LAS ASIGNATURAS DE PROCESADORES DE LENGUAJES EN LA UNIVERSIDAD DE LA LAGUNA

Coromoto León Hernández, Francisco de Sande González
Centro Superior de Informática. Universidad de La Laguna

***Resumen**: En este trabajo se presenta una propuesta para la enseñanza de las asignaturas de compiladores en el Centro Superior de Informática de la Universidad de La Laguna. En dicho centro se imparten en el segundo ciclo de la Ingeniería en Informática las asignaturas Procesadores de Lenguaje I y Procesadores de Lenguaje II de las cuales se aporta el contenido de sus programas de teoría y el conjunto de prácticas de laboratorio.*

1. INTRODUCCIÓN

Una asignatura con contenidos en compiladores debería mostrar cómo trabajan, cómo se organizan, cuál es la terminología, cuáles son los principales problemas y cómo han sido resueltos. Uno de los objetivos de una asignatura con estas características es proporcionar herramientas que permitan escribir un compilador simple, pero también se pretende que los alumnos sean capaces de dirigirse a libros como "Compiladores. Principios, Técnicas y Herramientas" [1], "Crafting a Compiler with C" [4] o "The Theory and Practice of Compilers Writing" [16], por citar sólo tres, y que se sientan cómodos con ellos y capacitados para encontrar las soluciones a los problemas que se les presenten.

En el Centro Superior de Informática de la Universidad de La Laguna se imparte la docencia para obtener el título de Ingeniero en Informática. Se trata de una enseñanza de segundo ciclo como queda reflejado en el plan de estudios aprobado en 1993 (11 de enero 1994, BOE núm. 9, págs. 711-716). Esta Universidad diversifica la materia troncal Procesadores de Lenguajes en dos asignaturas: Procesadores de Lenguajes I y Procesadores de Lenguajes II. Procesadores de Lenguaje I se imparte en el primer cuatrimestre, tiene un total de 4,5 créditos troncales más 1 añadido que se dividen en 3 créditos teóricos y 2,5 créditos prácticos (el crédito añadido pertenece a los prácticos). En la descripción de su contenido aparece: "Compiladores. Traductores e intérpretes. Fases de compilación.". Esta asignatura está vinculada a las áreas de conocimiento de Ciencias de la Computación e Inteligencia Artificial y a Lenguajes y Sistemas Informáticos. La descripción de Procesadores de Lenguaje II, que se imparte en el cuarto cuatrimestre, es idéntica salvo en los contenidos donde aparece: "Optimización de código. Macroprocesadores".

En este trabajo se presenta una propuesta para la elaboración del temario de la asignatura Procesadores de Lenguajes con contenidos en compiladores, traductores e intérpretes. La abundancia de técnicas para la implementación de compiladores admitiría la preparación de más de un curso avanzado. Sin embargo, el objetivo de las asignaturas es proporcionar algunas técnicas básicas de escritura de compiladores con las que se pueden construir traductores para una gran variedad de

lenguajes y máquinas. Los compiladores se suelen escribir como un conjunto de partes separadas llamadas fases. Cada fase depende una de otra y, lo más natural es que en la elaboración de los contenidos cada tema cubra una de ellas. La primera de las asignaturas en la que se ha dividido la asignatura troncal se centra en el estudio de la fase inicial de un compilador en la que se analiza el lenguaje fuente mientras que en la segunda se establece como tema central la fase de síntesis en un lenguaje objeto.

En Procesadores de Lenguaje I se estudian con detalle los distintos tipos de análisis que se le pueden hacer a un programa fuente, el análisis léxico, el análisis sintáctico y el análisis semántico, haciendo hincapié en que un buen diseño y una buena metodología de programación pueden producir una herramienta adecuada para tratar cualquier tipo de lenguaje, desde un lenguaje de alto nivel hasta un lenguaje de comandos. Procesadores de Lenguajes II asume que el análisis del programa fuente ya ha sido realizado y se centra en la tarea de traducción. Esta traducción puede ser a algún tipo de representación intermedia o bien a código máquina. Se resalta el hecho de que si la traducción se realiza de forma adecuada se pueden construir herramientas portables.

También es un objetivo de las asignaturas el fomentar el uso de herramientas de ayuda a la construcción de compiladores. En este sentido se han elaborado un conjunto de ejercicios prácticos encaminados a construir un compilador de un lenguaje sencillo (subconjuntos de lenguajes como C o Pascal) realizando el proceso de diseño e implementación a mano y también haciendo uso de tales herramientas.

Para describir los contenidos de las asignaturas se seguirá el siguiente criterio: en primer lugar se propone el título y los subtítulos de cada uno de los temas tratados y a continuación se comenta la bibliografía utilizada en la elaboración de dichos contenidos.

En los apartados primero y segundo de este artículo se presentan los contenidos que se están impartiendo en Procesadores de Lenguajes I y Procesadores de Lenguajes II respectivamente. En el tercer epígrafe se describen las prácticas de laboratorio que se han elaborado para las asignaturas y por último se enumera la bibliografía utilizada.

2. PROCESADORES DE LENGUAJES I

En esta asignatura se empieza describiendo los componentes de un compilador, para a continuación introducir los distintos tipos de análisis que se pueden realizar al programa fuente y algunas herramientas de programación que facilitan la construcción de compiladores.

Tema 1: *Conceptos de lenguajes de programación.*
Clasificación de los lenguajes de programación. Evolución de los lenguajes de programación.

En la introducción a la asignatura se realiza una clasificación de los lenguajes de programación, atendiendo a la evolución de los mismos y la incidencia de dicha evolución en los traductores. Para la elaboración de este tema se han utilizado los tres primeros capítulos del libro "High-Level Languages and Their Compilers" [18] y los cinco primeros de "Compiler Design" [19]. Se recomiendan como lecturas complementarias el libro "Conceptos de Lenguajes de Programación" [6] y "Lenguajes de Programación. Conceptos y Constructores" [15].

Tema 2: *El proceso de compilación.*
Generalidades. Conceptos de Lenguajes, sintaxis, semántica. Conceptos de traductores, compiladores e intérpretes. Esquema de un compilador.

Se introduce el esquema de compilador en el que se basan las asignaturas de Procesadores de Lenguajes. Se definen la fase de análisis del programa fuente y la de síntesis en el programa objeto y se establece como objetivo de la asignatura el estudio de la fase inicial [1], [2], [9], [19].

Tema 3: *Análisis léxico.*

Función del analizador léxico. Lexemas, expresiones regulares y componentes léxicos (*tokens*). Técnicas para la construcción de analizadores léxicos. Generación de analizadores léxicos. El generador de analizadores léxicos *lex*. El generador de analizadores léxicos *flex*. Expresiones regulares en *lex* y *flex*.

Se tratan técnicas para especificar e implementar analizadores léxicos. La forma más sencilla de crear un analizador léxico eficiente consiste en definir los componentes léxicos (*tokens*) de un lenguaje fuente y programar de forma estructurada una aplicación para encontrarlos. Otra forma consiste en utilizar un lenguaje patrón-acción para especificar los analizadores léxicos (se estudian lex y flex). En estos lenguajes, los patrones se especifican mediante expresiones regulares y sus compiladores han de ser capaces de generar un autómata finito eficiente que las reconozca.

En la elaboración de este tema se han utilizado los capítulos sobre análisis léxico de "Compiladores. Principios, técnicas y herramientas" [1] y "Compilers Design in C" [9]. Para la preparación de las prácticas de laboratorio se han tomado como base las implementaciones que se hacen del análisis léxico en "Introduction to compiler construction with Unix" [14] y "Brinch Hansen on Pascal Compilers" [3]. Además también se utiliza la excelente descripción del lenguaje *lex* del manual de "MKS lex & yacc" [5], los manuales de SunOS 5.5.1-*lex* y los de la herramienta de GNU *flex* [11].

Tema 4: *Análisis sintáctico.*

Clasificación de los analizadores sintácticos. Analizadores descendentes. Analizadores ascendentes. Analizadores LR. Generación de analizadores sintácticos. El generador de analizadores sintácticos *yacc*. Evaluación de atributos en *yacc*. Utilización conjunta de *lex* y *yacc*. Recuperación de errores sintácticos.

Una gramática proporciona una definición sintáctica precisa y fácil de entender de un lenguaje de programación. Existe una clase de gramáticas que permite construir automáticamente un analizador sintáctico eficiente para determinar si un programa fuente es o no sintácticamente correcto. Además, el proceso de construcción del analizador sintáctico puede revelar ambigüedades sintácticas y otras construcciones difíciles de analizar que de otro modo podrían quedarse sin detectar en el diseño de un lenguaje y su compilador. Este tema está dedicado al estudio de los métodos de análisis sintáctico más utilizados en compiladores. Primero se introducen los conceptos básicos, a continuación se esbozan las técnicas adecuadas para la aplicación manual y por último se estudian con detalle los algoritmos que han sido utilizados en la implementación de herramientas para la generación automática de analizadores sintácticos. Puesto que los programas pueden contener errores sintácticos se amplían los métodos de análisis sintáctico para que se recuperen de los errores más frecuentes. Se estudian los métodos de recuperación de errores en los que se añaden símbolos de sincronización y los métodos en los que se añaden producciones de error.

El material bibliográfico utilizado en la preparación de este tema es el mismo que en el del tema anterior escogiendo en este caso los capítulos sobre análisis sintáctico [1], [9]. Para la elaboración de las prácticas de laboratorio se han tomado como base la implementación del análisis sintáctico mediante la herramienta *yacc* en "Introduction to compiler construction with Unix" [14] y la implementación descendente recursiva que se encuentra en "Brinch Hansen on Pascal Compilers" [3]. También se utilizan el manual de "MKS lex & yacc" [5], el manual de SunOS 5.5-*yacc* y el de GNU-bison [11].

Tema 5: *Análisis semántico.*

Sistemas de tipos: expresiones de tipos; equivalencia estructural, nominal y funcional; sobrecarga de operadores. Tablas de símbolos: estructura de la tabla de símbolos; organización de la tabla de símbolos; lenguajes con estructura de bloque.

Se tratan la comprobación de tipos y el análisis de ámbito, que están constituidos principalmente por comprobaciones estáticas rutinarias que pueden formar parte de otras actividades (por ejemplo, de las rutinas de análisis sintáctico). En la elaboración de este tema se han utilizado el capítulo de análisis de tipo de "Compiladores. Principios, técnicas y herramientas"[1], los capítulos sobre tablas de símbolos y procesamiento de declaraciones de "Crafting a compiler with C" [4], "The theory and practice of compilers writing"[16] y de "Compilers construction" [17]. En la preparación de las prácticas de laboratorio se utiliza la misma bibliografía que en los temas anteriores [3], [14].

Tema 6: *Manipulación de Memoria.*

Organización de memoria en tiempo de ejecución. La memoria estática. La pila. El "*heap*".

Antes de considerar la generación de código, hay que relacionar el texto fuente estático de un programa con las acciones que deben ocurrir en el momento de la ejecución del programa. Se examinan las relaciones de los nombres en el texto fuente y los objetos de datos. En la elaboración de este tema se han utilizado el capítulo 9 de "Crafting a compiler with C" [4] y el tema sobre entornos en el momento de ejecución de [1].

Con estos contenidos se pretende estudiar un conjunto de técnicas que permitan realizar el diseño de un lenguaje de alto nivel e implementar el conjunto de programas necesarios para comprobar la buena sintaxis y el significado de los programas escritos con él. Finalmente se plantean algunas nociones acerca de las rutinas de apoyo en el momento de la ejecución que son necesarias para la generación de código aunque este tema se imparta en la asignatura Procesadores de Lenguajes II.

3. PROCESADORES DE LENGUAJES II

En La asignatura se introduce la traducción dirigida por la sintaxis a código intermedio y a partir de éste se estudian técnicas para la generación de código y la optimización de código independiente de la máquina.

Tema 1: *Introducción.*

Generalidades. Definiciones dirigidas por la sintaxis. Evaluación ascendente de definiciones con atributos sintetizados. Definiciones con atributos por la izquierda. Traducción descendente. Evaluación ascendente de atributos heredados.

Se desarrolla el tema de la traducción de lenguajes guiada por gramáticas independientes del contexto. Se introduce una notación alternativa a los esquemas de traducción estudiados en la asignatura previa para asociar reglas semánticas con producciones: las definiciones dirigidas por la sintaxis. Ambas notaciones se utilizan para describir la generación de código intermedio que se trata en el siguiente tema. Para el desarrollo de este tema se ha utilizado principalmente "Compiladores. Principios, técnicas y herramientas" [1] y "Compiler Design" [19].

Tema 2: *Generación de Código Intermedio.*

Lenguajes intermedios. Declaraciones. Sentencias de asignación. Expresiones booleanas. Sentencia *case*. Relleno de retroceso (*backpatching*). Llamadas a procedimientos.

En el modelo de análisis y síntesis de un compilador, la etapa inicial traduce un programa fuente a una representación intermedia a partir de la cual la etapa final genera el código objeto. Se

intenta posponer la utilización de los detalles del lenguaje objeto hasta la etapa final. Aunque la traducción de un programa fuente a un programa objeto se puede 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 tema se muestra cómo se pueden utilizar los métodos dirigidos por la sintaxis para traducir a una representación intermedia construcciones de lenguajes de programación como declaraciones, asignaciones y sentencias que modifican el flujo de control. Para la elaboración de este tema se añade a la bibliografía ya mencionada "Introduction to compiler construction" [13].

Tema 3: *Generación de Código.*
Bloques básicos y grafos de flujo. Distribución y asignación de registros. Optimización mediante mirilla. Generadores de generadores de código.

La fase final del modelo de compilador desarrollado en esta asignatura es el generador de código. Toma como entrada una representación intermedia del programa fuente y produce como salida un programa objeto equivalente [1], [4]. Para la elaboración de las prácticas se ha utilizado el libro "The Transputers Instruction Set: A Compiler Writer's Guide" [10].

Tema 4: *Optimización de Código.*
Optimización de bloques básicos. Bucles en los grafos de flujo. Introducción al análisis global del flujo de datos. Solución iterativa de las ecuaciones de flujo de control. Transformaciones para mejorar el código. Tratamiento de alias. Análisis de flujo de datos de grafos de flujo estructurados. Algoritmos eficientes para el flujo de datos.

Los compiladores deberían producir código objeto que fuera tan bueno como el escrito a mano. Esto supone que se pretendan conseguir objetivos contrapuestos, pues a menudo se pide que el código directamente producido por los algoritmos de compilación se ejecuten más rápidamente o que ocupe menos espacio, o ambas cosas a la vez. En este tema se consideran las principales optimizaciones independientes de la máquina, que son transformaciones de programas que mejoran el código objeto sin tener en cuenta las propiedades de la máquina objeto [1], [4], [13].

Finaliza así la descripción del contenido de las clases teóricas asociadas a las asignaturas. Se trata de unos contenidos clásicos en las asignaturas de compiladores en los que se ha intentado intensificar el desarrollo de prácticas de laboratorios. En la elaboración de los ejercicios 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.

4. PROGRAMA DE PRÁCTICAS

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. Con estas asignaturas 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 proponen el uso de herramientas que faciliten el diseño y la implementación de compiladores como son las herramientas *lex* y *yacc*.

El lenguaje de programación utilizado principalmente en las prácticas de la asignatura es C pero puede utilizarse cualquier otro lenguaje imperativo como Pascal o Modula-2.

El programa de prácticas de la asignatura Procesadores de Lenguajes I es el siguiente:

Práctica #1: *Modo de uso de la herramienta MKS-lex.*
Mediante un ejemplo sencillo se muestra la forma de funcionamiento de esta herramienta para la ayuda a la construcción de analizadores léxicos sobre DOS [5].

Práctica #2: *Modo de uso de las herramientas SunOS 5.5.1-lex y flex.*
Se utiliza un ejemplo sencillo que indique la forma de funcionamiento de estas herramienta de construcción de analizadores léxicos en el entorno Unix y ponga de manifiesto las diferencias con el utilizado en la práctica anterior [11].

Práctica #3: *Reconocimiento de componentes léxicos. Aplicación (I).*
Construir el analizador léxico de una línea de comandos en la que se puedan especificar distintas opciones [11]. Utilizar las herramientas MKS-lex, SunOS 5.5.1-lex y flex.

Práctica #4: *Reconocimiento de componentes léxicos. Aplicación (II).*
Construir un analizador léxico para un lenguaje sencillo, escribiendo el programa en un lenguaje de alto nivel. Se propone como lenguaje a analizar "*Pascal-*" [3].

Práctica #5: *Reconocimiento de componentes léxicos. Aplicación (III).*
Diseño e implementación de un analizador léxico para un lenguaje sencillo usando un generador de analizadores léxicos. Se propone "*Sample C*" [14] como lenguaje a tratar.

Práctica #6: *Modo de uso de las herramientas: MKS-yacc.*
Mediante un ejemplo sencillo se muestra la forma de funcionamiento de esta herramienta para la ayuda a la construcción de analizadores sintácticos para DOS [5].

Práctica #7: *Modo de uso de las herramientas: SunOS 5.5.1-yacc y bison.*
La implementación de una calculadora muestra la forma de funcionamiento de esta herramienta para la ayuda a la construcción de analizadores sintácticos en Unix [11].

Práctica #8: *Reconocimiento de lenguajes. Aplicación (I).*
Generar un analizador sintáctico para un lenguaje sencillo utilizando *yacc* [14].

Práctica #9: *Reconocimiento de lenguajes. Aplicación(II).*
Generar un analizador sintáctico para una lenguaje sencillo programando usando el método descendente recursivo [3].

Práctica #10: *Reconocimiento de lenguajes. Aplicación(III).*
Generar un analizador sintáctico para una lenguaje sencillo utilizando *yacc* y ampliarlo con acciones semánticas que permitan darle un formato determinado al fichero de entrada [14].

Práctica #11: *Recuperación de errores (I).*
Modificar el analizador sintáctico escrito para que sea capaz de recuperarse de los errores que se produzcan. Utilizar la técnica de inclusión de producciones de error [14].

Práctica #12: *Recuperación de errores(II).*
Ampliación del analizador sintáctico por descenso recursivo, ya implementado, introduciendo conjuntos de símbolos de sincronización para recuperación de errores [3].

Práctica #13: *Restricciones semánticas. Análisis de ámbito.*
Añadir los atributos y acciones semánticas al analizador sintáctico escrito con yacc para realizar el análisis de ámbito. La comprobación de tipos en este caso es muy sencilla puesto que sólo se admiten enteros [14].

Práctica #14: *Restricciones semánticas. Análisis de tipo.*
Partiendo del analizador sintáctico descendente recursivo con recuperación de errores desarrollado realizar las modificaciones necesarias para que tenga lugar el análisis de tipo. En este caso se cuenta con los tipos entero, boolean, array y registro [3].

Práctica #15: *Asignación de Memoria.*
Seleccionar e implementar las estructuras necesarias que permitan realizar la asignación de memoria [14].

En la Universidad de La Laguna al segundo ciclo que constituye la Ingeniería Superior en Informática, se accede directamente desde un primer ciclo, en Ingeniería Técnica en Informática de Sistemas en la que existe una asignatura obligatoria denominada Introducción a los Compiladores, o bien, se puede acceder desde la Ingeniería Técnica en Informática de Gestión en la que no existe esta asignatura. Los alumnos que cursan Introducción a los Compiladores han realizado unas prácticas para la implementación de un compilador utilizando el método descendente recursivo.

Al alumno se le presenta un conjunto de prácticas de entre las cuales puede elegir un conjunto que le permitan tener un compilador, ya sea utilizando herramientas de generación automática o bien una implementación a mano. Así pues, finalmente el conjunto de prácticas realizadas puede ser: 1,2,3,4,6,7,9,10,12,14,15 o bien 1,2,3,5,6,7,8,10,11,13,15. El material que se le proporciona al alumno consiste en el esqueleto del programa a realizar, especificando los nombres de los subprogramas a implementar y cual es la entrada y la salida de dichas rutinas. Se estima que en la realización de cada práctica el alumno ha de invertir como promedio entre dos y tres horas.

La propuesta de prácticas para la asignatura Procesadores de Lenguajes II es:

Práctica #1: *Lenguajes Intermedios.*
Implementación de un intérprete de código intermedio [3], [14].

Práctica #2: *Generación de código intermedio (Expresiones).*
Partiendo de la etapa inicial (*front-end*) de un compilador para un lenguaje sencillo, ampliarlo con la generación de un código intermedio para el tratamiento de expresiones [1], [14].

Práctica #3: *Generación de código intermedio (Tipos estructurados).*
Ampliar la etapa inicial (*front-end*) de un compilador para un lenguaje en el que sólo se reconozcan tipos enteros con la generación de un código intermedio para el tratamiento de arrays y registros [14].

Práctica #4: *Generación de código intermedio (Flujo de Control I).*
Resolver el problemas de las referencias hacia delante usando dos pasadas [3], [14].

Práctica #5: *Generación de código intermedio (Flujo de Control II).*
Solucionar el problema de las referencias hacia delante utilizando relleno con retroceso (*back-patching*) [1].

Práctica #6: *Generación de código intermedio (Subprogramas).*
Admisión de la declaración y llamada a funciones, generando la representación intermedia en cada caso [4].

Práctica #7: *Generación de código (Expresiones).*
Partiendo de un código intermedio generado en las prácticas anteriores implementar la generación de código para expresiones en un *transputer* [10].

Práctica #8: *Generación de código (Sentencias).*
A partir de un código intermedio implementar la generación de código para un *transputer* de las sentencias que modifican el flujo de control [10].

Práctica #9: *Generación de código (Subprogramas).*
Generar código para un transputer para las llamadas a subprogramas, partiendo de una representación intermedia [10].

Práctica #10: *Optimización de código.*
Implementar técnicas de optimización del código para una máquina hipotética [1], [13].

El las prácticas de esta asignatura se genera código para transputers puesto que los alumnos ya han estudiado dicha arquitectura en las asignaturas de Estructuras de Ordenadores. Sin embargo, se les permite generar código para cualquier otra máquina y en ese caso se recomienda utilizar un ensamblador para 80x86 [8], [12]. Para la elaboración de las prácticas en las que se desarrolla la escritura de un compilador a mano también se pueden utilizar como base otros lenguajes aparte del "*Pascal-*" [3] propuesto, como son el subconjunto de Pascal de [12] y el "*Small C*" de [7]. En "Introduction to compiling techniques" [2] se propone la escritura de un compilador para un lenguaje sencillo utilizando lex y yacc y que puede servir como alternativa a "*Sample C*" [14].

5. REFERENCIAS

[1] Aho, A.; Sethi, R.; Ullman, J., "COMPILADORES. Principios, Técnicas y Herramientas", Adisson-Wesley Iberoamericana, 1990.

[2] Bennett, J. P., "INTRODUCTION TO COMPILING TECHNIQUES. A First Course using ANSI C, LEX and YACC", McGraw-Hill, 1990.

[3] P. Brinch Hansen , "Brinch Hansen on Pascal Compilers", Prentice-Hall, 1985.

[4] Fischer, C.N., LeBlanc, R.; "Crafting a compiler with C". The Benjamin/Cummings Publishing Company, Inc. 1991.

[5] Gardner, J.; Linseman, A.; Nicol, S.; Retterath C. "MKS LEX & YACC", Mortice Kern Systems Inc, 1992.

[6] Ghezzi, C.; Jazayeri, M., "Conceptos de Lenguajes de Programación", Ediciones Díaz de Santos, 1986.

[7] Hendrix, J., "A Small C Compiler: Language, Usage, Theory and Design". M&T Pubishing.1988.

[8] Hendrix, J. "Small Assembler. An 80x86 Macro Assembler Written in Small C", M&T Publishing. 1988.

[9] Hollub, Allen, "Compiler Design in C", Prentice-Hall, 1990.

[10] INMOS Limited; "Transputer Instruction Set. A compiler writer's guide", Prentice Hall, 1988.

[11] Levine, J.; Mason, T.; Brown, D., "LEX & YACC", O'Reilly & Associates, Inc. 1990.

[12] Mak, R., "Writing Compilers and interpreters. An Applied Approach". John Wiley & Sons, Inc. 1991.

[13] Parsons, T., "Introduction to Compiler Construction". Computer Science Press.1992.

[14] Schreiner, A.; Friedman, Jr. H., "INTRODUCTION TO COMPILER CONSTRUCTION WITH UNIX". Prentice-Hall, 1985.

[15] Sethi, R., "Lenguajes de Programación. Conceptos y Constructores". Adisson Wesley Iberoamericana. 1992.

[16] Tremblay, J.; Sorenson, P. "The theory and practice of compilers writing". McGrawHill Computers Science Serie. 1988.

[17] Waite, W.; Goos, G. "Compilers construction". Springer-Verlag. 1985.

[18] Watson, D., "High-Level Languages and Their Compilers", Addison-Wesley, 1989.

[19] Wilhelm, R.; Maurer, D., "Compiler Design". Addison-Wesley. 1995.