



INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO



“INTÉRPRETE DE CÓDIGO PYTHON A C++”

COMPILADORES.

PROFESOR: SÁNCHEZ BRITO MIGUEL.

GRUPO: 3CV13.

ALUMNO: FLORES CASTRO LUIS ANTONIO.

17 ENERO 2023.

ÍNDICE.

INTRODUCCIÓN.	3
MÉTODOS.	4
AUTÓMATAS Y EXPRESIONES REGULARES.	6
RESULTADOS.	7
Diagrama de casos de uso del Intérprete de código python a C + +.	7
Ejecución del programa.	8
VENTAJAS Y DESVENTAJAS DEL INTÉRPRETE DE PYTHON A C + +.	13
COMPILADORES EN FORMACIÓN ACADÉMICA.	13
REFERENCIAS BIBLIOGRÁFICAS.	14

INTRODUCCIÓN.

Actualmente a nivel mundial hemos visto que los desarrollos tecnológicos se han dado de una forma bastante rápida, esto debido a que estamos en constante evolución y siempre buscamos el optimizar procesos y tareas con el fin de obtener resultados con costos menores, ya sea en cuestiones de tiempo o de esfuerzo. El área de sistemas computacionales es un claro ejemplo de la actualización constante, podemos ver que al mes de que se lanzó un nuevo lenguaje de programación o herramienta del mismo ya existe una versión con diferentes funcionalidades. Dentro de la gran área de sistemas tenemos la parte de los compiladores, primero ¿Qué es un compilador?, bien podemos decir que es un programa dedicado a traducir código en un lenguaje de programación a otro lenguaje. El lenguaje fuente normalmente suele ser de alto nivel, mientras que el lenguaje objeto podría ser de bajo nivel como el ya conocido lenguaje máquina aunque podría ser otro lenguaje sin problema. ¿Por qué razón se mencionan los compiladores?, bueno podemos decir que es la parte más importante para el desarrollo de un sistema que puede ser implementado para resolver cualquier problema, como ya se había mencionado estamos en constante cambio pero un compilador difícilmente cambiará la forma en la que se trabaja, mientras que su entorno si. Dentro de esta materia de compiladores se tiene como objetivo desarrollar un intérprete con base en herramientas generadoras de analizadores léxicos y sintácticos y programación orientada a objetos. En esta ocasión se desarrollará un intérprete del lenguaje de programación python a C ++, con el fin de generar un código ya implementado, sin demorar tanto tiempo y entender el mismo en el nuevo lenguaje, aplicando todas las herramientas y diferentes fases o etapas de análisis de un compilador, de esta forma cumpliendo los objetivos generales de la materia.

MÉTODOS.

Dentro del funcionamiento de un compilador tenemos ciertas fases o componentes que ayudan al objetivo general que es el interpretar cierto código y generar otro. En la primera fase tenemos la descripción de una tabla de símbolos.

Tabla de símbolos o identificadores.

Esta tabla está dada por una estructura, como su nombre lo indica lo más correcto a utilizar es una tabla hash, sin embargo puede ser otro tipo de estructura como listas lineales, arboles de búsqueda, entre otras.. Esta tabla es utilizada para almacenar los atributos que tiene cada símbolo, dichos atributos pueden ser el tipo, valor, dirección de memoria, número de línea y el tipo de declaración de la variable. En otras palabras se encarga de todo los aspectos dependientes del contexto [1].

Esta tabla es utilizada por el analizador léxico, el sintáctico, el semántico en cuál se introduce la información, el generador de código intermedio, la fase de optimización y la de generación de código.

Analizador léxico.

También conocido como analizador morfológico es la primera fase del compilador, aquí se recibe como entrada el código fuente en un determinado lenguaje y tiene como objetivo generar una salida la cual está conformada por tokens (componentes léxicos) o símbolos, los cuales serán la entrada de la siguiente etapa. Dentro de esta fase se define un conjunto de reglas que definen el léxico, usualmente dichas reglas son expresiones regulares que proporcionan la secuencia de caracteres que definen a un token o lexema.

En algunos lenguajes de programación es necesario establecer patrones para caracteres especiales (como el espacio en blanco) que la gramática pueda reconocer sin que constituya un token en sí. Finalmente aquí se detectan errores relacionados a lo morfológico es decir a la sintaxis de un lenguaje.

Analizador sintáctico.

Básicamente un analizador sintáctico o también conocido como parser, es la fase encargada de convertir la entrada en una estructuras que normalmente suelen ser los árboles de derivación, en los cuales se almacena la jerarquía implícita de la entrada. En pocas palabras, descompone y transforma la entrada para después realizar cierto procedimiento, analiza una cadena de instrucciones en un lenguaje y la descompone en componentes individuales. Aquí los autómatas pueden ser utilizados como herramienta para la detección de errores en las cadenas de entrada [2].

Analizador semántico.

En esta fase se hace uso de la salida del analizador sintáctico y la información en la tabla de símbolos con el fin de comprobar que haya consistencia semántica del programa fuente con el conjunto de reglas definidas por el lenguaje, de la misma forma recopila información sobre el tipo de datos y los guarda en la tabla de símbolos para usarla en la generación de código intermedio [3].

También se realiza la comprobación de tipos, en donde el compilador verifica que cada operador tenga operando que coincidan. Por ejemplo, muchas definiciones de lenguajes de programación requieren que el índice de un arreglo sea entero; el compilador debe reportar un error si se utiliza un número de punto flotante para indexar el arreglo.

Generador de código.

En esta fase básicamente ya se realiza la traducción del programa fuente al lenguaje máquina u objeto haciendo uso de toda la información proporcionada por las demás fases [4]. En resumen las posiciones de memoria se seleccionan para cada una de las variables usadas por el programa, después de esto cada una de las instrucciones se traduce a una secuencia de instrucciones que ejecuta la misma tarea.

Dentro del mismo generador de código tenemos otros componentes tales como:

Administración o gestión de memoria.

En este componente se verifica la correspondencia entre los nombres del programa fuente con los objetos de datos en la memoria que se utilizan en la primera parte del compilador. La tabla de símbolos interviene aquí nuevamente debido a que conforme se va examinando las declaraciones de un procedimiento, el tipo en una declaración determina la cantidad de memoria necesaria para el nombre declarado.

Optimizador de códigos.

Se busca mejorar la eficiencia del programa en cuanto a tiempo y ocupación de memoria, esto se alcanza realizando buenas prácticas de programación y un análisis detallado en las anteriores etapas.

Recuperación de errores.

Por último, en caso de que no se traduzca bien el código debido a errores sintácticos se realiza una recuperación de errores que son almacenados normalmente en una estructura que suele ser una pila o lista de datos, todo esto con el fin de proporcionar una especie de reporte al final de la generación de código e indicar los errores que causaron que no se completara bien este proceso y que estos mismos sean corregidos.

AUTÓMATAS Y EXPRESIONES REGULARES.

Durante todas estas fases, como se mencionó una de las herramientas principales son la construcción de autómatas y expresiones regulares.

Un autómata sencillamente es un modelo que transforma una entrada de datos a una salida de fácil interpretación. Está compuesto por un alfabeto, conjunto de estados y conjunto de transiciones entre los estados. Su principio de funcionamiento está basado en empezar en un estado inicial y una cadena de caracteres pertenecientes al alfabeto y va leyendo la cadena de entrada si cumple ciertas transiciones se avanza a otro estado y de esta forma hasta consumir toda la cadena y llegar a un resultado. Su principal objetivo es reconocer ciertos patrones o lenguajes regulares [5].

Por otro lado tenemos las expresiones regulares, las cuales son una cadena de caracteres que es utilizada para describir o encontrar patrones dentro de otros strings, en base al uso de delimitadores y ciertas reglas de sintaxis. Dentro de sus ventajas está la practicidad y la gran eficiencia ya que son muy exactas y específicas en hallar patrones [6].

Ambas herramientas son muy importantes ya que si se nos facilita el diseño y entendimiento de autómatas podemos convertir y optimizar estos mediante diferentes técnicas como el teorema de arden a expresiones regulares y de esta forma tener un resultado mucho más preciso para que nuestro análisis sea más detallado.

En lo personal las expresiones regulares siento que es la esencia de un compilador ya que para empezar a armar la tabla de símbolos, nos ahorran mucho recursos tanto de memoria y tiempo de ejecución ya que buscamos lo que queremos de forma única, así teniendo mejores resultados.

RESULTADOS.

Diagrama de casos de uso del Intérprete de código python a C ++.

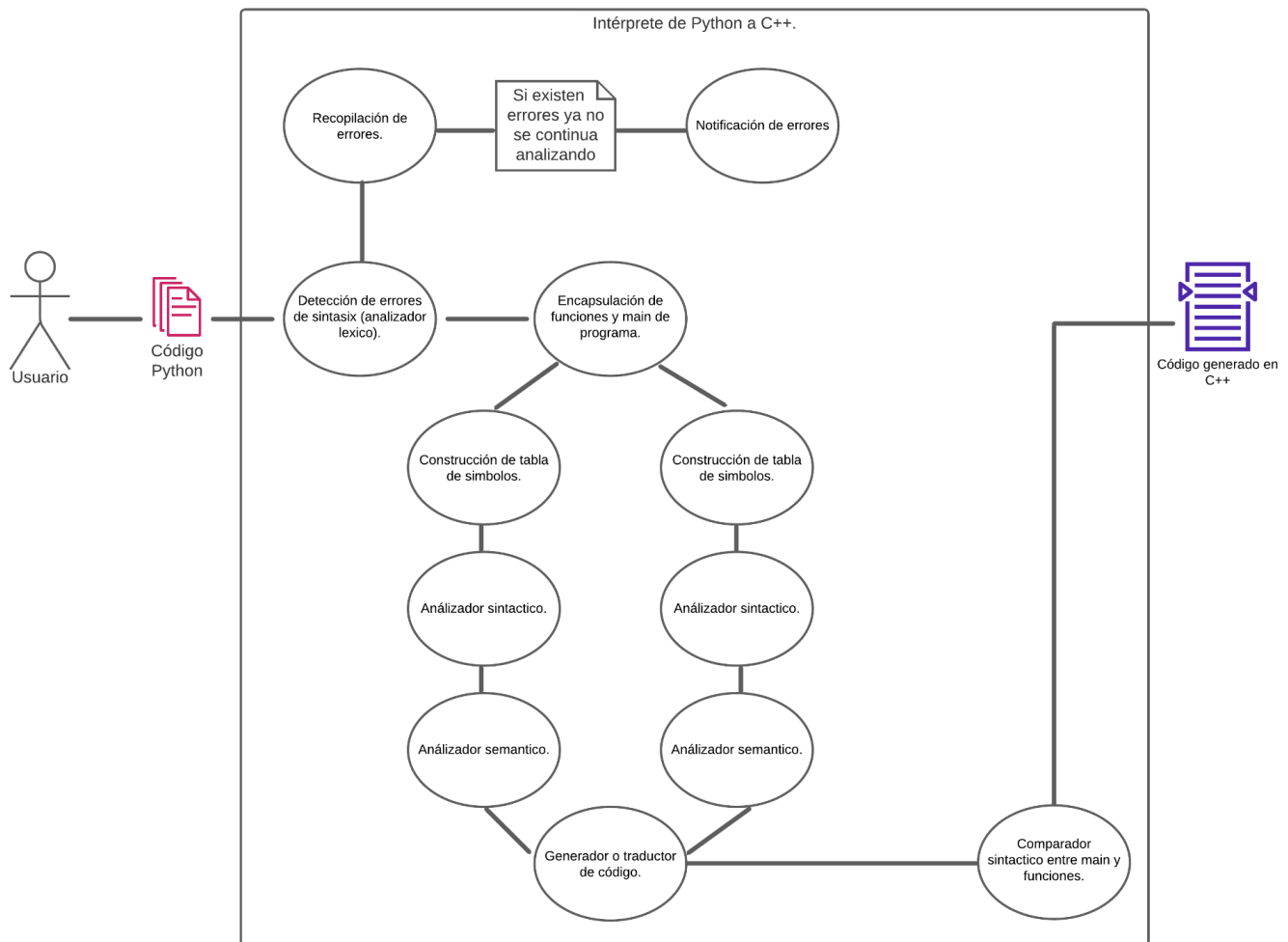
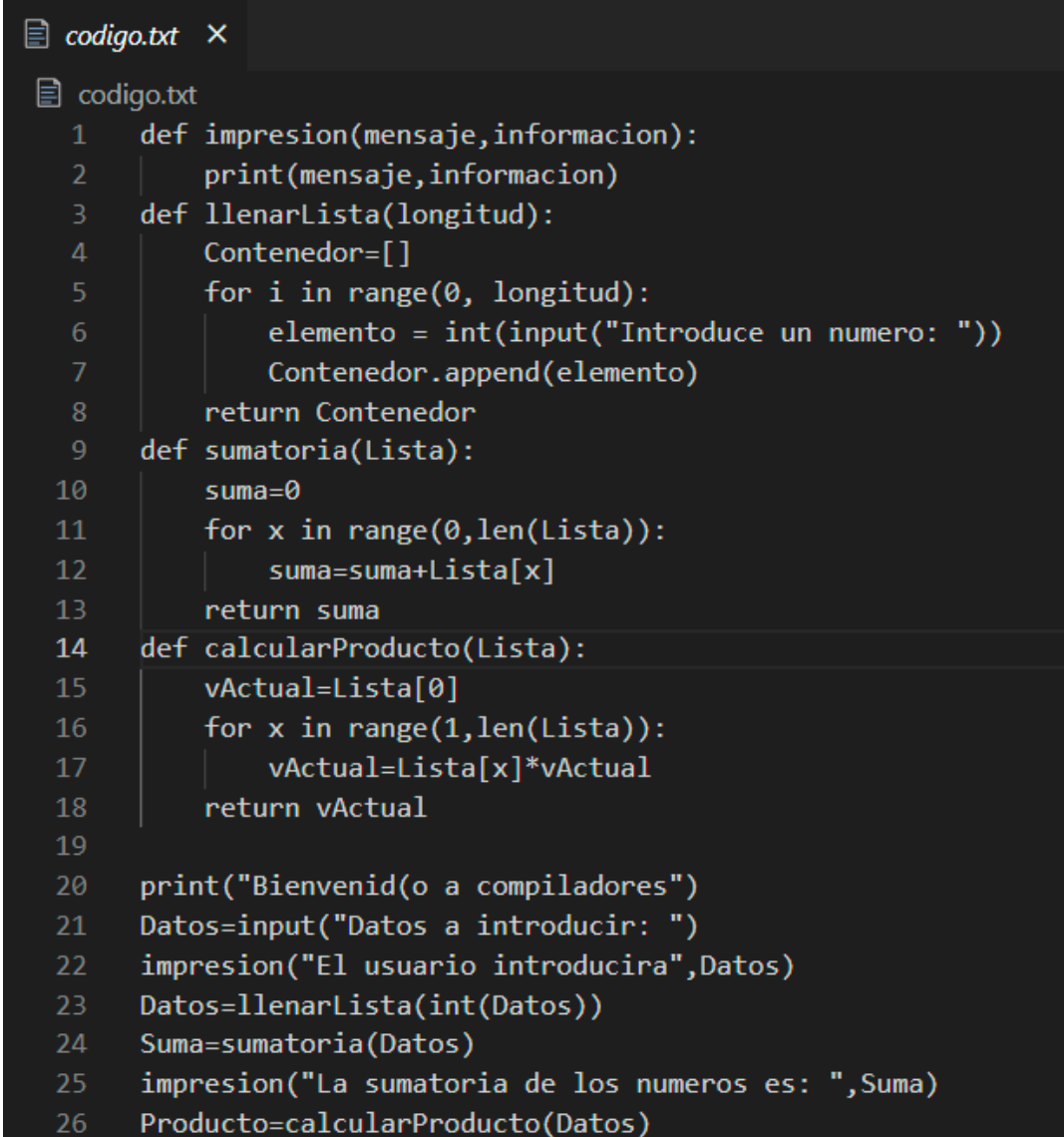


Figura 1. Diagrama de caso de uso del intérprete de código python a C ++.

Ejecución del programa.

1. Como primer paso de la ejecución del intérprete de código Python a C++, debemos ingresar el código python que deseamos traducir dentro de un archivo de texto plano el cual deberá ser guardado en carpeta donde se encuentre el archivo “compilador.py” y asignarlo con el nombre de “codigo.txt”.



```
codigo.txt x
codigo.txt
1  def impresion(mensaje,informacion):
2      print(mensaje,informacion)
3  def llenarLista(longitud):
4      Contenedor=[]
5      for i in range(0, longitud):
6          elemento = int(input("Introduce un numero: "))
7          Contenedor.append(elemento)
8      return Contenedor
9  def sumatoria(Lista):
10     suma=0
11     for x in range(0,len(Lista)):
12         suma=suma+Lista[x]
13     return suma
14 def calcularProducto(Lista):
15     vActual=Lista[0]
16     for x in range(1,len(Lista)):
17         vActual=Lista[x]*vActual
18     return vActual
19
20 print("Bienvenido a compiladores")
21 Datos=input("Datos a introducir: ")
22 impresion("El usuario introdujera",Datos)
23 Datos=llenarLista(int(Datos))
24 Suma=sumatoria(Datos)
25 impresion("La sumatoria de los numeros es: ",Suma)
26 Producto=calcularProducto(Datos)
```

Figura 2. Archivo de texto plano con código Python a traducir.

2. Para la ejecución del compilador es necesario la creación de otros 3 archivos texto planos.
 - El primero debe de tener el nombre de “*expresiones.txt*”, el cuál contendrá el primer análisis del código introducido, aquí se copiarán las líneas que tengan que sean correctas en cuanto a léxica y sintaxis.
 - El segundo archivo de texto a crear deberá tener el nombre de “*main.txt*” donde se guardarán todas las líneas de texto que correspondan a la función main del código Python.
 - El tercer y último archivo deberá tener el nombre de “*funciones.txt*” donde se guardarán las líneas del código Python que sean analizadas y detectadas como parte de una función o como función.

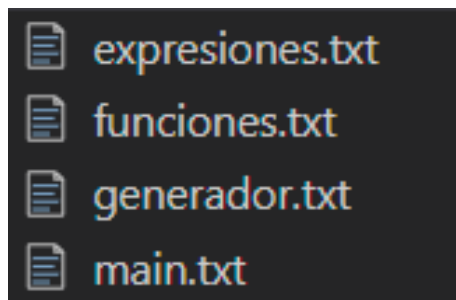


Figura 2.1. Creación de archivos de texto para la ejecución correcta del compilador.

3. De igual forma se deben de crear 3 archivos de tipo .cpp los cuales contendrán del código ya generado de Python a C ++.
 - El primer archivo deberá llamarse “*traduccion.cpp*”, el cuál contendrá la generación o traducción de código de la parte de las líneas del main.
 - El segundo deberá llamarse “*traduccionfun.cpp*”, en el cuál se guardarán la generación o traducción de código de las funciones ya detectadas.
 - Y el tercero “*codigo.cpp*”, contendrá la parte final analizada de las líneas de la parte del main.

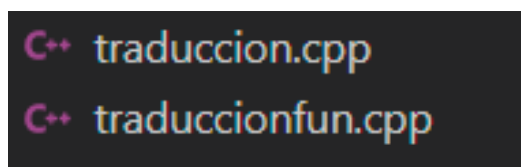


Figura 2.2. Creación de archivos tipo .cpp para la ejecución del compilador.

4. Se realiza la ejecución del archivo “*compilador.py*”, si existen errores en el código Python desde un inicio se mostrarán en terminal los errores y el número de línea en el que se encuentra.

```
Syntax error in line 1
                ef impresion(mensaje,informacion):
    verify.

Syntax error in line 3
                ef llenarLista(longitud):
    verify.
```

Figura 2.3. Recopilación y visualización de errores del código Python.

5. Si no existen errores se visualizará la siguiente leyenda en terminal.

```
Compiling...
Has been compiled
```

Figura 2.4. Mensaje de compilación exitosa y terminada.

6. Ahora dentro del primer archivo “*traduccion.cpp*” se podrá tener el código ya generado o traducido para la parte del main del código python.

```
C++ traduccion.cpp 5 X
C++ traduccion.cpp > ...
1  #include <iostream>
2  #include <cstring>
3  #include <cmath>
4  #include <vector>
5  #include <stdio.h>
6  using namespace std;
7  int main()
8  {
9      printf("Bienvenido a compiladores");
10     int Datos=0;
11     printf("Datos a introducir: ");
12     scanf("%d",&Datos);
13     impresion("El usuario introducirá %d ",Datos);
14     int tam=Datos;
15     int datos[tam];
16     datos[tam]=llenarLista(Datos);
17     int Suma=0;
18     Suma = sumatoria(datos);
19     impresion("La sumatoria de los numeros es: %d",Suma);
20     int Producto=0;
21     Producto = calcularProducto(datos);
22 }
```

Figura 2.5. Código traducido a C++ del main del código Python.

7. Después dentro del segundo archivo “*traduccionfun.cpp*” se podrá tener el código ya generado o traducido para la parte de las funciones correspondientes del código python.

```
C++ traduccionfun.cpp 5 X
C++ traduccionfun.cpp > impresion(string, int)
1 void impresion(string mensaje, int informacion){
2     printf("%s %d",mensaje.c_str(),informacion);
3 }
4 int llenarLista(int longitud){
5     int Contenedor[longitud];
6     for(int i=0; i<longitud; i++){
7         int elemento=0;
8         printf("Introduce un numero: ");
9         scanf("%d",&elemento);
10        Contenedor[i] = elemento;
11    }
12    return *Contenedor;
13 }
14
15 int sumatoria(int Lista[]){
16     int suma = 0;
17     for(int x=0; x<sizeof(Lista); x++){
18         suma = suma + Lista[x];
19     }
20 }
21     return suma;
22 }
23 int calcularProducto(int Lista[]){
24     int vActual = Lista[0];
25     for(int x=1; x<sizeof(Lista); x++){
26         vActual = Lista[x] * vActual;
```

Figura 2.6. Código traducido a C ++ correspondientes a funciones del código Python.

8. Por último se deben de unir el contenido de ambos archivos “*traduccion.cpp*” y “*traduccionfun.cpp*” en un archivo .cpp para su compilación en C ++.

```
#include <vector>
#include <stdio.h>
using namespace std;
void impresion(string mensaje, int informacion);
int llenarLista(int longitud);
int sumatoria(int Lista[]);
int calcularProducto(int Lista[]);
int main()
{
    printf("Bienvenid(o a compiladores");
    int Datos=0;
    printf("Datos a introducir: ");
    scanf("%d",&Datos);
    impresion("El usuario introducira %d ",Datos);
    int tam=Datos;
    int datos[tam];
    datos[tam]=llenarLista(Datos);
    int Suma=0;
    Suma = sumatoria(datos);
    impresion("La sumatoria de los numeros es: %d",Suma);
    int Producto=0;
    Producto = calcularProducto(datos);
}
void impresion(string mensaje, int informacion){
    printf("%s %d",mensaje.c_str(),informacion);
}
int llenarLista(int longitud){
    int Contenedor[longitud];
    for(int i=0; i<longitud; i++){
```

Figura 2.7. Códigos generados unidos en el archivo para compilar en IDE Dev C ++.

9. Finalmente se ejecuta el código y se visualiza el funcionamiento del programa ya en la terminal de Dev C ++.

```
Bienvenid(o a compiladoresDatos a introducir: 4
El usuario introducira %d 4Introduce un numero: 2
Introduce un numero: 3
Introduce un numero: 4
Introduce un numero: 5
La sumatoria de los numeros es: %d 14966275
-----
Process exited after 15.86 seconds with return value 0
Presione una tecla para continuar . . . █
```

Figura 2.8. Funcionamiento del código compilado traducido de Python a C ++ en Dev C ++.

VENTAJAS Y DESVENTAJAS DEL INTÉRPRETE DE PYTHON A C ++.

Respecto a las ventajas del intérprete que realice, puedo decir que fue hecho de la manera más completa posible. El primer punto favorable que puedo argumentar es que realiza una recopilación de errores, es decir, todos los errores que existan serán indicados en la terminal del IDE ya que muchos otros compiladores solo indican un error e inmediatamente se interrumpe el proceso. El segundo punto es el análisis mediante la técnica de divide y vencerás, esta técnica la emplee al dividir el código fuente en dos códigos para analizar por separado cada uno y así reducir la tasa de error y de tiempo de ejecución. El tercer punto es la creación de un módulo que detecta si las funciones declaradas en la parte del main coinciden con las funciones implementadas, ya que muchas veces no se declaran ocupando un espacio en memoria ocasionando desperdicio de recurso. La última ventaja que encontré fue el uso de apuntadores para el paso por parámetro de variables en funciones, esto ayudó a que el tiempo de ejecución bajará aún más, siendo más eficiente.

Por otro lado hay ciertas desventajas de mi compilador, la principal es que no genera un solo código, es decir se generan dos códigos uno en cada archivo.cpp para que después el usuario lo una en uno nuevo, esto puede provocar que se cometa algún error al momento de copiar y pegar los códigos. El último punto es que al utilizar apuntadores se afectó directamente al funcionamiento, puesto que en el proceso de generación de código se perdió algún dato de una variable provocando que el resultado de una operación de una función lo devolviera como tipo dirección y no como el valor real.

COMPILADORES EN FORMACIÓN ACADÉMICA.

El desarrollo de este intérprete de código python a C ++ como proyecto me sirvió bastante puesto que se retomaron conceptos de teoría computacional, la gran diferencia entre dicha materia y compiladores, es que aquí realmente se ve la aplicación de todas las herramientas, puedo decir que en teoría computacional los ejercicios eran hasta cierto punto tediosos y no entendía el propósito de realizar ciertos autómatas, expresiones regulares o máquinas de turing, sin embargo a lo largo de las fases del compilador pude relacionar cada vez más el cómo funcionaban dichos conceptos. Puedo decir que fue de mucha utilidad este proyecto ya que de igual forma se me ocurrieron nuevas formas de atacar diferentes problemas optimizando costos de tiempo y almacenamiento, un ejemplo claro que pude copiar a la materia de ingeniería de software fue el hacer validaciones de correo electrónico mediante expresiones regulares y esto me simplificó mucho el trabajo puesto que la lógica para los demás pasos era muy similar, sin dudas aprendí otra técnica para solucionar problemáticas y otro punto importante fue que el desarrollo de mi lógica se fortaleció. Finalmente me gustaría decir que siempre vemos una forma de resolver problemas ya que estamos acostumbrados a seguir ciertos patrones pero en ocasiones la solución puede ser demasiado y sencilla, de esta forma resolviendo la problemática sin mayor detalle.

REFERENCIAS BIBLIOGRÁFICAS.

- [1] ANÁLISIS SEMÁNTICO, LA TABLA DE SÍMBOLOS. (s.f.). Cartagena 99.
https://www.cartagena99.com/recursos/alumnos/apuntes/ININF2_M4_U5_T2.pdf
- [2] Analizador Sintáctico. (s.f.). RYTE Wiki. https://es.ryte.com/wiki/Analizador_Sintáctico
- [3] 2.4. Análisis semántico. (s.f.). Sites Google.
<https://sites.google.com/site/teoriadelenguajesycompiladores/procesadores-de-lenguaje/analisis-semantico>
- [4] 1.3.7. Generador de Código Objeto. (s.f.). CIDECAME.
http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/137_generador_de_codigo_objeto.html
- [5] Autómata finito. (s.f.). EcuRed. https://www.ecured.cu/Autómata_finito
- [6] Expresiones Regulares. Conócelas y Piérdeles el miedo. (s.f.). SG.
<https://sg.com.mx/content/view/545>