Documentación proyecto

Estudiantes:
Sebastian Granados Barrantes
Jousten Blanco Fonseca

Universidad Nacional sede regional Brunca

Ingeniería en sistemas de la Información

Paradigmas de programación

Profesor:
Josías Ariel Chaves Murillo

II Ciclo 2024

Nombre del lenguaje de programación:

Jäger Script

Planteamiento del problema :

En el presente trabajo se realizará un lenguaje de programación desde cero, definiendo la sintaxis y estructura semántica del lenguaje, analizando las 3 partes básicas o esenciales de cualquier lenguaje de programación que sería la semántica, sintáctica y léxica.

Se debe especificar mínimo 2 tipos de datos compuestos y 5 tipos de datos complejos, así como estructuras de control como condicionales simples y múltiples(**switch**), ciclos **for** y **while**, así como funciones que **retornan** algo y funciones que **no retornan** nada, así como operaciones aritméticas y lógicas y para finalizar se debe poder crear variables donde se pueda leer y escribir en estas así como imprimir en pantalla el contenido deseado.

Solución del problema:

Para el desarrollo de este proyecto usaremos como lenguaje de **“backend”** el conocido **Python**, esto porque es un lenguaje que en lo personal nos gusta mucho y aparte de eso tiene una librería gráfica muy sencilla de utilizar que es **PyQT** en su versión 6 y es muy práctico. A pesar de que nuestro lenguaje de programación es fuertemente tipado y Python no lo es, hicimos caso omiso a la recomendación del profesor y proseguimos con Python ya que existen funciones como **type()** que nos permite obtener directamente el tipo de dato y para los casos de los **char** que en Python como tal no existen lo que hicimos fue simplemente verificar que sea un string de longitud 1, con eso ya identificamos que manejamos un char.

En el apartado gráfico contamos con lo solicitado por el profesor, pero distribuido o nombrado de una forma distinta. Para el apartado de los **ejemplos de código** tenemos 2 apartados distintos, el section de **Ejemplos** que directamente muestra una serie de ejemplos por cada estructura de control y tipo de funciones con las que cuenta el lenguaje creado, el otro apartado sería el de **Funcionalidades** donde se pueden acceder a ciertas utilidades, entre las que está la inserción de código de ejemplo para el bloque principal main, esqueleto de las estructuras de control, esqueletos de ciclos, ejemplos de input, output y inclusive hasta la posibilidad de entrar a esta documentación con un solo click.

Aparte de lo mencionado anteriormente, cabe destacar que el código que construye nuestro lenguaje está divido en 3 grandes partes, clases o como se le quiera decir, estas partes son **Lexer**, **Parser** y **Interpreter**, cada una de estas partes cumplen una función en específico, el Lexer se encarga de hacer el análisis léxico que se solicita, esto mediante el uso de expresiones regulares que básicamente abarcan todos los tokens posibles que puede haber dentro de nuestro lenguaje, con el Parser hacemos el análisis sintáctico para determinar que lo que el usuario escribió en el QTextEdit de código de entrada sea correcto en cuanto a su estructura (**recordemos que algo que esté correctamente sintácticamente no necesariamente está bien semánticamente**) y por último el intérprete hace el análisis semántico del código conforme se va ejecutando y también va creando la tabla de símbolos. Cabe recalcar que las primeras dos etapas se ejecutan cuando el usuario presiona el botón de ejecutar y la última etapa se ejecuta con el botón de ejecutar. Cabe resaltar que en cada una de estas etapas pueden surgir errores, casi que todos los errores están manejados con **exceptions** por lo que cuando pase uno se va a mostrar un **popup** con la información del error, si el error es sintáctico se mostrará que se esperaba y que se obtuvo, si el error es de tipo de dato incorrecto se mostrará un mensaje que dice que el valor dado y el tipo de dato esperado no coinciden, este último tipo de errores solo se

detectan una vez que se ejecuta el código ya que hay es cuando hacemos el análisis sintáctico.

Tipado:

Como ya se mencionó anteriormente este proyecto cuenta con un tipado fuerte, este tipado está constituido por tipos primitivos y compuestos, a continuación, detallaremos la sintaxis de estos tipos de datos, la forma de declararlos, algunas funciones que operan en un tipo de dato especifico y demás aspectos relevantes dentro de este apartado del proyecto.

Tipos de datos primitivos:

- Enteros: En el tipo de dato vamos a tener números que no tienen extensión decimal, la palabra clave para este tipo de dato es **entero**, un ejemplo de este tipo de dato sería: **1**.
- Flotantes: En este tipo de dato vamos a tener números que si tienen extensión decimal, la palabra clave para este tipo de dato es **float**, un ejemplo de este tipo de dato sería: **1.5**.
- Caracteres : Este tipo de dato es el átomo del tipo de dato string, por ende representa un solo carácter, para diferenciarse con una cadena optamos por usar ‘’ (comillas simples) para definirlos, un ejemplo de este tipo de dato sería: **‘a’** .
- Booleanos : Los booleanos como en cualquier otra aplicación solo tienen dos posibles valores, verdadero y falso, que en nuestro lenguaje se representan como true y falso respectivamente.
- String: Este tipo de dato es la representación de una cadena de caracteres, para representarlo se usan las “” (comillas dobles), un ejemplo de este tipo de dato sería: **“prueba”**. Cabe destacar que este tipo de dato **NO** se puede recorrer elemento por elemento de manera directa con índices como en otros lenguajes de programación, se tomó está decisión por cuestiones de tiempo y complejidad.

Para crear una variable con alguno de estos tipos se sigue la siguiente sintaxis: **data_type var_name = value**, donde el parser y intérprete se encargan de verificar que los valores correspondan con el tipo de dato especificado.

Tipos de datos compuestos:

- Pilas: En este tipo de dato compuesto se va a poder crear pilas con un tipo de dato primitivo en específico, lo que significa que solo se acepta que los elementos sean ese tipo de dato en específico. Para cumplir con la naturaleza **LIFO** de una pila la estructura cuenta con unas funciones llamadas: **mete, saca y arriba**, donde estas funciones se encargan respectivamente de hacer pop, push y obtener cual es el elemento que está en la posición superior de la pila sin tener que sacar el elemento con un pop(saca). La sintaxis de los métodos saca y arriba es la siguiente: **nombre_funcion(nombre_pila)** y la del método mete es: **mete(nombre_pila, valor)**. Para declarar una pila usamos la palabra clave pila, podemos asignarle un valor de una vez usando [**elemento1, elemento2, elemento3]**, a continuación veremos un ejemplo de cómo podemos declarar una pila : **pila entero nombre_pila= [1, 2, 3 , 4]**

- Listas : En esta estructura de datos compuestos se crean listas convencionales que tienen un dato primitivo específico, se cuenta con funciones para operar sobre este tipo de dato, las funciones son **insertar, obtener, primero y ultimo**, la sintaxis para la primera función es: **insertar(nombre_lista,valor)**, el resultado es que se inserta el **valor** al final de la lista, para la función de **obtener** la sintaxis es: **obtener(nombre_lista, index)**, siendo el resultado la obtención del elemento que se encuentra en la posición indicada por **index,** para las otras dos funciones la sintaxis es: **nombre_funcion(nombre_lista)**, el resultado de estas funciones es el elemento que se encuentra en la primera posición y última posición respectivamente. Para declarar una lista se puede hacer con asignación de una vez o no, en cualquiera de los 2 casos la sintaxis es casi idéntica lo único que varía es el uso de **[elemento1, elemento2]** para indicar el valor de la lista, el siguiente sería un ejemplo de dicha sintaxis donde se crea una lista: **lista entero nombre_lista = [1, 2 , 3 ,4]**

Funciones:

Como mencionamos anteriormente este proyecto cuenta con funciones que retornan y no retornan, por gusto le asignamos a las funciones que no retornan nada la palabra clave **procedimienton** y a las que sí retornan la palabra clave **funcioncita.** A continuación se muestra la sintaxis de ambos tipos de funciones:

Retornan:

funcioncita nombre_funcion (tipo_parametro nombre_parametro, ...) : tipo_return{

*-*codigo de la función-**

retorna valor_o_variable

}

No retornan:

procedimienton nombre_funcion (tipo_parametro nombre_parametro, ...) {

-*codigo de la función-*

}

Invocación de funciones y procedimientos:

Para invocar tanto funciones como procedimientos se sigue la siguiente forma, **nombre(parametros,...)**, es importante tener en cuenta que si la función retorna se puede usar en asignaciones a variables, expresiones aritméticas o lógicas.

Estructuras de control y ciclos:

En la siguiente sección se detallarán las sintaxis y detalles sobre las propuestas hechas, para cada estructura de control o ciclo pedido para el proyecto:

Condicionales simples:

Aquí no hay mucho que explicar, simplemente son condicionales if, elif y else pero con una sintaxis diferente a lo convencional, cabe destacar que **no puede haber un sino o tons sin un si**, las siguientes son las sintaxis:

if:

si (condición) {

*-*código a ejecutar*-*

}

elif:

sino (condición) {

*-*código a ejecutar*-*

}

else:

tons {

*-*código a ejecutar*-*

}

Condicionales múltiples:

Aquí solo implementamos el **switch**, este puede tener n casos y puede tener uno por defecto que es el que se ejecuta si no hay ningúna coincidencia con los demás casos establecidos. La siguiente es la sintaxis:

switch:

casos(variable){
caso (condición){
*-*código a ejecutar*-*
}
defecto{
*-*código a ejecutar si no entra en ningún otro*-*
}
}

Ciclos:

En esta sección se definieron las estructuras repetitivas solicitadas que eran el for y el while, en este caso se ejecutan como en otros lenguajes de programación, el **while** se ejecuta mientras se cumpla la condición y el **for** tiene la peculiaridad de que la variable contadora (**en el ejemplo i**) debe declararse antes de usarla en el for. A continuación se muestran unos ejemplos que evidencian las sintaxis de estas estructuras:

For:

*haga(i = 0, condición, incremento o decremento){-*código a ejecutar*-}*

While:

mientras(condición){código a ejecutar}

Flujo de ejecución:

*Originalmente el código estaba previsto para que no necesitará un main, pero debido a la complejidad de no implementar uno al final tuvimos que usarlo. La función de este es la misma que en **C + +** sirve* como punto de partida para empezar a ejecutar el código. Si se intenta compilar un código **sin main** el IDE va a mostrar un error ya que no se espera que se pueda ejecutar o compilar ningún código sin main ya que carece de sentido. A continuación se muestra un ejemplo donde se aprecia la sintaxis del main:

```
main() {
    -*código a ejecutar*-
}
```

*Cabe resaltar que los **()** no tienen ningún uso dentro del lenguaje de programación creado, están más que nada por un tributo a* **C++** y por la posibilidad de usarlo en un futuro.

Operadores dentro del lenguaje:

A continuación se muestran los posibles operadores aritméticos que están definidos dentro del lenguaje de programación creado:

Operadores aritméticos:

Nota: Si se opera flotante con entero el resultado y se guarda en una variable int va a dar error.

Operación	Definición
Directas	
Multiplicación valores numéricos enteros y flotantes	valor_numerico_1 * valor_numerico_2
División valores numéricos enteros y flotantes	valor_numerico_1 / valor_numerico_2
Suma valores numéricos enteros y	valor_numerico_1 + valor_numerico_2

flotantes	
Resta valores numéricos enteros y flotantes	valor_numerico_1 - valor_numerico_2
Con variables	
División	var1_1 / var_2
Multiplicación	var1_1 * var_2
Suma	var1_1 + var_2
Resta	var1_1 - var_2

Operadores de comparación:

Desigualdad	var_1 <> var_2
Igualdad	var_1 == var_2
Mayor que	var_1 > var_2
Menor que	var_1 < var_2
Mayor igual que	var_1 >= var_2
Menor igual que	var_1 <= var_2

Operadores lógicos:

Estas operaciones sirven para evaluar condiciones dentro de las estructuras de control definidas más arriba o generar valores de verdad

And	expresión_1 and expresión_2

Or	expresión_1 or expresión_2
Not	not expresión_1

Entrada y Salida de Datos:

Para la entrada de datos definimos la palabra clave **lea**, esta función simplemente lo que hace es habilitar la consola para que el usuario ingrese un valor, dentro de esa función se le asigna el valor digitado a la variable, antes verificando que el tipo de la variable y el tipo del dato ingresado sean iguales, la sintaxis sería tal que así: **lea(variable_donde_guarda)**. Por otra parte se creó la palabra clave **escriba** para imprimir en el QTextEdit de salida lo que el usuario establezca, esta función cuenta con otra palabra reservada que es **salto** que básicamente agrega un salto de línea, cuando se pone dentro de los argumentos de la función escriba , la siguiente es la sintaxis de esta función: **escriba(“texto a mostrar”, salto, variable, salto, “ fin del comentario”)**. En el anterior ejemplo se mostró como escriba puede recibir **n argumentos**, siendo **salto** uno de estos.