Implementación de un Compilador Didáctico para un súper-conjunto de PL/0

Eduardo NAVAS

***Resumen*—En este artículo se describen las características de un compilador para un lenguaje súper-conjunto del conocido PL/0 creado por Niklaus Wirth. La característica principal es que implementa las fases de compilación de tal manera que la información que pasa entre cada una se refleja como un archivo XML.**

***Index Terms*—Compilación, XML, Fases de compilación, PL/0, árbol de sintaxis.**

I. Introducción

AL estudiar diseño de compiladores, siempre se habla de las fases tradicionales de Análisis Léxico, Análisis Sintáctico, Análisis Semántico y Generación de Código Objeto (ver [1], [2], [3], [4], [5], [6]). Conceptualmente estas fases son fáciles de diferenciar, sin embargo los programadores normalmente no piensan en ellas. El comportamiento por defecto de los compiladores tradicionales es ocultarlas para ofrecerle a los programadores una respuesta rápida y efectiva. Esto es razonable cuando lo que se quiere es un archivo ejecutable a partir de un conjunto de archivos fuente.

Eventualmente, al estudiar sobre diseño y construcción de compiladores, a los programadores les gustaría ver el proceso efectuado por las fases de compilación de los compiladores que se usan tradicionalmente, sin embargo, los compiladores tradicionales no permiten mostrar ese tipo de información.

Así, este artículo presenta un compilador para un súper-conjunto de PL/0, denominado aquí pl0+, que expone los datos que se transmiten entre cada fase de compilación. También se demuestra que es posible implementar un compilador en un lenguaje de alto nivel.[10]

II. Metodología

Las siguientes secciones describen brevemente el concepto, el diseño y las fases del compilador denominado `tradukilo.py`.

II-A. Diseño

El compilador ha sido diseñado pensando no en su velocidad de ejecución, sino en la posibilidad de usarlo para popósitos académicos-pedagógicos.

La idea es que el programador pueda elegir las fases de compilación a ejecutar (algunas combinaciones por supuesto no son posibles). Por ejemplo, si se desea estudiar la fase de análisis léxico, podría indicarse al compilador que se detenga al terminar dicha fase. La salida del compilador será entonces, un archivo en formato xml que contiene la secuencia lineal de tokens que componen el código fuente.

Por otro lado, si ya se tiene un archivo xml que contiene la especicación del árbol de sintaxis (que es la salida de la fase de análisis sintáctico), podría indicársele al compilador que sólo ejecute la fase de análisis semántico. La salida entonces será otro archivo xml que contenga el árbol de sintaxis con las validaciones de coincidencia de tipos y las otras tareas que realiza esta fase.

El funcionamiento por defecto del compilador deberá ser el de ejecutar todas las fases de compilación, pero sin escribir en memoria secundaria los flujos de comunicación de las diferentes fases.

II-B. Lenguaje fuente

El lenguaje fuente es un súper-conjunto del conocido lenguaje PL/0 creado por *Niklaus Wirth*[7]. Lo conoceremos como lenguaje pl0+ y su gramática se presenta a continuación:

```
<programa> ::= <bloque> '.'

<bloque> ::=
  {<declaración_constantes>}?
  {<declaración_variables>}?
  {<declaración_procedimiento>}*
  <instrucción> {';'}?

<declaración_constantes> ::= 'const'
  <identificador> '=' { '+' | '-' }? <número>
  { ','
    <identificador> '=' { '+' | '-' }? <número>
  }* ';'

<declaración_variables> ::= 'var' <identificador>
  { ',' <identificador> }* ;

<declaración_procedimiento> ::= 'procedure'
  <identificador> ';' <bloque> ';'

<instrucción> ::=
  <identificador> := <expresión> |
  'call' <identificador> |
  'begin' <instrucción> { ';' <instrucción> }*
    { ';' }? end |
  'if' <condición> 'then' <instrucción>
    { 'else' <instrucción> }? |
  'while' <condición> 'do' <instrucción> |
  'read' <identificador> |
  'write' <identificador> |
  <nada>

<condición> ::=
```

Eduardo NAVAS es Catedrático del Departamento de Electrónica e Informática de la Universidad Centroamericana "José Simeón Cañas" (enavas@ing.uca.edu.sv)

```
  'odd' <expresión> |
  <expresión> { '=' | '<>' | '<' | '>' | '<='
    | '>=' } <expresión>

<expresión> ::= { '+' | '-' }? <término>
  { { '+' | '-' } <término> }*

<término> ::= <factor> { { '*' | '/' } <factor>
    }*

<factor>::= { '-' }* { <identificador> | <número>
     | ( <expresión> ) }

<identificador> ::= <letra> { <letra> | <dígito>
  | '_' }*
<número> ::= { <dígito> }+

<letra> ::= { 'a' | ... | 'z' }
<dígito> ::= { '0' | ... | '9' }

<Comentarios> ::= '(*'  <lo-que-sea>* '*)'
```

Es un lenguaje de programación sencillo de alto nivel que permite anidamiento de procedimientos, recursión directa e indirecta, sólo tiene variables y constantes del tipo de dato entero de 32 bits (es decir en el intervalo $\left[-2^{31}, 2^{31}-1\right]$) y tiene los operadores aritméticos básicos y los relacionales para las condiciones. Los procedimientos no retornan ningún valor, es decir que no hay funciones. Y sólamente tiene instrucciones de entrada y salida básica de enteros por la entrada estándar y la salida estándar respectivamente.

A continuación se presenta un programa de ejemplo que calcula los números de la serie de Fibonacci:

Listing 1. fibonacci.pl0+ – Programa que muestra la serie de Fibonacci

```
(*
    Cálculo de los números de la serie de
        fibonacci:
    f_0 = 1
    f_1 = 1
    f_n = f_{n-1} + f_{n-2}, n>1
*)
const fib_0=1, fib_1=1;
var n, f;

procedure fibonacci;
    var i,    (* Variable para hacer el recorrido
        *)
        f_1, (* Número Fibonacci anterior *)
        f_2; (* Número Fibonacci anterior al
            anterior *)
    begin
        if n=0 then f:=fib_0;
        if n=1 then begin
            f:=fib_1;
            write f; (* Los primeros dos
                elementos son iguales *)
        end;
        if n>1 then begin
            f_1:=1;
            write f_1;

            f_2:=1;
            write f_2;

            i:=2;
            while i<n do begin
                f:=f_1+f_2;
                write f;
                f_2:=f_1;
                f_1:=f;
                i:=i+1;
            end;
            f:=f_1+f_2;
        end;
    end;

begin
    read n;
    call fibonacci;
    write f;
end.
```

II-C. Lenguaje objetivo

El lenguaje objetivo es una variante del código p definido para PL/0. Es un lenguaje tipo ensamblador y lo conoceremos en este artículo como lenguaje p+. A continuación se presenta la definición de sus instrucciones y mnemónicos:

`LIT <val>` Pone el literal numérico `<val>` en el tope de la pila.

`CAR <dif> <pos>`
Copia el valor de la variable que está en la posición `<pos>` en el bloque definido a `<dif>` niveles estáticos desde el bloque actual en el tope de la pila.

`ALM <dif> <pos>`
Almacena lo que está en el tope de la pila en la variable que está en la posición `<pos>` en el bloque definido a `<dif>` niveles estáticos desde el bloque actual.

`LLA <dif> <dir>`
Llama a un procedimiento definido a `<dif>` niveles estáticos desde el bloque actual, que comienza en la dirección `<dir>`.

`INS <num>` Instancia un procedimiento, reservando espacio para las `<num>` variables del bloque que lo implementa (este número incluye las celdas necesarias para la ejecución del código, que en el caso del lenguaje p+[1] son 3 enteros adicionales).

`SAC <dir>` Salto condicional hacia la dirección `<dir>` si el valor en el tope de la pila es 0.

`SAL <dir>` Salto incondicional hacia la dirección `<dir>`.

`OPR <opr>` Operación aritmética o relacional, según el número `<opr>`. Los parámetros son los valores que están actualmente en el tope de la pila y allí mismo se coloca el resultado. Los valores posibles para `<opr>` son:

1:	Negativo (menos unario)
2:	Suma (+)
3:	Resta (-)
4:	Multiplicación (*)
5:	División (/)
6:	Operador *odd* (impar)
8:	Comparación de igualdad (=)
9:	Diferente de (<>)
10:	Menor que (<)
11:	Mayor o igual que (>=)

[1] igual que en el caso del código p

12: Mayor que (>)
13: Menor o igual que (<=)

`RET` Retornar de procedimiento.

`LEE` Lee un valor de la entrada estándar y la almacena en el tope de la pila.

`ESC` Escribe en la salida estándar el valor del tope de la pila.

II-D. Interface del compilador

La interface del compilador es por línea de comandos. La sintaxis general para invocarlo es la siguiente:

```
$ python tradukilo.py [-a] [-m] [-e] [--lex] [--
    sin] [--sem] [--gen] programa
```

Las opciones y sus significados son los siguientes:

`-a` (`--ayuda`) Muestra un mensaje de ayuda y termina de inmediato.

`-m` (`--mostrar`) En caso de haber una compilación sin errores, muestra el resultado del proceso en la pantalla.

`-e` (`--errores-xml`) Imprime los errores y las advertencias en la salida estándar de error en formato de archivo xml. Esta opción está destinada a reportar los errores y advertencias de manera estructurada para ser procesados por una eventual interfaz gráfica para el compilador.

`--lex` Indica que se debe ejecutar la fase de análisis léxico.

`--sin` Indica que se debe ejecutar la fase de análisis sintáctico.

`--sem` Indica que se debe ejecutar la fase de análisis semántico.

`--gen` Indica que se debe ejecutar la fase de generación de código objeto.

Al ejecutar el comando, se intenta compilar el archivo `programa`. Si no se indica ninguna fase de compilación particular, se asumen todas. Si la compilación tiene éxito, se genera un archivo con extensión diferente, dependiendo de la última fase ejecutada.

La extensión de los programas pl0+ se asume como `.pl0+`, la extensión de salida del análisis léxico se asume `.pl0+lex`, la extensión de salida del análisis sintáctico se asume `.pl0+sin`, la de análisis semántico `.pl0+sem` y la de la generación de código objeto `.p+`.

Pueden ejecutarse combinaciones como:

```
$ python tradukilo.py programa.pl0+
$ python tradukilo.py --lex programa.pl0+
$ python tradukilo.py --lex --sin programa.pl0+
$ python tradukilo.py programa.pl0+ --lex --sin
    --sem
$ python tradukilo.py programa.pl0+lex --sin
    --sem
$ python tradukilo.py programa.pl0+sin --sem
    --gen
```

De las líneas anteriores, la primera ejecuta todas las fases de compilación sobre el archivo fuente `programa.pl0+` y genera un archivo llamado `programa.p+`. La segunda sólo ejecuta la fase de análisis léxico y genera un archivo llamado `programa.pl0+lex`. La tercera ejecuta las fases de análisis léxico y sintáctico y genera un archivo llamado `programa.pl0+sin`. La cuarta ejecuta las fases de análisis léxico, sintáctico y semántico y genera un archivo llamado `programa.pl0+sem`. La quinta toma un archivo `programa.pl0+lex` con la lista de *tokens* de un programa fuente, ejecuta las fases de análisis sintáctico y semántico y genera un archivo llamado `programa.pl0+sem`. La sexta toma un archivo `programa.pl0+sin` que contiene el árbol de sintaxis de un programa fuente, ejecuta las fases de análisis semántico y de generación de código objeto y genera un archivo llamado `programa.p+`.

Evidentemente no todas las combinaciones son posibles. Por ejemplo, no se puede solicitar ejecutar las fases de análisis léxico (`--lex`) y de análisis semántico (`--sem`). En tales casos, el compilador responderá con un mensaje de error al usuario.

Las extensiones de entrada y salida de cada fase se describen en la siguiente tabla y en la figura 1 en la página siguiente:

Código de la fase	Opción de la fase	Extensión entrada	Extensión salida
lex	`--lex`	`.pl0+`	`.pl0+lex`
sin	`--sin`	`.pl0+lex`	`.pl0+sin`
sem	`--sem`	`.pl0+sin`	`.pl0+sem`
gen	`--gen`	`.pl0+sem`	`.p+`

Cabe recalcar que sólo se crea el archivo de la última fase de compilación ejecutada, y no los intermedios.

II-E. Adición de fases complementarias

El compilador está implementado como un conjunto de programas Python 2.6 organizados de la siguiente manera:

```
tradukilo.py  --> Interface principal del
    compilador

fases/
  __init__.py --> Configuración global
  lex.py      --> Análisis léxico
  sin.py      --> Análisis sintáctico
  sem.py      --> Análisis semántico
  gen.py      --> Generación de código objeto

rulilo.py     --> Intérprete de programas p+
```

En el archivo `__init__.py` contiene una lista llamada `fasesDisponibles` que determina cuáles son las fases que pueden invocarse desde la interfaz principal del compilador. Puede agregarse una fase nueva, incluyendo las opciones de línea de comandos simplemente agregando la información del nombre del módulo, el archivo en el que está, el nombre de la función de traducción en ese módulo, la extensión de salida de esa fase y una descripción para usarse en los mensajes de error.

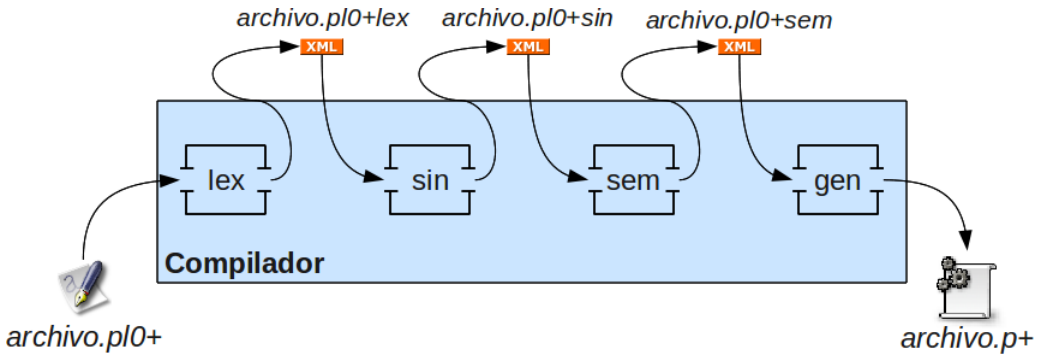

Figura 1. Esquema del compilador

El contenido actual de la lista está incluído en el siguiente fragmento de código:

```
fasesDisponibles = [ {
    'nombre': 'lex',
    'archivo': 'lex.py',
    'función': 'faseTraduccion',
    'extensiónSalida': '.pl0+lex',
    'descripción': 'Fase␣de␣análisis␣léxico',
  }, {
    'nombre': 'sin',
    'archivo': 'sin.py',
    'función': 'faseTraduccion',
    'extensiónSalida': '.pl0+sin',
    'descripción': 'Fase␣de␣análisis␣sintáctico',
  }, {
    'nombre': 'sem',
    'archivo': 'sem.py',
    'función': 'faseTraduccion',
    'extensiónSalida': '.pl0+sem',
    'descripción': 'Fase␣de␣análisis␣semántico',
  }, {
    'nombre': 'gen',
    'archivo': 'gen.py',
    'función': 'faseTraduccion',
    'extensiónSalida': '.p+',
    'descripción': 'Fase␣de␣generación␣de␣código␣
        objeto',
  } ]
```

II-F. Manejo y reporte de Errores

En este compilador los errores se registran en dos listas internas que se van pasando de fase en fase, diferenciando entre los "*errores*" y las "*advertencias*".

Los *errores* son aquellos fragmentos de código que hacen imposible la compilación porque es muy difícil o imposible determinar la intención del programador.

Por otro lado, las *advertencias* son fragmentos de código que permiten la compilación, porque puede suponerse la intención del programador, pero el programa fuente no es totalmente válido y el resultado de la compilación puede no coincidir del todo con la verdadera intención del programador.

Este compilador, así como muchos otros, muestra primero los *errores* y luego las *advertencias* y en cada grupo, los errores están ordenados respecto de su aparición en el código fuente.

A continuación se presenta un programa pl0+ con múltiples errores (programa 2) y luego la salida estándar del compilador al intentar compilarlo (código 3):

Listing 2. errores.pl0+ – Código con errores

```
const n=5;
var f, i;
begin
    f := 2  5 * 9
    i := 2 % f;
    f := 12*2+5*9/8;
    f := 8 / i * 2 -;
    f := 9 - i * 2
    if n<>1 then begin
        i:=2;
        while i 5 <= n 2 do begin
            f1:=f; i:=i+1;
        end
    end;
end.
```

Listing 3. Salida estándar al intentar compilar el programa anterior

```
ERROR***************
Fase de origen:sin
Línea 4: Falta un operador
    f := 2  5 * 9
----------^
ERROR***************
Fase de origen:sin
Línea 5: Falta un operador
    i := 2 % f;
----------^
ERROR***************
Fase de origen:lex
Línea 5: Caracter inválido.
    i := 2 % f;
-----------^
ERROR***************
Fase de origen:sin
Línea 7: Se esperaba 'identificador', 'numero', '
    -' o '(...)', pero se encontró ';'
    f := 8 / i * 2 -;
--------------------^
ERROR***************
Fase de origen:sin
Línea 11: Falta un operador
        while i 5 <= n 2 do begin
---------------^
ERROR***************
Fase de origen:sem
Línea 12: Referencia a variable no declarada
            f1:=f; i:=i+1;
------------^
ADVERTENCIA**********
```

```
Fase de origen:sin
Línea 8: Falta un ';'
    f := 9 - i * 2
------------------^
```

Para permitir que el compilador pueda ser acoplado con un posible editor especial de código fuente pl0+, que muestre los errores léxicos, sintácticos y semánticos de los programas, los *errores* y *advertencias* también se pueden desplegar en forma de archivo xml estructurado a través de la salida estándar de error.

El siguiente archivo representa el mismo reporte de errores para el programa 2 pero habiendo incluído la opción `-e` (o `--errores-xml`) al intentar compilarlo:

Listing 4. Errores del programa errores.pl0+ en formato xml

```
<?xml version="1.0" ?>
<errores_y_advertencias>
  <errores>
    <error columna="10" linea="4">
      <mensaje>
        Falta un operador
      </mensaje>
      <contexto>
<![CDATA[    f := 2  5 * 9]]>      </contexto>
      <fase nombre="sin">
        Fase de análisis sintáctico
      </fase>
    </error>
    <error columna="10" linea="5">
      <mensaje>
        Falta un operador
      </mensaje>
      <contexto>
<![CDATA[    i := 2 % f;]]>      </contexto>
      <fase nombre="sin">
        Fase de análisis sintáctico
      </fase>
    </error>
    <error columna="11" linea="5">
      <mensaje>
        Caracter inválido.
      </mensaje>
      <contexto>
<![CDATA[    i := 2 % f;]]>      </contexto>
      <fase nombre="lex">
        Fase de análisis léxico
      </fase>
    </error>
    <error columna="20" linea="7">
      <mensaje>
        Se esperaba 'identificador', 'numero', '-
            ' o '(...)', pero se encontró ';'
      </mensaje>
      <contexto>
<![CDATA[    f := 8 / i * 2 -;]]>      </contexto
    >
      <fase nombre="sin">
        Fase de análisis sintáctico
      </fase>
    </error>
    <error columna="15" linea="11">
      <mensaje>
        Falta un operador
      </mensaje>
      <contexto>
<![CDATA[        while i 5 <= n 2 do begin]]>
         </contexto>
      <fase nombre="sin">
        Fase de análisis sintáctico
      </fase>
    </error>
    <error columna="12" linea="12">
      <mensaje>
        Referencia a variable no declarada
      </mensaje>
      <contexto>
<![CDATA[            f1:=f; i:=i+1;]]>      </
    contexto>
      <fase nombre="sem">
        Fase de análisis semántico
      </fase>
    </error>
  </errores>
  <advertencias>
    <error columna="18" linea="8">
      <mensaje>
        Falta un ';'
      </mensaje>
      <contexto>
<![CDATA[    f := 9 - i * 2]]>      </contexto>
      <fase nombre="sin">
        Fase de análisis sintáctico
      </fase>
    </error>
  </advertencias>
</errores_y_advertencias>
```

Finalmente, la estrategia general de procesamiento de errores consiste en que cuando se encuentra un error, el compilador hace una serie de suposiciones y "salta" hasta algún otro punto del programa para intentar continuar con su trabajo.

En algunas ocaciones, esos conjuntos de suposiciones y saltos, no son correctos, y el compilador se "desestabiliza", provocando detección de errores falsos.

Para disminuir la frecuencia de las detecciones falsas de errores, se optó por aplicar —al menos en la fase de análisis sintáctico al construir el árbol de operaciones de las expresiones— la regla eurística que indica que lo normal es que sólo haya un error en una misma línea de código.

Esto, en términos de implementación se solucionó con que al momento de "identificar" un error o advertencia, se consulta si ya hay identificado un error o advertencia en la misma línea y de la misma fase. Si lo hay, no se registra el más reciente.

II-G. Interface del intérprete

La interface del intérprete es por línea de comandos. La sintaxis general para invocarlo es la siguiente:

```
$ python rulilo.py [-a] [-d] programa-objeto.p+
```

Las opciones y sus significados son los siguientes:

`-a` (`--ayuda`) Muestra un mensaje de ayuda y termina de inmediato.

`-d` (`--depurar`) Ejecuta cada instrucción haciendo una espera, para que el usuario observe el valor de los registros de la máquina virtual.

III. Conclusiones

1. Disponer de un compilador que permita examinar el resultado de cada fase de compilación, ayuda a comprender el propósito de estas.

2. Al implementar un compilador, la separación de las fases de compilación —es decir, implementándolas como módulos funcionales separados que se transfieren información entre sí de manera secuencial— contribuye a su mejor comprensión, permite una depuración más fácil y permite ampliaciones futuras. Sin embargo, también exige documentar detalladamente las transformaciones que sufre el programa entre ellas.
3. El diseño modular del compilador `tradukilo.py` permite que este sea ampliado adicionando fases de optimización y procesamiento adicional antes y después de las fases principales (ver sección II-E).
4. El formato xml permite el compartimiento genérico de información entre diferentes componentes de software de manera simple y legible por humanos.
5. Un compilador puede ser implementado en un lenguaje de alto nivel (como Python) y no necesariamente implica que el compilador será lento en la escala de tiempo humano.

IV. Discución de las conclusiones y resultados

Debido a la simplicidad y limitaciones del lenguaje pl0+ —no hay funciones, no hay parámetros en los subprocedimientos, sólo hay un tipo de dato, no hay estructuras, no hay clases, las declaraciones sólo pueden ir al principio de los bloques, etc.— las tareas de la fase de análisis semántico de `tradukilo.py` son muy limitadas y escuetas, y sería más interesante ver el análisis semántico necesario para lenguajes más ricos.

Hubo muchos aspectos sobre el tratamiento de errores que quedó fuera de este artículo y que merecen —por rigurosidad académica— ser presentados y analizados. Esto podría presentarse posteriormente en otro artículo.

Luego de la implementación del intérprete de código p+, se hace pantente que convendría hacer algunas reflexiones sobre las características del lenguaje objetivo y sobre la implementación del intérprete correspondiente. Por ejemplo pueden analizarse el lenguaje intermedio PIR y el de bajo nivel PASM del proyecto *Parrot* (ver [8]).

Finalmente, luego de haber implementado un compilador para un lenguaje sencillo de alto nivel (pl0+), y un intérprete para un lenguaje de bajo nivel (p+), surge la idea general de diseñar y construir un conjunto de herramientas de propósito pedagógico, herramientas propias (conocidas desde dentro) que sirvan para varios propósitos en varios ámbitos dentro del quehacer académico de las ciencias de la computación.

Por ejemplo, se puede diseñar un lenguaje de programación de alto nivel orientado a los algoritmos en pseudocódigo[2]; diseñar un lenguaje de programación de bajo nivel, genérico, orientado a código de máquina, en el que se pueda programar directamente (a diferencia del lenguaje p+); diseñar y construir un compilador que traduzca programas escritos del primer lenguaje al segundo y que permita, así como `tradukilo.py`, examinar el resultado de todas sus fases; y obviamente construir un intérprete de programas en el segundo lenguaje.

Un conjunto tal de herramientas tendría múltiples propósitos u objetivos, como los siguientes:

- Introducir los tópicos básicos de la algoritmia y la programación de computadoras en alto nivel, a estudiantes de educación media, estudiantes universitarios, y catedráticos que requieran de un formato genérico y simple para describir algoritmos.
- Introducir los conceptos básicos de programación de computadoras en bajo nivel, a estudiantes de educación media en el área de electrónica y estudiantes universitarios en el área de arquitectura de computadoras, en un entorno de ejecución seguro, sin posibilidades de dañar los equipos físicos y sin consecuencias colaterales por investigar o comenter errores.
- Contar con un compilador propio, hecho en casa, en el tercer mundo, diseñado para ser leído y estudiado, y poder profundizar en el diseño y construcción de ese tipo de herramientas o de piezas de software similares.
- Cumplir el importante propósito de contar con un amplio conjunto de herramientas que puedan ser estudiadas desde diferentes perspectivas a lo largo de una carrera de grado —y tal vez también de postgrado—, como la Licenciatura en Ciencias de la Computación que ofrece la Universidad Centroamericana “José Simeón Cañas”.

Yendo más allá —y tal vez tan sólo pensando en voz alta—, podría extrapolarse el desarrollo de estas herramientas a un sistema operativo completo con propósitos académicos, de investigación y desarrollo, como los presentados en [4] y [9] que tienen sus propios lenguajes de programación.

V. Anexos

V-A. *Descripción de la Fase de Análisis Léxico*

En cualquier compilador el propósito de esta fase es convertir los caracteres del archivo que contiene al programa fuente, en una secuencia lineal de los elementos mínimos con un significado en el lenguaje (y sus eventuales valores). Estos elementos mínimos con significado son llamados *Elementos Léxicos* o *Tokens*.[1], [5], [6]

Para cada elemento léxico de los programas en lenguaje pl0+, la fase de análisis léxico genera una etiqueta xml que lo representa. Cada etiqueta tiene los atributos `columna`, `linea` y `longitud`, que indican respectivamente la columna donde comienza el elemento, la línea en que se encuentra y la longitud del elemento léxico que representa.

Las palabras reservadas del lenguaje pl0+ son: ’`begin`’, ’`call`’, ’`const`’, ’`do`’, ’`end`’, ’`if`’, ’`odd`’, ’`procedure`’, ’`then`’, ’`var`’, ’`while`’, ’`else`’, ’`write`’ y ’`read`’. Cada una de ellas se representa con una etiqueta con el mismo nombre pero en mayúsculas.

Los identificadores se representan con una etiqueta con nombre `IDENTIFICADOR` con el atributo `nombre` que indica el nombre del identificador.

[2]El Departamento de Electrónica e Informática ya ha iniciado un esfuerzo similar y ya se tiene experiencia en esta área.

Los números que aparecen en los programas, se representan como etiquetas con nombre `NUMERO` con el atributo `valor` que contiene el valor del número.

Los demás elementos léxicos de pl0+ se representan según la siguiente tabla:

Símbolo	Etiqueta
=	igual
:=	asignacion
,	coma
;	punto_y_coma
(	parentesis_apertura
)	parentesis_cierre
<>	diferente
<	menor_que
>	mayor_que
<=	menor_igual
>=	mayor_igual
+	mas
-	menos
*	por
/	entre
.	punto

A continuación se presenta el resultado de aplicar el análisis léxico sobre el programa 1 en la página 2:

Listing 5. fibonacci.pl0+lex – Elementos léxicos de fibonacci.pl0+

```
<?xml version="1.0" ?>
<lexemas>
  <!--
**************************************
Por favor, no modifique este archivo
a menos que sepa lo que está haciendo
**************************************
-->
  <CONST columna="0" linea="7" longitud="5"/>
  <IDENTIFICADOR columna="6" linea="7" longitud="
      5" nombre="fib_0"/>
  <igual columna="11" linea="7" longitud="1"/>
  <NUMERO columna="12" linea="7" longitud="1"
      valor="1"/>
  <coma columna="13" linea="7" longitud="1"/>
  <IDENTIFICADOR columna="15" linea="7" longitud=
      "5" nombre="fib_1"/>
  <igual columna="20" linea="7" longitud="1"/>
  <NUMERO columna="21" linea="7" longitud="1"
      valor="1"/>
  <punto_y_coma columna="22" linea="7" longitud="
      1"/>
  <VAR columna="0" linea="8" longitud="3"/>
  <IDENTIFICADOR columna="4" linea="8" longitud="
      1" nombre="n"/>
  <coma columna="5" linea="8" longitud="1"/>
  <IDENTIFICADOR columna="7" linea="8" longitud="
      1" nombre="f"/>
  <punto_y_coma columna="8" linea="8" longitud="1
      "/>
  <PROCEDURE columna="0" linea="10" longitud="9"/
      >
  <IDENTIFICADOR columna="10" linea="10" longitud
      ="9" nombre="fibonacci"/>
  <punto_y_coma columna="19" linea="10" longitud=
      "1"/>
  <VAR columna="4" linea="11" longitud="3"/>
  <IDENTIFICADOR columna="8" linea="11" longitud=
      "1" nombre="i"/>
  <coma columna="9" linea="11" longitud="1"/>
  <IDENTIFICADOR columna="8" linea="12" longitud=
      "3" nombre="f_1"/>
  <coma columna="11" linea="12" longitud="1"/>
  <IDENTIFICADOR columna="8" linea="13" longitud=
      "3" nombre="f_2"/>
  <punto_y_coma columna="11" linea="13" longitud=
      "1"/>
  <BEGIN columna="4" linea="14" longitud="5"/>
  <IF columna="8" linea="15" longitud="2"/>
  <IDENTIFICADOR columna="11" linea="15" longitud
      ="1" nombre="n"/>
  <igual columna="12" linea="15" longitud="1"/>
  <NUMERO columna="13" linea="15" longitud="1"
      valor="0"/>
  <THEN columna="15" linea="15" longitud="4"/>
  <IDENTIFICADOR columna="20" linea="15" longitud
      ="1" nombre="f"/>
  <asignacion columna="21" linea="15" longitud="2
      "/>
  <IDENTIFICADOR columna="23" linea="15" longitud
      ="5" nombre="fib_0"/>
  <punto_y_coma columna="28" linea="15" longitud=
      "1"/>
  <IF columna="8" linea="16" longitud="2"/>
  <IDENTIFICADOR columna="11" linea="16" longitud
      ="1" nombre="n"/>
  <igual columna="12" linea="16" longitud="1"/>
  <NUMERO columna="13" linea="16" longitud="1"
      valor="1"/>
  <THEN columna="15" linea="16" longitud="4"/>
  <BEGIN columna="20" linea="16" longitud="5"/>
  <IDENTIFICADOR columna="12" linea="17" longitud
      ="1" nombre="f"/>
  <asignacion columna="13" linea="17" longitud="2
      "/>
  <IDENTIFICADOR columna="15" linea="17" longitud
      ="5" nombre="fib_1"/>
  <punto_y_coma columna="20" linea="17" longitud=
      "1"/>
  <WRITE columna="12" linea="18" longitud="5"/>
  <IDENTIFICADOR columna="18" linea="18" longitud
      ="1" nombre="f"/>
  <punto_y_coma columna="19" linea="18" longitud=
      "1"/>
  <END columna="8" linea="19" longitud="3"/>
  <punto_y_coma columna="11" linea="19" longitud=
      "1"/>
  <IF columna="8" linea="20" longitud="2"/>
  <IDENTIFICADOR columna="11" linea="20" longitud
      ="1" nombre="n"/>
  <mayor_que columna="12" linea="20" longitud="1"
      />
  <NUMERO columna="13" linea="20" longitud="1"
      valor="1"/>
  <THEN columna="15" linea="20" longitud="4"/>
  <BEGIN columna="20" linea="20" longitud="5"/>
  <IDENTIFICADOR columna="12" linea="21" longitud
      ="3" nombre="f_1"/>
  <asignacion columna="15" linea="21" longitud="2
      "/>
  <NUMERO columna="17" linea="21" longitud="1"
      valor="1"/>
  <punto_y_coma columna="18" linea="21" longitud=
      "1"/>
  <WRITE columna="12" linea="22" longitud="5"/>
  <IDENTIFICADOR columna="18" linea="22" longitud
      ="3" nombre="f_1"/>
  <punto_y_coma columna="21" linea="22" longitud=
      "1"/>
  <IDENTIFICADOR columna="12" linea="24" longitud
      ="3" nombre="f_2"/>
  <asignacion columna="15" linea="24" longitud="2
      "/>
  <NUMERO columna="17" linea="24" longitud="1"
      valor="1"/>
  <punto_y_coma columna="18" linea="24" longitud=
```

```
    "1"/>
  <WRITE columna="12" linea="25" longitud="5"/>
  <IDENTIFICADOR columna="18" linea="25" longitud
    ="3" nombre="f_2"/>
  <punto_y_coma columna="21" linea="25" longitud=
    "1"/>
  <IDENTIFICADOR columna="12" linea="27" longitud
    ="1" nombre="i"/>
  <asignacion columna="13" linea="27" longitud="2
    "/>
  <NUMERO columna="15" linea="27" longitud="1"
    valor="2"/>
  <punto_y_coma columna="16" linea="27" longitud=
    "1"/>
  <WHILE columna="12" linea="28" longitud="5"/>
  <IDENTIFICADOR columna="18" linea="28" longitud
    ="1" nombre="i"/>
  <menor_que columna="19" linea="28" longitud="1"
    />
  <IDENTIFICADOR columna="20" linea="28" longitud
    ="1" nombre="n"/>
  <DO columna="22" linea="28" longitud="2"/>
  <BEGIN columna="25" linea="28" longitud="5"/>
  <IDENTIFICADOR columna="16" linea="29" longitud
    ="1" nombre="f"/>
  <asignacion columna="17" linea="29" longitud="2
    "/>
  <IDENTIFICADOR columna="19" linea="29" longitud
    ="3" nombre="f_1"/>
  <mas columna="22" linea="29" longitud="1"/>
  <IDENTIFICADOR columna="23" linea="29" longitud
    ="3" nombre="f_2"/>
  <punto_y_coma columna="26" linea="29" longitud=
    "1"/>
  <WRITE columna="16" linea="30" longitud="5"/>
  <IDENTIFICADOR columna="22" linea="30" longitud
    ="1" nombre="f"/>
  <punto_y_coma columna="23" linea="30" longitud=
    "1"/>
  <IDENTIFICADOR columna="16" linea="31" longitud
    ="3" nombre="f_2"/>
  <asignacion columna="19" linea="31" longitud="2
    "/>
  <IDENTIFICADOR columna="21" linea="31" longitud
    ="3" nombre="f_1"/>
  <punto_y_coma columna="24" linea="31" longitud=
    "1"/>
  <IDENTIFICADOR columna="16" linea="32" longitud
    ="3" nombre="f_1"/>
  <asignacion columna="19" linea="32" longitud="2
    "/>
  <IDENTIFICADOR columna="21" linea="32" longitud
    ="1" nombre="f"/>
  <punto_y_coma columna="22" linea="32" longitud=
    "1"/>
  <IDENTIFICADOR columna="16" linea="33" longitud
    ="1" nombre="i"/>
  <asignacion columna="17" linea="33" longitud="2
    "/>
  <IDENTIFICADOR columna="19" linea="33" longitud
    ="1" nombre="i"/>
  <mas columna="20" linea="33" longitud="1"/>
  <NUMERO columna="21" linea="33" longitud="1"
    valor="1"/>
  <punto_y_coma columna="22" linea="33" longitud=
    "1"/>
  <END columna="12" linea="34" longitud="3"/>
  <punto_y_coma columna="15" linea="34" longitud=
    "1"/>
  <IDENTIFICADOR columna="12" linea="35" longitud
    ="1" nombre="f"/>
  <asignacion columna="13" linea="35" longitud="2
    "/>
  <IDENTIFICADOR columna="15" linea="35" longitud
    ="3" nombre="f_1"/>
  <mas columna="18" linea="35" longitud="1"/>
  <IDENTIFICADOR columna="19" linea="35" longitud
    ="3" nombre="f_2"/>
  <punto_y_coma columna="22" linea="35" longitud=
    "1"/>
  <END columna="8" linea="36" longitud="3"/>
  <punto_y_coma columna="11" linea="36" longitud=
    "1"/>
  <END columna="4" linea="37" longitud="3"/>
  <punto_y_coma columna="7" linea="37" longitud="
    1"/>
  <BEGIN columna="0" linea="39" longitud="5"/>
  <READ columna="4" linea="40" longitud="4"/>
  <IDENTIFICADOR columna="9" linea="40" longitud=
    "1" nombre="n"/>
  <punto_y_coma columna="10" linea="40" longitud=
    "1"/>
  <CALL columna="4" linea="41" longitud="4"/>
  <IDENTIFICADOR columna="9" linea="41" longitud=
    "9" nombre="fibonacci"/>
  <punto_y_coma columna="18" linea="41" longitud=
    "1"/>
  <WRITE columna="4" linea="42" longitud="5"/>
  <IDENTIFICADOR columna="10" linea="42" longitud
    ="1" nombre="f"/>
  <punto_y_coma columna="11" linea="42" longitud=
    "1"/>
  <END columna="0" linea="43" longitud="3"/>
  <punto columna="3" linea="43" longitud="1"/>
  <fuente>  <![CDATA[(*
    Cálculo de los números de la serie de
        fibonacci:
    f_0 = 1
    f_1 = 1
    f_n = f_{n-1} + f_{n-2}, n>1
*)
const fib_0=1, fib_1=1;
var n, f;

procedure fibonacci;
    var i,   (* Variable para hacer el recorrido
        *)
        f_1, (* Número Fibonacci anterior *)
        f_2; (* Número Fibonacci anterior al
            anterior *)
    begin
        if n=0 then f:=fib_0;
        if n=1 then begin
            f:=fib_1;
            write f; (* Los primeros dos
                elementos son iguales *)
        end;
        if n>1 then begin
            f_1:=1;
            write f_1;

            f_2:=1;
            write f_2;

            i:=2;
            while i<n do begin
                f:=f_1+f_2;
                write f;
                f_2:=f_1;
                f_1:=f;
                i:=i+1;
            end;
            f:=f_1+f_2;
        end;
    end;

begin
    read n;
    call fibonacci;
    write f;
end.
]]>  </fuente>
</lexemas>
```

V-B. *Descripción de la Fase de Análisis Sintáctico*

El propósito de esta fase es convertir —y al mismo tiempo verificar si es posible convertir— la secuencia de elementos léxicos proporcionada por la fase de análisis léxico, en un árbol de sintaxis que representa una cadena que puede ser generada por la gramática del lenguaje fuente.[1], [5], [6]

La fase de análisis sintáctico genera el árbol de sintaxis correspondiente al programa fuente. La estructura general del árbol de sintaxis para todo programa pl0+ es la siguiente:

```
<arbol_de_sintaxis>
  <programa>
    <bloque>
      ...
    </bloque>
  </programa>
  <fuente>  <![CDATA[...]]>  </fuente>
</arbol_de_sintaxis>
```

De acuerdo a la sintaxis de pl0+, todo programa está compuesto por un bloque principal de código.

Todo bloque se representa de la siguiente manera (una secuencia de declaraciones de constantes, variables y procedimientos y opcionalmente una instrucción):

```
<bloque>
  <constante columna="15" linea="7" nombre="fib_1
      " valor="1"/>
  .
  .
  .
  <variable columna="4" linea="8" nombre="n"/>
  .
  .
  .
  <procedimiento columna="10" linea="10" nombre="
      fibonacci">
    <bloque> ... </bloque>
  </procedimiento>
  .
  .
  .
  <!-- Una instrucción opcional -->
</bloque>
```

Las instrucciones de asignación, llamada a procedimiento, secuencias `begin`/`end`, condicionales `if`, ciclos `while` y operaciones de lectura y escritura se representan así:

```
<!-- n := ... -->
<asignacion variable="n">
  <!-- una expresión -->
</asignacion>

<!-- call fibonacci -->
<llamada procedimiento="fibonacci"/>

<!-- begin ... end -->
<secuencia>
  <!-- una o más instrucciones -->
</secuencia>

<!-- if ... then ... else ... -->
<condicional>
  <condicion operacion="...">
    <!-- una o dos expresiones operandos -->
  </condicion>
  <!-- instrucción para verdadero -->
  <!-- instrucción opcional para falso -->
</condicional>

<!-- while ... do ... -->
<ciclo>
  <condicion operacion="...">
    <!-- una o dos expresiones operandos -->
  </condicion>
  <!-- instrucción a repetir -->
</ciclo>

<!-- read n -->
<leer variable="n"/>

<!-- write f -->
<escribir simbolo="f"/>
```

Los parámetros válidos para el atributo `operacion` de la etiqueta `condicion` son: `comparacion` (=), `diferente_de` (<>), `menor_que` (<), `mayor_que` (>), `menor_igual_que` (<=), `mayor_igual_que` (>=).

Las etiquetas válidas como expresión son `suma`, `resta`, `multiplicacion`, `division`, `negativo` y las etiquetas especiales: `<identificador simbolo="f_1"/>` y `<numero valor="10"/>` para identificadores y literales respectivamente.

A continuación se presenta un ejemplo de programa sencillo —válido pero no funcional— con una expresión y un ciclo (programa 6) y la salida correspondiente de su análisis sintáctico (programa 7):

Listing 6. Programa con una expresión y un ciclo

```
const n=5;
var f, i;
begin
    i := (f*-2)*(n*i) + 5*-9/8*i;
    write i;
    while f>i do write f;
end.
```

Listing 7. Salida del Análisis Sintáctico del programa enterior

```
<?xml version="1.0" ?>
<arbol_de_sintaxis>
  <!--
************************************
Por favor, no modifique este archivo
a menos que sepa lo que está haciendo
************************************
-->
<programa>
<bloque>
  <constante columna="6" linea="1" nombre="n"
      valor="5"/>
  <variable columna="4" linea="2" nombre="f"/>
  <variable columna="7" linea="2" nombre="i"/>
  <secuencia columna="0" linea="3">
    <asignacion columna="4" linea="4" variable="i
        ">
      <suma columna="22" linea="4">
        <multiplicacion columna="15" linea="4">
          <multiplicacion columna="11" linea="4">
            <identificador columna="10" linea="4"
                simbolo="f"/>
            <negativo columna="12" linea="4">
              <numero columna="13" linea="4"
                  valor="2"/>
            </negativo>
          </multiplicacion>
          <multiplicacion columna="18" linea="4">
            <identificador columna="17" linea="4"
                simbolo="n"/>
```

```
            <identificador columna="19" linea="4"
                 simbolo="i"/>
          </multiplicacion>
        </multiplicacion>
        <multiplicacion columna="30" linea="4">
          <multiplicacion columna="25" linea="4">
            <numero columna="24" linea="4" valor=
                "5"/>
            <division columna="28" linea="4">
              <negativo columna="26" linea="4">
                <numero columna="27" linea="4"
                    valor="9"/>
              </negativo>
              <numero columna="29" linea="4"
                  valor="8"/>
            </division>
          </multiplicacion>
          <identificador columna="31" linea="4"
              simbolo="i"/>
        </multiplicacion>
      </suma>
    </asignacion>
    <escribir columna="10" linea="5" simbolo="i"/
        >
    <ciclo columna="4" linea="6">
      <condicion columna="10" linea="6" operacion
          ="mayor_que">
        <identificador columna="10" linea="6"
            simbolo="f"/>
        <identificador columna="12" linea="6"
            simbolo="i"/>
      </condicion>
      <escribir columna="23" linea="6" simbolo="f
          "/>
    </ciclo>
  </secuencia>
</bloque>
</programa>
  <fuente>
<![CDATA[const n=5;
var f, i;
begin
    i := (f*-2)*(n*i) + 5*-9/8*i;
    write i;
    while f>i do write f;
end.
]]>  </fuente>
</arbol_de_sintaxis>
```

V-C. Descripción de la Fase de Análisis Semántico

El propósito de esta fase es tomar el árbol de sintaxis y realizar las siguientes actividades:[5], [6]

1. Revisión de la Coherencia de tipos de dato.
   Por ejemplo, verificar que a una variable de tipo cadena no se le asigne un número entero (suponiendo que esto no sea válido en el lenguaje fuente).
2. Conversión implícita entre tipos de dato.
   Por ejemplo, cuando una variable de tipo numérico flotante se pasa como parámetro real a una función que recibe un número entero como parámetro formal, esta fase debe hacer la conversión implícita de flotante a entero si esta aplica o reportar el error.
3. Comprobación del flujo de control.
   Por ejemplo, las proposiciones `break` y `continue` de muchos lenguajes de programación transfieren el flujo de control de un punto a otro. Esta fase debe verificar si existe algún punto válido a dónde transferir el flujo de control (un `for`, `while`, `switch`, `case`, etc.).
4. Comprobaciones de unicidad.
   Por ejemplo verificar que las opciones de un bloque `case` o `switch` no estén repetidas, o que un identificador no esté declarado más de una vez en un mismo ámbito.
5. Comprobaciones relacionadas con nombres.
   Por ejemplo en los lenguajes en los que la palabra `end` (o equivalente) va seguida de un identificador que debe corresponder con el identificador al inicio del bloque.

En el caso del compilador `tradukilo.py`, las funciones de esta fase son:

1. Colocar un código único a cada símbolo (de variable, de constante y de procedimiento) para ser referenciado después. El código incluye información del tipo de símbolo y el ámbito en el que está declarado.
2. Verificar duplicidad de símbolos en el mismo ámbito.
3. Verificar si hay identificadores de procedimiento referenciados en una expresión o en una asignacion (lo cual no es válido en pl0+).
4. Verificar si hay identificadores de constante referenciados en el lado izquierdo de una asignación.
5. Verificar que cada identificador/símbolo referenciado esté en un ámbito válido. Es decir, verifica la "integridad referencial" de los símbolos.

Los códigos asignados en esta fase a los programas `pl0+` siguen las siguientes reglas:

1. El código de todo bloque tiene el prefijo "`b`".
2. El código de toda constante tiene el prefijo "`c`".
3. El código de toda variable tiene el prefijo "`v`".
4. El bloque principal siempre tiene el código "`b0`".
5. Todos los códigos (excepto el del bloque principal) tienen un prefijo formado por el caracter "`_`" y un número correlativo —que comienza desde cero— para el tipo de declaración dentro de ese ámbito.
   Por ejemplo, el código "`v0/2/1_3`" indica que es la cuarta variable del bloque donde está declarada; "`c0_0`" indica que es la primera constante del bloque donde está declarada; "`b0/2_1`" indica que es el segundo bloque declarado en su ámbito; etc.
6. El cuerpo de los códigos indica la ruta de anidamiento en el que los símbolos están declarados.

Para ilustrar la semántica de los códigos de los identificadores, consideremos el siguiente programa fuente:

Listing 8. codigos.pl0+ – Ejemplo de análisis semántico

```
const const_global=56, otra_const_global=9;
var var_global, otra_var_global;
procedure proc;
    write const_global;

procedure proc2;
    call proc;

procedure otro;
    const g=9, m=5;
    var temp, temp2;
    procedure otro2;
```

```
        procedure vacio;;
        procedure otro_vacio;; (*Procedimientos
            vacíos*)
        procedure otro3;
            var i,otro3_var;
            begin
                write g;
                write otra_const_global;
                write m;
                write temp2;
                call proc2;
            end
        ;
    ;
;

write const_global;
.
```

Por ejemplo, analicemos el código de la variable `otro3_var` en el programa 8, que es "v0/2/0/2_1", lo que puede verse en el programa 9. Este código indica que el elemento es una variable, y es la segunda de su ámbito (la primera es `i`).

El cuerpo del código —la secuencia "0/2/0/2" del código de la variable— indica que está declarada en el bloque con código "b0/2/0_2" que es `otro3` según vemos en el programa 9. El último correlativo del cuerpo del código de la variable (que es "2") indica el correlativo de declaración del bloque en el que está declarada, y el resto del cuerpo sin el caracter "/" (que es "0/2/0") indica el cuerpo del código del bloque en el que está declarada. De ahí se sabe que la variable `otro3_var` está declarada en el bloque `otro3`.

El cuerpo completo del código de la variable también permite rastrear toda la jerarquía de procedimientos en los que está inserta. La siguiente tabla lo ilustra:

Elemento	Código	Contenedor inmediato
otro3_var	v0/2/0/2_1	otro3
otro3	b0/2/0_2	otro2
otro2	b0/2_0	otro
otro	b0_2	bloque principal
bloque principal	b0	–

A continuación sigue el resultado del análisis semántico del programa 8:

Listing 9. codigos.pl0+sem – Análisis Semántico del programa anterior

```
<?xml version="1.0" ?>
<arbol_de_sintaxis_revisado>
  <!--
************************************
Por favor, no modifique este archivo
a menos que sepa lo que está haciendo
************************************
-->
<programa>
<bloque codigo="b0">
  <constante codigo="c0_0" columna="6" linea="1"
      nombre="const_global" valor="56"/>
  <constante codigo="c0_1" columna="23" linea="1"
       nombre="otra_const_global" valor="9"/>
  <variable codigo="v0_0" columna="4" linea="2"
      nombre="var_global"/>
  <variable codigo="v0_1" columna="16" linea="2"
      nombre="otra_var_global"/>
  <procedimiento columna="10" linea="3" nombre="
      proc">
    <bloque codigo="b0_0">
      <escribir codigo="c0_0" columna="10" linea=
          "4" simbolo="const_global" valor="56"/>
    </bloque>
  </procedimiento>
  <procedimiento columna="10" linea="6" nombre="
      proc2">
    <bloque codigo="b0_1">
      <llamada codigo="b0_0" columna="9" linea="7
          " procedimiento="proc"/>
    </bloque>
  </procedimiento>
  <procedimiento columna="10" linea="9" nombre="
      otro">
    <bloque codigo="b0_2">
      <constante codigo="c0/2_0" columna="10"
          linea="10" nombre="g" valor="9"/>
      <constante codigo="c0/2_1" columna="15"
          linea="10" nombre="m" valor="5"/>
      <variable codigo="v0/2_0" columna="8" linea
          ="11" nombre="temp"/>
      <variable codigo="v0/2_1" columna="14"
          linea="11" nombre="temp2"/>
      <procedimiento columna="14" linea="12"
          nombre="otro2">
        <bloque codigo="b0/2_0">
          <procedimiento columna="18" linea="13"
              nombre="vacio">
            <bloque codigo="b0/2/0_0"/>
          </procedimiento>
          <procedimiento columna="18" linea="14"
              nombre="otro_vacio">
            <bloque codigo="b0/2/0_1"/>
          </procedimiento>
          <procedimiento columna="18" linea="15"
              nombre="otro3">
            <bloque codigo="b0/2/0_2">
              <variable codigo="v0/2/0/2_0"
                  columna="16" linea="16" nombre=
                  "i"/>
              <variable codigo="v0/2/0/2_1"
                  columna="18" linea="16" nombre=
                  "otro3_var"/>
              <secuencia columna="12" linea="17">
                <escribir codigo="c0/2_0" columna
                    ="22" linea="18" simbolo="g"
                    valor="9"/>
                <escribir codigo="c0_1" columna="
                    22" linea="19" simbolo="
                    otra_const_global" valor="9"/
                    >
                <escribir codigo="c0/2_1" columna
                    ="22" linea="20" simbolo="m"
                    valor="5"/>
                <escribir codigo="v0/2_1" columna
                    ="22" linea="21" simbolo="
                    temp2"/>
                <llamada codigo="b0_1" columna="
                    21" linea="22" procedimiento=
                    "proc2"/>
              </secuencia>
            </bloque>
          </procedimiento>
        </bloque>
      </procedimiento>
    </bloque>
  </procedimiento>
  <escribir codigo="c0_0" columna="6" linea="28"
      simbolo="const_global" valor="56"/>
</bloque>
</programa>
  <fuente>
<![CDATA[const const_global=56, otra_const_global
    =9;
var var_global, otra_var_global;
procedure proc;
    write const_global;
```

```
procedure proc2;
    call proc;

procedure otro;
    const g=9, m=5;
    var temp, temp2;
    procedure otro2;
        procedure vacio;;
        procedure otro_vacio;; (*Procedimientos
            vacíos*)
        procedure otro3;
            var i,otro3_var;
            begin
                write g;
                write otra_const_global;
                write m;
                write temp2;
                call proc2;
            end
        ;
    ;
;

write const_global;
.
]]>  </fuente>
</arbol_de_sintaxis_revisado>
```

V-D. Descripción de la Fase de Generación de Código Objeto

La generación de código objeto (lenguaje p+) se realiza con el objetivo de transformar el código fuente original en "código máquina" para una máquina virtual que interprete exclusivamente programas en lenguaje p+.

A continuación se presentan las reglas generales de traducción al lenguaje p+, y para simplificar la lectura, se define la función $gen : C_{pl0+} \longrightarrow C_{p+}$ que hipotéticamente convierte un fragmento de código fuente en lenguaje pl0+ a una serie de instrucciones en lenguaje p+.

V-D1. Procedimientos: Se traducen en la siguiente secuencia de instrucciones:

$$
\begin{array}{rl}
i\text{:} & \texttt{SAL } b \\
 & \vdots \\
 & \text{(instrucciones de los subprocedimientos)} \\
 & \vdots \\
b\text{:} & \texttt{INS } num \\
 & \vdots \\
 & \text{(instrucciones del procedimiento)} \\
 & \vdots \\
f\text{:} & \texttt{RET}
\end{array}
$$

donde i, b y f son las direcciones de las respectivas instrucciones, $i < b < f$, i es la dirección de inicio del procedimiento y siempre es un salto incondicional a la instrucción de instanciación del bloque, que está en la dirección b, f es la dirección del fin del bloque o procedimiento, num es el número de variables locales al bloque más el espacio necesario para los registros dinámicos (que en el caso de p+ son 3).

En el caso del bloque principal, se cumple que $i = 0$ y f es la última instrucción del programa.

V-D2. Asignaciones: Las instrucciones del tipo `<var> := <expresión>;` se traducen en la siguiente secuencia de instrucciones:

$$
\begin{array}{l}
gen(\texttt{<expresión>}) \\
\texttt{ALM } dif \ \ pos
\end{array}
$$

Primero se genera la secuencia de instrucciones que evalúan la expresión y luego se almacena el resultado —que quedó en el tope de la pila—, en la celda de memoria reservado para la variable `<var>`.

V-D3. Condicionales `if`: Se traducen de manera diferente si tienen o no tienen parte `else`. A continuación se presentan ambas formas:

V-D3a. Sin `else`: La forma "`if` C `then` B_1" se traduce como:

$$
\begin{array}{rl}
 & gen(C) \\
 & \texttt{SAC } d_1 \\
 & gen(B_1) \\
d_1\text{:} & \vdots \\
 & \text{(lo que sigue al } \texttt{if}\text{)}
\end{array}
$$

Primero se genera la secuencia de instrucciones para la condición C, luego se coloca un salto condicional hacia las instrucciones que le siguen al `if`, y luego se genera la secuencia de instrucciones para B_1.

V-D3b. Con `else`: La forma "`if` C `then` B_1 `else` B_2" se traduce como:

$$
\begin{array}{rl}
 & gen(C) \\
 & \texttt{SAC } d_1 \\
 & gen(B_1) \\
 & \texttt{SAL } d_2 \\
d_1\text{:} & gen(B_2) \\
d_2\text{:} & \vdots \\
 & \text{(lo que sigue al } \texttt{if}\text{)}
\end{array}
$$

Primero se genera la secuencia de instrucciones para la condición C, luego se coloca un salto condicional hacia el inicio de las instrucciones de B_2 (la parte del `else`), luego se genera la secuencia de instrucciones de B_1 (la parte del `then`) y después un salto incondicional hacia las instrucciones que le siguen al `if`. Finalmente se generan las instrucciones para B_2 (la parte `else`).

V-D4. Ciclos `while`: Tienen la forma "`while` C `do` B" y se traducen en la siguiente secuencia de instrucciones:

$$
\begin{array}{rl}
d_1\text{:} & gen(C) \\
 & \texttt{SAC } d_2 \\
 & gen(B) \\
 & \texttt{SAL } d_1 \\
d_2\text{:} & \vdots \\
 & \text{(lo que sigue al } \texttt{while}\text{)}
\end{array}
$$

Primero se genera la secuencia de instrucciones para la condición C, luego se coloca un salto condicional hacia la instrucción que le sigue al ciclo `while`, luego se genera la secuencia de instrucciones del cuerpo del ciclo y luego se coloca un salto incondicional hacia el inicio de las instrucciones de la condición C.

V-D5. Llamadas a procedimiento: Tienen la forma "`call <proc>`" y se traducen de la siguiente manera:

$$
\begin{array}{rl}
 & \vdots \\
b\text{:} & \texttt{INS } n \\
 & \vdots \\
 & \text{(instrucciones del procedimiento } \texttt{<proc>}\text{)} \\
 & \vdots \\
f\text{:} & \texttt{RET} \\
 & \vdots \\
d\text{:} & \texttt{LLA } dif\ b \\
d+1\text{:} & \vdots \\
 & \text{(lo que sigue a la llamada)}
\end{array}
$$

La instrucción en la dirección b marca el inicio operativo del procedimiento invocado —que siempre está constituido por una instrucción `INS`— y f su fin —que siempre está consituido por una instrucción `RET`— (ver sección V-D1). Cuando se ejecuta una instrucción como la de la dirección d, se guardan los registros dinámicos necesarios y se salta a la dirección b. El valor dif sirve para calcular el enlace estático para poder identificar las variables de los procedimientos de ámbito superior. Cuando se llega a la instrucción en la dirección f, se regresa el control a la dirección $d+1$ y se reestablecen los registros dinámicos del bloque anterior.

V-D6. Escritura y Lectura: Las instrucciones de la forma "`read <var>`" leen un entero de la entrada estándar y la almacenan en la variable `<var>`. Las instrucciones de la forma "`write <var>`" escriben el valor de la variable `<var>` en la salida estándar. Se traducen a las siguientes secuencias:

`read <var>:`	`write <var>:`
`LEE`	`CAR` dif pos
`ALM` dif pos	`ESC`

V-D7. Expresiones: Un fragmento de código pl0+ del tipo "$operando_1$ $operador$ $operando_2$" se traducirá como sigue:

$$
\begin{array}{l}
gen(operando_1) \\
gen(operando_2) \\
\texttt{OPR } cód
\end{array}
$$

donde $cód$ es el código de operación de $operador$. Los códigos de operación son los mostrados en la siguiente tabla:

Operación	Código
`negativo`	1
`suma`	2
`resta`	3
`multiplicacion`	4
`division`	5
`odd` (impar)	6
`comparacion`	8
`diferente_de`	9
`menor_que`	10
`mayor_igual_que`	11
`mayor_que`	12
`menor_igual_que`	13

V-D8. Ejemplo: A continuación se presenta el resultado de la fase de generación de código del programa 1 en la página 2:

Listing 10. fibonacci.p+ – Código objeto del programa fibonacci.pl0+

```
<?xml version="1.0" ?>
<codigo_pmas>
  <!--
************************************
Por favor, no modifique este archivo
a menos que sepa lo que está haciendo
************************************
-->
<salto_incondicional direccion="0" parametro="55"
    >
  <informacion codigo="b0"
      inicio_de_procedimiento="--PRINCIPAL--"/>
</salto_incondicional>
<salto_incondicional direccion="1" parametro="2">
  <informacion codigo="b0_0" columna="10"
      inicio_de_procedimiento="fibonacci" linea="
      10"/>
</salto_incondicional>
<instanciar_procedimiento direccion="2" parametro
    ="6">
  <informacion codigo="b0_0" columna="10"
      inicio_de_procedimiento="fibonacci" linea="
      10"/>
</instanciar_procedimiento>
<cargar_variable diffnivel="1" direccion="3"
    parametro="3">
  <informacion columna="8" linea="15">
    Inicio de condicional (if-then)
  </informacion>
  <informacion codigo="v0_0" columna="11" linea="
      15" variable="n"/>
</cargar_variable>
<literal direccion="4" parametro="0">
  <informacion columna="13" linea="15"/>
</literal>
<operacion direccion="5" parametro="8">
  <informacion columna="11" linea="15" operacion=
      "comparacion"/>
</operacion>
<salto_condicional direccion="6" parametro="9">
  <informacion columna="8" linea="15">
    Inicio del cuerpo del 'then'
  </informacion>
</salto_condicional>
<literal direccion="7" parametro="1">
  <informacion codigo="v0_1" columna="20" linea="
      15" variable="f">
    Inicio de asignación de variable
  </informacion>
  <informacion codigo="c0_0" columna="23"
      constante="fib_0" linea="15"/>
</literal>
<almacenar_variable diffnivel="1" direccion="8"
```

```
    parametro="4">
  <informacion codigo="v0_1" columna="20" linea="
      15" variable="f"/>
  <informacion columna="8" linea="15">
    Fin de condicional (if-then)
  </informacion>
</almacenar_variable>
<cargar_variable diffnivel="1" direccion="9"
    parametro="3">
  <informacion columna="8" linea="16">
    Inicio de condicional (if-then)
  </informacion>
  <informacion codigo="v0_0" columna="11" linea="
      16" variable="n"/>
</cargar_variable>
<literal direccion="10" parametro="1">
  <informacion columna="13" linea="16"/>
</literal>
<operacion direccion="11" parametro="8">
  <informacion columna="11" linea="16" operacion=
      "comparacion"/>
</operacion>
<salto_condicional direccion="12" parametro="17">
  <informacion columna="8" linea="16">
    Inicio del cuerpo del 'then'
  </informacion>
</salto_condicional>
<literal direccion="13" parametro="1">
  <informacion codigo="v0_1" columna="12" linea="
      17" variable="f">
    Inicio de asignación de variable
  </informacion>
  <informacion codigo="c0_1" columna="15"
      constante="fib_1" linea="17"/>
</literal>
<almacenar_variable diffnivel="1" direccion="14"
    parametro="4">
  <informacion codigo="v0_1" columna="12" linea="
      17" variable="f"/>
</almacenar_variable>
<cargar_variable diffnivel="1" direccion="15"
    parametro="4">
  <informacion codigo="v0_1" columna="18" linea="
      18" variable="f"/>
</cargar_variable>
<escribir direccion="16">
  <informacion columna="8" linea="16">
    Fin de condicional (if-then)
  </informacion>
</escribir>
<cargar_variable diffnivel="1" direccion="17"
    parametro="3">
  <informacion columna="8" linea="20">
    Inicio de condicional (if-then)
  </informacion>
  <informacion codigo="v0_0" columna="11" linea="
      20" variable="n"/>
</cargar_variable>
<literal direccion="18" parametro="1">
  <informacion columna="13" linea="20"/>
</literal>
<operacion direccion="19" parametro="12">
  <informacion columna="11" linea="20" operacion=
      "mayor_que"/>
</operacion>
<salto_condicional direccion="20" parametro="54">
  <informacion columna="8" linea="20">
    Inicio del cuerpo del 'then'
  </informacion>
</salto_condicional>
<literal direccion="21" parametro="1">
  <informacion codigo="v0/0_1" columna="12" linea
      ="21" variable="f_1">
    Inicio de asignación de variable
  </informacion>
  <informacion columna="17" linea="21"/>
</literal>
<almacenar_variable diffnivel="0" direccion="22"
    parametro="4">
  <informacion codigo="v0/0_1" columna="12" linea
      ="21" variable="f_1"/>
</almacenar_variable>
<cargar_variable diffnivel="0" direccion="23"
    parametro="4">
  <informacion codigo="v0/0_1" columna="18" linea
      ="22" variable="f_1"/>
</cargar_variable>
<escribir direccion="24"/>
<literal direccion="25" parametro="1">
  <informacion codigo="v0/0_2" columna="12" linea
      ="24" variable="f_2">
    Inicio de asignación de variable
  </informacion>
  <informacion columna="17" linea="24"/>
</literal>
<almacenar_variable diffnivel="0" direccion="26"
    parametro="5">
  <informacion codigo="v0/0_2" columna="12" linea
      ="24" variable="f_2"/>
</almacenar_variable>
<cargar_variable diffnivel="0" direccion="27"
    parametro="5">
  <informacion codigo="v0/0_2" columna="18" linea
      ="25" variable="f_2"/>
</cargar_variable>
<escribir direccion="28"/>
<literal direccion="29" parametro="2">
  <informacion codigo="v0/0_0" columna="12" linea
      ="27" variable="i">
    Inicio de asignación de variable
  </informacion>
  <informacion columna="15" linea="27"/>
</literal>
<almacenar_variable diffnivel="0" direccion="30"
    parametro="3">
  <informacion codigo="v0/0_0" columna="12" linea
      ="27" variable="i"/>
</almacenar_variable>
<cargar_variable diffnivel="0" direccion="31"
    parametro="3">
  <informacion columna="12" linea="28">
    Inicio de ciclo (while)
  </informacion>
  <informacion codigo="v0/0_0" columna="18" linea
      ="28" variable="i"/>
</cargar_variable>
<cargar_variable diffnivel="1" direccion="32"
    parametro="3">
  <informacion codigo="v0_0" columna="20" linea="
      28" variable="n"/>
</cargar_variable>
<operacion direccion="33" parametro="10">
  <informacion columna="18" linea="28" operacion=
      "menor_que"/>
</operacion>
<salto_condicional direccion="34" parametro="50">
  <informacion columna="12" linea="28">
    Inicio del cuerpo del ciclo
  </informacion>
</salto_condicional>
<cargar_variable diffnivel="0" direccion="35"
    parametro="4">
  <informacion codigo="v0_1" columna="16" linea="
      29" variable="f">
    Inicio de asignación de variable
  </informacion>
  <informacion codigo="v0/0_1" columna="19" linea
      ="29" variable="f_1"/>
</cargar_variable>
<cargar_variable diffnivel="0" direccion="36"
    parametro="5">
  <informacion codigo="v0/0_2" columna="23" linea
      ="29" variable="f_2"/>
</cargar_variable>
```

```
<operacion direccion="37" parametro="2">
  <informacion columna="22" linea="29" operacion=
      "suma"/>
</operacion>
<almacenar_variable diffnivel="1" direccion="38"
    parametro="4">
  <informacion codigo="v0_1" columna="16" linea="
      29" variable="f"/>
</almacenar_variable>
<cargar_variable diffnivel="1" direccion="39"
    parametro="4">
  <informacion codigo="v0_1" columna="22" linea="
      30" variable="f"/>
</cargar_variable>
<escribir direccion="40"/>
<cargar_variable diffnivel="0" direccion="41"
    parametro="4">
  <informacion codigo="v0/0_2" columna="16" linea
      ="31" variable="f_2">
    Inicio de asignación de variable
  </informacion>
  <informacion codigo="v0/0_1" columna="21" linea
      ="31" variable="f_1"/>
</cargar_variable>
<almacenar_variable diffnivel="0" direccion="42"
    parametro="5">
  <informacion codigo="v0/0_2" columna="16" linea
      ="31" variable="f_2"/>
</almacenar_variable>
<cargar_variable diffnivel="1" direccion="43"
    parametro="4">
  <informacion codigo="v0/0_1" columna="16" linea
      ="32" variable="f_1">
    Inicio de asignación de variable
  </informacion>
  <informacion codigo="v0_1" columna="21" linea="
      32" variable="f"/>
</cargar_variable>
<almacenar_variable diffnivel="0" direccion="44"
    parametro="4">
  <informacion codigo="v0/0_1" columna="16" linea
      ="32" variable="f_1"/>
</almacenar_variable>
<cargar_variable diffnivel="0" direccion="45"
    parametro="3">
  <informacion codigo="v0/0_0" columna="16" linea
      ="33" variable="i">
    Inicio de asignación de variable
  </informacion>
  <informacion codigo="v0/0_0" columna="19" linea
      ="33" variable="i"/>
</cargar_variable>
<literal direccion="46" parametro="1">
  <informacion columna="21" linea="33"/>
</literal>
<operacion direccion="47" parametro="2">
  <informacion columna="20" linea="33" operacion=
      "suma"/>
</operacion>
<almacenar_variable diffnivel="0" direccion="48"
    parametro="3">
  <informacion codigo="v0/0_0" columna="16" linea
      ="33" variable="i"/>
</almacenar_variable>
<salto_incondicional direccion="49" parametro="31
    ">
  <informacion columna="12" linea="28">
    Fin del cuerpo del ciclo
  </informacion>
</salto_incondicional>
<cargar_variable diffnivel="0" direccion="50"
    parametro="4">
  <informacion codigo="v0_1" columna="12" linea="
      35" variable="f">
    Inicio de asignación de variable
  </informacion>
  <informacion codigo="v0/0_1" columna="15" linea
      ="35" variable="f_1"/>
</cargar_variable>
<cargar_variable diffnivel="0" direccion="51"
    parametro="5">
  <informacion codigo="v0/0_2" columna="19" linea
      ="35" variable="f_2"/>
</cargar_variable>
<operacion direccion="52" parametro="2">
  <informacion columna="18" linea="35" operacion=
      "suma"/>
</operacion>
<almacenar_variable diffnivel="1" direccion="53"
    parametro="4">
  <informacion codigo="v0_1" columna="12" linea="
      35" variable="f"/>
  <informacion columna="8" linea="20">
    Fin de condicional (if-then)
  </informacion>
</almacenar_variable>
<retornar direccion="54">
  <informacion fin_de_procedimiento="fibonacci"/>
</retornar>
<instanciar_procedimiento direccion="55"
    parametro="5">
  <informacion codigo="b0"
      inicio_de_procedimiento="--PRINCIPAL--"/>
</instanciar_procedimiento>
<leer direccion="56">
  <informacion codigo="v0_0" columna="9" linea="
      40" variable="n"/>
</leer>
<almacenar_variable diffnivel="0" direccion="57"
    parametro="3">
  <informacion codigo="v0_0" columna="9" linea="
      40" variable="n"/>
</almacenar_variable>
<llamar_procedimiento diffnivel="0" direccion="58
    " parametro="1">
  <informacion codigo="b0_0" columna="9" linea="
      41" procedimiento="fibonacci"/>
</llamar_procedimiento>
<cargar_variable diffnivel="0" direccion="59"
    parametro="4">
  <informacion codigo="v0_1" columna="10" linea="
      42" variable="f"/>
</cargar_variable>
<escribir direccion="60"/>
<retornar direccion="61">
  <informacion fin_de_procedimiento="--PRINCIPAL
      --"/>
</retornar>
  <ensamblador>
<![CDATA[
   0 SAL    -         55
   1 SAL    -          2
   2 INS    -          6
   3 CAR    1          3
   4 LIT    -          0
   5 OPR    -          8
   6 SAC    -          9
   7 LIT    -          1
   8 ALM    1          4
   9 CAR    1          3
  10 LIT    -          1
  11 OPR    -          8
  12 SAC    -         17
  13 LIT    -          1
  14 ALM    1          4
  15 CAR    1          4
  16 ESC    -          -
  17 CAR    1          3
  18 LIT    -          1
  19 OPR    -         12
  20 SAC    -         54
  21 LIT    -          1
  22 ALM    0          4
  23 CAR    0          4
```

```
  24 ESC     -           -
  25 LIT     -           1
  26 ALM     0           5
  27 CAR     0           5
  28 ESC     -           -
  29 LIT     -           2
  30 ALM     0           3
  31 CAR     0           3
  32 CAR     1           3
  33 OPR     -          10
  34 SAC     -          50
  35 CAR     0           4
  36 CAR     0           5
  37 OPR     -           2
  38 ALM     1           4
  39 CAR     1           4
  40 ESC     -           -
  41 CAR     0           4
  42 ALM     0           5
  43 CAR     1           4
  44 ALM     0           4
  45 CAR     0           3
  46 LIT     -           1
  47 OPR     -           2
  48 ALM     0           3
  49 SAL     -          31
  50 CAR     0           4
  51 CAR     0           5
  52 OPR     -           2
  53 ALM     1           4
  54 RET     -           -
  55 INS     -           5
  56 LEE     -           -
  57 ALM     0           3
  58 LLA     -           1
  59 CAR     0           4
  60 ESC     -           -
  61 RET     -           -
]]>  </ensamblador>
  <fuente>
<![CDATA[(*
    Cálculo de los números de la serie de
        fibonacci:
    f_0 = 1
    f_1 = 1
    f_n = f_{n-1} + f_{n-2}, n>1
*)
const fib_0=1, fib_1=1;
var n, f;

procedure fibonacci;
    var i,   (* Variable para hacer el recorrido
        *)
        f_1, (* Número Fibonacci anterior *)
        f_2; (* Número Fibonacci anterior al
            anterior *)
    begin
        if n=0 then f:=fib_0;
        if n=1 then begin
            f:=fib_1;
            write f; (* Los primeros dos
                elementos son iguales *)
        end;
        if n>1 then begin
            f_1:=1;
            write f_1;

            f_2:=1;
            write f_2;

            i:=2;
            while i<n do begin
                f:=f_1+f_2;
                write f;
                f_2:=f_1;
                f_1:=f;
                i:=i+1;
            end;
            f:=f_1+f_2;
        end;
    end;

begin
    read n;
    call fibonacci;
    write f;
end.
]]>  </fuente>
</codigo_pmas>
```

Referencias

[1] Niklaus Wirth, *Compiler construction*, Zürich, November 2005. ISBN 0-201-40353-6.

[2] Michael L. Scott, *A Guide to the Rochester PL/0 Compiler*, Computer Science Department of University of Rochester, Agosto de 2004.

[3] Niklaus Wirth, *The Programming Language Oberon*, Revision 1.10.90.

[4] Niklaus Wirth y Jürg Gutknecht, *Project Oberon – The Design of an Operating System and Compiler*, Edition 2005.

[5] María Luisa González Díaz, *Introducción a la construcción de compiladores*, Departamento de Informática de la Universidad de Valladolid.

[6] Alfred V. Aho, Ravi Sethi y Jeffrey D. Ullman, *Compiladores – Principios, técnicas y herramientas*, Pearson 1990. ISBN 968-444-333-1.

[7] Niklaus Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall 1975. ISBN 0-13-022418-9.

[8] Parrot Foundation, *Proyecto Parrot*. Web oficial: http://www.parrot.org/, web de documentación oficial: http://docs.parrot.org/ (revisión a fecha 24 de septiembre de 2010).

[9] Andrew S. Tanenbaum y Gregory J. Sharp, *The Amoeba Distributed Operating System*, Vrije Universiteit, Web oficial del proyecto: http://www.cs.vu.nl/pub/amoeba/ (revisión a fecha 23 de septiembre de 2010).

[10] Eduardo Navas, *Código de* pl0+. http://dei.uca.edu.sv/publicaciones/pl0+-2010-10-09-12-23.tar.gz