HERRAMIENTA DE APOYO AL DESARROLLO DE COMPILADORES EN PROGRAMACIÓN ORIENTADA POR OBJETOS

Humberto Bello
89-20552@usb.ve

Leonid Tineo R.
leonid@usb.ve

Rosseline Rodríguez
crodrig@usb.ve

Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información.
Aptdo. Postal 89000 Caracas 1080-A Venezuela

Resumen

En el desarrollo de compiladores e interpretadores suele ser preciso el uso de aplicaciones generadoras de analizadores sintácticos; labor que realizan muy bien herramientas conocidas. Sin embargo, éstas en su mayoría generan código para lenguajes tradicionales. Es necesario facilitar la construcción de compiladores mediante programación orientada por objetos lo que conduce a la creación de una herramienta en este paradigma. Se presenta a HADCO, un generador de analizadores sintácticos descendentes tipo LL(1), estructurados y concebidos bajo el paradigma de programación orientada por objetos. HADCO pretende mejorar la calidad del código generado con respecto a herramientas similares, para facilitar las labores de mantenimiento y reutilización del código, así como brindar a los desarrolladores que trabajan con programación orientada por objetos un generador de compiladores cuyos resultados no violen dicho paradigma.

Palabras Claves: Compiladores, Programación Orientada a Objetos, Parsers

1 Introducción

Al momento de desarrollar compiladores e interpretadores se hace muy necesario usar herramientas automatizadas de ayuda a la construcción de analizadores sintácticos, pues la creación manual de tales analizadores añade un volumen de trabajo el cual es de muy poco interés y suele ser repetitivo. Estas herramientas facilitan al implementador su labor, ya que se basan en especificaciones de alto nivel para el lenguaje a reconocer (gramáticas libres de contexto, no ambiguas); produciendo módulos compilables en algún lenguaje de programación, que se encargan de realizar el análisis sintáctico.

Si bien es cierto que herramientas conocidas en la actualidad y de larga trayectoria, como lo son LEX [11] y YACC[10], resuelven de manera versátil y eficiente el problema anterior; en su mayoría, éstas generan código en lenguajes de programación tradicionales en el enfoque imperativo, mas no se adaptan a tendencias más actuales en el desarrollo de aplicaciones como lo es la Programación Orientada por Objetos [8]. Si alguien pretende desarrollar un compilador en un lenguaje bajo este enfoque (tal vez C++ [13]), querría aprovechar las virtudes de este paradigma de programación en todo el desarrollo de su aplicación, por ejemplo desearía que: el reconocedor fuera implementado como una clase que ofreciera a través de sus métodos la manipulación de atributos tales como el estado actual del autómata y el token que se está procesando, así como su ubicación en el texto fuente; además de implantar un mecanismo decente de recuperación de errores sintácticos que fuera flexible y redefinible por los implementadores [3].

Con las herramientas actuales, el programador tendría que: o bien hacer una mezcla de paradigmas para aprovechar los módulos generados por las mismas [7], lo que no resulta muy engorroso, mas sí muy poco elegante e inadecuado, con lo que tendría que desistir a posibles manejos sofisticados que le permitiría la Programación Orientada por Objetos; o decidir construir "desde cero" su reconocedor, lo que involucraría un gran esfuerzo de trabajo el cual podría ser desechado con sólo pocas modificaciones del lenguaje a reconocer [3].

Es así como, se propone una Herramienta para el Apoyo al Desarrollo de COmpiladores que sea adecuada al paradigma de Programación Orientada por Objetos, es decir que genere la implementación de las clases necesarias para el reconocimiento sintáctico de un lenguaje [5]; tal herramienta es HADCO. Esta aplicación toma las especificaciones de un lenguaje, en un formato similar al de YACC y genera un reconocedor estructurado, que respeta el paradigma de programación empleado, lo cual redunda en mejoras de la calidad del código y en la facilidad de mantenimiento. Se tiene una experiencia exitosa de HADCO dentro de un proyecto para la enseñanza de programación que se está desarrollando [3][12].

Este trabajo se presenta en cinco secciones posteriores a esta: en la sección 2 se da una descripción completa de las especificaciones de entrada que recibe HADCO; luego se explica la funcionalidad de esta herramienta, en la sección 3; la sección 4 se ha dedicado a mostrar el diseño general de los módulos y clases generados por HADCO; en la sección 6 se comenta la experiencia tenida con el uso de esta herramienta; y finalmente se presentan las conclusiones (sección 6).

2 La Entrada de HADCO

HADCO recibe como entrada las especificaciones sintácticas del lenguaje a reconocer. Éstas especificaciones no son más que un conjunto de reglas de producción correspondiente a una gramática LL(1), factorizada y sin recursión izquierda [2]. La restricción a este tipo de gramáticas obedece al hecho de que se ha escogido como estrategia de parsing para la implantación del reconocedor, el parsing descendente predictivo con análisis de un símbolo de entrada [1]. Sin embargo, es de notar que esta restricción no es realmente fuerte, pues la mayoría de los lenguajes de programación conocidos pueden especificarse de esta manera [9].

En cuanto a la sintaxis del lenguaje de especificación de gramáticas hay que destacar que es muy similar al usado por YACC. Se ha decidido establecer en estas especificaciones que los identificadores para símbolos terminales se inicien con una letra minúscula y los no terminales con una mayúscula; con la aplicación de esta regla, el conjunto de símbolos gramaticales es inferido a

partir de las especificaciones de las producciones, sin la necesidad de tener una sección adicional para declarar los identificadores que se refieren a símbolos terminales de la gramática. La sintaxis en notación BNF de este lenguaje de especificación es presentada en la figura 1.

```
<gramática>::=<definiciones><sepdef><reglas>

<definiciones>::=Texto

<sepdef>::=@@

<reglas>::=<regla><sepreg><reglas> | <regla>

<regla>::=<l_izq><sepprod><l_der>

<l_izq>::=<NoTerm>

<sepprod>::= :

<l_der>::=<SimbGram><l_der> | <SimbGram>

<SimbGram>::=<NoTerm>|<Terminal>
<SimbGram>::=<Esquema>

<sepreg>::='\n'

<NoTerm>::=<Mayúscula><identif>

<Terminal>::=<Minúscula><identif> | $ | *

<Esquema>::=@ Texto @

<identificador>::=<letra>|<letra><identificador>
<identificador>::=<dígito><identificador>

<letra>::=<Mayúscula> | <Minúscula>

<Mayúscula>::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O
<Mayúscula>::=P|Q|R|S|T|U|V|W|X|Y|Z>

<Minúscula>::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p
<Minúscula>::=p|q|r|s|t|u|v|w|x|y|z

<dígito>::= 0|1|2|3|4|5|6|7|8|9
```

Figura 1: Definición en BNF para especificaciones en HADCO

Las definiciones que pueden colocarse en la especificación de la gramática, antes de las reglas, corresponde a definiciones en el lenguaje objeto de HADCO, las cuales son incorporadas en la implementación de los módulos generados por la herramienta, estas definiciones serán eventualmente usadas por los esquemas de traducción.

Los esquemas de traducción son especificados mediante la inserción de código en la gramática. El código insertado está delimitado entre un par de símbolos @ y puede ocurrir en cualquier parte del lado derecho de una producción. También se permite heredar y sintetizar atributos a través de la pila de símbolos gramaticales, mediante una manipulación de estos al estilo de YACC; para esto se usan los metasímbolos *$$* y *$<n>* que respectivamente se refieren al símbolo del lado izquierdo de la producción asociada y al n-ésimo símbolo de izquierda a derecha en el lado derecho de las producción.

De modo de facilitar al usuario la tarea de transformar la entrada para que cumpla con las restricciones anteriores, se explicitaron los terminales $\in$(épsilon o vacío) y *$* (delimitador derecho de la entrada), que son representados respectivamente por los caracteres (*) y (*$*). En la figura 2 se muestra un ejemplo de esquema de traducción para una gramática de operadores.

```
E: T @ $2.i:=$0.val; @ R @  $$.val:=$1.s @

R: plus T @ $3.y:= $$.y + $1.val @ R @  $$.s:=$3.s @
R: minus T @ $3.y:= $$.y + $1.val @ R @  $$.s:=$3.s @
R: * @ $$.s:= $$.i @

T: lpar E rpar @ $$.val := $1.val @
T: number @ $$.val:=$0.lexval @
```

Figura 2: Ejemplo de una gramática de operadores especificada en HADCO con esquemas de traducción.

3 Funcionalidades de la herramienta

Dada una gramática con la especificaciones mencionadas en la sección anterior, se pueden identificar dos funcionalidades que realiza HADCO con dicha entrada, las cuales comprenden la verificación de restricciones LL(1) para la eliminación de conflictos y a continuación, la generación del reconocedor.

Verificación de las restricciones LL(1)

Durante esta fase, HADCO detecta recursiones izquierdas directas e indirectas, así como conflictos comúnmente encontrados en gramáticas LL(1). Los conflictos detectados son reportados al usuario mediante mensajes relevantes, tomando las siguientes políticas para su tratamiento:

a) **Conflictos con producciones de la forma** $\{ S \rightarrow a\beta \}$ **y** $\{ S \rightarrow a\zeta \}$**:** en este caso, se toma como válida la primera producción que aparece en la gramática y se desechan todas las demás que originan conflicto con dicha producción.

b) **Conflictos con producciones de la forma** $\{ S \rightarrow \in \}$ **y** $\{ S \rightarrow a\zeta \}$**:** los conflictos causados por producciones épsilon son siempre resueltos en favor de la producción que deriva en la cadena no nula.

Generación del reconocedor

Una vez verificadas las restricciones sobre la entrada, ésta se transforma en una representación más compacta en memoria secundaria, que elimina elementos irrelevantes de la entrada, tales como espacios en blanco y separadores. Esta representación será posteriormente utilizada en el cálculo de la tabla de predicción y en la generación de los otros módulos involucrados en el analizador sintáctico. La escogencia de representar gramática, alfabetos y tabla de predicción en memoria secundaria, pretende mejorar la utilización de la memoria principal con respecto a los analizadores generados por YACC. La ganancia está en el uso de la memoria

dinámica versus implementación completa en la memoria dispuesta para código y datos (implementación de YACC), así en general todas las estructuras involucradas en los reconocedores generados por HADCO trabajan con memoria dinámica, y permiten de esta forma usar tamaños de tablas superiores a la memoria convencional.

4 Los módulos generados por HADCO

La salida producida por la herramienta está constituida fundamentalmente por cuatro módulos documentados: tabla de predicción, tabla de esquemas, parser, y manejador de parser; todos contenedores de las clases necesarias para llevar a cabo el proceso de reconocimiento sintáctico.

Tabla de predicción

Este módulo contiene la clase TABLALL, la cual se encarga de representar la función de transición de la máquina de estados asociada al reconocedor generado. Dicha función se define como

$$F: \Sigma x N \rightarrow (\Sigma \cup N)^{+} \cup \{error\}$$

donde Σ y N son respectivamente, el conjunto de símbolos terminales y no terminales de la gramática suministrada por el usuario.

El espacio en memoria ocupado por la tabla es $|\Sigma| \; x \; |N| \; x \; T$ donde T es el tamaño es bytes de una dirección de memoria. Esta clase provee los métodos requeridos por el parser para el cálculo de transiciones, manejo de conflictos gramaticales y depuración del autómata. A continuación se muestra la interfaz de la clase:

```
class TablaLL(Objeto)
   {
    public
       TablaLL(Alfabeto Alf);
       TablaLL~; virtual;
       TablaLL(Archivos Fuente,Alfabeto);
       TablaLL~(Archivos Fuente,Alfabeto); virtual;
```

```
    void Asignar_Entrada(Terminal T,NoTerminal NTE, Regla R);
    int Tam_en_k();
    int NumTerminales();
    int NumNo_terminales();
    int mostrar(); virtual;
    Alfabeto Alfabeto_Gram();
  private


}
```

Tabla de esquemas

Módulo responsable de la representación de los esquemas de traducción insertados en las producciones. Fundamentalmente se maneja una tabla, en donde se almacenan las funciones en el orden en que aparecen los esquemas asociados a ellas en la gramática. Esta clase ofrece al parser los métodos necesarios para la ejecución de los esquemas de traducción y depuración de los mismos. A continuación se muestra la interfaz de la clase:

```
class TablaEsquemas(Objeto)
 { public
    TablaEsquemas();
    TablaEsquemas~(); virtual;
     int Ejecutar(int num_esq, pila p);
     int mostrar(); virtual;
  private
    .
    .
 }
```

Módulo de Parser

La clase PARSER contenida en este módulo implementa el algoritmo de reconocimiento haciendo uso de una pila y la tabla de predicción. Se ofrecen aquí métodos de sincronización en la pila con la secuencia de entrada definible por el usuario para recuperación de errores. Así mismo se permite al parser heredar atributos iniciales externos suministrados por el usuario y después de completarse el proceso de compilación recuperar (sintetizar) atributos con el resultado del mismo (por ejemplo, árbol abstracto). Finalmente se da acceso a información sobre la entrada

suministrada por el lexicográfico tal como la posición (fila, columna), el token actual, entre otros.

A continuación se muestra la interfaz de la clase:

```
class Parserll(objeto)
 { public
    Parserll~(TablaLL PTLL_T);
    Parserll~(); virtual;
    int arrancar();
    int procesar_token(Terminal PTER_T);
    int exito();
    int sincronizar(Analizador_Lexico l,Pila P);
    Atributo atrib_sintetizar();
    void atrib_heredar(Atributo Attr);
  private
     .
     .
}
```

Manejador de Parser

La clase manejadora de parser ofrece al usuario un nivel de abstracción adecuado para el uso del parser. Se integran en este módulo todos los generados y se suministran métodos como asignar el archivo fuente a analizar, heredar los atributos iniciales, y sintetizar los resultantes de la compilación, así como estadísticas (líneas compiladas y otras definibles por el usuario). A continuación se muestra la interfaz de la clase:

```
class   MParserll(objeto)
 { public
        MParserll(NombreArchivos ArchTabl,ArchAlf);
        MParserll~();
        procedure asignar_fuente(NombreArchivos ArchFuente);
        int Reconocer();
        int Heredar(Atributo A);
        Atributo Resultado();
        int Linea();
        int Columna();
        int Mostrar_Estadisticas();
  private
        AnalizadorLexico L;
        ParserLL P;
              .
              .
}
```

Estos cuatro módulos mencionados actualmente se generan en lenguaje Turbo Pascal con Objetos[6], pero fácilmente se puede producir una versión para C++ que preservará la estructura de las clases empleadas en el prototipo.

El reconocedor generado puede integrarse perfectamente con cualquier analizador lexicográfico tipo LEX, que asocie los tokens reconocidos con números enteros. Además se le permite al usuario definir sus propias heurísticas de recuperación ante errores en la entrada, así como la limpia manipulación de atributos como el token procesado, el estado actual del autómata, entre otros. Desde el punto de vista de costos de representación, el espacio ocupado en memoria principal por las tablas, puede ser menor que el de las tablas generadas por YACC. El overhead producido al usar programación orientada por objetos es poco significativo en comparación con los beneficios que desde el punto de vista de desarrollo y mantenimiento ofrece esta técnica.

5 Experiencias del uso de HADCO

En la Universidad Simón Bolívar, se ha tenido una experiencia exitosa del uso de HADCO en la implementación de un compilador para el lenguaje algorítmico EPA [4] dentro del marco del proyecto HEEPA[12]. Este compilador fue desarrollado usando Turbo Pascal con Objetos [6], donde la utilidad de HADCO se hizo patente al compararla con el uso de programación híbrida en la implementación de un intérprete del mismo lenguaje [7], que fue necesaria debido a que originalmente se disponía de YACC como generador de reconocedores y la aplicación estaba siendo desarrollada en Turbo Pascal con Objetos. En este último caso se tuvo muchos problemas con la integración de los dos lenguajes (C y Turbo Pascal con Objetos) debido a la falta de documentación y a la disimilitudes entre los paradigmas usados, lo que produjo que la implementación se hiciera larga y tediosa.

Con esta experiencia se llegó a la conclusión de que la disposición de una herramienta como HADCO facilitaría no sólo el desarrollo de las fases de diseño e implementación de un compilador sino también la fase de mantenimiento.

6 Conclusiones

La herramienta de apoyo para el desarrollo de compiladores (HADCO), es un intento por mejorar la implementación de lenguajes bajo el paradigma de programación orientada por objetos, que se caracteriza por la búsqueda de una programación de mayor calidad y fácil mantenimiento, sin perder la eficiencia y poder que brindan otras herramientas similares.

La escogencia de la estrategia de parsing predictivo descendente permite la construcción de reconocedores eficientes tanto en el uso de memoria como del tiempo de computación. además, esta estrategia es aplicable a muchos de los lenguajes de programación conocidos, lo que da un gran potencial de uso de HADCO.

La sencillez de las especificaciones de entrada de las gramáticas para HADCO es también una virtud de esta herramienta que facilitaría su uso por parte de desarrolladores de compiladores.

Un reconocedor generado por HADCO puede hacer uso de cualquier analizador lexicográfico que asocie los tokens reconocidos con números enteros, lo cual le da versatilidad de poder ser integrado con otras herramientas. Se ha realizado una versión "escueta" de una herramienta para generar analizadores lexicográficos bajo el paradigma de programación orientada a objetos, esta versión por ahora se restringe a un conjunto limitado de expresiones, que se pretende extender a futuro de forma de tener una herramienta completa.

La versión actual de HADCO está hecha para la construcción de reconocedores en Turbo Pascal con Objetos, esta escogencia se basó en el hecho de que se quería aplicar esta herramienta en el desarrollo de un compilador integrable a un ambiente de programación desarrollado en dicho lenguaje. Sin embargo, esta herramienta puede ser adaptada fácilmente a otros lenguajes que usan el paradigma de la programación orientada por objetos como C++. Otro aporte de la actual versión es tener una herramienta de este tipo para Turbo Pascal con Objetos, ya que no se disponía previamente de un generador de reconocedores sintácticos para Pascal.

Referencias

[1] Aho A., Sethi R., Ullman J. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Aho A., Ullman J. *The Theory of Parsing, Translation and Compiling*. Prentice Hall, Englewood Cliffs, N.J., 192.

[3] Bello H. *Herramienta para la Enseñanza de Algoritmia: Módulo de Compilación de EPA* Proyecto de Grado Universidad Simón Bolívar, 1995.

[4] Bello H, Rodríguez R, Tineo L. Características de un Lenguaje de Programación para El Pensamiento Algorítmico. *Actas de la XLV Convención Anual de ASOVAC*, Caracas, 1995.

[5] Bello H, Rodríguez R, Tineo L. Desarrollo de un Compilador para el Lenguaje EPA. *Actas de la XLV Convención Anual de ASOVAC*, Caracas, 1995.

[6] Borland International. *Borland Pascal with Objects, Programmers's References*.1992

[7] Cárdenas J. *Herramienta para la Enseñanza de Algoritmia: Intérprete de EPA*. Proyecto de Grado Universidad Simón Bolívar, 1995.

[8] Cox B. *Object-Oriented Programming: An Evolutionary Approach*. Reading, Mass.: Addison-Wesley, 1986.

[9] Ghezzi C., Jazayeri M. *Programming Language Concepts*. John Wiley & Sons 1982.

[10]Johnson S. *YACC- Yet Another Compiler Compiler*. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N. J. 1975.

[11]Lesk M. *LEX- A lexical Analizer Generator*. Computing Science Technical Report 38, AT&T Bell Laboratories, Murray Hill, N. J. 1975.

[12]Rodríguez R., Tineo L. HEEPA; Una Herramienta para la Enseñanza de "El Pensamiento Algorítmico", *Actas de la Conferencia III WEI- IV EDUC* , Canela, Brasil 1995.

[13]Stroustrup B. *The C++ Programming Language*. Prentice-Hall, 1992.