≡ Novedades en SL ≡

Este es un resumen de las extensiones y mejoras que se han incorporado en SL recientemente. Estas adiciones simplifican y facilitan la escritura de los programas y brindan nuevas funcionalidades, las que benefician tanto al docente como al estudiante de fundamentos de programación.

Lo que sigue es una lista parcial de las novedades:

- Las variables pueden ser inicializadas al tiempo que se las declara.
- En la mayoría de los casos, el tipo de las variables puede ser omitido en su declaración si se asigna un valor inicial.
- Es posible salir anticipadamente de los ciclos con la sentencia "salir".
- Los registros y arreglos pueden ser leídos e impresos con una sola llamada a "leer" o "imprimir". Los valores lógicos (booleanos) también pueden ser leídos, además de impresos.
- El intento de leer más datos cuando ya no están disponibles porque se llegó al final del archivo no genera un error de ejecución; simplemente en tal caso basta con verificar el valor de retorno de la función eof().
- Es posible recuperar un carácter a la vez por cada llamada a leer().
- El cursor puede ser posicionado en cualquier parte de la pantalla antes de realizar una lectura o impresión.
- Se puede cambiar el color de fondo y de primer plano de los caracteres impresos en la pantalla.
- Se agregó una función que emite un pitido con una frecuencia y duración dadas.
- Es posible averiguar la cantidad de columnas y de líneas de la pantalla.
- La palabra "sub" se considera sinónimo de "subrutina".
- Los números pueden llevar un guión bajo entre sus dígitos, para facilitar la lectura.
- Ya no es obligatorio el uso de paréntesis en la expresión que sigue a la sentencia "retorna".
- La ejecución del programa SL se puede terminar con la subrutina "terminar()"
- Los operadores lógicos && (and) y || (or) implementan evaluación en modo "corto-circuito", lo que puede facilitar la escritura de ciertos ciclos.
- Se agregó una función para ejecutar programas externos.
- Se agregaron las funciones min() y max().

A continuación se explica con más detalles algunas de estas novedades.

1. SIMPLIFICACIÓN DE LA DECLARACIÓN DE VARIABLES

La nueva sintaxis para inicializar las variables al declararlas puede utilizarse cuando se conoce de antemano el valor inicial, o simplemente como una manera más compacta de escribir el programa.

Nótese que el compilador deduce el tipo de las variables del tipo de la expresión que se utiliza para inicializarlas. Es decir, si el inicializador es "Buen día" por ejemplo, la variable será del tipo "cadena". Una vez identificado el tipo de dato, el compilador de SL sigue realizando todos los chequeos de compatibilidad habituales.

Ejemplo 1: Imprimir los primeros 10 términos de la serie de Fibonacci de orden 2.

```
1.  var
2.     a        = 0
3.     b        = 1
4.     num_term = 2
5.
6.     /*
7.      * Como el valor de la variable c es cambiado por cada iteración,
8.      * no vale la pena asignarle un valor al declararla, aunque
9.      * no hay problemas en hacerlo.
10.     */
11.    c        : numerico
12. inicio
13.    imprimir (a, ' ', b)
14.    mientras ( num_term < 10 ) {
15.       c = a + b
16.       imprimir (' ', c)
17.       a = b
18.       b = c
19.       inc (num_term)
20.    }
21. fin
```

El inicializador puede ser un valor simple, como en el ejemplo previo, o podría ser una expresión más compleja que involucre a otras variables previamente declaradas o funciones. Por ejemplo:

```
1.  /*
2.   * Observacion: Este no es un programa SL completo. Solo muestra
3.   *              cómo el inicializador puede ser una expresión compleja.
4.   */
5.
6.  sub mostar_simplif (a, b : numerico) retorna cadena
7.  var
8.     m   = mcd (a, b)
9.     res = ""
10. inicio
11.    res = str(a/m, 0, 0) + "/" str(b/m, 0, 0)
12.    ...
13. fin
14.
15.
16. sub mcd (a, b : numerico) retorna numerico
17. /*
18.  * Calcula el MCD de dos numeros, que deben ser enteros positivos.
19.  */
20. inicio
21.    mientras ( a <> b ) {
22.       si ( a > b ) {
23.          a = a % b
24.       sino
25.          b = b % a
26.       }
27.    }
28.    retorna a
29. fin
```

Ejemplo 2: Mediante una función, asociar un entero (1-7) al nombre de un día de la semana que se pasa como parámetro.

```
1.  sub identificar_dia (dia : cadena) retorna numerico
2.  var
3.     nom_dias : vector [*] cadena = {"dom", "lun", "mar", "mie",
4.                                     "jue", "vie", "sab"
5.                                    }
6.     /*
7.      * num_dia contendrá el numero de dia (1-7). Retorna 0 si parámetro no
8.      * corresponde a un día de la semana.
9.      */
10.    num_dia = 0
11.    k       = 0
12. inicio
13.    /*
14.     * Se consideran solo los primeros 3 caracteres del parámetro dia.
15.     * Para facilitar la comparación, el nombre del día se pasa a minúsculas,
16.     * pues nom_dias contiene caracteres en minúsculas.
17.     */
18.    dia = lower(substr (dia, 1, 3))
19.    desde k=1 hasta alen(nom_dias) {
20.       si ( nom_dias [k] == dia ) {
21.          num_dia = k
22.          salir
23.       }
24.    }
25.    retorna num_dia
26. fin
```

En este ejemplo, es necesario especificar el tipo de `nom_dias` antes de asignar los valores porque de lo contrario el compilador no podría completar algunas verificaciones, entre ellas que todos los elementos del arreglo sean del mismo tipo.

2. SALIDAS ANTICIPADAS DE LOS CICLOS

Usando `salir`, que es una palabra reservada a partir de ahora, se puede terminar la ejecución de un ciclo (`mientras, repetir..hasta y desde`) sin que se haya cumplido la condición de salida.

Ejemplo: Leer una serie de números positivos y almacenarlos en un vector.

```
1.  sub leer_num_positivos (cant : numerico;
2.                          ref A : vector [*] numerico) retorna numerico
3.  /*
4.   * leer_num_positivos() lee sucesivamente una serie de
5.   * numeros y los deposita en un vector.
6.   *
7.   * Para de leer cuando:
8.   * a. Ya se leyeron cant numeros, o
9.   * b. Se llenaron las posiciones del vector, o
10.  * c. Se ingresa un numero negativo, el cual NO se almacena
11.  *    en el vector.
12.  *
13.  * RETORNA
14.  *    La cantidad de numeros leidos y almacenados en el vector.
15.  */
16. var
17.    k = 0
18.    cant_leidos = 0
```

```
19.  inicio
20.     desde k=1 hasta min (cant, alen (A)) {
21.        imprimir ("  Ingrese valor [", k, "]: ")
22.        leer (A [k])
23.        /*
24.         * Salir si el numero leido es negativo.
25.         */
26.        si ( A [k] < 0 ) {
27.           salir
28.        }
29.        inc (cant_leidos)
30.     }
31.     retorna cant_leidos
32.  fin
```

El uso de `salir` podría simplificar el código en ciertos casos, especialmente cuando se usan variables lógicas adicionales solo para controlar la salida anticipada de ciclos. Sin embargo, se recomienda utilizarlo con mesura, pues los programas pueden ser difíciles de comprender a causa de su uso excesivo.

3. LECTURA E IMPRESIÓN DE DATOS ESTRUCTURADOS

Para acortar la escritura de programas, se dotó a `leer()` e `imprimir()` de la capacidad de operar sobre datos estructurados, como ser arreglos y registros. Así, con una sola llamada a una de estas subrutinas es posible aceptar o mostrar varios datos.

Por ejemplo, si se está procesando un archivo cuyos registros contienen datos de un alumno (nombre, fecha de nacimiento, carrera y curso actual), una variación de lo que se muestra a continuación puede usarse para recuperar cada registro con un solo `leer()`.

```
 1.  /*
 2.   * Observacion: Este no es un programa SL válido. Solo muestra las
 3.   *              nuevas características consideradas.
 4.   */
 5.  var
 6.     reg : registro {
 7.              nombre       : cadena
 8.              fecha_nac    : numerico      // AAAAMMDD
 9.              cod_carrera  : cadena
10.              curso_actual : numerico
11.              notas_semestre : vector [5] numerico
12.           }
13.  inicio
14.     leer (reg)
15.     mientras ( not eof() ) {
16.        ...
17.        ...
18.        leer (reg)
19.     }
20.  fin
```

Cada operación de `leer()` esperará a que se ingresen en total 9 datos (4 + 5 notas), como si hubiesen sido leídos cada uno independientemente. Cada valor debe estar separado del siguiente por el carácter coma, o por lo que se haya configurado vía la subrutina predefinida `set_ifs()`.

Por otra parte, puede usarse `imprimir()` para desplegar todo el registro, o un arreglo,

independientemente de la cantidad de dimensiones. Suponga que desea desplegar las notas de un alumno en particular, donde cada nota se muestre en una línea independiente. Para ello puede utilizar el siguiente código:

```
1.  /*
2.   * Observación: Código incompleto; solo muestra el uso de set_ofs() e
3.   *              imprimir() con datos estructurados.
4.   */
5.     ...
6.     set_ofs ("\n")
7.     imprimir (reg.notas_semestre)
8.     ...
```

Aquí puede verse el uso de la nueva función `set_ofs()`, que sirve para indicar qué carácter utilizará `imprimir()` para separar los elementos cuando se le pasa para imprimir un arreglo completo o un registro. Vea más adelante detalles adicionales sobre `set_ofs()` que pueden resultarle de utilidad.

Como parte de esta ampliación, SL tiene actualmente la capacidad de leer datos del tipo `logico`. Se considera que "SI" y "TRUE" (tipeados en mayúsculas, en minúsculas o en cualquier combinación) corresponden al valor lógico "verdadero", mientras que "NO" y "FALSE", igualmente en cualquier combinación de mayúsculas/minúsculas, corresponden a "falso".

4. CONTROL SOBRE EL CURSOR, EL COLOR Y OTROS

Fueron agregadas varias funciones para controlar el color con el que se mostrarán los datos impresos, así como para cambiar la posición del cursor, para la lectura de teclas con un tiempo máximo de espera, la emisión de pitidos, entre otros. En su conjunto, estas nuevas funciones posibilitan escribir programas más vistosos e interactivos.

Estas adiciones tienen como objetivo dar soporte a aquellos docentes que deseen despertar el interés de sus alumnos sobre la programación con ejercicios más interactivos, los que a veces son presentados en las primeras etapas del estudio de fundamentos de programación.

Como ya es habitual en SL, estas rutinas fueron cuidadosamente diseñadas para que no tengan ningún efecto cuando la salida (o la entrada) es archivo y no la consola (pantalla y teclado), ya que no tiene sentido "cambiar el color del texto", por ejemplo, cuando `imprimir()` está grabando en un archivo, en vez mostrar datos en la pantalla.

Lo que sigue es un resumen de los cambios y adiciones, dando en cada caso el formato o prototipo del subprograma correspondiente.

1. Cambiar la posición del cursor

```
sub set_curpos (nueva_lin, nueva_col : numerico)
```

Posiciona el cursor en la línea y columna especificadas. La esquina superior izquierda de la pantalla corresponde a la posición (1, 1), es decir, línea 1 columna 1. Si solo desea

cambiarse la columna, conservando la fila actual, puede pasarse columna 0; similar efecto se logra pasando 0 como número de línea: solo se cambia la columna.

2. Obtener la posición del cursor

```
sub get_curpos (ref lin_actual, col_actual : numerico)
```

Cuenta la línea y columna donde se encuentra actualmente el cursor. Nótese que los dos parámetros que necesita esta rutina son pasados por referencia, es decir, deben ser nombres de variables numéricas.

3. Cambiar el color del texto impreso

```
sub set_color (primer_plano, fondo : numerico)
```

El segundo parámetro de set_color() indica el color de fondo que se desea, que va de 1 a 8, donde 1 generalmente es negro. El primer parámetro (primer_plano) indica el color del texto mismo, y puede ir de 1 a 16. Lo que sigue es un ejemplo:

```
1.  inicio
2.      set_color (1, 11)
3.      imprimir ("Este texto debe verse en color verde sobre negro")
4.  fin
```

Si cualquiera de los parámetros es 0, el color correspondiente no se ve afectado.

Por una decisión de diseño, los colores no tienen nombres o significados predefinidos, ya que al hacerlo se está limitando la posibilidad de portar SL a nuevas plataformas.

4. Obtener el color del texto impreso

```
sub get_color (ref primer_plano, fondo : numerico)
```

Obtiene el color del texto y del fondo vigentes. Nótese que ambos parámetros son pasados por referencia, por lo que se deben proveer el nombre de dos variables.

5. Cambiar el color de fondo de toda la pantalla

Este efecto se obtiene llamando primero a `set_color()`, seguido de `cls()`.

6. Obtener las dimensiones de la pantalla, en líneas y columnas.

```
sub get_scrsize (ref cant_lin, cant_cols : numerico)
```

Obtiene la cantidad de líneas y columnas que tiene la pantalla. Nótese que ambos parámetros son pasados por referencia, por lo que se deben proveer el nombre de dos variables. El tamaño habitual de la pantalla es 25 líneas por 80 columnas.

7. **Esperar a que se pulse una tecla**

```
sub readkey (milisegundos : numerico) retorna numerico
```

Espera a que se pulse una tecla y retorna un código numérico que identifica lo pulsado por el usuario. El tiempo que se espera depende del parámetro, que está en milisegundos.

Si se omite el parámetro, `readkey()` espera indefinidamente a que el usuario pulse una tecla. De contrario esperará el tiempo que se le indique (en milisegundos) y si nada se pulsó en ese tiempo, retornará 0.

Por una decisión de diseño y portabilidad de SL, no se asigna ningún significado específico al valor retornado por esta función, aunque el usuario es libre de experimentar y observar qué valores se generan para cada tecla.

8. **Emitir pitido**

```
sub beep (frecuencia, duracion_millis : numerico)
```

Emite un pitido a cierta frecuencia y durante el tiempo que se indica en el segundo parámetro (expresado en milisegundos).

Ambos parámetros pueden omitirse, en cuyo caso se utiliza automáticamente los valores (500, 100), es decir, se emite un pitido durante una décima de segundo.

Nótese que `beep()` puede ser utilizado para introducir una pausa en la ejecución, sin emitir pitido propiamente. Por ejemplo `beep(0, 5000)` hará una pausa de medio segundo.

5. EJEMPLO DE USO DE LA ENTRADA/SALIDA POR PANTALLA MEJORADA

El siguiente programa muestra el uso de las nuevas funciones de entrada/salida por pantalla. Despliega un texto deslizante, mientras muestra la hora actual. Termina cuando el usuario pulsa alguna tecla.

OBSERVACIÓN: Si quiere ver este programa funcionando, puede encontrar el programa compilado (ejecutable de consola de Win32) en **ftp://ftp.cnc.una.py/pub/slc/test_reloj.exe**

```
1.  /*
2.   * test_reloj.sl
3.   * (c) 2003, Juan Segovia
4.   *
5.   * Muestra el uso de algunas rutinas para cambiar color,
6.   * posicionar el cursor y leer directamente del teclado.
7.   *
8.   */
9.
10. const
11.    COLOR_blanco_fuerte = 16
12.    COLOR_azul          = 2
13.    COLOR_blanco        = 8
```

```
14.     COLOR_verde         = 11
15.     COLOR_negro         = 1
16.  inicio
17.     /*
18.      * Cambiar el color de fondo, que consiste en cambiar los colores y
19.      * luego limpiar la pantalla.
20.      */
21.     set_color (COLOR_blanco_fuerte, COLOR_azul)
22.     cls()
23.
24.     set_color (COLOR_blanco_fuerte, COLOR_blanco)
25.     set_curpos (8, 20)
26.     imprimir (" Muestra el uso de colores y readkey() en SL ")
27.     demo()
28.  fin
29.
30.
31.  sub demo()
32.  var
33.     marq_msg           = "Pulse alguna tecla para terminar...."
34.     marq_ancho_ventana = 80
35.     marq_col_actual    = marq_ancho_ventana
36.  inicio
37.     repetir
38.        desplegar_marquesina (marq_msg, marq_ancho_ventana, marq_col_actual)
39.        desplegar_reloj()
40.
41.        /*
42.         * Salir si se pulso alguna tecla.
43.         */
44.     hasta (readkey(100) > 0)
45.  fin
46.
47.
48.  sub desplegar_reloj()
49.  /*
50.   * Muestra la hora actual en formato HH:MM:SS
51.   *
52.   */
53.  var
54.     hh, mm, ss : numerico
55.     tiempo     : numerico
56.  inicio
57.     tiempo = sec()
58.     hh = int(tiempo/ 3600)
59.     dec (tiempo, hh*3600)
60.     mm = int (tiempo / 60)
61.     ss = tiempo % 60
62.
63.     set_curpos (10, 33)
64.
65.     set_color (COLOR_blanco_fuerte, COLOR_azul)
66.     imprimir ( str (hh, 2, 0, '0'), ':',
67.                str (mm, 2, 0, '0'), ':',
68.                str (ss, 2, 0, '0')
69.              )
70.  fin
71.
72.
73.  sub desplegar_marquesina (msg : cadena; ancho_vent : numerico
74.                            ref col_actual : numerico)
75.  /*
76.   * Muestra un texto en formato "marquesina", es decir, el texto
77.   * va apareciendo sucesivamente en la pantalla desde el borde
78.   * derecho y desaparece luego por la izquierda, repitiendose
79.   * sucesivamente esta secuencia.
80.   *
81.   * El parametro col_actual es fundamental que sea pasado por
```

```
 82.   * referencia.
 83.   *
 84.   */
 85.
 86.  var
 87.     parte_msg : cadena
 88.     len_msg   = strlen (msg)
 89.
 90.  const
 91.     LINEA_marq = 11
 92.
 93.  inicio
 94.     si ( col_actual > 0 ) {
 95.        set_curpos (LINEA_marq, col_actual)
 96.        parte_msg = substr (msg, 1, ancho_vent-col_actual+1)
 97.     sino
 98.        set_curpos (LINEA_marq, 1)
 99.        si ( col_actual == 0 ) {
100.           col_actual = -2
101.        }
102.        parte_msg = substr (msg, -col_actual)
103.     }
104.
105.     set_color (COLOR_verde, COLOR_negro)
106.     imprimir (parte_msg)
107.
108.     /*
109.      * Como se ha corrido el texto un lugar a la izquierda, debemos
110.      * tachar, con el color de fondo, el ultimo caracter del
111.      * despliegue previo.
112.      */
113.     set_color (COLOR_blanco_fuerte, COLOR_azul)
114.     imprimir (' ')
115.
116.     /*
117.      * Cuando se hayan desplegado TODOS los caracteres, preparar
118.      * para que el mensaje reaparezca por la derecha.
119.      */
120.     si ( col_actual < -len_msg ) {
121.        col_actual = ancho_vent+1
122.     }
123.     dec (col_actual)
124.  fin
```

6. NUEVOS OPERADORES LÓGICOS

Existen dos nuevos operadores lógicos, simbolizados por "`&&`" y "`||`". Estos operadores son una variante de los operadores `and` y `or` respectivamente. La diferencia es que estos nuevos operadores realizan la evaluación en modo "corto-circuito", como se lo conoce técnicamente. En este modo, una expresión puede ser evaluada parcialmente, hasta que se conozca el valor final de la expresión.

Por ejemplo, si el lado izquierdo de una expresión que involucra al operador `and` es falso, no tiene sentido evaluar el lado derecho, pues ya se sabe que la expresión completa será falsa. Al disponer de los operadores `&&` y `||` es posible escribir código más compacto.

Ejemplo: Determinar la posición de la primera aparición de un valor en un arreglo.

```
 1.  sub localizar_valor (v : numerico; A : vector [*] numerico) retorna numerico
 2.  /*
 3.   * Retorna la posición de la primera aparición de v en A. Si v no existe
 4.   * en A, se retorna 0.
 5.   */
 6.  var
 7.     g    = alen(A)
 8.     p    = 0
 9.  inicio
10.     p = 1
11.     mientras ( (p <= g) && (A [p] <> v) ) {
12.        inc (p)
13.     }
14.     /*
15.      * No se encontró v en A.
16.      */
17.     si ( p > g ) {
18.        p = 0
19.     }
20.     retorna p
21.  fin
```

Si en la rutina localizar_valor() usásemos el operador `and` en vez de `&&`, se produciría un error de ejecución cuando v no se encuentre en A, pues `A[p]` accedería a una posición inexistente (uno más que la última posición existente en A).

Cuando las expresiones son anidadas (parentizadas, por ejemplo), los operadores `&&` y `||` realizan la evaluación en "corto-circuito" de la subexpresión en la que están inmersos, no de toda la expresión.

Como consecuencia de la introducción de `&&` y `||`, los símbolos `&` y `|` fueron retirados de SL, que anteriormente eran sinónimos de `and` y `or` respectivamente.

En general, el uso de `&&` y `||` producirá programas que se ejecuten ligeramente más rápidos que los que se obtienen usando `and` y `or`.

7. MEJORAS COMPLEMENTARIAS PARA LA ENTRADA/SALIDA

Se han introducido ligeros cambios en el comportamiento de algunas subrutinas incorporadas para mejorar y facilitar la entrada/salida. A continuación los detalles:

1. Posibilidad de leer un archivo carácter por carácter

Es posible leer un archivo carácter a carácter si se hace `set_ifs("")`, es decir, si el parámetro es una cadena vacía. En este caso el carácter separador de línea se obtiene como un carácter más.

Esta funcionalidad puede ser útil, por ejemplo, para escribir un analizador lexicográfico , que es un componente clásico de un compilador.

Por otra parte, `set_ifs()` debe ser llamado luego de cada `set_stdin()`, pues se restaura a su valor original (**",**") al cambiar el archivo de entrada.

2. Las lecturas parciales no generan error

Si una operación de lectura no encuentra suficientes datos para todas las variables, no se genera un error de ejecución, a diferencia de versiones previas de SL. Simplemente se activa el indicador de fin de archivo, que puede averiguarse llamando a la función `eof()`. Este cambio posibilita que el siguiente fragmento de código funcione como se espera:

```
1.  /*
2.   * Cuenta la cantidad de líneas que existen en el archivo poesia.txt
3.   *
4.   * Observacion: No verifica si el archivo poesia.txt existe o si pudo
5.   *              abrirse.
6.   */
7.  var
8.     linea       = ""
9.     cant_lineas = 0
10. inicio
11.    set_stdin ("poesia.txt")
12.    set_ifs ("\n")
13.    leer (linea)
14.    mientras ( not eof() ) {
15.       inc (cant_lineas)
16.       leer (linea)
17.    }
18.    imprimir ("\nFueron leídas ", cant_lineas, " líneas\n")
19. fin
```

3. Especificador de separador de valores de datos estructurados

```
sub set_ofs (s : cadena)
```

La rutina `set_ofs()` sirve para configurar qué carácter utilizará `imprimir()` para separar los valores de los datos compuestos, como los registros y arreglos. Por defecto este carácter es una coma.

Además del carácter separador de valores `set_ofs()` configura lo que se mostrará cuando imprimir() encuentre un arreglo que no está dimensionado. Por defecto se imprime "`<nodim>`".

El primer carácter del parámetro de `set_ofs()` es el separador y lo que sigue es el indicador de "no dimensionado", por lo que la configuración por defecto actúa como si se hubiese hecho `set_ofs("**,**<nodim>**"**)`.

Para obtener estos valores vigentes se usa `get_ofs()`, cuyo prototipo es:

```
sub get_ofs (s : cadena) retorna cadena
```

8. MAYOR LEGIBILIDAD DE LAS CONSTANTES LITERALES NUMÉRICAS

Para facilitar la legibilidad de los programas, los valores enteros que aparezcan literalmente en el programa fuente (es decir, constantes literales numéricas) pueden llevar un guión bajo como un dígito "mudo". Por ejemplo, cada una de las siguientes líneas contiene el número

un millón (si bien la segunda forma es la que resulta más simple de leer):

```
1000000
1_000_000
10_000_00
100_000_0
```

El guión bajo puede aparecer luego de un carácter, es decir, el literal no puede empezar con un guión. La parte fraccionaria y el exponente de la notación científica también pueden usar el guión bajo.

9. OTROS CAMBIOS MÁS PEQUEÑOS

1. sub como sinónimo de subrutina

Dado que la palabra "subrutina" es usada a menudo, se introdujo la abreviatura "sub". Esta nueva forma la hemos utilizado a lo largo de los ejemplos de este documento.

2. Sintaxis del valor de retorno de las funciones

En versiones previas de SL era necesario parentizar la expresión que definía el valor de retorno de las funciones; tal requerimiento sintáctico fue eliminado, si bien quienes lo prefieran pueden seguir parentizando las expresiones.

En SL las expresiones que indican o provocan ejecución condicional llevan paréntesis. Tal es el caso de `mientras, si, repetir..hasta, eval..caso`. Dado que `retorna` no un especificador de ejecución condicional, no debe ser obligatorio que lleve paréntesis.

3. Terminación anticipada de la ejecución del programa

Se agregó la subrutina `terminar()` que provoca la terminación de la ejecución del programa. Esto es útil cuando se detecta una condición que imposibilita seguir ejecutando el programa, por ejemplo, cuando no se pudo abrir el archivo de entrada indicado por el usuario.

El prototipo de `terminar()` es

```
sub termin (msg : cadena)
```

donde el parámetro msg especifica un mensaje que se imprimirá antes de que termine la ejecución. Este parámetro es opcional; si no se especifica nada, se asume "".

Ampliaremos el ejemplo previo del contador de líneas de texto de un archivo para mostrar el uso de `terminar()`.

```
1.  /*
2.   * Cuenta la cantidad de líneas que existen en el archivo poesia.txt
3.   * Verifica si el archivo poesia.txt existe o si pudo
4.   * abrirse.
5.   */
```

```
 6.
 7.  var
 8.     linea       = ""
 9.     cant_lineas = 0
10.  inicio
11.     /*
12.      * Verificar si el archivo pudo abrirse. En tal caso, set_stdin()
13.      * retorna verdadero.
14.      */
15.     si ( not set_stdin ("poesia.txt") ) {
16.        imprimir ("\nNo se pudo abrir el archivo poesia.txt\n",
17.                  "El programa no puede continuar.")
18.        terminar("\nEjecución terminada.\n")
19.        /*
20.         * Lo que sigue NUNCA se hará, pues la rutina termin() NUNCA
21.         * retorna.
22.         */
23.        linea = "--nunca llega aquí--"
24.     }
25.     set_ifs ("\n")
26.     leer (linea)
27.     mientras ( not eof() ) {
28.        inc (cant_lineas)
29.        leer (linea)
30.     }
31.     imprimir ("\nFueron leídas ", cant_lineas, " líneas\n")
32.  fin
```

Nótese que `terminar()` es sintácticamente una subrutina pero no se comporta como tal pues NUNCA retorna. En el ejemplo, la línea 23 jamás se ejecutará.

4. Ejecución de programas externos

Se agregó la función `runcmd()` que ejecuta un programa externo cualquiera. El prototipo de esta función es:

```
sub runcmd (cmd : cadena) retorna numerico
```

El siguiente fragmento de código muestra cómo se podría usar `runcmd()`.

```
1.  /*
2.   * Observacion: Este no es un programa SL completo.
3.   */
4.  inicio
5.     rumcmd ("miejec.exe " + " param1 param2")
6.     ...
7.     ...
8.  fin
```

El parámetro pasado a `runcmd()` incluye el nombre del comando y los argumentos que éste necesite. La ejecución del comando se realiza vía el procesador de comandos del usuario y las reglas de localización del comando externo son las de aquel.

El valor de retorno de `runcmd()` es el valor de salida del comando ejecutado. Pero si la ejecución del comando no puede realizarse, `runcmd()` retorna 127. (Nótese que este valor es arbitrario y bien podría ser que un comando se ejecute correctamente y retorne 127).

5. Las funciones min() y max()

La nueva función `min()` retorna el menor de dos valores que se le pasa como parámetro, mientras que `max()` retorna el mayor. Ambas funciones pueden operar sobre cualquier tipo básico (`cadena, numerico, logico`). Sin embargo, ambos parámetros deben coincidir en tipo, ya que no tiene sentido encontrar el mayor de una cadena y un número, por ejemplo.

El siguiente ejemplo muestra el uso de `max()` para identificar el mayor de 3 números.

```
1.  /*
2.   * Lee 3 números e imprime el mayor de ellos.
3.   */
4.  var
5.     a, b, c : numerico
6.  inicio
7.     imprimir ("\nIngrese tres valores numéricos: ")
8.     leer (a, b, c)
9.     si ( a == b && b == c ) {
10.       imprimir ("\nLos tres números ingresados son iguales")
11.    sino
12.       imprimir ("\nEl mayor es ", max (max (a, b), c), "\n")
13.    }
14. fin
```

10.DISPONIBILIDAD DE ESTAS MEJORAS

Las características presentadas en este documento están incorporadas en SLE (Versión Windows) de fecha 24-feb-2003 o posterior. Puede acceder al ítem “Acerca del compilador” en SLE para verificar qué versión está utilizando. Si no tiene la versión apropiada, puede descargarla de

ftp://ftp.cnc.una.py/pub/slc

Por otra parte, en breve encontrará en esta dirección versiones del entorno de desarrollo para Linux y para Windows, en ambiente gráfico.

AGRADECIMIENTO

El autor agradece a Blanca de Trevisan, Cristian Cappo, Cristina Paiva, Pablo Greenwood y Rolando Chaparro, quienes cooperaron en la revisión de este documento.