Nottran Specification

Version 1.0.0-rc3

1 CONTENTS

2 Introduction

This document specifies the Nottran language used in COMP3012 Compilers. It is based on Fortran 90 with some simplifications. The Fortran 90 specification is linked in the reading list.

3 Changelog

3.1 2025-10-20 Release candidate 3

Added

- `read`, `result`, `type`, and `write` added to the list of reserved keywords
- textual versions of relational operators
- specified that the loop values in a `do` statement must be integers
- specified that lower bound of `do` is inclusive and upper bound is exclusive
- specified the permitted arguments to read and write statements

Changed

- reordered statements and expressions
- updated numeric values of valid characters to all use decimal ASCII instead of a random mix of decimal and octal
- clarified that derived types cannot include pointers
- clarified that arrays of arrays and pointers to pointers are not allowed
- replaced `!=` with `/=` in table of operator precedence, to match rest of grammar
- fixed bug in computing the dot product in extended example

Removed

- `begin`, `break`, `continue`, and `return` removed from list of reserved keywords (left over from a previous version)
- delete character no longer a valid character for use in character literals

4 LEXICAL GRAMMAR

A Nottran program is made of the following kinds of token: keywords, names, literal values, and special symbols.

All whitespace in between tokens is ignored in Nottran; indentation is used only for clarity and carries no meaning.

4.1 BASIC DEFINITIONS

A **letter** is a single lower-case letter a-z. ISO Fortran allows a mix of upper- and lower-case letters, but for simplicity Nottran is limited to lower-case only.

A **digit** (or **decimal** digit) is a single number 0-9. A **binary digit** is a 0 or a 1, an **octal digit** is a number 0-7 and a **hexadecimal digit** is a number 0-9 or a letter a-f.

An **alphanumeric** character is either a letter or a (decimal) digit.

4.2 KEYWORDS

Keywords are sequences of characters with specific meaning in Nottran. Each keyword forms a token class by itself. The following is a complete list of keywords:

- `allocate`
- `break`
- `call`
- `character`
- `deallocate`
- `do`
- `else`
- `end`
- `function`
- `if`
- `integer`
- `logical`
- `pointer`
- `program`
- `read`
- `real`
- `result`
- `subroutine`
- `then`
- `type`
- `while`
- `write`

4.3 NAMES

A name is a sequence of characters used to identify a program, function, subroutine or variable. Valid names begin with a letter and may be followed by any number of alphanumeric characters or underscores.

Note that all keywords also meet the definition of a name; a valid name may never be spelled the same as a keyword.

4.4 CONSTANTS

Constants are fixed values in a program. Constants are subdivided into **literal constants** and **named constants.**

4.4.1 Literal constants

A literal constant is an explicitly defined value of one of the intrinsic types of the language. Unlike Fortran, Nottran does not support complex values, but does support integers, reals, logical values, and characters.

4.4.1.1 Integers

Integers are usually specified in base 10. They are sequences of at least one decimal digit character, with an optional preceding sign character (+ or -).

```
42
+123
-876
```

Integers of alternate bases are defined as ‘BOZ’ literals. These literals use a letter to indicate the base and then include a sequence of digits inside double quotes.

```
b“0110101000”
o“734502”
z“af132cd”
```

For unknown reasons, Fortran uses Z to prefix hexadecimal values. ‘BOZ’ literals are always unsigned.

4.4.1.2 Reals

Real numbers are also known as floating point numbers. They have an optional sign, and a sequence of digits before and after a decimal point. Either the before or after part may be omitted.

```
12.345
+67.8
25.
-.777
```

4.4.1.3 Logical values

The two logical values in Nottran are written `.true.` and `.false.` including the full stops. They have their obvious meaning.

4.4.1.4 Characters

Character literals are printable values. Any printable ASCII character is a valid character value (i.e. characters 32-126), as is the horizontal tab (9). Character literals are written in between double quotes, even when there is only one character.

```
“boop”
“7”
“This isn’t NOT a character sequence?!”
```

Double quotes may be escaped inside a character literal by typing a double-double quote in this way:

```
“The name ““Ian”” is given to the most boring of programmers.”
```

4.4.2 Named constants

Any valid name can form a named constant on its own. Its meaning depends on its previous definitions and assignations.

4.5 Special characters

Special characters are used for operator symbols, bracketing, separating other tokens, and certain other special uses.

4.6 Comments

Comments in Nottran begin with an exclamation mark and may contain the same characters as a character sequence. Comments run until the end of the line and are entirely ignored. Note that an exclamation mark inside a character literal should be treated as part of the character literal, not as a comment

```
! This is a comment
! This is a separate comment
x = 32 ! This is a third comment, but the assignment is still parsed
```

5 Syntax

The syntax of Nottran is described from 'top to bottom', beginning with the syntax of a complete Nottran file and focusing on smaller scopes. The following Nottran program for calculating the greatest common divisor of two values showcases most of the common syntax:[1]

```
function gcd(a, b) result(ans)
! Recursive implementation of Euclid's algorithm
    integer :: a, b, ans

    if (b <= 0) then
        ans = a
    else
        ans = gcd(b, a-b)
    end if
end function gcd

program gcd_main
! Use Euclid's algorithm to compute GCD of two user-specified values
    integer :: x, y, ans

    read x, y
    ans = gcd(x, y)
```

[1] The separate function is of course technically unnecessary as Euclid's algorithm can be defined as a loop, but it is included to demonstrate the function definition syntax.

```
    write ans
end program gcd_main
```

5.1 PROGRAM UNITS

A well-formed Nottran file contains a single `program` (equivalent to the main function in C) and any number of other program units. Program units may appear in any order in the file.

5.1.1 Programs

A Nottran program begins with the keyword `program` followed by the name of the program. A program contains one or more statements (see below) and finishes with the keywords `end program` and the program name again.

```
program foo
    ! program goes here
end program foo
```

All variables that will be used in the program must be declared before anything else (see below for the syntax).

5.1.2 Subroutines and functions

Subroutines and functions are reusable sections of code. A subroutine is a section of code that does not return a value; a function has a return value. Their syntax is like that of a program except for the keyword used to declare them and the list of parameters.

```
subroutine foo(a, b, c)
    ! vars a, b, c must be declared
    ! statements go here
end subroutine foo
```

The parameter list is a sequence of comma-separated names.

Functions return by assigning a value to a specified result variable. By default, the result variable has the same name as the function (which in turn prevents recursion). Optionally, the programmer can specify a different result variable which would allow the function to be recursive.

```
function foo(a, b)
  integer :: a, b, foo ! result variable must have a type
  foo = a + b ! returns a + b
end function foo
```

```
function bar(a, b) result (c)
  integer :: a, b, c ! bar doesn’t need an assigned type
  c = a + b ! also returns a + b, but allows recursively calling bar
end function bar
```

All subroutines and functions must declare all the variables they will use before anything else (see below). This includes the parameters: each parameter must have its type specified by a variable declaration.

5.1.3 Derived types

Derived types are defined by the user, essentially the same as a struct in C. They use the `type` keyword and contain a sequence of variable declarations that define their content. None of the variables can be pointers (see below).

```
type pixel
    integer :: r, g, b
    real :: alpha
end type pixel
```

Previously defined types then become available for use in variable declarations and other derived type declarations. Types **may not** be self-referential.

```
type (pixel) :: p0, p1
```

```
type image
    type (pixel) (256, 256) :: pixels
    character :: alt_text
end type image
```

The derived type `image` above has a single member called `pixels`, which is a 256x256 array of individual `pixel` values, and a character member called `alt_text`.

5.1.4 Variable declarations

Variable declarations are not program units themselves, but appear as part of other units. A variable declaration consists of a type name, the special symbol `::`, and then a comma-separated list of names indicating which variables should have that type.

```
integer :: a, b
real :: c
character :: d, e, f
logical :: is_ok, is_bad
```

There is no way of declaring and defining a variable on the same line in Nottran.

Variables are visible in the entire section (program, subroutine, or function) they are declared in, and not visible anywhere else.

Arrays can be declared by including the required size of the array in parentheses. Multi-dimensional arrays are specified by giving the sizes of all required dimensions in a comma-separated list:

```
integer(10) :: nums ! equivalent to int[10] in C
real(5,5) :: more_nums ! equivalent to float[5][5] in C
```

Any type other than another array can be formed into an array.

5.1.5 Pointer declarations

Nottran supports dynamic memory allocation. To declare a variable as a pointer, add they keyword `POINTER` to its type name before the `::` operator. If the variable is to be a dynamically allocated array, replace each dimension with a `*`. Pointers to pointers are not allowed.

```
real pointer :: x
character(*) pointer :: strs
integer(*,*) pointer :: matrix
```

The value of these variables is undefined until they have been allocated (see below).

There is no way to directly manipulate the address of a pointer variable. Once allocated, pointer variables function identically to non-pointer variables.

5.2 STATEMENTS

Programs, subroutines, and functions are made up of statements. Statements control what should be calculated in what order. The various kinds of statements are described below.

5.2.1 Assignment

The assignment statement uses the = operator to set the value of some variable. The left-hand side must be a name, array index, or field of a derived type, and the right-hand side must be an expression (see later). Note that unlike some languages, assignment is a statement, not an expression: an assignment cannot be embedded in the logical test of an if-statement, for example.

```
x = 55
y(2) = 7
z % foo = “mega”
a % m(2,2) = 8
```

5.2.2 Call

A subroutine call statement activates a subroutine. It uses the `CALL` keyword to distinguish it from a function call which may have a return value.

```
call mysub(a, b)
```

The actual parameters to the call are given in the same way as the formal parameters in the definition.

5.2.3 If

An if statement is used for conditional execution of a single statement. It is a simpler version of the full if-then-else statement (see below). The expression in parentheses should be a logical expression.

```
if (somevar) x = 5 ! x set to 5 only if somevar is true
```

Only one statement can be listed after the condition. If multiple statements are needed, use the fuller form below.

5.2.4 If-then-else

The if-then-else statement is used for conditional execution. It specifies a logical expression that should be evaluated, and at least one statement that should be executed if it evaluates to true. Optionally, a different set of statements can be specified to be executed if the expression evaluates as false.

```
if (.true.) then
    ! do something
end if
```

```
if (x /= 2) then
    ! do something
else
    ! do something else
end if
```

5.2.5 Do loops

The do loop is like the for loop in other languages. A do loop specifies a variable name, the lower (inclusive) and upper (exclusive) bounds of the values it should take, and optionally the amount by which the variable should change each iteration. If the last value is not given it defaults to 1. The loop parameters must all be integers. The following do loop:

```
do i = 1, 10, 2
    ! do stuff with i
end do
```

means to assign 1 to the variable `i`, do something with it, then increment it by 2 and repeat until `i` reaches 10. This would be equivalent to the following for loop in C:

```
for (i = 1; i < 10; i += 2) {
    // do stuff with I
}
```

Note that the variable `i` must have already been declared.

5.2.6 Do-while loops

Do-while loops specify a logical test, and the statements contained in the body of the statement are repeatedly carried out so long as the expression is true, in the same way as a while loop in C. The expression is re-evaluated after every iteration.

```
do while (i < 10)
    i = i + 1
end do
```

5.2.7 Read and write

Read and write are special statements used to get input from the user and print output to the screen, respectively. Read expects a sequence of comma-separated variable names as arguments. Write expects a sequence of comma-separated expressions of any kind.

The read statement will read one value for each specified variable from standard input, waiting for the RETURN key to be pressed after each input. The write statement will print the value of each specified variable to standard output, with a newline following each.

```
read x, y
write y, 1
```

5.2.8 Allocate and deallocate

Allocate and deallocate are used to dynamically assign and free memory. Allocate requires the name of the variable that will hold the assigned address, and if the variable is an array the exact size of each dimension must be specified using either a literal integer or an integer variable. Deallocate just takes the name of the variable to free.

```
integer pointer :: x
integer(*) pointer :: y
allocate x
allocate y, 10 ! second argument could be a variable name instead
deallocate x
deallocate y
```

5.3 Expressions

Expressions have values that can be calculated. They consist of simple values, the results of applying operators, and calling functions. Expressions may be surrounded with parentheses to modify the normal rules of precedence and associativity.

5.3.1 Simple expressions

A variable name, number, character (or character sequence) or logical value forms an entire expression on its own. Variable names have the value that was most recently assigned to them, and other simple expressions have their natural meaning.

Array indexing is performed using parentheses. This is the same way Fortran did it; it is confusing because it looks the same as calling a function, but we stick with tradition. Arrays are 0-indexed.

```
integer(5) :: arr
integer(3,3) :: m
arr(1) = 77
m(0, 2) = 8
```

Individual fields of a derived type are accessed using the % operator.

```
p0 % r
p1 % alpha
```

5.3.2 Arithmetic operators

There are five arithmetic operators in Nottran: exponentiation, multiplication, division, addition, and subtraction. They may only be applied to integer or real values, and

they have their usual meaning. When dividing two integers, the result will be the floor of the true rational result.

The result of an arithmetic operation has the same type as its second argument.

Table 1 shows the order of precedence of all operators in Nottran along with their associativity.

Precedence	Operators	Associativity
9	`( .. )`	n/a
8	% (field access, **not** modulo)	Left
7	`**`	Right
6	`*, /`	Left
5	`+, -`	Left
4	`//`	Left
3	`<, <=, >, >=, ==, /=` `.lt., .le., .gt., .ge., .eq., .neq.`	Left
2	`.and.`	Left
1	`.or.`	Left

Table 1 - Operator precedence

5.3.3 Concatenation operator

The concatenation operator `//` can only be applied to characters or character sequences and has the effect of producing a new character sequence consisting of the first operand followed by the second with no additional characters inserted in between. It is left associative.

Example: `“a” // “bcd” == “abcd”`

5.3.4 Relational operators

The relational operators test ordering and equality between values. Integers and reals are compared in the usual way, and characters and character sequences are compared lexicographically. Logical values can be compared using equal-to and not-equal-to, but not any of the others.

Relational operators produce a logical result.

- Less-than (`<` or `.lt.`)
- Less-than or equal-to (`<=` or `.le.`)
- Greater-than (`>` or `.gt.`)
- Greater-than or equal-to (`>=` or `.ge.`)
- Equal-to (`==` or `.eq.`)
- Not equal-to (`/=` or `.neq.`)

5.3.5 Logical operators

The two logical operators are `.and.` and `.or.`. Both operands to these operators must be logical values, and the result is the Boolean conjunction or disjunction of the operands. They have lower precedence than all other operators.

5.3.6 Function calls

Functions are called by giving the name of the function and the list of actual parameters as a comma-separated list in the same way as for calling subroutines. The `call` keyword is not used for function calls (see subroutine calls above).

```
myfun(a, b, c)
```

All Nottran values are passed by value to a function. Changing the value of a parameter inside a function will not affect the variable it came from in the calling scope.

The return type of a function is determined by the type of its result variable (see above).

6 EXTENDED EXAMPLE

This section provides a much longer example of a Nottran program, which asks the user to input the values for two 3x3 matrices, computes their dot product, and prints the result.

This program isn't meant to demonstrate the most efficient way of doing this, it simply serves as an example of a more complex Nottran program.

```
function read_matrix() result(m)
    integer(3,3) :: m
    integer :: i, j

    write “Enter 3x3 matrix values:”
    do i = 0, 3
        do j = 0, 3
            read m(i,j)
        end do
    end do
end function read_matrix

function is_identity(m)
    integer(3,3) :: m
    logical :: is_identity

    is_identity =
        m(0,0) == 1 .and. m(0,1) == 0 .and. m(0,2) == 0 .and.
        m(1,0) == 0 .and. m(1,1) == 1 .and. m(1,2) == 0 .and.
        m(2,0) == 0 .and. m(2,1) == 0 .and. m(2,2) == 1
end function is_identity
```

```
subroutine write_matrix(m)
    integer(3,3) :: m
    integer :: i, j

    write “Showing dot product of two 3x3 matrices:”
    do i = 0, 3
        do j = 0, 3
            write m(i,j)
        end do
    end do
end subroutine write_matrix

function dot(a, b) result (c)
    integer(3,3) :: a, b, c
    integer :: i, j, i1, j1

    do i = 0, 3
        do j = 0, 3
            i1 = 0
            j1 = 0
            do while (i1 < 3 .and. j1 < 3)
                c(i,j) = a(i,j1) * b(i1,j)
                i1 = i1 + 1
                j1 = j1 + 1
            end do
        end do
    end do
end function dot

program dot_product
    integer(3,3) :: a, b, c

    a = read_matrix()
    b = read_matrix()

    if (is_identity(a)) then
        c = b
    else
        if (is_identity(b)) then
            c = a
        else
            c = dot(a, b)
        end if
    end if
    call write_matrix(c)
end program dot_product
```