Intrinsic Numeric Operations

- The following operators are valid for numeric expressions:

**	exponentiation (e.g., 10**2) evaluated right to left: 2**3**4 is evaluated as 2**(3**4)
* and /	multiply and divide (e.g, 10*7/4)
+ and -	plus and minus (e.g., 10+7-4 and -3)

- Can be applied to literals, constants, scalar and array objects. The only restriction is that the RHS of ** must be scalar, and expressions containing consecutive arithmetic operators are not allowed.

```
a = b - c     f = -3*6/5      x = y**3
a**-b  a*-b  BAD    but you can use a**(-b) and a*(-b)
```

Relational Operators

- The following **relational operators** deliver a LOGICAL result when combined with numeric operands:

  old form: .GE. .GT. .EQ. .NE. .LE. .LT.

  new form: >= > == /= <= <

- For example:

```
bool = i > j
if (i == j) c = d
```

- Use of the relational operators == and /= with floating point numbers (real variables) is **extremely dangerous** because the value of the numbers may be different from the expected mathematical value due to radix conversion and roundoff errors.

* INTEGERs are stored **exactly** (often in the range -32767 to 32767)
* REALs are stored **approximately**.
  * They are partitioned into a mantissa and an exponent, 6,6356 x 10**23
  * The exponent can take only a small range of values.
* **Instead, compare against a suitable range or tolerance.**

IF (a == b) then ...   this is BAD!!!
IF (ABS(a-b) <= EPS) ... where EPS is thoughtfully chosen!!!!

Intrinsic Logical Operators

- A LOGICAL or boolean expression returns a .TRUE. or .FALSE. result. The following are valid LOGICAL operands:

.NOT. : .true. if operand is .false.
.AND. : .true. if both operands are .true. (logical conjunction)
.OR. : .true. if at least one operand is .true. (logical disjunction)
.EQV. : .true. if both operands are the same (logical equivalence)
.NEQV. : .true. if both operands are different (logical nonequivalence)

- For example:

x = 5 > 3 ==> .true. y = 4*3>15 ==> false
.NOT. x is .false., .NOT. y is .true.
x .AND. y is .false., x .AND. x is .true.
x .OR. y is .true., y .OR. y is .false.
x .EQV. y is .false., x .EQV. x is .true., y .EQV. y is .true.
x .NEQV. y is .true., x .NEQV. x is .false., y .NEQV. y is .true.

Intrinsic Character Operations

Consider:

```
character(len=*), parameter :: str1 = “abcdef”
character(len=*), parameter :: str2 = “xyz”
```

Substrings can be taken:

```
str1(1:1) is ‘a’ ; str1(2:4) is ‘bcd’
```

The concatenation operator, //, is used to join two strings:

```
print*, str1 // str2
print*, str1(4:5) // str2(1:2)
```

would produce

```
abcdefxyz
dexy
```

Operator Precedence

Operator	Precedence	Example
user-defined monadic	highest	.INVERSE. A
**	.	10**4
* or /	.	89*55
monadic + or -	.	-4
dyadic + or -	.	5+4
//	.	str1//str2
>, >=, <, <=, etc.	.	A > B
.NOT.	.	.NOT. Bool
.AND.	.	A .AND. B
.OR.	.	A .OR. B
.EQV. or .NEQV.	.	A .EQV. B
user-defined dyadic	lowest	x .DOT. y

* In an expression with no parentheses, the highest precedence operator is combined with its operands first.
* In context of equal precedence, **left to right** evaluation is performed except for ** (exponentiation), which is performed **right to left**.

  2**3**2 = 512 (2**9)

* **Example:** The following expression

```
x = a+b/5.0-c**d+1*e
```

is equivalent to

```
x = a + (b/5.0) - (c**d) + (1*e)
```

as ** is highest precedence, and / and * are next highest. The remaining operators precedences are equal, so we evaluate from left to right.

Flow Control

Control constructs allow the normal sequential order of execution to be changed. Fortran 90 supports:

* Conditional execution statements/constructs (IF and IF-THEN-ELSEIF-ELSE-ENDIF)
* Loops (DO-ENDDO)
* Multi-way choice construct (SELECT CASE)

IF Statement

The basic syntax is

IF (<logical-expression>) <exec-statement>

If <logical-expression> evaluates to .TRUE., then execute <exec-statement>, otherwise do not.

For example:

```
if (x > y) maxval = x
```

means "if x is greater than y then set maxval to be equal to the value of x".

More examples:

```
if (a*b+c <= 47) Boolie = .true.
if (i /= 0 .and. j /= 0) k = 1/(i*j)
```

IF...THEN...ELSE Construct

The **block-IF** is a more flexible version of the single line IF. A simple example:

```
if (i == 0) then
  print*, “i is zero”
else
  print*, “i is NOT zero”
endif
```

You can also have one or more **ELSEIF** branches:

```
if (i == 0) then
  print*, “i is zero”
elseif (i > 0) then
  print*, “i is greater than zero”
else
  print*, “i must be less than zero”
endif
```

And you can use multiple **ELSEIF** branches. The first branch to have a true logical-expression is the one that is executed. If none are found, then the **ELSE** block (if present) is executed.

```
if (x > 3) then
  call sub1
elseif (x < 2) then
  a = b*c - d
elseif (x < 1)
  a = b*b
else
  if (y /= 0) a = b
endif
```

Notice how you can **nest** if-blocks.

Nested and Named IF Constructs

All control constructs can be both named and nested:

```
outa: if (a /= 0) then
  print*, "a /= 0"
  if (c /= 0) then
     print*, 'a/ = 0 AND c/= 0'
  else
     print*, 'a /= 0 BUT c == 0'
  endif
elseif (a > 0) then outa
  print*, "a > 0"
else
  print*, "a must be < 0"
endif outa
```

The names may only be used **once** per program unit and are only intended to make the code cleaner.

DO Loops

The general form of a DO loop is:

```
[name:] do [control clause]
  [block of code]
enddo [name:]
```

There are three possible control clauses:

* Iterative (or indexed)
* While
* Empty (use **EXIT** and **CYCLE**)

Indexed DO Loops

Loops can be written which cycle a fixed number of times. For example:

```
do i = 1, 100, 1
    ...  ! i is 1, 2, 3, ..., 100
enddo
```

The formal syntax is:

```
do <do-var> = <expr1>, <expr2> [,<expr3>]
    <executable statements>
enddo
```

The number of iterations, which is evaluated **before** execution of the loop begins, is calculated as

```
MAX(INT((<expr2> - <expr1> + <expr3>) / <expr3>), 0)
```

If this is zero or negative then the loop is not executed.

If <expr3> is absent it is assumed to be equal to 1.

Examples of Loop Counts

1. Upper bound not exact:

```
do i = 1, 30, 2
    ...  ! i is 1, 3, 5, 7, ..., 29
    ...  ! 15 iterations
enddo
```

2. Negative stride:

```
do j = 30, 1, -2
    ...  ! j is 30, 28, 26, 24, ..., 2
    ...  ! 15 iterations
enddo
```

3. A zero-trip loop:

```
do k = 30, 1, 2
    ...  ! 0 iterations -- loop skipped
enddo
```

Exit DO Loops

You can also set up a DO loop which is terminated by simply jumping out of it with an **EXIT** statement. Consider:

```
i = 0
do
  i = i + 1
  if (i > 100) exit
  print*, “i is “, i
enddo
! if i>100 control jumps here
print*, “Loop finished.  i now equals”, i
```

Example: exitloop.f90

Conditional Cycle Loops

You can set up a DO loop which, on some iterations, only executes a subset of its statements. Consider:

```
i = 0
do
  i = i + 1
  if (i >= 50 .and. i <= 59) cycle
  if (i > 100) exit
  print*, "i is ", i
enddo
print*, "Loop finished.  i now equals", i
```

CYCLE forces control to the **innermost** active DO statement and the loop begins a new iteration.

```
i is 1
i is 2
...
i is 49
i is 60
...
i is 100
Loop finished.  i now equals 101
```

Named and Nested Loops

Loops can be given names and an **EXIT** or **CYCLE** statement can be made to refer to a particular loop:

```
outa: do
   inna: do
      ...
      if (a > b) EXIT outa
      if (a == b) CYCLE outa
      if (c > d) EXIT inna
      if (c == a) CYCLE
   enddo inna
enddo outa
```

The (optional) name following the **EXIT** or **CYCLE** highlights which loop the statement refers to.

Loop names can only be used once per program unit.

EXAMPLE: nested_loops.f90

DO WHILE Loops

- The general form of a DO loop is:

```
[name:] do while [logical expression]
  [block of code]
enddo [name:]
```

- Generally the body of the do-loop will modify one of more of the variables contained or affecting the logical expression test.

```
do while (diff > somevalue)
  .
  .
  diff = ABS(old-new)
  .
enddo
```

SELECT CASE Construct

This is very useful if one of several paths must be chosen based on the value of a single expression.

The syntax is:

```
[<name>] select case (< case-expr >)
    case (< case-selector >) [ <name> ]
       < exec-statements >
    case default [ <name> ]
       < exec-statements >
end select [ <name> ]
```

Notes:

* the < case-expr> must be scalar and INTEGER, LOGICAL or CHARACTER valued.

* the < case-selector > is a parenthesised single value or range. for example, (.true.), (1), or (99:101).

* there can only be one CASE DEFAULT branch.
* control cannot jump into a CASE construct.

* **EXAMPLES:** select_example.f90 and select_example2.f90

Mixed Type Numeric Expressions

In the CPU calculations must be performed between objects of the **same** type, so if an expression mixes type some objects must change type.

Default types have an implied ordering:

1. INTEGER -- **lowest**
2. REAL
3. DOUBLE PRECISION
4. COMPLEX -- **highest**

The result of an expression is always of the **highest** type. For example:

* INTEGER * REAL gives a REAL **(3 * 2.0 = 6.0)**
* REAL * INTEGER gives a REAL **(3.0 * 2 = 6.0)**
* DOUBLE PRECISION * REAL gives DOUBLE PRECISION
* COMPLEX * <any type> gives COMPLEX
* DOUBLE PRECISION * REAL * INTEGER gives DOUBLE PRECISION

The actual operator is unimportant.

Mixed Type Assignment

Problems often occur with mixed-type arithmetic. The rules for type conversion are given below.

- INTEGER = REAL

  the RHS is evaluated, truncated (all of the decimal places lopped off) and assigned to the LHS.

- REAL = INTEGER

  the RHS is promoted to be REAL and stored (approximately) in the LHS.

Example: program mixedassign.f90

Intrinsic Procedures

Fortran 90 has over 100 built-in or intrinsic procedures to perform common tasks efficiently. They below to a number of classes:

* **Elemental**
  * Mathematical (SQRT, SIN, LOG, etc.)
  * Numeric (ABS, CEILING, SUM, etc.)
  * Character (INDEX, SCAN, TRIM, etc.)
  * Bit (IAND, IOR, ISHFT, etc.)
* **Inquiry** (ALLOCATED, SIZE, etc.)
* **Transformational** (REAL, TRANSPOSE, etc.)
* **Miscellaneous** or non-elemental subroutines (SYSTEM_CLOCK and DATE_AND_TIME)

Introduction to Formatting

Fortran 90 has extremely powerful, flexible and easy-to-use capabilities for output formatting.

* The default formatting may be sufficient on your computer for now, but sometimes roundoff error causes "ugly" looking real values.
  * It's not a malfunction of the computer's hardware, but a fact of life of finite precision arithmetic on computers.
  * Replace the asterisk denoting the default format with a custom format specification.
  * Example: add_2_reals.f90

Edit Descriptors

The three most frequently used edit descriptors are:

* f (floating point) for printing of reals

  syntax: fw.d

  w = total number of positions

  d = number of places after the decimal point

  - the decimal point occupies a position, as does a minus sign

* a (alphanumeric) for character strings
* i (integer) for integer - can use iw.d format, where the d will pad in front of the value with zeroes

Also the new line (/) and tab (t) edit descriptors.

Example: format_examples.f90