Universidad San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Organización de Lenguajes y Compiladores 1
Segundo Semestre 2022
Ing. Manuel Haroldo Castillo Reyna
Ing. Kevin Adiel Lajpop Ajpacaja
Ing. Mario Bautista Fuentes
Aux:
Mynor René Ruiz Guerra
Moises Gonzalez Fuentes
Jose Fernando Guerra Muñoz

Proyecto 1

Objetivos

Objetivos Generales

- Entender el funcionamiento de un compilador en sus dos primeras fases.
- Entender los conceptos de análisis.
- Realizar el uso correcto de herramientas de análisis.
- Comprender los tipos de análisis sintáctico.

Objetivos Específicos

- Establecer y programar el proceso de reconocimiento de análisis léxico y sintáctico.
- Reforzar el concepto y análisis del árbol sintáctico abstracto
- Realizar un esquema que permita recuperar el flujo sintáctico de los errores y reportar de forma descriptiva.
- Realizar la construcción correcta de un árbol sintáctico.
- Entender las diferencias sintácticas entre lenguajes.
- Utilizar correctamente herramientas de análisis léxico y sintáctico como JLex y Cup.

Descripción General

Un cliente ha solicitado a usted un Pseudo-Parser para que el nuevo personal que no conoce los lenguajes de Python y Golang, pueda aplicar sus conocimientos en pseudocódigo y utilizando una aplicación, traducir este código y ver cómo se comportan las diferentes sintaxis de cada uno de los lenguajes ya que para cada uno existen diferentes características, por lo cual, se le solicita a usted que a partir de sus conocimientos en compiladores haga una implementación de las primeras 2 fases de un compilador y ejecute una traducción con la entrada de pseudocódigo a Python y Golang. Además que se pueda visualizar el diagrama de flujo resultante de la entrada, para esto debe utilizar las herramientas de JFLEX y CUP e implementar su solución en el lenguaje de programación JAVA.

La solución debe ser intuitiva y fácil de utilizar, mostrando los resultados y reportes, tanto de errores como el diagrama de flujo, lo más estructurado posible, además como parte adicional de la aplicación el cliente solicita que se pueda visualizar el árbol de análisis sintáctico y este se pueda exportar a PDF, así como las traducciones y demás reportes.

Flujo del programa

El programa debe poseer una interfaz gráfica que permita un panel de entrada de texto tipo editor donde se podrá escribir código en lenguaje pseudocódigo y deberá dar las opciones de ver la traduccion de python o la de golang, al elegir cualquiera de estas opciones de preferencia se deberá abrir un modal donde se muestre la traducciones del

lenguaje y la opción de guardar en un archivo de salida (.py o .go según sea la traducción) o simplemente permita copiar el código para su posterior ejecución en un compilador externo (Golang Playground o Python Playground), para la sección de reportes, de existir errores se debe mostrar un modal (ventana emergente) que nos muestre los errores en el formato posteriormente explicado (sección de reportes) y que nos permita almacenar el archivo o inyectar el contenido del reporte en la caja de texto del código (al hacer esto automáticamente se debe guardar el archivo o el código que estaba anteriormente en la caja de texto) para su posterior solución, para el reporte de Diagrama de flujo se debe guardar esto en un pdf que se pueda visualizar. Adicionalmente se debe contar con controles de archivos (abrir, guardar).

Tipo de datos

Palabra reservada	Descripción	Ejemplos
Número	1) Conjunto de dígitos que tienen una única vez el carácter punto dentro del conjunto de caracteres. 2) conjunto de solo dígitos.	10.02 11.1 56.0 105.152 4568 5 94
Cadena	Conjunto de caracteres dentro de comillas dobles.	“Esta es una cadena” “hola mundo” “organización de lenguajes y compiladores 1”
Boolean	Tipo de dato verdadero o falso.	Verdadero Falso
Carácter	Un carácter dentro de una comilla simple. Es posible ingresar el código ascii (únicamente se usarán los ascii de caracteres del alfabeto).	‘o’ ‘L’ ‘${70}’ = ‘F’

Operaciones básicas

Suma

Esta operación usa el carácter “+”

<Operando1> +<operando2>

Resta

Esta operación usa el carácter “-”

<Operando1> - <operando2>

Multiplicación

Esta operación usa el carácter “*”

<Operando1> * <operando2>

División

Esta operación usa el carácter “/”

<Operando1> /<operando2>

Potencia

Esta operación usa la palabra reservada “potencia”

<Operando1> potencia [<operando2>]

Módulo

Esta operación usa la palabra reservada “modulo”

<Operando1> mod <operando2>

Paréntesis

Se usará para asociar un conjunto de operaciones aritméticas.

(<Conjunto de operaciones aritmeticas>)

Operadores relacionales

Operador	Ejemplo
Mayor	Expresion mayor Expresion
Menor	Expresion menor Expresion
Mayor o igual que	Expresion mayor_o_igual Expresion
Menor o igual que	Expresion menor_o_igual Expresion
Igual	Expresion es_igual Expresion
Diferente	Expresion es_diferente Expresion

Operadores lógicos

Operador	Ejemplo
Or	Expresion or Expresion
And	Expresion and Expresion
Not	not Expresion

Sintaxis del pseudocódigo

Las características de este lenguajes están descritas a continuación:

Global

Esta instrucción permite englobar todo el código del archivo dentro de palabras reservadas.

```
inicio
        //todo el código…
```

```
fin
```

Comentarios

Esta instrucción puede estar en cualquier parte del archivo. Existen dos tipo de comentarios: de una línea y de dos o más líneas

Los comentario de una línea iniciaran con los caracteres “//”, el cual indicará que en adelante todos los caracteres serán considerados como comentarios y el fin del comentario estará delimitado por un salto de línea.

Los comentarios de más líneas estarán encerrados dentro de los caracteres “/*” y “*/”

```
//comentario de una línea


/* *******comentario
de más
líneas */

/* este también es un comentario */
```

Declaración

Esta instrucción permite ingresar variables dentro del flujo del código. Para ingresar el nombre se necesita reconocer la palabra reservada “ingresar” y el tipo de dato. A continuación un ejemplo de una declaración:

```
ingresar <nombre variable> como <tipo de dato> con_valor <expresión>;
```

Nota: es posible realizar declaraciones múltiples, para ello es necesario ingresar una lista de nombres, separadas por comas.

```
ingresar <lista de nombre> como <tipo de dato> con_valor <expresión>;

<lista de nombre>= <nombre1>,  <nombre2>,  <nombre3> …
```

Para declarar un nombre de una variable es obligatorio iniciar con un carácter guión bajo y al finalizar también el mismo carácter.

Ejemplos:

myname
lastname
phone1
phone2last

Asignación

Esta instrucción cambia el valor de una variable determinada y la actualiza con el valor de una expresión, su estructura consta del nombre de la variable y una expresión.

```
<nombre variable> -> <expresión>;
```

Nota: es posible realizar múltiples asignaciones de la siguiente forma: antes del carácter “->” se ingresa el nombre de las variables separadas por el carácter coma.

```
<nombre1> , <nombre2> , <nombre3> -> <expresión>;
```

Condicionales

Condicional “Si”

Esta es una instrucción que permite ejecutar un bloque de instrucciones cuando una condición es válida. La instrucción necesita una condición para ejecutar un bloque asignado, cuando la condición es falsa se ejecuta otro bloque de instrucciones. Es posible que el bloque de instrucciones que ejecuta una condición falsa, sea opcional. Es obligatorio que ingrese una condición en esta instrucción.

```
//instrucción con bloque de instrucciones con condición verdadera
si <condición> entonces
    <instrucciones>
fin_si

//instrucción con bloque de instrucciones con condición verdadera y falsa
si <condición>
    <instrucciones>
de_lo_contrario
    <instrucciones>
fin_si
```

Caso especial: se usará la palabra reservada “o_si” cuando una condición es falsa y se trata de volver hacer otra validación con otra condición. Este tipo de ejecución puede ser opcional o repetirse una o muchas veces. A continuación se muestra un ejemplo de este caso especial

```
si <condición>
        <instrucciones>
o_si <condición_nueva> entonces
        <instrucciones>
de_lo_contrario
        <instrucciones>
fin_si


si <condición>
        <instrucciones>
o_si <condición_nueva> entonces
        <instrucciones>
o_si <condición_nueva> entonces
        <instrucciones>
de_lo_contrario
        <instrucciones>
fin_si
```

Selección múltiple

Este tipo de condición permite ejecutar una lista de instrucciones en base a un valor ingresado, existen varias opciones posibles y cuando el valor coincida con una de las opciones se ejecuta un conjunto de instrucciones. Existe una palabra reservada “de_lo_contrario” para ejecutar automáticamente cuando la lista de opciones no se cumple, pero es de forma opcional.

```
segun <valor> hacer

        ¿ <valor 1>  ? entonces
                        <instrucciones para valor 1>
        ¿  <valor 2>  ? entonces
                        <instrucciones para valor 2>
        ¿  <valor 3>  ? entonces
                        <instrucciones para valor 3>
        de_lo_contrario entonces
```

```
                                        <instrucciones por default>
    fin_segun
```

Nota: <valor> puede ser una variable o una expresión aritmética.

Ciclos

Este tipo de instrucción permite realizar tareas repetitivas en base a una condición dada. Cada una de las instrucciones necesita de una condición de forma obligatoria.

Ciclo “Para”

Este ciclo ejecuta un conjunto de instrucciones, con un límite de repeticiones. Es necesario ingresar un valor inicial y también un valor final, el valor final es el que le indica al ciclo, en momento para terminar de realizar las repeticiones. Otro de los elementos necesarios en este ciclo es el número de pasos que realizará entre cada repetición. Cuando el ciclo no tiene definido el número de pasos, se tomará como defecto el incremento en 1. Es posible que la lista de instrucciones esté vacía.

```
    //Estructura de ciclo para con salto no definido
    para <variable> -> <valor inicial> hasta <valor final> hacer
          <instrucciones>
    fin_para

    para <variable> -> <valor inicial> hasta <valor final> hacer
         //null
    fin_para


    //Estructura de ciclo para con salto definido
    para <variable> -> <valor inicial> hasta <valor final> con incremental
    <Valor del salto> hacer
           <instrucciones>
    fin_para
```

Nota: <valor inicial> y <valor final> puede ser considerado como el nombre de una variable, un número o una expresión aritmética.

Ciclo “Mientras”

Este ciclo ejecuta un conjunto de instrucciones sin límite definido. Para poder realizar una repetición es necesario que exista una condición. Es posible que la lista de instrucciones esté vacía.

```
mientras <condicion> hacer
      <intrucciones>
fin_mientas


mientras <condicion> hacer
     //null
fin_mientas
```

Ciclo “Repetir hasta”

Este ciclo ejecuta un conjunto de instrucciones sin límite definido. A diferencia del ciclo anterior, este ciclo realiza una única repetición sin restricción. Para poder realizar las demás repeticiones es necesario que exista una condición. Es posible que la lista de instrucciones esté vacía.

```
repetir
     <instrucciones>
hasta_que <condición>


repetir
    //null
hasta_que <condición>
```

Retorno

Esta instrucción está encargada de devolver un valor específicamente. Esta instrucción puede reconocer una <condición> , un número o una <expresión aritmética>.

```
retornar <expresión aritmética> ;
retornar <condición>;
retornar <número>;
```

Método

Esta instrucción permite agrupar un conjunto de instrucciones y asignarles un nombre para identificarlo dentro del contenido del archivo. No necesita una instrucción de “Retorno” y si en caso es reconocido este tipo de instrucción, debe reportar dicho error.

```
metodo <nombre>
        <instrucciones>
fin_metodo
```

Nota: es posible agregar parámetros al método, estos parámetros tendrá definido el tipo de dato y su respectivo nombre. Cada uno de los parámetros estará separado por un carácter coma.

```
metodo <nombre>  con_parametros (<lista de parametros>)
        <instrucciones>
fin_metodo


<lista de parametros> = <nombre> <tipo de dato> , <nombre> <tipo de dato> ,
<nombre> <tipo de dato> …
```

Funciones

Esta instrucción permite agrupar un conjunto de instrucciones y asignarles un nombre para identificarlo dentro del contenido del archivo. Esta instrucción si es posible reconocer una instrucción de “Retorno" en su estructura.

```
funcion <nombre> <tipo dato>
        <instrucciones>
fin_funcion
```

Nota: es posible agregar parámetros a la instrucción, estos parámetros tendrá definido el tipo de dato y su respectivo nombre. Cada uno de los parámetros estará separado por un carácter coma.

```
funcion <nombre> <tipo dato>   con_parametros (<lista de parámetros>)
        <instrucciones>
fin_funcion



<lista de parametros> = <nombre> <tipo de dato> , <nombre> <tipo de dato> ,
<nombre> <tipo de dato> …
```

Llamada de funciones y métodos

Este tipo de instrucción realiza la ejecución de un método o función, para poder realizarlo es necesario ingresar el identificador de la función o método y la lista de parámetros necesarios.

```
//ejecutar sin parámetros
ejecutar <identificador>();


//ejecutar con parámetros
ejecutar <identificador>(<Lista de parámetros>);
```

Nota: En <Lista de parámetros> los parámetros están separados por el carácter coma. Y Cada

Impresión

Esta instrucción muestra el contenido de una expresión o valor de una variable. Para poder utilizarla es necesario una expresión o valor de una variable. Al terminar de realizar la impresión se genera un salto de línea por defecto.

```
imprimir <expresión>; //impresión sin salto de línea
imprimir_nl <expresión>; //impresión con salto de línea
```

Salida en Golang

Tipo de datos

Pseudocódigo	GOLANG
Cadena	string -> “cualquier”
Caracter	byte -> ‘F’ ‘G’
Boolean	bool -> true, false
Numero	int -> 5,6,7,8||float64 -> 15.25, 2.5

Operaciones Básicas

Pseudocódigo	GOLANG
Suma	expresion + expresion
Multiplicación	expresión * expresión
Resta	expresion - expresion
Potencia	math.Pow(float64(expresión), float64(expresión)) nota: se debe agregar la importación “math” ej: import(“math”)
Asociación	(expresión)
Módulo	expresion % expresion

Operadores relacionales

Operador	Ejemplo
Mayor	Expresion > Expresion
Menor	Expresion <Expresion

Mayor o igual	Expresion >= Expresion
Menor o igual	Expresion <= Expresion
Igual	Expresion == Expresion
Diferente	Expresion != Expresion

Operadores lógicos

Operador	Ejemplo
Or	Expresion || Expresion
And	Expresion && Expresion
Not	! Expresion

Sintaxis del Lenguaje

Global

```
package main

import(
	…importaciones
)

func main(){
	…instrucciones
}
```

Comentarios

```
//comentario de una línea


/* *******comentario
de más
líneas */

/* este también es un comentario */
```

Declaración

```
var <nombre variable> <tipo de dato> = <expresión>
```

Declaraciones múltiples

```
var <lista de  nombres> <tipo dato> = <lista de valores>

var <nombre var1>  = <valor1>
var <nombre var2> = <valor2>
```

Pseudocódigo	GOLANG
ingresar var1, var2 como entero con_valor 5+5;	var var1, var2 int = 5+5, 5+5 var var1 = 5+5 var var2 = 5+5

Asignaciones

```
<nombre variable> = <expresión>

<lista variables> = <lista valores>
```

Pseudocódigo	GOLANG
var1, var2, var3 -> 5+5;	var1 = 5+5 var2 = 5+5 var3 = 5+5

Condicionales

IF

```
if <condicion> {

}else if <condicion> {

}else{

}

Ej:
```

```
if x == 50 {
        …instrucciones
} else if x == 100 {
        …instrucciones
} else {
        …instrucciones
}
```

Selección Múltiple

(Switch Case)

```
switch <variable> {
    case <expresión>:
        …instrucciones
    case <expresión>:
        …instrucciones
     default:
        …instrucciones
}

Ej:

switch <valor> {
    case <valor 1>:
        …instrucciones para valor 1
    case <valor 2>:
        …instrucciones para valor 2
     default:
        …instrucciones por default
}
```

Ciclos

For

```
for <asignacion>; <expresión>; <asignacion> {
    …instrucciones
}

Ej:

for i := 0; i < 5; i++ {
        fmt.Println(i)
}

for i := 0; i < 5; i=i+2 {
        fmt.Println(i)
}
```

While

```
for true {
    if !( <expresión lógica> ){
        break
    }

    …instrucciones
}
```

Pseudocódigo	GOLANG

mientras i<15 && i>0 hacer
  <instrucciones>
fin_mientas
	for true {
	  if !(i<15 && i>0) {
	  break
	  }
	  <instrucciones>
	}

Repetir Hasta

(do while)

```
for true {
    …instrucciones
    if ( <expresión lógica> ){
        break
    }
}
```

Pseudocódigo	GOLANG

repetir
  <instrucciones>
hasta_que (i>10)
	for true {
	  <instrucciones>
	  if (i>10) {
	  break
	  }
	}

Retorno

```
return <expresión>
```

Método

```
func <nombre> () {
    …instrucciones
}

func <nombre> (...params) {
    …instrucciones
}
```

Funciones

```
func <nombre> () <tipo dato> {
    … instrucciones
}

func <nombre> (...params) <tipo dato> {
    …instrucciones
}
```

Parámetros de funciones y métodos

```
<nombre> <tipo dato>
<nombre> <tipo dato>, <nombre> <tipo dato>

Ej:

func funcion(i int, j int) {
    …instrucciones
}
```

Llamada de funciones y métodos

```
<identificador>()
<identificador>(...parámetros)
```

Parámetros de llamada

```
  <expresion>, <expresion>, <expresion>

  Ej:
  llamada(5, true, “parámetro”, “A”)
```

Impresión

```
  fmt.Print(<expresion>)
```

fmt.Println(<expresion>)

nota: se debe agregar la importacion “fmt”

ej:
```
import(
    “fmt”
)
```

Salida en Python

Se debe tomar en cuenta que python utiliza la indentación para delimitar la estructura permitiendo establecer bloques de código. No existen comandos para finalizar las líneas ni llaves con las que delimitar el código. Los únicos delimitadores existentes son los dos puntos (:) y la indentación del código.

Tipo de datos

Pseudocódigo	**Python**
Cadena	str-> “cualquier”
Carácter	str-> ‘F’ ‘G’
Boolean	bool -> true, false
Número	int -> 5,6,7,8||float -> 15.25, 2.5

Operaciones Básicas

Pseudocódigo	**Python**
Suma	expresion + expresion
Multiplicación	expresión * expresión
Resta	expresion - expresion
Potencia	expresión ** expresión
Asociación	(expresión)
Módulo	expresion % expresion

Operadores relacionales

Operador	Ejemplo
Mayor	Expresion > Expresion
Menor	Expresion <Expresion
Mayor o igual	Expresion >= Expresion
Menor o igual	Expresion <= Expresion
Igual	Expresion == Expresion
Diferente	Expresion != Expresion

Operadores lógicos

Operador	Ejemplo
Or	Expresion or Expresion
And	Expresion and Expresion
Not	not Expresion

Sintaxis del Lenguaje

Global

```
def main():
    print("python main function")


if __name__ == '__main__':
    main()
```

Comentarios

```
#comentario de una línea

'''   comentario
      de múltiples
      líneas        '''

“””   también es un
      comentario de
      múltiples líneas  “””
```

Declaración y asignación

```
<nombre variable>  = <expresión>
```

Declaraciones múltiples

```
 <lista de  nombres> = <lista de valores>

Ejemplo:
    a, b, c, d = 4, "geeks", 3.14, True
    a = 4
    b = “geeks”
    c = 3.14
    d = True
```

Ejemplos:

Pseudocódigo	**Python**
ingresar var1, var2 como entero con_valor 5+5; var1-> 5+5; var1, var2, var3 -> 5+5;	var1, var2 = 5+5, 5+5 var1 = 5+5 var2 = 5+5 var1, var2, var3 = 5+5, 5+5, 5+5

Condicionales

IF

```
if primera_condicion:
      ejecutar sentencia
elif segunda_condicion:
```

```
        ejecutar sentencia
    else:
        ejecutar sentencia si todas las condiciones previas son falsas

    Ejemplo:

    if x == 2:
        bloque de instrucciones
    elif x == “Hello” :
        bloque de instrucciones
    else:
        bloque de instrucciones
```

Selección Múltiple

(Switch Case)

```
    if condicion_case:
        bloque de instrucciones
    elif condicion_case:
        bloque de instrucciones
    else:
        ejecutar sentencia si todas las condiciones previas son falsas


    Ejemplo:
    if x == 1:
        bloque de instrucciones
    elif x == 2 :
        bloque de instrucciones
    else:
        #(default)#
        bloque de instrucciones
```

Ciclos

FOR

```
    for <variable> in <expression>:
        bloque de instrucciones

    for <variable> in range(<Número inicial >,<Número final >):
        bloque de instrucciones

    for <variable> in range(<Número inicial >,<Número final >):
        bloque de instrucciones
```

Ejemplo:

```
for x in “palabra”:
    bloque de instrucciones

for x in range(6):
    bloque de instrucciones

for x in range(1,5):
    bloque de instrucciones
```

WHILE

```
while <expresión>:
    bloque de instrucciones
```

Pseudocódigo	**Python**
mientras i<15 && i>0 hacer <instrucciones> fin_mientas	while (i<15 && i>0): <instrucciones>

Repetir Hasta

(do while)

```
valor = true

while valor == true:
    bloque de instrucciones

    if valor== false:
        break
```

Pseudocódigo	**Python**

repetir
  <instrucciones>
hasta_que (i>10)
	valor = True
	
	while valor == True:
	  print(“una pasada”)
	  valor = False
	
	  if valor== False:
	  break

Retorno

```
return <expresión>
```

Método

```
def <nombre> () :
    bloque de instrucciones


def <nombre> (params):
    bloque de instrucciones
```

Funciones

```
def <nombre> ():
    bloque de instrucciones
    return <expresión>

def <nombre> (params):
    bloque de instrucciones
    return <expresión>
```

Parámetros de funciones y métodos

```
<nombre> <tipo dato>
<nombre> <tipo dato>, <nombre> <tipo dato>

Ejemplo:

def funcion(var1, var2) :
    bloque de instrucciones
```

Llamada de funciones y métodos

```
<identificador>()
<identificador>(parámetros)
```

Parámetros de llamada

```
<expresion>, <expresion>, <expresion>
```

Ejemplo:

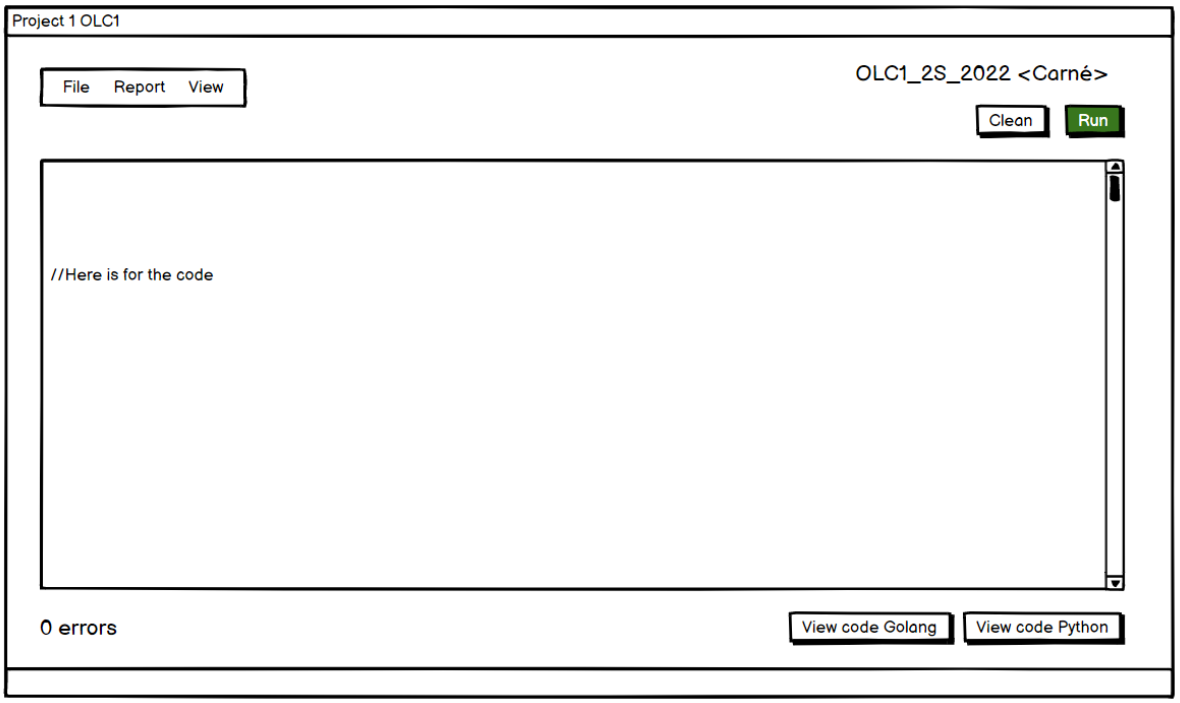
llamada(5, true, “parámetro”, “A”)

Impresión

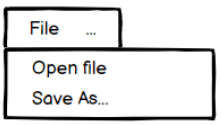
print(<expresión>)

Ejemplo:
print(“Esta es una salida”)

Interfaz gráfica



Menú Archivo



Abrir archivo: El usuario puede seleccionar un archivo [*.olc] de su directorio de archivos y el contenido se traslada al área de texto.

Guardar como: Guardara el contenido del área de texto en un archivo [*.olc] con una ruta especificada por el usuario.

Menú Reporte

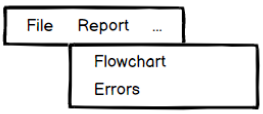
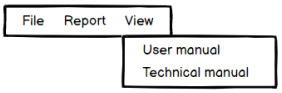


Diagrama de flujo: Mostrará al usuario la imagen con la ayuda del visualizador de archivos del dispositivo.

Errores: Muestra la lista de errores en un formato de tabla. Es posible mostrar una tabla con la ayuda de un archivo en html o gráfica. Los errores tienen que tener el detalle de errores: numeración del error, tipo de error, lexema, descripción.

Menú Ver



Este menú mostrará los manuales de usuario y manual técnico automáticamente al usuario, puede usar visualizador de pdf del dispositivo.

Botón Run

Este botón permite al usuario ejecutar el flujo del programa. Cuando el archivo de entrada tiene un error léxico o sintáctico, automáticamente debe desplegar una copia del archivo de entrada con todos los errores incrustados en el archivo. Crea el archivo en el dispositivo y abre automáticamente el archivo. A continuación un ejemplo de un error incrustado.

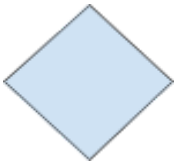
```
//Archivo original
…
 _nombreCurso_  “Organización de lenguajes y compiladores”;
 _nombreUniversidad_  -> “San Carlos de Guatemala”;
 …


//Archivo con error incrustado
…
 _nombreCurso_  “Organización de lenguajes y compiladores”;
                ^
                |
```

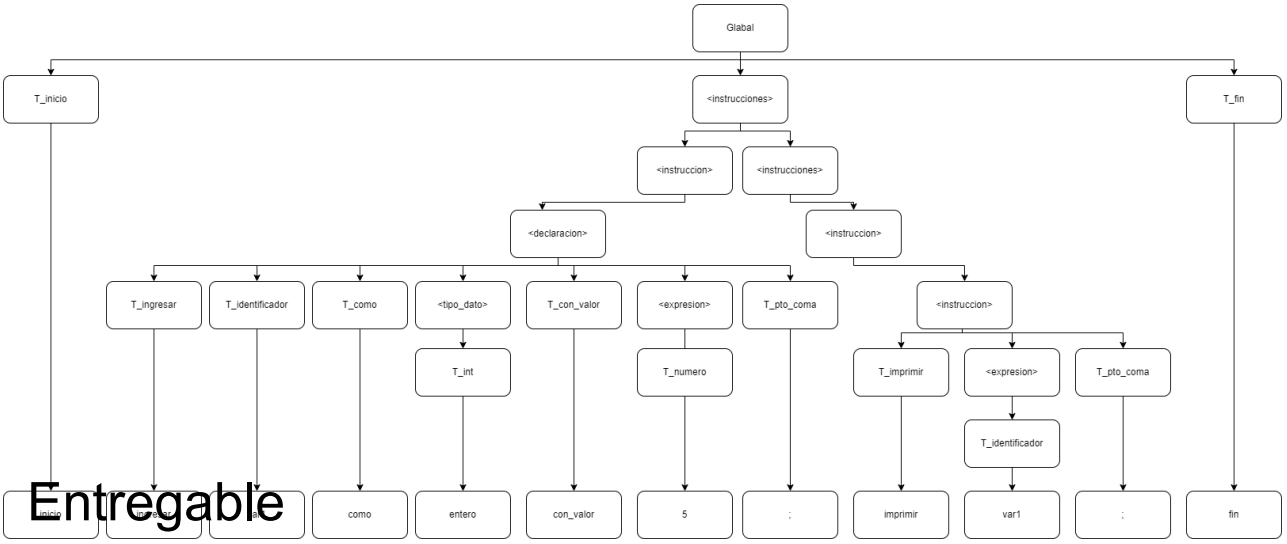
```
                    |
          Falta símbolo de asignacion | Aqui esta el error.
_nombreUniversidad_  -> “San Carlos de Guatemala”;
…
```

Reportes

Nomenclatura para diagrama de flujo

	Símbolo	Función
Terminal		Representa el inicio o fin de un flujo.
Entrada/salida		Representa cualquier tipo de instrucción que manipula los datos en memoria.
Proceso		Representa cualquier tipo de instrucción que realice una operación donde pueda originar un cambio de valor, formato o posición de la información almacenada.
Decisión		Representa una operación lógica o comparación entre datos y en función del resultado de la misma determina cuál de los distintos caminos alternativos del programa debe seguir
Conector misma página		Se utiliza para enlazar 2 partes cualesquiera de un programa a través de un conector de salida y otro conector de entrada.
Indicador de dirección o línea de flujo		Indica el sentido de la ejecución de las operaciones
Salida		Se utiliza para mostrar datos o resultados.

Árbol Sintáctico



Entregable

- Código Fuente del proyecto.
- Manuales de Usuario.
- Manual Técnico y al final agregar el link al repositorio privado de Github en donde se encuentra su proyecto.
- Archivo de Gramática para la solución (El archivo debe de ser limpio, entendible y no debe ser una copia del archivo de CUP).

Restricciones

1. Lenguajes de programación a usar: **Java**
2. Herramientas a de análisis léxico y sintáctico: **JFlex/CUP**
3. El proyecto es **Individual**
4. Para graficar se puede utilizar cualquier librería (Se recomienda **Graphviz**).
5. Copias completas/parciales de: código, gramática, etc. serán merecedoras de una nota de 0 puntos, los responsables serán reportados al catedrático de la sección y a la Escuela de Ciencias y Sistemas.
6. El resultado de la traducción será compilado en un compilador on-line del lenguaje correspondiente para validar que la traducción sea correcta.
7. Para tener derecho a calificación se debe de tener interfaz gráfica (no programas solamente en consola).
8. La calificación tendrá una duración de 30 minutos, acorde al programa del laboratorio.
9. Se usará la plataforma **GitHub** para llevar a cabo el control de versiones del proyecto. Crear el repositorio con el título “OLC1-<carné>” y crear una carpeta llamada “Proyecto 1”. Posteriormente se le compartirá el usuario del auxiliar y deberá agregar rol como contribuidor.

10. Las salidas(código traducido) serán ejecutadas en un **compilador web**, para verificar la correcta traducción y posterior calificación.

Fecha de entrega

Sábado ***17 de Septiembre*** de 2022.

La entrega será por medio de la plataforma **UEDI**, si existieran inconvenientes con la plataforma se utilizará Classroom previamente indicado con su respectivo auxiliar.

Entregas fuera de la fecha indicada, no se calificarán.

Se le calificará del commit realizado.