bcc: A suite of Tools for Introducing Compiler Construction Techniques in the Classroom

J. Velásquez, *Senior Member, IEEE*

***Abstract*—Teaching compiler construction principles in one-semester introductory courses is a very important and complex topic in the computer sciences curriculum. Most of the books are devoted to developing toy, mini or classroom language, but it is almost impossible to cover all material in one-semester course. In this paper, the bcc mini-language and its suite of tools are described. They are completely implemented in Python by hand as command line applications. The bcc language is composed of independent programs for executing different phases of the compiler. The suite of applications is the cornerstone of a compiler construction course, and they allow us to explain and exemplify theoretical and practical aspects, and to develop the capstone project of the course.**

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

I. INTRODUCCIÓN

ESTE artículo describe el lenguaje `bcc` y la suite de aplicaciones que lo implementan; la suite de herramientas es usada en el aula para exponer conceptos teóricos y ejemplificar la implementación práctica de los programas que ejecutan las diferentes fases del proceso de compilación. Esto permite dar a los estudiantes una visión global de las técnicas y herramientas sin entrar en detalles demasiado específicos que pueden oscurecer su aplicabilidad práctica. Durante el curso, la operación de un compilador es ejemplificada con una versión preliminar del compilador de `bcc`, la cual solo implementa una parte de la funcionalidad final. Durante el curso, y a medida que se explican los diferentes temas, los estudiantes deben agregar la funcionalidad faltante. En el desarrollo de la suite de herramientas, se recoge la experiencia de la enseñanza de esta asignatura durante más de diez años y pretenden que el proyecto pueda ser ejecutado de forma individual por cada estudiante.

La enseñanza y el aprendizaje de los principios y las técnicas fundamentales para la implementación de compiladores —y demás herramientas que se basan en estos mismos conceptos y metodologías— son un desafío tanto para docentes como para estudiantes [1]. Su importancia es tal que es uno de los temas más abordados en los currículos de programas de pregrado en ciencias de la computación [2]. Dichas técnicas y herramientas son la base para el desarrollo de muchos lenguajes de programación clásicos y modernos tales como C, C++, Java, Haskell, Python, Perl, R o Ruby; pero también, para el desarrollo de herramientas experimentales; por ejemplo, Meyerovich y otros [3] desarrollaron el lenguaje Flapjax para la creación de aplicaciones Web, el cual está caracterizado por tener un flujo de eventos y reaccionar de forma reactiva ante ellos. LISA es un sistema interactivo para el desarrollo de lenguajes de programación, el cual está compuesto por un conjunto de herramientas que permiten obtener el front-end (editor y herramientas de análisis) y el back-end (intérprete) a partir de la especificación formal del lenguaje [4]. También los mini-lenguajes han sido utilizados extensivamente para la enseñanza de la programación y el pensamiento algorítmico [5]. Las técnicas y herramientas para la construcción de compiladores también son la base conceptual para el desarrollo de aplicaciones y herramientas especializadas como lectores de archivos de configuración, intérpretes dirigidos por sintaxis, intérpretes basados en árboles sintácticos, traductores y generadores de código, herramientas para limpieza de datos, entre otros [6]. Usualmente, la enseñanza de estos principios y técnicas ha sido a través de la teoría y del desarrollo de mini-lenguajes como proyectos de programación, pero realmente poco se ha publicado sobre este aspecto. Entre las pocas iniciativas reportadas en lo referente al aula se encuentra el lenguaje Bantam Java [7], que es un lenguaje de alto nivel que incorpora un subconjunto de las características de Java y que está diseñado específicamente como un proyecto de programación para un curso de teoría de la computación; Bantam Java incorpora varias herramientas que permiten el diseño de tareas prácticas de programación. Aitken [8] desarrolló el lenguaje orientado a objetos para el aula llamado COOL, el cual ha venido siendo utilizado regularmente en la docencia por su autor desde 1996.

En este mismo sentido, la mayor parte de los proyectos de construcción de compiladores e intérpretes en el aula aparecen propuestos en los libros de texto sobre esta temática; sin embargo, su aplicabilidad se ve afectada por la dificultad conceptual y la cantidad de temas abordados en los cursos básicos de teoría de la compilación; a esto debe sumarse, que en muchos casos se requiere un conocimiento específico y detallado del lenguaje de implementación, que suele ser C [9], Java [10] [11] o ML [12]; esto hace que muchos de los conceptos teóricos presentados sean oscurecidos por la complejidad y particularidades del código que los implementa.

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

Por ejemplo, el libro clásico de Holub [9] discute el diseño de compiladores en más de 900 páginas de extensión y presenta una implementación detallada en lenguaje C de más de 5.000 líneas de código.

En este sentido, resulta fundamental el desarrollo de un mini-lenguaje y una máquina virtual que puedan ser fácilmente implementados a mano; esto permite que cada estudiante trabaje de forma individual sobre su proyecto, obligándolo a poner en práctica los conceptos teóricos abordados y a entender todas y cada una de las líneas de código de su implementación. Esto permite una mayor apropiación por parte del estudiante de los elementos teóricos y prácticos abordados en el curso.

El objetivo de este artículo es describir el mini-lenguaje `bcc` y las herramientas que lo implementan. Cada herramienta que realiza una de las fases de la compilación está diseñada como una aplicación de línea de comandos [13]. De esta forma, es posible concentrarse en el desarrollo y depuración de cada una de las aplicaciones, simplificando tanto el trabajo para el estudiante como para el docente.

El resto de este artículo está organizado como sigue. La Sección II presenta la arquitectura general de la suite. La Sección III describe las aplicaciones que conforman el compilador incluyendo la máquina virtual. El proyecto de referencia para los estudiantes es discutido en la Sección IV. Los resultados en el aula son discutidos en la Sección V. Finalmente, se concluye en la Sección VI.

II. El Lenguaje bcc

A. Descripción y Ejemplos

`bcc` es un mini-lenguaje imperativo fuertemente tipado. Permite datos escalares de tipo numérico (`num`) o booleano (`bool`); esto permite el poder ejemplificar diferentes aspectos de la construcción de tablas de símbolos y chequeo de tipos. Su sintaxis es similar a la implementada en la herramienta `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. Adicionalmente, es posible introducir pequeños cambios semestre tras semestre, de tal manera que se dificulte enormemente que los estudiantes comentan fraude presentando códigos desarrollados por otros estudiantes en semestres anteriores.

Algunos elementos sintácticos han sido agregados con el solo fin de enfatizar que la gramática final depende completamente del diseñador. La Fig. 1 contiene un ejemplo trivial en que se imprimen los números del 1 al 10 usando un ciclo **`while`**. Se puede observar que, a diferencia de otros lenguajes, `bcc` requiere que el tipo de la variable se declare después del identificador; al igual que en el lenguaje C, las variables deben ser declaradas y tipadas al inicio de las funciones y del programa principal.

En la Fig. 2 se presenta un programa con dos funciones de usuario y su llamada desde el programa principal (ubicado en la parte final de código).

```
var z:num;
z := 0;
while (z < 10)
  {
  z := z + 1;
  print z;
  }
end
# salida: 1 2 3 4 5 6 7 8 9 10
```

Fig. 1. Ejemplo de un ciclo `while`.

Note que esta estructura es similar a la utilizada en los lenguajes Basic y Fortran, donde las funciones deben ser definidas antes de ser llamadas y deben ubicarse antes del programa principal. `bcc` utiliza un modelo de memoria completamente estático tal como en los lenguajes Basic y Fortran; en este modelo, las funciones no pueden ser llamadas de forma recursiva (ya sea en forma directa o indirecta). Otra decisión de diseño corresponde a que una función no puede ser llamada si ella no ha sido definida previamente, tal como ocurre en los lenguajes interpretados. Estos lineamientos permiten simplificar enormemente el diseño de la máquina virtual que se describe posteriormente.

```
## función min(x, y)
function @min:num (x:num, y:num)
   {
   when ((x < y) == true) do return x;
   return y;
   }

## función max(x, y)
function @max:num (x:num, y:num)
   {
   if ((x < y) == false) do
      {
      return x;
      }
   else
      {
      return y;
      }
   }
print @min(1,2);
print @max(1,2);
end
```

Fig. 2. Ejemplo de la definición de funciones de usuario en el lenguaje bcc.

B. Gramática del Lenguaje

En la Fig. 3 se presenta la gramática del lenguaje `bcc`. En la definición de la gramática se usa intensivamente la misma notación usada en las expresiones regulares para especificar elementos gramaticales opcionales. Esto permite simplificar el diseño de la gramática (menos reglas) y facilita su posterior implementación. Es así como los paréntesis y los corchetes indican elementos opcionales, mientras que el asterisco (*) señala que dicho elemento opcional se repite cero, una o más veces.

El lenguaje `bcc` usa un analizador sintáctico recursivo descendente que requiere analizar a lo sumo dos tokens adelante. Su gramática es concreta (no ambigua) en el sentido de que el token actual permite definir directamente con cual regla gramatical se debe continuar realizando el análisis sintáctico.

```
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 ‘:=’ lexp ‘;’
   | 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. 3. Gramática del lenguaje bcc.

Nótese que la gramática es bastante simple, por lo que puede ser codificada fácilmente a mano.

Los nombres válidos de las variables están conformados por una letra seguida de cero o más letras o números. Los nombres válidos de funciones están conformados por el símbolo @ seguido de una letra seguida de cero o más letras o números.

Algunas anotaciones sobre la gramática son las siguientes. El condicional **UNLESS** es equivalente a WHEN NOT. El comando **UNTIL** es equivalente a **WHILE NOT**. El comando **LOOP** realiza un ciclo infinito. **REPEAT** ejecuta el bloque de código NUM veces. El comando

```
for(expr1; expr2; expr3) do block_stmt
```

es equivalente a:

```
expr1
while (expr2) do
    {
    stmt_block
    expr3
    }
```

Los comandos **NEXT** y **BREAK** modifican la ejecución de los comandos de iteración; **BREAK** fuerza la terminación del ciclo, mientras que **NEXT** es equivalente a un salto al inicio del ciclo.

Las declaraciones ID <op>= lexpr equivalen a ID := ID <op> ID. Las expresiones ID++, ++ID, ID-- y --ID son equivalentes a sus homologas en el leguaje C.

Sólo es posible construir expresiones con múltiples **AND** o múltiples **OR**; no es posible construir expresiones que mezclen directamente **AND** y **OR**, tal como es indicado en la sintaxis.

III. HERRAMIENTAS

A continuación, se describe la organización de la suite y las herramientas que la conforman. La estructura general de la suite de herramientas es presentada en la Fig. 4. La versión para los estudiantes y las presentaciones de clase están disponibles de forma gratuita en https://github.com/jdvelasq/bcc; estas tienen cambios menores semestre a semestre en la medida de que se mejora el curso y el sistema de evaluación de los estudiantes.

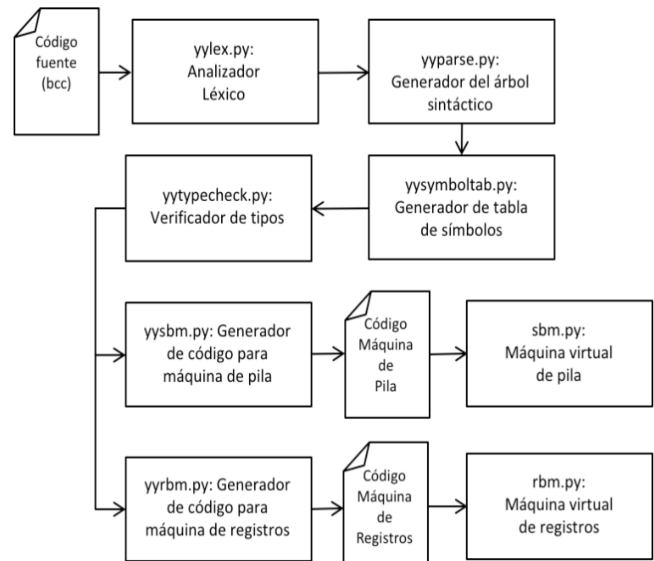

Fig. 4. Arquitectura de la suite de herramientas.

Toda la implementación es realizada en Python 3.5. El proyecto total es de aproximadamente 3500 líneas de código. El proceso se inicia con la lectura del código fuente escrito en bcc para luego realizar secuencialmente el análisis léxico,

sintáctico, semántico (construcción de la tabla de símbolos y verificación de tipos), generación de código de tres direcciones, optimización y ejecución del código assembler usando la aplicación **rbm**.

La llamada de cada una de las aplicaciones desde la línea de comandos del sistema operativo (excepto `sbm.py` y `rbm.py`) usa la siguiente sintaxis:

```
$ python app filename
```

donde `app` es la aplicación. *filename* es el nombre del archivo fuente. Toda la información generada durante el proceso de compilación es almacenada en un árbol cuya sintaxis es similar al lenguaje XML. Su implementación es realizada en el archivo `dataTree.py`.

A. yylex.py: Analizador Léxico

El analizador léxico es implementado en el archivo `yylex.py`; este realiza el análisis léxico del código fuente y almacena los tokens y lexemas reconocidos en un árbol donde cada nodo corresponde a un token; la demás información recolectada es almacenada como propiedades del correspondiente nodo.

El árbol es almacenado en el disco duro en el archivo binario `filename&`. Este archivo es leído por el resto de aplicaciones que requieren acceso a los resultados de las fases previas de análisis. En la Fig. 5 se presenta un ejemplo de la salida en que se presentan los resultados del análisis léxico.

```
$ python yylex.py example.txt
[000] +-- TOKENTABLE
[001]     +-- VAR {lexeme: var, lineno: 0}
[002]     +-- ID {lexeme: z, lineno: 0}
[003]     +-- : {lexeme: :, lineno: 0}
[004]     +-- DATATYPE {lexeme: num, lineno: 0}
[005]     +-- ; {lexeme: ;, lineno: 0}
[006]     +-- ID {lexeme: z, lineno: 1}
[007]     +-- := {lexeme: :=, lineno: 1}
[008]     +-- NUM {lexeme: 0, lineno: 1}
[009]     +-- ; {lexeme: ;, lineno: 1}
[010]     +-- WHILE {lexeme: while, lineno: 2}
[011]     +-- ( {lexeme: (, lineno: 2}
[012]     +-- ID {lexeme: z, lineno: 2}
[013]     +-- < {lexeme: <, lineno: 2}
[014]     +-- NUM {lexeme: 10, lineno: 2}
[015]     +-- ) {lexeme: ), lineno: 2}
[016]     +-- DO {lexeme: do, lineno: 2}
(continua ...)
```

Fig. 5. Tabla de tokens y lexemas para el programa de la Fig. 1.

B. yyparse.py: Analizador Sintáctico.

Esta aplicación genera el árbol sintáctico a partir de los tokens reconocidos por `yylex.py`. El código escrito en `yyparse.py` implementa un analizador recursivo descendente tipo LL(2). La estructura de datos que contiene el árbol sintáctico es adicionada a la generada por el analizador léxico. En la Fig. 6 se ilustra la salida por pantalla que es generada por esta aplicación.

C. yysymboltab.py: Generador de la tabla de símbolos.

Esta aplicación lee de forma automática la tabla de tokens y el árbol sintáctico. Durante el proceso de construcción de la tabla de símbolos, algunas propiedades del árbol sintáctico son actualizadas; como resultado, el archivo binario también es reescrito. En la Fig. 7 se presenta un ejemplo de la salida producida para la tabla de símbolos.

```
$ python yyparse.py example.txt
[000] +-- SYNTAXTREE
[001]     +-- MAINPROG {lineno: 0}
[002]         +-- VAR {lineno: 0}
[003]         |   +-- ID {datatype: num, lineno: 0, scope: 0, value: z}
[003]         +-- ASSIGN {lineno: 1, scope: 0, value: z}
[004]         |   +-- NUM {datatype: num, lineno: 1, value: 0}
[004]         +-- WHILE {lineno: 2}
[005]         |   +-- < {datatype: bool, lineno: 2}
[006]         |   |   +-- ID {lineno: 2, scope: 0, value: z}
[007]         |   |   +-- NUM {datatype: num, lineno: 2, value: 10}
[006]         |   +-- BLOCK {lineno: 4}
[007]         |       +-- ASSIGN {lineno: 4, scope: 0, value: z}
[008]         |       |   +-- + {datatype: num, lineno: 4}
[009]         |       |       +-- ID {lineno: 4, scope: 0, value: z}
[010]         |       |       +-- NUM {datatype:num,lineno: 4, value: 1}
[008]         |       +-- WRITE {lineno: 5}
[009]         |           +-- ID {lineno: 5, scope: 0, value: z}
[005]         +-- END {lineno: 7}
```

Fig. 6. Árbol sintáctico generado por bcc para el programa de la Fig. 1.

```
$ python yysymboltab.py example.txt
[000] +- SYMBOLTABLE
[001]    +-- z {datatype:num, kind:uvar, location:0, scope:0}
```

Fig. 7. Tabla de símbolos para el programa de la Fig. 1.

D. yytypecheck.py: Verificador de Tipos

Esta aplicación realiza la inferencia de tipos a medida que recorre el árbol sintáctico. Las reglas para la creación de la tabla se símbolos y para la inferencia y verificación de tipos son descritas a continuación. Solo existen los tipos `num` y `bool`. Los operadores aritméticos (`+`, `-`, `*`, `/`, `%`) son del tipo `num` × `num` → `num`. Los operadores relacionales `<`, `<=`, `>` y `>=` son del tipo `num` × `num` → `bool`. Los operador relacionales `==` y `!=` soportan `bool` × `bool` → `bool` y `num` × `num` → `bool`. Las variables y las funciones son fuertemente tipadas; su tipo es asignado al momento de su declaración.

Se pueden crear funciones de usuario. Las funciones pueden devolver un número (**num**) o un booleano (**bool**). Todas las variables son locales a las funciones en que son definidas. Las funciones no pueden ser llamadas recursivamente ni de forma directa ni indirecta. Sus argumentos pueden ser del tipo **num** o **bool**. Una función debe ser creada antes de que pueda ser llamada por otra función o por el programa principal.

Los condicionales en las declaraciones **when**, **if**, **unless**, **while**, **until**, **do..while** y **do..until** solo admiten datos tipo **bool**. La expresión **return** permite retornar datos de tipo **num** o **bool**.

Como resultado de este proceso, se modifican las propiedades de los nodos del árbol sintáctico. El árbol sintáctico modificado para el ejemplo de la Fig. 1 es presentado en la Fig. 8.

E. yyrbm.py: Generador de Código Ensamblador para una Máquina Virtual Basada en Registros (código de tres direcciones)

Esta es una aplicación que recorre el árbol sintáctico (tal como el presentado en la Fig. 8) y genera código de tres direcciones para una máquina virtual basada en registros. Las instrucciones son almacenadas como texto en el archivo `filename&.txt`.

En la Fig. 9 se presenta el código de tres direcciones para el programa de la Fig. 1. El código generado en esta fase puede ser ejecutado directamente por el programa `rbm.py`.

```
$ python yytypecheck.py example.txt
[000] +-- SYNTAXTREE
[001]     +-- MAINPROG {lineno: 0}
[002]         +-- VAR {lineno: 0}
[003]         |   +-- ID {datatype: num, lineno: 0, scope: 0, value: z}
[003]         +-- ASSIGN {lineno: 1, scope: 0, value: z}
[004]         |   +-- NUM {datatype: num, lineno: 1, value: 0}
[004]         +-- WHILE {lineno: 2}
[005]         |   +-- < {datatype: bool, lineno: 2}
[006]         |   |   +-- ID {datatype: num, lineno:2,scope:0, value: z}
[007]         |   |   +-- NUM {datatype: num, lineno: 2, value: 10}
[006]         |   +-- BLOCK {lineno: 4}
[007]         |       +-- ASSIGN {lineno: 4, scope: 0, value: z}
[008]         |       |   +-- + {datatype: num, lineno: 4}
[009]         |       |       +-- ID {datatype:num, lineno:4,scope:0, value: z}
[010]         |       |       +-- NUM {datatype:num, lineno:4, value:1}
[008]         |       +-- WRITE {lineno: 5}
[009]         |           +-- ID {datatype: num, lineno: 5, scope: 0, value: z}
[005]         +-- END {lineno: 7}
```

Fig. 8. Árbol sintáctico con propiedades actualizadas después de realizar el chequeo de tipos para el programa de la Fig. 1.

```
% python yyrbm.py example.txt
% cat example.txt&.txt
%%
          ADD+    1   0   0
          STO     1   0
LBL 255
          RCL     2   0
          ADD+    3   0  10
          <       1   2   3
          IFZ     1 254
          RCL     2   0
          ADD+    3   0   1
          +       1   2   3
          STO     1   0
          RCL     1   0
          PRN     1
          GTO    255
LBL 254
          HLT
```

Fig. 9. Código de tres direcciones para una máquina virtual basada en registros.

F. yysbm.py: Generador de Código Ensamblador para una Máquina Virtual Basada en Pila (Código de Cero Direcciones)

Esta función tiene la misma funcionalidad de `yyrbm.py`, y genera código para una máquina virtual que usa una pila y puede ser ejecutado directamente por `sbm.py`.

G. rbm.py: Máquina Virtual Basada en Registros

Este es un simulador de una máquina de registros que ejecuta código de tres direcciones. Tiene dos formas de ejecución: en la primera, recibe el nombre un archivo de código assembler de tres direcciones como parámetro; lee el correspondiente código y luego lo ejecuta.

El simulador contiene internamente los registros descritos en la Tabla I. La máquina virtual usa un modelo de memoria completamente estático en concordancia con el diseño del lenguaje `bcc`, de tal forma que todas las variables son almacenadas en el segmento `DS`. Esto implica que, durante la construcción de la tabla de símbolos, las variables de usuario son asignadas a posiciones fijas de memoria en el segmento `DS`. Las instrucciones de tres direcciones de esta máquina virtual aparecen descritas en la Tabla II. Para realizar la optimización del código el conjunto de instrucciones es ampliado de la siguiente forma (por el optimizador sin ampliar el conjunto de instrucciones de la máquina virtual):

- `add   i j k` equivale a `DR[i]:=DR[j]+DR[k]`
- `addi  i j k` equivale a `DR[i]:=DR[j] + k`
- `addii i j k` equivale a `DR[i]:= j + k`

y así sucesivamente; de esta forma es posible aplicar optimizaciones *peephole* y reducir la cantidad de instrucciones del código final.

TABLA I
SEGMENTOS DE MEMORIA Y REGISTROS DE LA MÁQUINA VIRTUAL BASADA EN REGISTROS

NOMBRE	CONTENIDO
`CS`	Segmento de código (code segment)
`DS`	Segmento de datos (data segment)
`SS`	Segmento de la pila (stack segment)
`HTL`	Registro de parada (halt)
`IP`	Segmento a la instrucción a ejecutar (intruction pointer)
`DR`	Registros de datos del procesador (data registers)
`JT`	Tabla de direcciones de salto

H. sbm.py: Máquina Virtual Basada en Pila

Este es un simulador de una máquina virtual basada en una pila para la ejecución de cálculos. El código que la implementa permite ejemplificar el comportamiento de este tipo de máquinas virtuales.

IV. COMPILADOR DE REFERENCIA Y EVALUACIÓN

Para las presentaciones magistrales y los ejemplos en el aula se entrega un compilador de referencia completamente operativo que contiene solo una parte de las reglas sintácticas definidas en la gramática de la Fig. 3. Por ejemplo, solo se implementan los operadores matemáticos `+` y `*`, y los estudiantes deben implementar `-`, `/` y `%`. De igual forma, solo se implementan las instrucciones **`when`** y **`while`**, y los estudiantes deben implementar el resto.

La forma en que están organizadas las distintas aplicaciones de la suite permite que el estudiante se concentre de forma secuencial en cada módulo ya que no existen interdependencias. Esto también permite que la evaluación pueda realizarse por módulos. Esto resulta especialmente útil cuando se considera que la evaluación puede realizarse en forma automática.

En el caso particular de `bcc`, dicha evaluación es realizada usando el plugin Virtual Programming Lab de Moodle bajo el siguiente esquema: para cada tarea se sube al sistema una versión completa de la respectiva herramienta y la versión del estudiante; se ejecuta la versión completa y se carga en memoria la estructura de datos correspondiente (tabla de tokens y lexemas, árbol sintáctico, etc.); se ejecuta la versión del estudiante y se carga la estructura de datos resultante. La evaluación se realiza mediante la comparación de las dos estructuras de datos; es decir, ya que toda la información generada durante las diferentes fases de la compilación es almacenada en un árbol, es posible comparar nodo a nodo los arboles correspondientes a la solución del docente y la solución entregada por el estudiante.

Adicionalmente, es posible generar proyectos opcionales. Por ejemplo, se puede desarrollar un optimizador de código de tres direcciones con el fin de ejemplificar técnicas básicas de optimización de código. La implementación debería leer el archivo de texto generado durante las fases previas, optimizar

el código, y finalmente, reescribe el archivo con el código optimizado. Para este caso hipotético, la optimización se realizaría a nivel local para los bloques básicos en que se divide el programa original.

TABLA II
INSTRUCCIONES EN LENGUAJE ENSAMBLADOR PARA LA MÁQUINA VIRTUAL BASADA EN REGISTROS

SINTAXIS	INTERPRETACIÓN
	Terminación
HLT	
	Entrada y Salida
PRN r	Imprime el contenido del registro r como valor numérico
PRB r	Imprime el contenido del registro r como valor booleano
	Instrucciones de salto y subrutinas
LBL r	Crea la etiqueta r.
GTO r	IP := JT[r]
IFZ r s	if (DR[r] == 0) IP := JT[s]
IFNZ r s	if (DR[r] != 0) IP := JT[s]
GSB r	Salta a la subrutina con etiqueta r
RET	Retorno desde la subrutina actual.
	Registros y Memoria
STO r s	DS[r] := DR[s]
RCL r s	DR[r] := DS[s]
	Operadores matemáticos
+ r s t	DR[r] := DR[s] + DR[t]
- r s t	DR[r] := DR[s] - DR[t]
* r s t	DR[r] := DR[s] * DR[t]
/ r s t	DR[r] := DR[s] / DR[t]
% r s t	DR[r] := DR[s] % DR[t]
< r s t	DR[r] := DR[s] < DR[t]
<= r s t	DR[r] := DR[s] <= DR[t]
> r s t	DR[r] := DR[s] > DR[t]
>= r s t	DR[r] := DR[s] >= DR[t]
== r s t	DR[r] := DR[s] == DR[t]
!= r s t	DR[r] := DR[s] != DR[t]
ADD+ r s t	DR[r] := DR[s] + t

V. RESULTADOS EN EL AULA

El curso de Teoría de la Computación y la Compilación ha sido dictado con esta metodología durante 6 semestres académicos; anteriormente fue dictado por el mismo docente usando un enfoque más tradicional. El curso en ambas versiones fue evaluado usando el mismo instrumento de medición. En la Tabla III se resumen los resultados en términos del estudiante.

TABLA III
EVALUACIÓN ESTUDIANTIL

ITEM	METODOLOGÍA PROPUESTA	METODOLOGÍA TRADICIONAL
Preparación adecuada de cada sesión	100%	70%
Alta motivación por el tema	88%	69%
Imparcialidad en la calificación	95%	85%
Conexión con otras asignaturas	96%	89%

VI. CONCLUSIONES

En este trabajo se presentó la especificación de un mini-lenguaje para el aula y la descripción de las herramientas que lo conforman. Es suficientemente simple, para que pueda ser, inclusive, implementado a mano por un solo estudiante durante un proyecto de duración semestral; pero es lo suficientemente complejo como para permitir explicar muchos conceptos fundamentales de la construcción de compiladores. El lenguaje es fácilmente modificable, de tal forma que es posible introducir variaciones semestre tras semestre que desalienten los intentos de fraude. La estructura de diseño del lenguaje permite su fácil evaluación a través de herramientas automáticas para la calificación de tareas de programación.

REFERENCIAS

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

[2] 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.

[3] L. A. Meyerovich, A. Guha, J. Baskin, G.H. Cooper, M. Greenberg, A. Bromfield, S. Krishnamurthi. “Flapjax: a programming language for Ajax Applications,” ACM SIGPLAN Notices –OOPSLA ’09, vol. 44, no. 10, pp. 1-20, 2009.

[4] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, Viljem Žumer. “LISA: An Interactive Environment for Programming Language Development”, Volume 2304 of the series Lecture Notes in Computer Science pp 1-4, March 2002.

[5] P. Brusilovsky, E. Calabrese, J. Hvorecky, A. Kouchnirenko and P. Miller. “Mini-languages: a way to learn programming principles,” Education and Information Technologies, vol. 2, no. 1, pp. 65-83, 1997.

[6] T. Parr, “Language Implementation Patterns,”, The Pragmatic Programmers, 2010.

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

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

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

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

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

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

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

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.