Programming Paradigms - SE2052

Worksheet 2

Discussion Questions

1. **Why was efficiency so important for early-generation programming languages?**

Early programming languages emphasized efficiency due to the restrictions modern computers face in comparison to the 1940s to 1960s machines.

- Limited processing power
- Tiny memory capacity
- High cost of computing time
- Specialized hardware

To sum, the need of space and time was the de facto rationale calculation for execution. It wouldn't be considered efficient for the slow and overburdened attempt to achieve value through execution.

2. **What were the advantages and disadvantages of pseudocode interpreters?**

Advantages

- Easier for novices to comprehend language.
- Easier to draft and evaluate concepts without the need to consider a specific programming language's syntax rules.
- Helpful during the organization of procedures prior to coding.

Disadvantages

- Significantly lower performance than compiled languages due to the need for real-time translation.
- Lack of uniformity; pseudocode varies from author to author and tool to tool.
- Lacks the comprehensiveness of a full programming language's features.

3. **Does FORTRAN comply with the syntactic consistency principle? Consider: (-B + SQRT(B**2-4*A*C)) / (2 * A)**

Yes, in this instance, FORTRAN adheres to the syntactic consistency principle because:

- All arithmetic operators (+, -, *, /, **) have a consistent application across all contexts.
- Operator precedence follows predictable, established mathematical hierarchies (** > *// > +/-).

- Parentheses work in standard algebraic fashion, providing control over evaluation order.
- Overall, the expression resembles conventional mathematical notations and, therefore, syntax mirrors operations performed.

Nevertheless, early FORTRAN had slight departures from strict mathematical uniformity:

- Mathematical operations such as SQRT utilize function names instead of symbol operators.
- Changes in type (integer vs. real) could alter the outcome in a computation without an apparent change in syntax.

4. Compare FORTRAN's syntax to C or Python. Which is more consistent?

Language	Strengths in Consistency	Weak Spots in Consistency
FORTRAN	Arithmetic syntax close to math; consistent operator precedence	Special function names (SQRT, ABS) instead of operators; integer vs. real division behaves differently without syntax change
C	Mostly consistent operators; predictable precedence	Different symbols for similar concepts (= vs ==); * can mean multiply or pointer dereference; implicit type conversions
Python	Very consistent syntax; explicit integer vs. float division; logical ops use clear keywords; same function call style for built-ins and user functions	Some math still requires named functions (math.sqrt)

5. The GOTO statement is unavoidable in early FORTRAN. How does it affect readability, writability, and reliability?

Aspect	Effect of GOTO in Early FORTRAN
Readability	Reduces readability control flow jumps arbitrarily to labeled lines, forcing the reader to trace code back and forth to understand logic ("spaghetti code").
Writability	Can make writing quick for short programs easier to implement loops and branches without complex structures. But for larger programs, it makes maintaining and extending code harder.
Reliability	Hurts reliability increases the chance of logic errors, infinite loops, and hard-to-find bugs because program flow is harder to reason about and verify.

6. Does FORTRAN allow for information hiding (a form of abstraction)?

FORTRAN is one of the oldest computer programming languages, and early versions of FORTRAN do not include information hiding as a form of abstraction. This is because of the lack of features like encapsulation, access control, or any form of access restrictions.

FORTRAN subroutines and variables from those early periods had a flat structure such that all of them were accessible from anywhere within the program. Such a design leads to lack of control over hiding the internal workings of the program. Consequently, provided structures and algorithms were essentially global, mitigating the ability to write modular and secure code. Through the introduction of Fortran 90 and later versions, FORTRAN incorporated the use of modules which improved information hiding and provided features of abstraction. Such modules and information hiding concepts were not a part of early FORTRAN versions.

7. **What are the problems with the COMMON block statement?**

- Global sharing of information.
- No encapsulation.
- Name disputes.
- Maintenance problems.
- Debug issues.
- Poor modularity.

8. **What is the computed GOTO statement? How does it work?**

It was an older control flow statement in FORTRAN and branching goes to one of a number of labelled statements if an integer expression is true or false.

It allows multi-way branching without employing multiple IF statements.

Syntax Example

GOTO (label1, label2, label3), expression

The expression is evaluated to an integer value, say n.

The program jumps to the label in the list that corresponds to position n. For example:

- If expression = 1, jump to label1
- If expression = 2, jump to label2
- If expression = 3, jump to label3

If expression is less than 1 or greater than the number of labels, program behavior is undefined or may cause an error.

9. **What were FORTRAN's major contributions to programming?**

- First high-level programming language.
- Introduction of compiled languages.
- Efficient numerical computation.
- Structured control constructs.
- Standardization.

- Influence on later languages.
- Encouraged algorithmic thinking.

10. What were FORTRAN's major contributions to programming?

- Particularly for scientific and numerical computing, its highly optimized compilers produce incredibly fast code.
- There is still a significant amount of FORTRAN-based legacy scientific and engineering code that is being actively maintained.
- The accuracy and functionality of FORTRAN are essential to many scientific libraries and applications.
- Mathematical computations and array operations benefit greatly from its syntax and structure.

11. Name two major problems associated with FORTRAN. Provide examples.

Use of GOTO leading to spaghetti code (poor readability and reliability)

Example: Early FORTRAN programs heavily relied on GOTO statements for flow control, which caused tangled jumps that made programs hard to follow and debug.

```
10 IF (X .GT. 0) GOTO 20
   PRINT *, 'X is zero or negative'
   GOTO 30
20 PRINT *, 'X is positive'
30 CONTINUE
```

This style quickly becomes confusing in larger programs.

Lack of information hiding and encapsulation (poor modularity)

Example: Variables in COMMON blocks are globally shared, making it difficult to track or protect data across modules.

```
COMMON /BLOCK/ A, B, C
```

Any subroutine using this COMMON block can modify A, B, or C, risking unintended side effects and bugs.

12. How does the INTEGER type affect syntax design in FORTRAN?

- Explicit type declaration.
- Different behavior in operations.
- No overloading of operators.
- Influences control structures.

13. **Name three features that could be removed to make FORTRAN more secure.**
    - GOTO Statement

    It introduces random jumps in code which makes program execution flow more difficult to follow and it leads to more errors and chances of malicious exploits.

    - COMMON Blocks

    They allow unconstrained global access to variables that could be modified by other routines inadvertently and leaves the program exposed to security exploits.

    - Lack of strong type checking and bounds checking

    FORTRAN did not historically enforce bounding on arrays and on conversions between disparate types such as CHAR and INTEGER. Thus, buffer overflows and non-determinate behavior arises.

14. **New: How were loops implemented in early FORTRAN before DO loops?**

    Prior to the introduction of the DO loop, early FORTRAN used GOTO statements with labels and conditional tests to implement loops.

    How it worked:

    - You would set a loop counter variable by hand.
    - You would create a label that marked the beginning of the loop body.
    - At the end of your loop, you would use a GOTO statement and jump back to the label if the loop condition was still true.

Comprehensive Exercise

Code

```
program guessing_game
  implicit none
  integer :: choice, target, guess, attempts, total_games
  real :: R

  total_games = 0

  ! Main menu loop
  10 continue
  print *, "==== Number Guessing Game Menu ===="
  print *, "1. Start New Game"
  print *, "2. Show Total Games Played"
  print *, "3. Exit"
  print *, "Enter your choice (1-3):"
  read(*,*) choice

  select case (choice)
  case (1)
    ! Start new game
    call random_number(R)
    target = int(R * 100) + 1
    attempts = 0
    print *, "I have chosen a number between 1 and 100. Try to guess it!"

    ! Guessing loop with GOTO
    20 continue
    print *, "Enter your guess:"
    read(*,*) guess
    attempts = attempts + 1

    if (guess > target) then
      print *, "Too High!"
      goto 20
    else if (guess < target) then
      print *, "Too Low!"
      goto 20
    else
      print *, "Correct! You guessed it in", attempts, "attempt(s)."
    end if

    total_games = total_games + 1
    goto 10

  case (2)
    ! Show total games played
    print *, "Total games played so far:", total_games
    goto 10

  case (3)
    ! Exit program
    print *, "Thanks for playing. Goodbye!"
    stop

  case default
    print *, "Invalid choice. Please select 1, 2, or 3."
    goto 10
  end select

end program guessing_game

```

Output

```
 1. Start New Game
 2. Show Total Games Played
 3. Exit
 Enter your choice (1-3):
1
I have chosen a number between 1 and 100. Try to guess it!
 Enter your guess:
60
Too High!
 Enter your guess:
80
Too High!
 Enter your guess:
30
Too High!
 Enter your guess:
5
Too Low!
 Enter your guess:
8
Too High!
 Enter your guess:
7
Correct! You guessed it in           6 attempt(s).
 ==== Number Guessing Game Menu ====
 1. Start New Game
 2. Show Total Games Played
 3. Exit
```