Analizador léxico y sintáctico

Directiva declare_list (tema P4) de PHP

Alumno: Boglioli, Alan
Legajo: 38507

Introducción

El proyecto realizado se enfoca en ser un analizador léxico y sintáctico para la declaración *declare_list* (tema P4) del lenguaje PHP. La directiva *declare_list* se refiere a la lista, que se pasa como argumento, cuando se llama a la función "declare(...)":

*declare(**declare_list**) {*
    ...
}

Según la documentación oficial de PHP:

> El constructor declare es usado para fijar directivas de ejecución para un bloque de código. La sintaxis dedeclare es similar a la sintaxis de otros constructores de control de flujo:
>
> declare (directive)
> statement
>
> La sección directive permite que el comportamiento de declare sea configurado. Actualmente, sólo dos directivas están reconocidas: ticks y encoding.
>
> La parte *statement* del bloque *declare* será ejecutada - como se ejecuta y que efectos secundarios ocurran durante la ejecución puede depender de la directiva fijada en el bloque *directive*.

Por lo tanto, la directiva *declare_list* es el objetivo a analizar del programa desarrollado en base a este proyecto

El ejecutable encargado de realizar dicho análisis se llama ***analizadorP4.exe***, el cual se encuentra en la carpeta principal del proyecto.

Disposición de carpetas y archivos dentro del proyecto

En la carpeta princiapl del proyecto se pueden encontrar 3 archivos importantes:

- ***analizadorP4.exe***: El programa que se encarga de analizar léxica y sintácticamente las cadenas ingresadas o los archivos de texto que se le pasen en base a la especificación de declare_list. Su utilización se detalla más adelante en este documento.
- ***src/***: carpeta que contiene los archivos fuentes para compilar el programa. En esta se encuentran tanto los archivos fuentes de flex y bison, y el código fuente en C.

- ***doc/***: carpeta que contiene la especificación de la gramática en .txt (la misma que se encuentra más adelante) y este mismo documento en formato .doc para su edición.
- ***compile_windows.ba***t: archivo por lotes que contiene la secuencia de ejecución de los progamas para compilar el proyecto. (compile_linus.sh contiene la misma secuencia para la compilación en el sistema Linux).
- Archivos .txt con el nombre "***testX***" y "***errorX***", los cuales contienen declaraciones de *declare_list* para probar el analizador. Los archivos "errorX" contienen errores para probar las funciones de detección de errores en el analizador

Herramientas utilizadas

- Flex (generador de analizador léxico)
- Bison (generador de analizador sintáctico)
- Plataforma MinGW para la utilización de herramientas GN
- Compilador GCC (perteneciente a MinGW)
- Sublime Text 3, editor de ficheros con resaltado de sintaxis para archivos bison y flex.
- LibreOffice para la creación de la documentación y especificación del proyecto.
-

Todos las herramientas mencionados anteriormente tienen licencia GPL, a excepción de Sublime Text 3. Es decir, son software libre.

Descripción de la gramática

Especificación EBNF

```
declare_list ::= T_STRING "=" static_scalar { "," T_STRING "=" static_scalar }

T_STRING ::= LABEL

static_scalar ::=       common_scalar
                |       T_STRING
                |       "+" static_scalar
                |       "-" static_scalar
                |       "array" "(" [static_array_pair_list] ")"
                |       static_class_constant

LABEL ::= (letter | "_") {letter | digit | "_"}

common_scalar ::=   T_LNUMBER
                |       T_DNUMBER
                |       T_CONSTANT_ENCAPSED_STRING
                |       "__LINE__"
                |       "__FILE__"
                |       "__CLASS__"
                |       "__METHOD__"
                |       "__FUNCTION__"
```

static_array_pair_list ::= static_array_pair { "," static_array_pair } [","]

static_class_constant ::= T_STRING "::" T_STRING

letter ::= "a".."z" | "A".."Z" | "\x7f".."\xff"

digit ::= "0".."9"

T_LNUMBER ::= LNUM

T_DNUMBER ::= DNUM | EXPONENT_DNUM

T_CONSTANT_ENCAPSED_STRING ::= single_quoted_constant_string
| double_quoted_constant_string

static_array_pair ::= static_scalar ["=>" static_scalar]

LNUM ::= octal | decimal | hexadecinal

DNUM ::= {digit} "." digit {digit} | digit {digit} "." {digit}

EXPONENT_DNUM ::= (LNUM | DNUM) ("e"|"E") ["+"|"-"] LNUM

single_quoted_constant_string ::= "'" { T_ANY_CHAR | "\" T_ANY_CHAR } "'"

double_quoted_constant_string ::= "\"" { T_ANY_CHAR | "\" T_ANY_CHAR } "\""

octal ::= "0" {"0".."7"}

decimal ::= "1".."9" {digit} ;

hexadecinal ::= "0x" hexdigit {hexdigit}

T_ANY_CHAR = "\x00" .. "\xff" /* Código ASCII */

hexdigit ::= {digit} | "a".."f" | "A".."F"

Observación del alumno

La especificación EBNF de *declare_list*, obtenida de la especificación EBNF proporcionada en el campus virtual de Sintaxis y Semántica de los Lenguajes recibió algunas correcciones ya que se encontraron diversos errores. Algunos de esos errores fueron:

- El error más grave que se encontró fue que las definiciones en EBNF estaba terminadas por un punto y coma (;). Al principio confunde, debido a que se piensa que todas las declaraciones deben estar terminadas por un ;. Pero analizando bien todas las

declaraciones, se puede observar que las que tienen que terminar con ; tienen al final el símbolo ";". Es decir, el símbolo ; al final está demás.
- En algunas declaraciones se encontraban como comentarios la frase "FIX ME", como por ejemplo, en la declaración de *single_quoted_constant_string* y *double_quoted_constant_string* los cuales fueron corregidas ya que solamente aparecía la frase "any char". Para esto se agregó T_ANY_CHAR, la cual reconoce todos los caracteres del estándar ASCII.
- En la definición de DNUM se arreglaron dos errores: el primer digit se definió para que se repita 0 o más veces, ya que, en PHP está permitido definir un real como .256, así como 125.. También el . (punto) estaba entre corchetes, lo cual significa que puede aparecer 0 o 1 vez, pero si se está definiendo un real, el punto siempre debe aparecer.
- Otro errores menos significativos.

La definición de esta gramática, a mi parecer, es muy pobre. Según esta, no se reconoce como válidas declaraciones en las que aparezcan operadores lógicos (*OR*, *AND*, *XOR*, étc.), ni potenciación (*num**exp*), ni asociación de operadores (*(1+2)*3*), ni la simple multiplicación. Tampoco, reconoce el condicional heredado de C: *condición* **?** *si es verdadera* **:** *si no es verdadera*.

Utilización del analizador

Al analizador se le pueden pasar las cadenas a ser reconocidas de dos formas distintas: a través de un archivo e ingresándolas por línea de comando.

Como argumento por línea de comandos

El nombre del archivo que se quiera analizar se puede pasar como argumento de línea de comando en el momento que se llama al analizador:

analizadorP4.exe nombre_archivo.txt

En cmd.exe, siendo *nombre_archivo.txt* el archivo a analizar

Por ejemplo: analizadorP4.exe test1.txt (text1.txt se encuentra en la carpeta principal del proyecto).

Dentro del programa

También, se puede ejecutar desde consola sin argumentos. Al ejecutarse de esta forma, el programa mostrará un menú con opciones. Para usar algunas de las funciones descriptas por las opciones se debe ingresar el número de opción. Por este método, también existe la opción de abrir un archivo.

La pantalla principal del programa, con su menú, se ve así:

```
### Analizador lexico y sintactico ###
--- PHP: declare_list ---

# 1) Ver especificacion del proyecto
```

```
# 2) Ingresar texto
# 3) Abrir archivo de texto
# 4) Salir

Opcion>
```

Una vez que se hayan pasado las cadenas a analizar, el programa mostrará por pantalla la tabla de símbolos primero y luego el análisis sintáctico, el cual muestra las respectivas derivaciones de los no terminales (símbolos en minúscula encerrados por < y >). Si se encuentran errores en el análisis solo se imprime la tabla de símbolos y los errores encontrados. Los errores se detallan más adelante.

Ingresar cadenas por línea de comandos

Para ingresar las cadenas por línea de comandos (opción 2 del menú), estas deben ser ingresadas en una sola línea. Una vez que se haya ingresado el texto a analizar se debe presionar ENTER, y a partir de aquí el programa comenzará el análisis para luego mostrarlo por pantalla. Por ejemplo:

w=5, s=array(9,r=>45), asd= saludo::hola, w=8e9 … [ENTER]
(Aquí comienza el análisis léxico y sintáctico)

Ingresar archivo de texto para analizar

También, dichas declaraciones se pueden realizar en múltiples líneas, debido a que PHP es un lenguaje muy flexible que no considera lo saltos de línea como tokens, sino que los borra, al igual que los espacios en blanco. Para ingresar múltiples línea se debe recurrir a analizar un archivo de texto (opción 3 del menú).

Para esto se crea un archivo .txt con cualquier editor y se ingresa el texto a analizar. Por ejemplo:

[analizar_esto.txt]
w=8,
arr=array(j => 5,
r=>98
,
www=65)

Una vez creado el archivo, se ejecuta *analizadorP4.exe*, se elige la opción 3 del menú y luego se ingrea el nombre del archivo. El analizador comenzará a leer el contenido del archivo para analizarlo.

Salida del analizador

Luego de que el usuario ingrese el texto que desee analizar para ver si es reconocido por la gramática, el programa comienza el análisis para luego, al instante, mostrar por pantalla dicho análisis.

Salida por consola

Luego de realizado el análisis se puede ver en pantalla la tabla de símbolos (tokens con sus respectivos valores) y las derivaciones del árbol sintáctico.

Por ejemplo, para la cadena: "*w=8, e=array(7e8)*", se reicibiría como salida:

```
Analsis lexico: tabla de simbolos
[T_STRING]: w
[T_CHAR]: =
[T_LNUMBER]: 8
[T_CHAR]: ,
[WHITESPACE]:
[T_STRING]: e
[T_CHAR]: =
[T_ARRAY]: array
[T_CHAR]: (
[T_LNUMBER]: 7
[T_CHAR]: )

Analisis sintactico: derivacions
<declare_list>
{ <declare_list> , T_STRING = <static_scalar> }
{ T_STRING = <static_scalar> } , T_STRING = { T_ARRAY ( <static_array_pair_list> ) }
T_STRING = { <common_scalar> } , T_STRING = T_ARRAY ( { <static_array_pair> } )
T_STRING = { T_LNUMBER } , T_STRING = T_ARRAY ( { <static_scalar> } )
T_STRING = T_LNUMBER , T_STRING = T_ARRAY ( { <common_scalar> } )
T_STRING = T_LNUMBER , T_STRING = T_ARRAY ( { T_DNUMBER } )
T_STRING = T_LNUMBER , T_STRING = T_ARRAY ( T_DNUMBER )
```

Nota importante

Como se puede apreciar, los no terminales están en minúscula y encerrados entre < y >. El texto que podemos ver encerrado entre corchetes { y } son el conjunto de terminales y no terminales que derivan del no terminal del nivel anterior. Si tenemos dos terminales, por ejemplo, el primer conjunto encerrado entre { } es el derivado del primer no terminal que aparece en el nivel anterior, y el segundo conjunto encerrado entre { } es el derivado del segundo no terminal que aparece en el nivel anterior.

Salida en el archivo analisis.html

Debido a las limitaciones de la consola de Windows (cmd.exe) para darle estilo al texto, se agregaron funciones al analizador para que su salida sea en un archivo HTML: *analisis.html*

Dicho archivo HTML está vinculado a un archivo *style.css* que se encuenrra en la carpeta src/. Este archivo le da estilo a la maquetación del HTML para que una mejor visualización.

Una vez terminado el análisis del programa, el archivo analisis.html se crea automáticamente en la carpeta principal del proyecto. Por lo tanto, se puede abrir directamente con el navegador y así se tendrá una mejor visualización del análisis realizado.

La tabla de símbolos se muestra con las etiquetas ***<table>*** de HTML. Y el análisis sintáctico como una lista ***<ol>*** numerado de HTML. A su vez, lo no terminales están encerrados por la etiqueta <b> y todo el texto derivado del no terminal anterior por la etiqueta <i>, las cuales le dan estilo de color a la salida.

Los no terminales, en el análisis sintáctico, están subrayados en una línea roja. Y las derivaciones de los no terminales de niveles anteriores están resaltadas con color verde.

Nota

El archivo analisis.html no tiene saltos de líneas, por lo que si se abre con un editor de texto no se podrá visualizar correctamente ya que todo el texto y etiquetas HTML estan en una sola línea debido a que es más fácil crearlo así desde analizadorP4.exe y el navegador lo entiende de igual forma que si tuviera dichos saltos.

Para poder visualizar un archivo HTML de ejemplo correctamente, se creó en la carpeta src/ el archivo base.html. Este es la base desde la que parte la creación desde el programa analizador.

Funciones principales declaradas por el usuario

Para imprimir la tabla de símbolos, se creó la función *printSymbolTable()*, la cual recorre un arreglo que contiene los tokens y sus respectivos valores y los va imprimiendo.

Para imprimir el desarrollo del análisis sintáctico, derivando los símbolos no terminales que van apareciendo, se desarrollaron dos funciones. A medida que se va analizando el texto ingresado, el analizador sintáctico (generado por bison: declare_list.y) va guardando en un arreglo que contiene una estructura, los niveles que va reconociendo y los niveles (conjunto de terminales y no terminales) a los que hacen referencia los no terminales reconocidos. Además, si se reconoce algún error es agregado en un arreglo que contiene todos los errores encontrado durante el análisis, para luego imprimirlos.

Las funciones:

- *addT(“T_TOKEN”)*: agrega un terminal al arreglo token.
- *addNT(“<no-terminal>”)*: agrega un no terminal al arreglo token.
- *addE(“Error”)*: agrega un mensaje de error al arreglo error.
- *addLevel()*: aumenta en 1 los niveles reconocidos hasta el momento, o sea, genera un nuevo nivel que contendrá un conjunto de terminales y no terminales derivados de un no terminal de algún nivel anterior.

Una vez terminado el análisis sintáctico realizado por Bison, se llama a *makeTree()*, función que permite armar el árbol sintáctico y luego de esto a *printTree()*, la cual a su vez imprime todos los niveles del árbol gracias a la función *printTreeLevel()*.

Las funciones:

- *makeTree*(): es la función más compleja que realicé. Esta se encarga de vincular los no terminales encontrados con el nivel al que hacen referencia. Entonces, una vez que se quiera imprimir el análisis sintáctico, los no terminales encontrados irán siendo reemplazados por el conjunto de terminales y no terminales a los que hagan referencia. En la estructura, la variable level_ref es la que se encarga de almacenar los niveles derivados del no terminal.
- *printTree*(): imprime el arbol sintáctico, o más bien, las derivaciones a partir de <declare_list> hasta llegar a la cadena ingresada, si dicha cadena no se reconoce se imprimirán los errores.

La función más importante es *makeTree()* debido a que, para derivar desde el primer nivel de no terminales (declare_list) hacia abajo, es necesario hacer uso de un arreglo que almacene las estructuras reconocidas por Bison, y luego las imprima en forma inversa.

Para el árbol sintáctico se usarion dos estructuras, *s_treeLevel* y *s_token*. La estructura *s_token* tiene una variable llamada *level_ref*, como se mencionó anteriormente, usada en caso de que este token sea un no terminal, para referenciar el nivel donde se encuentra la cadena que deriva de dicho no terminal. Y así, cuando se imprima el árbol, se reemplazará la cadena derivada del no terminal en lugar de este.

PrintSymbolsTable() es una simple función que imprime por consola los tokens y sus respectivos valores almacenados en los arreglo *stToken* y *stValue*.

Nota

Los terminales son los tokens reconocidos por el analizador léxico (generado por flex). O sea, los símbos que empiezan por T_..., genéricamente T_TOKEN.

Estos tokens representan el texto ingresado por el usuario, la cual se le asigno dicho token en el análisis léxico.

Funciones para generar el archivo analisis.html

Las mismas funciones usadas para imprimir por consola fueron redefinidas para imprimir a un archivo. Principalmente, lo que se cambió en las funciones fue la fución printf() por fprintf() para poder dirigir la salida a un archivo de texto. La funcion fprintf() recibe como primer argumento un puntero a FILE (FILE *htmlFile) en el que almacenará el texto ingresado como segundo argumento.

A estas funciones se les pasa como argumento un puntero a FILE que apunta al archivo “analisis.html”, donde se guardará la salida.

fprintf(), además, imprimirá la salida con las etiquetas HTML correspondientes para que le den estilo y maquetado, y a la hora de abrirlo con el navegador se pueda observar una estructura visualmente fácil de entender.

Las funciones para generar el archivo, tienen el mismo nombre que las usadas para la salida por consola con “ToHTML” al final.

Reconocimiento de errores

El analizador reconoce errores por extensión, es decir, los errores más frecuentes que pueden aparecer han sido agregados en las declaraciones del archivo bison (declare_list.y) como parte de la gramática. Así, en caso de que se produzca algunos de esos errores, poder agregarlos a un arreglo que contiene todos los errores que han ido apareciendo para luego mostrarlos. De esta forma, se le pueden dar más detalles al usuario de cuáles son los errores que cometió a la hora de programar.

En caso de que ocurran errores que no han sido especificados como parte de la gramática, se mostrará un mensaje que dice “Error desconocido: syntax error”.

Si en el análisis léxico se produce algún error, principalmente que no se reconozca algún simbolo, y este no se pueda asociar a ningun token, también se mostrará un error que dice: “Error léxico. Símbolo no reconocido: ...”.

Observaciones y conclusiones

Lo que más tiempo me llevó es lograr definir las funciones necesarias para lograr que se imprima el “árbol” sintáctico, con sus derivaciones, ya que Bison genera un analizador de abajo hacia arriba y de derecha a izquierda, por lo que debí invertir el orden de impresión. MakeTree() fue la función más compleja que logré desarrollar.

Es un proyecto muy interesante. A pesar de usar GNU/Linux nunca había utilizado estas herramientas (Flex y Bison), ni las conocía. He descubierto que son muy importantes para facilitar el trabajo a las personas que se encargan de desarrollar nuevos lenguajes de programación.

Pero no solamente para generar un analizador/compilador independiente o un intérprete independiente. También son muy importantes a la hora de realizar un intérprete embebido, o sea, un intérprete dependiente que se encuentre dentro de otro programa. Esto seriviría, por ejemplo, para agregarlo funcionalidades al programa mediantes su propio lenguaje de scripts.

Por ejemplo, últimamente estuve analizando Game Maker Studio, una herramienta para realizar juegos sencillos, principalmente en 2D. Esta herramienta posee su propio lenguaje de script para darle lógica al juego: Game Make Languaje (GML). Antes de conocer estas herramientas (Flex y Bisón), pensaba que crear un nuevo lenguaje y un intérprete para el mismo me parecía una pérdida de tiempo debido a que existen lenguajes como Python, Javascript, Lua (creado específicamente para incurstarlo en otros programas y así extenderlos) que permiten embeber sus intérpretes dentro de otros programas para dotarlos de nuevas funcionalidades. Pero luego

de conocer estas dos herramientas para el análisis léxico y sintáctico (y semántica) comprendí que crear un intérprete o compilador para un nuevo lenguaje no resulta tan difícil.

Como última observación, la gramática EBNF proporcionada para realizar este analizador no me pareció muy completa.