Universidad de Alcalá

Escuela Politécnica Superior

Universidad de Alcalá

Procesadores del lenguaje

PECL3

Miércoles 12:00 – 14:00

Grado en Ingeniería Informática – Curso 2019/2020

Marcos Barranquero Fernández – 51129104N

Eduardo Graván Serrano – 03212337L

Josué Sánchez Laguna – 09071165B

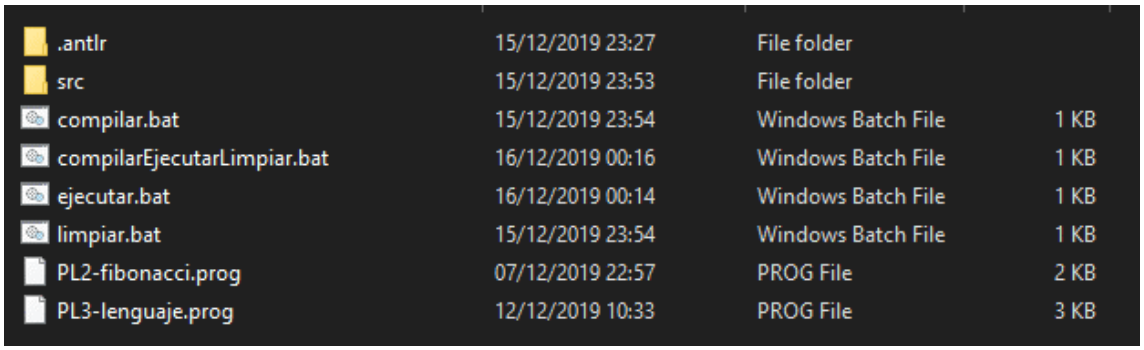
CONSIDERACIONES PREVIAS

Se parte de que se encuentran instalados ANTLR y Graphviz, y añadidos sus carpetas *bin* a la variable de entorno Path de Windows.

Si no es así, la ejecución de los comandos *dot* del Graphviz desde el programa Java fallarán.

Tener en cuenta también que se ha utilizado la consola PowerShell a la hora de llamar a los comandos dot, por lo que la ejecución fallará en sistemas que no cuenten con esta Shell.

La estructura del proyecto es la siguiente:



Dentro de la carpeta src se encuentra todo el código fuente del programa.

Se entregan a parte 4 ficheros .bat para facilitar la compilación y ejecución del programa. Los archivos Java compilados con "compilar.bat" se van a generar en la misma carpeta en la que se encuentran los .bats, por lo que puede ser muy caótico.

Para ello, se ha creado "ejecutar.bat" que, una vez se haya compilado, hace la llamada para ejecutar el programa pasándole por defecto el fichero "PL3-lenguaje.prog". Si se quisiese llamar a otro archivo, se puede hacer la llamada manualmente una vez se ha llamado al .bat de compilar, o simplemente modificar el .bat para que pase como parámetro al archivo que queramos. El resultado se genera en esta misma carpeta.

"limpiar.bat" borra todos los ficheros que no formen parte del resultado, es decir, borra todos los .class, los .dot, etcétera, por lo que se pueden ejecutar los 3 bats seguidos si lo único que se quiere es ver el resultado.

El último bat, "compilarEjecutarLimpiar.bat" hace exactamente eso, llama a los otros 3 bats en orden y deja la carpeta lo más limpia posible para que se pueda comprobar el resultado fácilmente.

Si únicamente se quiere comprobar el resultado para una ejecución en concreto, se recomienda el uso de este último .bat.

INTRODUCCIÓN

En esta práctica se hacen modificaciones sobre la práctica anterior para añadir funcionalidad relacionada con métrica del lenguaje.

BREVE RESUMEN

El proyecto entregado se podría dividir en 4 grandes partes:

- Gramática del lenguaje y Listener
- Funcionalidad del HTML, Tabla de símbolos con sus clases base y gestor de puntuación
- Creación y gestión de grafos y complejidad ciclomática
- El main para generar el output solicitado

Se pasa a describir brevemente cada una de estas partes.

GRAMÁTICA

La gramática ha sido adaptada de la PECL2 para aceptar el nuevo ejemplo de lenguaje propuesto. Algunos de los cambios más relevantes son:

- Introducción de arrays
- Introducción de bucles for step

El listener tiene asociados un **lexer** y un **parser** para extraer vocabulario de reglas y símbolos entre otras cosas. Tiene también asociada una **tabla de símbolos** utilizada para escribir la puntuación de complejidad, datos sobre la función actual, etc.

Respecto al listener, en prácticas anteriores se había construido el árbol con enterNombreFunc y exitNombreFunc. Ahora se construye el árbol con enterEveryRule.

Para rellenar la **tabla de símbolos**, se sobrescriben métodos como defFunction para capturar nombres de función, argumentos, etc.

Para la captura de puntuación de **complejidad**, se sobrescriben métodos relacionados con las operaciones básicas, con llamadas a funciones, asignaciones, etc.

Para crear los **grafos**, se sobrescriben los métodos relacionados con **bifurcaciones**, es decir, enterIf, enterWhile, exitFor, etc. Se tiene también una pila de else ya que, en nuestra gramática, el else está contenido dentro del if, y como es opcional, debemos saber si para un if existe un else o no a la hora de crear el grafo de la complejidad ciclomática.

FUNCIONALIDAD VARIA

TABLA DE SÍMBOLOS

La tabla de símbolos es una estructura que se va rellenando conforme el listener lee el archivo de lenguaje, y va capturando los elementos necesarios para luego realizar las operaciones.

Contiene un arraylist de funciones y otro de variables globales. Se tiene una variable que denota la función actualmente leída, para asignar los puntos de complejidad, líneas efectivas, declaraciones de variables y demás sobre la función adecuada.

Además, lleva la cuenta de la complejidad total, líneas efectivas totales, el gestor de puntuación y el flujo de llamadas.

Es necesario denotar que cada función contiene las variables declarad, funciones llamadas, parámetros y complejidad propias según su contenido. Cada función también tiene un gestor de grafos para su grafo asociado.

El listener rellena estos datos accediendo a través de la **función actual de la tabla de símbolos**.

GESTOR DE PUNTUACIÓN

El **gestor de puntuación** tiene una **pila** de **BigInteger** para llevar la cuenta de los puntos. Cuando se lee un contenido del archivo que incrementa los puntos, se sobrescriben los puntos del último elemento de la pila.

Cuando se entra en un bucle, se añade un nuevo elemento en la pila donde se escriben los puntos leídos dentro del bucle.

Cuando se sale de un bucle, se extrae el ultimo elemento de la pila, se elevan los puntos al cuadrado, y se suman a los puntos en el siguiente elemento de la pila.

ESCRIBEHTML

Respecto a la clase ocupada de **escribir el informe HTML**, cabe reseñar que el constructor crea todos los encabezados del informe, y que existe un método para añadir funciones y que escriba todos los datos solicitados de esa función.

Contamos con un método dedicado a realizar el resumen global del programa, leyendo desde el método principal que consideramos que es el *main*, aunque esto se puede cambiar desde el método de escribir el resumen.

Finalmente tenemos también un método para finalizar el archivo html.

GESTIÓN DE GRAFOS

Con relación a los grafos, se debe hablar de la creación de grafos y nodos, y de como los creamos para luego calcular las complejidades y llamadas a funciones.

GESTORGRAFO, GRAFOS Y NODOS

La clase Grafo contiene un nombre y un ArrayList de nodos. Un nodo tiene id y un arraylist de sus nodos hijos. El grafo permite añadir y unir nodos, y crear un archivo *dot* a partir del grafo.

La clase **gestorGrafo** asocia cada función a un grafo, y contiene métodos para escribir en el grafo según se entre a un if, else o bucle. También admite **recursividad**, distingue funciones llamadas mediante colores y un método crearDibujo que genera el archivo *dot* y lo transforma a *svg*.

El gestor tiene un arraylist que almacena las operaciones a las que se ha entrado en orden, permitiendo luego reconstruir el grafo leyendo este arraylist.

El gestor de grafos también tiene contadores de nodos, aristas y salidas, y los emplea para calcular la complejidad una vez se ha cerrado el grafo.

RECONSTRUCTORES

Se tienen dos clases que reconstruyen grafos para calcular la complejidad ciclomática y las llamadas a funciones.

Ambos funcionan de forma similar: tienen una función dada de la que construir su grafo y una tabla de símbolos.

El reconstructor irá construyendo el grafo de la función dada haciendo uso del array de comandos almacenados dentro del gestor de la función. Si encuentra una llamada a una función, la buscará en la tabla de símbolos y construirá las llamadas o la complejidad de esa función hasta que finalice y vuelva a seguir construyendo la función padre.

Aquí se detecta la **recursividad**: si vemos que una función llama a otra que vuelve a llamar a esa, lo anotamos en el grafo como recursividad en forma de bucle y detenemos la recursión infinita a la hora de dibujar el grafo.

El reconstructor ciclomático considera las bifurcaciones y bucles y construye el grafo ciclomático total, mientras que reconstructor de funciones considera las llamadas a funciones y construye un grafo de estas llamadas.

MAIN

La llamada al programa se hace con el siguiente comando:

```
java Main fichero.prog nombrefuncionprincipal nombreficheroinforme.html
```

De los cuales:

- **Main** es el nombre del fichero Java donde se encuentra el método main, que inicia la ejecución del programa.
- **Fichero.prog** es el fichero de código que se le pasa al programa para analizar. Al contrario que en la PECL2, esta vez es necesario introducir un fichero .prog.
- **NombreFuncionPrincipal** es el nombre de la función principal del programa, es decir, la función desde la cual se va a estudiar el flujo de llamadas que haga el programa.
- **NombreFicheroInforme.html** es el nombre que el programa le dará al fichero de salida del programa, en el cual se encuentra toda la información referente al fichero pasado por argumento.

Si no se pasa el número de argumentos correctos al hacer la llamada, el programa lanzará un mensaje de error y parará su ejecución.

En caso contrario, se recogen los argumentos y se almacenan en variables para ser utilizados más adelante.

La ejecución continua con las acciones propias de la PECL2; generar el lexer y sacar los tokens, crear el parser y el árbol sintáctico, crear el Listener y recorrer el árbol sintáctico asociado a ese Listener con el uso de un Walker. Una vez hecho todo esto, se recupera el árbol generado por el Walker como en la práctica 2, y además se recupera la tabla de símbolos que se ha ido rellenando.

El programa llama a writeToTxt igual que en la PECL2 y después llama a una función auxiliar en la cual se hacen todas las llamadas nuevas de la PECL3.

Esta función auxiliar se ha llamado **funcionalidadPECL3** y toma como parámetros todos los datos que necesita para realizar el informe html.

Desde esta función se hacen las llamadas para inicializar el archivo html, rellenar la información sobre cada función en concreto, rellenar la información para el resumen del programa, y finalizar el archivo html.

Si todo funciona correctamente, se imprimirán por consola dos mensajes que indican que se han conseguido generar los dos ficheros resultado: el .txt con el árbol, y el .html con el estudio del fichero de código.