Introducción al lenguaje SL

Referencia de subrutinas predefinidas
Ejemplos selectos

Juan Segovia Silvero

Centro Nacional de Computación
Universidad Nacional de Asunción

Introducción al lenguaje SL
por Juan Segovia Silvero

Revisión: Blanca de Trevisan
Tapa: Jorge Meza B.

Contenido

Capítulo

1

Introducción

SL es un lenguaje diseñado para apoyar la formación profesional de estudiantes de informática, proveyendo un entorno que acompañe el proceso de construcción de algoritmos, desde los más sencillos hasta aquellos que requieren técnicas avanzadas de programación. La sintaxis del lenguaje, sus construcciones y demás características fueron cuidadosamente seleccionadas para que el alumno se concentre en la búsqueda de soluciones y obvie detalles específicos que seguramente tendrá ocasión de ver en otras etapas de su aprendizaje.

El entorno de programación incluye un editor multiventanas con posibilidades de compilación, ejecución y depuración de los programas, apuntando a facilitar la experimentación y el pensamiento creativo del alumno.

El lenguaje presenta características que lo hacen apropiado para expresar algoritmos de las etapas iniciales del aprendizaje, pero simultáneamente reúne un rico conjunto de construcciones que posibilitan el tratamiento de tópicos más avanzados de estructuras de datos y programación modular.

Principales características de SL

SL es un lenguaje muy sencillo de aprender y utilizar:

- Posee un conjunto simplificado de tipos de datos, pero posibilitando la definición de tipos agregados. En este aspecto SL considera los números enteros y reales, positivos y negativos, bajo un mismo tipo de dato: `numerico`. Existen además variables lógicas y cadenas.
- Las cadenas son dinámicas, es decir, su longitud se ajusta automáticamente para contener la secuencia de caracteres que se requiera, sin obligar a la definición explícita de una longitud máxima. Además están predefinidos los operadores relacionales, el operador de concatenación, la posibilidad de leerlos e imprimirlos y el acceso a cada carácter en forma individual y directa.
- Además de los tipos de datos básicos (numerico, cadena, logico) pueden definirse registros y arreglos n-dimensionales de cualquier tipo de dato.
- Los arreglos pueden tener tamaño inicial definido, o ser dinámicamente dimensionados.
- Las variables, tipos de datos y constantes pueden ser locales o globales.
- El chequeo de compatibilidad de tipos de datos es estricto, aunque la compatibilidad es estructural y no simplemente por nombres.
- Los subprogramas comprenden subrutinas y funciones, los que pueden recibir parámetros por valor o por referencia. Las funciones pueden retornar valores de cualquier tipo de datos, incluyendo arreglos y registros. Cada subprograma puede tener su propio conjunto de símbolos locales.
- Los identificadores (nombre de variables, de constantes, de subrutinas, etc.) pueden tener hasta 32 caracteres de longitud, pudiendo la letra ñ ser parte de un identificador.
- Los identificadores deben ser definidos antes de su uso, a excepción de los subprogramas, evitándose así la necesidad de definir prototipos.
- Se cuenta con un rico conjunto de estructuras de iteración y selección.

- En general, la sintaxis del lenguaje presenta similitudes a las de Pascal y C, favoreciendo la programación estructurada.
- El lenguaje es de formato libre como Pascal o C, pero a diferencia de éstos, no requiere ';' (punto y coma) como separador de sentencias.
- El despliegue y la lectura de datos es muy sencillo.

SL aspira ser un medio de expresión de algoritmos, no un fin en sí mismo.

Un sencillo programa en SL

A continuación se presenta un sencillo programa SL que calcula la suma de los números pares comprendidos entre 1 y n. El programa pide al usuario que tipee un valor para n.

```
/*
   Calcular e imprimir la suma de los números pares comprendidos entre 1 y
   un n dado.
   Las variables utilizadas son:
      n    : para almacenar el valor proveido por el usuario como tope.
      suma : donde almacenamos el total calculado.
      k    : contador auxiliar para el ciclo.
   AUTOR   : Juan Segovia (jsegovia@cnc.una.py)
*/
var
   n, suma, k : numerico

inicio
   imprimir ("\nSuma de numeros pares entre 1 y n.\nPor favor ",
             "ingrese un valor para n: ")
   leer (n)
   suma = 0
   desde k=2 hasta n paso 2
   {
      suma = suma + k
   }
   imprimir ("\nLa suma es ", suma)
fin
```

> *Los ejemplos de este manual requieren que la versión del compilador sera 0.6.2 o posterior. El entorno de desarrollo, que se presenta en el siguiente capítulo, tiene una opción para verificar la versión del compilador.*

Organización del libro

En este libro se describe el lenguaje SL: su estructura general, sus tipos datos básicos y agregados, las sentencias, las expresiones, las subrutinas, etc. Se da la sintaxis y las principales características desde el punto de vista de funcionamiento.
Puede ser utilizado como guía para el alumno que está aprendiendo SL. Se hace además una breve introducción al entorno de desarrollo.

A los efectos de dar una visión global del lenguaje, en el Anexo A se incluyen programas ejemplo escritos en SL.

En el Anexo B se describen también en detalle las rutinas utilizadas para la entrada/salida, funciones de conversión de tipos de datos, etc.

Capítulo

2

El entorno de desarrollo de SL

El entorno de desarrollo está diseñado para brindar al que se inicia en el estudio de algoritmos un ambiente integrado con el cual pueda desarrollar sus programas. El entorno posibilita:

- Preparar o modificar los programas fuentes, contando con funciones para cortar y pegar textos, realizar búsquedas y sustituciones, etc.
- Mantener simultáneamente varios programas fuentes tal que sea posible realizar comparaciones, copiados de textos, etc. Cada programa fuente se sitúa en una ventana independiente.
- Compilar los programas editados, recibiendo indicación acerca de la ubicación y naturaleza de los errores sintácticos o semánticos, si los hubiere.
- Ejecutar programas.
- Depurar programas, pudiendo establecer puntos de ruptura, ejecución paso a paso, por niveles, visualizar valores de variables y expresiones a medida que avanza la ejecución del programa, etc.
- Establecer de una forma muy sencilla los archivos para entrada y salida de datos.
- Guardar y restablecer el entorno de trabajo (ventanas abiertas, posición del cursor dentro de cada ventana, colores, parámetros de ejecución, etc.)

Visión general del entorno de desarrollo

Para utilizar el entorno de desarrollo, debe ubicarse en el directorio donde se encuentre instalado el compilador, si dicho directorio no está en el camino (PATH). Tipee **sle** y pulse la tecla ENTER. Si su sistema operativo es Windows95 o Windows98 puede crear un acceso directo en el escritorio o en el menú inicio. Deberá ver una pantalla similar a la que se muestra en la figura 1.

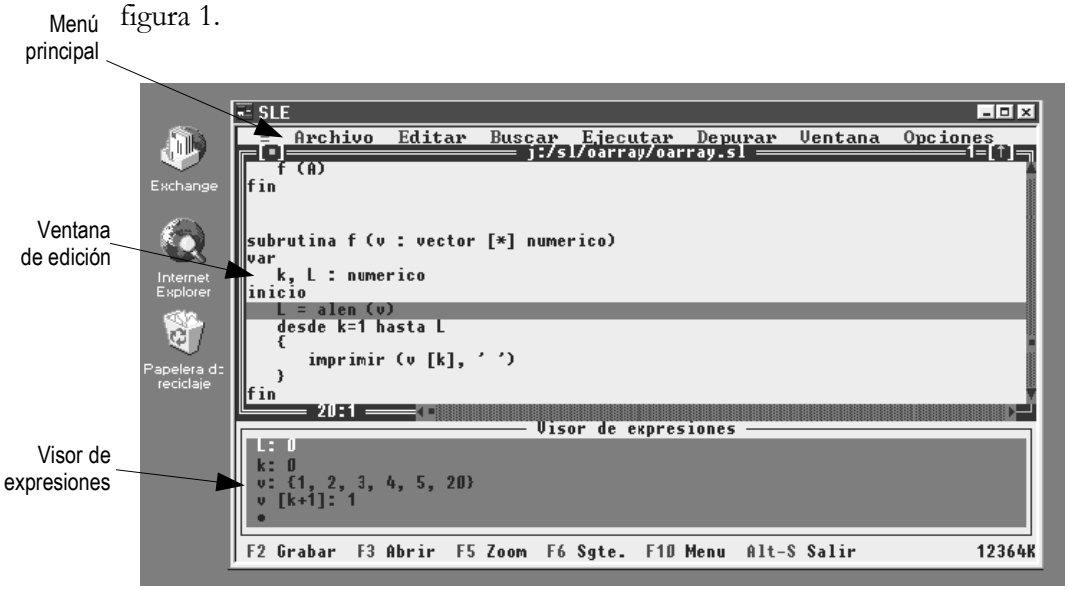


Figura 1. El ambiente de programación de SL (SLE)

Uso del entorno

Se utiliza para realizar acciones tales como compilar, ejecutar o establecer una opción del entorno. A este menú se accede pulsando ALT-F10. La barra del menú principal luce como sigue:

Figura 2. El menú principal de SLE

Un determinado ítem de esta barra puede ser seleccionado pulsando la tecla ALT en conjunción con la correspondiente letra resaltada. Por ejemplo, al ítem "Editar" se puede acceder pulsando ALT-E.

El primer ítem, señalado con , contiene una calculadora sencilla y una tabla ASCII, ambos de utilidad durante la edición y depuración de programas. Contiene además un subítem que despliega información sobre la versión del compilador, copyright y otros datos generales.

El entorno es muy sencillo de utilizar. Con unos pocos minutos de práctica y exploración podrá editar sus programas, compilarlos, ejecutarlos e incluso utilizar las opciones avanzadas de depuración.

Las opciones del menú de SLE

La siguiente tabla resume la función de cada opción del menú.

Tabla 1. Opciones del menú de SLE

Menú	Opción	Elija esta opción para
Archivo	**Nuevo**	Crear un nuevo archivo de programa fuente. Inicialmente el archivo no tiene nombre y debe proveer uno antes de grabarlo.
	Abrir…	Abrir un archivo de programa fuente ya existente. SLE le mostrará una ventana donde podrá elegir el archivo de una lista, o tipearlo directamente.
	Grabar	Grabar un archivo de programa fuente. Si éste aun no tiene nombre, SLE le pedirá que indique uno.
	Grabar como…	Crear un nuevo archivo de programa fuente en base al que se está editando. SLE le preguntará cómo debe llamarse este archivo.
	Cambiar directorio	Establecer en qué directorio se estará trabajando, es decir, dónde se buscarán los archivos de programas fuentes y dónde se grabarán los programas compilados.
	Ir al DOS	Acceder a la línea de comandos del sistema operativo, sin dejar completamente SLE. Al salir del procesador de comandos se volverá automáticamente a SLE.

Tabla 1. Opciones del menú de SLE (continuación)

Menú	Opción	Elija esta opción para
	Imprimir…	Imprimir el programa o archivo que se está editando.
	Salir	Abandonar SLE. Si algún programa fuente fue modificado, SLE le preguntará si desea grabarlo antes de salir.
Editar	**Deshacer**	Deshacer el último cambio realizado en el texto del programa fuente, por ejemplo borrar, cortar, pegar, etc. Solo podrá deshacer un cambio si permanece en la misma línea donde éste se efectuó.
	Cortar	Transferir al portapapeles la porción de texto que está marcada.
	Copiar	Insertar en el portapapeles una copia de la porción de texto que está marcada.
	Pegar	Insertar en el programa fuente que se está editando lo último que se haya copiado al portapapeles.
	Borrar	Eliminar la porción de texto que está marcada.
	Mostrar portapapeles	Activar la ventana del portapapeles.
Buscar	**Encontrar…**	Localizar un texto determinado dentro del programa fuente.
	Reemplazar…	Sustituir un texto determinado por otro en el programa fuente.
	Buscar de nuevo	Repetir la última operación de localización o sustitución.
Ejecutar	**Compilar**	Verificar que el programa está correcto, según las reglas de SL, y prepararlo para su ejecución posterior.
	Uso de memoria…	Cambiar el tamaño del área de código del área de datos. A menos que escriba programas de varios miles de líneas o que use miles de variables, no necesitará cambiar los valores por defecto establecidos por SLE.
	Ejecutar	Ejecutar el programa que se esté editando. El programa será primero compilado si SLE detecta que fue cambiado luego de la última compilación exitosa.
	Terminar	Hacer que termine la ejecución de un programa.
	Argumentos…	Establecer parámetros que luego el programa pueda recuperar al inicio de la ejecución, a través de la función paramval(). Encierre entre comillas los parámetros que consten de varias palabras.
Depurar	**Paso a paso**	Seguir línea a línea la ejecución de un programa.
	Por nivel	Seguir la ejecución de un programa. A diferencia de la opción "Paso a paso", las llamadas a subrutinas son ejecutadas en un solo paso, es decir, no se muestra la ejecución de las sentencias de las subrutinas.

Tabla 1. Opciones del menú de SLE (continuación)

Menú	Opción	Elija esta opción para
	Hasta línea actual	Ejecutar un programa hasta que la ejecución llegue a cierta línea en el programa fuente.
	Punto de parada	Establecer en qué líneas debe detenerse la ejecución de un programa.
	Evaluar expresión	Observar qué valor tiene una variable o una expresión.
	Agregar expresión	Incluir una expresión en "Visor de expresiones".
Ventana	**Tamaño/Mover**	Mover o cambiar de tamaño un ventana, usando las teclas del cursor. Pulse ENTER cuando haya terminado de ubicar o cambiar de tamaño la ventana.
	Zoom	Activar/desactivar el modo de pantalla completa de una ventana de edición.
	Mosaico	Organizar las ventanas de edición que estén abiertas de modo que todas sean vistas en el pantalla, al menos parcialmente.
	Cascada	Organizar las ventanas de edición que estén abiertas de modo que queden solapadas.
	Siguiente	Activar la siguiente ventana abierta. Luego de la última ventana se volverá a la primera.
	Previo	Traer al frente la ventana previa.
	Resultados	Observar los resultados que hayan impreso los programas durante su ejecución.
	Visor de expresiones	Activar la ventana que contiene la lista de expresiones que desea monitorearse durante la ejecución de un programa.
Opciones	**Colores...**	Cambiar los colores de los elementos de SLE: menúes, texto, bordes, etc.
	Grabar escritorio	Grabar en un archivo la combinación de colores, ventanas abiertas y demás elementos de SLE, de modo que luego pueda recuperarse.
	Restaurar escritorio	Recuperar la combinación de colores, ventanas abiertas y demás elementos de SLE que fueron almacenados en un archivo con la opción "Grabar escritorio".

Uso de las *teclas rápidas*

Las *teclas rápidas* son teclas de función o combinaciones de teclas que pueden ser usadas para ejecutar ágilmente acciones sin necesidad de acceder al menú y seleccionar la opciones correspondientes. Las opciones más usadas del menú tienen asociadas una tecla rápida. Otras *teclas rápidas* también están situadas en la última línea de la pantalla de SLE.

A continuación se muestra el ítem "Ejecutar" del menú principal, seguida de una tabla que resume las principales teclas.

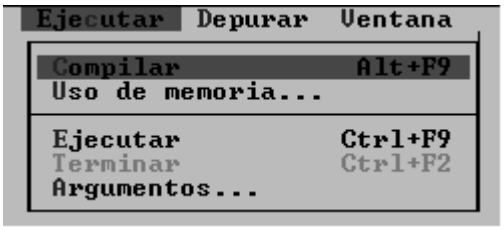


Figura 3. Ubicación de las *teclas rápidas*

Principales *teclas rápidas*

Tabla 2. Lista de las principales *teclas rápidas*

Tecla	Función
F2	Grabar el archivo que se está editando.
F3	Abrir un archivo que está almacenado en el disco.
F4	Ejecutar un programa hasta que la ejecución llegue a cierta línea en el programa fuente.
F5	Activar/desactivar el modo de pantalla completa de una ventana de edición.
F6	Activar la siguiente ventana abierta. Luego de la última ventana se volverá a la primera.
F7	Seguir línea a línea la ejecución de un programa.
F8	Seguir la ejecución de un programa. A diferencia de la opción "Paso a paso", las llamadas a subrutinas son ejecutadas en un solo paso, es decir, no se muestra la ejecución de las sentencias de las subrutinas.
F10	Acceder al menú principal.
Alt+F3	Cerrar una ventana.
Alt+F5	Observar los resultados que hayan impreso los programas durante su ejecución.
Alt+F9	Verificar que el programa está correcto, según las reglas de SL, y prepararlo para su ejecución posterior.
Ctrl+F2	Hacer que termine la ejecución de un programa.
Ctrl+F4	Observar qué valor tiene una variable o una expresión.
Ctrl+F7	Incluir una expresión en "Visor de expresiones".
Ctrl+F8	Establecer/desactivar punto de parada.
Ctrl+F9	Ejecutar el programa que se esté editando. El programa será primero compilado si SLE detecta que fue cambiado luego de la última compilación exitosa.

Teclas específicas del editor

Tabla 3. Lista de las principales teclas específicas del editor

Tecla	Función
Ctrl+Y	Eliminar una línea de texto.
Ctrl+QF	Localizar un texto determinado dentro del programa fuente.
Ctrl+QA	Sustituir un texto determinado por otro en el programa fuente.
Ctrl+L	Repetir la última operación de localización o sustitución.

Tabla 3. Lista de las principales teclas específicas del editor (continuación)

Tecla	Función
Ctrl+Ins	Insertar en el portapapeles una copia de la porción de texto que está marcada.
Shift+Ins	Insertar en el programa fuente que se está editando lo último que se haya copiado al portapapeles.
Ctrl+Del	Eliminar la porción de texto que está marcada.
Home	Ir al principio de la línea actual
End	Ir al final de la línea actual.
Ctrl+Backspace	Deshacer el último cambio realizado en el texto en edición.

Requerimientos de hardware y software

SLE requiere Windows95 o Windows98 para funcionar. Una configuración aceptable es una PC con:

- Procesador Pentium con 16MB de RAM (se recomienda 32MB).
- 10MB de espacio libre en disco duro.
- Monitor color.
- Mouse.

Capítulo

3

Aspectos básicos de SL

Los comentarios

Los comentarios son utilizados dentro de los programas SL para brindar información al programador que realizará, posiblemente, modificaciones al programa original. Es decir, sirve para documentar un programa o parte de él. Todos los comentarios insertados en un programa SL son completamente ignorados por el compilador.

Existen dos formas de introducir comentarios en el programa fuente:
Primera forma: Usando la doble barra (//).

El compilador ignora todo lo que sigue a la doble barra, hasta el final de la línea. Se utiliza generalmente para hacer acotaciones en la misma línea de alguna sentencia cuya compresión por parte del programador será facilitada con el comentario.

Si se desea escribir un comentario que ocupe más de una línea, cada una de éstas debe iniciarse con //.

Comentarios iniciados con // se utilizan a menudo para explicar el propósito de una variable.

Ejemplo:

```
programa cuadrado

//
// (Esto es un comentario!)
// Dado un número n, imprimir su cuadrado.
//

var
   n,                  // valor proveido por el usuario
   cuad : numerico     // cuadrado de n

inicio
   // Pidamos al usuario que ingrese un numero
   leer (n)
   // Calculemos el cuadrado de n para luego imprimirlo
   cuad = n * n
   imprimir (cuad)
fin
```

Segunda forma: Usando el par **/* … */**

A diferencia de la doble barra, el comentario puede ocupar varias líneas, es decir, el comentario termina solo cuando se encuentra la secuencia */ y no cuando se llega al final de la línea.

Lo que sigue es el mismo programa que imprime el cuadrado de un número, pero con comentarios multilíneas, combinados con el uso de la doble barra.

```
programa cuadrado
/*
   (Esto es un comentario!)
   Dado un número n, imprimir su cuadrado.
*/
```

```
var
   n,                  // valor proveido por el usuario
   cuad : numerico     // cuadrado de n

inicio
   /* Pidamos al usuario que ingrese un numero */

   leer (n)

   // Calculemos el cuadrado de n para luego imprimirlo

   cuad = n * n
   imprimir (cuad)
fin
```

Un comentario multilínea (delimitado por /* … */) puede contener otros comentarios iniciados con doble barra, aunque no a otro multilínea, es decir, no pueden anidarse.

Identificadores

Los identificadores son utilizados para dar nombre a los programas, variables, subrutinas, tipos de datos, constantes y otros elementos de un programa SL.

Existen algunas reglas a tener en cuenta en la creación de identificadores:

1. Deben comenzar con una letra o con el carácter '_' (guión bajo).
2. Pueden tener hasta 32 caracteres.
3. No deben contener espacios.
4. La combinación de mayúsculas y minúsculas en los identificadores hace que los mismos sean considerados diferentes. En otras palabras: **CANTIDAD, cantidad y Cantidad** representan tres identificadores distintos.
5. Pueden incluir las letras ñ y Ñ (eñe minúscula y mayúscula respectivamente).
6. No pueden contener caracteres acentuados.
7. Pueden incluir combinaciones de letras y números y el carácter '_' (guión bajo)

Así, los siguientes son identificadores válidos:

```
peso    total_puntos          MAXCOLUMNAS          Resultado
mat2    var_12_meses          sgte_año
```

Los siguientes no son aceptables como identificadores (las explicaciones van entre paréntesis):

```
2da_var          (debe empezar con letra o guión bajo)
$prueba          (debe empezar con letra o guión bajo)
cant alumnos           (contiene un espacio)
tot-puntos             (no puede contener guión)
```

Palabras reservadas

Las palabras utilizadas por SL para propósitos especiales son llamadas "palabras reservadas". Dado que las mismas tienen un significado específico para el compilador de SL, no pueden ser utilizadas como identificadores.

Todas las palabras reservadas deben ser escritas siempre completamente en letras minúsculas, de lo contrario el compilador SL no las reconocerá como tales.

Tabla 4. Palabras reservadas de SL

and	archivo	caso	const
constantes	desde	eval	fin
hasta	inicio	lib	libext
matriz	mientras	not	or
paso	subrutina	programa	ref
registro	repetir	retorna	si
sino	tipos	var	variables
vector			

Delimitadores de sentencias

Las sentencias pueden ocupar una o varias líneas, es decir:

```
a = (b + c) * (b / y) +
    (y / c)
```

es una sentencia válida, aunque ocupa dos líneas.

Puede escribirse más de una sentencia en una sola línea, si se las separa por el carácter punto y coma (';'). Así:

```
a = b + c;  n = a / y
```

son dos sentencias escritas en una sola línea, separadas por el carácter punto y coma (';').

Las expresiones numéricas complejas o las sentencias muy largas pueden ser libremente distribuidas en dos o más líneas para mejorar la legibilidad del programa fuente SL.

También debemos señalar que las líneas en blanco y los espacios entre las sentencias no son necesarios para la computadora interprete correctamente el programa, por lo tanto podríamos omitirlas, pero al suprimirlos haremos difícil a otras personas, e incluso a nosotros mismos, la comprensión de nuestros programas.

Estructura general de un programa SL

El siguiente diagrama muestra esquemáticamente las partes de un programa SL

```
programa nombre_del_programa
Const
   declaracion de una o más constantes
tipos
   declaracion de uno o más tipos de datos

var
   declaracion de una o más variables
inicio
   ...
   sentencias
   ...
fin
```

Esta especificación es opcional.

Constantes, tipos de datos y variables globales.

Cualquiera de estas cláusulas (const, tipos, var) pueden aparecer ninguna o varias veces, en cualquier orden.

Aquí empieza la ejecución del programa.

Aquí termina la ejecución del programa.

```
subrutina sub1() retorna tipo_dato
const
  ...
tipos
   ...
var
   ...
inicio
   ...
   sentencias
   retorna (algun_valor)
fin
```

Las subrutinas van después del programa principal, y a su vez pueden tener variables, constantes y tipos de datos locales.

Pueden recibir cualquier cantidad de parámetros. Si son funciones, retornan un valor.

```
subrutina sub2 (ref p : tipo_dato, ...)
const
  ...
tipos
   ...
var
   ...
inicio
   ...
   sentencias
   ...
fin
```

Los parámetros pueden ser pasados por valor o por referencia.

Capítulo

4

Tipos de datos básicos y constantes literales

El *tipo de dato* de una variable determina qué valores pueden ser almacenados en dicha variable y qué operaciones podemos realizar con ella. Por ejemplo, si usamos una variable para almacenar la altura de una persona estamos diciendo que dicha variable contendrá valores numéricos. En este caso diremos que el tipo de la variable `edad` es numérico.

En SL todos los valores y variables tienen asociados un tipo de dato y el compilador realizará verificaciones precisas para asegurar que las operaciones realizadas sean consistentes. Si por ejemplo se escribe un programa que contenga

```
a = "suma" + 10
```

el compilador señalará que existe un error pues sumar una palabra con 10 no produce un resultado significativo. El compilador dirá que no existe concordancia de tipos de datos pues, como veremos en seguida, "suma" es una cadena mientras que 10 es un numérico.

SL soporta tres tipos de datos básicos o fundamentales: numéricos, cadenas y booleanos. Estos tipos representan valores simples para los que el lenguaje tiene un rico conjunto de operadores y funciones predefinidas.

Existen además otros mecanismos para definir tipos de datos más complejos que se explican en capítulo "Tipos de datos agregados"

Tabla 5. Tipos de datos básicos de SL

Tipo de dato básico	Descripción de los valores que puede almacenar
cadena	Cualquier secuencia de caracteres ASCII, excepto el carácter NUL (valor ASCII 0). La cadena vacía se representa por "" y su longitud es 0.
logico	Los valores TRUE y FALSE, o sus sinónimos SI y NO respectivamente. Estos cuatro identificadores son constantes predefinidas.
numerico	Valores enteros y reales, con signo o sin signo.

Las constantes literales

Se llaman constantes literales a aquellos números o cadenas que forman parte del texto del programa.

Los números

Los números en SL pueden ser enteros o reales. Los enteros se componen únicamente de dígitos y opcionalmente el carácter de signo ('+' o '-') precediéndolos.

Los reales, en cambio, pueden contener además un punto decimal ('.') seguido de otros dígitos, que indican la parte fraccionaria del número. Además puede utilizarse notación científica. Algunos ejemplos de constantes numéricas son:

```
314          -100          +217
3.14         2e+10         5e-3          8E+131
```

Los valores máximos y mínimos son:

Valor mínimo : -1.7E-308
Valor máximo : 1.7E+308

Las cadenas de caracteres

Una cadena es una secuencia de caracteres encerrados entre comillas o apóstrofes.

Los siguientes son ejemplos de cadenas:

- "Aprendiendo a programar..."
- "Conteste con 'si' o 'no', por favor"
- 'María dijo: "¡Buenos días!"'

Como puede observarse, cualquier carácter puede ser incluido como parte de la cadena, excepto el delimitador. Si por ejemplo se desea incluir una comilla en un mensaje, debe iniciarse y finalizarse la cadena con apóstrofe, y viceversa, o utilizar \".

Las constantes de cadena deben caber completamente en una línea. Es decir, lo **siguiente no es válido** porque la comilla que cierra la cadena aparece recién en la segunda línea:

```
texto = "Aprender a programar
  es divertido"
```

Si es necesario distribuir una constante de cadena en más de una línea, puede usarse el operador de concatenación:

```
texto = "Aprender a programar" +
        "es divertido"
```

Las secuencias de escape

Las "secuencias de escape" permiten insertar en un programa fuente caracteres que, de ser tipeados directamente, serían mal interpretados por el compilador. La siguiente tabla resume estas secuencias y sus significados.

Tabla 6. Secuencias de escape

Secuencias	Significado
\n	Inicio de la siguiente línea
\r	Inicio de la línea actual
\t	Tabulador

Por ejemplo

```
imprimir ("Primera linea\nSegunda linea")
```

hará que lo que sigue a \n se imprima en otra línea:

```
Primera linea
Segunda linea
```

Cualquier otra combinación da como resultado el carácter que sigue a la primera barra invertida. Así:

```
'\.'                Da como resultado .
'\\'                Da como resultado \
'\''                Da como resultado '
```

No debemos olvidar que las secuencias de escape representan un solo carácter, aunque lo simbolicemos con dos. Es decir, la cadena "Buenos\ndías" contiene 11 caracteres:

1	2	3	4	5	6	7	8	9	10	11
B	u	e	n	o	s	\ n	d	í	a	s

Un ejemplo

El siguiente programa calcula la altura máxima que alcanza un proyectil lanzado con una velocidad y angulo proveídos por el usuario.

Como ya vimos con anterioridad, la cláusula "programa algun_nombre" es opcional. Su única función es proveer al programador una indicación del propósito del programa. Usemos o no la cláusula "programa", es importante poner un comentario al inicio al inicio del programa fuente, declarando los objetivos y otros datos adicionales de interés para quien deba leer y comprender nuestros programas.

```
/*
```

```
      OBJETIVO: Calcular la altura maxima alcanzada por un proyectil
                lanzado con una velocidad inicial y angulo dados.
      AUTOR:    jsegovia


      La formula utilizada es:
```

$$h = \frac{[V0 \cdot sen(a)]^2}{2 \cdot g}$$

```
      donde:
         V0   : velocidad inicial
         a    : angulo de disparo
         g    : aceleracion de la gravedad
   */

var
   v_disp,                          // Velocidad inicial
   a_disp : numerico                // Angulo de disparo

   a_disp_rad : numerico            // Igual que a_disp pero en radianes
   h_max      : numerico            // Altura maxima

inicio
   /*
      Pidamos al usuario que ingrese velocidad inicial y angulo de
      disparo. Este ultimo debe estar en grados sexagesimales.
   */
   imprimir ("\n\n",
             "Ingrese velocidad inicial, angulo de disparo en sexagesimales:")
   leer (v_disp, a_disp)

   // La funcion sin() opera con radianes y nosotros leimos en sexagesimales.

   a_disp_rad = a_disp * 3.141592654 / 180

   h_max = ( (v_disp * sin (a_disp_rad) ) ^ 2 )  /  (2 * 9.8)
   imprimir ("\n\nAltura maxima=", h_max)
fin
```

Capítulo

5

Uso de variables y constantes

Un programa SL debe declarar todas sus variables antes de usarlas. Si bien la declaración de las variables puede resultar tediosa al principio, al hacerlo estamos dando al compilador suficiente información que le permita:

- Ayudarnos en la detección de errores de tipeo. Muchas veces inadvertidamente escribimos mal el nombre de las variables, las palabras reservadas, etc.
- Mejorar el diagnóstico de errores de sintaxis o de uso del lenguaje.
- Optimizar el rendimiento del programa durante su ejecución y planificar el uso de la memoria.

A continuación veremos la declaración de variables simples, arreglos (vectores y matrices) y registros, así como el uso de constantes manifiestas.

Declaración de variables simples

Para declarar una variable, se escribe el nombre seguido de dos puntos (':'), seguido del tipo de dato:

```
var
   edad : numerico
   nombre : cadena
```

Si varias variables corresponden al mismo tipo, se las puede declarar de la siguiente forma:

```
var
      tot_mensual, tot_general : numerico
      nombre, apellido : cadena
```

Los valores por defecto

SL da automáticamente un valor inicial a las variables, dependiendo de qué tipo sean. La siguiente tabla resume los valores iniciales:

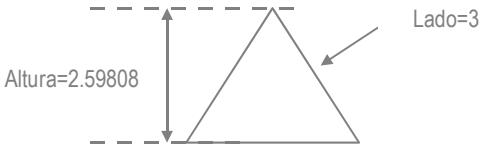
Tabla 7. Valores iniciales de las variables en SL

Tipo de dato básico	Valor inicial
cadena	Cadena vacía. Es decir, SL implícitamente hace `variable = ""`
logico	Valor lógico falso. Es decir, SL implícitamente hace `variable = FALSE`
numerico	Cero. Es decir, SL implícitamente hace `variable = 0`

Declaración de constantes manifiestas

Las constantes manifiestas (o simplemente constantes) son nombres a los que asignamos un valor que no cambiará durante la ejecución de todo el programa.

El siguiente programa utiliza una constante manifiesta (RAIZ_3) y calcula la altura de un triángulo equilátero, si se da la longitud del lado. El programa imprimirá 2.59808 si se ingresa 3 como longitud del lado, que corresponde a la siguiente figura:




```
/*
      OBJETIVO: Calcular la altura de un triangulo equilatero, dado el lado.
      AUTOR:    jsegovia
```

```
      La formula utilizada es:

      h = a * RAIZ_3 / 2

      donde:
         a    : lado del triángulo
   */

   const
      RAIZ_3 = 1.732050808

   var
      a, h     : numerico            // lado y altura, respectivamente

   inicio
      imprimir ("\n\nIngrese longitud del lado:")
      leer (a)

      h = a * RAIZ_3 / 2

      imprimir ("\n\nAltura del triángulo=", h)
   fin
```

Algunas observaciones sobre las constantes manifiestas

Las constantes manifiestas son útiles para la programación porque:

- Mejoran la legibilidad del programa, dando nombres descriptivos a valores que de otra forman pueden lucir "mágicos".
- Facilitan el mantenimiento, ya que si el valor constante se usa en muchas partes del programa y necesita ser modificado, solo debe hacerse en una sola parte.

Todas las constantes manifiestas corresponden a un tipo de dato, que depende del valor que se les asocie. Por ejemplo la siguiente es una constante manifiesta de cadena:

```
const
      DIGITOS = "0123456789"
```

Además el compilador SL verifica que el valor de la constante nunca sea alterado, directa o indirectamente. Por ejemplo, ocurrirá un error de compilación si en el cuerpo del programa se intenta asignar algún valor a la constante `DIGITOS`.

Constantes manifiestas predefinidas

SL tiene cuatro constantes manifiestas lógicas predefinidas: TRUE, FALSE, SI, NO.

Tabla 8. Constantes manifiestas predefinidas de SL

Constante manifiesta	Valor asociado
TRUE	Correspondiente al valor de verdad verdadero
FALSE	Correspondiente al valor de verdad falso
SI	Sinónimo de TRUE
NO	Sinónimo de FALSE

Declaración de vectores y matrices

La declaración de vectores en SL es muy sencilla: se declara como una variables más, indicando al compilador los datos adicionales que necesita:

- La cantidad de elementos que tendrá, y
- De qué tipo serán los elementos, es decir, si serán numéricos, de cadena o lógicos. En capítulos siguientes veremos que incluso pueden tener elementos mucho más complejos que estos tres básicos.

Por ejemplo, para declarar un vector numérico de 100 elementos escribiremos:

```
var
   A : vector [100] numerico
```

Para imprimir el valor del primer elemento escribiremos:

```
imprimir (A [1])
```

Con el siguiente ciclo podemos imprimir todos los elementos del vector, uno en cada línea:

```
desde k=1 hasta 100
{
   imprimir ("\n", A [k])
}
```

Un ejemplo sencillo de uso de vectores

A continuación presentamos un programa que calcula la cantidad de alumnos que obtuvieron una nota inferior al promedio del curso para cierta materia. Tendremos en consideración los siguientes datos adicionales:

- El curso tiene 20 alumnos
- Todos los alumnos se presentaron al examen
- Las notas van del 0 al 100

Dado que necesitaremos almacenar la nota de cada alumno para luego compararlo con el promedio del curso, usaremos un vector.

```
   /*
      OBJETIVO: Calcular la cantidad de alumnos que obtuvieron nota
                inferior al promedio del curso en cierta materia.
                Hay 20 alumnos. Todos rindieron. Las notas van del 0 al 100.
                Asumimos que todas las notas son correctas.
      AUTOR:    jsegovia
   */

const
   MAX_ALUMNOS = 20

var
   notas      : vector [MAX_ALUMNOS] numerico
   suma_notas : numerico                  // suma de todas las notas leidas
   prom  : numerico                       // promedio del curso
   cnb   : numerico                       // cantidad con notas bajas
   k     : numerico

inicio
   desde k=1 hasta MAX_ALUMNOS
   {
      imprimir ("\nIngrese nota para alumno numero ", k, ":")
      leer (notas [k])


      /*
         Vayamos obteniendo la suma de las notas para que, una vez
         terminada la lectura, podamos calcular el promedio.
         Podriamos hacerlo en otro ciclo, pero ya que tenemos el dato...
      */
      suma_notas = suma_notas + notas [k]
   }

   prom = suma_notas / MAX_ALUMNOS
```

```
    /*
       Observar que la cantidad de alumnos con notas inferiores
       al promedio del curso SOLO podremos calcular una vez
       obtenido el promedio.
    */

    cnb = 0

    desde k=1 hasta MAX_ALUMNOS
    {
       si ( notas [k] < prom )
       {
          cnb = cnb + 1
       }
    }
    imprimir ("\nExisten ", cnb, " alumnos con notas inferiores",
              " al promedio del curso, que es ", prom)
fin
```

Si los datos proveídos al programa fueran:

89	73	84	94	86	94	59	91	98	43	53	75	62	43	88	77	94	87	92	90

el resultado sería:

```
Existen 8 alumnos con notas inferiores al promedio del curso, que es 78.6
```

Declaración y uso de matrices

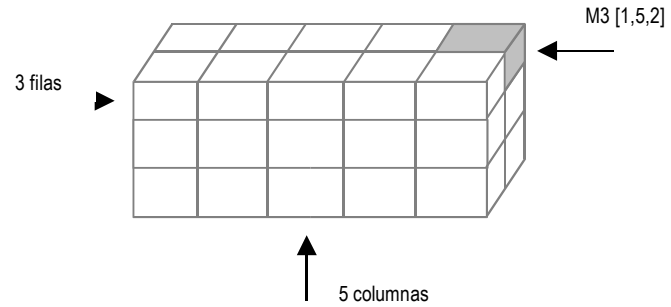
Las matrices son declaradas exactamente igual que los vectores, excepto que se especifica más de una dimensión entre los corchetes:

```
var
   M : matriz [5, 10] cadena
```

En este caso M es una matriz de 5 filas por 10 columnas, donde cada elemento es una cadena.

La siguiente matriz tridimensional se declara con:

```
   M3 : matriz [3, 5, 2] numerico
```



Declaración de registros

Se llama "registro" (algunos usan el término "estructura") a una variable que está compuesta a su vez de otras variables.

La ventaja de agrupar variables para formar una estructura más compleja es que de esa forma se mantiene la unidad conceptual del dato. Por ejemplo, una fecha es un solo "dato", a pesar de que se compone de día, mes y año. En muchos casos es deseable tratarla como una sola entidad y en otros deseamos operar sobre sus componentes.

A continuación un ejemplo de declaración de registro:

```
var
   r : registro
       {
          nombre     : cadena
          edad, peso : numerico
          direccion  : cadena
       }
```

A las variables contenidas en un registro se las llama campos. En este caso, r es una variable que contiene tres campos, los que pueden ser accedidos de la siguiente forma:

```
r.nombre    = "Maria Gonzalez"
r.edad      = 15
r.peso      = 48
r.direccion = "Algun lugar"
```

es decir, los campos son accedidos poniendo el nombre de la variable-registro, seguido de un punto y a continuación el nombre del campo. Los campos son utilizados como cualquier otra variable.
Los registros pueden contener variables de cualquier tipo, incluyendo vectores, otros registros, etc.

Dadas las siguientes definiciones:

```
var
   p1, p2 : registro
        {
           nombre     : cadena
           edad, peso : numerico
           direccion  : cadena
           fecha_nac  : registro
                        {
                           dia, mes, año : numerico
                        }
           notas      : vector [100] numerico
        }
```

Las variables p1 y p2 son registros, ambos conteniendo 6 campos, uno de los cuales (fecha_nac) es a su vez un registro y el último es un vector numérico de 100 elementos. Por lo tanto, para asignar un valor al campo dia de la variable p1 se usaría:

```
p1.fecha_nac.dia = 23
```

y para asignar un valor al segundo elemento del vector notas de la variable p2 se usaría:

```
p2.notas [2] = 98
```

En el capítulo 8 veremos en más detalle estas y otras posibilidades de los registros.

Otras consideraciones sobre el uso de registros

- Los nombres de campos no pueden duplicarse dentro de un mismo registro.
- Para leer o imprimir un registro, debe realizarse la operación deseada sobre cada uno de sus campos, toda vez que estos sean de tipos simples.
- Una definición de registro debe especificar al menos un campo.

Capítulo

6

Operadores y expresiones

Llamamos *expresión* a una combinación válida de operadores, constantes, variables y llamadas a funciones. Las expresiones tienen un papel fundamental en SL pues a través de ellas expresamos los cálculos y otras operaciones que deseamos que el programa realice.

El uso de expresiones es muy sencillo porque en su mayoría se componen de operaciones aritméticas de uso cotidiano: sumas, multiplicaciones, potencias, etc.

Algunos ejemplos de expresiones son:

- `peso * 10`
- `cantidad + 1`
- `sqrt (a^2 + b^2)  // sqrt() calcula la raiz cuadrada`
- `apellido + ", " + nombre  // produce, por ejemplo, "Gonzalez, Marta"`

Operadores

Los *operadores* son símbolos que usamos para indicar la realización de operaciones con ciertos datos, a los que llamamos *operandos*. Los operadores

de SL se dividen en cuatro grupos: aritméticos, lógicos, relacionales y de concatenación de cadenas.

Operadores aritméticos

La tabla siguiente resume los operadores aritméticos disponibles en SL. Estos operadores pueden aplicarse a cualquier valor numérico, sea que este tenga parte fraccionaria o no.

Como puede apreciarse, todos estos operadores, con excepción de los dos últimos, son binarios (requieren dos operandos).

Tabla 9. Operadores aritméticos

Opera-dor	Significado	Ejemplos y observaciones
+	Suma	`a = b + 10`
-	Resta	`a = b – 10`
*****	Multiplicación	`a = b * 10`

/	División.	`a = 1234 / 10  // producirá 123,4`
		
		Si se desea solo la parte entera, puede usarse la función int():
		`a = int (1234 / 10) // producirá 123`
		
		Si el divisor es cero, se produce un error de ejecución y el programa termina.

%	Módulo. Calcula el resto de dividir el primer operando por el segundo	`a = 10 % 3  // producirá 1`
		
		Este operador IGNORA las partes fraccionarias de ambos operandos:
		`a = 10.4 % 3.01  // producirá también 1`
		
		Si el divisor es cero, se produce un error de ejecución y el programa termina.

Tabla 9. Operadores aritméticos (continuación)

Opera -dor	Significado	Ejemplos y observaciones

^	Potenciación.	`a = 2 ^ 4` `// producirá 16`
		`a = 2 ^ 0.5` `// producirá 1.41421 (raíz cuadrada de 2)`
		El exponente puede ser un valor entero o fraccionario.
		`a = 2 ^ 3 ^ 2`
		se interpreta como
		`a = 2 ^ (3 ^ 2)`

-	Cambio de signo	`a = 10; b = -a` `// b tendrá -10` Se le llama también "menos unitario" pues actúa sobre un solo operando.
+	Identidad de signo. El signo del operando al que se aplica queda inalterado.	`a = -10; b = +a` `// b tendrá -10`

Operadores lógicos

La tabla siguiente resume los operadores lógicos o booleanos disponibles en SL. Los operadores **and** y **or** son binarios, mientras que **not** es unitario.

Tabla 10. Operadores lógicos o booleanos

Operador	Significado	Observaciones

and	Conjunción	`// está a entre 1 y 100?`
		`si ( a >= 1 and a <= 100 )`
		`{`
		`  ...`
		`}`

or	Disyunción	`extraño = 100 > 2000 or 1 < 10` La variable "extraño", que debió se declarada como de tipo lógico, tendrá el valor de verdad verdadero, ya que 1 es menor que 10, a pesar de que 100 no es mayor que 2000.
not	Negación	`extraño = not(1 < 10)` La variable "extraño", que debió se declarada como de tipo lógico, tendrá el valor de verdad falso. Los paréntesis no son necesarios aquí, aunque deseables por claridad.

Operadores relacionales

Los operadores relacionales disponibles en SL se detallan en la tabla siguiente. Todos estos operadores son binarios, es decir, requieren dos operandos. Pueden ser aplicados a cualquier tipo simple (numérico, cadena o lógico) siempre que ambos sean del mismo tipo y producen un valor booleano (verdadero o falso).

Tabla 11. Operadores relacionales

Operador	Significado	Ejemplos
<	Menor que	`a < 20`
>	Mayor que	`a > 1`
<=	Menor o igual que	`a <= 20` El signo = debe seguir inmediatamente al signo < para que el compilador interprete <= como "menor o igual que". Además, no es equivalente escribir =<.
>=	Mayor o igual que	`a >= 20` El signo = debe seguir inmediatamente al signo > para que el compilador interprete >= como "menor o igual que". Además, no es equivalente escribir =>.
==	Igual que	`a == b` No debe existir ningún otro carácter entre el primer signo = y el segundo para que el compilador interprete == como "igual que".
<>	Distinto de	`a <> b` El signo > debe seguir inmediatamente al signo < para que el compilador interprete <> como "distinto de". Además, no es equivalente escribir ><.

Operadores relacionales y cadenas

Cuando dos cadenas, sean éstas literales o variables, aparecen como operandos de un operador relacional, cada carácter de la primera cadena es comparado el carácter que se encuentra en la segunda cadena en la misma posición desde la izquierda.

SL considera que "A" es menor que "B" porque la primera letra aparece antes que la segunda en la tabla ASCII. Esta comparación de posiciones en la

tabla ASCII determina el funcionamiento de los operadores relacionales aplicados a cadenas. A continuación algunos ejemplos.

Tabla 12. Algunos ejemplos del uso de cadenas con operadores relacionales

Comparación	Resultado	Explicación
"ABC" < "ACC"	Verdadero	El segundo carácter de la primera cadena es "B", mientras que de la segunda cadena es "C". En la tabla ASCII "B" está antes que "C".
"a" <= "A"	Falso	En la tabla ASCII las letras minúsculas aparecen mucho después que las mayúsculas, es decir, son "mayores".
"ABA" > "AB"	Verdadero	Los dos primeros caracteres de ambas cadenas coinciden pero la primera cadena tiene un carácter más que la segunda, lo que la convierte en "mayor" que la segunda.
"AbCdE" == "AbCdE"	Verdadero	Los caracteres de ambas cadenas son los mismos, en el mismo orden.
"Z" > ""	Verdadero	Una cadena de longitud uno es mayor que la cadena vacía.
"" == ""	Verdadero	Ambas cadenas tienen longitud 0.

Concatenación de cadenas

Llamamos concatenación a la operación de tomar dos cadenas y generar una tercera que contiene todos los caracteres de la primera cadena seguidos de los de la segunda. Veamos un ejemplo:

```
/*
   OBJETIVO: Mostrar el uso del operador de concatenación de cadenas.
   AUTOR:    jsegovia
*/

var
   cad1, cad2, cad3 : cadena

inicio
   cad1 = "Primer"
   cad2 = "Curso"
   cad3 = cad1 + cad2
   imprimir (cad3)
fin
```

El programa imprimirá

```
PrimerCurso
```

Si deseamos dejar un espacio entre la primera y la segunda cadena, podemos escribir:

```
cad3 = cad1 + " " + cad2
```

El signo utilizado para concatenar cadenas es el mismo que usamos para sumar números. El compilador puede interpretar correctamente lo que deseamos realizar porque inspecciona el tipo de las variables que acompañan al signo '+': Si ambos operandos son numéricos, se trata de una suma; si ambos son cadenas, se requiere una concatenación; de lo contrario hay un error.

Es importante señalar que la concatenación, a diferencia de la suma de números, NO es conmutativa. Es decir, no da lo mismo

```
cad3 = cad1 + cad2
```

que

```
cad3 = cad2 + cad1
```

En capítulos siguientes veremos que existen varias funciones predefinidas y otras operaciones sofisticadas que podemos realizar con cadenas de caracteres.

Precedencia de los operadores

Las reglas de precedencia de los operadores determinan en qué orden SL evalúa las expresiones. En general, los operadores aritméticos tienen la precedencia usual. La siguiente tabla define las precedencias de los operadores en SL.

Tabla 13. Precedencia de los operadores

Precedencia	Operadores
1 (mayor precedencia)	**^** (potenciación)
2	**-, +** (cambio de signo, identidad de signo)
3	***, /, %**
4	**+, -**

Tabla 13. Precedencia de los operadores (continuación)

Precedencia	Operadores
5	**==, <>, <, <=, >, >=**
6	**not**
7	**and**
8 (menor precedencia)	**or**

Los operadores que se encuentran en la misma línea tienen igual precedencia. Si forman parte de una expresión o subexpresión no parentizada, se evaluarán de izquierda a derecha.

Siguiendo la tabla anterior, la siguientes sentencias hacen que la variable termine con el valor 31:

```
a = 2
b = 4 + a * 3          // b = 4 * (2 * 3)          => 4 * 6          => 24
c = a ^ 2 * -b         // c = (2 ^ 2) * -24        => 4 *-24         => -96
a = b - c % 10 + 1     // a = 24 - (-96 % 10) + 1  => 24 - (-6) + 1  => 31
```

Las expresiones complejas y el uso de paréntesis

Podemos utilizar paréntesis con dos propósitos:

- Para hacer que las expresiones sean más legibles.
  La "abundancia" de paréntesis no tiene ningún impacto negativo sobre la ejecución del programa y en cambio puede facilitar su comprensión, no solo por parte de uno mismo sino por otros que deban entender el programa después. Un ejemplo que muestra el uso de paréntesis con este propósito es:

```
y = v0y * t + 1/2 * -g * t^2          // sin parentización "redundante"
y = (v0y * t) + ( (1/2)*(-g)*(t^2) )
```

- Para alterar el orden en que se evalúan las expresiones, si la precedencia por defecto no produce el resultado que necesitamos. Veamos dos ejemplos:

```
a) 3 * 2 + 1          // es 7, pero
   3 * (2 + 1)        // es 9

b) -2^2          // es -4, pero
     (-2)^2            // es 4
```

Capítulo

7

Sentencias

Las sentencias en SL especifican cómo se irán ejecutando las distintas partes de un programa. Existen varias clases de sentencias: algunas que permiten ejecutar repetidamente un conjunto de instrucciones (ciclos), otras para ejecutarlas selectivamente dependiendo de ciertas condiciones (sentencias condicionales), alterar el valor de las variables, ejecutar un subprograma, etc.

En este capítulo veremos la sintaxis de cada una de las sentencias de SL, así como algunos ejemplos de uso.

Sentencia de asignación

La asignación a una variable tiene la siguiente forma general:

```
nombre_de_variable = expresión
```

Ejemplos de asignaciones son:

```
a = 10.231
b = a + 1
hipotenusa = sqrt (a^2 + b^2)
```

Más adelante veremos que la asignación entre vectores, matrices y registros tiene otras opciones que pueden facilitar la escritura de los programas.

Sentencia condicional (si)

En SL existen dos sentencias condicionales: sentencia **si** y sentencia **eval**. La primera posibilita elegir qué parte de un programa se ejecutará si cierta condición se cumple. La segunda puede ayudar a mejorar la legibilidad de un grupo de sentencias **si** muy largas o complejas.

La sintaxis básica de la sentencia **si** es:

```
si ( condicion )
{
   sentencias...
sino
   sentencias...
}
```

La cláusula "sino" y las sentencias correspondientes son opcionales: solo se usan cuando el programa necesita tomar acciones si la condición es falsa.

Primer ejemplo

Una característica peculiar de los triángulos es que ninguno de sus lados puede ser superior a la suma de los otros dos. El siguiente programa verifica si tres valores ingresados por el usuario pueden ser utilizados como longitudes de un triángulo. El programa utiliza una construcción sencilla **si…sino**.

```
/*
      OBJETIVO: Determinar si tres valores numericos ingresados por
                el usuario pueden ser los lados de un triángulo.
      AUTOR:    jsegovia
      FECHA:    18-dic-1999
*/
var
   a, b, c : numerico
inicio
   leer (a, b, c)
```

```
    si ( (a > b + c) or (b > a + c) or (c > a + b) )
    {
        imprimir ("\nNo pueden ser los lados de un triángulo")
    sino
        imprimir ("\nPueden formar un triángulo")
    }
fin
```

Segundo ejemplo

El siguiente ejemplo usa una variante de la sentencia **si** que posibilita verificar varias condiciones y ejecutar acciones para cada caso. El programa debe procesar los puntajes de un grupo de 20 alumnos correspondientes a cierta asignatura y determinar cuántos aplazados hubo, cuántos obtuvieron 3 y cuántos 5.

El programa usa una variante de la sentencia **si** cuya sintaxis es:

```
si ( condicion )
{
   sentencias...
sino si ( condicion )
   sentencias...
sino
   sentencias...
}
```

Como puede apreciarse, esta variante permite "encadenar" varios condicionales en una sola sentencia si. Esta estructura es muy útil porque permite obviar anidamientos innecesarios. Debe observarse que las secuencias "sino si (condición)" no van seguidas de "{". Además la palabra "si" debe ubicarse en la misma línea que el "sino" correspondiente.

```
/*
   OBJETIVO: Calcular cuántos unos, tres y cincos se dieron en un grupo
de
             20 alumnos considerando los puntajes de una asignatura
             determinada.

             Los puntajes están expresados en pocentajes.
             Escala:
                0 -  59  -> 1
               60 -  74  -> 2
```

```
                75 -  84  -> 3
                85 -  94  -> 4
                95 - 100  -> 5

   AUTOR:    jsegovia (jsegovia@cnc.una.py)
*/
var
   n1,                  // Cantidad de unos,
   n3,                  // de tres y
   n5 : numerico        // de cincos

   pnt : numerico       // Puntaje obtenido por un alumno
   k : numerico

inicio
   k = 1
   n1 = 0; n3 = 0; n5 = 0
   repetir
      leer (pnt)
      si ( pnt >= 95 )
      {
          n5 = n5 + 1
      sino si ( pnt >= 85 )
          n3 = n3 + 1
      sino si ( pnt < 60 )
          n1 = n1 + 1
      }
      k = k + 1
   hasta ( k > 20 )
   imprimir ("\nCantidad de unos: ", n1,
             "\nCantidad de tres: ", n3,
             "\nCantidad de cincos: ", n5);
fin
```

Ciclo mientras

Esta construcción posibilita ejecutar repetidamente un conjunto de sentencias mientras se cumpla cierta condición definida por el programador.

La sintaxis del ciclo mientras es:

```
mientras ( condición )
{
    sentencias...
}
```

Debemos notar que la condición va siempre entre paréntesis.

Primer ejemplo

Como ejemplo de uso del ciclo mientras escribiremos un programa para calcular cuántos guaraníes recaudaremos en un sorteo si vendemos tíckets numerados del 1000 al 9999, asumiendo que vendemos todos los números.

Cada tícket cuesta lo que su número indica, es decir, el tícket número 425 cuesta cuatrocientos veinticinco.

```
/*
   OBJETIVO: Calcular cuanto se recaudara' en un sorteo, si vendemos
todos
             Los tickets numerados del 1000 al 9999.
   AUTOR:    desconocido

   OBSERVACION: El autor de este programa no sabe que la formula directa
                para calcular el total de guaranies es:

     total = (ult_numero - prim_numero + 1) / 2 * (prim_num + ult_numero)

                Por eso hace un programa "largo" usando ciclos.
*/

var
   total : numerico
   num   : numerico

inicio
   total = 0
   num = 1000



   mientras ( num <= 9999 )
   {
      total = total + num
      num   = num + 1
   }
   imprimir ("\nSera' recaudada la suma de Gs. ", total);
fin
```

Segundo ejemplo

El ciclo mientras tiene una peculiaridad: las sentencias contenidas en el ciclo puede que nunca se ejecuten. Esto ocurrirá cuando la condición del ciclo sea falsa incluso antes de entrar al ciclo. Esta característica del ciclo mientras es comúnmente utilizada en muchas aplicaciones cotidianas.

Como ejemplo, el programa que sigue calcula el cociente de la división entera de dos números también enteros.

```
/*
   OBJETIVO: Calcular el cociente de a/b, donde a y b son enteros.
             La division se hace por restas sucesivas.
   AUTOR:    desconocido

   OBSERVACION: El autor de este programa no sabe que puede usarse
                   cociente = int (a / b)

                Por eso hace un programa "largo" usando ciclos.

                Ademas, el programa trata de ser "comunicativo" con el
                 usuario para mostrar distintos usos de imprimir().
*/

var
   a, b, cociente : numerico

inicio
   imprimir ("\nPrograma para calcular cociente de la division",
             "entera de dos numeros, ambos enteros."
            )
   imprimir ("\nIngrese dos numeros enteros, separados por comas:")
   leer (a, b)

   cociente = 0



   /*
      Si el usuario tipea:
          8, 15

      el ciclo nunca se ejecutara. La variable cociente seguira
      teniendo 0, que es precisamente la respuesta que el programa
      debe dar en este caso.
   */
```

```
   mientras ( a >= b )
   {
      a = a  - b
      cociente = cociente + 1
   }

   imprimir ("\nEl cociente es ", a);
fin
```

Ciclo repetir...hasta

A diferencia del ciclo mientras, el ciclo repetir-hasta ejecuta al menos una vez las sentencias definidas en el cuerpo del ciclo, pues la verificación para continuar o no en el ciclo se realiza al final, como puede apreciarse en la sintaxis:

```
repetir
   sentencias...
hasta ( condicion_de_fin )
```

Debemos observar que la condición de fin va entre paréntesis.

Si bien esta forma de ciclo es utilizado con menor frecuencia que el ciclo mientras, hay situaciones en las que con él el algoritmo resultante es más sencillo y directo. A continuación un ejemplo:

```
/*
   OBJETIVO: Separar cada digito de un entero positivo ingresado por
             el usuario,  empezando por el primero de la derecha.
             Cada digito se imprime en  una linea independiente.
   AUTOR:    jsegovia
*/


var
   num, dig : numerico

inicio
   imprimir ("\nIngrese un numero entero positivo:")
   leer (num)

   imprimir ("\nLos digitos de ", num, " son:")

   /*
      Un caso "especial" lo representa el numero 0, que consta
```

```
        de un solo digito (el 0!)
        Usando el ciclo mientras la solucion seria un poco mas larga.
    */
    repetir
        dig = num % 10
        imprimir ("\n", dig)
        num = int (num / 10)
    hasta ( num == 0 )
fin
```

Sentencia eval

La sentencia eval es funcionalmente equivalente a la forma extendida de la sentencia si, pero aporta mayor legibilidad cuando las condiciones son varias o complejas.

La sintaxis de la sentencia eval es:

```
eval
{
    caso ( condicion )
         sentencias...
    caso ( condicion )
         sentencias...
    sino
         sentencias...
}
```

El programa verificará el valor de verdad de la primera condición. Si se cumple, ejecutará todas las acciones definidas para ese caso en particular y luego irá a ejecutar la sentencia que siga a la llave que cierra el cuerpo de la sentencia eval. Si no se cumple, pasará a verificar la siguiente condición y así sucesivamente. Si ninguna condición fuese verdadera y existiese una cláusula "sino", se ejecutarán las acciones definidas bajo esta cláusula. La cláusula "sino" es opcional. Si está presente debe ser la última.

Ejemplo

El siguiente programa clasifica lo ingresado por el usuario en "letra mayúscula", "letra minúscula", "dígito", o "algún otro". Para determinar a qué categoría pertenece un carácter se aprovecha la distribución de los

caracteres en la tabla ASCII junto con los operadores relaciones mayor o igual (“>=”) y menor o igual (“<=”).

```
/*
   OBJETIVO: Clasificar un carácter ingresado por el usuario en
             “letra mayúscula”, “letra minúscula”, “dígito” o
             “algún otro”.
   AUTOR:    jsegovia
*/
var
   let : cadena
   msg : cadena

inicio
   leer (let)

   /*
      Consideremos solo el primer carácter, por si haya tipeado
      más de uno.
      Observar cómo se usan los corchetes para seleccionar el
      primer carácter.
   */
   let = let [1]
   eval
   {
      caso ( let >= ‘A’ and let <= ‘Z’ )
           msg = “letra mayúscula”
      caso ( let >= ‘a’ and let <= ‘z’ )
           msg = “letra minúscula”
      caso ( let >= ‘0’ and let <= ‘9’ )
           msg = “dígito”
      sino
           msg = “algún otro”
   }
   imprimir (“\nEl carácter que usted ingresó es “, msg)
fin
```

Ciclo desde

Utilizamos el ciclo desde cuando conocemos previamente el número de veces que deseamos ejecutar un conjunto de instrucciones.

La sintaxis es:

```
desde variable_de_control = valor_inicio hasta valor_fin [paso
incremento]
{
   sentencias...
}
```

Al igual que el ciclo mientras, el conjunto de sentencias que forman parte de un ciclo desde puede que nunca se ejecute. Esto ocurrirá si:

- El valor inicial es superior al valor final y el incremento es positivo, ó
- El valor inicial es inferior al valor final y el incremento es negativo.

Ejemplo

El siguiente programa imprime el cuadrado de los primeros 10 números enteros positivos.

```
/*
   OBJETIVO: Imprimir una tabla con el cuadrado de los primeros 10
             numeros enteros.
   AUTOR:    jsegovia
*/

var
   k : numerico

inicio

   desde k=1 hasta 10
   {
        imprimir ("\n", k, " :\t", k^2)
   }
fin
```

El programa imprime:

```
1 :     1
2 :     4
3 :     9
4 :     16
5 :     25
6 :     36
7 :     49
8 :     64
9 :     81
10 :    100
```

Mas detalles acerca del ciclo desde

Cuando usamos el ciclo desde debemos tener presente algunos detalles:

- El valor de inicio, el valor de fin y el incremento deben ser expresiones numéricas. El primero (inicio) y el último (incremento) son siempre automáticamente truncados a enteros.
- El valor del incremento puede ser negativo (por ejemplo, para ir de 100 a 1).
- El ciclo puede realizarse de 0 a n veces.
- La variable de control debe ser numérica, pudiendo ser un elemento de un vector o matriz.
- Alterar el valor de inicio, el valor de fin o el incremento no tiene ningún efecto sobre la cantidad de veces que se ejecutará el ciclo (esto se calcula justo antes de iniciar el ciclo).
- Si se desea salir del ciclo en forma anticipada, puede asignarse a la variable de control el valor de fin.

Sentencia de llamada a subrutinas

Una subrutina puede ser ejecutada escribiendo su nombre, seguido de los parámetros reales que ella necesite. Podemos invocar la ejecución de subrutinas que hemos escrito en nuestro programa u otras que estén predefinidas en SL. La definición de subrutinas, los parámetros formales, los valores de retorno y otros detalles veremos en el apartado "Subrutinas".

La sintaxis de llamada a subrutinas es:

```
nombre_subrutina (parametro1, parametro2, ...)
```

La cantidad de parámetros y el tipo de dato de cada parámetro depende de la definición de la subrutina.

Ejemplo

El siguiente programa imprime la suma de los primeros n números enteros. El valor de n es proveído por el usuario. Vea el apartado "Subrutinas" para más detalles.

```
/*
   OBJETIVO: Calcular e imprimir la suma de los primeros n
             numeros enteros. Muestra el uso de subrutinas.
   AUTOR:    jsegovia
*/

var
   tot : numerico
   n : numerico
inicio
   /*
      cls() es una subrutina predefinida que limpia la pantalla
   */
   cls()
   imprimir_explicacion()
   imprimir ("Numero entero? ")
   leer (n)
   calcular_suma (n)
   imprimir ("\nLa suma de los primeros ", n, "números enteros es ", tot)
fin

/* continuación del programa ejemplo_llamada_subrutinas */

subrutina imprimir_explicacion()
inicio
   imprimir ("\nEste programa de ejemplo calcula la suma de los enteros",
             "\npositivos comprendidos entre 1 y un valor ",
             "que se ingresa\n")
fin

subrutina calcular_suma (entero_tope : numerico)
/*
   Ejemplo de subrutina que recibe un parametro.
*/
var
   k : numerico
inicio
   tot = 0

   desde k = 1 hasta entero_tope
   {
      tot = tot + k
   }
fin
```

Sentencia retorna

Finaliza la ejecución de una subrutina que retorna un valor (funciones) y debe ser la última sentencia, por lo que solo puede aparecer una vez en cada subrutina.

La sintaxis es:

```
retorna ( expresion )
```

Capítulo

8

Características avanzadas de los arreglos y registros

En el capítulo 5 ya hemos visto cómo declarar vectores, matrices y registros. En éste veremos con mayor detalle otros aspectos de SL particularmente interesantes: creación dinámica de arreglos, inicialización de arreglos y registros con literales estructurados, arreglos con contornos no regulares, asignación entre arreglos, asignación entre registros, y arreglos de registros.

Creación dinámica de arreglos

Como ya habíamos visto, la siguiente declaración habilita un vector de 100 elementos numéricos:

```
v : vector [100] numerico
```

En este caso el tamaño del vector ya está definido al momento de la compilación. Sin embargo existen algunos problemas de la vida real en los que no es posible conocer o no es conveniente predefinir la cantidad máxima de elementos que se requerirán. Puede incluso que el valor máximo varíe de

tiempo en tiempo. En estas situaciones podemos usar arreglos cuya creación sea dinámica, que se declaran como sigue:

```
v : vector [*] numerico
```

es decir, en vez del tamaño ubicamos un asterisco para indicar que dicho valor aún no conocemos.

Ya en el cuerpo del programa, cuando sepamos la cantidad de elementos que necesitamos, usamos la subrutina predefinida dim() que posibilita asignar efectivamente la memoria necesaria:

```
leer (n)
dim (v, n)
...
```

Esta posibilidad existe también para las matrices, sin importar la cantidad de dimensiones. Por ejemplo, lo que sigue declararía una matriz en las que la cantidad de filas y columnas no son conocidas al momento de la compilación:

```
M : matriz [*, *] numerico

inicio
   leer (cant_fil, cant_col)
   dim (M, cant_fil, cant_col)
   ...
```

> *A los vectores y matrices creables dinámicamente los llamamos genéricamente **"arreglos abiertos**". Como veremos en capítulos siguientes, SL aprovecha la disponibilidad de arreglos abiertos para facilitar la escritura de algoritmos más generales.*

Combinaciones interesantes de arreglos abiertos y de tamaño fijo

La definición de un arreglo puede incluir una mezcla de tamaño fijo con tamaño abierto. Por ejemplo, si al momento de escribir un algoritmo sabemos que requerimos una matriz que tiene 8 columnas pero no está definida la cantidad de filas, podemos escribir:

```
   M : matriz [*, 8] numerico

inicio
   leer (cant_fil)
   dim (M, cant_fil)
   ...
```

En este caso dim() solo requiere dos parámetros: la matriz M y la cantidad de filas. La cantidad de columnas es tomada de la definición.

Sin embargo debemos recordar que, una vez especificado un valor concreto para una dimensión, ya no es posible usar asterisco para las siguientes dimensiones. Por lo tanto lo que sigue dará error de compilación:

```
M : matriz [5, *] numerico          // !!! No es válido !!!
```

> *Si un programa intenta usar un arreglo abierto no inicializado, se genera un error de ejecución y el programa para.*

Inicialización de arreglos con literales estructurados

Los elementos de un arreglo pueden ser inicializados completamente con una sola sentencia. Dependiendo de que el arreglo tenga tamaño fijo o sea abierto, las opciones varían ligeramente.

> *La inicialización de arreglos es naturalmente una asignación. Como tal, el valor previo del arreglo se pierde completamente.*

A continuación unos ejemplos:

a. Arreglos de tamaño fijo

```
var
   tope_meses : vector [12] numerico
   dias_sem   : vector [7] cadena
   mat        : matriz [5, 3] numerico

inicio


   tope_meses = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
   dias_sem   = {"lunes",   "martes", "miercoles", "jueves",
                 "viernes", "sabado", "domingo"
                }
```

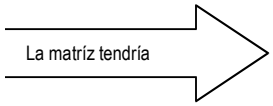
```
        mat       = { { 1,  2,  3},
                      { 4,  5,  6},
                      { 7,  8,  9},
                      {10, 11, 12},
                      {13, 14, 15}
                    }
    fin
```

Si a partir de cierto punto el resto de los elementos de un arreglo tiene el mismo valor, puede utilizarse la siguiente notación:

```
var
   vec : vector [100] numerico
inicio
   vec = {120, 40, 73, 5, 0, ... } // todos cero desde el sexto elemento.
```

La matriz mat del ejemplo anterior podría inicializarse con:

```
mat       = { { 1,  2,  3},
              { 4,  5,  6},
              { 7,  8,  9},
              ...
            }
```



1	2	3
4	5	6
7	8	9
7	8	9
7	8	9

Es decir, una asignación como la que sigue puede inicializar con ceros un vector numérico de tamaño fijo, independientemente de cuántos elementos tenga:

```
v = {0, ...}
```

b. Arreglos abiertos

Los arreglos abiertos también pueden ser inicializados con literales estructurados:

```
var
   v : vector [*] numerico
inicio
   v = {10, 20, 30, 40, 50}
   ...
```

> *Cuando un arreglo abierto es inicializado de esta forma, no es necesario el uso de dim(). Por otra parte, con los arreglos abiertos no es posible el uso de la opción de autocompletado (los tres puntos seguidos) como componente de inicialización.*

Un arreglo abierto, independientemente de la cantidad de dimensiones, puede volver al estado de "no inicializado" usando el inicializador {}:

```
var
   v : vector [*] numerico
   m : matriz [*, *] numerico

inicio
   dim (v, 10)
   dim (m, 5, 3)

   /*
`     Tanto v como m ya estan inicializadas. Ahora eliminemos los elementos
      de ambos arreglos.
   */

   v = {}
   m = {}

   /*
      Vuelven a estar como si nunca hayan sido inicializados.
   */
fin
```

Arreglos con contornos no regulares

Dado que en SL una matriz es un vector de vectores, es posible tener arreglos en los que las filas pueden tener longitudes diferentes unas de otras. El siguiente programa crea un arreglo como el que se muestra gráficamente a continuación:

```
var
   M : matriz [*, *] numerico
   f, c : numerico

inicio
   M = { { 1,  3, 13, 31},
         { 7, 21, 5,  17, 19, 2},
         {},
         { 71, 23}
       }
fin
```

1	3	13	31		
7	21	5	17	19	2
71	23				

Asignación entre arreglos

Un arreglo puede ser asignado a otro con una sola sentencia de asignación, es decir, no se requiere asignar elemento a elemento. A continuación algunos fragmentos de código que muestran su uso:

```
var
   M1 : matriz [5, 3] numerico
   M2 : matriz [*, *] numerico
   V1 : vector [3] numerico
   V2 : vector [*] numerico
inicio
   M1 = { {2, 11, 43},
          {1, 21, 14},
          {4,  1, 27},
          {5, 45, 89},
          {3, 19, 42}
        }

   M2 = M1

   V1 = M1 [2]             // {1, 21, 14}
   V2 = M1 [3]             // {4, 1, 27}

   V2 = V1                 // V2 = {1, 21, 14}
   ...
```

Para que dos arreglos sean asignables los tipos de sus elementos deben ser compatibles. Además debe cumplirse que:

- El destino (el que está a la izquierda del signo de asignación) tiene la misma cantidad de elementos que el origen, ó
- El destino es un arreglo abierto.

Un arreglo abierto no puede ser asignado a uno de tamaño fijo. Si el programa está en ejecución y SL detecta que esto ocurriría, se despliega un mensaje de error y la ejecución se cancela.

Inicialización de registros con literales estructurados

Al igual que los arreglos, los registros pueden ser inicializados con literales estructurados. Algunos ejemplos son:

```
var
   r : registro
       {
          nombre     : cadena
          fecha_nac  : registro
                       {
                          dia, mes, año : numerico
                       }
          notas      : vector [*] numerico
       }
inicio
   r = { "Carolina Gómez",    // campo nombre
         {12, 07, 1969},      // campo fecha
         {87, 91, 99, 93}     // 4 notas
       }
   ...
```

Asignación entre registros

También es posible asignar una variable de tipo registro a otra variable coincidente en tipo. En este caso los valores de todos los campos (incluyendo los que sean registros, vectores, etc.) serán copiados sobre los campos de la variable de tipo registro que aparezca a la izquierda de la asignación.

Para que dos registros sean asignables directamente se deben cumplir las siguientes condiciones:

- Ambos registros deben poseer igual cantidad de campos.
- Cada campo del primer registro debe coincidir en características (tipo básico, si el campo fuese simple; cantidad de dimensiones y tipo de sus elementos, si fuese arreglo, etc.) con su correspondiente en la secuencia de declaración del segundo registro.

Los nombres de los campos no son tenidos en cuenta al momento de verificar si dos registros son asignables. En el fragmento de código que

sigue se muestra la definición de dos variables de tipo registro, que luego son usadas en una asignación en el cuerpo del programa:

```
var
   a   : registro
         {
            x, y : numerico
            v    : vector [*] numerico
         }
   b   : registro
         {
            m, n : numerico
            c    : vector [100] numerico
         }
inicio
   b = { 10, 20,      // corresponden a m y n respectivamente
         {0, ...}     // vector c inicializado con ceros

   a = b
   . . .
```

Arreglos de registros

Los elementos de un arreglo pueden ser de cualquier tipo de dato. A continuación veremos con más detalle cómo se usan los arreglos cuando sus elementos son registros:

```
var
      v : vector [100] registro
          {
             a, b : numerico
          }
```

Por lo tanto, para asignar un valor al campo b del tercer elemento del vector v se usará:

```
v [3].b = 35
```

A continuación vemos cómo puede definirse un vector en el que cada elemento guarde datos generales de un alumno (nombre y número de cédula) y sus notas para cierta asignatura. La cantidad de alumnos por curso no es fijo:

```
var
   A : vector [*] registro
       {
          nombre : cadena;
          cedula : numerico
          notas  : vector [*] numerico
       }
```

Luego podemos inicializar el vector A, por ejemplo, para 25 alumnos:

```
dim (A, 25)
```

Podemos también, por ejemplo, inicializar el vector A con los datos para 3 alumnos:

```
A = { {"Mirta", 12781891, {78, 96, 100}},     // alumno no. 1
      {"Jose",  14271843, {91, 68, 83} },     // alumno no. 2
      {"Luisa", 13238713, {61, 72, 78} }
    }
...
```

Cantidad de elementos de un arreglo abierto: Uso de la función alen()

Para obtener la cantidad de elementos de un arreglo podemos usar la función predefinida alen(), que opera tanto sobre arreglos abiertos como sobre los que tienen tamaño fijo.

El siguiente programa hace uso de alen() para recorrer dos vectores:

```
programa uso_alen
var
   v1 : vector [10] numerico
   v2 : vector [*] numerico

var
   k  : numerico

inicio

   desde k=1 hasta alen (V1)
   {
      V1 [k] = 0
   }
```

```
    dim (V2, 15)
    desde k=1 hasta alen (V2)
    {
       V2 [k] = 0
    }
fin
```

La función alen() retorna 0 cuando el arreglo abierto no fue inicializado aún.

Capítulo

9

Operaciones con cadenas

Como ya hemos visto, una de las operaciones básicas con cadenas es la concatenación. Existen además otras operaciones y funciones aplicables a ellas que son de utilidad en muchas aplicaciones. Las principales son el acceso a los caracteres y las conversiones de tipos de datos.

Longitud de una cadena: la función strlen()

La longitud de una cadena puede determinarse con la función predefinida strlen(). Por ejemplo,

```
var
   s : cadena
   g : numerico
inicio
   s = "prueba"
   g = strlen (s)
   imprimir ("\n", g)
   imprimir ("\nLongitud de cadena vacia es ", strlen (""))
fin
```

imprimirá

```
6
Longitud de cadena vacia es 0
```

Acceso a los caracteres de una cadena

Una cadena puede utilizarse como si fuese un vector: cada elemento es un carácter y puede ser referenciado por su posición dentro del conjunto. El siguiente programa imprime los dígitos del '9' al '0'.

```
   /*
      OBJETIVO: Mostrar en la pantalla los digitos del '9' al '0' (es decir,
                en orden inverso) cada uno en una linea independiente.
      AUTOR: jsegovia
      FECHA: 19-dic-1999
   */
var
   dig : cadena
   k : numerico
inicio
   dig = "0123456789"
   desde k=strlen (dig) hasta 1 paso -1
   {
      imprimir ("\n", dig [k])
   }
fin
```

> *A diferencia de los vectores, solicitar un carácter que no existe (por ejemplo, pedir el que está en la posición cero) simplemente retorna la cadena vacía y no genera error de ejecución.*

Modificación de caracteres de una cadena

Los caracteres de una cadena pueden ser también modificados, es decir, pueden ser usados a la izquierda de una asignación. El siguiente ejemplo reemplaza todas las letras 'a' de una cadena por la letra 'b'.

```
/*
   OBJETIVO: Reemplazar todas las letras 'a' de una cadena por la por letra
'b'.
   AUTOR: jsegovia
*/
var
   z : cadena

inicio
   z = "Esta es una prueba"
   desde k=1 hasta strlen (z)
   {
      si ( z [k] == 'a' )
      {
         z [k] = 'b'
```

```
      }
   }
   imprimir ("\n", z)
fin
```

El programa imprime

```
Estb es unb pruebb
```

> *A diferencia de los vectores, la solicitud de asignación a una posición que no existe (por ejemplo, el que está en la posición 21 cuando la cadena tiene solo 20 caracteres) es completamente ignorada y no genera error de ejecución.*

Subcadenas

Para tomar (copiar) una subcadena podemos usar la función predefinida substr(), que tiene el siguiente formato:

```
subrutina substr (z : cadena; p_inicial, cant : numerico) retorna cadena
```

donde

```
z         : cadena que proveerá los caracteres para la subcadena.
p_inicial : posición del primer carácter que se extraerá.
cant      : cantidad de caracteres a tomar. Este parámetro puede omitirse,
            en cuyo caso se toman todos los caracteres desde el inicial
            hasta el final de z.
```

- Cadena z queda inalterada.
- Si p_inicial es mayor a la longitud de la cadena, substr() retorna una cadena vacía.

El siguiente programa imprime:

```
1
23
456
789A
BCDEF
```

```
/*
   OBJETIVO: Mostrar el uso de subcadenas.
   AUTOR: jsegovia
*/

var
   z, subcad : cadena
   g, k, p : numerico
```

```
inicio
   z = "123456789ABCDEF"
   g = strlen (z)
   p = 1
   k = 1

   mientras (p < g )
   {
      subcad = substr (z, p, k)
      imprimir ("\n", subcad)
      p = p + k
      k = k + 1
   }
fin
```

Conversión a numérico: la función val()

El valor numérico de una cadena puede obtenerse usando la función predefinida val(), toda vez que dicha cadena contenga la secuencia válida de caracteres para un número.

Ejemplo:

```
var
   n : numerico
   z : cadena
inicio
   z = "300"
   n = val (z) + val (substr ("415", 2, 1))   // n valdrá 301
   ...
```

La función val() retorna cero cuando recibe una cadena que no puede convertirse en número.

Capítulo

10

Subrutinas

Las subrutinas son un componente fundamental de programación: ellas permiten que un problema complejo sea dividido en partes más pequeñas, comprensibles y manejables. Podemos conceptualizarla como un completo subprograma (programa más pequeño) porque ella no solo agrupa y da un nombre a un conjunto de sentencias lógicamente relacionadas, sino que además puede tener sus propias variables, constantes y otros elementos.

En SL las subrutinas se escriben a continuación del programa principal. La invocación de una subrutina se realiza escribiendo su nombre en los puntos del programa donde su acción sea requerida.

Dependiendo de su función dentro del programa, existen dos tipos de subrutinas: los procedimientos y las funciones. Ambas veremos con más en detalle en este capítulo.

Ámbito de los identificadores

Decimos que un identificador es *global* si fue declarado al inicio del programa principal. En cambio, decimos que es *local* si fue declarado dentro de alguna subrutina o es un parámetro formal (Ver más adelante el apartado "Paso de parámetros").

La importancia de esta distinción radica en que:

- los identificadores locales solo pueden ser utilizados (solo están "visibles") dentro de la subrutina que contiene su definición.
- los identificadores globales pueden ser utilizados en cualquier parte del programa.

Corresponden al primer grupo (global) los que son declarados al inicio del programa principal, tal como se muestra en "Estructura general de un programa en SL".

Los considerados identificadores locales son aquellos que son declarados dentro de subrutinas o utilizados como parámetros formales de éstas y solo pueden ser utilizados dentro de dichas subrutinas, a diferencia de los globales, que pueden ser utilizados en cualquier parte del programa.

Si un identificador es declarado o forma parte de los parámetros formales de una subrutina y su nombre coincide con el de otro identificador declarado globalmente, éste último queda temporalmente inaccesible durante la ejecución de la subrutina.

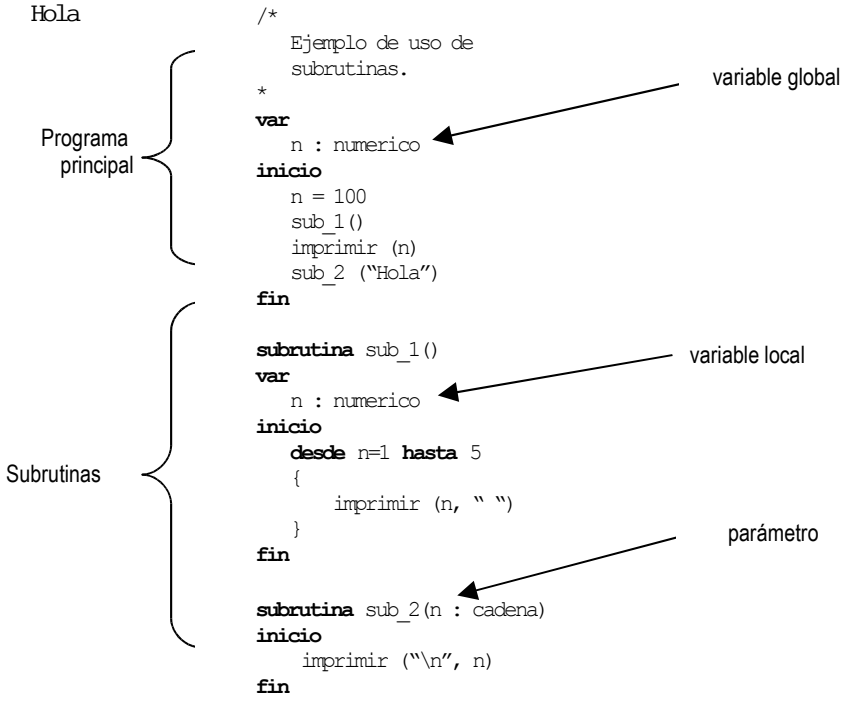
La existencia de identificadores locales posibilita:

- que dos identificadores puedan ser definidos con el mismo nombre, pero cada uno teniendo propiedades independientes: dirección de memoria, tipo, duración (constante, variable).
- mejorar la legibilidad de las subrutinas y fortalecer su independencia del contexto.

- la escritura de subrutinas recursivas.

El programa que sigue muestra en la pantalla lo que sigue:

```
1 2 3 4 5
100
Hola
```




```
/*
   Ejemplo de uso de
   subrutinas.
*
var
   n : numerico
inicio
   n = 100
   sub_1()
   imprimir (n)
   sub_2 ("Hola")
fin

subrutina sub_1()
var
   n : numerico
inicio
   desde n=1 hasta 5
   {
       imprimir (n, " ")
   }
fin

subrutina sub_2(n : cadena)
inicio
    imprimir ("\n", n)
fin
```

La primera línea (1 2 3 4 5) es impresa por la subrutina sub_1(); la segunda (100) es generada por el programa principal mientras que la tercera (Hola) imprime la subrutina sub_2(). Como puede apreciarse, el valor de la variable global n no fue afectada por la variable local n de la subrutina sub_1() pues el compilador les asigna espacios de memoria distintos.

Tipos de subrutinas

SL soporta dos tipos de subrutinas:

- Funciones: son las que retornan un valor y pueden utilizarse como operandos en expresiones.

- Procedimientos: las que no retornan un valor, y por lo tanto no pueden usarse como parte de expresiones.

A continuación analizamos por separado estas dos formas de subrutinas.

Funciones: Subrutinas que retornan valor

Una función es una subrutina que produce un valor que puede ser utilizado por la parte del programa que la llamó.

El valor producido y retornado por la función puede ser de cualquier tipo: cadena, numérico, lógico, arreglos con cualquier dimensión y elementos, registros, etc.

Como ejemplo, a continuación tenemos un sencillo programa que incluye una subrutina para obtener la parte entera de un número positivo.

```
/*
   OBJETIVO: Obtener la parte entera de un número real. Para el efecto
             se implementa la función entero().
   AUTOR:    jsegovia
*/

var
   x : numerico
inicio
   x = entero (15.23)                          // x vale 15
   imprimir (x)
   x = entero (x * 0.5) + entero (x)           // x vale 22
   imprimir (x)
   imprimir ( entero (19.45) )          // imprime 19
fin


subrutina entero (n : numerico) retorna numerico
/*
   OBJETIVO:
      Calcular la parte entera de un numero positivo.
      Se usa el método de las restas sucesivas.

   PARAMETRO:
      n : número del que se quiere obtener su parte entera.
*/
```

```
var
   ent : numerico
inicio
   ent = 0
   mientras ( n >= 1 )
   {
      ent = ent + 1
      n = n - 1
   }
   retorna ( ent )
fin
```

El valor de retorno de una función puede ser descartado. Como ejemplo vea la función predefinida inc().

Procedimientos: subrutinas que no retornan valor

Un ejemplo de subrutina que no retorna valor puede encontrarse en el ítem "Sentencia de llamada a subrutinas".

Definición de parámetros formales

Llamamos *parámetros formales* a los identificadores que ubicamos al inicio de una subrutina para indicar la cantidad y tipo de cada uno de los datos que la subrutina recibe. En la siguiente definición

```
subrutina max_comun_div (a, b : numerico) retorna numerico
```

a y b son los parámetros formales, ambos numéricos en este caso.
La sintaxis para la definición de parámetros formales es similar la declaración de variables.

La siguiente podría ser la definición de una subrutina que inserta espacios en una línea de tal forma que toda ella ocupe exactamente cierta cantidad de caracteres:

```
subrutina ajustar (linea : cadena; ancho : numerico) retorna cadena
```

Paso de parámetros

Existen dos formas de pasar parámetros: por valor y por referencia. Ambos veremos a continuación.

Por valor

La expresión que se pasa como parámetro es copiada como valor inicial de la variable correspondiente en la lista de parámetros formales de la subrutina.

Lo que caracteriza al paso de parámetros por valor es que los cambios que se hagan a la variable que recibe el parámetro no se ven reflejados en el programa que llama a la subrutina.

A continuación un ejemplo:

```
/*
   OBJETIVO: Mostrar el paso de parámetros por valor.
   AUTOR: jsegovia
*/

var
   a, b : numerico

inicio
   a = 1; b = 10
   mi_sub (a, b)
   imprimir ("\n", a, ´ ´, b)
   mi_sub (a*10, b-1)
fin


subrutina mi_sub (a, b : numerico)
inicio
   a = 5
   b = b * 5
   imprimir ( "\n", a, ´ ´, b)
fin
```

El programa anterior imprime:

```
5 50
1 10
5 45
```

Obsérvese que los cambios que se hicieron a las variables locales a y b, parámetros formales de la subrutina mi_sub(), no afectaron a las variables globales con el mismo nombre.

Por referencia

Cuando una variable se pasa por referencia, lo que en realidad se recibe en la subrutina es la dirección de memoria de la variable original. Por ello, cualquier cambio que se haga al parámetro, se ve reflejado en la variable original.

A continuación un ejemplo:

```
/*
   OBJETIVO: Mostrar el paso de parámetros por referencia.
   AUTOR: jsegovia
*/

var
   a, b : numerico

inicio
   a = 1; b = 10
   mi_sub (a, b)
   imprimir ("\n", a, ' ', b)
   mi_sub (a*10, b)
fin



subrutina mi_sub (a : numerico; ref b : numerico)
/*
   a se recibe por valor, pero b por referencia.
*/
inicio
   a = 5
   b = b * 5
   imprimir ( "\n", a, ' ', b)
fin
```

El programa anterior imprime:

```
5 50
1 50
5 250
```

Obsérvese que al cambiar el valor de la variable b en mi_sub(), también cambió el valor de b en el programa principal.

Ejemplo

El siguiente programa muestra el uso del concepto del paso de parámetros por referencia para escribir una rutina general para intercambiar el contenido de dos variables numéricas.

```
/*
   OBJETIVO: Mostrar el uso de los parámetros por referencia, implementando
             una subrutina que intercambia el contenido de dos
             variables numéricas.
   AUTOR: jsegovia
*/

var
   a, b : numerico
inicio
   a = 10; b = 20
   intercambiar_num (a, b)
   imprimir (a,' ', b)          // imprimira 20 10
fin

subrutina intercambiar_num (ref primera_var, segunda_var : numerico)
var
   aux : numerico
inicio
   aux = primera_var
   primera_var = segunda_var
   segunda_var = aux
fin
```

Como puede observarse, el nombre de la variable que recibe el dato que se pasa como parámetro no tiene porqué coincidir con el nombre la variable que se pasa como parámetro.

Subrutinas y arreglos abiertos

El siguiente programa muestra el uso de arreglos abiertos como parámetros de subrutinas:

Primer ejemplo

```
/*
   OBJETIVO: Mostrar el uso de arreglos abiertos como parámetros de subrutinas.
   AUTOR: jsegovia
*/

var
   A : vector [5] numerico
   B : vector [*] numerico
inicio
   A = {1, 2, 3, 5, 7}
   B = A
   impr_vect (A)
   impr_vect (B)
   impr_vect ({100, 200, 300})
fin

subrutina impr_vect (v : vector [*] numerico)
var
   k : numerico
inicio
   desde k=1 hasta alen (v)
   {
      imprimir (v [k], " ")
   }
fin
```

Segundo ejemplo

El siguiente programa desarrolla una subrutina que produce la transpuesta de una matriz dada. A continuación se muestra lo que produce:

```
Matriz original
10 11 12
20 21 22
30 31 32
40 41 42
50 51 52

Matriz traspuesta
10 20 30 40 50
11 21 31 41 51
12 22 32 42 52
```

```
/*
   OBJETIVO: Mostrar el uso de arreglos abiertos a través de una función de
             que calcula la traspuesta de una matriz.
   AUTOR: jsegovia
*/
```

```
var
   A , T : matriz [*,*] numerico
   f, c : numerico
inicio
   A = {{10, 11, 12},
        {20, 21, 22},
        {30, 31, 32},
        {40, 41, 42},
        {50, 51, 52}
       }
   imprimir ("\nMatriz original\n")
   impr_mat (A)
   T = trasp (A)
   imprimir ("\nMatriz traspuesta\n")
   impr_mat (T)
fin


subrutina trasp (m : matriz [*,*] numerico) retorna matriz [*,*] numerico
   /*
      OBJETIVO: Producir la traspuesta de la matriz m.

      Se asume que m tiene igual cantidad de columnas en todas sus filas,
      es decir, tiene "contorno" regular.
   */
var
   t : matriz [*,*] numerico
   cf, cc : numerico       // cantidad de filas y columnas de m
   kf, kc : numerico       // indice de fila y columna

inicio
   cf = alen (m [1])
   cc = alen (m)
   dim (t, cf, cc)

   desde kf=1 hasta cf
   {
      desde kc=1 hasta cc
      {
         t [kf, kc] = m [kc, kf]
      }
   }
   retorna ( t )
fin
```

```
subrutina impr_mat (m : matriz [*,*] numerico)
var
   f, c : numerico
inicio
   desde f=1 hasta alen (m)
   {
      desde c=1 hasta alen (m[f])
      {
         imprimir (m [f, c], ' ')
      }
      imprimir ("\n")
   }
fin
```

Capítulo

11

Definición de nombres de tipos de datos

SL proporciona un mecanismo para crear nuevos nombres de tipos de datos. Por ejemplo, el siguiente fragmento define un tipo de dato "PRODUCTO" y declara algunas variables de este tipo:

```
tipos
   PRODUCTO : registro
            {
               codigo  : numerico
               descrip : cadena
               precio  : numerico
            }
var
   monitor, teclado : PRODUCTO
   lista_prod       : vector [*] PRODUCTO
```

Es decir, una vez definido el nombre PRODUCTO, podemos usarlo en cualquier contexto donde SL requiera un tipo de dato.

Los nombres de tipos de datos definidos por el programador pueden ser útiles para clarificar y/o simplificar los programas y facilitar su posterior mantenimiento.

Las reglas para nombrar tipos de datos son las mismas ya vistas para variables, constantes y demás identificadores.

Definición de alias

El mecanismo de definición de nombres de tipos de datos puede ser utilizado para introducir alias a nombres de tipos ya existentes. Por ejemplo, podemos definir

```
tipos
   TLineaTexto : cadena
```

y luego

```
var
   primera_linea, segunda_linea : TLineaTexto
```

en cuyo caso ambas variables son en realidad cadenas, y por lo tanto podrán usarse libremente en todo contexto donde se espere una cadena.

Ejemplos

A continuación veremos dos ejemplos que hacen uso de la definición de nombres de tipos de datos.

Primer ejemplo

El objetivo del siguiente programa es leer los nombres y las notas de los alumnos de cierta asignatura e imprimir una lista ordenada por las notas, de mayor puntaje a menor puntaje. La cantidad de alumnos debe ser ingresada por el usuario.

El programa define un tipo llamado “ALUMNO” que agrupa en un registro el nombre y la nota de cada alumno. Define también el tipo de dato “ACTA”, que es un vector abierto de “ALUMNO”.
A continuación el programa completo. Observemos especialmente los puntos indicados con recuadros.

```
programa uso_de_nuevos_tipos_1

tipos
   ALUMNO  : registro
             {
                 nombre : cadena
                 nota   : numerico
             }
   ACTA    : vector [*] ALUMNO
var
   A : ACTA

inicio
   leer_acta (A)
     ordenar_por_nota (A)
   imprimir_acta (A)
fin


subrutina imprimir_acta (A : CLASE)
/*
      Imprimir el nombre y la nota de cada alumno.
*/
var
   k : numerico
inicio
   desde k=1 hasta alen (A)
   {
      imprimir ("\n", A [k].nombre, "\t", A [k].nota)
   }
fin


subrutina leer_acta  (ref c : ACTA)
 /*
    Leer los nombres y notas de los alumnos.
    Primero pide cantidad de alumnos e inicializa el vector de
    acuerdo a esto.

    Observar que c DEBE ser recibido por referencia, pues de
    lo contrario los datos leidos se perderan al salir de la subrutina.
 */

var
   cant, k : numerico

inicio
   imprimir ("\nIngrese cantidad de alumnos: ")
   leer (cant)
   dim (c, cant)
```

```
   imprimir ("\nA continuacion tipee nombre, nota para cada alumno\n")

   desde k=1 hasta cant
   {
      leer ( c [k].nombre, c [k].nota )
   }
fin


subrutina ordenar_por_nota  (ref A : ACTA)
   /*
      Ordernar A, considerando las notas, de mayor a menor. El
      algoritmo es el de la burbuja.

      El parametro A DEBE ser recibido por referencia, pues de
      lo contrario los datos leidos se perderan al salir de la subrutina.

      Muestra el uso de asignacion entre registros (variable aux).
   */

var
   aux  : ALUMNO
   k, n : numerico
   g : numerico   // longitud de A
inicio
   g = alen (A)
   desde n=1 hasta (g - 1)
   {
         desde k=n+1 hasta g
         {
            si ( A [n].nota < A [k].nota  )
            {
               aux  = A [n]
               A [n] = A [k]
               A [k] = aux
            }
         }
   }
fin
```

Segundo ejemplo

El siguiente programa imprime el día de la semana que corresponde a una fecha dada. Define un nombre de tipo de dato "FECHA".

```
/*
   OBJETIVO: Mostrar el uso de tipos definidos por el usuario.
   AUTOR: jsegovia
*/
```

```
tipos
   FECHA : registro
           {
              d, m, a : numerico
           }
var
   f : FECHA

inicio
   imprimir ("\n", strdup ("-", 79))
   imprimir ("\nEste programa calcula el dia de la semana que corresponde ",
             "a una fecha\n")
   f=leer_fecha ("\nIngrese una fecha (dd,mm,aaaa):")



   si ( fecha_valida (f) )
   {
      imprimir ("\nDia=", nombre_dia_sem (calc_dia_sem (f)))
   sino
      imprimir ("\nFecha ingresada no es valida\n")
   }
fin

subrutina leer_fecha (msg : cadena) retorna FECHA
var
   f : FECHA

inicio
   imprimir (msg)
   leer (f.d, f.m, f.a)
   retorna (f)
fin

subrutina fecha_valida (f : FECHA) retorna logico
var
   mal : logico
inicio
   mal = (f.a < 1)  or (f.m < 0  or f.m > 12) or (f.d < 1  or f.d > 31)

   si ( not mal )
   {
      si ( f.m == 2 )
      {
         mal = f.d > 28 and not bisiesto (f.a)
      sino
         mal = (f.m == 4 or f.m == 6 or f.m == 9 or f.m == 11) and f.m > 30
      }
   }
   retorna ( not mal )
fin
```

```
subrutina calc_dia_sem (f : FECHA) retorna numerico
var
   d, m, y1, y2 : numerico
inicio

   si ( f.m < 3 )
   {
      m = f.m + 10;
      f.a = f.a - 1
   sino
      m = f.m - 2;
   }

   y1 = int (f.a / 100);
   y2 = f.a % 100;

   d =  int ( (
              f.d + int (2.6*m - 0.1) + y2 + int (y2 / 4)
                  + int (y1 / 4) - 2*y1 + 49
            ) % 7
          ) + 1;

   retorna (d)
fin

subrutina bisiesto (a : numerico) retorna logico
inicio
   retorna (a % 4 == 0) and not ((a % 100 == 0) or (a % 400 == 0))
fin

subrutina nombre_dia_sem (d : numerico) retorna cadena
var
   dsem : vector [8] cadena
inicio
   dsem = {"domingo", "lunes", "martes", "miércoles", "jueves",
           "viernes", "sábado", "**inválido**" }
   si ( d < 1 or d > 7 )
   {
      d = 8
   }
   retorna ( dsem [d] )
fin
```

Anexo A. Ejemplos Selectos

Serie de Fibonacci

Imprime los primeros n términos de la serie de Fibonacci. El valor de n es proveído por el usuario.

```
programa fibo
var
   a, b, c : numerico
   n : numerico;
inicio
   leer (n);
   a = 0; b = 1;
   si ( n >= 1 )
   {
      imprimir ("\n", a);
   }
   si ( n >= 2 )
   {
      imprimir ("\n", b);
   }
   n = n - 2;
   mientras ( n >= 1 )
   {
      c = a + b;
      imprimir ("\n", c);
      a = b;
      b = c;
      n = n - 1;
   }
fin
```

Conversión de decimal a binario (usando restas sucesivas)

Dado un número entero positivo en base 10, imprimirlo en base 2.
El algoritmo calcula el residuo de la división entera usando restas sucesivas.
El número a convertir es proveído por el usuario.

```
programa conv_bin
var
   nDec, nBin, pot, resto, cociente : numerico
inicio
   nBin = 0
   pot = 0

   imprimir ("\nIngrese un numero decimal:")
   leer (nDec)

   mientras ( nDec > 0 )
   {
      cociente = 0
      resto = nDec
      mientras ( resto > 2 )
      {
          resto = resto - 2
          cociente = cociente + 1
      }
      nDec = cociente
      si ( resto == 1 )
      {
          nBin = nBin + 10^pot
      }
      pot = pot + 1
   }
   imprimir (nBin)
fin
```

Conversión a hexadecimal

Dado un número entero positivo en base 10, imprimirlo en base hexadecimal. La implementación es particularmente interesante porque utiliza constantes locales, funciones, acceso directo a caracteres de cadenas y operador de módulo.

```
var
   n : numerico
   h : cadena
   sl : numerico
inicio
   imprimir ("\nIngrese un numero entero positivo:")
   leer (n)
   imprimir ("\n", n, " en hex es ", dec_a_hex (n))
fin
```

```
subrutina dec_a_hex (n : numerico) retorna cadena
const
   HEX_DIG = "0123456789ABCDEF"
var
   s : cadena
   r : numerico
inicio
   mientras ( n >= 16 )
   {
      r = n % 16
      s = HEX_DIG [r+1] + s
      n = int (n / 16 )
   }
   s = HEX_DIG [n+1] + s
   retorna ( s )
fin
```

Tabla ASCII

Imprimir la tabla ASCII para los caracteres ubicados desde la posición 32 en adelante.

El programa utiliza la función ascii() para generar los caracteres.

```
const
   inicio_tabla = 32;
   fin_tabla    = 255

   INCR         = 16
   RELLENO      = ' '

var
   k, tope : numerico

inicio
   k = inicio_tabla
   cls()

   repetir
      tope = k + INCR
      si ( tope > fin_tabla )
      {
         tope = fin_tabla
      }
      prt_ascii (k, tope, RELLENO);
      imprimir ("\n")
      k = tope + 1
   hasta ( tope == fin_tabla )
fin
```

```
subrutina prt_ascii (v1, v2 : numerico; r : cadena)
var
   k : numerico
inicio
   desde k=v1 hasta v2
   {
      imprimir ( ascii (k), r)
   }
fin
```

Números Primos

Imprimir todos los números primos comprendidos entre 1 y un número dado.

Se utiliza un vector de valores lógicos para ir "tachando" los no primos.

```
programa NumerosPrimos
const
     TACHADO     = TRUE
     NO_TACHADO  = FALSE
tipos
   VEC_PRIMOS : vector [*] logico
var
  N : numerico
  P : VEC_PRIMOS
inicio
  imprimir ("\n=== CALCULO DE NUMEROS PRIMOS ===")

  si ( pcount() < 1 )
  {
     imprimir ("\nIngrese último número entero a evaluar: ");
     leer (N)
  sino
     N = val (paramval (1))
  }

  si ( int (N) <> N )
  {
     imprimir ("\nEl numero ingresado debe ser entero")
  sino
     dim (P, N)
     calc_primos (P, N)
     imprimir ("\n")
     imprimir_primos (P)
  }
fin
```

```
subrutina calc_primos (ref P : VEC_PRIMOS; N : numerico)
var
  k, n : numerico
inicio
   desde k = 1 hasta N
   {
      P [k] = NO_TACHADO
   }
   n = 2
   mientras ( n + n < N )
   {
      desde k = n + n hasta N paso n
      {
        P [k] = TACHADO
      }
      n = n + 1
      mientras ( n <= N and P [n] )
      {
         n = n + 1
      }
   }
fin

subrutina imprimir_primos (v : VEC_PRIMOS)
var
   k : numerico
inicio
   desde k = 2 hasta alen (v)
   {
      si ( v [k] == NO_TACHADO)
      {
         imprimir (str (k, 5, 0) )
      }
   }
fin
```

Movimientos del alfil en un tablero de ajedrez

Usando una matriz, genera e imprime los posibles movimientos de un alfil a partir de la fila y columna iniciales proveídas por el usuario.

```
programa alfil
const
   TAM_TAB = 8
var
   f_inic, c_inic  : numerico
   T : vector [TAM_TAB,TAM_TAB] cadena
```

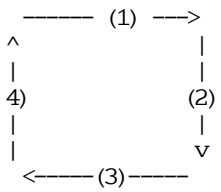
```
var
   k : numerico
inicio
  imprimir("\nIngrese fila y columna donde se encuentra el alfil:")
  leer (f_inic, c_inic)

  T = { {'.', ...},
        ...
      }

  T [f_inic, c_inic] = 'A'
  k = 1

  mientras ( k <= TAM_TAB )
  {
     marcar (f_inic - k, c_inic - k)
     marcar (f_inic - k, c_inic + k)
     marcar (f_inic + k, c_inic - k)
     marcar (f_inic + k, c_inic + k)
     k = k + 1
  }
  impr_tablero()
fin


subrutina marcar (f, c : numerico)
inicio
    si ( (f > 0 and f <= TAM_TAB) and
         (c > 0 and c <= TAM_TAB)
       )
    {
       T [f, c] = 'P'
    }
fin

subrutina impr_tablero()
var
   f, c : numerico
inicio
  desde f = 1 hasta TAM_TAB
  {
    imprimir ("\n")


    desde c = 1 hasta TAM_TAB
    {
      imprimir ( T [f, c], ' ')
    }
  }
fin
```

Matriz con valores en caracol

Completar una matriz como se muestra a continuación:

1	2	3	4	5
12	13	14	15	6
11	10	9	8	7

Para recorrer la matriz usamos un indicador de dirección como se indica a continuación:




```
programa caracol
const
   MAX_FIL = 3
   MAX_COL = 5

var
  mat : vector [MAX_FIL, MAX_COL] numerico
  k, dir        : numerico
  mincol, maxcol : numerico
  minfil, maxfil : numerico
  fil, col      : numerico

inicio
  minfil = 1
  mincol = 1
  maxcol = MAX_COL
  maxfil = MAX_FIL
  col = 1; fil = 1; dir = 1

  desde k=1 hasta MAX_FIL*MAX_COL
  {
     mat [fil, col] = k
     eval
     {
        caso ( dir == 1  )
             inc (col)
             si ( col == maxcol )
             {
                dir = 2
```

```
                inc (minfil)
            }
        caso ( dir == 2 )
            inc (fil)
            si ( fil == maxfil )
            {
                dir = 3
                dec (maxcol)
            }
        caso ( dir == 3 )
            dec (col)
            si ( col == mincol )
            {
                dir = 4
                dec (maxfil)
            }
        caso ( dir == 4 )
            dec (fil)
            si ( fil == minfil )
            {
                dir = 1
                inc (mincol)
            }
    }
  }
  imprimir_matriz()
fin


subrutina imprimir_matriz()
var
   f, c : numerico
inicio
   cls()
   desde f=1 hasta MAX_FIL
   {
      imprimir ("\n")


      desde c=1 hasta MAX_COL
      {
         imprimir (str (mat [f,c], 4, 0))
      }
   }
fin
```

Corte de control

Los alumnos de un curso libre de inglés tienen sus notas (parciales y finales) consignadas en registros de la sgte. manera:

```
Nro. de Alumno, Codigo (1-parcial, 2-Final), Puntaje (1-100)
```

Los registros están ordenados por nro. de alumno. La cantidad de parciales que dieron varía de un alumno a otro.

Las notas parciales y la final vienen en cualquier orden, ya sean primero las parciales y luego la final o mezcladas. Existe un solo registro que contiene la nota final.

Obs.: Los puntos se calcularán de la sgte. forma:
Puntos = (40% promedio de los parciales) + (60% de examen final).

La nota final se calcula de la sgte. manera:

```
de   1 a  49% = 1
    50 a  60% = 2
    61 a  75% = 3
    76 a  94% = 4
    95 a 100% = 5
```

Se desea un programa que imprima:

a) la sgte. planilla:

Nro. Alumno	Prom.Parc.	Exam.Final	Puntos	Nota Final
999	999	999	999	999

b) El numero del "mejor alumno".
El mejor alumno se considera a aquel que hizo 70% o más en todos los parciales y el final.

Si hay más de uno, tomar el último.

Ejemplo:

```
Nro. de Alumno   - Codigo     - Puntaje
-------------------------------------------
     1              1            80
     1              2            70
     1              1            90
     2              2            100
     2              1            45
     2              1            40
     2              1            90
     9              3            100
```

Planilla a imprimir:

```
Nro. Alumno     Prom.Parc.      Exam.Final      Puntos  Nota Final
-----------     ----------      ----------      ------  ----------
1               85              70              76              4

Nro. Alumno     Prom.Parc.      Exam.Final      Puntos  Nota Final
-----------     ----------      ----------      ------  ----------
2               58.33           100             83.33           4
```

```
programa notas_finales
var
   nroAlu,
   codigo,
   puntaje : numerico

   gNroAlu,
   sumaParc,
   cantParc,
   puntExaFinal : numerico
inicio
   leer ( nroAlu, codigo, puntaje )

   gNroAlu  = nroAlu
   sumaParc = 0
   cantParc = 0


   mientras ( codigo <> 3 )
   {
      si ( gNroAlu <> nroAlu )
      {
         corte()
      }
```

```
      proceso()
      leer ( nroAlu, codigo, puntaje )
   }
   corte()
fin


subrutina proceso()
inicio
   si ( codigo == 1 )
   {
      sumaParc = sumaParc + puntaje
      cantParc = cantParc + 1
   sino
      puntExaFinal = puntaje
   }
fin


subrutina corte()
var
   puntos, notaFinal : numerico

inicio
   puntos    = calc_puntos()
   notaFinal = calc_nota_final (puntos)

   imprimir ("\nNro. Alumno      Prom.Parc.     Exam.Final    Puntos",
             "    Nota Final")
   imprimir ("\n-----------      ----------     ----------    ------",
             "    ----------")
   imprimir ("\n", gNroAlu, "    ", (sumaParc/cantParc), "    ", puntExaFinal,
             "    ", puntos, "    ", notaFinal
           )
   primerReg()
fin


subrutina calc_puntos() retorna numerico
var
   promParc, puntos : numerico
inicio
   promParc = sumaParc / cantParc
   puntos   = (promParc * 40 / 100) + (puntExaFinal * 60 / 100)
   retorna (puntos)
fin
```

```
subrutina calc_nota_final (puntos : numerico) retorna numerico
var
   nota : numerico
inicio

   si ( puntos >= 95 )
   {
      nota = 5
   sino si ( puntos >= 76 )
      nota = 4
   sino si ( puntos >= 61 )
      nota = 3
   sino si ( puntos >= 50 )
      nota = 2
   sino
      nota = 1
   }
   retorna (nota)
fin
```

CML: Intérprete para un lenguaje sencillo

CML es un intérprete para un lenguaje sencillo, con sintaxis vagamente similar a Pascal. Se desarrolló en SL con el objetivo de mostrar a alumnos de un curso introductorio de compiladores las técnicas para el análisis sintáctico descendente recursivo.

El "código" generado es para una máquina de pila. Un listado del código generado, en formato simbólico, puede obtenerse pasando -d como primer parámetro del programa.

Ejemplo de uso de CML

Lo que sigue es un ejemplo de un pequeño programa escrito en CML. Podemos probarse el funcionamiento de nuestro intérprete grabando el programa fuente en un archivo llamado fibo.cml, que luego será proveído a CML como entrada.

```
-- Programa escrito en el pequeño lenguaje CML.
-- Imprime los primeros n numeros de la serie de Fibonacci
-- Observar que esto no es SL!

var
   a=0, b=1, c;
   n=1;
begin
   read n;
   if ( n >= 1 )
   {
      print a;
   }
   if ( n >= 2 )
   {
      print b;
   }
   n = n - 2;
   while ( n >= 1 ) {
      c = a+b;
      print c;
      a=b; b=c;
      n = n - 1;
   }
end
```

Figura 4. Serie de Fibonacci, implementado en CML

Características de CML

A continuación damos un panorama general de CML.

Declaración de variables

Las variables deben ser declaradas antes de su uso. Siempre son numéricas. Al momento de declararlas se puede indicar un valor inicial, que implícitamente es 0.

Cuerpo del programa

Las sentencias ejecutables del programa van entre el par "begin"..”end".

Sentencias

Las sentencias de CML son:

- de entrada/salida: read y print. Solo pueden imprimirse valores numéricos.
- condicional: if, la condición expresada entre paréntesis.
- ciclo: while, la condición de permanencia expresada entre paréntesis.

- asignación: tiene la forma “nombre_variable = expresión”.

Expresiones

Los operadores disponibles son:

- aritméticos: suma (+), resta (-), multiplicación (*), división (/), cambio de signo (-).
- relacionales: mayor (>), mayor o igual (>=), menor (<), menor o igual (<=), igual (==), distinto (<>).

La precedencia de estos operadores es la usual. Pueden usarse paréntesis para especificar un orden de evaluación particular.

Programa fuente del intérprete

```
// [=========================================================]
// [           Elementos para el analisis lexicografico      ]
// [=========================================================]
//  (c) Juan Segovia Silvero (jsegovia@cnc.una.py)

tipos
   LEXEMA : numerico

var
   tk : LEXEMA              // sgte. lexema que debe ser analizado
   lt : cadena              // linea del texto analizado
   llt: numerico            // longitud de lt
   up : numerico            // ultima posicion leida dentro de lt

var
   val_num : numerico       // valor numerico de la constante leida
   val_time : logico
   subcad : cadena          // el componente lexico en forma de cadena.
                            // Ejemplo:
                            //     var
                            //        a, total;
                            //     tendra "a", luego la sgte. vez "total".

tipos
   INFO_PAL_RESERV : registro
                     {
                        pal  : cadena        // "if", "while", etc.
                        lex  : LEXEMA        // su valor numerico simbolico
                     }
```

```
var
   pal_reserv : vector [*] INFO_PAL_RESERV

const
   S_CONST_NUM    = 0
   S_NOMBRE_VAR   = 2

   S_MENOS        = 100
   S_MAS          = 101
   S_MULT         = 102
   S_DIV          = 103
   S_PARENT_I     = 110
   S_PARENT_D     = 111
   S_LLAVE_I      = 112
   S_LLAVE_D      = 113

   S_MENOR        = 120
   S_MENOR_IGUAL  = 121
   S_MAYOR        = 122
   S_MAYOR_IGUAL  = 123
   S_IGUAL        = 124
   S_DISTINTO     = 125

   S_ASIGNACION   = 130
   S_COMA         = 131
   S_PUNTO_Y_COMA = 132

   R_VAR          = 200
   R_INICIO       = 201
   R_FIN          = 202
   R_IF           = 203
   R_ELSE         = 204
   R_WHILE        = 205
   R_READ         = 220
   R_PRINT     = 221

// Utilizados mas bien como indicadores

   S_EOF          = -1
   S_NADA         = -2
   S_ERROR        = -3

// [=============================================================]
// [           Elementos de la tabla de simbolos                 ]
// [=============================================================]

const
   MAX_SIMBOLOS = 300

tipos
   TTipo      : numerico
```

```
const
   t_NUM  = 1024
   t_TIME = 1025


tipos
   INFO_SIMB  : registro
              {
                 nombre      : cadena
                 tipo        : TTipo
                 dir         : numerico
                 val_inicial : numerico
              }

var
  tabs : vector [MAX_SIMBOLOS] INFO_SIMB
  cs   : numerico                         // cantidad simbolos ya ingresados


// [====================================================================]
// [            Elementos de la generacion de "codigo"                  ]
// [====================================================================]

const
   MAX_CODIGO             = 300          // solo 300 instrucciones...
   TOT_INSTRUCCIONES      = 21


tipos
   INSTRUCCION : numerico
   INFO_CODIGO : registro
               {
                  inst   : INSTRUCCION
                  op     : numerico
               }

   INFO_INSTR  : registro
               {
                   nombre : cadena
                   usa_op : logico
               }
var
   codigo     : vector [MAX_CODIGO] INFO_CODIGO
   tot_inst   : numerico
   info_instr : vector [TOT_INSTRUCCIONES] INFO_INSTR


/*
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
              ATENCION ATENCION ATENCION ATENCION ATENCION
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
      El "codigo" generado corresponde a una maquina hipotetica, basda en
      pila, cuyas instrucciones son:

      a) INSTRUCCIONES "RELACIONALES"
            cmp_<      : es tope-1 < tope?
            cmp_>      : es tope-1 > tope?
            cmp_==     : es tope-1 == tope?
            cmp_>=     : es tope-1 >= tope?
            cmp_<=     : es tope-1 <= tope?
            cmp_<>     : es tope-1 <> tope?

         Todas estas instrucciones deben "evaluar" la condicion, eliminar
         de la pila los dos valores y apilar el resultado, como
         el valor 0 (falso) o no 0 (verdadero).

      b) INSTRUCCIONES "ARITMETICAS"
            restar     : (tope-1) - (tope)
            sumar      : (tope-1) + (tope)
            mult       : (tope-1) * (tope)
            div        : (tope-1) / (tope)
            camb_signo : -tope

         Todas estas instrucciones deben realizar la operacion, eliminar
         de la pila los dos valores y apilar el resultado.

      c) MANIPULACION DE LA PILA
            apilar_const       : apilar el valor que se encuentra en op.
            apilar_valor_var   : apilar el valor de la variable cuya
                                 direccion es dir y que se encuentra en op.
            ivar               : crear una variable en el tope de la
                                 pila con valor inicial en op.
            apilar_dir         : apilar la direccion que se encuentra en op.
            asignar            : copiar el valor del tope de la pila
                                 en la direccion apuntada por (tope-1).

      d) SALTOS
            saltar             : saltar a la direccion indicada por op.
            saltar_si_falso    : si valor del tope == 0, saltar a direccion
                                 indicada por op.

      e) ENTRADA/SALIDA
            leer               : leer un valor numerico y cargar en la
                                 direccion apuntada por op.
            imprimir           : imprimir el valor del tope de la pila.

      f) VARIOS
            parar              : terminar ejecucion.
*/

const
    CX_CMP_MENOR              = 1
    CX_CMP_MAYOR              = 2
```

```
    CX_CMP_MENOR_IGUAL      = 3
    CX_CMP_MAYOR_IGUAL      = 4
    CX_CMP_IGUAL            = 5
    CX_CMP_DISTINTO         = 6

    CX_MAT_SUMAR            = 7
    CX_MAT_RESTAR           = 8
    CX_MAT_MULT             = 9
    CX_MAT_DIV              = 10
    CX_MAT_CAMB_SIGNO       = 11


    CX_APILAR_CONST         = 12
    CX_APILAR_VALOR_VAR     = 13
    CX_IVAR                 = 14
    CX_APILAR_DIR           = 15
    CX_ASIGNAR              = 16

    CX_SALTAR               = 17
    CX_SALTAR_SI_FALSO      = 18

    CX_LEER                 = 19
    CX_IMPRIMIR             = 20

    CX_PARAR                = 21

// [=============================================================]
// [           Elementos para la ejecucion                       ]
// [=============================================================]

const
   MAX_MEMORIA = 500          // la memoria tiene lugar para 500 variables

tipos
   ELEM_PILA : numerico

var
    pila : vector [MAX_MEMORIA] ELEM_PILA
    tp   : numerico


// [=============================================================]
// [           Otras variables                                   ]
// [=============================================================]

var
   cant_error   : numerico            // cuantos errores se encontraron?
   mostrar_inst : logico

const
   SE_ESPERA    = "Se espera "        // mensaje comun
```

```
// [=============================================================]
// [            PROGRAMA PRINCIPAL                               ]
// [=============================================================]

inicio
   cant_error = 0
   inicializar_pal_reserv()
   inicializar_scanner()
   inicializar_tabs()
   inicializar_codigo()
   si ( pudo_abrirse_fuente() )
   {
      sgte_lex()
      analizar_fuente()
      si ( cant_error == 0 )
      {
         si ( mostrar_inst )
         {
            imprimir_codigo()
         sino
            inicializar_interprete()

            /*
               Estuve leyendo desde el archivo que me pasaron como
               parametro. A partir de ahora, debo leer del teclado.
               Eso se hace con set_stdin ("")
            */

            set_stdin ("")

            ejecutar()
         }
      }
   }
fin

subrutina pudo_abrirse_fuente() retorna logico
var
   nomb_fuente  : cadena
   p_archivo    : numerico
   ok           : logico
inicio
   ok = NO
   si ( paramval (1) == "-d" )
   {
      mostrar_inst = SI
      p_archivo    = 2
   sino
      mostrar_inst = NO
      p_archivo    = 1
   }
```

```
   si ( pcount() < p_archivo )
   {
      imprimir ("\nFalta especificar archivo fuente MicroL")
   sino
      nomb_fuente = paramval (p_archivo)
      ok = set_stdin (nomb_fuente)
      si ( not ok )
      {
         imprimir ("\nNo pudo abrirse archivo ", nomb_fuente)
         ok = NO
      }
   }
   retorna ( ok )
fin


// ------------------%%%%%%%%%--------------------
//        RUTINAS DEL SCANNER
// ------------------%%%%%%%%%--------------------

subrutina inicializar_pal_reserv()
inicio
   pal_reserv = { {"var",      R_VAR      },
                  {"begin",    R_INICIO   },
                  {"end",      R_FIN      },
                  {"if",       R_IF       },
                  {"else",     R_ELSE     },
                  {"while",    R_WHILE    },
                  {“read",     R_READ     },
                  {"print",    R_PRINT }
                }
fin


subrutina inicializar_scanner()
  /*
     - Indicar que aun no se leyo' nada (llt=0)
     - Cambiar el separador de campo a fin-de-linea.
       De lo contrario "var a,b,c;" leera "var a", luego "b" y luego "c;".
       El cambio se hace con set_ifs().
  */
inicio
   lt = ""
   llt = 0
   up = 1
   set_ifs ('\n')
fin


subrutina leer_sgte_linea()
  /*
      Si no es eof, leer sgte. linea.
```

```
      Si eof, indicar tal situacion a sgte_lex() haciendo llt=-1.
   */
inicio
   si ( not eof() )
   {
      leer (lt)
      llt = strlen (lt)
      up  = 0
   sino
      llt = -1
   }
fin


subrutina es_letra (c : cadena) retorna logico
var
   r : logico
inicio
   r = (c >= 'a' and c <= 'z') or
       (c >= 'A' and c <= 'Z')
   retorna ( r )
fin



subrutina es_digito (c : cadena) retorna logico
var
   r : logico
inicio
   r = (c >= '0' and c <= '9')
   retorna ( r )
fin


subrutina es_palabra_reserv (s : cadena
                             ref ipr : INFO_PAL_RESERV) retorna logico
   /*
      Buscar la cadena s en la lista de palabras reservadas.
      Observar que ipf se pasa por referencia.
   */

var
   k : numerico
   enc : logico
inicio
   enc = NO
   desde k=1 hasta alen (pal_reserv)
   {
      si ( s == pal_reserv [k].pal )
      {
         ipr = pal_reserv [k]
         enc = SI
```

```
         k = alen (pal_reserv)
      }
   }
   retorna ( enc )
fin


subrutina leer_identif()
   /*
      Leer un identificador, es decir, una secuencia de letra seguida
      de letra o digito.

      No existe un limite especifico en la longitud.

      Se viene aqui si...
         Caracter anterior fue una letra (que esta en lt [up]).

      Dado que una palabra reservada luce "igual" que una variable,
      al terminar de leer el identificador se verifica si no es
      una palabra reservada, que de ser, tk tendra su valor simbolico.
   */
var
   pos_i  : numerico
   c      : cadena
   ipr    : INFO_PAL_RESERV
inicio
   pos_i = up
   repetir
      inc (up)
      c  = substr (lt, up, 1)
   hasta ( not (es_letra (c) or es_digito (c)) )

   dec (up)
   subcad = substr (lt, pos_i, (up - pos_i) + 1)

   si ( es_palabra_reserv (subcad, ipr) )
   {
      tk = ipr.lex
   sino
      tk = S_NOMBRE_VAR
   }
fin


subrutina leer_constante_entera() retorna cadena
var
   pos_i  : numerico
   c      : cadena
inicio
   pos_i = up
   repetir
      inc (up)
```

```
      c  = substr (lt, up, 1)
   hasta ( not (es_digito(c)) )

   dec (up)
   retorna (substr (lt, pos_i, (up-pos_i) + 1))
fin


subrutina leer_constante_numerica()
   /*
      Leer una secuencia de digitos.

      Se viene aqui si...
         Caracter anterior fue un digito.

      Observar que aqui no se considera el signo.
      Una constante numerica puede se entera o decimal.
      No puede usarse notacion cientifica.
   */
var
   num_cad : cadena
inicio
   tk = S_NADA
   num_cad = leer_constante_entera()
   si ( substr (lt, up+1, 1) == '.' )
   {
      inc (up, 2)
      si ( es_digito (substr (lt, up, 1)) )
      {
         num_cad = num_cad + '.' + leer_constante_entera()
      sino
         tk = S_ERROR
      }
   }
   val_num = val (num_cad)
   si ( tk == S_NADA )
   {
      tk = S_CONST_NUM
   }
fin


subrutina sgte_lex()
   /*
      - Identifica el sgte. lexema (token).
      - Si la linea se "acaba" lee otra linea, llamando a leer_sgte_linea().
   */
var
   c : cadena
inicio
   tk = S_NADA
   subcad = ""
```

```
mientras ( tk == S_NADA )
{
   si ( up >= llt )
   {
      leer_sgte_linea()
      si ( llt == -1 )
      {
         tk = S_EOF
      }
   }
   si ( tk <> S_EOF )
   {
      inc (up)
      c = substr (lt, up, 1)
      eval
      {
         caso ( c == '' )
              ;
         caso ( c == '\t' )
              ;
         caso ( c == ' ' )
              ;
         caso ( es_letra (c) )
              leer_identif()
         caso ( es_digito (c) )
              leer_constante_numerica()
         caso ( c == '+' )
              tk = S_MAS
         caso ( c == '-' )
              si ( lt [up+1] == '-' )
              {
                 up = llt + 1
              sino
                 tk = S_MENOS
              }
         caso ( c == '*' )
              tk = S_MULT
         caso ( c == '/' )
              tk = S_DIV
         caso ( c == '(' )
              tk = S_PARENT_I
         caso ( c == ')' )
              tk = S_PARENT_D
         caso ( c == '{' )
              tk = S_LLAVE_I
         caso ( c == '}' )
              tk = S_LLAVE_D
         caso ( c == ',' )
              tk = S_COMA
         caso ( c == ';' )
              tk = S_PUNTO_Y_COMA
         caso ( c == '<' )
```

```
                si ( lt [up+1] == '=' )
                {
                   tk = S_MENOR_IGUAL
                   inc (up)
                sino
                   tk = S_MENOR
                }
            caso ( c == '>' )
                si ( lt [up+1] == '=' )
                {
                   tk = S_MAYOR_IGUAL
                   inc (up)
                sino
                   tk = S_MAYOR
                }
            caso ( c == '=' )
                si ( lt [up+1] == '=' )
                {
                   tk = S_IGUAL
                   inc (up)
                sino
                   tk = S_ASIGNACION
                }
            caso ( c == '!' )
                si ( lt [up+1] == '=' )
                {
                   tk = S_DISTINTO
                   inc (up)
                sino
                   tk = S_ERROR
                }
            sino
                tk = S_ERROR
                subcad = c
        }
     }
  }
fin


// ------------------%%%%%%%%%%-------------------
//        RUTINAS DEL PARSER
// ------------------%%%%%%%%%%-------------------


subrutina es_oprel (s : LEXEMA) retorna logico
var
   r : logico
inicio

   r = (s == S_MENOR       or
```

```
        s == S_MENOR_IGUAL or
        s == S_MAYOR       or
        s == S_MAYOR_IGUAL or
        s == S_DISTINTO    or
        s == S_IGUAL
       )
   retorna ( r )
fin


subrutina es_signo (s : LEXEMA) retorna logico
var
   r : logico
inicio
   r = (s == S_MAS or s == S_MENOS)
   retorna ( r )
fin


subrutina expresion()
var
   op : LEXEMA
inicio
   expr_simple()
   si ( es_oprel (tk) )
   {
      op = tk;
      sgte_lex()
      expr_simple()
      GEN_OP_REL (op)
   }
fin


subrutina expr_simple()
var
   op : LEXEMA
inicio
   sub_expr()
   mientras ( es_signo (tk) )
   {
     op = tk
     sgte_lex()
     sub_expr()
     GEN_OP_MAT (op)
   }
fin


subrutina sub_expr()
var
   op : LEXEMA
```

```
inicio
   factor()
   mientras ( tk == S_MULT or tk == S_DIV )
   {
      op = tk
      sgte_lex()
      factor()
      GEN_OP_MAT (op)
   }
fin


subrutina factor()
var
   dir_var : numerico
inicio
   eval
   {
      caso ( tk == S_CONST_NUM )
           GEN_APILAR_CONST (val_num)
           sgte_lex()
      caso ( tk == S_NOMBRE_VAR )
           dir_var = ubicar_simb (subcad)
           si ( dir_var <> 0 )
           {
              GEN_APILAR_VAR (tabs [dir_var].dir)
              sgte_lex()
           sino
              error ("Variable '" + subcad + "' no fue declarada")
           }
      caso ( tk == S_PARENT_I )
           sgte_lex()
           expresion()
           chk_lex (S_PARENT_D, "')'")
      caso ( tk == S_MENOS )
           sgte_lex()
           factor()
           GEN_CAMB_SIGNO()
      caso ( tk == S_MAS )
           sgte_lex()
      sino
           error ("Simbolo no reconocido:")
   }
fin


subrutina decl_una_var()
var
   nueva_var : INFO_SIMB
   signo     : numerico
   hay_signo : logico
inicio
```

```
   signo = 1
   hay_signo = FALSE
   si ( tk = S_NOMBRE_VAR)
   {
      si ( ubicar_simb (subcad) <> 0 )
      {
         error ("Variable '" + subcad + "' ya fue declarada")
      sino
         nueva_var.nombre      = subcad
         nueva_var.val_inicial = 0
         nueva_var.dir         = 0

         sgte_lex()
         si ( tk = S_ASIGNACION )
         {
            sgte_lex()
            si ( tk = S_MENOS )
            {
               signo = -1
               sgte_lex()
               hay_signo = TRUE
            sino si ( tk = S_MAS )
               sgte_lex()
               hay_signo = TRUE
            }
            si ( tk = S_CONST_NUM )
            {
               nueva_var.val_inicial = val_num * signo
               sgte_lex()
            sino
               error (SE_ESPERA + "contante numerica")
            }
         }
      }
      agregar_simb (nueva_var)
      GEN_IVAR (nueva_var)
   sino
      error (SE_ESPERA + "nombre de variable")
   }
fin


subrutina decl_lista_var()
inicio
   decl_una_var()
   mientras ( tk = S_COMA )
   {
      sgte_lex()
      decl_una_var()
   }
   chk_lex (S_PUNTO_Y_COMA, "';'")
fin
```

```
subrutina decl_variables()
inicio
   decl_lista_var()
   mientras ( tk == S_NOMBRE_VAR )
   {
      decl_lista_var()
   }
fin


subrutina chk_lex (cmp : LEXEMA; msg : cadena)
inicio
   si ( tk == cmp )
   {
      sgte_lex()
   sino
      error (SE_ESPERA + msg)
   }
fin


subrutina sent_asignacion()
var
   dir_var : numerico
inicio
   dir_var = ubicar_simb (subcad)
   si ( dir_var <> 0 )
   {

      GEN_APILAR_DIR (tabs [dir_var].dir)

      sgte_lex()
      chk_lex ( S_ASIGNACION, "'='")
      expresion()

      GEN_ASIGNAR()

   sino
      error ("Variable '" + subcad + "' no fue declarada")
   }
   chk_lex (S_PUNTO_Y_COMA , "';'")
fin


subrutina bloque()
inicio
   chk_lex (S_LLAVE_I, "'{'")
   grupo_sentencias()
   chk_lex (S_LLAVE_D, "'}'")
fin
```

```
subrutina sent_if()
var
   hueco_if : numerico
   hueco_salto_a_fin_if : numerico
inicio
   sgte_lex()
   chk_lex (S_PARENT_I, "'('")
   expresion()

   hueco_if = dir_sgte_inst()

   chk_lex (S_PARENT_D, "')'")

   GEN_SALTO_FALSO (-1)

   bloque()

   si ( tk <> R_ELSE )
   {
      GEN_COMPLETAR_SALTO (hueco_if, dir_sgte_inst())
   sino
      GEN_COMPLETAR_SALTO (hueco_if, dir_sgte_inst() + 1)

      hueco_salto_a_fin_if = dir_sgte_inst()
      GEN_SALTO (-1)

      sgte_lex()

      bloque()

      GEN_COMPLETAR_SALTO (hueco_salto_a_fin_if, dir_sgte_inst())
   }
fin


subrutina sent_while()
var
   inicio_while : numerico
   hueco : numerico
inicio
   sgte_lex()

   inicio_while = dir_sgte_inst()

   chk_lex (S_PARENT_I, "'('")
   expresion()
   chk_lex (S_PARENT_D, "')'")

   hueco = dir_sgte_inst()
```

```
   GEN_SALTO_FALSO (-1)

   bloque()
   GEN_SALTO (inicio_while)

   GEN_COMPLETAR_SALTO (hueco, dir_sgte_inst())

fin


subrutina sent_read()
var
   dir_var : numerico
inicio
   sgte_lex()
   si ( tk <> S_NOMBRE_VAR )
   {
      error (SE_ESPERA + " nombre de variable")
   sino
      dir_var = ubicar_simb (subcad)
      si ( dir_var == 0 )
      {
         error ("Variable '" + subcad + "' no fue declarada")
      sino
         GEN_LEER (tabs [dir_var].dir)
         sgte_lex()
      }
      chk_lex (S_PUNTO_Y_COMA , "';'")
   }
fin


subrutina sent_print()
inicio
   sgte_lex()
   expresion()
   GEN_IMPRIMIR()
   chk_lex (S_PUNTO_Y_COMA , "';'")
fin


subrutina grupo_sentencias()
var
   ok : logico
inicio
   ok = SI
   mientras ( ok )
   {
      eval
      {
         caso ( tk == S_PUNTO_Y_COMA )
              sgte_lex()
```

```
            caso ( tk == S_NOMBRE_VAR )
                 sent_asignacion()
            caso ( tk == R_IF )
                 sent_if()
            caso ( tk == R_WHILE )
                 sent_while()
            caso ( tk == R_READ )
                 sent_read()
            caso ( tk == R_PRINT )
                 sent_print()
            sino
                 ok = NO
         }
      }
      si ( not (tk == R_FIN or tk == S_LLAVE_D) )
      {
         error ("Sentencia no reconocida")
      }
fin


subrutina analizar_fuente()
inicio
   si ( tk == R_VAR )
   {
      sgte_lex()
      decl_variables()
   }

   chk_lex (R_INICIO, "'inicio'")
   grupo_sentencias()
   chk_lex (R_FIN, "'fin'")
   GEN_PARAR()
   si ( tk <> S_EOF )
   {
      error ("Programa no termina correctamente")
   }
fin


subrutina error (s : cadena)
inicio
   si ( cant_error == 0 )
   {
      imprimir ("\n", lt, "\n", strdup (" ", up), "^ ", s, "\n")
      inc (cant_error)
   }
   sgte_lex()
fin


// ------------------%%%%%%%%%--------------------
```

```
//        RUTINAS DE LA TABLA DE SIMBOLOS
// ------------------%%%%%%%%%--------------------

subrutina inicializar_tabs()
inicio
   cs = 0
fin


subrutina ubicar_simb (s : cadena) retorna numerico
var
   k, ubic : numerico
inicio
   ubic = 0
   desde k = 1 hasta cs
   {
      si ( s == tabs [k].nombre )
      {
         ubic = k
      }
   }
   retorna ( ubic )
fin


subrutina agregar_simb (simb : INFO_SIMB)
inicio
   si ( cs < MAX_SIMBOLOS )
   {
      tabs [inc(cs)] = simb
      tabs [cs].dir  = cs
   sino
      error ("Tabla de simbolos llena")
   }
fin


// --------------- GENERACION DE "CODIGO" ---------------

subrutina inicializar_codigo()
var
   k : numerico
inicio
   tot_inst = 0

   info_instr [CX_CMP_MENOR         ] = {"cmp_<",          NO}
   info_instr [CX_CMP_MAYOR         ] = {"cmp_>",          NO}
   info_instr [CX_CMP_MENOR_IGUAL   ] = {"cmp_<=",         NO}
   info_instr [CX_CMP_MAYOR_IGUAL   ] = {"cmp_>=",         NO}
   info_instr [CX_CMP_IGUAL         ] = {"cmp_=",          NO}
   info_instr [CX_CMP_DISTINTO      ] = {"cmp_<>",         NO}
```

```
   info_instr [CX_MAT_SUMAR         ] = {"sumar",           NO}
   info_instr [CX_MAT_RESTAR        ] = {"restar",          NO}
   info_instr [CX_MAT_MULT          ] = {"mult",            NO}
   info_instr [CX_MAT_DIV           ] = {"div",             NO}
   info_instr [CX_MAT_CAMB_SIGNO    ] = {"camb_signo",      NO}

   info_instr [CX_APILAR_CONST      ] = {"apilar_const",    SI}
   info_instr [CX_APILAR_VALOR_VAR  ] = {"apilar_valor_var", SI}
   info_instr [CX_IVAR              ] = {"ivar",            SI}
   info_instr [CX_APILAR_DIR        ] = {"apilar_dir",      SI}
   info_instr [CX_ASIGNAR           ] = {"asignar",         NO}

   info_instr [CX_SALTAR            ] = {"saltar",          SI}
   info_instr [CX_SALTAR_SI_FALSO   ] = {"saltar_si_falso",  SI}

   info_instr [CX_LEER              ] = {"leer",            SI}
   info_instr [CX_IMPRIMIR          ] = {"imprimir",        NO}

   info_instr [CX_PARAR             ] = {"parar",           NO}
fin


subrutina gen_cod (inst: INSTRUCCION; op : numerico)
inicio
   si ( tot_inst < MAX_CODIGO )
   {
      inc (tot_inst)
      codigo [tot_inst].inst   = inst
      codigo [tot_inst].op     = op
   sino
      error ("Tabla de codigos llena")
   }
fin


subrutina dir_sgte_inst() retorna numerico
inicio
   retorna (tot_inst + 1)
fin


subrutina imprimir_codigo()
var
   k : numerico
   ii : INFO_INSTR
inicio
   desde k=1 hasta tot_inst
   {
      imprimir ("\n", str (k, 10, 0), "\t")
      si ( codigo [k].inst >= 1 and codigo[k].inst <= TOT_INSTRUCCIONES )
      {
         ii = info_instr [codigo [k].inst]
```

```
        imprimir (substr (ii.nombre + strdup (' ', 20), 1, 20) )
        si ( ii.usa_op == SI )
        {
            imprimir (codigo [k].op)
        }
      sino
        imprimir ("** Instruccion desconocida **")
      }
   }
fin


subrutina GEN_OP_REL (l : LEXEMA)
inicio
   eval
   {
      caso ( l == S_MENOR )
                          gen_cod (CX_CMP_MENOR, 0)
      caso ( l == S_MAYOR )
                          gen_cod (CX_CMP_MAYOR, 0)
      caso ( l == S_IGUAL )
                          gen_cod (CX_CMP_IGUAL, 0)
      caso ( l == S_MAYOR_IGUAL )
                          gen_cod (CX_CMP_MAYOR_IGUAL, 0)
      caso ( l == S_MENOR_IGUAL )
                          gen_cod (CX_CMP_MENOR_IGUAL, 0)
      caso ( l == S_DISTINTO )
                          gen_cod (CX_CMP_DISTINTO, 0)
   }
fin

subrutina GEN_OP_MAT (l: LEXEMA)
inicio
   eval
   {
      caso ( l == S_MENOS )
                         gen_cod (CX_MAT_RESTAR, 0)
      caso ( l == S_MAS )
                         gen_cod (CX_MAT_SUMAR, 0)
      caso ( l == S_MULT )
                         gen_cod (CX_MAT_MULT, 0)
      caso ( l == S_DIV )
                         gen_cod (CX_MAT_DIV, 0)
   }
fin


subrutina GEN_APILAR_CONST (n : numerico)
inicio
  gen_cod (CX_APILAR_CONST, n)
fin
```

```
subrutina GEN_APILAR_VAR (dir : numerico)
inicio
  gen_cod (CX_APILAR_VALOR_VAR, dir)
fin


subrutina GEN_CAMB_SIGNO()
inicio
  gen_cod (CX_MAT_CAMB_SIGNO, 0)
fin


subrutina GEN_IVAR (s : INFO_SIMB)
inicio
   gen_cod (CX_IVAR, s.val_inicial)
fin


subrutina GEN_APILAR_DIR (dir : numerico)
inicio
   gen_cod (CX_APILAR_DIR, dir)
fin


subrutina GEN_ASIGNAR()
inicio
   gen_cod (CX_ASIGNAR, 0)
fin


subrutina GEN_SALTO (dir : numerico)
inicio
   gen_cod (CX_SALTAR, dir)
fin


subrutina GEN_SALTO_FALSO (dir : numerico)
inicio
   gen_cod (CX_SALTAR_SI_FALSO, dir)
fin


subrutina GEN_COMPLETAR_SALTO (dir_cod, dir : numerico)
inicio
    codigo [dir_cod].op = dir
fin


subrutina GEN_LEER (dir : numerico)
inicio
```

```
   gen_cod (CX_LEER, dir)
fin


subrutina GEN_IMPRIMIR()
inicio
   gen_cod (CX_IMPRIMIR, 0)
fin


subrutina GEN_PARAR()
inicio
   gen_cod (CX_PARAR, 0)
fin


// [=========================================================]
// [           Rutinas del interprete                        ]
// [=========================================================]

subrutina inicializar_interprete()
inicio
   tp = 0
fin

subrutina ejecutar()
var
   i  : INSTRUCCION
   pi : numerico               // indice de instruccion que esta en ejecucion
inicio
   pi = 0
   repetir
      i = codigo [inc (pi)].inst
      eval
      {
         caso ( i ==  CX_CMP_MENOR )
              dec (tp)
              si ( pila [tp] < pila [tp+1] )
              {
                 pila [tp] = 1
              sino
                 pila [tp] = 0
              }

         caso ( i == CX_CMP_MAYOR )
              dec (tp)
              si ( pila [tp] > pila [tp+1] )
              {
                 pila [tp] = 1
              sino
                 pila [tp] = 0
              }
```

```
caso ( i == CX_CMP_MENOR_IGUAL )
     dec (tp)
     si ( pila [tp] <= pila [tp+1] )
     {
       pila [tp] = 1
     sino
       pila [tp] = 0
     }

caso ( i == CX_CMP_MAYOR_IGUAL )
     dec (tp)
     si ( pila [tp] >= pila [tp+1] )
     {
       pila [tp] = 1
     sino
       pila [tp] = 0
     }

caso ( i == CX_CMP_IGUAL )
     dec (tp)
     si ( pila [tp] == pila [tp+1] )
     {
       pila [tp] = 1
     sino
       pila [tp] = 0
     }

caso ( i == CX_CMP_DISTINTO )
     dec (tp)
     si ( pila [tp] <> pila [tp+1] )
     {
       pila [tp] = 1
     sino
       pila [tp] = 0
     }

caso ( i == CX_MAT_SUMAR )
     dec (tp)
     pila [tp] = pila [tp] + pila [tp+1]

caso ( i == CX_MAT_RESTAR )
     dec (tp)
     pila [tp] = pila [tp] - pila [tp+1]

caso ( i == CX_MAT_MULT )
     dec (tp)
     pila [tp] = pila [tp] * pila [tp+1]

caso ( i == CX_MAT_DIV )
     dec (tp)
     pila [tp] = pila [tp] / pila [tp+1]
```

```
        caso ( i == CX_MAT_CAMB_SIGNO )
             pila [tp] = -pila [tp]

        caso ( i == CX_APILAR_CONST )
             pila [inc (tp)] = codigo [pi].op

        caso ( i == CX_APILAR_VALOR_VAR )
             pila [inc (tp)] = pila [codigo [pi].op]

        caso ( i == CX_IVAR )
             pila [inc (tp)] = codigo [pi].op

        caso ( i == CX_APILAR_DIR )
             pila [inc (tp)] = codigo [pi].op

        caso ( i == CX_ASIGNAR )
             pila [pila [tp-1]] = pila [tp]
             dec (tp, 2)

        caso ( i == CX_SALTAR )
             pi = codigo [pi].op - 1       // -1 pues luego se incrementara...

        caso ( i == CX_SALTAR_SI_FALSO )
             si ( pila [tp] == 0 )
             {
                pi = codigo [pi].op - 1    // -1 pues luego se incrementara...
             }
             dec (tp)

        caso ( i == CX_LEER )
             leer ( pila [codigo [pi].op] )

        caso ( i == CX_IMPRIMIR )
             imprimir ("\n", pila [tp] )
             dec (tp)

        caso ( i == CX_PARAR )
             ;
        sino
             imprimir ("\n", "** Instruccion desconocida **")
             i = CX_PARAR
     }
   hasta ( i == CX_PARAR )
fin
```

Anexo B. Subrutinas y funciones predefinidas

Subrutinas predefinidas

dim (arreglo, tam_dim1, tam_dim2, ...)

Inicializa un arreglo con los tamaños para cada dimensión especificados.

Ejemplo:

```
var
    M3 : matriz [*, *, *] numerico
inicio
   dim (M3, 5, 8, 3)
   ...
```

Crea un matriz tridimensional de 5 x 8 x 3.

imprimir (var1, var2,..., varn)

Imprime los valores que se pasan como parámetros.

```
imprimir (a, b*10, s + "algo")
```

Ver la sección "4.1.2 Las cadenas de caracteres" con relación a las secuencias especiales.

Se pueden imprimir variables de cadena, numéricas o lógicas. Estas últimas se imprimirán como "TRUE" o "FALSE".

cls()

Limpia la pantalla.

leer (var1, var2,..., varn)

Lee uno o mas valores y los asigna a las variables que se pasan como parámetros.

Ejemplo:

```
var
    n : numerico
    s : cadena
inicio
   leer (n, s)
…
```

Leerá dos valores y asignará a n y s respectivamente. El separador de campos es la coma. Es decir, para responder a la lectura anterior se deberá tipear:

```
100,ejemplo de cadena
```

seguido de ENTER. Al leer números se ignoran los espacios que lo preceden; al leer cadenas no se los ignora.
Se pueden leer variables numéricas o de cadena, pero no las que sean del tipo lógico.
Observación: el separador de campo se puede cambiar con set_ifs().

Si ya no existen datos en la entrada y se llama a leer(), se producirá un error de ejecución.

set_ifs (c : cadena)
Establece cuál carácter utilizará leer() como separador de campos.
Ejemplo:

```
var
    n : numerico
    s : cadena
inicio
   set_ifs ('#')
   leer (n, s)
…
```

Para responder a la lectura anterior se deberá tipear:

```
100#Perez, Juan
```

seguido de ENTER. Siendo # el separador de campos, n valdrá 10 y s "Perez, Juan".

Funciones predefinidas

abs (n: numerico) retorna numerico

Calcula el valor absoluto de n.

arctan (gr: numerico) retorna numerico

Calcula el arco tangente de gr, que debe estar expresado en radianes.

ascii (pos_tabla: numerico) retorna cadena

Retorna el carácter ASCII que se encuentra en la posición pos_tabla.

cos (gr: numerico) retorna numerico

Calcula el coseno de gr, que debe estar expresado en radianes.

dec (ref n: numerico; a : numerico) retorna numerico

Decrementa el valor de la variable n en a.

Por defecto a es 1, pero puede ser positivo o negativo.

Retorna el nuevo valor de n.

Ejemplo:

```
n = 10
dec (n)                 // es lo mismo que n = n - 1
dec (n, 2)              // es lo mismo que n = n - 2

A [dec (n)] = 32        // n será 6 y A [6] tendrá 32.
```

eof() retorna logico

Retorna verdadero cuando ya no existen datos en la entrada para ser leídos.

exp (n: numerico) retorna numerico

Calcula e^n (siendo e base de los logaritmos naturales).

get_ifs() retorna cadena

Retorna el carácter que leer() utilizará para separar los campos durante una operación de lectura.

Vea también set_ifs().

inc (ref n: numerico; a : numerico) retorna numerico

Incrementa el valor de la variable n en a.

Por defecto a es 1, pero puede ser positivo o negativo.

Retorna el nuevo valor de n.

Ejemplo:

```
n = 10
inc (n)                  // es lo mismo que n = n + 1
inc (n, 2)               // es lo mismo que n = n + 2

A [inc (n)] = 32         // n sera´ 14 y A [14] tendra 32.
```

int (n: numerico) retorna numerico

Extrae la parte entera de n.

Ejemplo:

```
y = int (3.45)             // y tendra´ 3
```

log (n: numerico) retorna numerico

Calcula logaritmo base 10 de n.

lower (s: cadena) retorna cadena

Retorna los caracteres alfabéticos de s convertidos a minúsculas.

mem() retorna numerico

Retorna la cantidad de memoria disponible para los programas SL, en bytes.

ord (c: cadena) retorna numerico

Retorna la posición en que se encuentra en la tabla ASCII el carácter que contiene el parámetro c.

Si el valor de c contiene más de un carácter se toma el primero.

```
a = ord (´A´)      // a tendra´ 65
```

paramval (k: numerico) retorna cadena

Retorna el k-ésimo parámetro que fue pasado al programa SL desde la línea de comandos al momento de ser invocado. Obsérvese que el valor retornado es una cadena.

pcount() retorna numerico

Retorna la cantidad de parámetros que fueron pasados al programa SL desde la línea de comandos al momento de ser invocado.

pos (s1, s2 : cadena: pinicial : numerico) retorna numerico

Retorna la posición donde se inicia la cadena s2 dentro de s1 considerando desde el carácter de s1 que se encuentra en pinicial. Si p inicial es omitido, se considera desde el inicio de s1. Si s2 no se encuentra en s1, retorna 0.

random (tope : numerico; sem : numerico) retorna numerico

Retorna un número randómico entero mayor o igual a 0 y menor a tope.
El parámetro sem es opcional. Si está presente, se reinicia el generador de números randómicos usando sem como semilla.

sec() retorna numerico

Retorna la cantidad de segundos transcurridos desde medianoche.

set_stdin (nom_archivo : cadena) retorna logico

Cambia el archivo de donde leer() tomará los datos.
Retorna falso si el archivo (cuyo nombre se pasa en nom_archivo) no existe o no puede abrirse. En este caso además se establece como archivo de lectura la entrada standard (usualmente el teclado).
Ejemplo:

```
inicio
   leer (n)                    // lee de la entrada standard, usualmente el teclado.
   set_stdin ("misdatos.txt")
   leer (n)                    // leerá del archivo "misdatos.txt" si éste
                               //    pudo ser abierto
fin
```

set_stdout (nom_archivo, modo: cadena) retorna logico

Cambia el archivo de donde se imprimir() tomará los datos.
El parámetro nom_archivo indica el nombre completo del archivo (incluyendo camino) que se desea utilizar como archivo de salida.
El parámetro modo tiene los siguientes valores y significados:

"wt" : sobreescribir el contenido del archivo, si éste ya existe.
"at" : agregar al final del archivo todo lo que se imprima, sin destruir su contenido.

Por defecto el parámetro modo toma el valor "wt".

Retorna falso si el archivo (cuyo nombre se pasa en nom_archivo) no pudo abrirse. En este caso además se establece la salida standard (usualmente la pantalla) como destino de las llamadas a imprimir().

Ejemplo:

```
inicio
   /*
      Imprimirá en la salida standard, usualmente la pantalla.
   */
   imprimir ("\nHola")
   set_stdout ("saludos.txt")
   /*
      Imprimirá en el archivo "saludos.txt" si éste pudo ser abierto.
   */
   imprimir ("\nAdiós")
fin
```

sin (gr: numerico) retorna numerico

Calcula el seno de gr, que debe estar expresado en radianes.

sqrt (n: numerico) retorna numerico

Calcula la raíz cuadrada de n.

str (n, a, cant_dec: numerico; r : cadena) retorna cadena

Convierte a cadena el número n, con cant_dec decimales (por defecto 2), con un ancho total de a caracteres (por defecto 0) y, si fuera necesario, rellenando a la izquierda con el carácter contenido en r (por defecto un espacio).
Ejemplos:

```
n = 123.40451
s = str (n)                // -> "123.40"
s = str (n,10)             // -> "     123.40"
s = str (n,10,3)           // -> "     123.405"      (redondeado!)
s = str (n,0,0)            // -> "123"
s = str (n,10,0,'*')       // -> "*******123"
s = str (n,1,1)            // -> "123.4"
```

Si el numero n convertido a cadena tiene más de a caracteres, a es ignorado.
El signo negativo y el punto decimal son tenidos en cuenta para el cálculo del ancho total.

strdup (cadena s: cant : numerico) retorna cadena

Replica la cadena s cant veces.

strlen (cadena s) retorna numerico

Retorna la cantidad de caracteres contenidos en la cadena s.
La cadena vacía ("") tiene longitud 0.

substr (s: cadena; inicio, cant : numerico) retorna cadena

Extrae una subcadena de s, a partir del carácter que se encuentra en inicio, cant caracteres.
Si el parámetro cant se omite, se extrae hasta el final de la cadena.

Ejemplo:

```
s = substr ("ABC", 1, 1)        // -> "A"
s = substr ("ABCD", 2)          // -> "BCD"
s = substr ("ABCD", 5, 1)       // -> ""       (cadena vacia)
```

tan (gr: numerico) retorna numerico

Calcula la tangente de gr, que debe estar expresado en radianes.

upper (s: cadena) retorna cadena

Retorna los caracteres alfabéticos de s convertidos a mayúsculas.

val (s: cadena) retorna numerico

Retorna el valor numérico de lo que contiene s.
Retorna 0 cuando lo que contiene s no puede ser interpretado como un valor numérico.
Ejemplo:

```
s = "123.4"
n = val (s)                 // n tendra´ 123.4
n = val ("-12")             // n tendra´ -12
n = val ("abc")             // n tendra´ 0
```