Automatic Assessment of Programming Projects in a Compiler Construction Course

J. Velásquez, *Senior Member, IEEE*

***Abstract*—Techniques for compiler construction are usually taught in companion of a capstone project that has the aim of build a mini-language. However, it seems that students will never build a real compiler during their professional life, but it is common the implementation of tools as interpreters, translators, code generators, among others. In this article, we describe how to introduce several programming projects as part of a traditional compiler construction course where a very simple compiler for a mini-language is used as the cornerstone for a capstone project. The complementary projects allow us to expose students to the practical application of techniques for compiler construction and encourage the development of programming and problem-solving skills. In addition, we discuss the implementation of a scheme for automatic assessment using the Virtual Programming Lab plugging of the Moodle project. Student evaluation of the course shows a strong agreement about high course preparation, high motivation for the development of projects, total impartiality for grading, and high connection with the subjects of other courses.**

***Index Terms*—Classroom examples, Compiler construction, Interpreters, Programming languages design, Virtual machines.**

I. Introducción

ESTE artículo describe los proyectos de programación usados en la evaluación de un curso introductorio de Teoría de la Computación y las estrategias usadas para su calificación automática usando el *Virtual Programming Lab* (VPL) del proyecto Moodle [1]. Estos proyectos incluyen la implementación de traductores, intérpretes, generadores de código y otras herramientas basadas en las mismas técnicas usadas en la construcción de compiladores.

La enseñanza de cómo implementar un compilador ha sido tradicionalmente uno de los objetivos primordiales en el currículo de las ciencias de la computación; sin embargo, las asignaturas que tienen este objetivo también han sido reconocidas por la complejidad de los temas abordados [2][3][4] y la dificultad inherente de los proyectos de programación. Un factor común en estos cursos es el desarrollo de un mini-lenguaje cuya finalidad es la puesta en práctica de todas las técnicas y herramientas en que se fundamenta la construcción de compiladores. La mayor parte de los proyectos de construcción de compiladores e intérpretes en el aula de clase aparecen propuestos en los libros de texto sobre esta temática [2], aunque también se han publicado algunos proyectos de aula desarrollados por docentes, tal como es el caso de los mini-lenguajes COOL [5] y Bantam Java [6]. Sin embargo, todas estas aproximaciones siguen la misma receta: "el objetivo de este proyecto es desarrollar un compilador para el mini-lenguaje…". Claramente esta aproximación se ve limitada por la dificultad de los temas mismos, la falta de familiaridad de los estudiantes con el lenguaje de programación (usualmente C [7], Java [8] [9] o ML [10]), y la complejidad del proceso de implementación de las componentes del compilador.

Más aún, la idea central de desarrollar un compilador como parte de los cursos ha sido cuestionada recientemente, toda vez que este tema es realmente complejo y muchos de los estudiantes nunca desarrollarán compiladores en su vida profesional [3]. Pero sí resulta muy probable que se desarrollen otras herramientas basadas en los mismos principios conceptuales en que se basan los compiladores —como lectores de archivos de configuración, intérpretes dirigidos por sintaxis, intérpretes basados en árboles sintácticos, traductores y generadores de código entre otros [11] — y en lenguajes diferentes a los que se han usado tradicionalmente en la enseñanza, tales como Python, Perl, C#, R, Scala, Ruby o Matlab. En este sentido, el autor sólo conoce dos trabajos que se diferencian en gran medida de los textos tradicionales para la enseñanza de la teoría de la compilación: Parr [11] aplica los conceptos fundamentales de la teoría de la computación a la implementación práctica (en el lenguaje Java) de traductores, intérpretes y generadores de código; Frenz [12] describe diferentes estrategias para construir analizadores sintácticos en el lenguaje Perl, los cuales tienen como fin el analizar contenido web, documentos XML y ejecutar minería de datos y texto.

Otra de las dificultades en los cursos es el tema evaluativo; la gran mayoría de los cursos dedican una parte muy importante de su esfuerzo a la evaluación de conceptos teóricos, pero la parte práctica es evaluada como todo o nada, o, en otras palabras, funciona o no funciona; esto se debe a que la evaluación detallada de todos los aspectos de la implementación de un mini-lenguaje resulta demasiado tediosa y consume demasiado tiempo si no puede ser automatizada de alguna forma. En este mismo sentido, en algunos cursos solo se aborda el desarrollo de una o unas pocas componentes del compilador (usualmente las más fáciles).

Este artículo tiene dos objetivos: el primero, es plantear una serie de proyectos alternativos (o complementarios) a la implementación de un compilador para un mini lenguaje, los

J. D. Velásquez, Profesor Titular, Universidad Nacional de Colombia, Sede Medellin, Colombia (e-mail: jdvelasq@unal.edu.co).

cuales pueden ser fácilmente implementados como aplicaciones de línea de comandos [13]. Para ello, se parte de la implementación de las herramientas que conforman el mini-lenguaje bcc que es descrito en [2]. A partir de las aplicaciones que lo conforman o los resultados parciales del proceso de compilación es posible implementar manualmente diferentes herramientas que permiten construir muchos ejemplos y prácticas de clase que incluyen analizadores recursivos descendentes de uno o más tokens adelante, generadores de árboles sintácticos, intérpretes dirigidos por sintaxis, intérpretes basados en árboles sintácticos, máquinas virtuales de pila y de registros, optimizadores de código, perfiladores de código, analizadores estáticos y traductores basados en sintaxis, entre otros. El desarrollo de estos proyectos alternativos expone a los estudiantes a todo un ecosistema de herramientas que se nutren de los mismos conceptos y técnicas clásicas que se usan en el desarrollo de compiladores, permitiendo una mayor aprehensión de conocimientos teóricos y el desarrollo de habilidades prácticas. La implementación de estas herramientas por parte de los estudiantes está acorde con la sugerencia de Debray [3], en la que se señala que deberían abordarse proyectos de programación más acordes con lo que los estudiantes realizarán durante su vida profesional.

El segundo objetivo de este trabajo es describir cómo las tareas y prácticas desarrolladas a partir de las herramientas anteriores pueden ser calificadas automáticamente en el sistema Moodle usando el VPL. Esto permite que cada implementación pueda ser evaluada en su totalidad en una forma rápida y eficaz y sin sesgos por parte del calificador, lo cual ya es ventajoso cuando el docente debe evaluar grupos pequeños de estudiantes, pero que resulta invaluable cuando los grupos son grandes y hay muchas tareas de programación durante el semestre; en este sentido, se puede evaluar de forma simultánea y masiva a un gran número de estudiantes. Más aún, el uso de un sistema automático de calificación incrementa la objetividad de la evaluación, ya que todos los alumnos son calificados exactamente bajo las mismas reglas; el costo de aplicación y calificación es muy bajo, en comparación a la cantidad de recursos requeridos cuando se realiza la calificación manual.

El resto de este artículo está organizado como sigue. La Sección II presenta el lenguaje `bcc` y las herramientas que conforman el compilador. La Sección III describe las aplicaciones que configuran los proyectos alternativos. El esquema de evaluación automática es discutido en la Sección IV. Seguidamente en la Sección V se discute su implementación en el aula. Finalmente, se concluye en la Sección VI.

II. El Lenguaje bcc

A. Descripción

La descripción presentada aquí es basada en [2]. El mini-lenguaje `bcc` es imperativo y fuertemente tipado. Permite datos escalares de tipo numérico (`num`) o booleano (`bool`). Su sintaxis es similar a la de la aplicación `bc` de los sistemas Unix, pero también incorpora elementos del lenguaje BCPL. Su diseño es completamente modular, de tal forma que es posible modificar y verificar cada módulo de forma independiente. Ya que está escrito en Python es completamente portable a diferentes plataformas. Tiene una estructura similar a la utilizada en los lenguajes Basic y Fortran, que obliga a que las funciones deban ser definidas antes de ser llamadas y a estar ubicadas antes del programa principal. `bcc` utiliza un modelo de memoria completamente estático, tal como en el Basic y en el Fortran; en este modelo las funciones no pueden ser llamadas de forma recursiva (ya sea en forma directa o indirecta). Estos lineamientos permiten simplificar enormemente el diseño de las máquinas virtuales que realizan la ejecución del código.

B. Gramática

El lenguaje `bcc` usa un analizador sintáctico recursivo descendente que requiere analizar a lo sumo dos tokens adelante. Su gramática es presentada en la Fig. 1. Ella no presenta ambigüedades ya que el token actual permite decidir con cuál regla gramatical se debe continuar realizando el análisis sintáctico. Nótese que la gramática es bastante simple, por lo que puede ser codificada fácilmente a mano.

C. Implementación

El lenguaje `bcc` está conformado como un conjunto de aplicaciones independientes y modulares que realizan las distintas fases del proceso de compilación. La programación fue realizada en Python 3. La estructura general de la suite de herramientas es presentada en la Fig. 2(a) y fue desarrollada originalmente en [2].

Los programas que realizan el proceso de compilación son los siguientes:

`yylex.py`	analizador léxico.
`yyparse.py`	analizador sintáctico.
`yysymboltab.py`	generador de la tabla de símbolos.
`yytypecheck.py`	verificador de tipos.
`yyrbm.py`	generador de código ensamblador de tres direcciones para una máquina virtual basada en registros.
`yysbm.py`	generador de código ensamblador de cero direcciones para una máquina virtual basada en pila.
`rbm.py`	máquina virtual basada en registros.
`sbm.py`	máquina virtual basada en pila.

III. Proyectos Alternativos

En la Fig. 2(b) se presentan la dependencia entre los módulos de la suite de aplicaciones considerando los proyectos alternativos que son la parte central de este trabajo. Estos proyectos no requieren que se modifique la funcionalidad original de la suite `bcc` y se pueden implementar simultáneamente con las demás componentes requeridas para construir el intérprete de referencia. Dentro de los proyectos propuestos no se ha considerado el desarrollo de generadores automáticos de analizadores léxicos y sintácticos, ya que no pueden ser desarrollados juntamente con otros proyectos en un curso de un semestre académico de duración. A continuación, se describen los proyectos alternativos propuesto.

A. Grepc: Clon de Grep

Esta aplicación no aparece en la Fig. 2 ya que es una herramienta independiente. Ella emula el comportamiento básico de la herramienta grep de los sistemas Unix. Sólo permite un subconjunto de la notación usada en `grep` para definir expresiones regulares.

```
prog
   : fn_decl_list main_prog

var_decl
   : ID ':' DATATYPE  [',' ID ':' DATATYPE]*

fn_decl_list
   : [ FUNCTION FID ':' DATATYPE '(' [var_decl] ')'
       [ '[' lexp ']' ] VAR var_decl ';']
       stmt_block ]*

stmt_block
   : '{' stmt+ '}'
   | stmt

stmt
   : PRINT lexpr ';'
   | INPUT ID ';'
   | WHEN  '(' lexpr ')' DO stmt_block
   | IF '(' lexpr ')' DO stmt_block ELSE stmt_block
   | UNLESS '(' lexpr ')' DO stmt_block
   | WHILE '(' expr ')' DO stmt_block
   | RETURN  expr ';'
   | UNTIL '(' lexpr ')' DO stmt_block
   | LOOP stmt_block
   | DO stmt_block WHILE '(' lexpr ')'
   | DO stmt_block UNTIL '(' lexpr ')'
   | REPEAT NUM ':' stmt_block
   | FOR '(' lexpr ';' lexpr ';' lexpr ')' DO
stmt_block
   | END ';'
   | NEXT ';'
   | BREAK ';'
   | ID ':=' lexpr ';'
   | ID '+=' lexpr ';'
   | ID '-=' lexpr ';'
   | ID '*=' lexpr ';'
   | ID '/=' lexpr ';'
   | ID '%=' lexpr ';'
   | ID '++' ';'
   | ID '--' ';'
   | '--' ID ';'
   | '++' ID ';'

lexpr
   : nexpr [[AND nexpr]* | [OR nexpr]*]

nexpr
   : NOT '(' lexpr  ')'
   | rexpr

rexpr
   : simple_expr [('<'|'=='|'<='|'>'|'>='|'!=')
simple_expr]

simple_expr
   : term [('+'|'-') term]*

term
   : factor [('*'|'/'|'%') factor]*

factor
   : NUM
   | BOOL
   | ID ['++'| '--']
   | ['++'|'--'] ID
   | ID
   | '(' expr ')'
   | FID '(' [lexpr [',' lexpr]*]  ')'

main_prog
   : [VAR var_decl ';'] stmt* END
```

Fig. 1. Gramática del lenguaje bcc.

B. Bcc2python: Traductor Dirigido por Sintaxis de bcc a Python

Traduce de forma directa las instrucciones de un programa escrito en el lenguaje `bcc` a Python sin construir la tabla de símbolos ni realizar la verificación de tipos. Genera un archivo de texto ejecutable directamente en Python.

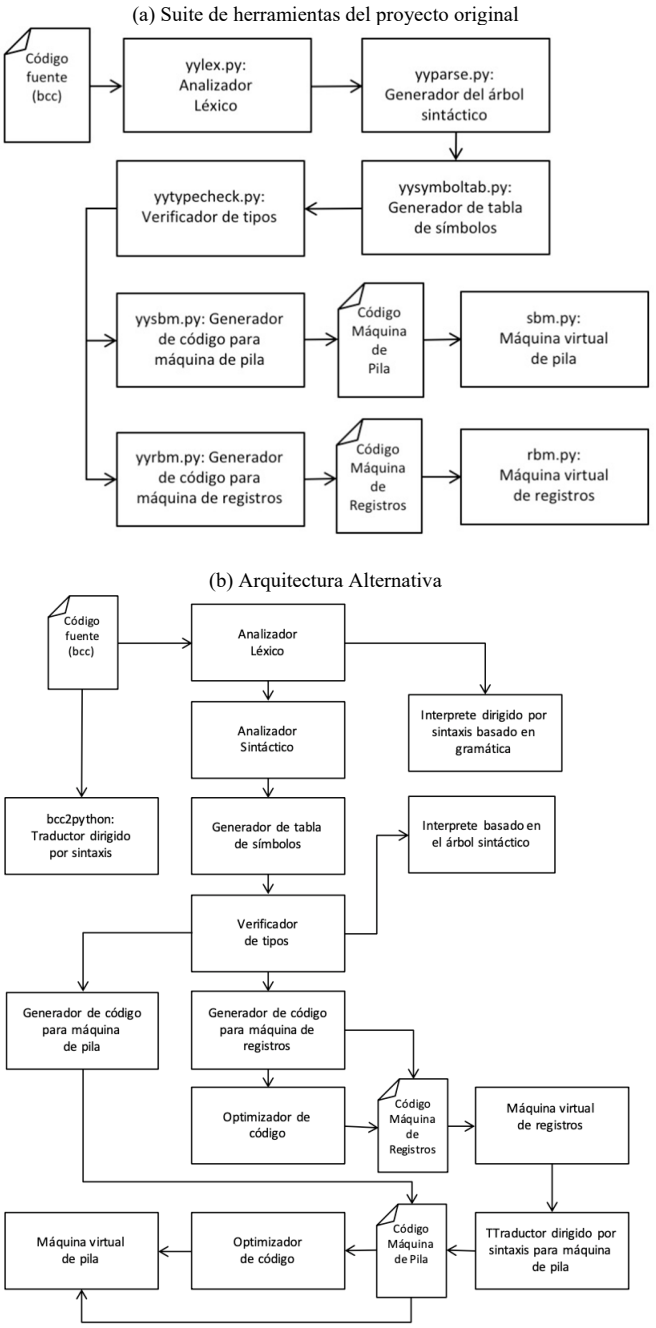

Fig. 2. (a) Arquitectura del proyecto del compilador del lenguaje bcc, y (b) Suite de herramientas con proyectos alternativos.

C. Intérprete Dirigido por Sintaxis Basado en Gramática

Este es un intérprete poco robusto que puede emplearse para lenguajes simples. La diferencia con los demás intérpretes considerados en esta suite es que la gramática misma contiene

la codificación de las acciones directamente en Python; de esta forma, las instrucciones del lenguaje `bcc` van siendo ejecutadas de forma simultánea con el análisis sintáctico. Esta implementación permite ejemplificar lo poco conveniente que puede resultar la mezcla de la fase de reconocimiento con la fase de ejecución, aunque en ciertos casos se puede justificar su uso.

D. Intérprete Basado en el Árbol Sintáctico.

Esta aplicación recibe el árbol sintáctico obtenido después de realizar la construcción de la tabla de símbolos y la verificación de tipos. Durante la ejecución, se recorren los nodos del árbol y dependiendo del tipo de nodo se ejecutan las instrucciones pertinentes. Nótese que, en la aplicación anterior, las instrucciones se ejecutan a medida que se realiza el análisis sintáctico mientras que en esta el árbol ya está construido. Para esta implementación en particular, las instrucciones están codificadas en lenguaje interpretable por alguno de los dos tipos considerados de máquina virtual. Otro punto crucial que diferencia este tipo de intérprete con los demás, es que no hay generación de instrucciones que luego serán ejecutadas en una etapa posterior.

E. Traductor Dirigido por Sintaxis de Código de Máquina Basada en Registros a Código de Máquina Basada en Pila

Esta es una aplicación muy simple que convierte código de tres direcciones a código de cero direcciones. Aunque es muy poco común que un intérprete conviva con los dos tipos de máquinas simultáneamente, el proyecto resulta interesante en el sentido que permite reforzar los conceptos desarrollados. El código generado puede ser ejecutado directamente por la aplicación `sbm.py`.

F. Optimizador de código de cero direcciones para máquina de pila

Esta es una aplicación complementaria al optimizador de código de tres direcciones. En este caso resulta interesante el análisis de cómo optimizar código de cero direcciones basándose en las optimizaciones típicas aplicadas al código de tres direcciones.

IV. Esquema de Evaluación usando el VPL

A. Herramienta VPL

Esta es una herramienta gratuita distribuida bajo la licencia GNU/GPL que permite la calificación automática de programas de computador escritos en diferentes lenguajes, tales como C, Java, Perl, Python y Ruby, entre otros. El VPL es un módulo que adiciona funcionalidad al sistema Moodle. Para usar el VPL, el instructor define una tarea de programación cuyo objetivo es que el estudiante remita un código fuente que realice un proceso específico. Después de subida la tarea, el módulo VPL la ejecuta en un *jail server* por fuera de Moodle, con el fin de evitar la caída del servidor de Moodle si hay un problema con el compilador. Durante la evaluación, el programa del estudiante es ejecutado para diferentes casos de prueba que difieren en los datos de entrada y para los que se conoce cuál es la salida correcta del programa. La calificación se realiza al comparar la salida entregada por el código del estudiante y la salida correcta especificada en la definición de cada caso de evaluación [1].

Cada tarea de programación puede ser personalizada de acuerdo con las opciones de configuración que tiene el VPL; la personalización incluye, entre otros, limitar la cantidad de archivos que el estudiante puede subir, restringir el periodo de entrega u obligar a que se tenga que programar directamente en el navegador de Internet (deshabilitando las opciones de copiar y pegar y de subir un archivo). Igualmente, también es posible indicar si la evaluación se hace automáticamente con el envío de la tarea por parte del estudiante, o si se realiza después de finalizar el periodo de envíos [1].

El proceso de evaluación realizado por el VPL puede ser personalizado a través de tres archivos que lo controlan:

- `vpl_run.sh`: contiene el código que prepara la ejecución del programa remitido.
- `vpl_evaluate.sh`: contiene el código para realizar la evaluación.
- `vpl_evaluate.cases`: que contiene la definición de los casos de prueba. Cada caso está caracterizado por su título, la entrada y la salida deseada.

En la tarea más simple, se solicita al estudiante que suba un archivo de nombre predefinido cuya extensión es acorde al lenguaje en que se desea que se haga la compilación (por ejemplo, py para Python), y se define, al menos, un caso de evaluación. El VPL, de acuerdo con la extensión del archivo, realiza la ejecución del programa utilizando el compilador correspondiente.

No obstante, la aproximación anterior no es suficiente para los proyectos de programación abordados en este artículo y se requiere modificar los archivos que controlan la evaluación. En [14] se presentan varios ejemplos de cómo personalizar los archivos de configuración de la evaluación. Para ejemplificar la configuración de la evaluación, suponga que se pide al estudiante que suba el archivo `sbm.txt` que contiene código ejecutable por la máquina de cero direcciones (aplicación `sbm.py`). El proceso de configuración del VPL es el siguiente:

1. Se indica que hay un archivo requerido, llamado `sbm.txt`.

2. Se agrega el siguiente contenido al archivo `vpl_run.sh`:

```
#! /bin/bash
echo "#! /bin/bash" > vpl_execution
echo "/usr/bin/python3 sbm.py" >> vpl_execution
chmod a+x  vpl_execution
```

   La primera línea indica que el archivo `vpl_run.sh` contiene un programa en el lenguaje del intérprete bash. Las siguientes dos líneas crean el archivo `vpl_execution`, en cuyo contenido se indica que se debe compilar el código fuente de la máquina virtual (`sbm.py`) usando Python. La cuarta línea cambia los permisos del archivo `vpl_execution` haciéndolo ejecutable.

3. La implementación de la máquina virtual (`sbm.py`) debe modificarse para que abra el archivo con la tarea del

estudiante (`sbm.txt`) y la ejecute.

4. En la configuración se debe indicar que el archivo `sbm.py` se mantiene durante la ejecución del script de evaluación.

Igualmente, es posible modificar `vpl_evaluate.sh` para crear programas que evalúen de forma personalizada la entrega de los estudiantes.

B. Proyectos de Programación

En las distintas tareas asignadas, el estudiante debe implementar totalmente o completar las herramientas a partir del código semilla entregado. A continuación, se describen las tareas asignadas a los estudiantes a partir de la arquitectura presentada en la Fig. 2.

1) Aumentar las Expresiones Regulares Soportadas por Grepc

Tal como ya se indicó, se entrega una versión de grepc con funcionalidad limitada (soporta únicamente los operadores `^`, `$`, `.` y `*`); los estudiantes deben ampliar la funcionalidad para permitir rangos de caracteres mediante el uso de corchetes cuadrados (`[` y `]`), la negación (no contiene) de rangos de caracteres y otros operadores que puedan resultar de interés. En el sistema VPL se definen diferentes búsquedas sobre uno o más archivos predefinidos; la calificación se realiza comparando la salida del código remitido por el estudiante con el resultado correcto.

2) Programación en Código Máquina de Cero Direcciones

Esta tarea tiene como fin que el estudiante se familiarice con el lenguaje de programación de la máquina virtual basada en una pila. En este caso se entrega un algoritmo simple que use condicionales e iteraciones (entre 15 y 25 líneas de código en Python) y se solicita al estudiante que suba un archivo de texto con nombre predefinido que contiene el código equivalente en el lenguaje para una máquina virtual de cero direcciones. El programa en Python debe permitir la entrada de datos de usuario con el fin de realizar la evaluación para diferentes casos de prueba que tienen entradas diferentes. Al VPL se sube una máquina virtual personalizada que cuando se ejecuta busca el archivo entregado por el estudiante e interpreta el código en lenguaje ensamblador.

3) Traductor de bcc a Python

Se pide al estudiante que implemente (o complete) un traductor del lenguaje `bcc` a Python. Como casos de prueba se suministran varios programas simples en `bcc` y el resultado de ejecutar el código equivalente en Python. Para cada caso de prueba, el VPL ejecuta el traductor suministrado por el estudiante y luego ejecuta el código generado en Python. La calificación se basa en que el código traducido genere una salida igual a la suministrada como ejemplo.

4) Intérprete Dirigido por Sintaxis Basado en Gramática

El estudiante debe completar el código de esta aplicación. El estudiante sube como tarea su código fuente y el VPL lo ejecuta, para luego comparar los resultados obtenidos contra los predefinidos para cada caso de prueba suministrado.

5) Intérprete Basado en el Árbol Sintáctico

Al igual que en el caso anterior, el estudiante debe completar el código semilla suministrado para esta aplicación. El VPL compila varios programas de prueba usando este intérprete y compara la salida obtenida contra la predefinida para cada caso.

6) Optimizador de Código de Cero Direcciones

En este caso se usa una máquina de cero direcciones personalizada que cuenta las instrucciones. Se verifica que el código optimizado entregue la misma salida que el código sin optimizar, y se asigna una nota de acuerdo con la cantidad de instrucciones que se lograron eliminar del código suministrado.

7) Traductor Dirigido por Sintaxis de Código de Tres Direcciones a Código de Cero Direcciones

El estudiante debe implementar en su totalidad (o completar) este traductor. Para la evaluación, la aplicación debe traducir un programa con instrucciones de cero direcciones a su equivalente en tres direcciones. El código traducido es ejecutado con la correspondiente máquina virtual y la nota se asigna con base en la concordancia entre la salida del código traducido y el código original.

V. Implementación dentro de un Curso y Resultados

A continuación, se describe el proceso de implementación dentro del aula en el contexto del curso. Como ejemplo se usará el proyecto de programación correspondiente a la emulación de la herramienta grep de los sistemas Linux. Para ello, el docente prepara una versión mínima de `grepc` que implementa un conjunto mínimo de operadores `^`, `$`, `+` y `*`. Esta versión es usada en el aula para explicar el tema de reconocimiento de texto usando expresiones regulares y tiene suficiente funcionalidad para preparar ejemplos que permitan demostrar su utilidad práctica. Esta versión del código es suministrada a los estudiantes como código semilla. Para el proyecto de aula, el docente solicita aumentar la funcionalidad de `grepc` agregando otros operadores para la especificación de expresiones regulares, como, por ejemplo, los presentes en el lenguaje Perl.

Para la evaluación, el docente también desarrolla su propia versión completa de `grepc` e implementa el esquema de evaluación descrito en la Sección IV, en el cual se contrastan los resultados generados por la versión desarrollada por el estudiante. En otras palabras, se contrastan los resultados de la implementación del estudiante contra los resultados de la versión del docente y se emite una calificación basada en dicha comparación.

Ya que el código del estudiante seguramente contendrá errores en su primera versión, se permite que el estudiante suba el código un número ilimitado de veces, realizando la evaluación en cada subida. En este punto se produce la retroalimentación a la tarea del estudiante.

Este esquema ha sido probado en el aula por cinco semestres académicos y los resultados obtenidos se han contrastado contra la versión original del curso, en la que se hacia el desarrollo

típico de un interprete como proyecto de clase. Los resultados muestran una gran aceptación de este esquema por parte de los estudiantes. En primer lugar, hay un acuerdo total en que hay una muy buena preparación de cada una de las sesiones de clase. Segundo, hay una alta motivación en relación con el reto que implica cada uno de los proyectos; esto se deba a que es posible que el estudiante remita varias veces un mismo proyecto y reciba retroalimentación automática por parte del sistema que evalúa el código, lo que permite la mejora continua del producto en relación con un esquema en se entrega una sola vez. Tercero, el sistema de calificación es automático promoviendo la imparcialidad en la asignación de la calificación del proyecto. Y cuarto, la realización de los proyectos propuestos incentiva la conexión del contenido curso con otros contenidos desarrollados en otras asignaturas.

VI. Conclusiones

En este trabajo se apoya la tesis de Debray [3] en la que se postula la necesidad de que los cursos orientados a la construcción de compiladores deben proveer ejemplos relacionados con la práctica profesional, ya que en su vida laboral los estudiantes nunca desarrollarán un compilador real. En la práctica profesional, las técnicas y herramientas para la construcción de compiladores tienen una clara aplicabilidad en el desarrollo de aplicaciones tales como los lectores de archivos de configuración, traductores de código y de formato, generadores e intérpretes de código. En este trabajo, se discute cómo se puede enriquecer la variedad de ejemplos, tareas de programación y proyectos finales a través de la incorporación de ocho herramientas basadas en las técnicas propias de la construcción de compiladores. Ya que uno de los grandes inconvenientes en la metodología de enseñanza propuesta es la evaluación, en este trabajo también se describe el sistema de calificación automática usando el plugin Virtual Programming Lab del sistema Moodle. En la práctica, el dessarrollo de laboratorios propuestos ha sido evaluado muy positivamente por parte de los estudiantes en términos de la percepción de la organización y estructura de todo el curso, la alta motivación generada en términos del reto que se impone el estudiante a sí mismo, la imparcialidad y completitud de la calificación en términos de que se evalúan todos y cada uno de los aspectos de cada proyecto propuesto.

Referencias

[1] J. C. Rodriguez-del-Pino, E. Rubio-Royo, Z. J. Hernández-Figueroa, "A Virtual Programming Lab for Moodle with Automatic Assessment and Anti-Plagiarism Features," Proceedings of the 2012 International Conference on e-Learning, e-Bussines, Enterprise Information Systems & e-Government 2012.

[2] J. D. Velasquez, "bcc: A suite of Tools for Introducing Compiler Construction Techniques in the Classroom," Unpublished.

[3] S. Debray, "Making Compiler Design Relevant for Students Who Will (Most Likely) Never Design a Compiler," ACM SIGCE Bulletin, vol. 34, no. 1, 341-345, 2002.

[4] D. Baldwin, "A compiler for teaching about compilers," ACM SIGCE Bulletin, vol. 35, no. 1, 220-223, 2003.

[5] A. Aiken. "Cool: A Portable Project for Teaching Compiler Construction," ACM SIGPLAN Notices, vol. 31, no. 7, pp. 19-24, 1996.

[6] M.L. Corliss and L. E. Christopher, "Bantam: A Customizable, Java-based, Classroom Compiler," ACM SIGCE Bulletin, vol. 40, no. 1, 38-42, 2008.

[7] A. I. Holub, *Compiler Design in C*. Prentice-Hall International, Inc. 1990.

[8] D. A. Watt and D. F. Brown, *Programming Language Processors in JAVA: Compilers and Interpreters*. Prentice Hall, 2000.

[9] A. J. D. Reis, *Compiler Construction Using Java, JavaCC, and Yacc*. Wiley-IEEE Computer Society, 2011.

[10] A. Appel, *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[11] T. Parr. *Language Implementation Patterns*. The Pragmatic Programmers, 2010.

[12] C.M. Frenz, *Pro Perl Parsing*, Apress, 2005.

[13] D. B. Cooperland. *Build awesome command-line applications in Ruby*. The Pragmatic Bookshelf, 2012.

[14] D. Thiebaut, "Automatic evaluation of computer programs using Moodle's virtual programming lab (VPL) plug-in," Journal of Computing Sciences in Colleges, vol. 30, no. 6, pp. 145-151, 2015.

Juan D. Velásquez (M'14–SM'14). He received the Bs. degree in civil engineering in 1994, the MS degree in Systems Engineering in 1997, and the PhD degree in Energy Systems in 2009, all of them from the Universidad Nacional de Colombia, Medellin, Colombia. During 1994–1999, he worked for electricity utilities and consulting companies within the power sector. He joined the Universidad Nacional de Colombia, Medellín, Colombia, in 2000 and became a Full Professor of Computer Science in 2012. During 2004–2006, he was an Associate Dean (Research) and during 2009-2018 was the head of the Computing and Decision Science Department of Facultad de Minas, Universidad Nacional de Colombia, Medellín, Colombia. His current research interests and publications are in the areas of simulation, modeling, optimization, and forecasting in energy markets; nonlinear time-series analysis and forecasting using statistical and computational intelligence techniques; numerical optimization using metaheuristics; analytics and data science. Dr. Velásquez is member of the International Institute of Forecasters, USA.