## **Sintaxis de SMath Studio para Generación de Código**

Necesito un script para una función personalizada de SMath Studio, compatible con una utilidad que transforma texto en funciones. El código generado debe seguir rigurosamente estas reglas de sintaxis:

### **I. Estructura Básica de la Función**

1. **Inicio de la Función (Code >>):**
   * La línea que define la firma de la función (su nombre y argumentos) debe comenzar con **Code >>** .
   * **No** debe terminar con dos puntos (:).
   * Ejemplo: Code >> MiFuncion(arg1, arg2)
2. **Sangría de Primer Nivel (Importante):**
   * **Todas las líneas de código que están directamente anidadas bajo la declaración Code >> MiFuncion(...) (es decir, el primer nivel del cuerpo de la función) deben iniciar en la primera columna (sin sangría inicial).**
   * Las sangrías se utilizan *internamente* a los bloques de control de flujo (como dentro de un for, if, while, etc.).
   * Ejemplo Correcto:  
     Code >> MiFuncion(x, y)  
     variable1 : valor\_inicial  
     if x > 0  
      resultado : "positivo" // Esta línea sí tiene sangría  
     resultado\_final : variable1 + 1 // Esta línea vuelve a la primera columna
3. **Valor de Retorno Implícito:**
   * El valor de la **última asignación** en el script se considera el valor de retorno de la función. No es necesario repetir la variable en la última línea si ya fue asignada justo antes.

### **II. Control de Flujo (Bloques sin end)**

1. **Asignaciones Múltiples con do:**
   * Para ejecutar múltiples asignaciones o expresiones en una sola línea lógica, usa **do** seguido de las asignaciones separadas por comas.
   * El bloque que sigue a do (si son múltiples líneas) debe estar sangrado.
   * La palabra clave **do** no va seguida de paréntesis, pero la expresión a evaluar sí puede tenerlos.
   * Ejemplo: do A:M , m:rows(A) , n:cols(A)
2. **Bucles (for, while):**
   * **for k, values** (bucle que itera k sobre una colección de values, como un rango).
   * **for k:k\_initial, condition\_over\_k, k:k\_increment\_expression** (bucle flexible con inicialización, condición y expresión de incremento explícitas).
     + *Nota:* k es la variable de iteración; k\_initial es su valor inicial; condition\_over\_k es una expresión booleana que debe ser verdadera para que el bucle continúe; k\_increment\_expression es la expresión que actualiza k en cada iteración.
   * **while condicion**.
   * El cuerpo de los bucles debe estar sangrado.
   * **NO usan end**.
   * Usa variables de contador como k, j, idx (evitar i por unidad imaginaria).
   * Las palabras clave **for** y **while** no van seguidas de paréntesis, pero las expresiones de la condición o rango sí pueden usarlos.
3. **Sentencias Condicionales (if, elif, else):**
   * Sintaxis: **if condicion**, **elif condicion**, **else**.
   * La palabra clave **if** o **elif** no va seguida de paréntesis, pero la expresión de la condición sí puede usarlos.
   * Los bloques de código deben estar sangrados.
   * **NO usan end**.
   * Ejemplo:  
     if (x ≥ 0) & (y < 10)  
      resultado : "positivo o cero y menor que diez"  
     else  
      resultado : "otra categoría"
4. **Manejo de Errores con try:**
   * Sintaxis: **try expresion\_principal ¦ expresion\_si\_error**.
   * **No usa catch ni end**.
   * La palabra clave **try** no va seguida de paréntesis, pero la expresión a intentar sí puede usarlos.
   * Ejemplo: try (Σ : stack(Σ, substr(S, n, k - n))) ¦ (error("Error al procesar")) // Los paréntesis en las expresiones son opcionales pero aclaran.
5. **break:**
   * Para salir de un bucle prematuramente, usa la palabra clave **break**.

### **III. Operadores y Sintaxis Específica**

1. **Asignaciones:**
   * Usa **:** (por ejemplo, miVariable : miExpresion).
2. **Operadores de Comparación:**
   * **≤** (menor o igual que)
   * **≥** (mayor o igual que)
   * **≡** (es igual a)
   * **≠** (es diferente de)
   * También se usan > (mayor que) y < (menor que).
3. **Operadores Lógicos:**
   * **|** (OR)
   * **&** (AND)
   * **¬** (NOT)
4. **Variables con Subíndices (y Nombres "Locales"):**
   * Las variables pueden llevar un subíndice (ej. xₒ, Mₒ). Cuando se solicite una variable que tenga un comportamiento "local" o para evitar conflictos de nombres en un contexto simbólico, el carácter **ₒ** puede usarse como marcador para ser reemplazado por el nombre de la función contenedora. Por ejemplo, xₒ se usaría para indicar una variable que se "volverá" x\_NombreDeLaFuncion.
5. **Concatenación de Líneas:**
   * Para continuar una línea de código en la siguiente, usa **...** al final de la línea.
6. **eval():**
   * Para forzar la evaluación de una expresión, úsalo como eval(expresion).
7. **¦ (Sobrecarga/Redefinición en Línea):**
   * Se utiliza para sobrecargar funciones o redefinir expresiones en la misma línea, principalmente para lograr eficiencia visual o para definiciones de biblioteca/constantes. No es para valores por defecto en el sentido de argumentos de función ni para continuación de líneas.
   * Ejemplo: Code >> MiFuncion(arg) ¦ MiFuncion(arg, nuevo\_comportamiento)
   * Ejemplo: 'TOL() ¦ 10^(-9) (para definir constantes globales de librería).

### **IV. Manipulación de Datos (Matrices, Cadenas, Tipos)**

1. **Indexación:**
   * **Las matrices y vectores en SMath se indexan a partir de 1.**
2. **Acceso a Elementos de Matrices/Vectores:**
   * Usa **el(Matriz, fila, columna)** o **el(Vector, índice)**.
3. **Construcción de Matrices/Vectores:**
   * **mat(elemento1, ..., filas, columnas)** para crear una matriz/vector.
   * **augment(elemento1, elemento2, ...)** para concatenar horizontalmente (filas).
   * **stack(elemento1, elemento2, ...)** para apilar verticalmente (columnas/filas).
   * Para inicializar matrices/vectores vacíos para acumulación: variable : 0 o variable : eval(matrix(filas, columnas)).
4. **Rangos Numéricos:**
   * Usa **range(start, end)** o range(start, step, end).
5. **Conversiones y Manipulaciones de Cadenas:**
   * num2str(numero): Número a cadena.
   * str2num(cadena): Convierte una cadena a un número o expresión evaluable. Para obtener un *handle* de función o invocar una función por su nombre de cadena, se puede construir la llamada dinámicamente.
     + Ejemplo para pasar un nombre de función y su argumento: resultado : str2num(concat(nombre\_funcion\_str, "(", num2str(argumento), ")"))
   * strrep(cadena, buscar, reemplazar): Reemplaza subcadenas.
   * concat(cadena1, cadena2, ...): Concatena cadenas.
   * substr(cadena, inicio, longitud): Extrae subcadena.
   * strlen(cadena): Longitud de cadena.
   * strsplit(cadena, delimitador): Divide cadena en vector de cadenas.
6. **Definición de Función Anidada/Temporal (con let):**
   * Sintaxis: **let nombre\_funcion\_local(args) : cuerpo\_funcion\_local**.
   * Ejemplo: let f\_temp(x) : x^2 + 1

### **V. Utilidades y Miscelánea**

1. **Comentarios:**
   * Usa **//** para comentarios de una sola línea.
2. **Manejo de Unidades:**
   * Usa **UnitsOf(expresion)** para obtener la unidad de medida.
3. **Lanzar Errores:**
   * Usa **error("mensaje")**.
4. **Verificación de Definición:**
   * Usa **IsDefined(variable)**.