TUDI

Diseño de compiladores

Grupo 2
Prof. Elda Quiroga y Héctor Ceballos

Fecha:
2022-11-22

Ángel Adrián Guevara Hernández
A01570288

David Alejandro García Treviño
A01570231

Índice

Documentación y descripción técnica

Descripción del proyecto

Propósito y alcance del proyecto

El propósito de ***TUDI*** es ser un lenguaje de programación para desarrollar videojuegos sencillos, o prototipos de videojuegos, con una vista 2D. La idea de ***TUDI*** es ser un lenguaje con el cuál un usuario pueda introducirse en el mundo de la programación a través del desarrollo de un pequeño proyecto que tenga en mente, ya que además de ser fácil de leer y entender, también contará con las principales características básicas que podrá encontrar en otros lenguajes en su futuro, como lo son: *tipos de datos, entradas y salidas, variables, condicionales, ciclos, funciones y estructuras de datos básicas (arreglos y matrices)*.

El alcance del proyecto es completar un lenguaje de programación básico que tenga la permisividad de utilizar varios de los componentes importantes de la librería **Pygame** para tener cubierta la parte de videojuegos y que conviva con el compilador.

Descripción de casos de prueba

Para comprobar el funcionamiento de ***TUDI*** se planea utilizar todos los elementos básicos que cualquier lenguaje de programación imperativo de alto nivel debe de cumplir. Los casos de prueba deben de cumplir con la siguiente lista:

- Posibilidad de declarar variables en los tipos de datos implementados.
- Posibilidad de declarar variables de tipos estructurados (arreglos y matrices) de un mismo tipo.
- Declaración de funciones
- Llamada de funciones
- Manejo de scopes en funciones y variables
- Cumplir estatutos condicionales
- Tener la posibilidad de crear loops de tipo *for* y *while*
- Manejo de entradas y salidas de datos
- Implementación de elementos para renderizar videojuegos simples en 2D.

A continuación, se detalla una pequeña descripción de cada caso de prueba principal:

- **factorial_{iterativo, recursivo}.tudi** -> Este programa debe de calcular el factorial de N, tanto en su versión iterativa como recursiva. El trayecto del programa sigue el siguiente camino:
    - Pedir al usuario un número mayor o igual a 0.
    - Si el número es menor a 0, entonces se debe volver a pedir.

    - Después de obtener el número, llamar a la función **factorial(int num) -> int,** pasando el número como argumento y guardando el resultado de retorno en una variable.
    - Imprimir la variable donde se guarda el resultado.
- **fibonacci_{iterativo, recursivo}.tudi** -> Este programa debe de calcular el término N de la serie de fibonacci, tanto en su versión iterativa como recursiva. El trayecto del programa sigue el siguiente camino:
    - Pedir al usuario un número mayor a 0.
    - Si el número es menor o igual a 0, entonces se debe volver a pedir.
    - Después de obtener el número, llamar a la función **fibonacci(int num) -> int,** pasando el número como argumento y guardando el resultado de retorno en una variable.
    - Imprimir la variable donde se guarda el resultado.
- **find_vec.tudi** -> Este programa debe de crear un vector de enteros de tamaño N (máximo 10), pedir un valor entero a buscar e imprimir el índice en donde se encontró. El trayecto del programa sigue el siguiente camino:
    - Pedir al usuario un número en el rango de [1, 10].
    - Si el número está fuera del rango, entonces se debe volver a pedir.
    - Después de obtener el número, llamar a la función **create_vec(int len) -> int[10],** pasando el número como argumento y guardando el resultado de retorno en una variable.
    - Guardar el resultado de la función en una variable *vec*, y pedir al usuario un número entero a buscar dentro del vector recién creado.
    - Llamar a la función **find(int[10] vec, int len, int val) -> int**, el cual debe de recibir un arreglo y su tamaño, así como el valor a buscar dentro de él. Si el valor se encuentra entonces regresar el índice en donde se encontró, sino regresar -1.
    - Dependiendo del resultado, informarle al usuario la posición en donde se encontró el valor o si no se encontró el valor en el vector.
- **sort_n2.tudi** -> Este programa debe de crear un vector de enteros de tamaño N (máximo 10), ordenar de menor a mayor los valores e imprimirlo. El trayecto del programa sigue el siguiente camino:
    - Pedir al usuario un número en el rango de [1, 10].
    - Si el número está fuera del rango, entonces se debe volver a pedir.
    - Después de obtener el número, llamar a la función **create_vec(int len) -> int[10],** pasando el número como argumento y guardando el resultado de retorno en una variable.
    - Guardar el resultado de la función en una variable *vec*, y llamar a la función **sort(int[10] vec, int len) -> vec**, el cual debe de recibir un arreglo y su tamaño, y regresar el arreglo ordenado. La función de sort, realiza un *selection sort*.
    - Imprimir el vector ordenado.

- **sort_merge.tudi** -> Este programa es similar al **sort_n2.tudi**, pero la función del sort se trata de un *merge sort.*
- **multiplicacion_matriz.tudi** -> Este programa debe de crear dos matrices que se puedan multiplicar entre sí de tamaño [N, N] (N es máximo 10), multiplicarlas, guardar el resultado e imprimirlo. El trayecto del programa sigue el siguiente camino:
    - Pedir al usuario un número *m* en el rango de [1, 10],.
    - Si el número está fuera del rango, entonces se debe volver a pedir.
    - Realizar lo mismo para un número *n* y *p*.
    - Después de obtener los números, llamar a la función **create_mat(int m, int n) -> int[10, 10],** pasando el número *m* y *n*, para crear la matriz A.
    - Después, llamar a la función **create_mat(int m, int n) -> int[10, 10],** pasando el número *n* y *p*, para crear la matriz B.
    - Después de obtener las dos matrices, llamar a la función **mat_mul(int[10,10] mat_a, int[10,10] mat_b, int m, int p, int n) -> int[10, 10],** pasando los valores correspondientes. Esta función lleva a cabo la multiplicación de matrices, y regresa la matriz resultante.
    - Guardar la matriz que regresó la función e imprimirla.
- **snake.tudi** -> Este programa debe de crear el clásico juego de ***snake*** con las reglas básicas de puntos y vidas**.**

Proceso

El proceso de desarrollo se considera en dos fases:

1. Planeación y diseño del lenguaje
2. Generación de código y pruebas

Para la primera etapa, se realizaron juntas en persona para trabajar. Las juntas iniciales se realizaron para definir el objetivo del lenguaje y las características especiales planeadas. Después de haber estipulado esto (junto con el nombre del lenguaje) se realizaron videollamadas para crear los diagramas de sintaxis. Estos ocurrieron en varias iteraciones hasta llegar a una versión, que si bien, no tenía los puntos neurálgicos, servía para establecer una base.

La segunda etapa fue más enfocada en procesos de desarrollo de código iterativo. Para lo que teníamos que hacer en cada iteración y las fechas de entrega nos basamos en lo recomendado por el calendario de la clase de *Diseño de compiladores (Véase Apéndice A).*

Para cada etapa de los avances dividimos la carga en dos para repartir esfuerzo. Era que para antes de las fechas estipuladas, cada miembro del equipo completara lo asignado y subiera un “pull request” (llamados PR a partir de esta sección) a la plataforma GitHub con su código para que el otro integrante lo revisara e hiciera pruebas en su máquina.

La estructura de un PR normal consistía en una pequeña documentación de lo realizado así como una lista autogenerada por Github de cada commit. La mayoría de los PR contienen

sugerencias de código o de resaltado de bugs junto con una sugerencia de parte del otro miembro, en algunos casos tienen un pequeño comentario al final de la revisión. Los PR en ciertos casos requirieron de refactorizaciones más grandes que requerían resolverse en otro PR.

En el **Apéndice B** se encuentra una liga a un Google Drive con todas las capturas de pantalla de los distintos PR junto con sus commits en orden cronológico de más antigüo a más reciente (entre menor sea el número, más antigüo). Dentro de los screenshots hay textos que inician un asterisco (*) que representan los commits (abajo de algunos commits se pueden encontrar algunos comentarios) y en **bold** el nombre de los PR.

A continuación se presentan los párrafos de reflexión de los miembros del equipo:

Ángel: Este ha sido uno de mis proyectos favoritos a lo largo de la carrera, muy posiblemente mi favorito, y aunque me llevo una gran satisfacción con el resultado logrado y los aprendizajes obtenidos, me voy con una pequeña espina, porque sé que no es el mejor proyecto que puedo hacer. El trabajo en equipo siento fue muy bueno, ya que considero que durante el todo el proyecto estuvimos en constante comunicación y nos enfocamos no solamente en completar cada parte asignada, sino también en revisar el código del otro, tanto para encontrar errores o mejoras, pero también con el propósito de mantenernos al tanto de los cambios que iban sucediendo commit a commit. Definitivamente el proyecto te abre los ojos para que te puedas imaginar lo complejo que debe ser un compilador de los que estamos acostumbrados a usar. Ahora, creo que pudimos hacer un trabajo mejor si hubiéramos sido más disciplinados con los avances semanales, pero en lo personal, me divertí y estuve estresado, pero terminé contento y satisfecho.

David: Siempre he escuchado que hay una diferencia notoria entre alguien que en su carrera ha desarrollado un compilador a uno que no, y después de llevar la clase puedo ver el porqué se dice esto. La clase de sistemas operativos me enseñó bastante acerca del funcionamiento de los procesos en un computador, pero hasta esta clase pude ver un conjunto de todo lo técnico unido. Desde las estructuras de datos necesarias, hasta la optimización de recursos, hasta la representación de datos, todo se junta en este proyecto grande que es hacer un compilador desde la raíz. Mientras programaba el compilador con mi compañero, más me hacía sensible acerca de las implicaciones de lo que yo mismo programaba en otros lenguajes así como el gran esfuerzo que hay detrás de todos los lenguajes. Esta clase también me enseñó a apreciar todavía más el diseño de los programas y de la importancia de tener dónde consultar futuros y actuales avances así como del correcto trabajo en equipo. Aquí se nos dió la oportunidad de diseñar las cosas a nuestra manera desde el nivel léxico hasta el semántico, saber lo que hace un compilador es una cosa, pero hacerlo es completamente diferente. Sin duda ha sido de mis proyectos más interesantes de la carrera y a la vez de los más enriquecedores.

Ángel Adrián Guevara Hernández
A01570288

David Alejandro García Treviño
A01570231

Descripción del lenguaje *TUDI*

Descripción genérica de las características del lenguaje

El lenguaje sigue muchos de los fundamentos creados por el lenguaje **C**. Características que fueron tomadas fueron: el uso de una función principal para empezar a correr el programa (la función Start); en el caso de tipos de dato estructurados que tuviera que siempre iniciar de la casilla cero; terminar cada instrucción de código con un “;” ; estipular los tipos de datos desde la declaración de las variables así como en las funciones; sintaxis de los estatutos condicionales (declaraciones y ciclos); orden asociativo derecha para la evaluación de expresiones y asociativo izquierda para la asignación; también utiliza pura lógica numérica, esto significa que los booleanos son realmente 1 y 0 y por lo tanto, se pueden sumar e interactuar con los otros tipos de datos de forma más sencilla.

Lo que ya es más propio del lenguaje en comparación a C se puede ver en la sintaxis de funciones, que estipula primero que es una función, para luego declarar el nombre de la función (que no se puede repetir sin importar los parámetros). Al inicio del programa se necesita escribir el nombre del programa (o juego) y escribir el tamaño que va a tener el canvas (o ventana) donde se desarrollará el juego, también es obligatorio que se declaren las variables globales luego de estipular el tamaño del canvas, ya que al iniciar a desarrollar funciones ya no se podrán declarar. Las variables locales solo se pueden declarar en la primera instrucción del programa utilizando la palabra reservada “declare”. Las variables declaradas siempre se inicializan con un valor default (estipulado a continuación) y solo se le puede asignar los valores ya en el bloque de código. Otra característica es que es obligatorio tener los métodos Start y Update para que el programa inicie. La función de Start, como se dijo anteriormente, actúa como una clase de función **main** de **C**, mientras que Update se corre en cada tick de frame, como en un videojuego. El lenguaje permite que se manden arreglos como parámetros y como valores de retorno (también aplica a matrices), aunque está limitado a ser *pass-by-value*, no es posible pasar por referencia, por lo que se crea una copia.

Tipos de errores

- **Re-declaración de variable/function/parameter**: Sale cuando una variable o parámetro que ya está en el mismo scope se intenta declarar de nuevo. En el caso de funciones, ***TUDI*** no permite funciones nombradas igual, en todo caso, sucede cuando se crea otra función con el mismo nombre.

- **Return expected value of type [ ] but got [ ]** : sucede cuando en el valor de retorno enviamos una constante o variable que no es del tipo estipulado en el tipo de la función.
- **Function [ ] expected [ ] arguments, but got [ ]** : este error sucede cuando en la llamada enviamos menos argumentos de los requeridos en una llamada a función.
- **Function [ ] was not declared**: este error sucede cuando hacemos una llamada de función a una que no existe
- **Function [ ] expected [ ] arguments, but got more than needed**: sucede cuando haces una llamada a una función con más argumentos que los parámetros que tiene la función.
- **Function [ ] expected argument [ ] of type [ ] but got [**]: sucede cuando enviamos un tipo de dato que no es el que viene en el parámetro de la función.
- **Expecting a boolean, int or float, instead got [ ]**: Sucede cuando en los estatutos *if, for y while* enviamos una expresión que no regresa alguno de esos tipos de datos en la comparación (por ejemplo, un arreglo entero).
- **Cannot assign value of type [ ] to variable [ ] of type [ ]**: sucede cuando intentamos asignar un valor que no corresponde al tipo de la variable, este estrictamente se tiene que cumplir.
- **Array dimensions must be greater than 0**: ***TUDI*** pide que en la declaración de listas se coloque un número entero mayor a 0 ya que esta representa el tamaño del array.
- **Variable [ ] was not declared**: sucede cuando queremos utilizar una variable que todavía no ha sido declarada
- **Variable [ ] is not an array/matrix**: sucede cuando una variable que no representa un arreglo le tratan de indexar. Por ejemplo, una variable entera “num” intentando hacer “num[0]” o “num[0][0]”..
- **Index of [ ] must be an integer**: sucede cuando intentas acceder a un elemento de un arreglo con algo que no sea un entero. Ejemplo: arr[3.14] daría este error.
- **Syntax error input at line [ ]**: error que aparece por default cuando no está manejado y hay un error de sintaxis.
- **Address does not belong to any type**: sucede cuando enviamos una dirección de memoria que no pertenece a ninguno de los rangos estipulados
- **SCOPE TYPE MEMORY EXCEEDED**: sucede cuando nos excedemos del máximo de direcciones disponibles para un tipo de dato en un scope.

Compilador

Equipo de cómputo y utilerías especiales

Para crear el compilador se utilizó la herramienta **PLY** y el lenguaje de programación **Python3**. La herramienta **PLY** es una reescritura de las ya conocidas herramientas **Lex & Yacc** pero para Python.

La idea es que ***TUDI*** pueda ser corrido en sistemas operativos Windows. Para lo relacionado al *game engine*, se está utilizando **PyGame.**

PLY viene ya con el código de TUDI, pero para poder utilizar **PyGame**, es necesario que el usuario lo instale con *pip*. (**python -m pip install pygame**)

Análisis Léxico

A continuación se enlistan los patrones de construcción con expresiones regulares, tokens y palabras reservadas junto con su explicación de uso. En el caso de los tokens y palabras reservadas, no se escribirá un código asociado ya que la herramienta **PLY** puede utilizar los tokens específicos (llamados literales) como su propia representación. Esto quiere decir que en la gramática se representan estos literales tal cual como se escriben. En el caso de las palabras reservadas, todas sus representaciones se escriben igual pero usando solamente mayúsculas. Ejemplo: el carácter '**{**' es tal cual así y la palabra reservada **Start** sería **START.**

PATRONES DE CONSTRUCCIÓN

REL_OPS	<=|>=|>|<|!=|==	Enlista todos los operandos de tipo relacional.
LOGIC_OPS	\b(o | y)\b	Enlista los operandos de tipo lógico
ASSIGN_OP	=	Un signo de igual para representar la asignación.
INT_LITERAL	[0-9]+	Servirá para validar los datos de tipo entero.
FLOAT_LITERAL	[0-9]+(\. [0-9]+)	Servirá para validar los datos de tipo flotante.
BOOL_LITERAL	(true) | (false)	Servirá para identificar un valor de verdadero o falso.
STRING_LITERAL	\"(\w|\s|\n|[\\\-*+/():=\[\],:<>.])+\"	Servirá para identificar todo aquello que sea una palabra entre comillas.
ID	[a-zA-Z_][a-zA-Z_0-9]*	Servirá para crear identificadores para las variables.

TOKENS Y PALABRAS RESERVADAS

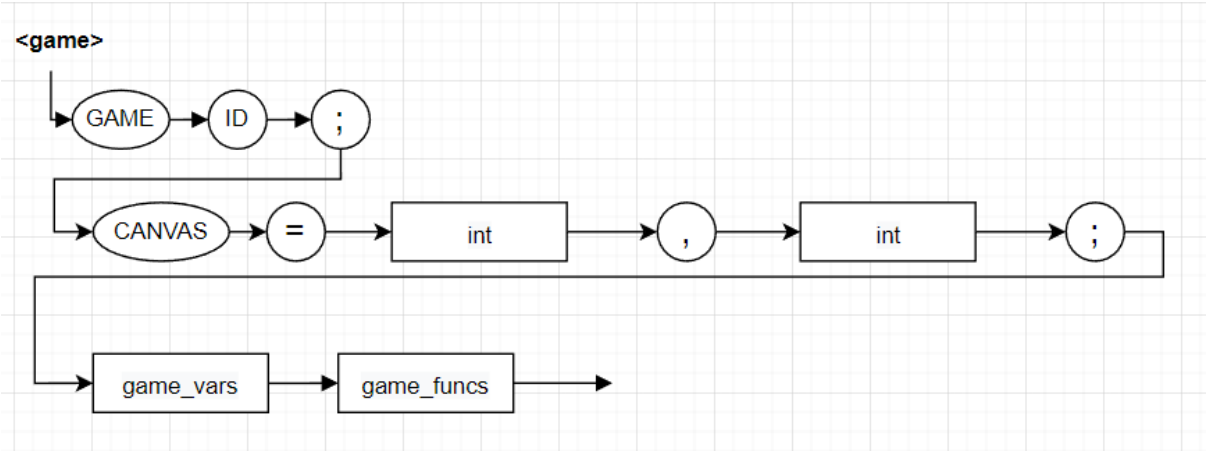
Token	**Descripción**

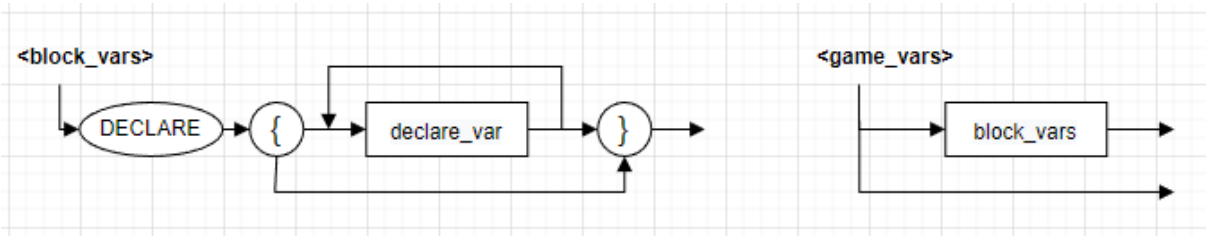
START	Es la función inicial, la primera que se va a correr al iniciar el programa.
UPDATE	Es la función que se ejecuta durante cada frame del programa.
GAME	Va a servir para guardar el título del juego desde el programa.
PRINT	Va a servir para mostrar output al usuario en la consola.
READ	Va a servir para recibir input del usuario.
INT	Inicia la declaración del tipo de dato entero.
FLOAT	Inicia la declaración del tipo de dato float.
BOOLEAN	Inicia la declaración del tipo de dato booleano.
CHAR	Inicia la declaración del tipo de dato char.
VOID	Tipo de dato de retorno de una función que no regresará nada.
CANVAS	Representa el espacio en el que se podrán dibujar sprites.
FUNC	Representa que lo siguiente será una función.
RETURN	Palabra reservada cuyo siguiente valor o variable tendrá que ser del mismo tipo de la función en la que está dentro.
DECLARE	Palabra reservada cuya función es iniciar el bloque donde se declaran las variables.
IF	Palabra reservada para inicializar un condicional.
ELSE	Palabra reservada para indicar un caso alterno en una condicional.
FOR	Inicialización de un ciclo for.
NOT	Negación booleana *not*.
WHILE	Inicialización de un ciclo while.
;	Al final de declarar estatutos se colocará este token para indicar su fin.

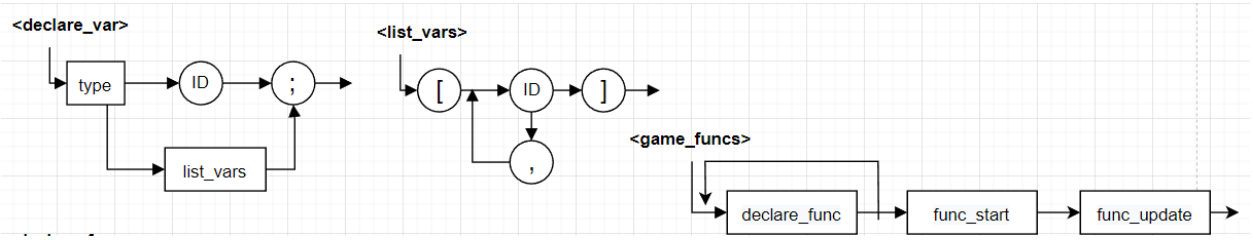
:	Para declarar el tipo de retorno de una función.
.	Servirá para marcar las llamadas a los métodos. Ejemplo: obj.metodo()
,	Servirá para listar valores dentro de un arreglo (hacer la división entre valores) así como ayudar a declarar distintas dimensiones en un arreglo.
=	Se utilizará para las asignaciones de variables.
{	Se utilizará para iniciar un bloque de código.
}	Se utilizará para finalizar un bloque de código.
(	Tendrá como propósito indicar las llamadas a funciones o inicializar los paréntesis en una expresión.
)	Tendrá como propósito indicar el final de una llamada a función, después de llamar a los parámetros o finalizar los paréntesis en una expresión.
[	Servirá para iniciar la declaración de un arreglo o intento de acceso a un arreglo.
]	Servirá para finalizar la declaración de un arreglo o intento de acceso a un arreglo.
+	Servirá para hacer sumas en las expresiones, o en el caso de los chars, anidar un char al final del arreglo de chars.
-	Servirá para hacer las restas en las expresiones.
/	Servirá para hacer las divisiones en las expresiones
*	Servirá para hacer las multiplicaciones en las expresiones.
INIT_GAME	Inicialza pygame en la máquina virtual
RANDOM	Recibe dos números enteros y la variable donde se va a guardar como argumentos,

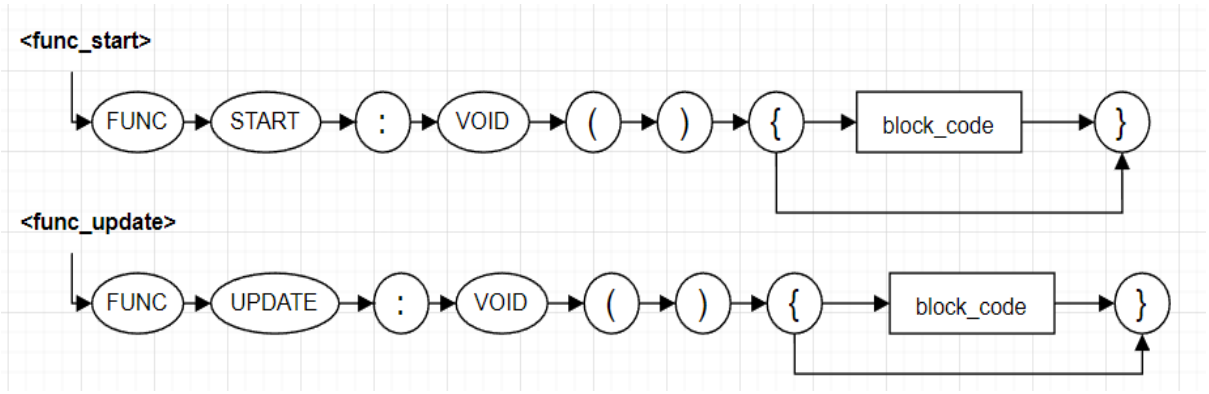
	sirve para generar números aleatorios en un rango.
GAME_OVER	Da la terminación del programa, hace un salto directo al último cuádruplo
WRITE_SCREEN	Escribe un texto en la parte superior izquierda de la pantalla.
GET_EVENT	Recibe las entradas del teclado dentro del videojuego, tiene como argumento la variable donde se guardara el evento utilizado.
DRAW_RECT	Llama a la funcion de renderizado de rectángulos de pygame, recibe un color (de la lista delimitada), sucedido de las posiciones y los tamaños del rectángulo.
SET_FILL	Recibe una string de un color (de una lista delimitada) y rellena la ventana de ese color.
TICK	Sirve para delimitar los frames per second del juego a correr
UPDATE_GAME	Funcion necesaria para pygame, útil para “repintar” sobre el canvas.

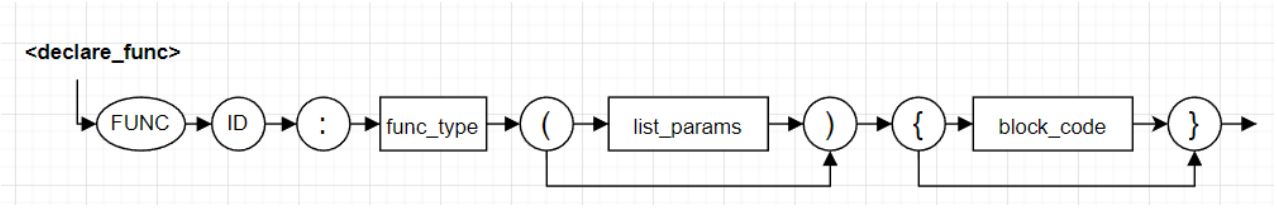
Análisis de Sintaxis

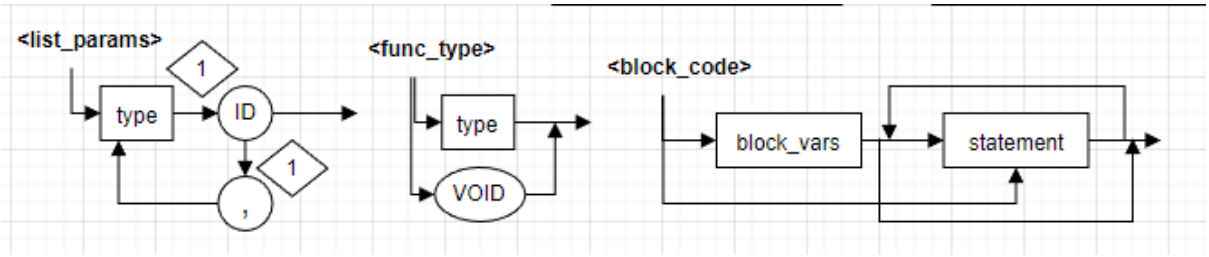


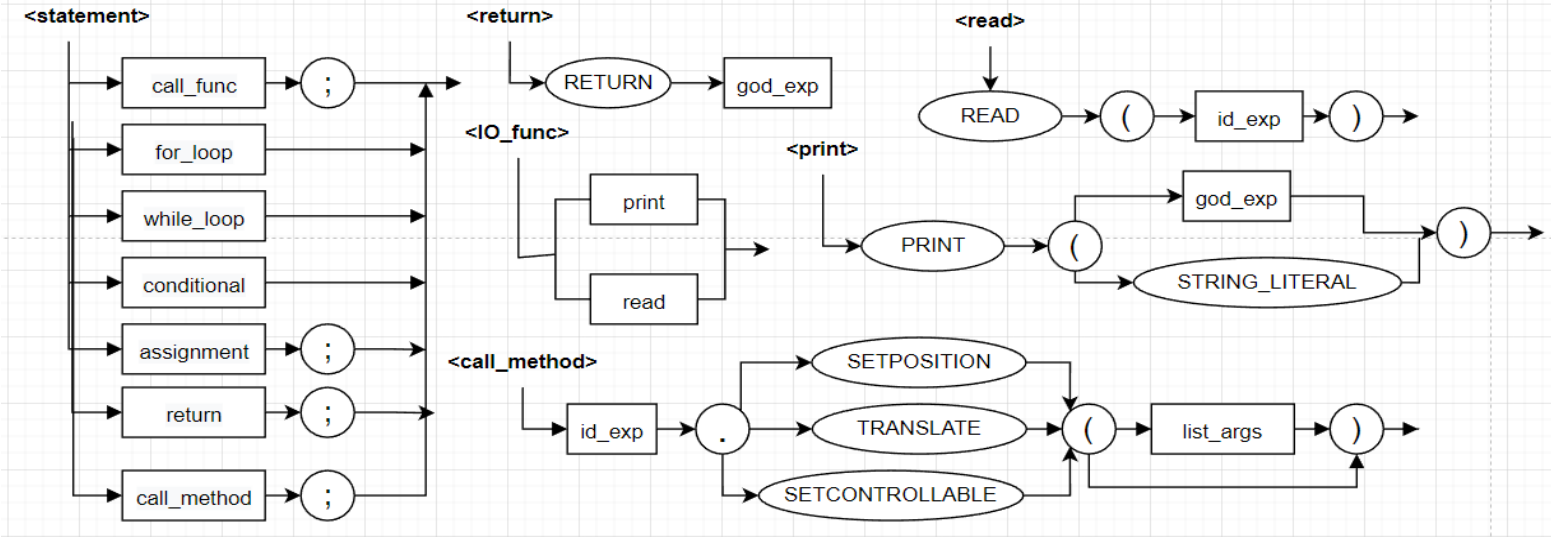


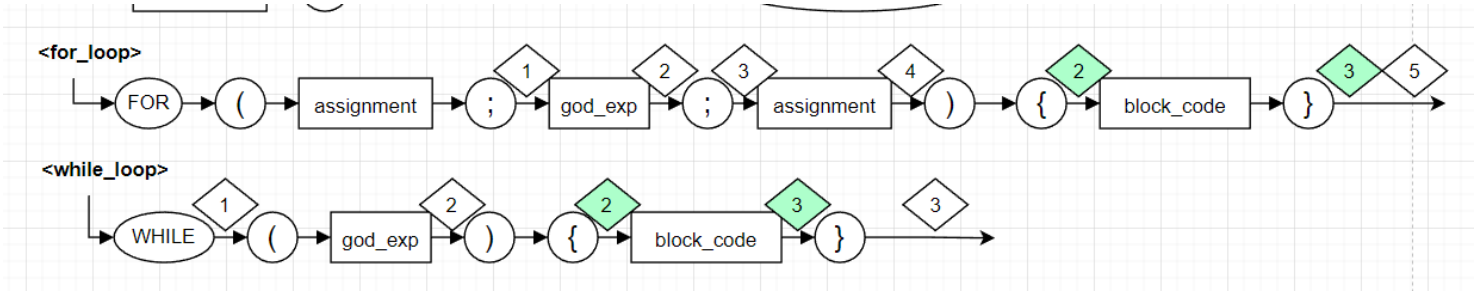

<declare_var>
type
ID
;
list_vars
<list_vars>
[
ID
]
,
<game_funcs>
declare_func
func_start
func_update

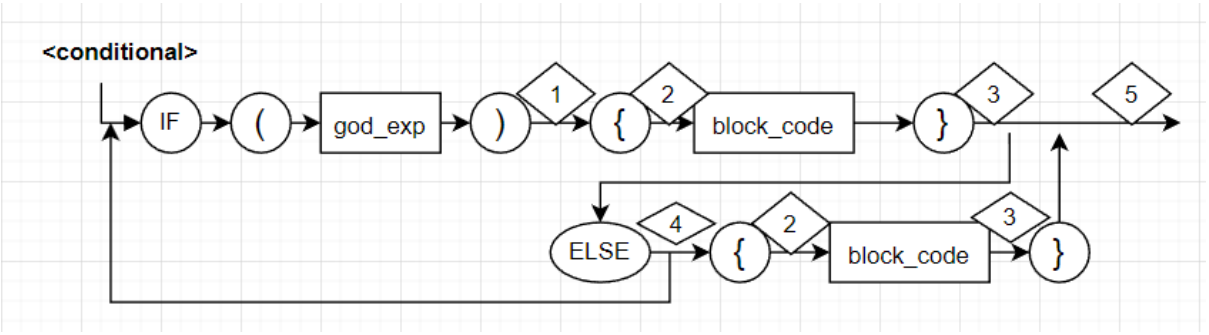

<func_start>
FUNC
START
:
VOID
(
)
{
block_code
}
<func_update>
FUNC
UPDATE
:
VOID
(
)
{
block_code
}

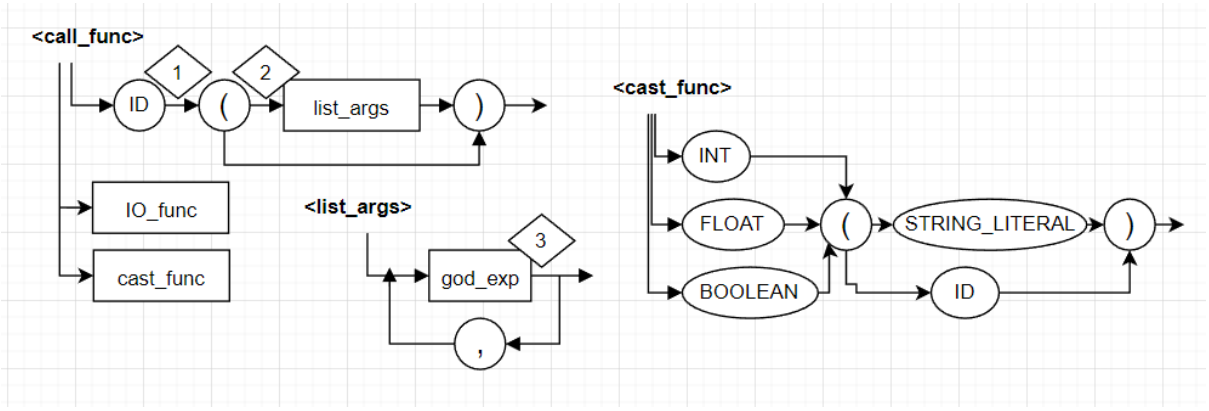

<declare_func>
FUNC
ID
:
func_type
(
list_params
)
{
block_code
}

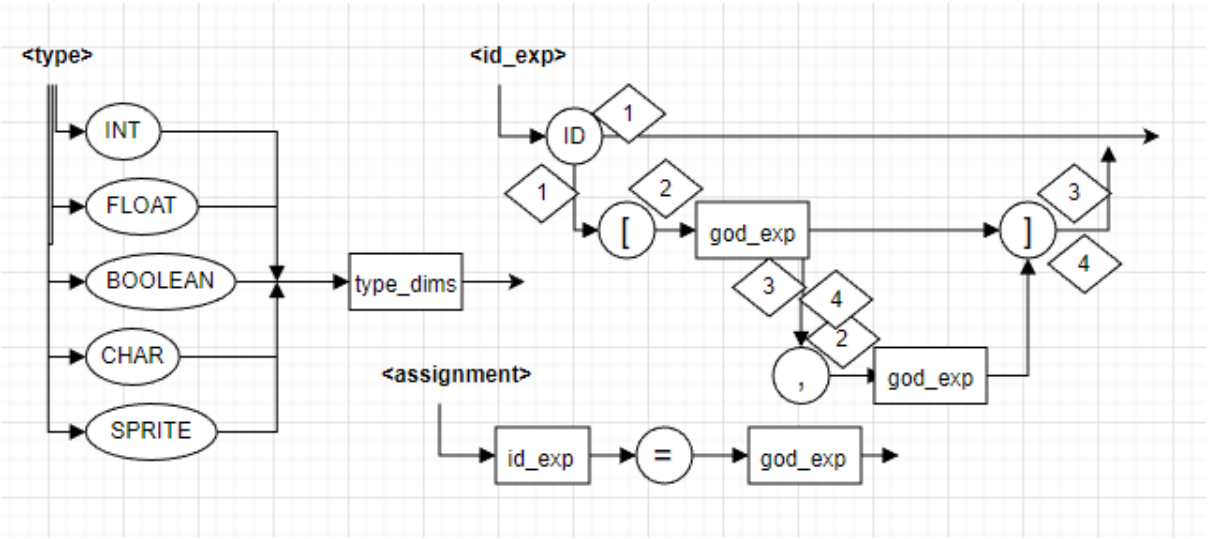

<list_params>
1
type
ID
1
,
<func_type>
type
VOID
<block_code>
block_vars
statement

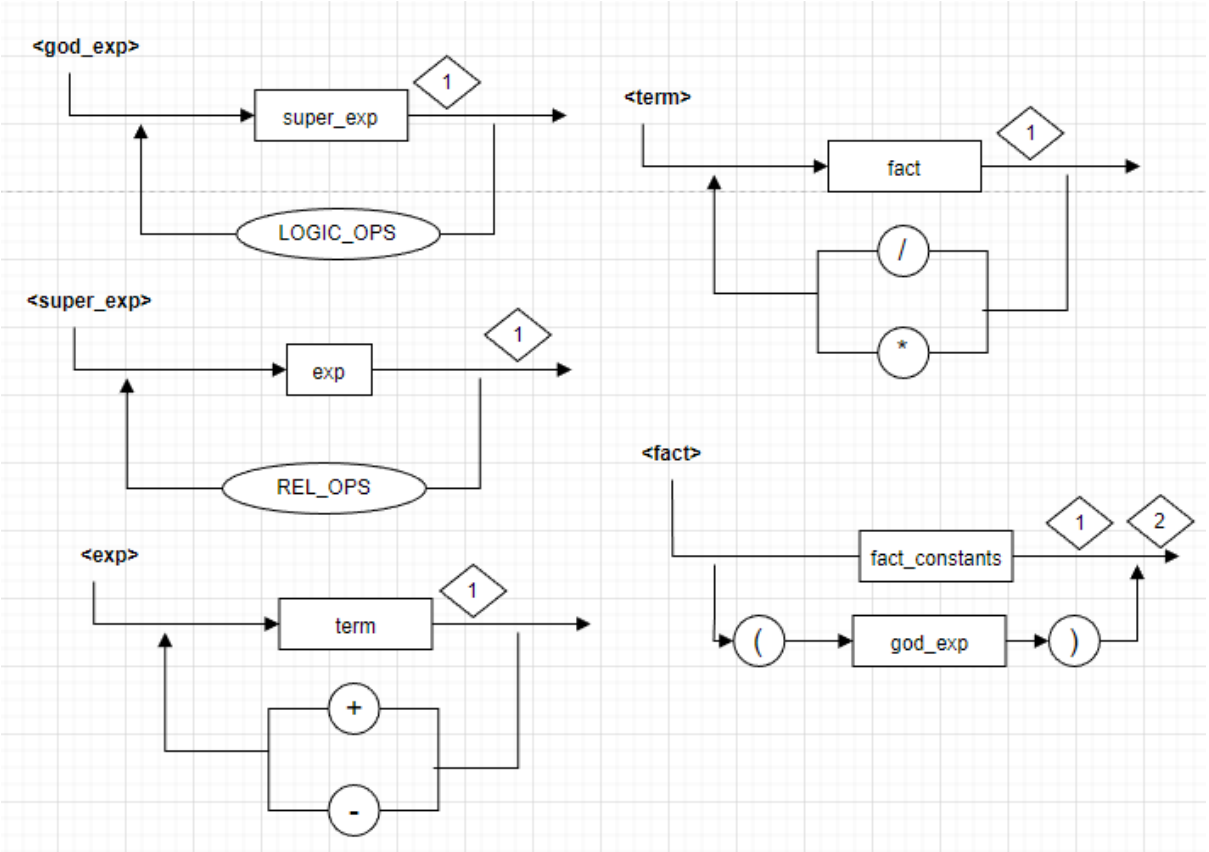

<statement>
call_func
;
for_loop
while_loop
conditional
assignment
;
return
;
call_method
;
<return>
RETURN
god_exp
<IO_func>
print
read
<read>
READ
(
id_exp
)
<print>
PRINT
(
god_exp
STRING_LITERAL
)
<call_method>
id_exp
.
SETPOSITION
TRANSLATE
SETCONTROLLABLE
(
list_args
)

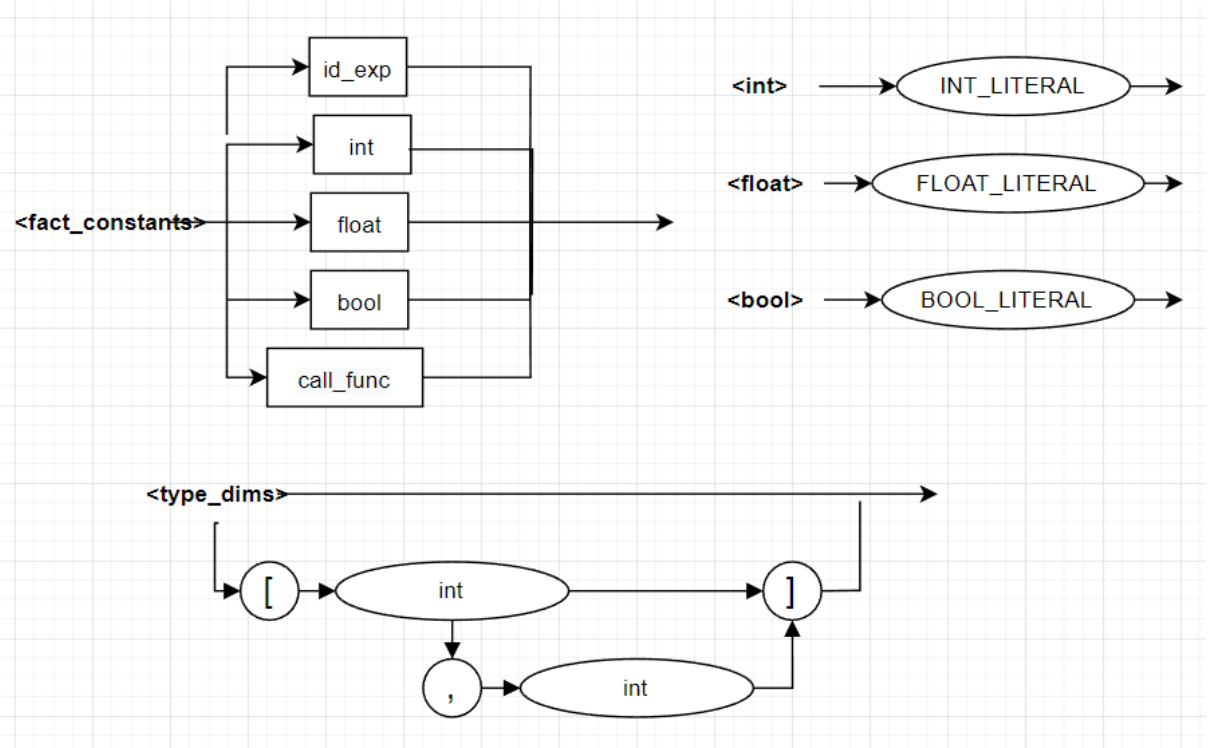

<for_loop>
FOR
(
assignment
;
1
god_exp
2
;
3
assignment
4
)
2
{
block_code
}
3
5
<while_loop>
1
WHILE
(
god_exp
2
)
2
{
block_code
3
}
3


<conditional>
IF
(
god_exp
)
1
{
2
block_code
}
3
5
ELSE
4
{
2
block_code
3
}


<call_func>
1
ID
(
2
list_args
)
IO_func
cast_func
<list_args>
3
god_exp
,
<cast_func>
INT
FLOAT
BOOLEAN
(
STRING_LITERAL
ID
)


<type>
INT
FLOAT
BOOLEAN
CHAR
SPRITE
type_dims
<id_exp>
ID
1
1
2
[
god_exp
]
3
4
3
4
2
,
god_exp
<assignment>
id_exp
=
god_exp


<god_exp>
super_exp
1
LOGIC_OPS
<term>
fact
1
/
*
<super_exp>
exp
1
REL_OPS
<exp>
term
1
+
-
<fact>
fact_constants
1
2
(
god_exp
)

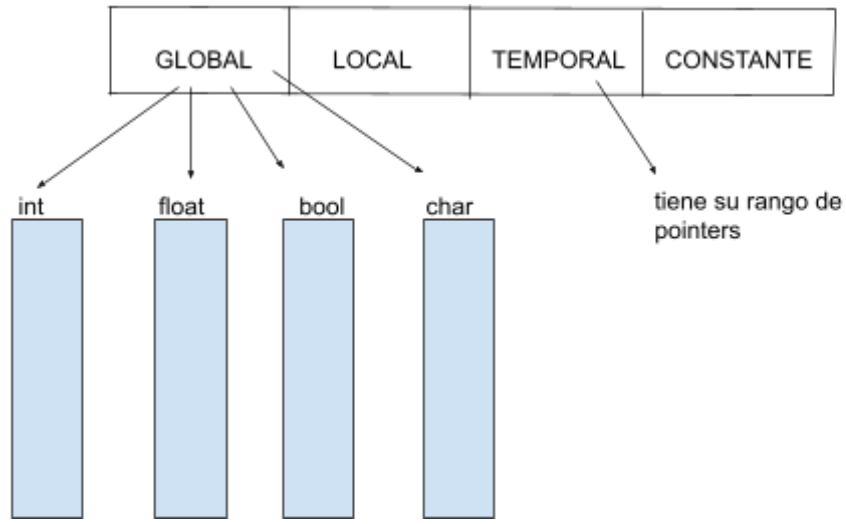


Los diagramas fueron dibujados utilizando app.diagrams.net y la liga se encuentra a continuación:
https://app.diagrams.net/#G1eFfoS52eKJXA6nEuccpJmoo8VIqltSNJ

Generación de Código intermedio y análisis semántico

Para la generación de código intermedio utilizamos código de Python en nuestras reglas gramaticales. En este código hacemos la generación de cuádruplos utilizando puntos neurálgicos y manejo de las colas de operaciones y operandos. En la codificación se hace uso de variables globales, locales, temporales y constantes en forma de direcciones virtuales, estas son escritas según el tipo de datos y el scope que tengan las variables, esto quiere decir que las locales enteras van a tener direcciones dentro de un rango y así con los diferentes tipos de datos y scopes.

Las direcciones de memoria son alocadas siguen la siguiente estructura:



Los rangos son definidos por la siguiente tabla:

Espacio	Rango
GLOBAL_INT	0-3999
GLOBAL_FLOAT	4000-7999
GLOBAL_BOOL	8000-11999
GLOBAL_CHAR	12000-19999
LOCAL_INT	20000-23999
LOCAL_FLOAT	24000-27999
LOCAL_BOOL	28000-31999
LOCAL_CHAR	32000-39999
TEMP_INT	40000-43999
TEMP_FLOAT	44000-47999
TEMP_BOOL	48000-51999
TEMP_CHAR	52000-55999
TEMP_POINTER	56000-59999
CONST_INT	60000-63999
CONST_FLOAT	64000-67999
CONST_BOOL	68000-68001

CONST_CHAR	68002-79999

A continuación se explicarán algunos diagramas de sintaxis, cabe recalcar que no todos tienen puntos neurálgicos, sin embargo, hay código que se ejecuta luego de revisar la regla, estos no fueron numerados pero serán explicados de igual manera (los que cumplan esta condición no será numerado):

Game

Se generan los cuádruplos de ir a las funciones **Start** y **Update** con sus reservas de espacio (ERA) y llamada a función (GOSUB).

game_vars

Se crean los cuádruplos de inicio, estos son el tamaño del canvas donde se va a dibujar, el nombre del juego (se hará el display en la ventana) y el goto hacia la función **Start**.

declare_var

Toma la lista de variables y guarda cada una de estas (con sus dimensiones e incrementó) con el valor de la variable como su dirección virtual (revisando si es local o global).

list_vars

Va guardando los tokens enlistados en el bloque de declaración de variables con su id (que incluye, su nombre y su tipo).

func_start/func_update

Se pone al final un cuádruplo **ENDFUNC** y hace el proceso de terminación de función (enumerar recursos, limpiar tabla de variables, revisar redeclaración y resetear temporales y locales).

declare_func

Funciona igual que **func_start** pero revisa una lista de parámetros (siguiente apartado) y tiene un retorno (se revisa el tipo) a menos de que sea una función *void*.

list_params

1. Revisa las dimensiones para que no sean parámetros, le asigna una dirección de memoria local y lo añade a la lista de parámetros de la función

func_type

Revisa que las dimensiones del tipo no sean un arreglo y guarda el tipo de dato que regresará la función, incluido void.

block_code

Reúne todos los estatutos (statement).

return

Revisa que lo regresado por la **god_exp** sea del mismo tipo que la función y guarda un cuádruplo de retorno junto con el operando de lo que regresó la **god_exp**.

read

Agrega un cuádruplo de lectura junto con la dirección de memoria de la variable al que se le asignará el valor escrito por el usuario.

print

Crea un cuádruplo de impresión en el que guarda un **letrero** en caso de escribir un string, o guarda la dirección de memoria de la variable a imprimir.

for_loop

Los números en verde representan los puntos 2 y 3 de **conditional**, revisar en ese apartado.

1. Crea una checkpoint en la pila de saltos para antes del condicional
2. Hace pop del resultado de la **god_exp** y revisa que sea un tipo valido. Crea cuádruplos de GOTOF y GOTOV y añade checkpoints de ambos a la pila de saltos.
3. Agrega un checkpoint en la pila de saltos para la asignación del for.
4. Agrega un GOTO vacío que va a servir para regresar a la asignación y añade un checkpoint para rellenarlo luego.
5. Hace pop de todos los checkpoints guardados, en el GOTOF se pone el siguiente número de cuádruplo, en el GOTOV se pone uno luego del condicional, y rellena el GOTO con lo puesto antes del condicional.

while_loop

Los números en verde representan los puntos 2 y 3 de **conditional**, revisar en ese apartado.

1. Crea una checkpoint en la pila de saltos para antes del condicional
2. Hace pop del resultado de **god_exp** y revisa que el tipo sea válido. Crea un checkpoint para luego de evaluar la condición y agrega un GOTOF vacío.
3. Al final del while, rellenar el GOTOF con la dirección que sigue en el conteo y agregar un cuádruplo de GOTO al primer checkpoint.

conditional

1. Obtenemos el operando creado por el **god_exp** y checamos que sea válido. Hacemos un checkpoint y un cuádruplo GOTOF para rellenar después.
2. Creamos fondo falso en la pila de saltos, útil para no mezclar condicionales dentro de otros condicionales

3. Quitamos el fondo falso de la pila de saltos después del bloque de código
4. Recuperamos el checkpoint y rellenamos ese GOTOF y agregamos un GOTO con checkpoint para brincar los else.
5. Llenamos todos los GOTO pendientes con el conteo de cuádruplos.

call_func

Esta sección genera un GOSUB con el nombre de la función, guarda el valor en donde se llamó y revisa que se pasaron los parámetros necesarios para la llamada.

1. Busca que el nombre de la función exista en el directorio, de lo contrario marca error
2. Genera el cuádruplo ERA, con el nombre de la función y guarda los datos de la función encontrada en la pila de parámetros.

list_args

Aqui se encuentra un punto neurálgico utilizado en **call_func**

3. Genera cuádruplo PARAM para cada parámetro en la llamada, y verifica que el tipo enviado sea del mismo tipo que el del parámetro en la función.

type

Sirve para revisar el tipo de función (para revisar el retorno) y para verificar el tipo que se intenta asignar a una variable.

id_exp

Revisa las variables de todas las dimensiones.

1. Revisa que la variable (ID) exista y guarda la información.
2. Abre fondo falso para que la **god_exp** no tome ningun operando fuera de su expresión.
3. Cierra el fondo falso del paso 2
4. Revisa las dimensiones de la variable tomada y que los índices estén dentro de los límites. En el caso de arreglos y matrices, realiza las operaciones necesarias para saber la dirección con la que vamos a trabajar.

assignment

Se toma el operando tomado de la **god_exp** y luego se verifica que sea del mismo tipo al que se le quiere asignar. Crea cuádruplo de asignación.

God exp

1. Termina la expresión haciendo lo mismo que hacen los otros tipos de expresiones (revisar abajo) hasta encontrarse con un fondo falso.

Super_exp/exp/term/fact

1. Todos revisan la pila de operandos a ver si hay una operación de mayor prioridad y en caso de que no, crean el cuádruplo de ese operando revisando que los tipos se pueden combinar y guarda en variables temporales los resultados intermedios.

En el caso de super_exp, que revisa operadores relacionales, ***TUDI*** solo se permite hacer una, por lo que se coloca el punto neurálgico en un lugar diferente.

En el caso de fact se coloca un fondo falso si encuentra un paréntesis para alterar el orden natural de las operaciones.

fact_constants

Tiene las constantes y llamadas a funciones.

int/bool/float

Consiguen y ayudan a crear las direcciones de memoria virtual de las constantes dependiendo del tipo de dato en cuestión.

type_dims

Crean la estructura de tuplas que definen los límites de lo que se puede acceder en un arreglo o matriz al momento de declarar.

Para las consideraciones semánticas de tipo utilizamos un cubo semántico basado en la siguiente tabla:

op1	op2	+	-	*	/	>	>=	==	!=	<	<=	y	o	no
int	**int**	int	int	int	float	bool	bool	bool	bool	bool	bool	bool	bool	error
int	**float**	float	float	float	float	bool	bool	bool	bool	bool	bool	bool	bool	error
int	**bool**	int	int	int	float	bool	bool	bool	bool	bool	bool	bool	bool	error
unary	**int**	error	int	error	error	error	error	error	error	error	error	error	error	error
float	**int**	float	float	float	float	bool	bool	bool	bool	bool	bool	bool	bool	error
float	**float**	float	float	float	float	bool	bool	bool	bool	bool	bool	bool	bool	error
float	**bool**	float	float	float	float	bool	bool	bool	bool	bool	bool	bool	bool	error
unary	**float**	error	float	error	error	error	error	error	error	error	error	error	error	error
char	**char**	error	error	error	error	bool	bool	bool	bool	bool	bool	bool	bool	error

bool	**int**	int	int	int	float	bool	bool	bool	bool	bool	bool	bool	bool	error
bool	**float**	float	float	float	float	bool	bool	bool	bool	bool	bool	bool	bool	error
bool	**bool**	int	int	int	float	bool	bool	bool	bool	bool	bool	bool	bool	error
unary	**bool**	error	error	error	error	error	error	error	error	error	error	error	error	bool

La razón de que el lenguaje sea tan permisivo es por la naturaleza de lógica numérica que tiene el lenguaje. Los errores de tipo pueden suceder al momento de la asignación de valores a las variables y cuando no se regresan los tipos correctos en las funciones

Administración de memoria en compilación

Estructura de datos	Justificación
Quadruple	Este representa los cuádruplos, representados de la siguiente manera: (id, operador, operando izquierdo, operando derecho, temp). Esta estructura es el corazón de las acciones que va a realizar nuestra máquina virtual ya que el operador define la acción que se va a realizar y el resto son los elementos con los que se va a trabajar.
QuadrupleGenerator	Esta estructura de datos contiene la lista de los cuádruplos generados así como un contador de la cantidad de cuádruplos. Existe para guardar los *Quadruples* generados desde el parser y para mantener un control de estos (quiere decir que cuádruplos que están “pendientes” se guardan aquí para luego rellenarse). A nivel de código, también tiene como atributos las pilas de control para facilitar la codificación.
Tabla de variables	Cada función tendrá una tabla de estas. Dentro de esta se añaden variables que están compuestas por: - nombre - tipo - dirección virtual - dimensiones - longitud (tamaño total) El nombre y tipo son definidos por el usuario desde que declara la variable. La dirección virtual es asignada igual en ese momento dependiendo de cual sea el scope donde se declara, esto puede ser a nivel global o a nivel local. Las dimensiones son utilizadas para arreglos y matrices, cada dimensión es representada como una *tupla* con los límites superiores y los límites inferiores (en el caso de las matrices, m1 en lugar del cero, el cual es el límite superior de la dimensión 1 por la obligación de iniciar las casillas en cero). La longitud se utiliza para saber qué tanto espacio va a ocupar la variable, en el caso de variables singulares siempre va a ser uno. La tabla de variables, menos la global, desaparece después de salir de ese *scope*.

Directorio de funciones	Aquí se guarda un diccionario cuyas llaves son los nombres de las funciones y tienen su respectiva tabla de variables. La única función que no se guarda con su nombre es el ambiente global (con un nombre 0) el cuál será la primera entrada de este directorio. Esta estructura de datos se compone de: - start - return_type - params - table - return_address - resources Start representa el índice del cuádruplo en donde inicia a correr la función. La tabla contiene la tabla de variables mencionada anteriormente. Params representa la secuencia paramétrica. Return type guarda el tipo de dato que va a regresar la función, este está definido por el usuario. Return_address guarda la dirección global virtual donde está guardando el resultado de lo que va a regresar la función, lo cuál es útil para la asignación al terminar una llamada a función, claro que esto no aplica para funciones de tipo void. Al final se guarda la cantidad de recursos que va a utilizar la función en el momento de la ejecución, el cuál será útil en la alocación de memoria en el momento de ejecución.
Pila de operadores	Esta pila guarda los operandos (+,-,<, etc.) para los cálculos de las expresiones, esta pila existe para establecer las prioridades entre los operandos (esto quiere decir que se evalúen operaciones con mayor prioridad primero).
Pila de operandos	Aquí están los elementos que son evaluados por los operadores, necesario para saber que operandos corresponden a los operadores de la pila de operadores.
Pila de tipos	Esta pila se encarga de revisar que los tipos entre los operandos haciendo la operación sean compatibles, esto quiere decir que es útil para revisar cuál es el tipo de los operandos en cuestión. Existe para revisar vs el cubo semántico los tipos que van a regresar y para revisar que las asignaciones sean de los mismos tipos.
Pila de saltos	Esta sirve para guardar “checkpoints” de cuádruplos con operaciones “GOTO”. Estos tipos de cuádruplos cuando son creados quedan sin una dirección hacia el cuál ir y al guardar el número del cuádruplo en la pila de saltos nos puede servir para rellenarlo cuando lleguemos al paso al cual deben saltar.
Pila de parámetros	Esta pila sirve para las llamadas a función. En el momento que hay una llamada se guardan aquí los atributos de la función que hizo una llamada, después de esto se utiliza para verificar que la cantidad y tipos de parámetros coincidan con lo estipulado en la función llamada.

Máquina virtual

Equipo de cómputo y utilerías especiales

Para completar la máquina virtual hicimos uso de **Python3** para la codificación del manejo de memoria, los resultados de las operaciones, y la ejecución del código intermedio (nuestra estructura de cuádruplos). Aparte de esto utilizamos la librería **Pygame** para el manejo de gráficos, eventos y renderizado.

Administración de memoria en ejecución

Para la administración de memoria en ejecución, inicialmente tenemos ya nuestra memoria virtual “vacía” ya que realmente todavía no hay nada alocado

Estructura de datos	Justificación
FunctionMemory	Es una estructura de datos que sirve para administrar la memoria de una función (recordemos que Start y Update son funciones obligatorias, por lo que todo código está dentro de una función). La estructura esta conformada por: - memory_map - params_sequence - start - return_address - return_length (tamaño total del dato de retorno) Todas estas variables se construyen con lo que está estipulado en el **directorio de funciones** (revisar sección anterior), los atributos de params_sequence, start y return_address se toman directo del directorio de funciones. Mientras tanto, el memory map es algo que se crea con base a lo escrito en los recursos de la función (que siempre son variables locales y temporales), que nos describe cuánto espacio de memoria va a necesitar una función. El mapa se crea poniendo los valores default desde la dirección base correspondiente en el rango a través de todos los recursos que se va a necesitar. Ejemplo, si necesitaremos dos temporales integers en la funcion, se crearia un espacio de temporales enteras marcadas con 0 ([0, 0]).
VirtualMemory	Esta estructura de datos contiene un contador de donde se encuentra actualmente la variable para cada tipo y scope encontrado en memoria, es útil para alocar los valores de las variables de forma consecutiva. esto quiere decir que si iniciamos con una variable flotante global (direccion 4000), el contador estará en la siguiente dirección disponible (4001), lo que imita el comportamiento de **new**. Las constantes se guardan en el código intermedio como direcciones de memoria, por lo cual sirve para saber el valor de estas constantes. Esta estructura contiene métodos útiles en ejecución como el

	resetear las direcciones temporales y locales dentro de una función (ya que estas solo sirven dentro de la función)
VirtualMachine	Esta es la clase que se encarga de correr todo el código intermedio. Lo primero que hace esta clase es inicializar el lexer y el parser y parsear todo el input (en este caso, un archivo de texto con la terminación .tudi), esto corre lo que ya esta de la gramática. Al final de parsear, le pedimos a este que nos regrese todos los cuádruplos (código intermedio), el directorio de funciones que este genera, la memoria global (que es el mapeo de todas las variables globales con su valor y alocación) y la tabla de constantes (generada con la clase de VirtualMemory en compilación). Aparte de esto tiene un contador que marca el cuádruplo que está corriendo actualmente, una variable que tiene la memoria de la función corriendose en el momento y un par de colas que se explicarán en breve (func_call_stack y goto_stack). La acción fundamental de la clase está en el método start_machine() que hace un ciclo while a través de los cuádruplos hasta llegar al final. Para cada cuádruplo se realiza una acción atómica con su operador, tomando en cuenta los operandos y el valor temporal guardado al final. Las expresiones se calculan de manera lineal mientras que acciones **goto** se encargan de cambiar el valor del contador. Dentro de las acciones también están las de entrada y salida que se manejan como se hace en Python. La asignación funciona directo a menos de que se trate de un apuntador el cual tiene que conseguir un valor distinto para ya saber donde guardarlo. Aquí también se encuentran las acciones correspondiente a funciones y a el manejo del videojuego.
func_call_stack (VM)	Sirve para guardar las llamadas a funciones, claro que la primera que se guarda es la de la función Start ya que es la primera. Esta tiene la utilidad de que administra la memoria de la función que está en ejecución ese momento y que sabe que al encontrar un **ENDFUNC o RET** este tiene que hacer pop hacia la siguiente función.
goto_stack (VM)	Esta sirve para el manejo de llamada de funciones, se hacen adiciones cada vez que se usa la acción del **GOSUB** ya que indicará a dónde debemos de volver apenas terminada la ejecución de una función. Hacemos pop de esta función cuando termina una ejecución de una función void (**ENDFUNC**) o cuando la función retorna un valor (encuentra operando **RET**).

Pruebas de funcionamiento

FACTORIAL ITERATIVO

Código:

```
game factorial_iterativo;
canvas = 10, 10;

func factorial : int (int num) {
    declare {
        int i;
        int res;
    }

    res = 1;
    for (i = 2; i <= num; i = i + 1) {
        res = res * i;
    }

    return res;
}

func Start : void () {
    declare {
        int x;
        int res;
    }

    x = -1;
    while (x < 0) {
        Print("Calcula el factorial de N (N >= 0): ");
        Read(x);
    }

    Print("Resultado: ");
    res = factorial(x);
    Print(res);
    Print("\n");
}

func Update : void () {
    game_over;
}
```

Output:

```
1: [GAME, None, None, factorial_iterativo]
```

2: [CANVAS, 60000, 60000, None]
3: [GOTO, None, None, 36]
4: [=, 60003, 1, 20002]
5: [=, 60004, 1, 20001]
6: [<=, 20001, 20000, 48000]
7: [GOTO_F, 48000, None, 15]
8: [GOTO_V, 48000, None, 12]
9: [+, 20001, 60003, 40000]
10: [=, 40000, 1, 20001]
11: [GOTO, None, None, 6]
12: [*, 20002, 20001, 40001]
13: [=, 40001, 1, 20002]
14: [GOTO, None, None, 9]
15: [RET, None, None, 20002]
16: [ENDFUNC, None, None, int]
17: [-, 60006, 60003, 40000]
18: [=, 40000, 1, 20000]
19: [<, 20000, 60006, 48000]
20: [GOTO_F, 48000, None, 24]
21: [Print, None, None, "Calcula el factorial de N (N >= 0): "]
22: [Read, None, None, 20000]
23: [GOTO, None, None, 19]
24: [Print, None, None, "Resultado: "]
25: [ERA, None, None, factorial]
26: [PARAM, 20000, None, par1]
27: [GOSUB, None, None, factorial]
28: [=, 1, 1, 40001]
29: [=, 40001, 1, 20001]
30: [Print, None, None, 20001]
31: [Print, None, None, "\n"]
32: [ENDFUNC, None, None, None]
33: [GAME_OVER, None, None, None]
34: [GOTO, None, None, 33]
35: [ENDFUNC, None, None, None]
36: [ERA, None, None, Start]
37: [GOSUB, None, None, Start]
38: [ERA, None, None, Update]
39: [GOSUB, None, None, Update]
40: [END_PROGRAM, None, None, None]
Calcula el factorial de N (N >= 0): 10
Resultado: 3628800

Thank you for using TUDI :)

FACTORIAL RECURSIVO

Código:

```
game factorial_recursivo;
canvas = 10, 10;

func factorial : int (int num) {
    if (num <= 1) {
        return 1;
    }

    return num * factorial(num - 1);
}

func Start : void () {
    declare {
        int x;
        int res;
    }

    x = -1;
    while (x < 0) {
        Print("Calcula el factorial de N (N >= 0): ");
        Read(x);
    }

    Print("Resultado: ");
    res = factorial(x);
    Print(res);
    Print("\n");
}

func Update : void () {
    game_over;
}
```

Output:

```
1: [GAME, None, None, factorial_recursivo]
2: [CANVAS, 60000, 60000, None]
3: [GOTO, None, None, 34]
4: [<=, 20000, 60001, 48000]
5: [GOTO_F, 48000, None, 7]
6: [RET, None, None, 60001]
```

7: [ERA, None, None, factorial]
8: [-, 20000, 60001, 40000]
9: [PARAM, 40000, None, par1]
10: [GOSUB, None, None, factorial]
11: [=, 1, 1, 40001]
12: [*, 20000, 40001, 40002]
13: [RET, None, None, 40002]
14: [ENDFUNC, None, None, int]
15: [-, 60004, 60001, 40000]
16: [=, 40000, 1, 20000]
17: [<, 20000, 60004, 48000]
18: [GOTO_F, 48000, None, 22]
19: [Print, None, None, "Calcula el factorial de N (N >= 0): "]
20: [Read, None, None, 20000]
21: [GOTO, None, None, 17]
22: [Print, None, None, "Resultado: "]
23: [ERA, None, None, factorial]
24: [PARAM, 20000, None, par1]
25: [GOSUB, None, None, factorial]
26: [=, 1, 1, 40001]
27: [=, 40001, 1, 20001]
28: [Print, None, None, 20001]
29: [Print, None, None, "\n"]
30: [ENDFUNC, None, None, None]
31: [GAME_OVER, None, None, None]
32: [GOTO, None, None, 31]
33: [ENDFUNC, None, None, None]
34: [ERA, None, None, Start]
35: [GOSUB, None, None, Start]
36: [ERA, None, None, Update]
37: [GOSUB, None, None, Update]
38: [END_PROGRAM, None, None, None]
Calcula el factorial de N (N >= 0): 10
Resultado: 3628800

Thank you for using TUDI :)

FIBONACCI ITERATIVO

Código:

```
game fibonacci_iterativo;
canvas = 10, 10;

func fibonacci : int (int fib) {
```

```
    declare {
        int i, res, prev, temp;
    }

    prev = 1;
    res = 1;
    for (i = 2; i < fib; i = i + 1) {
        temp = res;
        res = res + prev;
        prev = temp;
    }

    return res;
}

func Start : void () {
    declare {
        int x;
        int res;
    }

    x = 0;
    while (x <= 0) {
        Print("Calcula el fibonacci N (N >= 1): ");
        Read(x);
    }

    Print("Resultado: ");
    res = fibonacci(x);
    Print(res);
    Print("\n");
}

func Update : void () {
    game_over;
}
```

Output:

```
1: [GAME, None, None, fibonacci_iterativo]
2: [CANVAS, 60000, 60000, None]
3: [GOTO, None, None, 38]
4: [=, 60005, 1, 20003]
5: [=, 60005, 1, 20002]
```

6: [=, 60006, 1, 20001]
7: [<, 20001, 20000, 48000]
8: [GOTO_F, 48000, None, 18]
9: [GOTO_V, 48000, None, 13]
10: [+, 20001, 60005, 40000]
11: [=, 40000, 1, 20001]
12: [GOTO, None, None, 7]
13: [=, 20002, 1, 20004]
14: [+, 20002, 20003, 40001]
15: [=, 40001, 1, 20002]
16: [=, 20004, 1, 20003]
17: [GOTO, None, None, 10]
18: [RET, None, None, 20002]
19: [ENDFUNC, None, None, int]
20: [=, 60008, 1, 20000]
21: [<=, 20000, 60008, 48000]
22: [GOTO_F, 48000, None, 26]
23: [Print, None, None, "Calcula el fibonacci N (N >= 1): "]
24: [Read, None, None, 20000]
25: [GOTO, None, None, 21]
26: [Print, None, None, "Resultado: "]
27: [ERA, None, None, fibonacci]
28: [PARAM, 20000, None, par1]
29: [GOSUB, None, None, fibonacci]
30: [=, 1, 1, 40000]
31: [=, 40000, 1, 20001]
32: [Print, None, None, 20001]
33: [Print, None, None, "\n"]
34: [ENDFUNC, None, None, None]
35: [GAME_OVER, None, None, None]
36: [GOTO, None, None, 35]
37: [ENDFUNC, None, None, None]
38: [ERA, None, None, Start]
39: [GOSUB, None, None, Start]
40: [ERA, None, None, Update]
41: [GOSUB, None, None, Update]
42: [END_PROGRAM, None, None, None]
Calcula el fibonacci N (N >= 1): 8
Resultado: 21

Thank you for using TUDI :)

FIBONACCI RECURSIVO

Código:

```
game fibonacci_recursivo;
canvas = 10, 10;

func fibonacci : int (int fib) {
    if (fib <= 2) {
        return 1;
    }

    return fibonacci(fib - 2) + fibonacci(fib - 1);
}

func Start : void () {
    declare {
        int x;
        int res;
    }

    x = 0;
    while (x <= 0) {
        Print("Calcula el fibonacci N (N >= 1): ");
        Read(x);
    }

    Print("Resultado: ");
    res = fibonacci(x);
    Print(res);
    Print("\n");
}

func Update : void () {
    game_over;
}
```

Output:

```
1: [GAME, None, None, fibonacci_recursivo]
2: [CANVAS, 60000, 60000, None]
3: [GOTO, None, None, 38]
4: [<=, 20000, 60001, 48000]
5: [GOTO_F, 48000, None, 7]
6: [RET, None, None, 60002]
7: [ERA, None, None, fibonacci]
```

8: [-, 20000, 60001, 40000]
9: [PARAM, 40000, None, par1]
10: [GOSUB, None, None, fibonacci]
11: [=, 1, 1, 40001]
12: [ERA, None, None, fibonacci]
13: [-, 20000, 60002, 40002]
14: [PARAM, 40002, None, par1]
15: [GOSUB, None, None, fibonacci]
16: [=, 1, 1, 40003]
17: [+, 40001, 40003, 40004]
18: [RET, None, None, 40004]
19: [ENDFUNC, None, None, int]
20: [=, 60005, 1, 20000]
21: [<=, 20000, 60005, 48000]
22: [GOTO_F, 48000, None, 26]
23: [Print, None, None, "Calcula el fibonacci N (N >= 1): "]
24: [Read, None, None, 20000]
25: [GOTO, None, None, 21]
26: [Print, None, None, "Resultado: "]
27: [ERA, None, None, fibonacci]
28: [PARAM, 20000, None, par1]
29: [GOSUB, None, None, fibonacci]
30: [=, 1, 1, 40000]
31: [=, 40000, 1, 20001]
32: [Print, None, None, 20001]
33: [Print, None, None, "\n"]
34: [ENDFUNC, None, None, None]
35: [GAME_OVER, None, None, None]
36: [GOTO, None, None, 35]
37: [ENDFUNC, None, None, None]
38: [ERA, None, None, Start]
39: [GOSUB, None, None, Start]
40: [ERA, None, None, Update]
41: [GOSUB, None, None, Update]
42: [END_PROGRAM, None, None, None]
Calcula el fibonacci N (N >= 1): 8
Resultado: 21

Thank you for using TUDI :)

FIND EN UN ARRAY

Código:

```
game find_vec;
canvas = 10, 10;

func create_vec : int[10] (int len) {
    declare {
        int[10] new_vec;
        int i;
    }

    for (i = 0; i < len; i = i + 1) {
        Print("Vec[");
        Print(i);
        Print("]: ");
        Read(new_vec[i]);
    }

    return new_vec;
}

func find : int (int[10] vec, int len, int value) {
    declare {
        int i;
    }

    for (i = 0; i < len; i = i + 1) {
        if (value == vec[i]) {
            return i;
        }
    }

    return -1;
}

func Start : void () {
    declare {
        int[10] vec;
        int len, idx, val;
        int i;
    }

    len = 0;
    while (no (0 < len y len <= 10)) {
```

```
        Print("Capacidad de tu arreglo (0 < N <= 10): ");
        Read(len);
    }

    Print("Rellena tu arreglo:\n");
    vec = create_vec(len);

    Print("Elemento a buscar: ");
    Read(val);

    Print("Buscando ando\n");
    idx = find(vec, len, val);

    if (idx != -1) {
        Print("Valor encontrado en (0-based): ");
        Print(idx);
    } else {
        Print("No se encontro el valor en el vector");
    }
    Print("\n");
}

func Update : void () {
    game_over;
}
```

Output:

```
1: [GAME, None, None, find_vec]
2: [CANVAS, 60000, 60000, None]
3: [GOTO, None, None, 73]
4: [=, 60001, 1, 20011]
5: [<, 20011, 20000, 48000]
6: [GOTO_F, 48000, None, 18]
7: [GOTO_V, 48000, None, 11]
8: [+, 20011, 60004, 40000]
9: [=, 40000, 1, 20011]
10: [GOTO, None, None, 5]
11: [Print, None, None, "Vec["]
12: [Print, None, None, 20011]
13: [Print, None, None, "]: "]
14: [VER, 20011, None, 60000]
15: [+, 20011, 60002, 56000]
16: [Read, None, None, 56000]
```

17: [GOTO, None, None, 8]
18: [RET, None, None, 20001]
19: [ENDFUNC, None, None, int]
20: [=, 60001, 1, 20012]
21: [<, 20012, 20010, 48000]
22: [GOTO_F, 48000, None, 33]
23: [GOTO_V, 48000, None, 27]
24: [+, 20012, 60004, 40000]
25: [=, 40000, 1, 20012]
26: [GOTO, None, None, 21]
27: [VER, 20012, None, 60000]
28: [+, 20012, 60006, 56000]
29: [==, 20011, 56000, 48001]
30: [GOTO_F, 48001, None, 32]
31: [RET, None, None, 20012]
32: [GOTO, None, None, 24]
33: [-, 60001, 60004, 40001]
34: [RET, None, None, 40001]
35: [ENDFUNC, None, None, int]
36: [=, 60001, 1, 20010]
37: [<, 60001, 20010, 48000]
38: [<=, 20010, 60000, 48001]
39: [y, 48000, 48001, 48002]
40: [no, 60001, 48002, 48003]
41: [GOTO_F, 48003, None, 45]
42: [Print, None, None, "Capacidad de tu arreglo (0 < N <= 10): "]
43: [Read, None, None, 20010]
44: [GOTO, None, None, 37]
45: [Print, None, None, "Rellena tu arreglo:\n"]
46: [ERA, None, None, create_vec]
47: [PARAM, 20010, None, par1]
48: [GOSUB, None, None, create_vec]
49: [=, 1, 10, 40000]
50: [=, 40000, 10, 20000]
51: [Print, None, None, "Elemento a buscar: "]
52: [Read, None, None, 20012]
53: [Print, None, None, "Buscando ando\n"]
54: [ERA, None, None, find]
55: [PARAM, 20000, None, par1]
56: [PARAM, 20010, None, par2]
57: [PARAM, 20012, None, par3]
58: [GOSUB, None, None, find]
59: [=, 11, 1, 40010]

```
60: [=, 40010, 1, 20011]
61: [-, 60001, 60004, 40011]
62: [!=, 20011, 40011, 48004]
63: [GOTO_F, 48004, None, 67]
64: [Print, None, None, "Valor encontrado en (0-based): "]
65: [Print, None, None, 20011]
66: [GOTO, None, None, 68]
67: [Print, None, None, "No se encontro el valor en el vector"]
68: [Print, None, None, "\n"]
69: [ENDFUNC, None, None, None]
70: [GAME_OVER, None, None, None]
71: [GOTO, None, None, 70]
72: [ENDFUNC, None, None, None]
73: [ERA, None, None, Start]
74: [GOSUB, None, None, Start]
75: [ERA, None, None, Update]
76: [GOSUB, None, None, Update]
77: [END_PROGRAM, None, None, None]
Capacidad de tu arreglo (0 < N <= 10): 3
Rellena tu arreglo:
Vec[0]: 1
Vec[1]: -1
Vec[2]: 7
Elemento a buscar: -1
Buscando ando
Valor encontrado en (0-based): 1

Thank you for using TUDI :)
```

SORT EN UN ARRAY

Código:

```
game sort_n2;
canvas = 10, 10;

func create_vec : int[10] (int len) {
    declare {
        int[10] new_vec;
        int i;
    }

    for (i = 0; i < len; i = i + 1) {
        Print("Vec[");
        Print(i);
```

```
        Print("]: ");
        Read(new_vec[i]);
    }

    return new_vec;
}

func print_vec : void (int[10] vec, int len) {
    declare {
        int i;
    }

    for (i = 0; i < len; i = i + 1) {
        Print("Vec[");
        Print(i);
        Print("]: ");
        Print(vec[i]);
        Print("\n");
    }
}

func sort : int[10] (int[10] vec, int len) {
    declare {
        int i, j;
        int temp;
    }

    for (i = 0; i < len; i = i + 1) {
        for (j = i; j < len; j = j + 1) {
            if (vec[i] > vec[j]) {
                temp = vec[j];
                vec[j] = vec[i];
                vec[i] = temp;
            }
        }
    }

    return vec;
}

func Start : void () {
    declare {
        int[10] vec;
```

```
        int len;
        int i;
    }

    len = 0;
    while (len <= 0 o len > 10) {
        Print("Capacidad de tu arreglo (0 < N <= 10): ");
        Read(len);
    }

    Print("Rellena tu arreglo:\n");
    vec = create_vec(len);

    Print("Sorteando ando\n");
    vec = sort(vec, len);

    Print("Resultado del sort:\n");
    print_vec(vec, len);
}

func Update : void () {
    game_over;
}
```

Output:

```
1: [GAME, None, None, sort_n2]
2: [CANVAS, 60000, 60000, None]
3: [GOTO, None, None, 101]
4: [=, 60001, 1, 20011]
5: [<, 20011, 20000, 48000]
6: [GOTO_F, 48000, None, 18]
7: [GOTO_V, 48000, None, 11]
8: [+, 20011, 60004, 40000]
9: [=, 40000, 1, 20011]
10: [GOTO, None, None, 5]
11: [Print, None, None, "Vec["]
12: [Print, None, None, 20011]
13: [Print, None, None, "]: "]
14: [VER, 20011, None, 60000]
15: [+, 20011, 60002, 56000]
16: [Read, None, None, 56000]
17: [GOTO, None, None, 8]
18: [RET, None, None, 20001]
```

19: [ENDFUNC, None, None, int]
20: [=, 60001, 1, 20011]
21: [<, 20011, 20010, 48000]
22: [GOTO_F, 48000, None, 35]
23: [GOTO_V, 48000, None, 27]
24: [+, 20011, 60004, 40000]
25: [=, 40000, 1, 20011]
26: [GOTO, None, None, 21]
27: [Print, None, None, "Vec["]
28: [Print, None, None, 20011]
29: [Print, None, None, "]: "]
30: [VER, 20011, None, 60000]
31: [+, 20011, 60005, 56000]
32: [Print, None, None, 56000]
33: [Print, None, None, "\n"]
34: [GOTO, None, None, 24]
35: [ENDFUNC, None, None, None]
36: [=, 60001, 1, 20011]
37: [<, 20011, 20010, 48000]
38: [GOTO_F, 48000, None, 69]
39: [GOTO_V, 48000, None, 43]
40: [+, 20011, 60004, 40000]
41: [=, 40000, 1, 20011]
42: [GOTO, None, None, 37]
43: [=, 20011, 1, 20012]
44: [<, 20012, 20010, 48001]
45: [GOTO_F, 48001, None, 68]
46: [GOTO_V, 48001, None, 50]
47: [+, 20012, 60004, 40001]
48: [=, 40001, 1, 20012]
49: [GOTO, None, None, 44]
50: [VER, 20011, None, 60000]
51: [+, 20011, 60005, 56000]
52: [VER, 20012, None, 60000]
53: [+, 20012, 60005, 56001]
54: [>, 56000, 56001, 48002]
55: [GOTO_F, 48002, None, 67]
56: [VER, 20012, None, 60000]
57: [+, 20012, 60005, 56002]
58: [=, 56002, 1, 20013]
59: [VER, 20012, None, 60000]
60: [+, 20012, 60005, 56003]
61: [VER, 20011, None, 60000]

62: [+, 20011, 60005, 56004]
63: [=, 56004, 1, 56003]
64: [VER, 20011, None, 60000]
65: [+, 20011, 60005, 56005]
66: [=, 20013, 1, 56005]
67: [GOTO, None, None, 47]
68: [GOTO, None, None, 40]
69: [RET, None, None, 20000]
70: [ENDFUNC, None, None, int]
71: [=, 60001, 1, 20010]
72: [<=, 20010, 60001, 48000]
73: [>, 20010, 60000, 48001]
74: [o, 48000, 48001, 48002]
75: [GOTO_F, 48002, None, 79]
76: [Print, None, None, "Capacidad de tu arreglo (0 < N <= 10): "]
77: [Read, None, None, 20010]
78: [GOTO, None, None, 72]
79: [Print, None, None, "Rellena tu arreglo:\n"]
80: [ERA, None, None, create_vec]
81: [PARAM, 20010, None, par1]
82: [GOSUB, None, None, create_vec]
83: [=, 1, 10, 40000]
84: [=, 40000, 10, 20000]
85: [Print, None, None, "Sorteando ando\n"]
86: [ERA, None, None, sort]
87: [PARAM, 20000, None, par1]
88: [PARAM, 20010, None, par2]
89: [GOSUB, None, None, sort]
90: [=, 11, 10, 40010]
91: [=, 40010, 10, 20000]
92: [Print, None, None, "Resultado del sort:\n"]
93: [ERA, None, None, print_vec]
94: [PARAM, 20000, None, par1]
95: [PARAM, 20010, None, par2]
96: [GOSUB, None, None, print_vec]
97: [ENDFUNC, None, None, None]
98: [GAME_OVER, None, None, None]
99: [GOTO, None, None, 98]
100: [ENDFUNC, None, None, None]
101: [ERA, None, None, Start]
102: [GOSUB, None, None, Start]
103: [ERA, None, None, Update]
104: [GOSUB, None, None, Update]

```
105: [END_PROGRAM, None, None, None]
Capacidad de tu arreglo (0 < N <= 10): 5
Rellena tu arreglo:
Vec[0]: -1
Vec[1]: 65
Vec[2]: -10
Vec[3]: 5
Vec[4]: 3
Sorteando ando
Resultado del sort:
Vec[0]: -10
Vec[1]: -1
Vec[2]: 3
Vec[3]: 5
Vec[4]: 65

Thank you for using TUDI :)
```

MULTIPLICACION DE MATRICES

Código:

```
game multiplicacion_matriz;
canvas = 10, 10;

func print_mat : void (int[10, 10] mat, int m, int n) {
    declare {
        int i, j;
    }

    for (i = 0; i < m; i = i + 1) {
        for (j = 0; j < n; j = j + 1) {
            Print("Mat[");
            Print(i);
            Print(", ");
            Print(j);
            Print("]: ");
            Print(mat[i, j]);
            Print("\n");
        }
    }
}

func create_mat : int[10, 10] (int m, int n) {
    declare {
```

```
    int[10, 10] new_mat;
    int i, j;
  }

  for (i = 0; i < m; i = i + 1) {
    for (j = 0; j < n; j = j + 1) {
      Print("Mat[");
      Print(i);
      Print(", ");
      Print(j);
      Print("]: ");
      Read(new_mat[i, j]);
    }
  }

  return new_mat;
}


func mat_mul : int[10, 10] (int[10, 10] mat_a, int[10, 10] mat_b, int m, int p, int n) {
  declare {
    int[10, 10] mat_c;
    int i, j, k;
  }

  for (i = 0; i < m; i = i + 1) {
    for (j = 0; j < p; j = j + 1) {
      for (k = 0; k < n; k = k + 1) {
        mat_c[i, j] = mat_c[i, j] + mat_a[i, k] * mat_b[k, j];
      }
    }
  }

  return mat_c;
}

func Start : void () {
  declare {
    int m, n, p;
    int i, j, k;
    int[10, 10] mat_a, mat_b, mat_res;
  }
```

```
    Print("Multiplicación de matrices -> M(m x n) * M(n x p) = M(m x p)\n");

    m = 0;
    while (m <= 0 o m > 10) {
        Print("Dimension m (0 < m <= 10): ");
        Read(m);
    }

    n = 0;
    while (n <= 0 o n > 10) {
        Print("Dimension n (0 < n <= 10): ");
        Read(n);
    }

    p = 0;
    while (p <= 0 o p > 10) {
        Print("Dimension p (0 < p <= 10): ");
        Read(p);
    }

    Print("Rellena matriz A:\n");
    mat_a = create_mat(m, n);

    Print("Rellena matriz B:\n");
    mat_b = create_mat(n, p);

    Print("Multiplicando matrices ando\n");
    mat_res = mat_mul(mat_a, mat_b, m, p, n);

    Print("Resultado de MatA x MatB:\n");
    print_mat(mat_res, m, p);
}

func Update : void () {
    game_over;
}
```

Output:

```
1: [GAME, None, None, multiplicacion_matriz]
2: [CANVAS, 60000, 60000, None]
3: [GOTO, None, None, 170]
4: [=, 60001, 1, 20102]
5: [<, 20102, 20100, 48000]
```

6: [GOTO_F, 48000, None, 32]
7: [GOTO_V, 48000, None, 11]
8: [+, 20102, 60004, 40000]
9: [=, 40000, 1, 20102]
10: [GOTO, None, None, 5]
11: [=, 60001, 1, 20103]
12: [<, 20103, 20101, 48001]
13: [GOTO_F, 48001, None, 31]
14: [GOTO_V, 48001, None, 18]
15: [+, 20103, 60004, 40001]
16: [=, 40001, 1, 20103]
17: [GOTO, None, None, 12]
18: [Print, None, None, "Mat["]
19: [Print, None, None, 20102]
20: [Print, None, None, ", "]
21: [Print, None, None, 20103]
22: [Print, None, None, "]: "]
23: [VER, 20102, None, 60000]
24: [*, 20102, 60000, 40002]
25: [VER, 20103, None, 60000]
26: [+, 40002, 20103, 40003]
27: [+, 40003, 60005, 56000]
28: [Print, None, None, 56000]
29: [Print, None, None, "\n"]
30: [GOTO, None, None, 15]
31: [GOTO, None, None, 8]
32: [ENDFUNC, None, None, None]
33: [=, 60001, 1, 20102]
34: [<, 20102, 20000, 48000]
35: [GOTO_F, 48000, None, 60]
36: [GOTO_V, 48000, None, 40]
37: [+, 20102, 60004, 40000]
38: [=, 40000, 1, 20102]
39: [GOTO, None, None, 34]
40: [=, 60001, 1, 20103]
41: [<, 20103, 20001, 48001]
42: [GOTO_F, 48001, None, 59]
43: [GOTO_V, 48001, None, 47]
44: [+, 20103, 60004, 40001]
45: [=, 40001, 1, 20103]
46: [GOTO, None, None, 41]
47: [Print, None, None, "Mat["]
48: [Print, None, None, 20102]

49: [Print, None, None, ", "]
50: [Print, None, None, 20103]
51: [Print, None, None, "]: "]
52: [VER, 20102, None, 60000]
53: [*, 20102, 60000, 40002]
54: [VER, 20103, None, 60000]
55: [+, 40002, 20103, 40003]
56: [+, 40003, 60006, 56000]
57: [Read, None, None, 56000]
58: [GOTO, None, None, 44]
59: [GOTO, None, None, 37]
60: [RET, None, None, 20002]
61: [ENDFUNC, None, None, int]
62: [=, 60001, 1, 20303]
63: [<, 20303, 20200, 48000]
64: [GOTO_F, 48000, None, 109]
65: [GOTO_V, 48000, None, 69]
66: [+, 20303, 60004, 40000]
67: [=, 40000, 1, 20303]
68: [GOTO, None, None, 63]
69: [=, 60001, 1, 20304]
70: [<, 20304, 20201, 48001]
71: [GOTO_F, 48001, None, 108]
72: [GOTO_V, 48001, None, 76]
73: [+, 20304, 60004, 40001]
74: [=, 40001, 1, 20304]
75: [GOTO, None, None, 70]
76: [=, 60001, 1, 20305]
77: [<, 20305, 20202, 48002]
78: [GOTO_F, 48002, None, 107]
79: [GOTO_V, 48002, None, 83]
80: [+, 20305, 60004, 40002]
81: [=, 40002, 1, 20305]
82: [GOTO, None, None, 77]
83: [VER, 20303, None, 60000]
84: [*, 20303, 60000, 40003]
85: [VER, 20304, None, 60000]
86: [+, 40003, 20304, 40004]
87: [+, 40004, 60007, 56000]
88: [VER, 20303, None, 60000]
89: [*, 20303, 60000, 40005]
90: [VER, 20304, None, 60000]
91: [+, 40005, 20304, 40006]

92: [+, 40006, 60007, 56001]
93: [VER, 20303, None, 60000]
94: [*, 20303, 60000, 40007]
95: [VER, 20305, None, 60000]
96: [+, 40007, 20305, 40008]
97: [+, 40008, 60005, 56002]
98: [VER, 20305, None, 60000]
99: [*, 20305, 60000, 40009]
100: [VER, 20304, None, 60000]
101: [+, 40009, 20304, 40010]
102: [+, 40010, 60011, 56003]
103: [*, 56002, 56003, 40011]
104: [+, 56001, 40011, 40012]
105: [=, 40012, 1, 56000]
106: [GOTO, None, None, 80]
107: [GOTO, None, None, 73]
108: [GOTO, None, None, 66]
109: [RET, None, None, 20203]
110: [ENDFUNC, None, None, int]
111: [Print, None, None, "MultiplicaciÃ³n de matrices -> M(m x n) * M(n x p) = M
(m x p)\n"]
112: [=, 60001, 1, 20000]
113: [<=, 20000, 60001, 48000]
114: [>, 20000, 60000, 48001]
115: [o, 48000, 48001, 48002]
116: [GOTO_F, 48002, None, 120]
117: [Print, None, None, "Dimension m (0 < m <= 10): "]
118: [Read, None, None, 20000]
119: [GOTO, None, None, 113]
120: [=, 60001, 1, 20001]
121: [<=, 20001, 60001, 48003]
122: [>, 20001, 60000, 48004]
123: [o, 48003, 48004, 48005]
124: [GOTO_F, 48005, None, 128]
125: [Print, None, None, "Dimension n (0 < n <= 10): "]
126: [Read, None, None, 20001]
127: [GOTO, None, None, 121]
128: [=, 60001, 1, 20002]
129: [<=, 20002, 60001, 48006]
130: [>, 20002, 60000, 48007]
131: [o, 48006, 48007, 48008]
132: [GOTO_F, 48008, None, 136]
133: [Print, None, None, "Dimension p (0 < p <= 10): "]

134: [Read, None, None, 20002]
135: [GOTO, None, None, 129]
136: [Print, None, None, "Rellena matriz A:\n"]
137: [ERA, None, None, create_mat]
138: [PARAM, 20000, None, par1]
139: [PARAM, 20001, None, par2]
140: [GOSUB, None, None, create_mat]
141: [=, 1, 100, 40000]
142: [=, 40000, 100, 20006]
143: [Print, None, None, "Rellena matriz B:\n"]
144: [ERA, None, None, create_mat]
145: [PARAM, 20001, None, par1]
146: [PARAM, 20002, None, par2]
147: [GOSUB, None, None, create_mat]
148: [=, 1, 100, 40100]
149: [=, 40100, 100, 20106]
150: [Print, None, None, "Multiplicando matrices ando\n"]
151: [ERA, None, None, mat_mul]
152: [PARAM, 20006, None, par1]
153: [PARAM, 20106, None, par2]
154: [PARAM, 20000, None, par3]
155: [PARAM, 20002, None, par4]
156: [PARAM, 20001, None, par5]
157: [GOSUB, None, None, mat_mul]
158: [=, 101, 100, 40200]
159: [=, 40200, 100, 20206]
160: [Print, None, None, "Resultado de MatA x MatB:\n"]
161: [ERA, None, None, print_mat]
162: [PARAM, 20206, None, par1]
163: [PARAM, 20000, None, par2]
164: [PARAM, 20002, None, par3]
165: [GOSUB, None, None, print_mat]
166: [ENDFUNC, None, None, None]
167: [GAME_OVER, None, None, None]
168: [GOTO, None, None, 167]
169: [ENDFUNC, None, None, None]
170: [ERA, None, None, Start]
171: [GOSUB, None, None, Start]
172: [ERA, None, None, Update]
173: [GOSUB, None, None, Update]
174: [END_PROGRAM, None, None, None]
MultiplicaciÃ³n de matrices -> M(m x n) * M(n x p) = M(m x p)
Dimension m (0 < m <= 10): 3

Dimension n (0 < n <= 10): 2
Dimension p (0 < p <= 10): 2
Rellena matriz A:
Mat[0, 0]: 5
Mat[0, 1]: 3
Mat[1, 0]: 1
Mat[1, 1]: 6
Mat[2, 0]: 4
Mat[2, 1]: 3
Rellena matriz B:
Mat[0, 0]: 1
Mat[0, 1]: 8
Mat[1, 0]: 6
Mat[1, 1]: 4
Multiplicando matrices ando
Resultado de MatA x MatB:
Mat[0, 0]: 23
Mat[0, 1]: 52
Mat[1, 0]: 37
Mat[1, 1]: 32
Mat[2, 0]: 22
Mat[2, 1]: 44

Thank you for using TUDI :)

Apéndice

A.

Semana	L	Mi	V	AVANCE	Contenido esperado de la entrega
Septiembre (26- 30)	26	28	30	#1	Análisis de Léxico y Sintaxis (Scanner y Parser)
Octubre (3 - 7)	3	5	7	#2	Semántica Básica de Variables: Directorio de Procedimientos y Tablas de Variables Semántica Básica de Expresiones: Tabla de Consideraciones semánticas (Cubo Semántico)
Octubre (10 - 14)	10	12	14	#3	Generacion de Código de Expresiones Aritméticas y estatutos secuenciales: Asignación, Lectura, etc. Generacion de Código de Estatutos Condicionales: Decisiones
Octubre (17 – 21)	17	19	21	#4	Generacion de Código de Estatutos Condicionales: Ciclos
Octubre (24 - 28)	24	26	28	---	***S E M A N A - i***
Octubre/ Noviembre (31 – 4)	31	2	4	#5	Generacion de Código de Funciones Mapa de Memoria de Ejecución para la Máquina Virtual Máquina Virtual: Ejecución de Expresiones Aritméticas
Noviembre (7 – 11)	7	9	11	#6	Generacion de Código de Arreglos /Tipos estructurados Máquina Virtual: Ejecución de Estatutos Secuenciales y Condicionales
Noviembre (14– 18)	14	16	18	#7	1era versión de la Documentación Máquina Virtual: Ejecución de Módulos y Arreglos Generacion de Código y Máquina Virtual para una parte de la aplicación particular
Noviembre (21 - 25)	21	23	25	FINAL	***ENTREGA FINAL DEL PROYECTO MARTES Noviembre 22, 12:00pm***

Diseño de Compiladores--PROGRAMACIÓN DE AVANCES
Semestre Agosto-Diciembre 2022

B. Liga de Drive con las capturas de pantalla de la bitácora y commits: https://drive.google.com/drive/folders/1vl-7BoIG5X_Mv1ZdUD2fw_6VKsxi42sj?usp=sharing

Manual de Usuario

Quick reference manual

La estructura general de un programa escrito en ***TUDI*** es:

```
/* Nombre del programa y tamaño de canvas */
game <name>;
canvas = 10, 10;

/* Declarar variables globales */
declare {
   float a, b, c;
```

```
    int[5] vector;
}

/* Definir módulos */
func DoSomethingA : int () {
    declare {
        int x;
    }
    x = 1;
    return x;
}

/* Función Start */
func Start : void () {
    /* Código que se llama únicamente una vez antes de todo */
}

/* Función Update */
func Update : void () {
    /* Código que se llama en cada frame */
}
```

Estructura de estatutos y bloques

Declaración de Variables

declare {
    tipo nombre_var1, nombre_var2, nombre_var3;
    tipo nombre_variable;
    tipo[5] arr;
    tipo[5,5] mat;
}

Para tipos atómicos se podrán crear de los siguientes: **int**, **float**, y **bool**.

Para elementos estructurales, se deberá de declarar previamente el tamaño de estos, se podrán crear arreglos de 1 y máximo 2 dimensiones dónde las de dos dimensiones se declararán así: **arr[N,M]**
Y se podrá acceder hasta los valores N-1 y M-1. N y M deben de ser (int) para la declaración de tamaños.

Para tipos específicos del lenguaje tenemos: **sprite**.

Asignación de Variables

La asignación de variables seguirá el formato tradicional de otros lenguajes, con prioridad izquierda, que tendrá este formato:

nombre_variable = valor_del_tipo_asignado;

El lenguaje no tendrá soporte para asignación de variables en el momento de la declaración.

Declaración de funciones

func *nombre_función* : *tipo* (*lista_params*) {
    // BLOQUE DE CÓDIGO
    // Si es que no es void…
    return *valor_tipo;*
}

Las funciones se declararán siempre escribiendo la palabra reservada **“func”** seguido del **nombre de la función**, para después poner el token **“:”** y luego entre paréntesis una **lista de parámetros** cuyos parámetros estarán escritos de esta manera y separados por comas: ***tipo_param1 nombre_param1***.
Al final se pedirá que haga ***return*** de un valor del mismo tipo de dato que se pide en la función, a menos que sea de tipo ***void***.

Estatuto de repetición for

for (ASIGNACIÓN ; EXPRESIÓN; ASIGNACIÓN) {
  // Bloque de código
}

El estatuto de repetición **for** sigue la estructura clásica de otros lenguajes primero se tiene una asignación inicial, después una condición que se checa después de cada ejecución, y después una asignación para actualizar la variable en la condición, aunque a diferencia de otros lenguajes, aquí en **TUDI** no es posible declarar y asignar la variable inicial.

Estatuto de repetición while

while (EXPRESIÓN) {
  // Bloque de código
}

El estatuto de repetición **while** sigue la estructura clásica de otros lenguajes en el que la EXPRESIÓN siempre tiene que ser un número distinto a 0 para ser tomado como positivo.

Estatuto de decisión

```
if (EXPRESIÓN) {
   // Bloque de código
} else if (EXPRESIÓN) {
   // Bloque de código
} else {
   // Bloque de código
}
```

El estatuto de decisión está compuesto por **if** y **else**. Primero inicia con un **if** y después puede seguir concatenando **else if**, y solamente puede existir un **else** solitario si este está al final de todas.

Expresiones

Las expresiones se crearán de la misma manera que se hace en lenguajes comunes como C + + o Python. Habrán operadores aritméticos, de relación y lógicos (+ ,- ,* ,/ ,> ,<, == , y , o) y tendrán el orden de precedencia natural teniendo paréntesis para cambiar el orden.

Entradas y salidas

Print

```
Print(“letrero:\n”);
Print(var1);
```

La función **Print** está diseñada para sólo recibir un argumento, este puede ser un letrero, que no se guarda en memoria, o cualquier variable, se imprimirá de la misma manera en la que Python lo maneja. Por default los **Prints** no hacen un newline al terminar de imprimirse, por lo que se tiene que escribir el caracter de escape **\n** para hacer esta acción.

Read

```
Read(var1);
Read(var2);
```

Tipos de datos

- **int/entero**: Servirá para guardar datos enteros y también para tener la posibilidad de crear contadores para ciclos
- **float/flotante**: Servirá para que el usuario pueda guardar datos con información de número decimal

- **bool/booleano**: Servirá para que el usuario pueda guardar datos con información de verdadero o falso.
- **array/arreglo y matriz**: Servirá para poder guardar varios datos (del mismo tipo) dentro de una sola variable cuyo número de casillas será declarado en el momento de crear la variable. Se podrán declarar las dimensiones por dentro utilizando una coma en su declaración. Ejemplo: int[2,2] arr.

VIDEO-DEMO

https://youtu.be/MQrD5GQkiPE

Referencias

https://www.dabeaz.com/ply/
https://www.pygame.org/