UNIVERSIDAD PANAMERICANA
Campus México

UNIVERSIDADPANAMERICANA

Materia: Teoría de Lenguajes Programación (COM153)

Profesor: Félix Orlando Martínez Ríos

Fecha de entrega: 30/11/2022

Ciclo: 1228

Nombre del proyecto: Equipo 2

Miembros del Equipo		
ID	**Nombre**	**Carrera**
0219008	Ian Nikolay Doshner Galland	LTISC
02173088	Bernardo Gonzalez Herrera	LTISC
0212627	Miguel Ángel Richter Flores	LTISC
0217377	Pilar Pineda Cervantes	LTISC
0228654	Santiago Valera Barreiros	LTISC

Rúbricas												
ID	**1-solve**	**4-ethical**		**4-impact**	**5-teams**					**7-knowledge**		
	C	**REI**	**ED-MF**	**IE**		**ER**	**CW**	**TM**	**CDDM**	**AK**	**AK**	**ALS**
0219008												
02173088												
0212627												
0217377												
0228654												

***Abstract*:**

El lenguaje es fundamental en la interacción humana, con el paso del tiempo y la introducción a la relación humano con máquina ha surgido la necesidad de entablar un lenguaje entre los dos. Gracias a esta necesidad surgieron los compiladores. Este proyecto tiene objetivos tanto de investigación como educativos. Los objetivos de investigación resaltan las herramientas de la lingüística implementadas en el área de la informática, incluyendo las gramáticas, las expresiones regulares y demás. Los educativos proporcionan un nuevo nivel de entendimiento con el lenguaje utilizado en la tecnología y el impacto de otras ciencias en la ciencia computacional.

Index Terms—Analizador Lexicográfico, Analizador sintáctico, Analizador semántico, Expresión regular, Compilador, Lenguajes de Programación, Pascal, C, Tabla de símbolos, Árbol sintáctico.

INTRODUCCIÓN

Este proyecto tiene como objetivo elaborar un compilador funcional que traduzca cualquier código fuente escrito en el lenguaje de programación pascal a código objeto escrito en el lenguaje de programación c. Para ello se desea programar e implementar lo siguiente:

- Un Analizador Lexicográfico diseñado en Flex que permite reconocer tokens a partir de expresiones regulares y una gramática definida.
- Un Analizador Sintáctico: diseñado en Yacc/Bison que permita reconocer la estructura de las cadenas capturadas por el lexicográfico y compararla con la estructura definida por la gramática establecida.
- Un Analizador Semántico: que haga uso de la tabla de símbolos y del árbol sintáctico para detectar errores semánticos y poder traducir código correctamente.
- Una Tabla de símbolos que capture el ámbito, el tipo, el nombre, la línea en la que se declara y las líneas en las que se usen los distintos símbolos del código fuente.
- Un Árbol Sintáctico que almacene la estructura del código fuente.

La estructura del documento es la siguiente: sección 2 nos introduce brevemente a lo que es un lenguaje de programación, el lenguaje de programación y por último el lenguaje de programación pascal. La sección 3 define lo que es una gramática, tipos de gramática, el motivo por el que es necesaria una gramática en este proyecto, una liga a la gramática que se utiliza y la definición de lo que es una expresión regular. La sección 4 define que es un compilador, sus partes y sus fases. Las secciones 5 hasta la 8 nos definen con más detalle las partes mencionadas en la sección anterior, su implementación en el proyecto y las herramientas utilizadas para la creación de cada parte. Por último las secciones 9 y 10 mostraran los resultados y las conclusiones alcanzadas dentro del proyecto.

LENGUAJES DE PROGRAMACIÓN

Un lenguaje de programación como lo dice su nombre es un lenguaje el cual tiene como objetivo establecer un medio de comunicación/control entre Hombre y Máquina. En la actualidad hay diversos tipos de lenguajes de programación, todos ellos tienen un conjunto de reglas sintácticas y semánticas que definen los tipos de datos a utilizar y, por tanto, los tipos de operaciones a realizar sobre ellos. **Conjuntos y Strings**: Un lenguaje de programación consiste en un conjunto de programas o un conjunto de strings. Es por eso por lo que en este proyecto se vio conveniente utilizar una gramática como un motor para generar estas sentencias y programas. En este proyecto nos centraremos en dos: Pascal y C.

- Pascal: Es un lenguaje de programación creado entre 1968 y 1969 por el profesor suizo Niklaus Wirth y publicado en 1970. Su objetivo era crear un lenguaje que facilitara a los estudiantes el aprendizaje de la programación a través de la programación estructurada y la estructuración de datos. Sin embargo, con el tiempo, su uso se extendió más allá de la academia para convertirse en una herramienta para crear una variedad de aplicaciones.
- C: Es un lenguaje de programación de propósito general desarrollado por Dennis Ritchie en Bell Labs entre 1969 y 1972 como una evolución del lenguaje B. Al igual que B, es un lenguaje para implementar sistemas operativos, especialmente Unix. C es el lenguaje de programación más popular utilizado para crear software de sistemas y aplicaciones, valorado por la eficiencia del código que genera.

GRAMÁTICAS Y EXPRESIONES REGULARES

Desde el punto de vista de la teoría de los autómatas, una gramática (”G”) es un conjunto acotado de reglas que describen todas las secuencias de símbolos pertenecientes a un lenguaje L dado. Se define a la gramática como un conjunto de reglas que se deben aplicar y seguir al escribir un lenguaje o código para que sean considerados correctos en ese lenguaje código. Las gramáticas tienen la siguiente estructura y elementos: **G** = *{*NT,T,S,P*}* Donde

- **NT** es el conjunto de elementos No terminales.
- **T** es el conjunto de elementos Terminales.
- **S** es el símbolo inicial de la gramática.
- **P** es el conjunto de reglas de producción.

A. Tipos de Gramáticas

- Gramáticas Regulares.
- Gramáticas Independientes del contexto.
- Gramáticas Dependientes del contexto.
- Gramáticas sin restricciones o sin Reglas.

Teniendo todo esto en cuenta, es necesario implementar una gramática para identificar correctamente el lenguaje de pascal que pase por nuestro compilador con el fin de poder generar código en c de manera adecuada. Para eso se utilizaron las reglas gramáticas detalladas en el documento de la siguiente liga: Gramatica Pascal

B. Expresiones regulares

Una expresión regular es un patrón o secuencia de caracteres que conforman al patrón, se utilizan para hacer coincidir combinaciones de caracteres en una cadena, englobar patrones
o operaciones de sustituciones. En el proyecto se definirán las expresiones regulares dentro del analizador lexicográfico para reconocer ciertos tokens de manera más eficaz.

COMPILADOR

Un compilador es un software que traduce un programa escrito en un lenguaje de programación de alto nivel (C/C++, COBOL, etc.) a lenguaje máquina. Los compiladores generan lenguaje ensamblador primero y luego traducen el lenguaje ensamblador a lenguaje de máquina. El lenguaje de entrada normalmente se le conoce como fuente y al de salida como objeto. El código fuente es un string de símbolos, estos símbolos consisten generalmente de letras, dígitos o ciertos símbolos especiales como +, - y (,). El código fuente también puede contener construcciones elementales del lenguaje, estos son nombres de variables, etiquetas, constantes, palabras claves y operadores.

A. Partes de un compilador

El proceso de compilación o traducción consiste en distintas partes o fases que realizan operaciones lógicas. Las partes de un compilador se muestran en la siguiente figura:

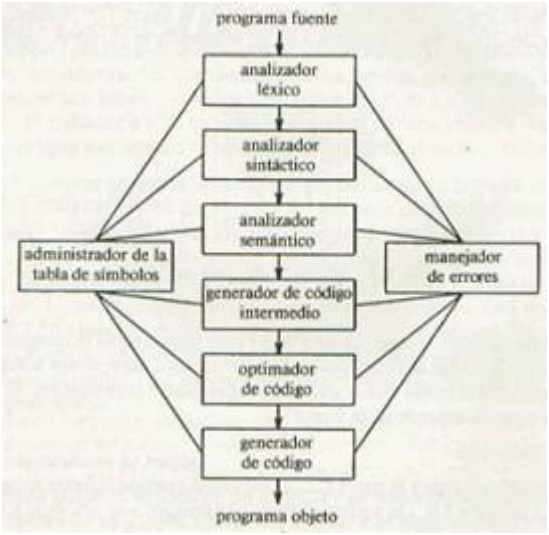


Fig. 1. Partes de un Compilador

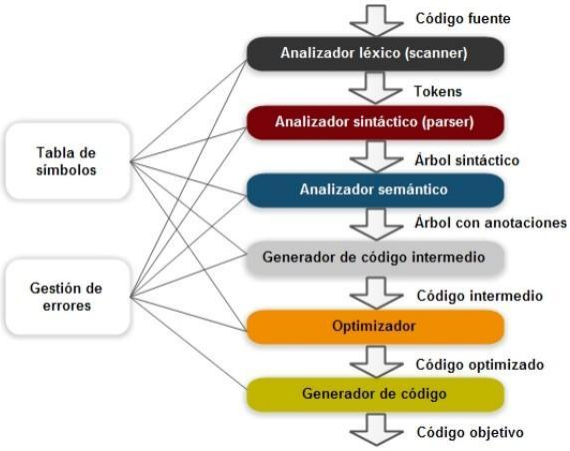


Fig. 2. Fases de un Compilador

ANALIZADOR LEXICOGRÁFICO

La primera fase de un compilador es el analizador lexicográfico o analizador léxico el cual tiene como objetivo leer el código fuente y agrupar las secuencias de caracteres en componentes léxicos o ”tokens”. Esta clasificación depende de los tokens o expresiones regulares que están definidas en el código de Flex utilizado para la creación del analizador lexicográfico. Como se definió anteriormente un código fuente contiene palabras clave, identificadores, operadores, constantes numéricas, signos de puntuación como separadores de sentencias, paréntesis, llaves, corchetes y demás. Todos estos componentes se pueden clasificar como componentes léxicos.

A. Flex

”Flex es una herramienta para generar escáneres: programas que reconocen patrones léxicos en un texto. flex lee los ficheros de entrada dados, o la entrada estándar si no se le ha indicado ningún nombre de fichero, con la descripción de un escáner a generar. La descripción se encuentra en forma de parejas de expresiones regulares y código C, denominadas reglas. flex genera como salida un fichero fuente en C, ‘lex.yy.c’, que define una rutina ‘yylex()’. Este fichero se compila y se enlaza con la librería ‘-lfl’ para producir un ejecutable. Cuando se arranca el fichero ejecutable, este analiza su entrada en busca de casos de las expresiones regulares. Siempre que encuentra uno, ejecuta el código C correspondiente.”(Vern Paxson,1995,p.4)

B. Implementación con Flex del Lexicográfico

Para el analizador lexicográfico se definieron expresiones regulares, palabras reservadas y variables para capturar los tokens escaneados de cualquier código de pascal introducido y poder almacenar en qué línea y columna del archivo se encuentran. Las expresiones regulares definidas son las siguientes:

```
const_cadena \"[^\n]*\"
digito [0-9]
letra [A-Za-z]
identif {letra}({letra}|{digito})*
addop [\+]|[-]
mulop [\*]|[\/]
simbolo [\<\.\(\>\);=\[:,\]\"]
entero [+-]?[0-9]+
real [+-]?{entero}+\.{digito}+([eE]{addop}?{digito}+)?
```

Fig. 3. Expresiones Regulares

Las expresiones regulares consisten en lo siguiente:

- const_cadena:
- digito: Números entre el ”0” y el ”9”.
- letra: Letras Mayúsculas de la ”A” a ”Z” y después letra minúsculas de la ”a” a la ”z”.
- identif: Cualquier combinación de n ”letra” o de n ”di ito”
- addop: Los símbolos ”+” o ”-”.
- mulop: Los símbolos ”*” o ”/”.
- simbolo: Cualquier carácter alfanumérico. (Para más información visualizar la regla de alfanumérico en la gramática utilizada Gramática Pascal)
- entero: Combinación de números entre el ”0” y el ”9” que puede ser tanto positivo ”+” como negativo ”-”.
- real: Cero o un signo ”+” o ”-” seguido de uno o más enteros un punto seguido por uno o más dígitos y por ultimo uno o ningún conjunto de expresiones constituidas por una e minúscula o una E mayúscula seguido de uno o ningún símbolo de addop seguido de uno o más dígitos

ANALIZADOR SINTACTICO

La segunda fase del compilador es el analizador sintáctico. El analizador sintáctico o parser usa los primeros componentes de los tokens que se producen por el analizador lexicográfico, para crear un árbol sintáctico, que tiene como función representar la estructura gramatical del flujo de tokens. Típicamente dentro de la representación del árbol los nodos representan una operación y los hijos de los nodos representan los argumentos de una operación.

A. Yacc/Bison

Yacc viene de las siglas *Yet Another Compiler-Compiler* que significa, otro compilador de compiladores. Yacc genera un analizador sintáctico basado en una gramática analítica dada, el código generado para el analizador sintáctico está escrito en el lenguaje de programación c. Como en este proyecto era necesario el uso de un analizador sintáctico se utilizó la versión GNU de Yacc, la cual es Bison. Esta herramienta nos permite utilizarla junto con Lex/Flex y con ello cumplir 2 partes de nuestro compilador, el analizador lexicográfico y el analizador sintáctico. El flujo de la creación del programa objetivo es el siguiente:

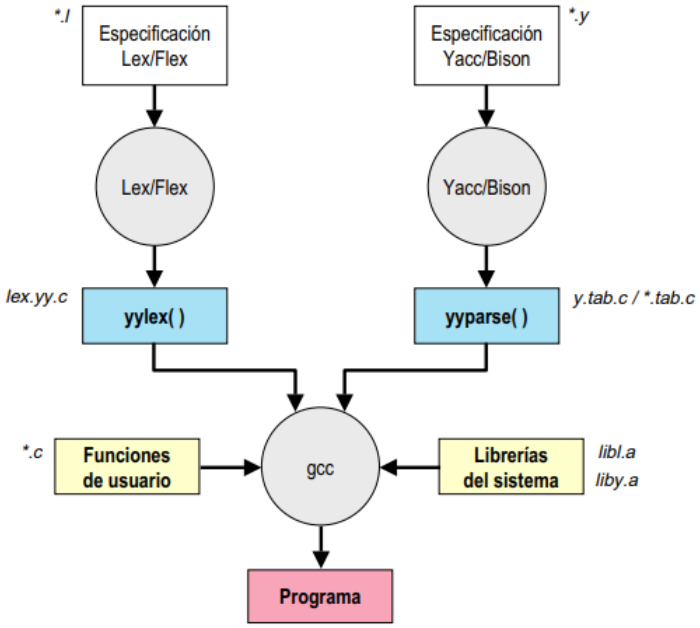


Fig. 4. Construcción del programa objetivo

ANALIZADOR SEMÁNTICO

El analizador semántico tiene como objetivo analizar la consistencia semántica del programa fuente con la definición del lenguaje. El lexicógrafo utiliza el árbol sintáctico y la información almacenada en la tabla de símbolos para revisar esta consistencia semántica. Adicionalmente recolecta los tipos de información y los almacena tanto en el árbol sintáctico como en la tabla de símbolos.

Para resumir estas 3 secciones podemos decir lo siguiente: El analizador lexicográfico identifica tokens y que estén bien escritos, el sintáctico se asegura de que el orden de los tokens sea el correcto, por último el semántico identifica si existe un significado válido en el conjunto de tokens validados por el lexicográfico y el sintáctico. Por ejemplo. ”El ratón le dispara al elefante.” Esta oración está escrita correctamente pero no tiene algún sentido lógico ya que es imposible que un ratón le dispare a un elefante. Por lo tanto no es válida.

IMPLEMENTACIÓN

A. Arbol Sintactico

Un árbol de la estructura sintáctica simplificada del código fuente escrito en el lenguaje de programación. Cada nodo del árbol representa una construcción que aparece en el código fuente. La gramática es abstracta porque no representa todos los detalles de la gramática real. Por ejemplo, la agrupación de paréntesis está contenida dentro de una estructura de árbol, y una construcción sintáctica tal como IF condición THEN puede ser denotada por un solo nodo con dos ramas. Un árbol más completa se puede observar con la siguiente entrada:

```
While b! = 0
        if  a > b
                a : = a – b
        else
                b : = b – a
return a
```

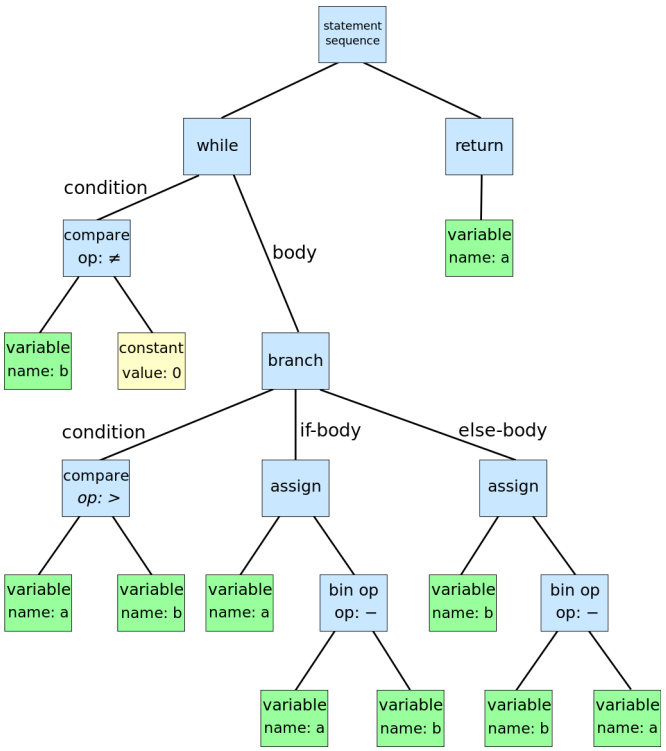


Fig. 5. Árbol sintáctico ejemplo.

La implementación del árbol consistió en realizar una estructura con enumerados, uniones y punteros para almacenar el tipo, valor, nombre o ámbito de lo reconocido en las reglas sintácticas. El nodo consiste en un puntero a su padre, un puntero a su hijo izquierdo y 2 punteros a su hermano izquierdo y derecho. Esto representaba un árbol con hojas amarradas en un estilo de lista ligada. Con esto podríamos tener control sobre n hijos ya que nos fijamos en los hermanos del primer hijo.
Lamentablemente la implementación del árbol no fue exitosa, se encontraron muchos errores al momento de implementarlo dentro del sintáctico.

B. Tabla de símbolos

La tabla de símbolos es una estructura de datos que contiene un registro de cada nombre de variable, con campos para cada atributo de la misma. La estructura puede variar pero debe estar diseñada de tal manera que le permita al compilador consultar el registro de cada variable rápidamente y que almacene o retire datos de ese registro. Todo esto es necesario ya que una función importante de los compiladores es tomar registro de las variables usadas en el lenguaje fuente y reunir información sobre los varios ámbitos de dichas variables. Estos atributos pueden proporcionar información sobre el almacenamiento adecuado para la variable, su tipo, el ámbito, el uso, argumentos y demás.

C. Gestión de errores

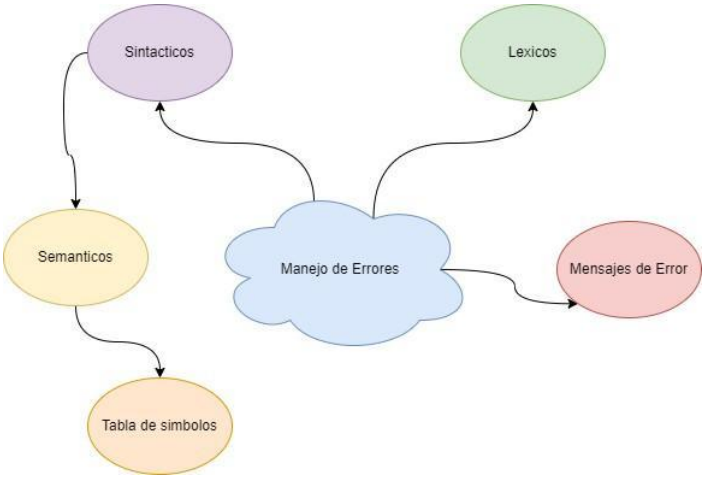


Fig. 6. Manejo de Errores.

Existen distintos tipos de errores durante la fase de análisis, existen los errores léxicos que consisten en la introducción de símbolos que no se encuentran en el alfabeto, estos son sencillos de detectar solo se tiene que comparar con el alfabeto existente cada que se encuentre un símbolo. Después tenemos los errores sintácticos, estos al igual que los léxicos son sencillos de detectar ya que hacen referencia a la estructura de la cadena. Un compilador se enfoca en la estructura, son fáciles de manejar si se implementa un analizador ascendente o descendente. Por último tenemos los errores semánticos los cuales son los más difíciles de detectar. Un compilador generalmente no los detecta o se les pueden pasar, mediante una gramática libre de contexto ya que la gramática libre de contexto se enfoca en la estructura, no en el significado. Si se busca la definición de la palabra semántica encontraremos que se refiere al significado de una expresión lingüística y precisamente ese es un error semántico. Veamos ejemplos de ellos.

- **Error semántico de Tipado** Estos errores hacen referencia a la conversión de tipos.

```
int  x ;
x = 3 . 5
```

Otro ejemplo de error es el siguiente, la función es de tipo void y se intenta retornar algo, sintácticamente está bien escrito pero semánticamente causa un error.

```
void  miFuncion ( ) {
return 3 ;
}
```

- **Error de ejecución** La siguiente expresión es correcta sintácticamente, y será traducida a código binario. Sin embargo cuando se efectúe la operación se producirá un error, ya que en las matemáticas no se puede dividir entre cero.

```
int  c ;
c = 5 / 0
```

- **Error lógico** En el siguiente ejemplo podemos ver que el programador desea escribir un código que le permita sumar 2 números pero termina escribiendo algo distinto. Estos errores son los más difíciles de detectar ya que el programa no marcará error sin embargo no funcionará correctamente. En este caso los números se multiplicarán y no se sumarán.

```
Res = a * b ;
```

Un último ejemplo de error es el siguiente, se crea un arreglo de cierto espacio o dimensión y se intenta manipular algo fuera de ella.

```
double a = new double [ 5 ] ;
   .
   .
a [ 7 ] = 3 ;
```

Para manejar todo estos errores debe existir un mecanismo para detectarlos. Para ello existen todos los analizadores de un compilador. Especialmente el semántico, todas las acciones semánticas se embebe al análisis sintáctico, el análisis sintáctico llama a las subrutinas del semántico y estas subrutinas hacen uso de la tabla de símbolos para detectar todo tipo de error. El compilador debe estar diseñado de una manera que permita anticipar estos casos o lo que un programador le va a mandar. Al manejarlos o detectarlos se pueden terminar con mensajes de error, ya que es imposible saber con certeza la lógica del programador del código fuente, no podemos cambiar la lógica del código.

Manejo de errores en el proyecto:

- **Lexicográfico** Flex nos ayuda a detectar errores con la función *yyerror(yytext);* esta es utilizada cuando nuestro analizador lexicográfico encuentra algún símbolo que no sea parte de los símbolos definidos(expresiones regulares, tokens y demás).

- **Sintactico** Bison nos ayuda a detectar errores con una función llamada yyerror, esta se llama cuando un conjunto de cadenas o una sola cadena no tiene la estructura adecuada a la gramática definida. La función se muestra a continuación:

```
int yyerror ( char* s )
{
  printf ( ” Error %s \ n ” , s ) ;

    e x i t ( 1 ) ;  / * Sale del programa * /
     return 0 ;

}
```

- **Semántico** El semántico hace uso de nuestra tabla de símbolos y nuestro árbol sintáctico. Si no se encuentra algún símbolo previamente definido en ellos se marca un error o si el significado no cuadra con el definido en la tabla marca error.

D. Generador de Código

Para realizar la traducción se utilizó código c en la gramática dentro del archivo de Yacc/Bison. Se utilizó fprintf para ir guardando en un archivo de salida el código correspondiente en C++ de lo que se iba reconociendo. y se utilizó cadenas de caracteres para insertar los includes y otras sentencias por defecto del lenguaje. Se utilizaron varios ejemplos .pas para verificar la correcta traducción del proyecto.

EXPLICACIÓN CÓDIGO

Para la solución, no se implementó un árbol sintáctico, sin embargo, se logró realizar la traducción utilizando cadenas de caracteres. Cada vez que se sintetiza una producción, los terminales y no terminales de la producción, junto con el código necesario para realizar la traducción a C++ es concatenado poco a poco para finalmente, en cierto punto de la gramática, escribir en el archivo.

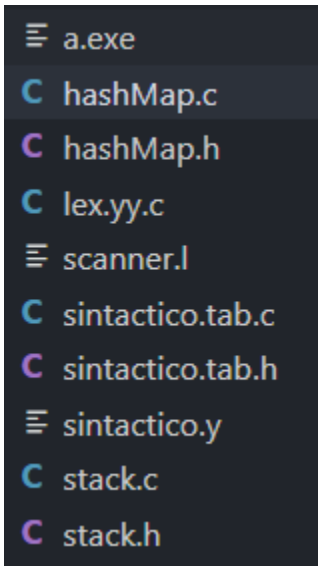
Para la tabla de símbolos se implementó una tabla hash apoyada por una pila que lleva el seguimiento del ámbito en el que se encuentra cada identificador. Antes de agregar o utilizar cualquier identificador, éste es buscado en la tabla de símbolos, y dependiendo de si se encuentra o no, se notificará al usuario si una variable está siendo utilizada sin declarar o declarada dos veces.

RESULTADOS

Creando analizador lexicográfico, analizador sintáctico y el ejecutable

```
PS C:\Lenguajes\ProyectoFinal> flex -l scanner.l
PS C:\Lenguajes\ProyectoFinal> bison -ld sintactico.y
sintactico.y: conflictos: 3 desplazamiento/reducción, 6 reducción/reducción
```

Archivos creados




```
IDENTIFICADOR   |       TIPO    |       FILA DE DECLARACION     |       AMBITO          |       FILAS DE USO
-------------------------------------------------------------------------------------------------------------------
nombre                  string                  3                       HelloWorld                      7, 8,
-------------------------------------------------------------------------------------------------------------------

Reconocido correctamente

C:\Lenguajes\S\Solucion\x64\Debug\Solucion.exe (proceso 22768) se cerró con el código 0.
Para cerrar automáticamente la consola cuando se detiene la depuración, habilite Herramientas ->Opciones ->Depuración ->
Cerrar la consola automáticamente al detenerse la depuración.
Presione cualquier tecla para cerrar esta ventana. . .
```

```
#include <stdio.h>
#include <string>
#include <iostream>

using namespace std;

string nombre;

int main() {
cout << "Ingresa tu nombre" << endl;
cin >> nombre;
cout << "Hola, " << nombre << endl;
}
```

CONCLUSIONES

Haciendo uso le lex y yacc se pudo hacer un reconocedor sintáctico capaz de no solo reconocer una gramática sino también de traducirla de un lenguaje a otro, de Pascal a C + +. El uso de estas herramientas nos permite facilitar ciertas tareas como reconocer patrones, reconocer gramáticas o traducir entre diversos lenguajes de programación, usando estructuras de datos que nos permiten reconocer los elementos de la gramática como se implementó en la tabla de símbolos, se almacenó, identificó y reconoció la gramática para hacer la traducción entre los lenguajes de programación.

REFERENCES

[1] GRUPO. ”Compilador Con Bison y Flex”. BLOG DE COMPILADORES, 27 de agosto de 2017. http://analizadorsemanticolab1.blogspot.com/2017/08/complidor-con- bison-y-flex.html.

[2] Manual de Bison. https://www.gnu.org/software/bison/manual/bison.pdf

[3] Tremblay, J., Sorenson. The Theory And Practice of Compiler Writing.

Ejemplos para ejecutar el código:

Hola mundo
Este código pide ingresar una cadena de caracteres y lo concatena con un mensaje de hola que se imprime en pantalla

```
program HelloWorld();
var
   nombre : string;

begin
   writeln("Ingresa tu nombre");
   readln(nombre);
   writeln("Hola, ", nombre)
end.
```

Número máyor
Este ejemplo en Pascal regresa el número máyor entre dos números

```
program maxNum();
  var
    a, b, ret : integer;

  function max(num1: integer ; num2: integer): integer;

  begin
    if num1 > num2 then
      max := num1
    else
      max := num2
  end;

begin
  a := 100;
  b := 200;
  ret := max(a, b);

  write("Max value is : ");
  writeln(ret)
end.
```

Principal

```
program main();
    var a,b,c,d: integer;

    function Add(a: integer; b: integer) : integer;
    begin
```

```
        Add := a + b
    end;

    function Mult(c: integer; a: integer) : integer;
    begin
        Mult := c * a
    end;

begin
    a := 9;
    b := 7;
    c := Add(a, b);
    d := Mult(c, a);
    while (a > b) do
      b := b + 1;
    writeln(c);
    writeln(d)
end.
```