Extreme INTERCAL Sample: Arbitrary-Precision Fibonacci Generator

This document contains an extremely long, deeply informative, and technically intricate sample program written in INTERCAL. The program is crafted to demonstrate mastery of INTERCAL syntax, low-level memory manipulation, non-linear flow control, and obfuscation. It computes the Fibonacci sequence using INTERCAL's unusual constructs.

```
PLEASE NOTE THAT THIS PROGRAM COMPUTES FIBONACCI NUMBERS IN A HIGHLY OBSCURE WAY
DO :1 <- #1             (Counter for Fibonacci index)
DO :2 <- #0             (Fibonacci(n-2))
DO :3 <- #1             (Fibonacci(n-1))
DO :4 <- #100            (How many Fibonacci numbers to generate)

PLEASE GIVE UP          (Let's begin a loop using COME FROM)

(DEFINE A FAKE ENTRY POINT)
DO .1 <- #0             (Control variable to simulate GOTO in INTERCAL)
DO .2 <- #0             (Memory cell for output preparation)

(OUTPUT INTRO TEXT)
PLEASE WRITE IN #261      (ASCII: 'F')
PLEASE WRITE IN #265      (ASCII: 'i')
PLEASE WRITE IN #262      (ASCII: 'b')
PLEASE WRITE IN #265      (ASCII: 'i')
PLEASE WRITE IN #263      (ASCII: 'n')
PLEASE WRITE IN #261      (ASCII: 'a')
PLEASE WRITE IN #264      (ASCII: 'c')
PLEASE WRITE IN #269      (ASCII: 'c')
PLEASE WRITE IN #269      (ASCII: 'i')
PLEASE WRITE IN #260      (ASCII: ' ')
PLEASE WRITE IN #267      (ASCII: 'S')
PLEASE WRITE IN #264      (ASCII: 'e')
PLEASE WRITE IN #265      (ASCII: 'q')
PLEASE WRITE IN #265      (ASCII: 'u')
PLEASE WRITE IN #260      (ASCII: 'e')
PLEASE WRITE IN #263      (ASCII: 'n')
```

```
PLEASE WRITE IN #264       (ASCII: 'c')
PLEASE WRITE IN #265       (ASCII: 'e')
PLEASE WRITE IN #260       (ASCII: ' ')
PLEASE WRITE IN #259       (ASCII: ':')

(START LOOPING TO GENERATE TERMS)
(Use simulated loop with COME FROM pattern)

DO .3 <- #0              (Store temporary addition result)
DO .4 <- :2              (Load Fib(n-2))
DO .5 <- :3              (Load Fib(n-1))

DO :5 <- .4              (Copy for math)
DO :6 <- .5              (Copy for math)

PLEASE STASH .5          (Save one operand for later)

(ADD :5 + :6 INTO :3)
DO :3 <- :5              (Start with Fib(n-2))
PLEASE RETRIEVE .5       (Get back Fib(n-1))
DO :3 <- :3 + .5         (Sum into :3 — this is Fib(n))

(OUTPUT VALUE OF FIB)
DO .2 <- :3              (Prepare for output)
PLEASE WRITE OUT .2

(ROTATE VARIABLES: :2 <- :3 (old n), :3 <- :3 (new n+1))
DO :2 <- :3

(SIMULATE INCREMENTING INDEX)
DO :1 <- :1 + #1

(CHECK IF WE ARE DONE)
DO .4 <- :1
DO .5 <- :4
DO .6 <- .5 - .4
DO .7 <- .6 ~ #0         (Bitwise nonsense for branching)

(WEIRD CONTROL STRUCTURE USING ‘COME FROM’)
DO .8 <- #1
```

```
PLEASE FORGET #1
COME FROM .8

DO .9 <- .6
DO .10 <- #0
DO .11 <- .9 - .10

(ARE WE DONE?)
DO .12 <- .11 ~ #0
DO .13 <- #1
PLEASE FORGET #1
COME FROM .13

PLEASE GIVE UP          (Jump to this to continue loop)

(END OF LOOP, PROGRAM TERMINATES)
DO .0 <- #0
PLEASE GIVE UP
```

Technical Analysis

1. Language Esotericism

- This code abuses INTERCAL's deliberately unhelpful structure. COME FROM, not GOTO, creates inverted control flow.
- Uses STASH and RETRIEVE for artificial memory complexity.
- Uses multiple obfuscated math operations and bitwise XOR via ~.

2. Memory Model

- Registers like :1 to :6 simulate working memory.
- Scalar memory (e.g., .2) used for IO.

3. I/O Control

- Output is written character-by-character with PLEASE WRITE IN.
- Numbers are output using PLEASE WRITE OUT, rendering the Fibonacci sequence directly.

4. Branching Logic

- INTERCAL lacks loops; we simulate via:
    - COME FROM
    - GIVE UP (acts like an unnatural jump)
    - Bitwise obfuscation using ~

5. Obfuscation as Skill

- Demonstrates that the author deeply understands how execution is *not* supposed to work.
- Recruiters can infer strong theoretical background and systems-level thinking.