SL: un lenguaje para la introducción a la Algoritmia

Juan Segovia Silvero
Centro Nacional de Computación
Universidad Nacional de Asunción
Asunción, Paraguay

jsegovia@cnc.una.py

Resumen

Este documento presenta el lenguaje SL: un lenguaje diseñado para servir de apoyo a la enseñanza de Algoritmia en los primeros años de las carreras de Informática. Como tal pretende servir tanto al estudiante como al docente, acompañando el proceso de construcción de algoritmos y posibilitando que el alumno vea este proceso como una experiencia vivencial, enriquecedora. La implantación actual incluye un compilador, un ambiente de desarrollo amigable con depurador simbólico. El lenguaje se viene utilizando con resultados muy alentadores en la Facultad Politécnica de la Universidad Nacional de Asunción.

1 Introducción

La Algoritmia es un componente importante del plan curricular de las carreras de Informática pues fortalece el "pensamiento algorítmico" del estudiante, es decir, su capacidad de abordar y resolver diferentes problemas mediante una serie de pasos concretos, variando adecuadamente el nivel de abstracción pero sin perder la perspectiva global.

La enseñanza de Algoritmia requiere un enfoque teórico-práctico ya que el estudiante debe adquirir conceptos y a la vez aprender técnicas específicas de programación. El proceso debe servir además para que el estudiante vaya formando criterios de evaluación de sus propios algoritmos y de terceros.

En las etapas introductorias, este proceso de aprendizaje suele ser doloroso para el estudiante pues por lo general implica un rompimiento con esquemas previos de estudio: si antes existían respuestas "correctas" y más o menos establecidas para los problemas, ahora se espera que él como estudiante plantee soluciones, demuestre la validez de su propio trabajo, analice otras soluciones y explore variantes, etc.

En esta etapa inicial el problema se profundiza, creemos, por el hecho de que el estudiante no siempre reconoce con la suficiente rapidez que la construcción de algoritmos y la programación son experiencias vivenciales, es decir, *requieren* que él se involucre intelectualmente; él no puede ser un mero expectador [1].

La transformación de expectador a actor puede darse con mayor facilidad si los algoritmos estudiados (analizados en clase, propuestos como tareas, presentados en exámenes, etc.) pueden ser ejecutados en un computador, es decir, convertirse en programas, con un mínimo de esfuerzo. En apartados siguientes presentaremos SL, un lenguaje diseñado específicamente con este propósito.

2 Un lenguaje algorítmico para el estudiante y el docente

La utilización de un lenguaje de computación específico como medio de expresión de algoritmos no está excenta de riesgos. La elección de un lenguaje inapropiado puede terminar dificultando el proceso tanto para el alumno como para el profesor: El alumno no solo debe estudiar el contenido mismo de la asignatura, sino que además debe estudiar las peculiaridades del lenguaje en cuestión; y el profesor debe dedicar tiempo para explicar aspectos sintácticos y semánticos del lenguaje, características específicas de una implantación del compilador y hasta detalles del entorno de programación. En resumen: un lenguaje inapropiado se convierte en un distractor y ataca los objetivos educacionales antes que apoyarlos [2].

Para el caso del modelo computacional imperativo, una de las primeras propuestas de lenguajes orientados hacia la docencia fue Pascal. Los méritos y desventajas de este lenguaje fueron y aun son ampliamente debatidos, siendo la conclusión general que existen elementos que imposibilitan catalogar a Pascal como el lenguaje ideal en el contexto que estamos señalando [2].

Requerimientos de un lenguaje de introducción a la Algoritmia

¿Qué características debe tener un lenguaje que cumpla los propósitos educacionales de un curso introductorio de Algoritmia, que no se convierta en un mero distractor? La respuesta a esta pregunta puede ser extensa, o incluso muy puntual, dependiendo de factores tales como el objetivo específico del curso, la orientación de la carrera, la duración, el perfil de los estudiantes, etc., pero en general podemos considerar como requisitos esenciales los siguientes [3]:

- Las construcciones del lenguaje deben ser simples de explicar, escribir y comprender. La sintaxis debe ser consistente; el propósito y comportamiento de cada construcción deben ser incluso deducibles. Además, cada construcción debe ser fácilmente justificable, es decir, no debe lucir "caprichosa". (La sentencia condicional de Pascal en cuanto al uso del punto y coma puede considerarse "caprichosa").
- Debe poseer un conjunto no redundante de construcciones y tipos de datos primitivos (la distinción entre enteros con signo, sin signo, números de punto flotante de precisión simple y doble, etc. son todos distractores para esta etapa del aprendizaje).
- Debe ser expresivo tanto para los ejercicios y conceptos sencillos, presentados al inicio de curso, como para los tópicos más o menos avanzados, con mayor énfasis en abstracciones de datos y procesos.
- No debe tener limitaciones arbitrarias (longitud absurda de los identificadores, capacidad de las cadenas, cantidad de dimensiones de los arreglos, cantidad de parámetros de los subprogramas, tipos de los valores de retorno de los subprogramas, etc.)
- Los conceptos y construcciones del lenguaje deben ser, en general, los que se encuentran en lenguajes y herramientas contemporáneas para no empobrecer o entorpecer la habilidad del futuro profesional.
- El lenguaje debe ser seguro y confiable: El sistema de tipos debe ser sencillo de comprender y los chequeos de compatibilidad estrictos. Además, en tiempo de ejecución deben reportarse todos los errores, referenciando la sección del programa fuente que lo provocó. En este contexto "confiable" quiere decir que todo programa que compila correctamente, solo podrá considerarse defectuoso por responsabilidad del estudiante-programador (acceso a elementos inexistentes de arreglos, división por cero, etc; la lógica expresada en el programa no resuelve el problema).
- No debe descuidar su propósito académico, es decir, los diseñadores del lenguaje no deben caer en la tentación de "hacerlo completo" pues entonces terminará siendo un lenguaje de propósito general, posiblemente con los mismos defectos de éstos (desde el punto de vista de la docencia).

Si bien no es parte del lenguaje mismo, un aspecto fundamental es la disponibilidad de un buen ambiente de desarrollo. Este ambiente debe incluir un depurador de código simbólico, con posibilidades de ejecución paso a paso. Además el sistema de diagnóstico de errores del compilador debe ser tal que ayude efectivamente al alumno en la identificación de los errores detectables en esta fase.

3 El Lenguaje SL y su ambiente de desarrollo

A continuación presentaremos brevemente las principales características de un lenguaje llamado SL. Este lenguaje fue 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 más avanzadas de programación. La implementación actual incluye un compilador, un ambiente de desarrollo amigable y materiales didácticos complementarios.

La sintaxis de SL, 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. Como tal lo hemos venido utilizado en los últimos años con los estudiantes de las carreras de Programador de Computadoras y Licenciatura en Análisis de Sistemas de la Facultad Politécnica de la Universidad Nacional de Asunción.

Una visión general de SL

Siendo un lenguaje de introducción a la Algoritmia, SL es un lenguaje imperativo [4] Es sintácticamente cercano a Pascal y C, pero reúne las características que hemos señalado como requisitos para un lenguaje de introducción a la Algoritmia.

Algunos puntos resaltantes son:

- 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 pueden 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 se definen antes de su uso, a excepción de los subprogramas. Mediante esta excepción se evita la "incomprensible" y molesta (para el estudiante de Introducción a la Algoritmia) necesidad de definir prototipos.

- Se cuenta con un rico conjunto de estructuras de iteración y selección.
- 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 ambiente de desarrollo

La figura 1 muestra una pantalla del ambiente de desarrollo. El estudiante puede utilizar este ambiente para:

- Preparar o modificar los programas fuentes, contando con funciones para cortar y pegar textos, realizar búsquedas y sustituciones, etc.
- Mantener simultáneamente varios programas fuentes, cada una situándose en una ventana independiente pero fácilmente accesible.
- Compilar los programas editados, recibiendo indicación acerca de la ubicación y naturaleza de los errores sintácticos o semánticos, si los hubiere.
- Ejecutar programas y depurarlos, 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.

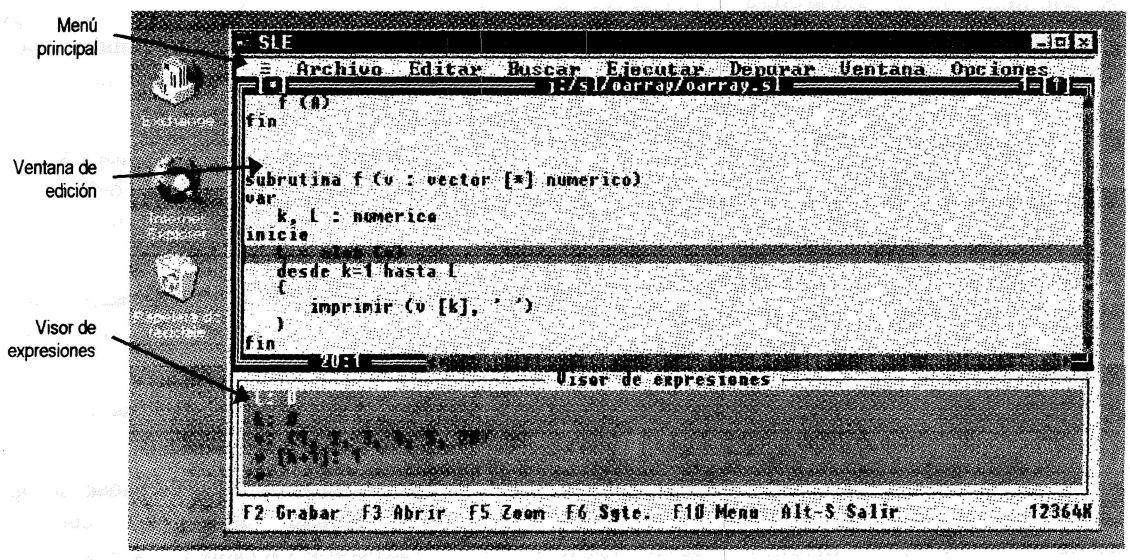

Figura 1. El ambiente de desarrollo de SL

La evolución de SL

SL ha ido evolucionando durante estos años de su utilización en aulas. Las ampliaciones están motivadas en su mayoría por la necesidad de eliminar detalles a medida que el desarrollo del curso avanza, es decir, mejorar la expresión de abstracciones. Nuestro criterio es que toda construcción que usemos en clase debe ser "ejecutable" en SL. En este sentido hemos agregado arreglos y registros literales, arreglos de inicialización dinámica, parámetros con arreglos "abiertos", entre otros.

Algunos ejemplos

A continuación veremos dos ejemplos del uso de SL para resolver problemas sencillos presentados habitualmente en los cursos de introducción a la Algoritmia.

Primer ejemplo: Conversión de decimal a hexadecimal

Dado un número entero positivo en base 10, producir su representación en hexadecimal.

La implementación hace uso del acceso directo a caracteres de cadenas y del operador de módulo. La conversión se implementa en una función cuyo valor de retorno es la cadena representando un número en hexadecimal.

```
var
   n : numerico
   h : cadena
   sl : numerico
inicio
   imprimir ("\nIngrese un numero entero > 0")
   leer (n)
   imprimir ("\n", n, " es ", dec_a_hex (n),
            " en hexadecimal")
fin

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

Figura 2. Conversión a hexadecimal en SL

Segundo ejemplo: Búsqueda binaria

Dado un número, determinar en qué posición de un arreglo ordenado ascendentemente se encuentra dicho número. Una posición 0 debe indicar que el elemento no se encuentra en el arreglo.

La figura 3 muestra la implementación en SL. Dado que el problema no especifica un máximo en la cantidad de elementos del arreglo, el parámetro formal de la función busq binaria() es un arreglo "abierto".

```
var
   V : vector [*] numerico
   p : numerico
inicio
   // Inicializamos el arreglo V
   V = {5, 21, 46, 78, 300, 1450}
   p = busq_binaria (V, 300)

   // Usando arreglos literales como parametros

   p = busq_binaria ({10,20,30,40,50}, 20)
fin

subrutina busq_binaria (A : vector [*] numerico
                        e : numerico
                        ) retorna numerico
   /*
      Retorna:
         Indice dentro de A donde se encuentra
         e (primera aparicion). Retorna 0
         cuando e no se encuentra.

         A debe estar ordenado ascendentemente.
   */
var
   p  : numerico
   izq, der, med : numerico
inicio
   izq = 1
   der = alen (A)  // cantidad de elem. en A
   p   = 0         // asumamos que no encontramos
   mientras ( izq <= der and p == 0 )
   {
      med = int ((izq + der) / 2)
      si ( e > A [med] )
      {
         izq = med + 1
      sino si ( e < A [med] )
         der = med - 1
      sino
         p = med
      }
   }
   retorna ( p )
fin
```

Figura 3. Búsqueda binaria

4 Implantación del compilador y del entorno de desarrollo de SL

El compilador de SL y su entorno de programación fueron desarrollados en el lenguaje C++. El compilador utilizado fue GNU GCC 2.7.2 en su versión DJGPP, para ambiente DOS de 32 bits. (La versión del compilador de SL fue producto de un proyecto aprobado por la Dirección de Investigaciones y

Relaciones Internacionales de la Universidad Nacional de Asunción).

Las fases de análisis y generación de código (código intermedio para una máquina abstracta basada en pila) así como el *runtime* son totalmente independientes de la interfase de usuario. De esta forma, además de la versión DOS, han sido producidos exitosamente versiones para las principales variantes de Unix (FreeBSD, Linux, SCO OpenServer, Digital UNIX y otros más). El entorno de desarrollo es específico para Windows95/98.

La adaptación de la interfase al compilador, el *runtime* y el depurador simbólico se realiza a través de funciones virtuales previstas para el efecto.

Siendo SL un lenguaje orientado a la docencia, la implantación favorece la robustez del compilador/entorno sobre la extrema eficiencia del código generado [5].

5 Consideraciones finales

Si bien aún no se han hecho experimentos controlados sobre el impacto del uso de SL en el proceso de aprendizaje de la construcción de algoritmos, la respuesta de los estudiantes ha sido muy positiva, no observándose dificultades originadas con la sintaxis o semántica del lenguaje.

Para los docentes resulta igualmente positivo que todo el curso, en sus diferentes niveles de complejidad, pueda desarrollarse con el mismo lenguaje, produciendo siempre algoritmos que los alumnos puedan poner en ejecución directamente.

Dónde obtener SL

SL está disponible en el siguiente URL:

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

REFERENCIAS

[1] Knuth D. "Algoritmos Fundamentales. El arte de programar ordenadores". Vol I. Editorial Reverté, 1986, pp. XVII.

[2] Martin L. "Is Turing a better language for teaching programming than Pascal?". Honours Dissertation . Dept. Computing Science, University of Stirling, 1996.

[3] Kölling, M., B. and Rosenberg, J. "Requirements for a First year Object-Oriented Teaching Language", ACM SIGCSE Bulletin, 27, 1, 1995, pp. 173-177.

[4] Baeza-Yates R.A. "Teaching algorithms". ACM SIGACT News, 26(4):51--59, Dec 1995.

[5] Kölling, M., B. and Rosenberg, J. "Testing Object-Oriented Programs: Making it Siimple", Proceedings of the 28th SIGCSE Symposium, 1997.