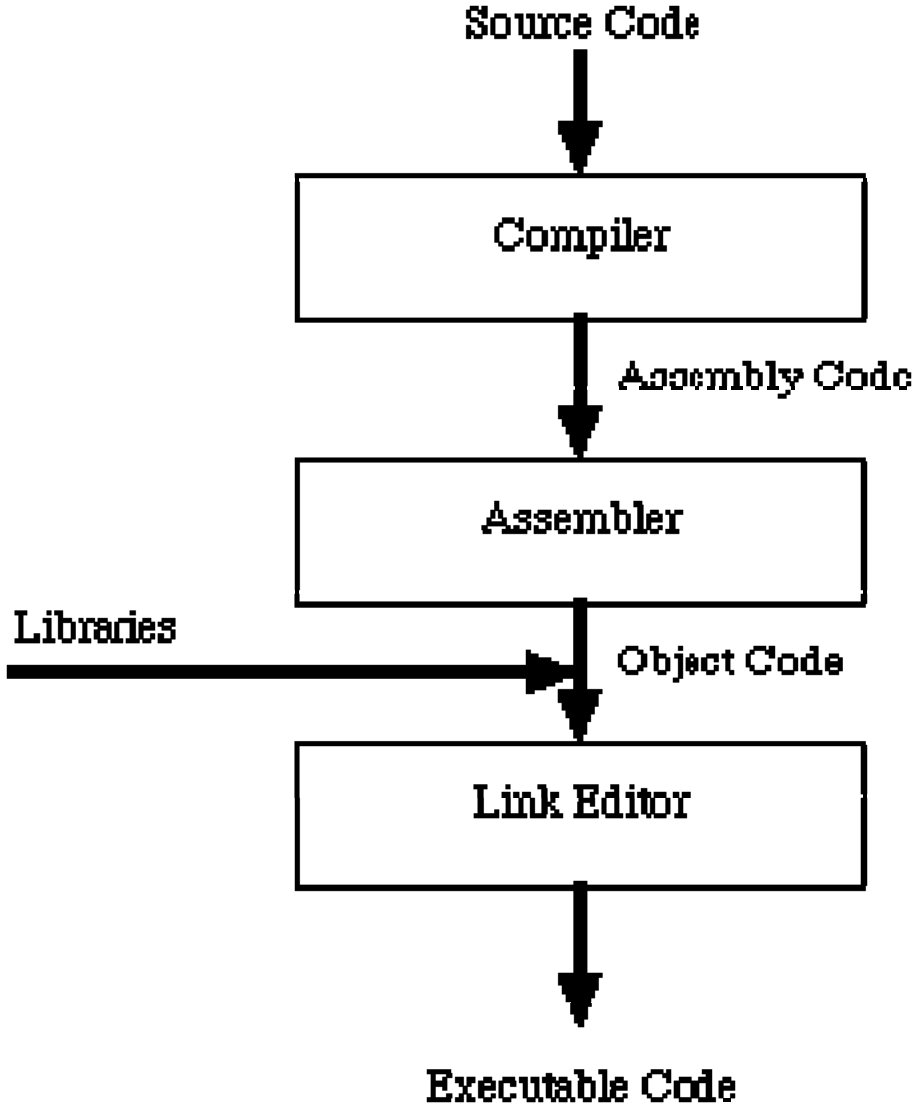
Compilation (cont.)

- There are various compilers that work slightly different (i.e. allow for more or less sloppy programming)
- If you are on a “real computer” (real computers run under unix) the standard command to compile is

  pgf90 program_name.f90 or

  f90 program_name.f90 or

  qsub script_name
- The latter is for large programs like WRF. Here script_name is one of the scripts provided at the ARSC web page



From http://www.qmw.ac.uk/~cgaa260/BUILDING/INTR_F90/OUTLINE.HTM

Execution

- To run the program the user types either the name of the executable file, e.g. **./a.out** or submits a script by **qsub script_name**
- Typically compilation and execution are all in one script
- An execution error (the most common execution error is division by zero) can lead to a crash of the program
- Errors of logic (multiply rather than add) can not be checked by the compiler, i.e. the programmer has to identify and eliminate these him/herself
- Test against data with known results
- Take great care during the initial design because identifying errors at the design phase is cheaper and easier than a day before your defense

Debugging techniques

Typically a “freshly coded” program still has some errors

There are three sorts of errors

- Real FORTRAN errors
- It has to be debugged to remove them and to build an executable code

- The program has no FORTRAN errors, but crashes
- There are some debugging programs out there that work in conjunction with the core file (however, most institutes do not have that software)
- Insert write statements to figure out where it crashes (division by zero or taking the logarithm of a negative number are common reasons)

- The program runs and provides results that are notably wrong
- Insert write statements to figure out where it goes wrong

Debugging

Example 1:

Debug this program

```
  PROGRAM myfirst
C
C This program serves as an example for debugging
C Author: Nicole Molders
C Version: 9/13/2006
C
      IMPLICIT NONE
      REAL HM
      INTEGER OEHM,I,J,MIX,MJX,MKX
      REAL DAMAGE(3,2),TESTDAT(2,6),OUTPU,GRAV
      PARAMETER (MIX=40,MJX=56,MKX=MIX+MJX)
      COMMON /PETER/ OEHM,TESTDAT
      DATA GRAV /9.81/
C
      GRAV=GRAV+6.
      WRITE(6,*)'GRAV=',GRAV
      DAMAGE(1,1)=1.
      DAMAGE(1,2)=8.
      DAMAGE(2,1)=4.
      DAMAGE(2,2)=6.
      DAMAGE(3,1)=66.
      DAMAGE(3,2)=89.
      HM=261.
      OEHM=HM
C...we create a loop
      DO II=1,12
      IF(I < 6)THEN
      OEHM=OEHM+1
      HM=HM+1.
      ELSE
      OEHM=OEHM-1
      HM=HM-1
      ENDIF
      WRITE(6,1000)HM,OEHM,I
      ENDDO
      DO J=1,2
      DO I=1,3
      DAMAGE(I,J)=DAMAGE(I,J)+2.
      WRITE(6,*)'i=',I,' j=',J,' DAMAGE(i,j)=',DAMAGE(I,J)
      ENDDO
      ENDDO
C read something
      WRITE(6,*)'Give me a number'
      READ(5,*)J
      WRITE(6,*)'J=',J
      OPEN(14,FILE='../test/test.dat',STATUS='OLD')
      READ(14,*)TESTDAT
      WRITE(6,*)'TESTDAT=',TESTDAT
C
C... Call a subroutine
      DO I=1,7
      WRITE(6,*)'OUTPU=',OUTPU
      WRITE(6,*)'before TESTDAT and OEHM are',TESTDAT,OEHM
      CALL SAL(HM,DAMAGE,OUTPU)
      WRITE(6,*)'OUTPU=',OUTPU
      WRITE(6,*)'MIX,MJX,MKX,GRAV=',MIX,MJX,MKX,GRAV
      ENDDO
C
1000  FORMAT(1X,'HM=',F6.0,1X,' OEHM=',I3,1X,' I=',I2)
C
      STOP'all done'
      END
C**********************************************************************
      SUBROUTINE SAL(HM1,DAMAG5,OUTUP)
C**********************************************************************
      IMPLICIT NONE
      REAL HM1
      REAL DAMAG5(3,2),OUTUP
      INTEGER I,OEHM
      REAL TESTDAT(2,6)
      COMMON /PETER/ OEHM,TESTDAT
      SAVE
      REAL counter
C
      OUTUP=0.
      write(6,*)'counter=',counter
      counter=0.
      write(6,*)'counter=',counter
      DO I=1,3
      DO J=1,2
      DAMAG5(I,J)=DAMAG5(I,J)+HM1
      OUTUP=OUTUP+DAMAG5(I,J)
      counter=counter+1.
      ENDDO
      ENDDO
C
      WRITE(6,*)'in TESTDAT and OEHM are',TESTDAT,OEHM
      WRITE(6,*)'counter=',counter
      RETURN
      END
```

Debugging (cont.)

f90 myfirst.f90

What you get is:

```
C
^
"myfirst.f90", Line = 2, Column = 1: ERROR: IMPLICIT NONE is specified in the local scope, therefore an explicit type must be specified for data object "C".
      ^
"myfirst.f90", Line = 2, Column = 7: ERROR: Unknown statement.  Expected assignment statement but found "EOS" instead of "=" or "=>".

C This program serves as an example for debugging
 ^
"myfirst.f90", Line = 3, Column = 3: ERROR: Unknown statement.  Expected assignment statement but found "T" instead of "=" or "=>".

C Author: Nicole Molders
  ^
"myfirst.f90", Line = 4, Column = 3: ERROR: Unknown statement.  Expected assignment statement but found "A" instead of "=" or "=>".

C Version: 9/13/2006
  ^
"myfirst.f90", Line = 5, Column = 3: ERROR: Unknown statement.  Expected assignment statement but found "V" instead of "=" or "=>".

C
 ^
"myfirst.f90", Line = 6, Column = 2: ERROR: Unknown statement.  Expected assignment statement but found "EOS" instead of "=" or "=>".

C
 ^
"myfirst.f90", Line = 14, Column = 2: ERROR: Unknown statement.  Expected assignment statement but found "EOS" instead of "=" or "=>".

C...we create a loop
 ^
"myfirst.f90", Line = 25, Column = 2: ERROR: Unknown statement.  Expected assignment statement but found "." instead of "=" or "=>".

      DO II=1,12
        ^
"myfirst.f90", Line = 26, Column = 10: ERROR: IMPLICIT NONE is specified in the local scope, therefore an explicit type must be specified for data object "II".

C read something
  ^
"myfirst.f90", Line = 42, Column = 3: ERROR: Unknown statement.  Expected assignment statement but found "R" instead of "=" or "=>".

C
 ^
"myfirst.f90", Line = 49, Column = 2: ERROR: Unknown statement.  Expected assignment statement but found "EOS" instead of "=" or "=>".
```

Debugging (cont.)

f90 myfirst.f90

Error listing continued:

```
C... Call a subroutine
 ^
"myfirst.f90", Line = 50, Column = 2: ERROR: Unknown statement.  Expected assignment statement but found "." instead of "=" or "=>".

C
 ^
"myfirst.f90", Line = 58, Column = 2: ERROR: Unknown statement.  Expected assignment statement but found "EOS" instead of "=" or "=>".

C
 ^
"myfirst.f90", Line = 60, Column = 2: ERROR: Unknown statement.  Expected assignment statement but found "EOS" instead of "=" or "=>".

C**************************************************************************
^
"myfirst.f90", Line = 63, Column = 1: ERROR: This compilation unit contains multiple main program units.
^
"myfirst.f90", Line = 63, Column = 1: ERROR: This unnamed main program unit is missing an END statement.
 ^
"myfirst.f90", Line = 63, Column = 2: ERROR: Unknown statement.  Expected assignment statement but found "*" instead of "=" or "=>".

C**************************************************************************
 ^
"myfirst.f90", Line = 65, Column = 2: ERROR: Unknown statement.  Expected assignment statement but found "*" instead of "=" or "=>".

C
 ^
"myfirst.f90", Line = 74, Column = 2: ERROR: Unknown statement.  Expected assignment statement but found "EOS" instead of "=" or "=>".

C
 ^
"myfirst.f90", Line = 86, Column = 2: ERROR: Unknown statement.  Expected assignment statement but found "EOS" instead of "=" or "=>".

f90: COMPILE TIME 0.040000 SECONDS
f90: MAXIMUM FIELD LENGTH 4766732 DECIMAL WORDS
f90: 90 SOURCE LINES
f90: 20 ERRORS, 0 WARNINGS, 0 OTHER MESSAGES, 0 ANSI
```

Debugging (cont.)

```
 PROGRAM myfirst
!
! This program serves as an example for debugging
! Author: Nicole Molders
! Version: 9/13/2006

      IMPLICIT NONE
      REAL :: HM
      INTEGER :: OEHM,I,J,MIX,MJX,MKX
      REAL :: DAMAGE(3,2),TESTDAT(2,6),OUTPU,GRAV
      PARAMETER (MIX=40,MJX=56,MKX=MIX+MJX)
      COMMON /PETER/ OEHM,TESTDAT
      DATA GRAV /9.81/
!
      GRAV=GRAV+6.
      WRITE(6,*)'GRAV=',GRAV
      DAMAGE(1,1)=1.
      DAMAGE(1,2)=8.
      DAMAGE(2,1)=4.
      DAMAGE(2,2)=6.
      DAMAGE(3,1)=66.
      DAMAGE(3,2)=89.
      HM=261.
      OEHM=HM
!...we create a loop
      DO I=1,12
      IF(I < 6)THEN
      OEHM=OEHM+1
      HM=HM+1.
      ELSE
      OEHM=OEHM-1
      HM=HM-1
      ENDIF
      WRITE(6,1000)HM,OEHM,I
      ENDDO
DO J=1,2
      DO I=1,3
      DAMAGE(I,J)=DAMAGE(I,J)+2.
      WRITE(6,*)'i=',I,' j=',J,' DAMAGE(i,j)=',DAMAGE(I,J)
      ENDDO
      ENDDO
! read something
      WRITE(6,*)'Give me a number'
      READ(5,*)J
      WRITE(6,*)'J=',J
      OPEN(14,FILE='../test/test.dat',STATUS='OLD')
      READ(14,*)TESTDAT
      WRITE(6,*)'TESTDAT=',TESTDAT
```

- Main error: C is not a comment in F90
- After fixing all errors you should get

```
!... Call a subroutine
      DO I=1,7
      WRITE(6,*)'OUTPU=',OUTPU
      WRITE(6,*)'before TESTDAT and OEHM are',TESTDAT,OEHM
      CALL SAL(HM,DAMAGE,OUTPU)
      WRITE(6,*)'OUTPU=',OUTPU
      WRITE(6,*)'MIX,MJX,MKX,GRAV=',MIX,MJX,MKX,GRAV
      ENDDO
!
1000  FORMAT(1X,'HM=',F6.0,1X,' OEHM=',I3,1X,' I=',I2)
!
      STOP'all done'
      END
!***********************************************************************
      SUBROUTINE SAL(HM1,DAMAG5,OUTUP)
!***********************************************************************
      IMPLICIT NONE
      REAL, intent(in) :: HM1
      REAL, intent(inout) :: DAMAG5(3,2),OUTUP
      INTEGER :: I,J,OEHM
      REAL :: TESTDAT(2,6)
      COMMON /PETER/ OEHM,TESTDAT
      SAVE
      REAL :: counter

      OUTUP=0.
      write(6,*)'counter=',counter
      counter=0.
      write(6,*)'counter=',counter
      DO I=1,3
      DO J=1,2
      DAMAG5(I,J)=DAMAG5(I,J)+HM1
      OUTUP=OUTUP+DAMAG5(I,J)
      counter=counter+1.
      ENDDO
      ENDDO

      WRITE(6,*)'in TESTDAT and OEHM are',TESTDAT,OEHM
      WRITE(6,*)'counter=',counter
      RETURN
      END subroutine SAL
```

Debugging (cont.)

Example 2: Debug this program

```
program mysecond
! Purpose: calculate the day of year
! version: 9-7-2006
! author: Nicole Moelders
! data requirements: none
! output: day of year

implicit none

! declare variables
 integer :: day, doy    ! day (dd), day of year
 integer :: i,j         ! index variable for loop
 integer :: leap_day    ! extra day for Feburary 29th
 integer :: month       ! month (mm)
 integer :: year        ! year(yyyy)

! get date to convert
 write(6,*)'type in date as day, month and year'
read(5,*)day,month,year
 write(6,*)day
 write(6,*)month
 write(6,*)year

! check for leap year
 if(mod(year,400) == 0)then begin
  leap_day=1     ! years divided by 400 are leap years
  write(6,*)year," is a leap year"
 elseif (mod(yeas,100) == 0) then
  leap_day=0   ! other centuries are not leap years
 else if (mod(year,4) ==0)then begin
  leap_day=1 ! otherwise every 4th year is a leap year
  write(6,*)year," is a leap year"
 else
  leap_day=0    ; no leap year
  write(6,*)year," is no leap year"
 endif
```

```
! calculate the day of year
 doy=day
 do i=1,month-1
 if(i == 1)write(6,*)"working on ",i," st month"
 if(i == 2)write(6,*)"working on ",i," nd month"
 if(i >= 3)write(6,*)"working on ",i," th month"
!add days in months from January to last month
   select case(i)
   case(1, 3, 5, 7, 8, 10, 12)
      doy=doy+31
   case(4, 6, 9, 11, 5)
      doy=doy+30
   case(2)
       doy=doy+28+leap_day
   end select
 end do

! output
 write(6,*)day,month,year,' corresponds to the ",doy," day of the year'

 end program
```

Debugging (cont.)

```
 if(mod(year,400) == 0)then begin
                   ^
"mysecond.f90", Line = 25, Column = 24: ERROR: IMPLICIT NONE is specified in the local scope, therefore an explicit type must be specified for data object "THEN".
                    ^
"mysecond.f90", Line = 25, Column = 29: ERROR: Unknown statement.  Expected assignment statement but found "B" instead of "=" or "=>".

 elseif (mod(yeas,100) == 0) then
 ^
"mysecond.f90", Line = 28, Column = 2: ERROR: This ELSE IF statement has no matching IF statement.
           ^
"mysecond.f90", Line = 28, Column = 14: ERROR: IMPLICIT NONE is specified in the local scope, therefore an explicit type must be specified for data object "YEAS".

 else if (mod(year,4) ==0)then begin
 ^
"mysecond.f90", Line = 30, Column = 2: ERROR: This ELSE IF statement has no matching IF statement.

 else
 ^
"mysecond.f90", Line = 33, Column = 2: ERROR: This ELSE statement has no matching IF statement.

  leap_day=0    ; no leap year
               ^
"mysecond.f90", Line = 34, Column = 19: ERROR: IMPLICIT NONE is specified in the local scope, therefore an explicit type must be specified for data object "NO".
                ^
"mysecond.f90", Line = 34, Column = 22: ERROR: Unknown statement.  Expected assignment statement but found "L" instead of "=" or "=>".

 endif
 ^
"mysecond.f90", Line = 36, Column = 2: ERROR: This END IF statement has no matching IF statement.

   case(4, 6, 9, 11, 5)
                 ^
"mysecond.f90", Line = 48, Column = 22: ERROR: The case value has the same value as a case value on line 46.

f90: COMPILE TIME 0.020000 SECONDS
f90: MAXIMUM FIELD LENGTH 4760588 DECIMAL WORDS
f90: 58 SOURCE LINES
f90: 10 ERRORS, 0 WARNINGS, 0 OTHER MESSAGES, 0 ANSI
```

Debugging (cont.)

The debugged program should read:

```
program mysecond
! Purpose: calculate the day of year
! version: 9-7-2006
! author: Nicole Moelders
! data requirements: none
! output: day of year

implicit none

! declare variables
 integer :: day, doy    ! day (dd), day of year
 integer :: i,j         ! index variable for loop
 integer :: leap_day    ! extra day for Feburary 29th
 integer :: month       ! month (mm)
 integer :: year        ! year(yyyy)

! get date to convert
 write(6,*)'type in date as day, month and year'
read(5,*)day,month,year
 write(6,*)day
 write(6,*)month
 write(6,*)year

! check for leap year
 if(mod(year,400) == 0)then
  leap_day=1     ! years divided by 400 are leap years
  write(6,*)year," is a leap year"
 elseif (mod(year,100) == 0) then
  leap_day=0   ! other centuries are not leap years
 else if (mod(year,4) ==0)then
  leap_day=1 ! otherwise every 4th year is a leap year
  write(6,*)year," is a leap year"
 else
  leap_day=0    ! no leap year
  write(6,*)year," is no leap year"
 endif
```

```
! calculate the day of year
 doy=day
 do i=1,month-1
 if(i == 1)write(6,*)"working on ",i," st month"
 if(i == 2)write(6,*)"working on ",i," nd month"
 if(i >= 3)write(6,*)"working on ",i," th month"
!add days in months from January to last month
   select case(i)
   case(1, 3, 5, 7, 8, 10, 12)
      doy=doy+31
   case(4, 6, 9, 11)
      doy=doy+30
   case(2)
       doy=doy+28+leap_day
   end select
 end do

! output
 write(6,*)day,month,year," corresponds to the ",doy," day of the year"

 end program
```