Instituto Politécnico Nacional

Escuela Superior de Cómputo

Compiladores

Practica 6: Intérprete

Nombre: *Mauro Sampayo Hernández*

Grupo: *3CV17*

Profesor: *Hernández Olvera Luis Enrique*

Fecha de entrega: *27 de diciembre del 2021*

1. Introducción:

1.1 Intérprete

Un intérprete, al igual que un compilador, es un programa que tiene como objetivo ejecutar código fuente de un lenguaje de alto nivel o un lenguaje de scripting pero sin tener que obtener código máquina como resultado final. Unos ejemplos de lenguajes interpretados son el lenguaje PHP, Python o el shell de comandos de Linux.

Para la creación de estos, existen diversas herramientas que nos permiten llevar a cabo la generación de analizadores léxicos, semánticos y sintácticos. Para este práctica se usarán las herramientas Flex y Bison, las cuales se explicaran a continuación:

1.2 Flex

Flex, también conocida como Lex, es una herramienta para generar escáneres, es decir programas que reconocen patrones léxicos en un texto a partir de la especificación de expresiones regulares para describir dichos patrones.

El compilador Lex transforma los patrones de entrada en un diagrama de transición y genera código, en un fichero fuente en C llamado “lex.yy.c”, que simula este diagrama de transición. 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, y siempre que encuentra uno, ejecuta el código C correspondiente.

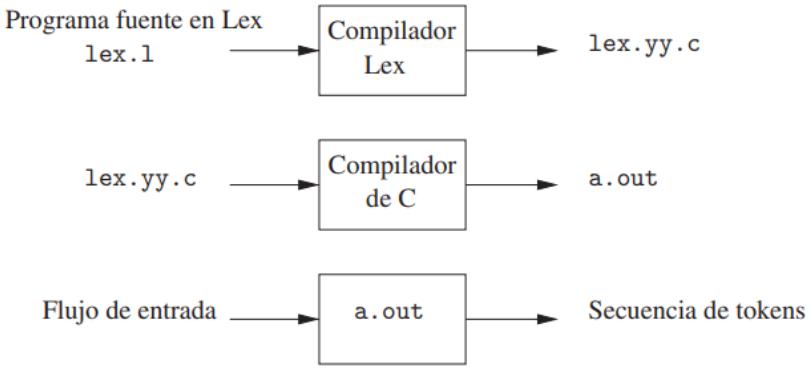

Figura 1. Creación de un analizador léxico en Lex

1.2.1 Estructura de los programas en Lex

Un programa en Lex tiene la siguiente forma:

Definiciones

%%

Reglas de traducción

%%

Código de usuario

1.2.1.1 Definiciones

La sección de Definiciones incluye las declaraciones de variables, constantes de manifiesto (identificadores que se declaran para representar a una constante; por ejemplo, el nombre de un token) y definiciones regulares. Las definiciones de nombre tienen la forma:

nombre definición

El **nombre** es una palabra que comienza con una letra o un subrayado "_" seguido por cero o más letras, dígitos, subrayados "_" o guiones "- ". La **definición** se considera que comienza en el primer caracter que no sea un espacio en blanco siguiendo al nombre y continuando hasta el final de la línea. Posteriormente se puede hacer referencia a la definición utilizando **"{nombre}",** que se expandirá a **"(definición)".** Por ejemplo,

```
DIGITO    [0-9]
ID        [a-z][a-z0-9]*
```

define "DIGITO" como una expresión regular que empareja un dígito sencillo, e "ID" como una expresión regular que empareja una letra seguida por cero o más letras o dígitos. Una referencia posterior a

```
{DIGITO}+"."{DIGITO}*
```

es idéntica a

```
([0-9])+"."([0-9])*
```

y empareja uno o más dígitos seguido por un punto "." seguido por cero o más dígitos.

1.2.1.2 Reglas de Traducción

La sección de Reglas de Traducción tiene una serie de reglas con la siguiente forma:

Patrón {Acción}

Donde el **patrón** es una expresión regular, la cual puede usar las definiciones regulares de la sección de declaraciones, y debe ir sin sangrar. Las **acciones** son fragmentos de código, por lo general escritos en C, que definen las acciones a ejecutar en caso de que se encuentre alguna concordancia con el patrón de texto de entrada, y que deben comenzar en la misma línea que el **patrón.**

1.2.1.3 Código de Usuario

Finalmente, la sección de Código de Usuario contiene las funciones adicionales que se pueden utilizar en las acciones y que se copia a "lex.yy.c" literalmente. La presencia de esta sección es opcional. Si se omite, el segundo **"%%"** en el fichero de entrada se podrá omitir también.

En las secciones de definiciones y reglas, cualquier texto sangrado o encerrado entre **"%{' y `%}"** se copia íntegramente a la salida (sin los **%{}'s**). Los **%{}'s** deben aparecer sin sangrar en líneas ocupadas únicamente por estos.

En la sección de reglas, cualquier texto o **%{}** sangrado que aparezca antes de la primera regla podrá utilizarse para declarar variables que son locales a la rutina de análisis y (después de las declaraciones) al código que debe ejecutarse siempre que se entra a la rutina de análisis. Cualquier otro texto sangrado o **%{}** en la sección de reglas sigue copiándose a la salida, pero su significado no está bien definido y bien podría causar errores en tiempo de compilación.

En la sección de definiciones (pero no en la sección de reglas), un comentario sin sangría (es decir, una línea comenzando con **"/*"**) también se copia literalmente a la salida hasta el próximo **"*/"**.

1.2.2 Patrones

Los patrones en la entrada se escriben utilizando un conjunto extendido de expresiones regulares. Estas son:

x ⟶ Empareja el caracter "x"
. ⟶ Cualquier caracter (byte) excepto una línea nueva
[xyz] ⟶ Una "clase de caracteres"; en este caso, el patrón empareja una "x", una "y", o una "z".
[abj-oZ] ⟶ Una "clase de caracteres" con un rango; en este caso empareja una "a", una "b", cualquier letra desde la "j" hasta la "o", o una "Z".

[^A-Z] → Una "clase de caracteres negada", es decir, cualquier caracter menos los que aparecen en la clase. En este caso, cualquier caracter EXCEPTO una letra mayúscula.
[^A-Z\n] → Cualquier caracter EXCEPTO una letra mayúscula o una línea nueva.
r* → Cero o más r's, donde r es cualquier expresión regular.
r+ → Una o más r's.
r? → Cero o una r (es decir, "una r opcional").
r{2,5} → De dos a cinco r's.
r{2,} → Dos o más r's.
r{4} → Exactamente 4 r's.
{nombre} → La expansión de la definición de "nombre".
"[xyz]\"foo" → La cadena literal: [xyz]"foo.
\x → Si x es una “a”, “b”, “f”, “n”, “r”, “t”, o “v”, entonces la interpretación ANSI-C de \x. En otro caso, un literal “x” (usado para indicar operadores tales como “*”)
\0 → Un caracter NUL (código ASCII 0).
\123 → El caracter con valor octal 123.
\x2a → El caracter con valor hexadecimal 2a.
(r) → Empareja una r; los paréntesis se utilizan para anular la precedencia (ver más abajo).
rs → La expresión regular r seguida por la expresión regular s; se denomina "concatenación".
r|s → Bien una r o una s.
r/s → Una r pero sólo si va seguida por una s.
^r → Una r, pero sólo al comienzo de una línea (es decir, justo al comienzo del análisis, o a la derecha después de que se haya analizado una línea nueva).
r$ → Una r, pero sólo al final de una línea (es decir, justo antes de una línea nueva).
<s>r → Una r, pero sólo en la condición de arranque s.
<s1,s2,s3>r → Una r, pero en cualquiera de las condiciones de arranque s1, s2, o s3.
<*>r → Una r en cualquier condición de arranque, incluso una exclusiva.
<<EOF>> → Un fin-de-fichero.
<s1,s2><<EOF>> → Un fin-de-fichero en una condición de arranque s1 o s2.

1.3 Bison

Bison es un generador de analizadores sintácticos de propósito general que convierte una descripción gramatical para una gramática independiente del contexto LALR(1) en un programa en C que analice esa gramática.

Para que Bison analice un lenguaje, este debe ser descrito por una **gramática independiente del contexto**. Esto quiere decir que debe especificar uno o más grupos sintácticos y dar reglas para construirlos desde sus partes.

En las reglas gramaticales formales para un lenguaje, cada tipo de unidad sintáctica o agrupación se identifica por un **símbolo**. Aquellos que son construidos agrupando construcciones más pequeñas de acuerdo con reglas gramaticales se denominan **símbolos no terminales**; aquellos que no pueden subdividirse se denominan **símbolos terminales** o **tipos de tokens**. Denominamos **token** a un fragmento de la entrada que corresponde a un solo símbolo terminal, y **grupo** a un fragmento que corresponde a un solo símbolo no terminal.

Cada símbolo no terminal debe poseer reglas gramaticales mostrando como está compuesto de construcciones más simples.

Se debe distinguir un símbolo no terminal como el símbolo especial que define una declaración completa en el lenguaje. Este se denomina **símbolo de arranque**.

El analizador de Bison lee una secuencia de tokens como entrada, y agrupa los tokens utilizando las reglas gramaticales. Si la entrada es válida, el resultado final es que la secuencia de tokens entera se reduce a una sola agrupación cuyo símbolo es el símbolo de arranque de la gramática.

1.3.1 Estructura de los programas en Bison

La forma general de una gramática de Bison es la siguiente:

```
%{
declaraciones en C
%}
Declaraciones de Bison
%%
Reglas gramaticales
%%
Código C adicional
```

Los “%%”, “%{“ y “%}” son signos de puntuación que aparecen en todo archivo de gramática de Bison para separar las secciones.

1.3.1.1 Declaraciones en C

La sección de declaraciones en C contiene definiciones de macros y declaraciones de funciones y variables que se utilizan en las acciones en las reglas de la gramática. Estas se copian al principio del archivo del analizador de manera que precedan la definición de yyparse. Se puede utilizar “#include” para obtener las declaraciones de un archivo de cabecera. Si no se necesita ninguna declaración en C, se pueden omitir los delimitadores “%{“ y “%}” que delimitan esta sección.

1.3.1.2 Declaraciones de Bison

Las declaraciones de Bison declaran los nombres de los símbolos terminales y no terminales, y también pueden describir la precedencia de operadores y los tipos de datos de los valores semánticos de varios símbolos.

1.3.1.3 Reglas Gramaticales

Las reglas gramaticales definen cómo construir cada símbolo no terminal a partir de sus partes. Debe haber siempre al menos una regla gramatical, y el primer “%%” (que precede a las reglas gramaticales).

1.3.1.4 Código C adicional

La sección de *código C adicional* se copia al pie de la letra a la salida del fichero del analizador, al igual que la sección de *declaraciones en C* que se copia al principio. Este es el lugar más conveniente para poner cualquier cosa que quiera tener en el archivo del analizador pero que no deba venir antes que la definición de yyparse. Las definiciones de yylex e yyerror a menudo van en esta parte. Si esta sección está vacía, puede omitir el “%%” que los separa de las reglas gramaticales.

2 Problemas enfrentados al programar la práctica:

Si bien, durante el desarrollo de esta práctica hubo diversos problemas enfrentados, solo se mencionarán los 2 problemas principales que surgieron, debido a que el resto se trataron de problemas muy mínimos del lado del analizador sintáctico.

2.1 Creación de la Tabla de Símbolos

El mayor problema enfrentado en esta práctica fue la parte de la creación de la Tabla de Símbolos para el almacenamiento de variables con sus respectivos valores. Si bien en un principio no tenía muy claro la manera en la que esta sería implementada ultimadamente opte por usar de base una Lista Doblemente Ligada, en cuyos nodos fueran almacenados las variables declaradas durante la ejecución del Intérprete con sus respectivos datos.

La estructuración de los nodos se modificó para que en cada uno de ellos pudiese residir una estructura de datos para variables, en la cual se pudiese almacenar el nombre y tipo de dato (int, double o string) de una variable, de tal manera que cada nodo emulara una fila de la Tabla de Símbolos.

Adicionalmente en la estructura de datos de las variables, se incluyó la presencia de una segunda estructura de datos en la cual se pudiese almacenar el valor que tendría la variable. En la estructura de datos para valores se tiene un apartado para guardar valores enteros, decimales o de cadenas de texto.

Finalmente se realizó la modificación y creación de funciones especializadas para poder realizar el manejo de los datos de las variables dentro de la Tabla de Símbolos. Dichas funciones se describen brevemente a continuación:

- **Inicializar Tabla:** Inicializa la Tabla de Símbolos y los elementos que contienen para poder almacenar variables en ella posteriormente.
- **Destruir Tabla:** Destruye la Tabla de Símbolos por medio de la eliminación de todos los nodos que estén contenidos en la misma.
- **Declaración Variable:** Almacena variables nuevas que hayan sido declaradas durante la ejecución del Intérprete dentro de la Tabla de Símbolos.
- **Asignación Variable:** Asigna valores a las variables que residan dentro de la Tabla de Símbolos.
- **Búsqueda Variable:** Realiza la búsqueda de variables específicas dentro de la Tabla de Símbolos. En caso de que sea encontrada, devuelve un puntero al nodo donde está almacenada, en caso contrario devuelve un valor nulo.
- **Tipo Dato:** Realiza la búsqueda de variables específicas dentro de la Tabla de Símbolos para determinar el tipo de dato de la variable en cuestión. En caso de que la variable sea encontrada, devuelve su tipo de dato, en caso contrario devuelve un valor nulo.

- **Imprimir Tabla:** Imprime la Tabla de Símbolos para su visualización de forma gráfica.

2.2 Potencia entre Cadenas

El segundo mayor desafío enfrentado durante la práctica fue la de realizar la potencia de cadenas, al ser que esta función no existe de manera estandarizada en C, por lo que se tuvo que realizar su implementación desde cero.

La solución usada para lograr su implementación fue crear una función, la cual reciba la cadena a potenciar, y el exponente de la función, y de esta manera generar la potenciación de la cadena, considerando lo siguiente:

- Si el exponente es igual a cero, se devuelve una cadena vacía.
- Si el exponente es mayor a cero se replica la cadena tantas veces como especifique el exponente.
- Si el exponente es menor a cero se replica la cadena de manera invertida tantas veces como especifique el exponente.

Para lograr la inversión de la cadena en el último caso, se realizó otra función para poder invertir cadenas de texto.

3 Pruebas y Capturas de Pantalla:

Se realizo el desarrollo de un Intérprete con Flex y Bison, el cual reconoce ciertos lexemas escritos en ANSI C los cuales se enlistarán a continuación junto con algunas pruebas realizadas con el objetivo de mostrar su correcto funcionamiento:

3.1 Números Enteros con y sin signo

- Número entero con signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
-33
->Operador -
->Numero Entero 33
->Salto de linea

        ->Resultado: -33

+14
->Operador +
->Numero Entero 14
->Salto de linea

        ->Resultado: 14
```

Figura 2. Números enteros CON signo

- Número entero sin signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
33
->Numero Entero 33
->Salto de linea

        ->Resultado: 33

12
->Numero Entero 12
->Salto de linea

        ->Resultado: 12
```

Figura 3. Números enteros SIN signo

3.2 Números Decimales con y sin signo

- Número decimal con signo:

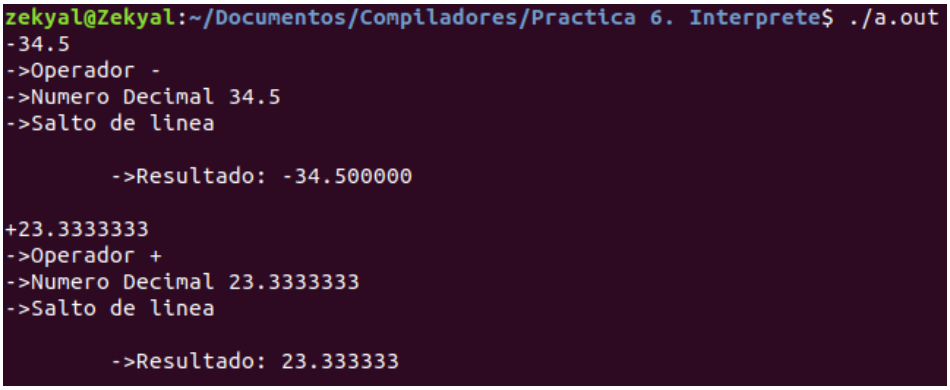

Figura 4. Números decimales CON signo

- Número decimal sin signo:

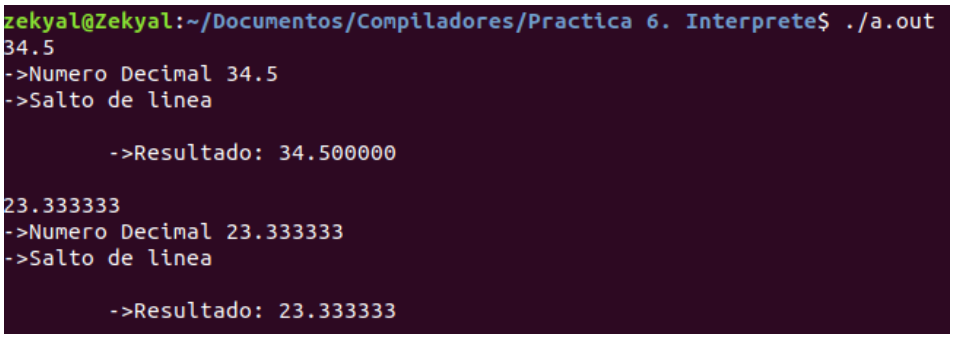

Figura 5. Números decimales SIN signo

3.3 Cadenas de Texto

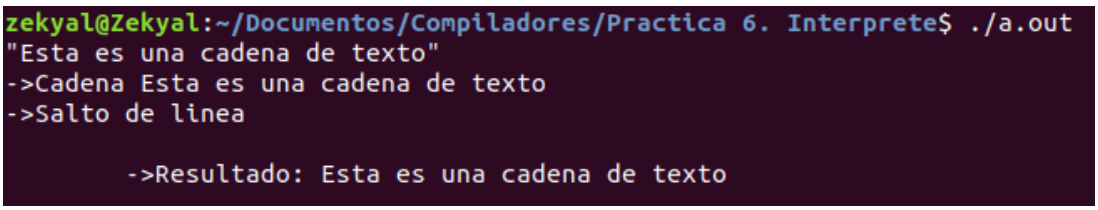

Figura 6. Cadenas de Texto

3.4 Operaciones Matemáticas

A continuación, se enlistan todas las operaciones matemáticas que reconoce el analizador léxico y sintáctico, junto con algunas pruebas realizadas utilizando números con y sin signo:

3.4.1 Suma “+”

- Suma con números con signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
-26+-13
->Operador -
->Numero Entero 26
->Operador +
->Operador -
->Numero Entero 13
->Salto de linea

        ->Resultado: -39

27.4+2
->Numero Decimal 27.4
->Operador +
->Numero Entero 2
->Salto de linea

        ->Resultado: 29.400000
```

Figura 7. Sumas con valores numéricos CON signo

- Suma son números sin signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
234+12
->Numero Entero 234
->Operador +
->Numero Entero 12
->Salto de linea

        ->Resultado: 246

3.333+6.666
->Numero Decimal 3.333
->Operador +
->Numero Decimal 6.666
->Salto de linea

        ->Resultado: 9.999000
```

Figura 8. Sumas con valores numéricos SIN signo

3.4.2 Resta "-"

- Resta con números con signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
-234--234
->Operador -
->Numero Entero 234
->Operador -
->Operador -
->Numero Entero 234
->Salto de linea

        ->Resultado: 0

-5.6-+1.1
->Operador -
->Numero Decimal 5.6
->Operador -
->Operador +
->Numero Decimal 1.1
->Salto de linea

        ->Resultado: -6.700000
```

Figura 9. Restas con valores numéricos CON signo

- Resta con números sin signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
6-6
->Numero Entero 6
->Operador -
->Numero Entero 6
->Salto de linea

        ->Resultado: 0

4.4-2
->Numero Decimal 4.4
->Operador -
->Numero Entero 2
->Salto de linea

        ->Resultado: 2.400000
```

Figura 10. Restas con valores numéricos SIN signo

3.4.3 Multiplicación ""*

- Multiplicación con números con signo:

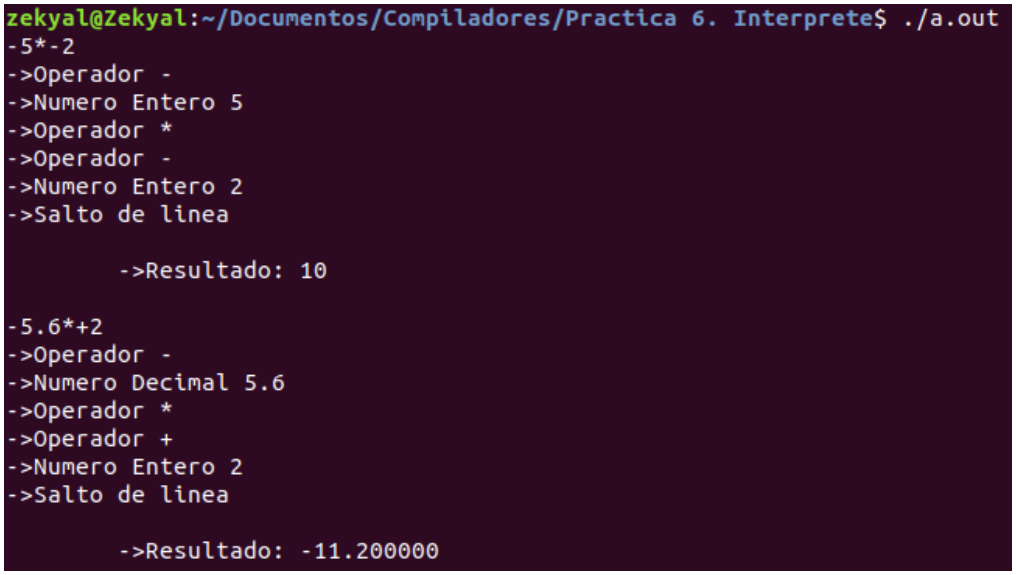

Figura 11. Multiplicaciones con valores numéricos CON signo

- Multiplicación con números sin signo:

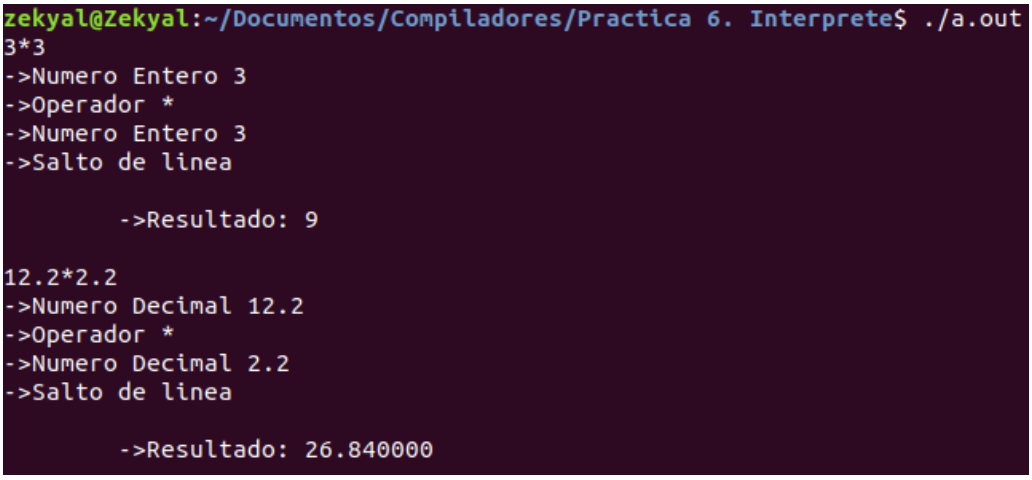

Figura 12. Multiplicaciones con valores numéricos SIN signo

3.4.4 División "/"

- División con números con signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
-11/-11
->Operador -
->Numero Entero 11
->Operador /
->Operador -
->Numero Entero 11
->Salto de linea

        ->Resultado: 1

-5.5/+2
->Operador -
->Numero Decimal 5.5
->Operador /
->Operador +
->Numero Entero 2
->Salto de linea

        ->Resultado: -2.750000
```

Figura 13. Divisiones con valores numéricos CON signo

- División con números sin signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
11/11
->Numero Entero 11
->Operador /
->Numero Entero 11
->Salto de linea

        ->Resultado: 1

3.3333/2
->Numero Decimal 3.3333
->Operador /
->Numero Entero 2
->Salto de linea

        ->Resultado: 1.666650
```

Figura 14. Divisiones con valores numéricos SIN signo

3.4.5 Modulo “%”

- Modulo con números con signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
-11%-11
->Operador -
->Numero Entero 11
->Operador %
->Operador -
->Numero Entero 11
->Salto de linea

        ->Resultado: 0

-45%+2
->Operador -
->Numero Entero 45
->Operador %
->Operador +
->Numero Entero 2
->Salto de linea

        ->Resultado: -1
```

Figura 15. Modulo con valores numéricos CON signo

- Modulo con números sin signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
3%2
->Numero Entero 3
->Operador %
->Numero Entero 2
->Salto de linea

        ->Resultado: 1

56.44%234.5
->Numero Decimal 56.44
->Operador %
->Numero Decimal 234.5
->Salto de linea

        ->Resultado: 56.440000
```

Figura 16. Modulo con valores numéricos SIN signo

3.4.6 Función Potencia

Para esta operación, se mostrará la ejecución de la función utilizando combinaciones de letras mayúsculas y minúsculas para cada prueba.

- Potencia con números con signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
POW(-23,4)
->Potencia
->Simbolo (
->Operador -
->Numero Entero 23
->Simbolo ,
->Numero Entero 4
->Simbolo )
->Salto de linea

        ->Resultado: 279841

POW(-6.6, +2)
->Potencia
->Simbolo (
->Operador -
->Numero Decimal 6.6
->Simbolo ,
 ->Operador +
->Numero Entero 2
->Simbolo )
->Salto de linea

        ->Resultado: 43.560000
```

Figura 17. Función Potencia con valores numéricos CON signo

- Potencia con números sin signo:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
pOw(5,6)
->Potencia
->Simbolo (
->Numero Entero 5
->Simbolo ,
->Numero Entero 6
->Simbolo )
->Salto de linea

        ->Resultado: 15625

PoW(5.6,3.33)
->Potencia
->Simbolo (
->Numero Decimal 5.6
->Simbolo ,
->Numero Decimal 3.33
->Simbolo )
->Salto de linea

        ->Resultado: 310.074556
```

Figura 18. Función Potencia con valores numéricos SIN signo

- Potencia combinada con otros operadores:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
6/PoW(3,4)+12-poW(6.6,2)/Pow(12,3.3)--3
->Numero Entero 6
->Operador /
->Potencia
->Simbolo (
->Numero Entero 3
->Simbolo ,
->Numero Entero 4
->Simbolo )
->Operador +
->Numero Entero 12
->Operador -
->Potencia
->Simbolo (
->Numero Decimal 6.6
->Simbolo ,
->Numero Entero 2
->Simbolo )
->Operador /
->Potencia
->Simbolo (
->Numero Entero 12
->Simbolo ,
->Numero Decimal 3.3
->Simbolo )
->Operador -
->Operador -
->Numero Entero 3
->Salto de linea

        ->Resultado: 14.988038
```

Figura 19. Función Potencia combinada con otros operadores

- Potencia de cadenas:

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
pow("hola", 0)
->Potencia
->Simbolo (
->Cadena hola
->Simbolo ,
 ->Numero Entero 0
->Simbolo )
->Salto de linea

        ->Resultado:

POW("hola", 3)
->Potencia
->Simbolo (
->Cadena hola
->Simbolo ,
 ->Numero Entero 3
->Simbolo )
->Salto de linea

        ->Resultado: holaholahola

PoW("hola", 4)
->Potencia
->Simbolo (
->Cadena hola
->Simbolo ,
 ->Numero Entero 4
->Simbolo )
->Salto de linea

        ->Resultado: holaholaholahola
```

Figura 20. Función Potencia con Cadenas de Texto

3.5 Declaración de Variables

Para mostrar de una mejor manera el funcionamiento de la Tabla de Símbolos cuando se realizan declaraciones de variables, se incluyó la función “Imprimir_Tabla” para mostrar como es que las variables se almacenas en la tabla de manera gráfica.

- Variables sin Inicializar (tipo_dato VARIABLE ;)

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
int var1;
->Tipo de dato entero
 ->Variable var1
->Simbolo ;
->Salto de linea
=================================================
|               TABLA DE SIMBOLOS                |
=================================================
| NOMBRE       | var1                           |
| TIPO DE DATO | int                            |
| VALOR        | 0                              |
-------------------------------------------------
double var2;
->Tipo de dato flotante double
 ->Variable var2
->Simbolo ;
->Salto de linea
=================================================
|               TABLA DE SIMBOLOS                |
=================================================
| NOMBRE       | var1                           |
| TIPO DE DATO | int                            |
| VALOR        | 0                              |
-------------------------------------------------
| NOMBRE       | var2                           |
| TIPO DE DATO | double                         |
| VALOR        | 0.000000                       |
-------------------------------------------------
string var3;
->Tipo de dato cadena de texto
 ->Variable var3
->Simbolo ;
->Salto de linea
=================================================
|               TABLA DE SIMBOLOS                |
=================================================
| NOMBRE       | var1                           |
| TIPO DE DATO | int                            |
| VALOR        | 0                              |
-------------------------------------------------
| NOMBRE       | var2                           |
| TIPO DE DATO | double                         |
| VALOR        | 0.000000                       |
-------------------------------------------------
| NOMBRE       | var3                           |
| TIPO DE DATO | char                           |
| VALOR        | (null)                         |
```

Figura 21. Declaración de Variables sin inicializar

- Inicialización de variables (tipo_dato VARIABLE = VALOR ;)

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
int var1 = 4;
->Tipo de dato entero
 ->Variable var1
 ->Operador =
 ->Numero Entero 4
->Simbolo ;
->Salto de linea
=================================================
|               TABLA DE SIMBOLOS                |
=================================================
| NOMBRE       | var1                             |
| TIPO DE DATO | int                              |
| VALOR        | 4                                |
-------------------------------------------------
string var3 = "hola mundo";
->Tipo de dato cadena de texto
 ->Variable var3
 ->Operador =
 ->Cadena hola mundo
->Simbolo ;
->Salto de linea
=================================================
|               TABLA DE SIMBOLOS                |
=================================================
| NOMBRE       | var1                             |
| TIPO DE DATO | int                              |
| VALOR        | 4                                |
-------------------------------------------------
| NOMBRE       | var3                             |
| TIPO DE DATO | char                             |
| VALOR        | hola mundo                       |
-------------------------------------------------
```

Figura 22. Declaración de Variables inicializadas 1

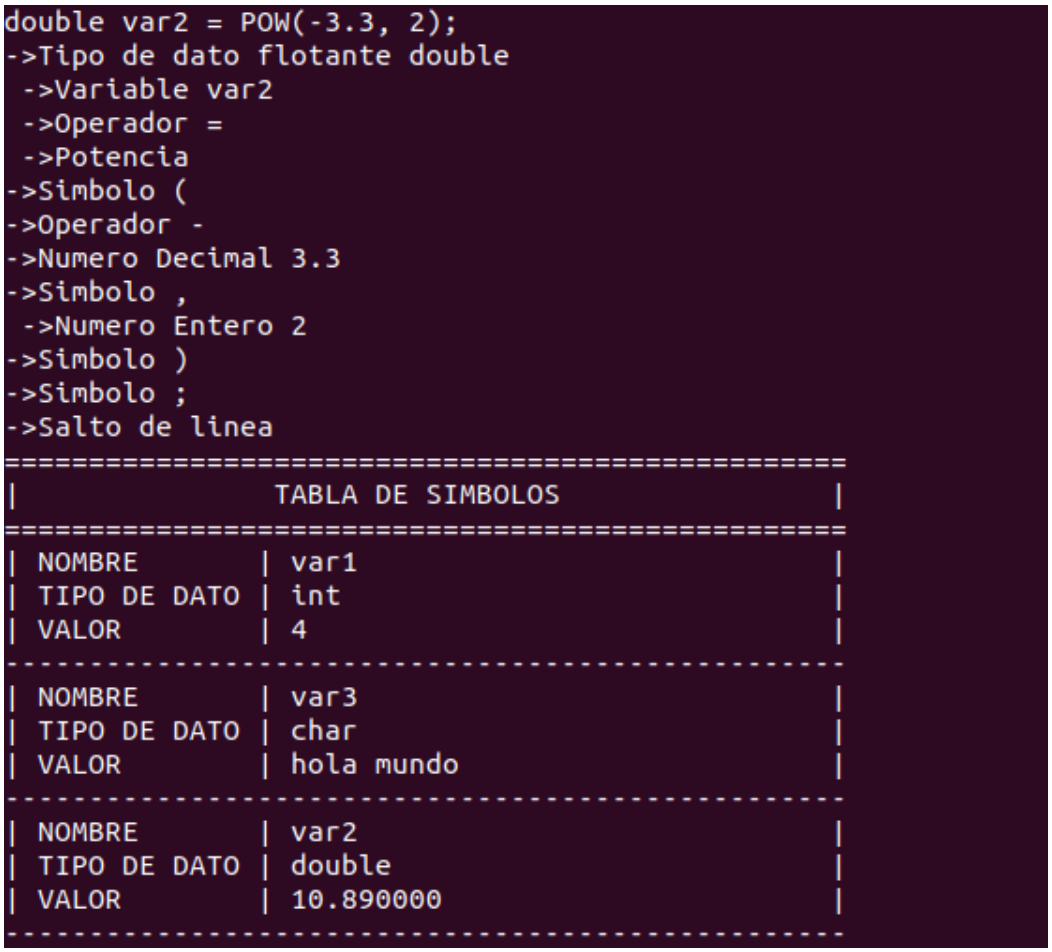

Figura 23. Declaración de Variables inicializadas 1

3.6 Asignación de Valores a Variables

Para mostrar de una mejor manera el funcionamiento de la Tabla de Símbolos cuando se realizan asignaciones de valores a variables, se incluyó la función “Imprimir_Tabla” para mostrar cómo es que las variables se almacenas en la tabla de manera gráfica.

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
int var1;
->Tipo de dato entero
 ->Variable var1
->Simbolo ;
->Salto de linea
================================================
|               TABLA DE SIMBOLOS               |
================================================
| NOMBRE       | var1                            |
| TIPO DE DATO | int                             |
| VALOR        | 0                               |
------------------------------------------------
var1 = 5;
->Variable var1
 ->Operador =
 ->Numero Entero 5
->Simbolo ;
->Salto de linea
================================================
|               TABLA DE SIMBOLOS               |
================================================
| NOMBRE       | var1                            |
| TIPO DE DATO | int                             |
| VALOR        | 5                               |
------------------------------------------------
var1 = pow(23,3)-456;
->Variable var1
 ->Operador =
 ->Potencia
->Simbolo (
->Numero Entero 23
->Simbolo ,
->Numero Entero 3
->Simbolo )
->Operador -
->Numero Entero 456
->Simbolo ;
->Salto de linea
================================================
|               TABLA DE SIMBOLOS               |
================================================
| NOMBRE       | var1                            |
| TIPO DE DATO | int                             |
| VALOR        | 11711                           |
------------------------------------------------
```

Figura 24. Asignación de Valores a Variables 1

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
string var3;
->Tipo de dato cadena de texto
 ->Variable var3
->Simbolo ;
->Salto de linea
=================================================
|               TABLA DE SIMBOLOS                |
=================================================
| NOMBRE       | var3                             |
| TIPO DE DATO | char                             |
| VALOR        | (null)                           |
-------------------------------------------------
var3 = "hola de nuevo";
->Variable var3
 ->Operador =
 ->Cadena hola de nuevo
->Simbolo ;
->Salto de linea
=================================================
|               TABLA DE SIMBOLOS                |
=================================================
| NOMBRE       | var3                             |
| TIPO DE DATO | char                             |
| VALOR        | hola de nuevo                    |
-------------------------------------------------
var3 = pOw("texto", -2);
->Variable var3
 ->Operador =
 ->Potencia
->Simbolo (
->Cadena texto
->Simbolo ,
 ->Operador -
->Numero Entero 2
->Simbolo )
->Simbolo ;
->Salto de linea
=================================================
|               TABLA DE SIMBOLOS                |
=================================================
| NOMBRE       | var3                             |
| TIPO DE DATO | char                             |
| VALOR        | otxetotxet                       |
-------------------------------------------------
```

Figura 25. Asignación de Valores a Variables 2

3.7 Validaciones

- Mensaje de error de variable no declarada

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
var1 = 12;
->Variable var1
 ->Operador =
 ->Numero Entero 12
->Simbolo ;
->Salto de linea
ERROR: Variable var1 no ha sido inicializada
var3 = "hola";
->Variable var3
 ->Operador =
 ->Cadena hola
->Simbolo ;
->Salto de linea
ERROR: Variable var3 no ha sido inicializada
```

Figura 26. Mensaje de error de variable no declarada

- Mensaje de error de operaciones incompatibles

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
2+"texto"
->Numero Entero 2
->Operador +
->Cadena texto
->Salto de linea
ERROR: Operacion de tipo "int" requiere de valores enteros, pero esta recibiendo
 un valor de tipo "string"


        ->Resultado: 2

int var2 = 3+"hola";
->Tipo de dato entero
 ->Variable var2
 ->Operador =
 ->Numero Entero 3
->Operador +
->Cadena hola
->Simbolo ;
ERROR: Operacion de tipo "int" requiere de valores enteros, pero esta recibiendo
 un valor de tipo "string"

->Salto de linea
================================================
|               TABLA DE SIMBOLOS               |
================================================
| NOMBRE       | var2                           |
| TIPO DE DATO | int                            |
| VALOR        | 3                              |
------------------------------------------------
```

Figura 27. Mensaje de error de operaciones incompatibles

- Mensaje de error de variable declarada 2 veces

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
int var1;
->Tipo de dato entero
 ->Variable var1
->Simbolo ;
->Salto de linea
================================================
|               TABLA DE SIMBOLOS               |
================================================
| NOMBRE       | var1                            |
| TIPO DE DATO | int                             |
| VALOR        | 0                               |
------------------------------------------------
int var1;
->Tipo de dato entero
 ->Variable var1
->Simbolo ;
->Salto de linea
ERROR: Variable var1 ya ha sido declarada previamente
================================================
|               TABLA DE SIMBOLOS               |
================================================
| NOMBRE       | var1                            |
| TIPO DE DATO | int                             |
| VALOR        | 0                               |
------------------------------------------------
double var1;
->Tipo de dato flotante double
 ->Variable var1
->Simbolo ;
->Salto de linea
ERROR: Variable var1 ya ha sido declarada previamente
================================================
|               TABLA DE SIMBOLOS               |
================================================
| NOMBRE       | var1                            |
| TIPO DE DATO | int                             |
| VALOR        | 0                               |
------------------------------------------------
```

Figura 26. Mensaje de error de variable declarada 2 veces

- Mensaje de error de asignaciones incompatibles

```
zekyal@Zekyal:~/Documentos/Compiladores/Practica 6. Interprete$ ./a.out
int var1 = "hola";
->Tipo de dato entero
 ->Variable var1
 ->Operador =
 ->Cadena hola
->Simbolo ;
->Salto de linea
ERROR: Variable de tipo "int" requiere de valor de tipo entero, pero esta recibi
endo un valor de tipo "string"

string var2;
->Tipo de dato cadena de texto
 ->Variable var2
->Simbolo ;
->Salto de linea
================================================
|               TABLA DE SIMBOLOS               |
================================================
| NOMBRE       | var2                          |
| TIPO DE DATO | char                          |
| VALOR        | (null)                        |
------------------------------------------------
var2 = 13+2;
->Variable var2
 ->Operador =
 ->Numero Entero 13
->Operador +
->Numero Entero 2
->Simbolo ;
->Salto de linea
ERROR: Variable de tipo "string" requiere de una cadena de texto, pero esta reci
biendo un valor de tipo "int"
```

Figura 26. Mensaje de error de asignaciones incompatibles

4 Conclusión:

Los Intérpretes son de las herramientas más importantes alguna vez creadas, al ser estas la base esencial de prácticamente cualquier lenguaje de programación existente, al ser este el que le da un significado a cada una de las instrucciones que conforman a los lenguajes de programación, y a partir de dicho significado es que los compiladores pueden llevar a cabo las acciones especificadas por el o los programadores, y de esta manera poder lograr la creación de software o programas de una manera efectiva y relativamente más rápida y efectiva que usando métodos más tradicionales tales como el uso de lenguaje máquina.

Adicionalmente y relacionado a la práctica, el uso de herramientas como Flex y Bison, que nos permiten crear analizadores sintácticos, semánticos y léxicos, resultan de gran ayuda no solo para entender a mayor detalle como es el funcionamiento de lenguajes de programación que se usan día a día en la carrera o en la vida profesional, si no que también a partir de estos se pueden crear nuevos lenguajes de programación.

5 Referencias:

- Aho, A., 2011. Compiladores. 2da edición. Pearson Educación de México, SA de CV, pp.140-144.
- Levine, J., 2009. Flex & Bison. 1ra edición. O'Reilly Media Inc., pp.19-79.
- Donnelly, C. and Stallman, R., 1999. Bison 1.27. [online] es.tldp.org. Available at: <http://es.tldp.org/Manuales-LuCAS/BISON/bison-es-1.27.html> [Accedido el 20 de diciembre de 2021].
- Paxson, V., 1995. Flex - un generador de analizadores léxicos. [online] es.tldp.org. Disponible en: <http://es.tldp.org/Manuales-LuCAS/FLEX/flex-es-2.5.html#SEC5> [Accedido el 20 de diciembre de 2021].