Theory for Lab-work 2 onwards

A brief history of Fortran

The first version of this programming language was created by a team from IBM under the leadership of John W. Backus, being released in 1957 under the name "IBM Mathematical Formula Translating System" (in short: FORTRAN, from the combination of FORmula TRANslation parts), this being the first high-level programming language (close to natural language). In 1958 IBM published a revised version, called FORTRAN II, which provided support for procedural programming by introducing specifications for subroutines and functions. Because of its popularity, IBM decided to remove the features that limited the use of the language on IBM systems, and in 1964 released a variant called FORTRAN IV that could run on any computer. The FORTRAN 66 version appeared in 1966, as a result of the standardization carried out by the American Standards Association (ASA, the precursor of ANSI), being the first programming language defined by a standard. The “ANSI FORTRAN” committee (known as "X3J3") began developing a new version in 1969, and as a result, FORTRAN 77 appeared, the most widely used version of the language.
The next version was aspected to be released in the 1980s (Fortran 8X), but it was released only in 1991 introducing the free format and became known as Fortran 90, opening a path for HPF (High Performance Fortran). In 1997, the standard for Fortran 95, the first object-oriented version, was published.
This document mainly refers to this version (Fortran 95) of the language, presenting only notions for beginners and aiming to provide the necessary information for the practical work of Civil Engineering students during "Computer programming and programming languages".
Compared to C++ (an object-oriented language that supports polymorphism and inheritance), Fortran has introduced some similar features (through modules and derived types), but has no automatic inheritance. On the other hand, Fortran is easier to learn and use for scientific computing than C++, having native support for complex values, multidimensional arrays, etc., which C++ lacks. Fortran 2003 represents a significant turn in object-oriented features, also ensuring interoperability with C/C++, and in 2010 Fortran 2008 was released with new provisions (sub-modules, co-arrays, the contiguous attribute, etc.) and having implemented parallel processing with distributed memory. After Fortran 2018, which was a revision of the previous version with additional support for parallel processing, Fortran 2023 is the latest standardized version with even more features.
Here is a quoted fragment (from the web-page accesible at https://fortran-lang.org/) for those interested in the use of this language: ”Fortran is mostly used in domains that adopted computation early–science and engineering. These include numerical weather and ocean prediction, computational fluid dynamics, applied math, statistics, and finance. Fortran is the dominant language of High Performance Computing and is used to benchmark the fastest supercomputers in the world.”
A significant part of the following is based on the adapted contents of the Romanian book "*Inițiere în programare și în limbajul Fortran*" (F.-Zs. Gobesz, C. Bacoțiu, UTPres Publisher, Cluj-Napoca, 2003, ISBN 973-662-005-0).

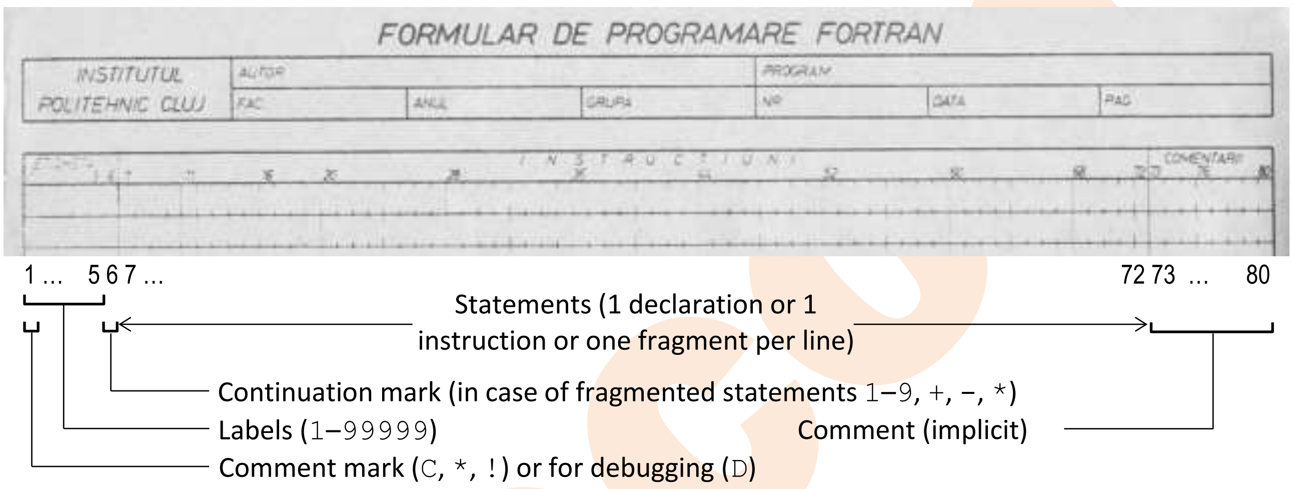
Source file structure

A source file can contain one or more program units (these will be presented later), or fragments of them (in the form of sections). The source file can be created with any text editor, provided that it results in a character content (ASCII file).
The character set usable in the Fortran language contains the alphanumeric characters (the 26 upper and lower case letters of the English alphabet: `a–z`, `A–Z`, and the digits: `0–9`), plus 4 symbols for arithmetic operations (`+`, `–`, `*`, `/`) and a set of special characters (blank or space, horizontal tab, comma, period, apostrophe, open and closed parentheses, equal sign, dollar sign, ampersand). The Fortran 90 language extended this list with the following allowed special characters: `_`, `!`, `:`, `;`, `"`, `%`, `<`, `>`, `?`, `^` and `#`. The comma has the role of separating elements within a list, while the dot is the decimal separator.

Symbolic names are used to name variables, different program parts, and to identify functions. If the conventions of older versions of the language allowed the use of only 8 characters (consisting of alphanumeric characters and the special character $), Fortran 95 allows the use of 31 characters (consisting of alphanumeric characters, the special character $ and the special character _). The first character must always be a letter. Program unit and section names are considered global and must be unique throughout the source, and entity names must be unique within the same program unit. The Fortran language is not case sensitive for symbolic names.

The editing mode of the source file can be in fixed form (Fortran 77), tabular form or free form (the latter being introduced by Fortran 90 and allowed by following versions of the language).

The **fixed form** respects the editing structure based on punched cards and old template sheets (like the one in the image), considering 80 characters as the maximum length of a line (record), having the following structure (below the image of the template sheet, the relevant column numbers and the content allowed for each field are marked):


FORMULAR DE PROGRAMARE FORTRAN
INSTITUTUL POLITEHNIC CLUJ
AUTOR
PROGRAM
FAC
ANUL
GRUPA
NR
DATA
PAG
INSTRUCȚIUNI
COMENTARII
1 ... 5 6 7 ...
72 73 ... 80
Statements (1 declaration or 1 instruction or one fragment per line)
Continuation mark (in case of fragmented statements 1–9, +, –, *)
Labels (1–99999)
Comment (implicit)
Comment mark (C, *, !) or for debugging (D)

The labels are integers of at most 5 digits, with a reference role within the program section, marking the instructions before which they appear (in the respective line). Their use is optional and subject to some restrictions (only lines with executable statements can be labeled and labels cannot exceed the range of columns 1–5). For a label to be valid, its value must be in the range 1–99999. If a line should be marked as a comment, the letter C or the character * (respectively ! starting with Fortran 90) should be written in the first column. In this case the structure and content of the line will be ignored during compilation. Some compilers also allow the use of the character D to mark the current line in the first column as a comment, thus allowing the optional compilation (interpretation) of these lines in case of debugging the source.
In Fortran 77 only one statement was written on a line, but since Fortran 90 it is allowed to write more than one statement on a line, in this case the character ; is used as separator between them.
If the statement is longer than the space between columns 7 and 72, it can be continued on the following lines by marking in column 6 (with a number or one of the symbols: +, -, *) that the fragments are the continuation of the previous ones. From Fortran 90 onwards, any character other than the digit 0 can be used for this continuation mark. The number of continuation lines allowed also depends on the compiler chosen. Fortran 77 allowed 99 fragments (1 initial line and 98 continuation lines), but the Fortran 90 standard allows only 19 fragments in fixed form and 39 fragments in free form. Fortran 95 allows up to 90 continuation lines in fixed form and only 31 continuation lines in free form.
Some compilers allow the line interpretation range to be extended up to column 80 (even 132, starting with Fortran 90), but as standard any content in the range of columns 72-80 is considered comment by default and as such is ignored by the compiler.

The **free form** does not have the restrictions illustrated above, the statements are not limited to any particular fitting on the line columns, each line can contain up to 132 characters. Instead, spaces are significant, and in some cases act as separators (for names, constants, keywords, or as spacers between labels and instructions). This form has only been introduced since Fortran 90 (but Fortran 90 also supports fixed and tabular formats). In the free form, the comment is indicated by the character ! (starting from any column) or by the letter `C` written in the first column (beware of names starting with this letter, do not write them from the first column), while the `&` character marks the break of a statement (at the end) which will be continued on the next line. It is allowed to write more than one statement on a line if they are separated by the `;` character (which is ignored at the end of a line, of course).

Tabular form is actually a variant of both fixed and free form, and is so called because of the use of the horizontal tab character at the beginning of lines. If this <*Tab*> character is the first on a line, then the line contains a statement (declaration or instruction, or maybe a marker). If this first character is followed by a non-zero digit, the digit marks a continuation fragment of the previous line and must be followed by a space to separate it from the continuation content. The <Tab> character may be preceded only by a comment mark or a label. Line lengths must not exceed column 72 for fixed form and column 132 for free form.

Regardless of the horizontal structure (fixed, free or tabular form), the vertical structure of a source file must respect the following sequence of specifications: declarations (concerning the program unit, the entities used), body (containing the statements to executed at runtime) and final marker. If the source file contains only a segment of a program unit, any of the three parts (declarations, body, final marker) may be missing, but the order must be respected. In such cases, the contents of such a source file shall be included (using the `INCLUDE` specification) in another source file before compilation.

Notice: In the following chapters, syntax (writing rules) and examples are given where other characters are used. The square brackets are not part of the syntax, but mark the optionality of the included content, and the consecutive dots (...) mark repeatable elements. Italic sequences mark elements that replace content in the positions in which they appear.

Entity types

In Fortran, every entity has a type, either implicit, or explicitly declared. There are intrinsic types and derived types (defined by the programmer using intrinsic types or previously defined derived types). Intrinsic types are `INTEGER` (integer numbers), `REAL` (real numbers, with a decimal part), `COMPLEX` (complex numbers, viewed as pairs of numbers with a decimal part), `LOGICAL` (logical values, there are only two, the constants `.TRUE.` and `.FALSE.` constants), `CHARACTER` (character or string), and `BYTE` (8-bit value, used in older versions of the language).

Explicit type declaration of entities can be done according to syntax:

type[`(`*kind*`)`][[`,` *attribute*]... `::`] *entity_list*

The keywords for *type* are `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL` and `CHARACTER` (in some versions of Fortran there is also `BYTE`), or `TYPE(`*name*`)`, where *name* refers to a type previously defined by the programmer. The *kind* specifies the number of bytes used for storage (optionally preceded by the keyword `KIND=`, or `LEN=` in the case of `CHARACTER` type). This value depends on the *type* of entities, but there are also compiler dependent default values (usually 4 bytes for `REAL` type entities and 2 or 4 bytes for `INTEGER` type entities). Explicit values can be: 1, 2 or 4, eventually 8 for the `INTEGER` and `LOGICAL` types; 4 or 8, (eventually 16) for the `REAL` and `COMPLEX` types. Single characters and `BYTE` type entities are stored on 1 byte, so their storage length cannot be changed explicitly (if the *kind* is specified for `CHARACTER` type, it defaults to the number of characters in the string). `INTEGER(1)` and `LOGICAL(1)` type entities will also be stored on 1 byte.

The following can be specified as an *attribute*:

- `ALLOCATABLE` for arrays with dynamically allocated memory or `DIMENSION(`*limits*`)` for arrays with statically allocated memory (will be presented later),
- `EXTERNAL` for entities redefined by the programmer or `INTRINSIC` for entities predefined in Fortran,
- `INTENT(`*direction*`)` for input/output purpose (where *direction* can be `IN` for input, `OUT` for output, default `INOUT`),

- `PARAMETER` for constant values,
- `PUBLIC` for visible entities, `PRIVATE` for local entities (only accessible in the current program unit),
- `POINTER` for indicators or `TARGET` for targets,
- `OPTIONAL` for temporary entities, `SAVE` for stored entities.

If no *attribute* is specified, the `::` separator can be omitted (it only serves to delimit the list of keywords on the left, from the *entity_list* on the right of the specification).

For numeric entities there is an implicit rule regarding their type, which (of course) can be changed or canceled with the following syntax of the `IMPLICIT` statement:

`IMPLICIT` *type* `(`*c*[`,`*c*]…`)` [`,` *type* `(`*c*[`,`*c*]…`)`]…

where *type* must be an intrinsic type specifier (or previously defined derived type) and *c* stands for a letter or range of letters in alphabetical order. To cancel any implicit rule, write:

`IMPLICIT NONE`

When canceling the implicit rule, the types of all entities must be explicitly declared. According to the predefined implicit rule in Fortran, entities whose name starts with one of the letters I, J, K, L, M, or N will be of type `INTEGER`, and the rest will be of type `REAL`. Consequently, unless this rule is changed or canceled, type declarations can be omitted while respecting the rule.

The definition of a derived type is done according to the syntax:

`TYPE` *name*
specifications
`END TYPE` [*name*]

Once defined, such derived types can be used to specify the type of entities by replacing the *type* keyword with `TYPE(`*name*`)` in the explicit type declaration. Reference to a component in such a derived type can be made using the % selector, in the form *parent*`%`*component*[`%`*subcomponent*...], as will be illustrated in an example below.

When the entity type is explicitly declared, initial values can also be attributed. The attribution can be done within the *entity_list* or separately, through the `DATA` statement. The syntax of this statement is as follows:

`DATA` *variable_list*`/`*value_list*`/`[[`,`]*variable_list*`/`*value_list*`/` …]

where for each entity in the *variable_list* there must correspond a value from the *value_list* (the list delimited with `/` characters), in order of succession from left to right.

Examples:	Explanations:

`REAL(KIND=8) Di,e33`
`! Equivalent to:`
`REAL(8) dI,E33`
	The entities (variables) named DI and E33 are of type `REAL` and are stored on 8 bytes each (in older versions of Fortran, the type `DOUBLE PRECISION` was used in such a case). As you can see, it doesn't matter if the names of the entities are in lower case or upper case.

`COMPLEX(KIND=8) xC,Y1`
`! Equivalent to:`
`COMPLEX(8) Xc,y1`
	The entities (variables) named XC and Y1 are of type `COMPLEX` and are stored on 8 bytes each (in older versions of Fortran, the `DOUBLE COMPLEX` type was used in such a case). Since we are dealing with complex values consisting of pairs of values (the "real" part and the "imaginary" part), 16 bytes are actually be used for each entity.

`INTEGER(2),INTENT(IN) :: Q`	The Q entity is of type `INTEGER`, stored on 2 bytes and used only for input values. Since an attribute (`INTENT`) is also specified, it is mandatory to use the `::` characters to separate the left list from the one right list, even if there is only one element on the right.
`REAL,PARAMETER :: pi=3.14159`	The entity named PI is of type `REAL` and with a constant

	(unchangeable) value of 3.14159.
`EXTERNAL :: SIN`	The entity named SIN is declared as a variable with the default type `REAL` (because the name starts with the letter S). In this situation the SIN name will not be usable for the intrinsic trigonometric function in Fortran.
`REAL,POINTER,PRIVATE :: p,Q1`	The entities named P and Q1 will be pointers of type `REAL`, accessible only in the current program unit.
`IMPLICIT INTEGER(B,f-H,k)`	All entities whose name begins with one of the letters B, F, G, H or K will be of type `INTEGER` (regardless of whether they are written in uppercase or lowercase).
`IMPLICIT REAL(n),COMPLEX(A-C)`	All entities whose name begins with the letter N will be of type `REAL`, and those whose name begins with one of the letters A, B, or C will be of type `COMPLEX`.

`IMPLICIT NONE`
`INTEGER I,j,K`
`REAL X,Y`
	The implicit rule has been canceled and the types of all entities must be explicitly defined. The one named I, J and K will be of type `INTEGER`, and the ones named X and Y will be of type `REAL`. As no attributes were specified, the `::` separator was omitted (only the right is list).

`TYPE comp`
`  CHARACTER(LEN=24) name`
`  INTEGER day`
`  CHARACTER(3) month`
`  INTEGER :: year=2023`
`END TYPE`
`     …`
`TYPE(comp) r23,r24`
 	
`CHARACTER at,stars*3`
`INTEGER m1,m2,m3`
 	
`DATA at,m1,m2/"@",2*1/,m3/5/`
`DATA stars/"***"/`
`DATA r24%year,r24%day/2024,12/`
`DATA r24%month/"AUG"/`
	The derived type named COMP is defined as consisting of two character strings (NAME having 24 positions and MONTH 3) and two integers (DAY and YEAR, the latter being also initialized with value 2023). Note the optionality of the `LEN=` keyword, as it is not used for the MONTH string.
	
	Entities named R23 and R24 will have the type defined above.
	Declaration of `CHARACTER` type entities: AT will contain 1 character and STARS will contain 3 characters (an old syntax was used instead of LEN=3), followed by the declaration of `INTEGER` type entities M1, M2 and M3.
	The variable named AT gets the `@` character, the variables M1 and M2 get the value 1 (2 pieces, for the 2 entities), and M3 gets the value 5, after which the STARS string is also initialised with the `***` characters. The YEAR, DAY and MONTH components of entity R24 will be given the values 2024, 12 and AUG.

Expressions

Expressions can be arithmetic (numeric), string (character), logical, or initialisation and specification (from Fortran 90), and consist of operators, operands, and parentheses. An operand is a value represented by a constant, variable, array or array element, or resulting from the evaluation of a function. Operators can be intrinsic (implicitly recognised by the compiler and of global in nature, so always available to all sequences of code) or user-defined (when an operator is explicitly described as a function by the programmer). Depending on how they work, we can talk about unary operators (acting on a single operand) and binary operators (acting on a pair of operands). Unary operators take precedence over binary operators. Evaluating an expression always produces a single result, which can be used for attribution or as a reference. The type of value resulting from the evaluation of a numeric expression depends on the type of operands and their rank. If the operands within the expression have different ranks, the resulting value will be of the type of the operand with the highest rank (unless an operation involves a complex value and one in double precision, the result in such situations being of double complex type). When checking the correctness of a combined numerical expression, it is recommended to take into account the type of partial values resulting during the evaluation.

Arithmetic expressions, as their name suggests, represent numerical calculations, made up of arithmetic operators and operands, giving a numerical result that must be defined mathematically (division by zero, raising a base of zero value to a zero or negative power, or raising a base of negative value to a real power are invalid operations). The term numeric operand can also include logical values, since they can be treated as integers in a numerical context (the logical value `.FALSE.` corresponds to the integer value `0`). The numeric operators are: `**` (exponentiation), `*` (multiplication), `/` (division), `+` (addition), `−` (subtraction). In an arithmetic expression with several operators, the parts enclosed in parentheses (from the inside to the outside) and the functions are always evaluated first, with the evaluation priority of the intrinsic operators being as follows: exponentiation, multiplication and division, unary plus and minus, addition and subtraction. Operators with the same priority are evaluated from left to right. By local effect, unary operators can affect this rule, generating exceptions in the case of compilers that accept such expressions.

Expression	Formula

`(3*X**2+1)/(2*Y)-1`
`(3*X**2+1)/2/Y-1`
	$\frac{3x^2+1}{2y}-1$

`X/(-5)*Y`	$\frac{x}{-5}y$
`X**(-Y)*3`	$x^{-y}3$

Expression	Formula
`(3*X**2+1)/2*Y-1`	$\frac{3x^2+1}{2}y-1$

`X/(-5*Y)`
`X/(-5)/Y`
	$\frac{x}{-5y}$

`X**(-Y*3)`	x^{-y3}

Character (string) expressions can be composed using the `//` concatenation operator (in older versions of Fortran using the `+` intrinsic operator) or using programmer-defined functions, applied to `CHARACTER` type constants or variables. Evaluating such an expression produces a single string value. Concatenation is performed by joining the character contents from left to right, without parentheses affecting the result. Blanks (spaces) contained in the operands are also included in the result.

Logical expressions consists of logical or numerical operands combined with logical and/or relational operators. The result of a logical expression is normally a logical value (equivalent to one of the logical literal constants `.TRUE.` or `.FALSE.`), but logical operations applied to integer numeric values will still result in integer values, being performed bit by bit in order corresponding to the internal representation of those values. Logical operations cannot be performed directly on values of type of `REAL`, `COMPLEX` or `CHARACTER`, but these types of values can be handled using relational operands within logical expressions. The relational and logical operators are as follows:

Relational operators		
Syntax	Meaning	Older syntax*
`<`	Less Then	`.LT.`
`<=`	Less or Equal to	`.LE.`
`==`	Equal	`.EQ.`
`/=`	Not Equal	`.NE.`
`>`	Greather Than	`.GT.`
`>=`	Greather or Equal to	`.GE.`

* Older versions (marked with dots in the last column) are also allowed to be used.

Logical operators	
Syntax	Meaning
`.NOT.`	Logical negation, returns true if the operand has the value false and false if the operand has the value true.
`.AND.`	Logical conjunction, returns true only if both operands have the true value, otherwise returns false.
`.OR.`	Logical disjunction, returns true if one of the operands has the true value, otherwise returns false.
`.EQV.`	Logical equivalence, results true if both operands have the same value, if they have different values then results false.
`.NEQV.`	Logical inequality, returns true if the operands are different, and false if they are the same.
`.XOR.`	Exclusive logical disjunction (eXclusive OR), similar effect to logical inequality.

The relational operators have equal priority (they are executed from left to right, but before the logical and after the numerical), and the logical operators are executed in the order of their evaluation priority.

Relational operators are binary (they act on two operands), as are logical operators, except for the logical negation operator (`.NOT.`), which is unary.

Initialisation and specification expressions can be considered as those that contain intrinsic operations and constant parts, or a whole scalar expression. As their name suggests, they are used to initialise values (for example, the index to control an implicit cycle) or to specify properties (for example, to declare array bounds or string lengths).

There are homogeneous expressions (where the operators and operands are of the same type) and non-homogeneous expressions (where the operators and operands are of several types). The evaluation priority of operators within non-homogeneous expressions is as follows (in descending order):

- defined unary operators and functions;
- numeric operators (in the following order: `**`, `*` or `/`, `+` or `−`);
- concatenation operator for strings (characters);
- relational operators (with equal priority);
- logical operators (in order: `.NOT.`, `.AND.`, `.OR.`, `.XOR.` or `.EQV.`or `.NEQV.`).

Operands can be variables (only named entities can have variable values) or constant values. Constant values are specified according to their type, as shown in the following table:

Constant type	Examples:	Explanations:

Character string	`"Bla 3-1a"`
	`"anii '80"`
	`'anii ''80'`
		Printable characters are quoted. If there are apostrophes or quotation marks within a character string, either the inner apostrophe can be doubled (see the third string), or the other character is used for as a delimiter.

Decimal number	`231`
	`50.66`
	`-.13`
	`256.`
		The decimal separator is the point, negative values are indicated by the minus sign. Non significant digits can be omitted (the first value is an integer and the last three values are real).

Binary number	`B"1001"`
	`b"1011"`
	`B'1100'`
		When quoting the value after the `B` mark, only the digits 0 or 1 are allowed (max. 256 positions).
		The minus sign before the `B` mark has no effect and is not accepted in the quoted content (there are no such negative values). Quotations can be made either with quotation marks or with apostrophes (without combining them).

Octal number	`O"152"`
	`O'223'`
	`o"107"`
		Only the digits 0 to 7 can be used (max. 86 positions) in the value that is quoted after the `O` mark. As before, the minus sign in front has no effect and is not allowed inside.

Hex number	`Z"15F"   X"15f"`
	`Z'1B0'   x'1B0'`
	`z"A28"   x"a28"`
		The digits 0 to 9 and letters A to F can be used (max. 64 positions) by quoting the value after the `Z` or `X` mark. As before, the minus sign in front has no effect and is not accepted inside.

Hollerith	`1H&`
	`3H123`
	`12H1a "Taverna"`
	`12Hab"1 x'+#.%@`
		They are constants that can contain any printable character. Their syntax is: *n*`H`*string*, where *n* is the number of characters (positions in the string), `H` is the Hollerith mark and *string* stands for the content.
		Although these constants were originally defined to contain up to 2000 characters, the number of characters can be between 1 and 32767 ($2^{15}-1$) on 32-bit platforms, or between 1 and 2147483647 ($2^{31}-1$) on 64-bit platforms.

Intrinsic functions

Intrinsic functions are specific to the libraries used, and have predefined (reserved) symbolic names. Some of them are not part of the standard kit of the programming environment, since they are not found in all variants of the Fortran language. The fact that the names of these functions are reserved means that there should be no entities with names that coincide with those of the intrinsic functions. Also, the names of these functions are not recommended to appear in a list of an `EXTERNAL` statement, which leads to the cancellation of their intrinsic definition. In such cases, by including their names in lists of the `INTRINSIC` statement, they can be used in procedures defined as program units (user-defined subroutines or functions). The general syntax of functions is as follows:

function_name (*a*,[*a*]…)

where *function_name* is the symbolic name of the function and *a* represents the argument(s).

Some intrinsic functions, in alphabetical order of their role:

Role:	Function:	Result:
|x|	`ABS`(*x*)	The absolute value (modulus) of the specified X argument.
arccos(x)	`ACOS`(*x*)	The arccosine of the X argument expressed in radians.
arcsin(x)	`ASIN`(*x*)	The arcsine of the X argument expressed in radians.
arctg(x)	`ATAN`(*x*)	The arctangent of the X argument expressed in radians.
character	`ACHAR`(*x*)	Returns the character at position X in the code table.
complex-i	`AIMAG`(*x*)	The imaginary part of a complex number X.
complex-r	`REAL`(*x*)	The real part of a complex number X.
cos(x)	`COS`(*x*)	The cosine value of the X argument expressed in radians.
cosh(x)	`COSH`(*x*)	The hyperbolic cosine of the X argument.
e^x	`EXP`(*x*)	The exponential value of the Euler constant (e=2.71828...).
ln(x)	`LOG`(*x*)	The value of the natural logarithm of the X argument.
log(x)	`LOG10`(*x*)	The logarithm with base 10 of the X argument.
length	`LEN`(*string*)	The number of characters in the STRING considered argument.
max(x,y,...)	`MAX`(*value_list*)	The maximum value among the items contained in the argument list.
min(x,y,...)	`MIN`(*value_list*)	The minimum value among the items contained in the argument list.
random	`RAN`(*x*)	Returns a pseudorandom number between 0 and 1.
rest of div.	`MOD`(*x1,x2*)	The remainder of the argument division (X1/X2, with the sign of X1).
round	`NINT`(*x*) `ANINT`(*x*)	The value of the X argument rounded to the nearest integer. The rounded value of the X argument to zero decimal places.
sin(x)	`SIN`(*x*)	The value of the sine of the X argument expressed in radians.
sinh(x)	`SINH`(*x*)	The hyperbolic sine of the X argument.
$\sqrt{x}$	`SQRT`(*x*)	The square root (radical) of the X argument.
substring	`INDEX`(*string,ss*)	The starting position of the SS substring in the first argument STRING.
tg(x)	`TAN`(*x*)	The tangent of the X argument expressed in radians.
tgh(x)	`TANH`(*x*)	The hyperbolic tangent of the X argument.
truncate	`INT`(*x*) `AINT`(*x*)	The truncated value of the argument X to the nearest integer. Truncated value of argument X with zero decimals.

Input and output (I/O) statements

Read operations are called inputs (I) and write or display operations are called outputs (O). For sequential inputs, the `READ` statement can be used, with the following syntax variants:

`READ` *f*[, *input_list*]

when reading from the default logical unit (usually the console, so the keyboard), where *f* is the format specifier (shown later), or

`READ`([`UNIT=`]*u*[, [`FMT=`]*f*][, `ERR=`e_1][, `END=`e_2][, `IOSTAT=`*var*]) [*input_list*]

where the UNIT= keyword can be omitted if it is the first parameter and *u* is the logical unit number (the value is * for the default logical unit, i.e. console), the FMT= keyword can be omitted if it is the second parameter or if it is not desired to use a format specifier *f* (the case of reading without format), e_1 is the label of an executable instruction to jump to if the end of file (EOF) is encountered or if there are no values to read, e_2 is the label of an executable instruction to jump to if a read error is encountered, and *var* is the name of an INTEGER variable in which the success / failure of the read operation would be recorded (successful reads result in 0, unsuccessful reads result in higher values marking error codes). The entities in which the read values are to be stored form the *input_list*, If there is no *input_list*, the only effect of the instruction is to temporarily stop the execution of the program (until the <*Enter*> key is pressed).
There are also other variants, such as internal reading (to convert characters into integers corresponding to the positions in the character table), direct reading (to jump to the position number of a record in a fixed formatted logical unit), or keyed reading (in the case of indexed files).

The following statements can be used for sequential output operations:

PRINT *f*[, *output_list*]

when writing to the default logical unit (usually the console, hence the monitor display), where *f* is the format specifier, or

WRITE([UNIT=]*u*[,[FMT=]*f*][,ERR=e_1][,IOSTAT=*var*]) [*output_list*]

where the notation is the same as for reading (without END=e_2, as it makes no sense for writing). The entities whose values are to be written make up the *output_list*. If *output_list* is missing, an empty line is written (similar to the effect of the <*LF*> character, which means line feed).
There are also other variants, such as internal writing (to convert integers to characters, according to the positions in the character table), direct writing (jumping to the position number of a record in a fixed formatted logical unit), or rewriting a record. Writing to indexed files uses sequential writing with format specifier, where key fields are among the entities in *output_list*.

When * is used as a format specifier (i.e. default format), the value type in the entity list is usually taken into account. For so-called long values, such as REAL(8) or DOUBLE PRECISION, REAL(16), COMPLEX(8) or DOUBLE COMPLEX, COMPLEX(16), the default format cannot be used, so a format specification appropriate to the type must be used.

Examples:	Explanations:

`READ *`
`! Equivalent to:`
`READ(*,*)`
	Apparent reading (no input). Waiting for the <*Enter*> (carriage return) key to be pressed to continue.

`READ *,I,j`
`! Equivalent to:`
`READ(*,*)i,J`
	Two numerical values (of type INTEGER) entered from the keyboard are read and stored in variables I and J respectively. The two values can be entered separately (the program will continue only after both values have been entered) or on the same line, separated by a comma (or a blank).

`PRINT *`
`! Equivalent to:`
`WRITE(*,*)`
	A blank line will be displayed on the screen (similar to the effect of the <LF> character).

`PRINT *,”n= ”`
`! Equivalent to:`
`WRITE(*,*)”n= ”`
	The quoted string will be displayed (without the quotation marks).

`PRINT *,”Max= ”,max`
`! Equivalent to:`
`WRITE(*,*)”Max= ”,MAX`
	The quoted string will be displayed, followed by the contents (value) of the MAX variable.