**Compilador**

**Definición general del Proyecto:**

Idea general: El Proyecto debe ser capaz de crear grafos de expresiones regulares para que el lexer sepa que aceptar y que no, el Proyecto debe tirar como salida una linkedlist con tokens válidos o mostrar un reporte de errores.

Objetivos: El objetivo principal es que la persona pueda definir su lenguaje de programación desde el input y luego pueda probarlo.

Usuarios: Esto está dirigido a programadores.

**Requerimientos del software**

Requisitos generales:

* El proyecto debe ser capaz de aceptar cualquier expresión regular
* El proyecto debe de ser capaz de mostrar errores
* El proyecto debe de ser capaz de tokenizar incluso si se encuentran errores y dar un reporte detallado de ambos
* El proyecto a su vez debe ser manejable por líneas de comando
* EL proyecto debe de ser capaz de realizar el análisis sintáctico correctamente
* El proyecto debe realizar el análisis semántico correctamente y mostrar los errores debido.

Requisitos funcionales

* El proyecto es capaz de aceptar cualquier expresión regular.
* El proyecto crea grafos a partir de las expresiones regulares
* Se puede escribir líneas de programación en un txt y el programa genera errores o éxitos.
* El programa muestra la información de una manera ordenada
* El programa muestra la información de errores y brinda un reporte de ello
* El programa es manejable desde la terminal por comandos.
* El programa sea capaz de procesar la gramática y crear su respectivo grafo de estados con la misma
* El programa es capaz de crear un AST correctamente a partir del input del usuario
* El programa es capaz de mostrar que hay error si en dado caso el usuario se equivoca

Información de autoría

* El proyecto es 100% original, se usaron algunos recursos para definir ciertas partes del código, pero como tal el código fue diseñado por nosotros.

**Especificaciones de procedimiento**

* Procedimiento de desarrollo: Para este proyecto utilizamos visual studio code para poder programar nuestro lexer a su vez utilizamos herramientas como las librerías dentro de visual studio code y utilizamos stackoverflow como referencia a dudas que surgían en el camino.
* Planificación: La forma como planificamos este proyecto fue que íbamos a ir trabajando por iteraciones entonces nos guiamos por la guía de miu y la primera semana fue dedicada a investigación y desarrollo de los nfa, la segunda semana fue dedica al desarrollo de los nfa a dfa y exportación y la última semana fue dedicada al lexer como tal y documentación.

**Procedimiento de instalación y prueba**

* Obtención e instalación: Para instalar nuestro software debes ir al siguiente link: <https://github.com/pjvl99/compilador>, una vez ahí le das en el botón verde de “code” y lo descargas como zip, para instalarlo se hace desde la terminal, se compila el principal “Compiler.cpp” con: g++ Compiler.cpp -o, esto creara un ejecutable, una vez hecho esto con los comandos –target, -debug puedes compilar lo demás, si en dado caso tuvieras duda con los comandos solo pon: “./a.exe” y esto te mostrara la descripción de los comandos, recuerda siempre estar ubicando en la carpeta “Compiler” del proyecto para que te funcione, en la misma encontraras los .txt, el de regex sirve para definir las expresiones regulares, en la primera línea defines el nombre de esa expresión y en la segunda línea la expresión regular como tal, se vuelve a repetir con la 3ra, 4ta etc. No olvides poner “END” para que el programa sepa hasta donde leer, luego el otro .txt editable es el de “código” ahí puedes escribir tu código para probar.
* A su vez también dentro de la carpeta de parser en su carpeta de data podras encontrar el txt de Estructura donde ahí podrás definir la gramática de tu lenguaje de programación, todo de la mano con los tokens que ya creaste, a su vez hay una carpeta llamada src donde encontrarás un html que te mostrará el árbol de una forma más ordenada y dinámica
* Especificaciones de prueba y ejecución: Para correr el programa basta con tener instalado algún compilador de C/C++ o usar directamente visual studio code con la librería C/C++ (recomendado), no olvides que puedes correr directamente desde visual studio code o desde la terminal, solo debes tener instalado g++.

**Arquitectura del sistema**

* Descripción jerárquica: El programa esta ordenado por una clase padre llamada “Compiler” que controla las demás, en este caso debajo de esta están Scanner y Lexer y Parser.
* Diagrama de módulos:

Compiler

Input regex.txtzx

Tokens (Linkedlist)

Lexer

Grafos.txt

Scanner

Input Codigo.txt

Input Estructura.txt

AST

Semantic

**Descripción de módulos**

* Compiler: Este módulo tiene por objetivo manejar los demás módulos dentro del compilador, este módulo depende del input que el usuario ingrese en la terminal, este se encuentra en la carpeta de Compiler, a su vez todos los archivos en cada carpeta se encuentran implementados en el módulo.
* Input regex: Este módulo simplemente sirve de input, controlado por la librería “iostream” de C/C++, aquí el usuario define las expresiones regulares que el scanner pasara a interpretar.
* Grafos: Este archivo es el output de Scanner, aca están los grafos codificados para el lexer.
* Lexer: Acá se recibe por input los grafos y el código.txt donde simplemente en base a los grafos definidos así tokenizara o tirara error.
* Código: En este archivo el usuario simplemente escribe lo que quiere programar.
* Scanner: Aca simplemente se interpretan las expresiones regulares y se exporta a un .txt los grafos creados.
* Tokens: En el último modulo simplemente se muestran errores y tokens.
* Input Estructura: Archivo donde el usuario define la gramática de su lenguaje de programación
* AST: El árbol donde el usuario podrá visualizar su código jerarquizado.
* Semantic: Modulo donde simplemente se recorre el AST en busca de errores semánticos.

**Funciones:**

* Compiler:
  + Main(): Se controla todo el compilador
* Scanner:
  + Scanner(): Corazon del programa, aquí se controla la creación y exportación de grafos
  + Casos(): Se analiza carácter por carácter del regex para discriminar y crear nfa
  + Parentesis(): Se analizan paréntesis para anticiparse a eventualidades
  + Asterisco(): Hace la función de \* o ? según sea necesario.
  + Transform(): Recibe el nfa y lo transforma a dfa y exporta.
* Lexer:
  + Lexer(): Controla el tema de tokenizacion y discriminación de input.
  + Crear(): Recibe los grafos del scanner y los guarda en variables.
  + Punteros(): Con los punteros se verifica información en base al input de código para mostrar errores o tokenizar.
* Parser:
  + Parser(): Controla el tema de la creación del grafo de estados, gramática y tokens.
  + Check(): Hash table para no terminals.
  + Search(): Donde se buscan los simbolos de apertura en los tokens.
  + Search2(): Donde se buscan los símbolos de cierre en los tokens.
  + Sum(): Aplica la función de + o \* a los tokens que se vean afectados por la misma.
  + Braces(): Aplica la función de las llaves que es la de agrupar o insertar lo agrupado.
  + Simbology(): Una función para insertar los datos de decaf y poder trabajar con la simbología con un switch case para poder afectar la gramática por lo mismo.
  + ToDo(): Función donde simplemente vamos separando los símbolos de los tokens en la gramática.
  + WordKey(): Hash table para los tokens de la linkedlist.
  + Read(): Esta función es para ir creando el AST en base al input del usuario
  + Action(): Función recursiva donde se va comparando el input del usuario con el grafo de estados y en base a lo mismo se hace reduce, shift o goto, accept o error.
  + Printtree(): Función donde simplemente mostramos el árbol en la terminal y en el html para tener una mejor vista.
* Semantic:
  + Semantic(): Funcion que maneja las demás funciones y muestra errores o accepts.
  + Typecheck(): Funcion donde simplemente se revisa el tipo de variables y que las operaciones sean las debidas.
  + Mostraarbol(): Funcion donde nuevamente volvemos a mostrar el árbol solo que con las anotaciones de tipo
  + Uniqueness(): Funcion donde se revisa que las variables sean declaradas y llamadas como debe ser y que sean únicas o accesibles.
  + Posibilities(): Funcion donde revisa que las operaciones entre variables sean validas
  + Searchstack(): Funcion donde simplemente se va buscando en el stack para cada scope en el código para las variables.

**Conclusiones**

Un scanner cumple la función de traductor en este caso véase de un regex a grafos para poder traducir código de programación, esto lo hace algo potente e interesante sin embargo es algo trabajoso y requiere tiempo.

El parser es algo fundamental dentro de un compilador porque es donde tenemos la información jerarquizada y guardada ordenadamente y su aplicación para traducción de lenguajes y verificación a nivel sintáctico y semántico es algo que no puede dejarse olvidado.

El análisis semántico pese a que ya no es tan complicado como un parser o un lexer representa el ultimo verificador para que nuestro código pueda ser exportado correctamente y al menos en opinión propia también representa una pieza fundamental en un compilador o en un traductor.

Por último, se puede concluir que a partir del Lexer y el Parser la traducción de un lenguaje y verificación del mismo no es un proceso sencillo y es bastante trabajoso sin embargo es algo fundamental para poder ir aprendiendo y creciendo como ingenieros.