Laboratorios de Compiladores

Coromoto León, Casiano Rodríguez, Francisco de Sande

Universidad de La Laguna. Dpto. Estadística, I.O. y Computación
c/Astrofísico Fco. Sánchez s/n. 38071 La Laguna (cleon@ull.es)

Resumen

Aunque existe numeroso material bibliográfico de inmejorable calidad sobre las teorías que sustentan la construcción de compiladores, no hemos encontrado un texto que sirva de apoyo al profesor en el desarrollo de los contenidos de los laboratorios. Este trabajo proporciona una visión global y resumida de los ejercicios de laboratorio que se especifican en el libro "Prácticas de Compiladores en C y Pascal". En este texto se divide la implementación de un compilador en tareas más reducidas que se plantean como laboratorios prácticos interelacionados entre sí.

1 Introducción

En la elaboración de los contenidos de una asignatura cuyo objetivo sea la enseñanza de técnicas para la construcción de compiladores se ha de prestar especial atención a las clases prácticas. Se ha de mostrar cómo se organiza un compilador, cómo trabaja, cuáles son los principales problemas y cómo han sido resueltos. El objetivo principal ha de ser proporcionar al alumno herramientas que permitan escribir un compilador simple, así como, conseguir que sean capaces de dirigirse a la bibliografía adecuada para encontrar soluciones a los problemas que se les presenten.

La programación tiene un papel muy importante a la hora de escribir un compilador ya sea para un lenguaje complejo de alto nivel o para un sencillo lenguaje de comandos. El diseño del lenguaje y la posterior implementación del compilador constituyen un proyecto de envergadura. Resulta entonces natural el planteamiento de un proyecto final único en el que se apliquen las técnicas estudiadas en las clases teóricas, así como los conocimientos adquiridos en asignaturas como Ingeniería del Software. Una alternativa a este planteamiento consiste en dividir la implementación de un compilador, para un lenguaje sencillo, en pequeños proyectos que se le plantean al alumno una vez finalizadas las descripciones teóricas. Este trabajo proporciona una visión global y resumida de los ejercicios de laboratorio que se especifican en el libro "Prácticas de Compiladores en C y Pascal" [1].

En la elaboración de las prácticas de laboratorio que se proponen se ha procurado complementar la información recibida en las clases de teoría con la experiencia que aporta la aplicación de tales conocimientos de forma práctica. Con estos ejercicios prácticos se quiere mostrar la complejidad en el diseño y la escritura de un compilador a mano, cosa que conlleva cientos y cientos de líneas de código. Por ello se potencia el uso de herramientas que faciliten el diseño y la implementación de compiladores como son las herramientas *lex* y *yacc*.

Los lenguaje de programación utilizados en las prácticas son C y Pascal, pero puede utilizarse cualquier otro lenguaje imperativo como Modula 2, Ada, etc. Se genera código para *transputers* por la simplicidad del juego de instrucciones del mismo, pero puede usarse cualquier otro procesador.

Se proponen ocho módulos de prácticas que proporcionan material para la enseñanza de la construcción de compiladores a niveles de primero y segundo ciclos. Se han desarrollado para la enseñanza de técnicas de construcción de compiladores tanto descendentes como ascendentes. Los módulos de introducción son específicos para un lenguaje pero proporcionan ejemplos de cómo implementar los laboratorios de iniciación con otros.

Cada módulo contiene entre dos y doce prácticas. Estas se pueden utilizar como laborato-

rios formales y cerrados a realizar en las horas de prácticas, o como laboratorios que se fijan a los estudiantes para realizar por su cuenta en un laboratorio abierto. Cada módulo incluye referencias bibliográficas sobre el tópico. La colección completa de módulos incluye conceptos que se pueden utilizar en varios cursos diferentes. Por ejemplo, un curso puede estar orientado al análisis descendente recursivo, otro hacia el análisis ascendente, o bien organizar un curso para el desarrollo de la fase de análisis y otro para el de la fase de síntesis, etc.

Los dos módulos iniciales incluyen varios ejercios para la iniciación en el uso de las herramientas de ayuda a la construcción de compiladores *lex* y *yacc*. Estos módulos son los únicos que son completamente dependientes del lenguaje C, pero se pueden adaptar de forma sencilla para otros lenguajes como Pascal, Modula, etc.

Los seis módulos restantes muestran diferentes principios fundamentales en la construcción de las fases de análisis y síntesis de un compilador: el análisis léxico, el análisis sintáctico, el análisis semántico, la generación de código intermedio, la optimización de código y la generación de código para una máquina concreta. Se decide utilizar como lenguaje intermedio una representación postfija, pero esta se puede cambiar por una implementación del código de tres direcciones o a una representación gráfica. Las prácticas de generación de código se hacen para *transputers* pero la máquina objeto se puede cambiar de forma sencilla puesto que la fase inicial y la fase final de los compiladores que se desarrollan están claramente diferenciadas.

2 Laboratorios de Iniciación

En los dos módulos de prácticas que constituyen este apartado se estudian las herramientas para la generación automática de analizadores léxicos y analizadores sintácticos. En total se proponen diez laboratorios cuyo objetivo es proporcionar una primera toma de contacto con los entornos de programación de los generadores.

2.1 La Herramienta *Lex*

La tarea a realizar por un analizador léxico aunque sencilla, suele ser difícil de expresar de forma concisa en un lenguaje de propósito general. *Lex* es una herramienta que escribe partes significativas de un analizador léxico automáticamente basándose en una descripción dada por el usuario a base de expresiones regulares. El programa generado por *lex* recibe como entrada una cadena de caracteres arbitraria y la divide en sus componentes léxicos (*tokens*). Esta salida está escrita en un lenguaje de alto nivel y puede constituir la solución a un problema particular o bien se puede utilizar junto con la salida de la herramienta *yacc* para solucionar un problema más general. En este módulo se proponen un total de cinco prácticas.

Laboratorio 1.1 *Un programa lex sencillo*

Mediante un programa sencillo se pretende mostrar la forma de funcionamiento de la herramienta de construcción de analizadores léxicos *mks-Lex*.

Laboratorio 1.2 *Tamaño de los autómatas generados por lex*

Con otro ejemplo sencillo se introduce la utilidad *lex* del sistema operativo Unix para la construcción de analizadores léxicos. Además en esta práctica, se propone también el uso de *Flex* (Faster Lex), un generador de analizadores léxicos que está disponible como software de dominio público bajo Unix.

Laboratorio 1.3 *Número de caracteres, palabras y líneas de un conjunto de ficheros*

El analizador léxico generado por *lex* al encontrar el caracter de final de fichero llama a la función *yywrap()* que devuelve el valor uno en el caso de no tener que analizar más entrada. Puesto que se trata de analizar más de un fichero de entrada, se hace necesaria la implementación de una función *yywrap()* más compleja.

Laboratorio 1.4 *Reconocimiento de comentarios anidados*

El objetivo es describir el funcionamiento de las condiciones iniciales, una característica avanzada de la utilidad *lex*.

Laboratorio 1.5 *Análisis de una línea de comandos*

Normalmente la entrada para los programas generados por *lex* suele ser un fichero, pero algunas veces puede ser necesario utilizar otra fuente tal como una cadena almacenada en memoria. Todas las implementaciones de *lex* en los distintos sistemas permiten realizar esta modificación, pero los detalles varían considerablemente de una a otra. En esta práctica se propone la implementación de un analizador léxico de una línea de comandos, para la utilidad *lex* del sistema operativo Unix, para el generador de analizadores léxicos *mks-lex* y para *flex* mostrando así las diferencias entre los distintos generadores de analizadores léxicos.

2.2 La Herramienta *Yacc*

En este módulo se describe *yacc* una herramienta que facilita el desarrollo de la etapa inicial de un compilador. Mientras *lex* reconoce patrones representados mediante expresiones regulares, *yacc* reconoce gramáticas completas. *Lex* divide la entrada en *tokens* y *yacc* toma esas piezas y las agrupa de alguna forma lógica según las reglas de producción que le hallan proporcionado. Con *yacc* se puede escribir cualquier programa que analice la entrada de acuerdo a unas reglas gramaticales estrictas.

Laboratorio 2.1 *Un programa yacc sencillo*

En esta práctica se estudia la forma de utilizar un generador de analizadores sintácticos para facilitar la construcción de la etapa inicial de un compilador. Se utiliza la herramienta de construcción de analizadores sintácticos LALR *mks-yacc*.

Laboratorio 2.2 *Ejemplo de depuración de un programa yacc*

Yacc proporciona opciones que permiten hacer una traza de las acciones de desplazamiento y reducción que tienen lugar durante el análisis sintáctico LALR. Es interesante comprobar el funcionamiento de este algoritmo cuando se trabaja con una gramática ambigua, concretamente se propone la de las expresiones ($E \rightarrow E + E \mid E * E \mid ...$). Se ha de añadir a los operadores la precedencia y asociatividad (%left, %right, etc.) adecuada para solucionar los conflictos que se presentan.

Laboratorio 2.3 *Implementación de una calculadora*

Par hacer uso de la utilidad *yacc* del sistema operativo Unix se propone la implementación de una calculadora que permita asignaciones de variables y llamadas a funciones trigonométricas, logaritmos, etc. Además en esta práctica, se describe el modo de uso de *Bison* un generador de analizadores sintácticos que está disponible como software de dominio público bajo Unix.

Laboratorio 2.4 *Evaluación de polinomios*

El objetivo es ejercitarse en el uso de la herramienta *yacc* utilizando cualquiera de sus implementaciones.

Laboratorio 2.5 *Simulador de una máquina RAM*

Este laboratorio sirve de base para describir el funcionamiento de las acciones múltiples en *yacc*.

3 Laboratorios de desarrollo

La descripción de un compilador se divide en un conjunto de fases, cada fase depende una de la otra y lo más lógico es que cada práctica de laboratorio cubra una de ellas.

3.1 El análisis léxico

La forma más eficiente de crear un analizador léxico eficiente consiste en definir los componentes léxicos (*tokens*) del lenguaje fuente y programar de forma estructurada una aplicación para encontrarlos utilizando un diagrama de transiciones. La forma más sencilla de crear el mismo analizador léxico consiste en definir los *tokens* mediante expresiones regulares y generarlo con *lex*. Las prácticas que componen este módulo contemplan ambas alternativas.

Laboratorio 3.1 *Un analizador léxico para un subconjunto de Pascal*

Partiendo de los diagramas de transición de los *tokens* de un subconjunto de Pascal se propone la implementación del analizador léxico

asociado en un lenguaje de alto nivel. El objetivo es localizar los *tokens* del lenguaje y hacer una primera aproximación a la manipulación de tablas de símbolos.

Laboratorio 3.2 *Un analizador léxico para un subconjunto de C*

Este subconjuto del lenguaje C sólo trabaja con el tipo de datos entero, admite funciones con un número de parámetros arbitrario y con resultado entero; variables globales, locales a una función y locales a un bloque y las estructuras de control if-else y while. Se trata de definir los *tokens* correspondientes mediante expresiones regulares y generar el analizador léxico usando la herramienta *lex*.

3.2 El análisis sintáctico

El cuarto módulo de prácticas está dedicado al estudio de los métodos de análisis sintáctico más utilizados: el análisis sintáctico descendente recursivo y el análisis sintáctico por desplazamiento y reducción.

Puesto que los programas suelen contener errores sintácticos se pueden ampliar las metodologías estudiadas para que se recuperen de los errores más frecuentes. Se estudian los métodos de recuperación de errores que introducen símbolos de sincronización y los que añaden reglas de producción de error. En total se proponen cinco prácticas.

Laboratorio 4.1 *Un analizador sintáctico descendente recursivo*

La práctica consiste en la realización de un analizador sintáctico descendente recursivo predictivo para un subconjunto de Pascal. Dicho analizador tiene que reconocer las sentencias correctas y detectar errores sintácticos pero sin recuperarse de los mismos.

Laboratorio 4.2 *Analizador genérico de gramáticas LL(1)*

Dada una gramática y una frase la práctica consiste en escribir un programa que determine si la frase pertenece o no al lenguaje generado por la gramática. Concretamente, se intenta producir una derivación por la izquierda para dicha cadena.

Laboratorio 4.3 *Formateador de ficheros ("pretty printer")*

La práctica consiste en la realización de un formateador de ficheros escritos en lenguaje C. Para reconocer las construcciones gramaticales se utiliza la herramienta *yacc*. El analizador léxico es el desarrollado en el Laboratorio 3.2. El alumno ha de seleccionar las acciones semánticas necesarias para que se inserte el número de tabulaciones adecuado.

Laboratorio 4.4 *Recuperación de errores sintácticos mediante conjuntos de sincronización*

Los pasos que realiza el compilador inmediatamente después de descubrir un error sintáctico se conocen como "recuperación de errores". El propósito de la recuperación de errores es hacer al compilador capaz de continuar el análisis del programa fuente para encontrar el mayor número de errores en un sólo paso y asegurarse de que se emite sólo un mensaje por cada error. La práctica consiste en la ampliación del analizador sintáctico descendente recursivo predictivo descrito en el laboratorio 4.1 con recuperación de errores en modo de pánico.

Laboratorio 4.5 *Recuperación de errores sintácticos mediante la inclusión de producciones de error*

Los compiladores trabajan la mayoría de las veces con ficheros de entrada incorrectos. El problema en general consiste en ampliar las gramáticas introduciendo el símbolo error en algunas reglas. Afortunadamente se encuentra una manera directa de extender las reglas de producción de manera que se pueda analizar cualquier error. La práctica consiste en escribir un analizador sintáctico para un subconjunto del lenguaje C que incluya recuperación de errores insertando reglas de producción de error, haciendo uso del símbolo **error** que proporciona *yacc*.

3.3 El análisis semántico

Para describir el conjunto de tareas que componen el análisis semántico se utiliza una notación que proporciona una descripción formal de la semántica de un lenguaje de programación mediante la definición de atributos.

La búsqueda de nombres en una tabla de símbolos es un campo para el que existen múltiples técnicas disponibles. En general, la función que realiza el análisis léxico es la encargada de localizar en la tabla de símbolos un nombre definido por el usuario y si todavía es desconocido ha de introducirlo.

Partiendo del esqueleto de compilador que el alumno ha desarrollado en las prácticas anteriores, se le propone que realice la comprobación de tipos, para lo cual ha de definir el sistema de tipos y las reglas de compatibilidad.

Laboratorio 5.1 *Manipulación de bloques en Pascal*

La práctica consiste en ampliar el analizador sintáctico descendente recursivo con recuperación de errores del Laboratorio 4.4 para que compruebe un conjunto de reglas que permitan al compilador asociar cada referencia a un objeto con una declaración del mismo.

Laboratorio 5.2 *Manipulación de bloques en C*

En esta práctica se cambia el problema general del reconocimiento de un lenguaje al más específico de analizar el programa fuente para traducirlo a una versión ejecutable. Se trata de imponer ciertas restricciones que permitan completar la fase de análisis del programa fuente. Concretamente, se añadirá al analizador sintáctico descrito en el laboratorio 4.5 del módulo anterior el conjunto de acciones necesarias para manipular las declaraciones y referencias a objetos.

Laboratorio 5.3 *Análisis de tipo en Pascal*

Se trata de ampliar el programa que realiza el análisis de ámbito sobre un subconjunto de Pascal para que, utilizando las definiciones de los objetos, pueda llevar a cabo el análisis de tipo de las sentencias. El lenguaje cuenta con los tipos simples entero y boolean y los tipos estructurados *array* y registro.

Laboratorio 5.4 *Análisis de tipo en C*

En este laboratorio no es necesario preocuparse por la imcompatibilidad de tipos, puesto que el subconjunto de C con el que se está trabajando sólo cuenta con el tipo de datos entero (*int*). Sin embargo, se ha de tener en cuenta el hecho de que en el lenguaje C las funciones no necesiten ser definidas antes de que sean usadas y que no se pueden anidar las funciones. También se ha de comprobar el uso consistente de las funciones, esto es, que siempre se llamen con el mismo número de argumentos. Otra pequeña complicación surge del hecho de que en C los parámetros no necesitan ser declarados explícitamente: una vez que sus nombres han sido mencionados en una lista de parámetros, se convierten en variables enteras por defecto.

En general, las restricciones semánticas necesitan ser puestas en vigor en su contexto (es decir, cuando se está reduciendo por la regla de producción adecuada), esto hace que la pila de *yacc* sea el mejor sitio para pasar los valores intermedios de los atributos.

3.4 La generación de código Intermedio

En el modelo de análisis y síntesis de un compilador, la etapa inicial traduce un programa fuente a un representación intermedia a partir de la cual la etapa final genera el código objeto. Aunque la traducción de un programa fuente a un programa objeto se pueda realizar directamente existen algunas ventajas si se utiliza una representación intermedia independiente de la máquina. Entre ellas está la posibilidad de crear un compilador para una máquina distinta uniendo una etapa final para la nueva máquina a una etapa inicial ya existente. Además se puede aplicar a la representación intermedia un optimizador de código independiente de la máquina.

En este módulo se proponen doce laboratorios. Los cuatro primeros se utilizan para fijar las ideas acerca de las distintas representaciones intermedias. El quinto y el sexto establecen los ámbientes en el momento de la ejecución para los lenguajes con los que se ha estado trabajando. Los restantes proponen la traducción a una representación intermedia de las expresiones, las construcciones que modifican el flujo de control y los subprogramas.

Laboratorio 6.1 *Lenguajes intermedios: P-código para Pascal*

El lenguaje intermedio que se define en esta práctica pretende modelizar una máquina virtual a la que se denominará Máquina Pascal. Al código generado para esta máquina se le llamará Código Pascal. Esta representación es la ideal para el lenguaje definido en el Laboratorio 4.1 porque las instrucciones se corresponden directamente con conceptos del lenguaje. El propósito de esta práctica es la escritura de un simulador de código Pascal.

Laboratorio 6.2 *Lenguajes intermedios: P-código para C*

En esta práctica se propone la implementación de un ensamblador y un intérprete de P-código que sirva como máquina virtual para la traducción del subconjunto de C definido en el laboratorio 4.5.

Laboratorio 6.3 *Lenguajes intermedios: Árboles*

Si la salida del analizador sintáctico está en una forma desde la cual se pueda generar código fácilmente, después de un paso de optimización, esta debería ser un árbol sintáctico, también llamado un árbol sintáctico abstracto. Los árboles sintácticos para una sentencia dada tienen la misma forma general que un árbol de análisis sintáctico, pero los operadores sustituyen a las variables sintácticas en los nodos interiores. Se propone la creación de esta representación intermedia para el lenguaje de las expresiones utilizando *yacc*.

Laboratorio 6.4 *Lenguajes intermedios: Grafos dirigidos acíclicos*

Un grafo dirigido acíclico de una expresión identifica sus subexpresiones comunes. Se propone una ampliación de la práctica anterior con la identificación de las subexpresiones comunes a través del método del número de valor.

Laboratorio 6.5 *Asignación de memoria en Pascal*

Antes de asignar las variables declaradas en el programa fuente a posiciones de la memoria, es necesario definir ciertas cantidades como el número de localizaciones de memoria de los que se dispone y cuántas se va asignar a cada uno de los distintos tipos de objetos: variables enteras, reales, caracteres, etc. El objetivo de esta práctica es implementar un esquema de asignación de memoria muy simple para Pascal. La máquina para la que se realiza esta asignación es la que se describe en la práctica 6.1.

Laboratorio 6.6 *Asignación de memoria en C*

Llegado este punto, se está listo para definir una implementación del subconjunto de C para la máquina particular descrita en el laboratorio 6.2. Por lo tanto, se deben desarrollar políticas de asignación de memoria y ese es el objetivo de este ejercicio.

Laboratorio 6.7 *Generación de código intermedio: Expresiones en Pascal*

Se pretende traducir las expresiones y las sentencias de asignación del dialecto de Pascal con el que se ha estado trabajando a P-código de la máquina Pascal diseñada especialmente para ejecutar de forma eficiente el código de entrada.

Laboratorio 6.8 *Generación de código intermedio: Expresiones en C*

El objetivo es la traducción de las expresiones y la sentencia de asignación de C a la máquina descrita en el laboratorio 6.2.

Laboratorio 6.9 *Generación de código intermedio: Flujo de Control en Pascal*

Una vez traducidas las expresiones, el paso siguiente consiste en traducir las sentencias que modifican el flujo de control. En el caso de Pascal se trata de traducir las sentencias condicionales if-else y los bucles while.

Laboratorio 6.10 *Generación de código intermedio: Flujo de Control en C*

El objetivo de esta práctica es implementar la traducción de las sentencias que modifican el flujo de control en el compilador del lenguaje C que se ha venido desarrollando en los laboratorios anteriores. Se ha de tener en especial cuidado en la traducción de las sentencia *break* y *continue*.

Laboratorio 6.11 *Generación de código intermedio: Subprogramas en Pascal*

Finalmente sólo resta por traducir los subprogramas en Pascal. En esta práctica se establecen cuales son las acciones que se llevan a cabo en una secuencia de llamada y en una secuencia de retorno. Así como, la forma de manipular los parámetros.

Laboratorio 6.12 *Generación de código intermedio: Subprogramas en C*

La sentencia de "llamada a un subprograma" se utiliza muy a menudo, debido a esto es fundamental que un compilador genere código eficiente para manipular las llamadas y retornos de los subprogramas. El objetivo de esta práctica es ampliar el compilador del lenguaje C que se ha venido desarrollando de manera que admita la declaración e invocación de funciones generando la representación intermedia adecuada a cada caso.

3.5 La optimización de Código

El problema de la optimización consiste en eliminar la mayor cantidad de código inútil sin cambiar el significado del programa. Esta tarea debería llamarse "*mejora del código*" puesto que la palabra optimización sugiere que el código resultante debería ser óptimo y estó es muy difícil de conseguir. Se distinguen tres niveles de optimización: optimización local, que se limita al del bloque básico en curso, optimización global, que debe tener en cuenta el flujo de información entre bloque y la optimización entre procedimientos que considera el flujo de información entre procedimientos. En este módulo se proponen tres prácticas, todas ellas englobadas dentro de la optimización local e independiente de la máquina.

Laboratorio 7.1 *Bloques básicos*

Reconocer los bloques básicos en una entrada de código intermedio y representar el grafo de flujo correspondiente.

Laboratorio 7.2 *Optimizaciones locales*

Se pueden realizar una gran cantidad de mejoras en el ámbito de un bloque básico. Se propone la implementación de transformaciones que mantengan la estructura y de transformaciones algebraicas en los bloque básico representados en el laboratorio anterior.

Laboratorio 7.3 *Optimizaciones de bucles*

La mayoría de los programas invierten la mayor cantidad de sus tiempos de ejecución en los bucles, y especialmente en los niveles más profundos de los bucles anidados. En esta práctica se describen estas situaciones y se propone la implementación de posibles soluciones.

3.6 La generación de código

El objetivo de las prácticas de este módulo es la implementación de un generador de código simple considerando como máquina objeto un *Transputer*. Se ha seleccionado este procesador por la simplicidad de su juego de instrucciones.

Laboratorio 8.1 *Generación de código: Expresiones*

El objetivo de esta práctica es la traducción del código intermedio asociado a las expresiones obtenido en el laboratorio 6.8 a código para la máquina objeto, teniendo en cuenta la particularidad de que su pila es de tamaño tres.

Laboratorio 8.2 *Generación de código: Flujo de Control*

En este laboratorio, se ha de tener en cuenta el conjunto de instrucciones que ofrece la máquina objeto para evaluar las condiciones de los bucles y realizar las traducciones adecuadas.

Laboratorio 8.3 *Generación de código: Subprogramas*

Finalmente sólo resta por traducir los subprogramas. En esta práctica se han de traducir las acciones asociadas a la secuencia de llamada y a la secuencia de retorno. Así como, la forma de manipular los parámetros.

4 Conclusiones

En el Centro Superior de Informática de la Universidad de La Laguna se imparten estudios para obtener las titulaciones de Ingeniería Técnica en Informática de Sistemas (BOE 30.06.96), Ingeniería Técnica en Informática de Gestión (BOE 29.06.96) y el segundo ciclo de la Ingeniería Superior en Informática (BOE 11.01.94). En los planes de estudio de todas estas titulaciones se

Figura 1: Portada del libro

pueden encontrar asignaturas con contenidos en compiladores. La asignatura "Introducción a los Compiladores I" (6 créditos) es troncal en Sistemas y Optativa en Gestión, mientras que "Introducción a los Compiladores II" (3 créditos) es optativa en ambas titulaciones. En la Ingeniería Superior, se ha dividido la asignatura troncal "Procesadores de Lenguajes" en dos asignaturas de 5,5 créditos cada una. Los módulos que se han descrito en este trabajo constituyen las clases prácticas de estás asignaturas.

En las asignaturas de las Ingenierías Técnicas se plantea la realización de los laboratorios asociados al lenguaje Pascal. Se utiliza el método de análisis sintáctico descendente recursivo y se finaliza con la generación de código intermedio, lo que constituye un total de nueve prácticas. En la Ingeniería Superior, en "Procesadores de Lenguajes I" se propone la implementación de la fase de análisis del compilador del lenguaje C y en la segunda parte de la asignatura se aborda la generación de código.

El libro "*Prácticas de compiladores en C y Pascal*"[1] contiene una descripción detallada de cada uno de los laboratorios descritos en este trabajo. Al plantearle al alumno la realización de un laboratorio, junto con el enunciado de la práctica se le proporciona un esqueleto de programa a partir del cual empezar a trabajar.

Las prácticas de la asignatura se plantean como laboratorios abiertos, reservando las horas de prácticas para la consulta de dudas y la evaluación.

Agradecimientos

La plubicación del libro que contine la descripción detallada de los laboratorios que se resumen en este trabajo se ha realizado bajo el programa de ayudas para la publicación de textos universitarios del Gobierno Autónomo de Canarias (Figura 1).

Referencias

[1] Coromoto León. *Prácticas de Compiladores en C y Pascal.* Colección Textos Universitarios. Gobierno de Canarias. 1997.