***Icono

Descripción generada automáticamente***

***Universidad Tecnológica Nacional***

**Sintaxis y Semántica de los Lenguajes**

**2023**

**Curso:** K2055

**Profesora:** Roxana Leituz

**Fecha Estipulada de Entrega: 17/11**

**Grupo N° 32**

|  |  |
| --- | --- |
| **Alumno** | **Legajo** |
| **Gonzalo Menese** | **173.087-3** |
| **Jean Pierre Luna Rumualdo** | **206.431-5** |
|  |  |
|  |  |

Documentación del TP Flex y Bison 2

Ejemplos 5

Documentación del TP Flex y Bison

Nuestro grupo decidió utilizar una máquina virtual con una ISO de Ubuntu para resolver el TP ya que se nos hacía más fácil la instalación de Flex y Bison en un entorno de Linux.

Empezamos creando un archivo(./correr) , más que nada para no estar escribiendo el mismo código todo el rato y que la ejecución del programa sea sencilla, cuando lo ejecutemos este va a generar el analizador léxico , sintáctico y compilarlos.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Luego creamos un archivo Flex(que lo llamamos prueba.l) , donde definiremos todos los lexemas que pertenecen a nuestro lenguaje en la sección definiciones .

Patrón de fondo

Descripción generada automáticamente con confianza baja

En la sección de reglas definimos las acciones a tomar en caso de capturar un token, en nuestro caso utilizamos returns para que Bison pueda utilizar los tokens para definir las reglas gramaticales que definiremos nosotros para nuestro lenguaje micro .

Texto

Descripción generada automáticamente

Previamente creamos una variable “errorlexico” inicializada en 0 (si esta variable no presentó cambios, significa que no existieron errores léxicos en la ejecución), los lexemas o caracteres desconocidos los tratamos con un printf para indicar que existe un error léxico utilizando yytext para agarrar el token actual y cambiamos la variable “errorlexico” a 1 indicando que existió algún error léxico.





Luego creamos el archivo Bison(prueba.y) donde indicamos los tokens que vamos a utilizar luego.



Definimos el tipo de dato Entero para utilizarlo en algunos no terminales para así poder realizar luego un chequeo semántico. Algunos tokens los definimos con el tipo de dato Entero por la misma razón.

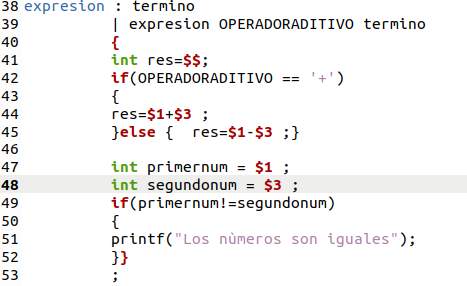
Gráfico

Descripción generada automáticamente



Definimos las reglas gramaticales utilizando producciones con recursividad a izquierda en nuestros no terminales y además definiendo los terminales.

En el no terminal “expresión”, creamos una rutina semántica que chequea que los valores sean iguales (a modo de ejemplo para realizar una rutina semántica cualquiera), y hacemos las operaciones de suma y resta según corresponda.



Además, en el main chequeamos la salida de yyparse para saber si hubieron errores sintácticos con un mensaje de printf, y si no existió ninguno, indicamos que el análisis fue exitoso y no hubo errores sintácticos. También chequeamos si hubo errores léxicos si la variable “errorlexico” presento un cambio, significando que hubo algún error léxico.

Texto

Descripción generada automáticamente

Ejemplos

El código lo ejecutamos de la siguiente forma



textoprueba con errores léxicos

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Aunque nuestro archivo(textoprueba) no presente errores sintácticos, va a presentar errores léxicos porque no reconoce el carácter “?” en nuestro lenguaje micro

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

El código presenta errores sintácticos leer espera un token “ID” y le estamos pasando un token “inicio” por lo cual nos va a tirar error

Texto

Descripción generada automáticamente

Si el código no presenta ni errores léxicos ni sintácticos , entonces va a compilarse correctamente y nos mostrará un mensaje

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente