**Teoría de Lenguajes y Automatas**

Trabajo Practico Especial: Analizador Sintáctico Descendente

Alumnos:

Federico Di Nucci, legajo 50120

Agustín Marseillan, legajo 50134

Nicolás Loreti, legajo 49479

Índice

Introducción y Objetivo 3

Consideraciones Realizadas 4

Desarrollo del Trabajo Práctico 5

Problemas durante el Desarrollo 6

Posibles Extensiones 7

Introducción y Objetivo

El trabajo consiste en escribir un programa el cual dado una gramatica libre de contexto, genere el analizador sintactico descendente para dicha gramatica en lenguaje C, el mismo debera recibir una cadena como argumento y decidir si la misma es generada con la gramatica usada para crear el analizador y ademas dar las producciones utilizadas para generar dicha cadena.

Consideraciones Realizadas

**Gramaticas:** Suponemos que los simbolos no terminales de la gramatica son letras mayusculas, y que los terminales son minusculas. Tambien asumimos que todos los simbolos que se encuentran en las producciones son simbolos validos, es decir que fueron declarados como terminales o no terminales.

**Espacios Vacíos:** Ya que en ningun momento se especifica como tratar los espacios vacios dentro de las gramáticas decidimos representarlos pediantes un 0. Es decir que la producción A->c quedaría representada por A->c0 dentro del programa.

**Archivos de Salida:** No se realiza un nuevo archivo por cada ejecución sino que la salida se da siempre en los mismos archivos que se van sobreescribiendo para dar lugar a los nuevos. Esto hace que si el usuario quiere ir guardando distintas salidas tenga que ir copiando los resultados a una carpeta personal para que estos no se pisen con la próxima ejecución del conversor.

Descripción del desarrollo del TP

El desarrollo, al igual que en el primer trabajo practico especial, estuvo separado basicamente en dos etapas. La primera correspondió a el parseo utilizando Lex , para esto se pudo utilizar el parser del trabajo anterior con algunas modificaciones, ya que ahora se debera aceptar que del lado derecho de una gramatica pueda haber mas de dos simbolos. Al haber parseado la gramatica y el automata en diferentes archivos Lex en el trabajo anterior, se pudo adaptar muy facilmente a este. Gracias a esto se pudo reutilizar las mismas estructuras tambien.

La segunda etapa consistió en la implementación del programa el cual sera el encargado de escribir el analizador sintactico, para ellos se utilizo los algoritmos vistos en la teorica. Se creo un procedimiento por cada simbolo no terminal los cuales brindaban sus producciones a la funcion procesar que se encarga del rechazo o aceptacion de la produccion , una funcion main que llamaba al procedimiento del simbolo inicial y luego se verificaba si habia salido con error o si no se habia consumido toda la cadena, en caso que no fuese asi, la cadena seria aceptada.

Dificultades Encontradas

Dentro de lo que fue el desarrollo en C la mayor dificultad estuvo en pasar los algorítmos vistos en clase y hechos en papel a algoritmos computadorizados. Si bien los algoritmos son sencillos y facilmente implementables en la realidad, a la hora de usar C y manejar punteros hubo que tener mucho cuidado con los valores que uno coloca en las producciones mientras las iba a modificando para poder obtener la salida esperada.

Un gran desafio con el cual nos enfrentamos fue la implementacion de un programa el cual a su vez escriba otros programas. Hubo que pensar bien que cosas habia que hacer. La deteccion de errores fue mas complicada, ya que habia que verificar dentro de que archivo se encontraba el error.

Otra barrera con la cual nos chocamos fueron las transiciones lambda, al principio fue un gran dolor de cabeza tratar con las mismas, ya que no sabiamos como manejarlas.

De habérsenos dado la opción del lenguaje hubiésemos preferido usar Java ya que mediante el manejo de objetos y clases, getters y setters todo el desarrollo hubiese sido más ameno y se hubiese realizado en menor cantidad de tiempo.

Futuras Extensiones

Como futura implementacion, se podria considerar la opcion de eliminar repetidos, recursividad a izquierda, transcisiones Lambda para asi optimizar el algoritmo. Tambien se podría ,utilizando mismo parser y estructuras, implementar un analizador sintactico ascendente.