



Tecnológico de Monterrey

Proyecto Final para la clase Diseño de Compiladores AD22
Proyecto Individual

Lenguaje LEAD 2.0

Profesores:

Ing. Elda Guadalupe Quiroga González
Dr. Héctor Gibrán Ceballos Cancino

Joel Hiram Chávez Verástegui

Monterrey, Nuevo León, México
22 de Noviembre del 2022

Índice

Descripción del Proyecto	2
Propósito y Alcance del Proyecto	2
Requerimientos y Casos de Uso	2
Proceso de Desarrollo del Proyecto	4
Historial Git	5
Reflexión Sobre el Proyecto	5
Descripción del Lenguaje	7
Nombre del Lenguaje	7
Características Generales del Lenguaje	7
Errores que pueden Ocurrir	8
Descripción del Compilador	9
Equipo de Cómputo, Lenguaje y Utilerías Especiales	9
Descripción del Análisis Léxico	9
Descripción del Análisis de Sintaxis	11
Generación de Código Intermedio y Análisis Semántico	14
Diagramas de Sintaxis	16
Estructura del Lenguaje	27
Consideraciones Semánticas	31
Administración de Memoria	31
Descripción de la Máquina Virtual	34
Equipo de Cómputo, Lenguaje y Utilerías	34
Administración de la Memoria	34
Pruebas de Funcionamiento del Lenguaje	36
Referencias	45

Descripción del Proyecto

Propósito y Alcance del Proyecto

El propósito de este proyecto es realizar un lenguaje de programación desde cero. Esto usando como base todas las materias de la carrera de Ingeniería en Tecnologías Computacionales, que incluyen las clases desde fundamentos de programación y programación orientada a objetos hasta estructuras de datos y sistemas operativos. El alcance de este proyecto es crear un lenguaje de programación que pueda usar vectores y/o matrices con el objetivo de realizar análisis estadísticos. El lenguaje de programación podrá usar funciones, recursión, vectores y los tipos de variables que ya conocemos como ints, floats y chars. Asimismo, tendrá la posibilidad de usar ciclos como el do-while y el while.

Requerimientos y Casos de Uso

En el caso de los requerimientos del lenguaje de programación, se definieron los siguientes:

1. Tener int, float y char como tipos de dato y tipos de variables
2. Tener al menos dos tipos de ciclos (while, do-while)
3. Tener un tipo de condicional (if, if-else)
4. Tener funciones implementadas
 - a. Las funciones deberán tener variables de retorno o ser void
 - b. Las funciones deberán poder ser usadas recursivamente
5. Poder realizar operaciones aritméticas
6. Poder realizar llamadas a funciones
7. Poder hacer uso de vectores
8. Poder pedir al usuario que introduzca información
9. Poder imprimir datos en pantalla

Para los casos de uso del programa, se definieron los que se pidieron como parte de la entrega del proyecto y algunos otros más sencillos para verificar la funcionalidad del proyecto:

- **Vectores:** Revisar la funcionalidad de los vectores, asignando valores y escribiendo valores en pantalla.
- **Recursión y Parámetros Recursivos:** Revisar que la recursión funcione, así como la asignación de parámetros recursivos y el correcto funcionamiento del retorno en todos los niveles recursivos.
- **Factorial Cíclico:** Revisar la funcionalidad de las funciones así como el uso de ciclos while y condicionales if-else
- **Factorial Recursivo:** Revisar la funcionalidad de la recursividad en un ejemplo real, usando condicionales, valores de retorno y parámetros a distintos niveles de recursividad.
- **Fibonacci Cíclico:** Revisar la funcionalidad de las funciones, así como el read, write, ciclos y expresiones
- **Fibonacci Recursivo:** Revisar la funcionalidad de la recursividad en otro ejemplo real, usando múltiples llamadas por función, read, write, condicionales if-else y ciclos while.
- **Recursión Infinita:** Revisar el buen manejo de la memoria haciendo una recursión infinita, para que llegue a su límite y nos muestre el error en la pantalla

Proceso de Desarrollo del Proyecto

El proceso de desarrollo del proyecto se hizo por medio de avances semanales. Estos avances se pueden ver a continuación:

- **Primera Semana:** Se definió la propuesta para el lenguaje de programación LEAD. En esta propuesta se incluyó información de cómo se debía ver el lenguaje de programación, su estructura general y algunos ejemplos de código. Asimismo, se desarrollaron diagramas de sintaxis para definir la estructura general del lenguaje.
- **Segunda Semana:** Empieza el desarrollo del lenguaje de programación, durante estos siete días se genera la gramática del lenguaje de programación, con base en lo que se había definido en la propuesta inicial. Se hacen pruebas con código para verificar que el programa esté funcionando correctamente. Se identificaron errores y se modificaron los diagramas de sintaxis. Por último se crea el oráculo para revisar las operaciones.
- **Tercera Semana:** No se avanzó durante esta semana debido a que era semana de exámenes parciales y entregas de proyectos parciales. Esto atrasó el desarrollo del lenguaje de programación
- **Cuarta Semana:** No se avanzó durante esta semana debido a que continuamos en exámenes y entregas de proyectos. En esta semana se atrasó el desarrollo del proyecto al igual que la pasada.
- **Quinta Semana:** No se avanzó durante esta semana debido a factores externos al Tec. En este caso fue una semana con bastante trabajo en mis prácticas, por lo que le tuve que poner más atención a ese tema.
- **Sexta Semana:** Se investiga y se crea el directorio de funciones y el directorio de variables usando Hash Tables en Python. Esto se divide en una tabla de funciones y en

una tabla de variables, cada una guarda información distinta que se usa en ejecución.

Asimismo, se asigna una dirección virtual a las variables que se crean.

- **Séptima Semana:** Se completan las expresiones y sus puntos neurálgicos. Se crean los cuádruplos y los puntos neurálgicos del read, write, asignación, if/if-else, while, do-while, funciones, retornos, llamadas y parámetros. Se corrige el oráculo, se revisa si la variable que se quiere crear ya existe, se cuentan las variables y parámetros por función, se crea la máquina virtual y se crea el manejo de memoria dentro de la misma. Asimismo, se arreglan ciertos aspectos del while y do/while, ya que tenían problemas en los goto y se completan las operaciones de suma, resta, multiplicación y división, así como las operaciones booleanas.
- **Octava Semana:** Se logra la recursión en las funciones, se corrige la asignación del retorno cuando se hacen llamadas recursivas y se revisa la memoria de las funciones para ajustar los vectores. Se completan los vectores y se logran asignar y hacer operaciones con sus valores (sin embargo, no se pueden usar expresiones para obtener un índice del vector, solamente números enteros).

Historial Git


Al ser un proyecto individual, todo el desarrollo del proyecto se llevó a cabo de manera local, por lo que no se usó GitHub para los avances del proyecto. Todos los avances se reportaron en el documento llamado “ReadMe.txt” incluido en el Zip del proyecto.

Reflexión Sobre el Proyecto

Realizar un compilador es algo que me daba muchísimo miedo. Miedo de empezar, miedo de no lograrlo y miedo al fracaso. Este miedo me mantuvo atrapado durante mucho tiempo hasta que un día decidí empezar a programar, sin rumbo específico pero con un objetivo claro. Considero que el realizar este proyecto, mi propio lenguaje de programación fue algo que avivó algo en mí. Ese deseo de seguir aprendiendo, de usar las matemáticas y los

algoritmos y seguir avanzando en este camino de la programación. Usar todo el conocimiento de casi todas las materias de la carrera no fue nada sencillo, pero creo que fue de mucho provecho para cerrar (o casi cerrar) con broche de oro la carrera.

Considero que este ha sido uno de los proyectos con los que más he aprendido y a los que más les he dedicado tiempo, investigación y cariño. El hecho de haber hecho este proyecto solo no fue un capricho mío para no tener equipo, sino que fue un reto para obligarme a aprender cosas nuevas, investigar y que todo dependiera de mí. Hubo cosas que no pude lograr, como todo en la vida, pero creo que el trabajo que hice fue algo muy bueno y de lo que me siento muy orgulloso. Solo algunas personas y las paredes de mi cuarto sabrán lo feliz que me ponía cuando algo funcionaba y lo estresado que me sentía cuando no encontraba el error en la lógica de mi compilador. Es cierto que pude haber hecho más, pero creo que a mis capacidades, tiempo y esfuerzo el proyecto que hice me hace sentir orgulloso de mi trabajo.



Joel Hiram Chávez Verástegui

Descripción del Lenguaje

Nombre del Lenguaje

El nombre de mi lenguaje de programación es LEAD 2.0. El LEAD viene de la palabra liderazgo en inglés, algo que me apasiona y que dediqué mucho tiempo de mi vida universitaria. El 2.0 viene de que este es mi segundo intento, ya que por problemas de salud di de baja la materia el semestre pasado a la entrega de este proyecto.

Características Generales del Lenguaje

LEAD 2.0 es un lenguaje de programación básico que busca ser fácil de aprender y con una sintaxis bastante sencilla, expresiva y lógica. Este lenguaje tiene como objetivo ser un pequeño escalón antes de pasar a los lenguajes de programación de alto nivel, así como enseñar cómo funciona un lenguaje de programación “tras bambalinas” viendo las instrucciones que se generan y cómo se ejecutan.

LEAD 2.0 es un lenguaje de programación que tiene como objetivo la realización de cálculos estadísticos mediante vectores. Este lenguaje de programación soporta únicamente tres tipos de valores, que se pueden usar tanto en funciones como en variables, los cuales son int, float y char. Sin embargo, también se pueden realizar operaciones booleanas. El lenguaje permite escribir expresiones y variables, así como strings (aunque únicamente se pueden utilizar al momento de escribir). Asimismo, puede leer variables y guardar sus valores. Por otro lado, el lenguaje LEAD tiene la capacidad de ejecutar if/if-else, así como while y do-while. El lenguaje de programación soporta vectores, pero no se puede tomar una expresión para llegar a un índice dentro de un vector. LEAD permite el uso de funciones, así como recursión dentro de las mismas.

Errores que pueden Ocurrir

Errores en Compilación	
"Function does not exist"	La función a la que quiere llamar el usuario no existe
"El tipo del argumento de la llamada no es igual al tipo de parámetro"	El usuario declaró un argumento en una llamada que no corresponde con el tipo de parámetro
"El retorno no es del mismo valor que el tipo de función"	El valor de retorno que el usuario quiere regresar no corresponde con el tipo de función
"There's a variable that already exists"	La variable que el usuario definió ya existe
"Type Mismatch"	El usuario quiere hacer una operación que entre dos tipos de variable que no se puede hacer
"Type Mismatch, IF needs a Bool Expression to Succeed"	El usuario no usó una expresión booleana al momento de hacer un IF
"Type Mismatch, DO WHILE needs a Bool Expression to Succeed"	El usuario no usó una expresión booleana al momento de hacer un do-while
"No hay suficiente memoria"	El usuario sobrepasó la cantidad de memoria asignada a un tipo de variable
Errores en Ejecución	
"No hay suficiente memoria"	El usuario sobrepasó la cantidad de memoria asignada a un tipo de variable

Descripción del Compilador

Equipo de Cómputo, Lenguaje y Utilerías Especiales

El lenguaje de programación se creó utilizando Python y la librería PLY. De igual forma la máquina virtual se creó usando Python. En cuanto a librerías, además de PLY, no se usó ninguna extra, los cuádruplos y las pilas se crearon usando listas y el equipo de cómputo utilizado para el proyecto fue MacOS con Python 3.9.7, utilizando VSCode como IDE.

Descripción del Análisis Léxico

La descripción del análisis léxico contiene tanto las palabras reservadas como los tokens que se utilizaron al momento de desarrollar LEAD. Los tokens se pueden ver a continuación en la siguiente tabla:

Nombre	Expresión Regular	Valor
PLUS	$r'\backslash+'$	+
MINUS	$r'\backslash-'$	-
MULT	$r'\backslash*'$	*
DIV	$r'\backslash/'$	/
EQUAL	$r'\backslash='$	=
NEQUAL	$r'\backslash!='$!=
GTHAN	$r'\backslash>'$	>
LTHAN	$r'\backslash<'$	<
GEQUALTHAN	$r'\backslash>='$	>=
LEQUALTHAN	$r'\backslash<='$	<=
EEQUAL	$r'\backslash=='$	==
LPAR	$r'\backslash('$	(
RPAR	$r'\backslash)'$)
LBRACKET	$r'\backslash\{'$	{

RBRACKET	r'\}'	}
LSQUARE	r'\['	[
RSQUARE	r'\]']
COLON	r'\.'	:
SCOLON	r'\,'	;
COMMA	r'\,'	,
DOT	r'\.'	.
AND	r'\&&'	&&
OR	r'\ '	
CTEINT	r'[+-]?[0-9]+'	Número entero positivo o negativo
CTEF	r'[0-9]*\.[0-9]+'	Número flotante
CTEC	r'(\^[^\])'	Un caracter
CSTR	r'"(.*)*\"'	Un string
IGNORE	" \t"	Espacios a ignorar
ID	r'[a-zA-Z_][a-zA-Z_0-9]*'	Cualquier string que no empiece en número
COMMENT	r'\[\\\$\\\$][.]*'	Un comentario, empieza con \$\$
NEWLINE	r'\n+'	Nueva línea

Las palabras reservadas para el lenguaje de programación LEAD se pueden encontrar en la siguiente Tabla:

Palabras Reservadas
program
main
var
void

int
float
char
write
read
if
else
while
for
function
call
return
do

Descripción del Análisis de Sintaxis

Nombre	Regla
programa	programa : PROGRAM ID SCOLON vars funcion main
main	main : MAIN p_n_scopeNow p_n_addFunc LPAR parametro RPAR vars p_n_empiezaMain bloque p_n_addMemory
vars	vars : VAR tipo COLON vars2 SCOLON varTemp empty
varTemp	varTemp : tipo COLON vars2 SCOLON varTemp empty
vars2	vars2 : ID p_n_nameID p_n_cambioTamVar p_n_saveVar ID p_n_nameID p_n_cambioTamVar p_n_saveVar COMMA vars2 ID p_n_nameID LSQUARE CTEINT p_n_cambioTamArr RSQUARE p_n_saveVar ID p_n_nameID LSQUARE CTEINT p_n_cambioTamArr RSQUARE p_n_saveVar COMMA vars2
tipo	tipo : INT p_n_typeNow

	FLOAT p_n_typeNow CHAR p_n_typeNow
tipoFunc	tipoFunc : INT p_n_typeNowFunc FLOAT p_n_typeNowFunc CHAR p_n_typeNowFunc VOID p_n_typeNowFunc
parametro	parametro : empty param2
param2	param2 : tipo varParamTemp tipo varParamTemp COMMA param2
varParam	varParamTemp : ID p_n_nameID p_n_cambioTamVar p_n_saveVar p_n_tamPar ID p_n_nameID LSQUARE CTEINT p_n_cambioTamArr p_n_saveVar RSQUARE
funcion	FUNCTION tipoFunc ID p_n_scopeNow p_n_addFunc LPAR parametro RPAR vars p_n_iniciaFunc p_n_addMemory bloque p_n_endFunc funcion empty
bloque	bloque : LBRACKET bloqueTemp RBRACKET
bloqueTemp	bloqueTemp : estatuto estatuto bloqueTemp empty
estatuto	estatuto : asignacion lectura escritura condicion llamar while comentario doWhile retorno empty
asignacion	asignacion : varParam p_n_asignacionTemp EQUAL expresion p_n_asignacion SCOLON
lectura	lectura : READ LPAR lecturaTemp p_n_lectura RPAR SCOLON
escritura	escritura : WRITE LPAR escrituraTemp RPAR SCOLON
escrituraTemp	escrituraTemp : varTemp p_n_escrituraExp expresion p_n_escrituraExp p_escrituraTemp2 p_escrituraTemp2 COMMA escrituraTemp

	varTemp p_n_escrituraExp COMMA escrituraTemp expresion p_n_escrituraExp COMMA escrituraTemp
escrituraTemp2	p_escrituraTemp2 : CSTR p_n_guardaString CSTR p_n_guardaString p_escrituraTemp2
condicion	condicion : IF LPAR expresion p_n_revisaIF RPAR bloque p_n_acabaIF IF LPAR expresion p_n_revisaIF RPAR bloque ELSE p_n_else bloque p_n_acabaIF
llamar	llamar : CALL LPAR ID p_n_nameID p_n_revisaFuncExiste p_n_generaEra LPAR llamarTemp RPAR p_n_goSub RPAR SCOLON CALL LPAR ID p_n_nameID p_n_guardaVarRet EQUAL ID p_n_nameID p_n_revisaFuncExiste p_n_generaEra LPAR llamarTemp RPAR p_n_goSub RPAR SCOLON
llamarTemp	llamarTemp : empty llamarTemp2
llamarTemp2	llamarTemp2 : expresion p_n_revisaParam expresion p_n_revisaParam COMMA p_n_nextParam llamarTemp2
retorno	retorno : RETURN SCOLON RETURN LPAR expresion p_n_asignaRet RPAR SCOLON
while	while : WHILE p_n_whileInicio LPAR expresion p_n_revisaIF RPAR bloque p_n_whileFinal
comentario	comentario : COMMENT
doWhile	doWhile : DO p_n_doWhileInicio bloque WHILE LPAR expresion p_n_doWhileFinal RPAR SCOLON
lecturaTemp	lecturaTemp : varParam lecturaTemp empty
expresion	expresion : exp
exp	exp : t p_n_checkPM exp PLUS p_n_addOper exp exp MINUS p_n_addOper exp
t	t : y p_n_checkMD

	<pre> t MULT p_n_addOper t t DIV p_n_addOper t </pre>
y	<pre> y : varTemp y GTHAN p_n_addOperBool y y LTHAN p_n_addOperBool y y GEQUALTHAN p_n_addOperBool y y LEQUALTHAN p_n_addOperBool y y EEQUAL p_n_addOperBool y y NEQUAL p_n_addOperBool y </pre>
varTemp	<pre> varTemp : varParam p_n_addPilaO CTEINT p_n_addCTEINT CTEF p_n_addCTEF CTEC p_n_addCTEC LPAR p_n_falsoParentesis exp RPAR p_n_sacarFalsoParentesis </pre>
empty	empty :

Generación de Código Intermedio y Análisis Semántico

Las direcciones virtuales con las cuales se realizó el programa para la generación de código intermedio se pueden ver a continuación:

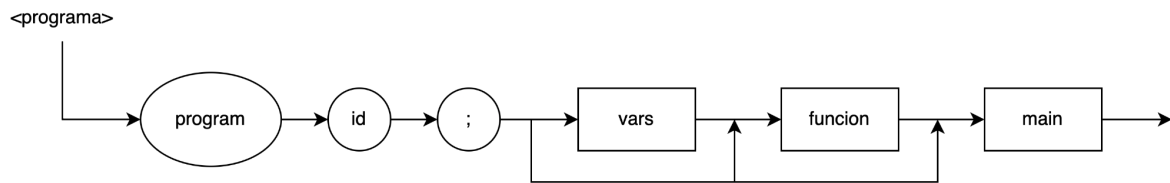
Nombre	Rango
constantInt	0 - 9999
constanFloat	1000 - 1999
constantChar	2000 - 2999
localInt	3000 - 3999
localFloat	4000 - 4999
localChar	5000 - 5999
localBool	6000 - 6999
globalInt	7000 - 7999
globalFloat	8000 - 8999
globalChar	9000 - 9999
globalBool	10000 - 10999
tempInt	11000 - 15999

tempFloat	16000 - 20999
tempChar	21000 - 25999
tempBool	26000 - 30999

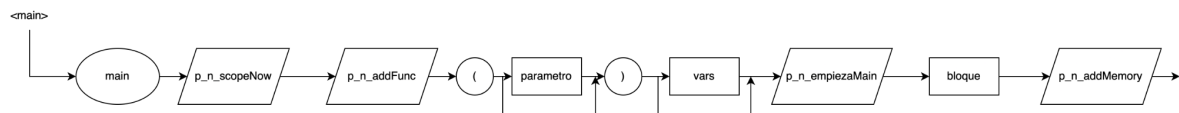
Para la generación de cuádruplos, se tienen diferentes comandos con los que la máquina virtual puede hacer las operaciones. Estos comandos se pueden ver a continuación:

Comando	¿Qué hace?
GOTO	Hace referencia al cuádruplo a la que la Máquina Virtual (MV) tiene que ir
GOTOF	Hace referencia al cuádruplo a la que la MV debe ir cuando una condición es falsa
GOTOV	Hace referencia al cuádruplo a la que la MV debe ir cuando una condición es verdadera
GOSUB	Hace referencia al cuádruplo en donde la MV llama a una función, guarda el valor de retorno, asigna memoria a la función y se le asignan los parámetros en caso de haber
RETURN	Hace referencia a cuando regresa un valor una función, la MV regresa al cuádruplo que se guardó en gosub
PARAMETER	Hace referencia al parámetro que se tiene que pasar a la función que se está llamando
WRITE	Hace referencia a cuando se le pide a la MV a desplegar un valor
READ	Hace referencia a cuando se le pide a la MV recibir un valor
ENDFUNC	Hace referencia al final de una función, se pasa al alcance anterior
ERA	Hace referencia a cuando se llama a una función

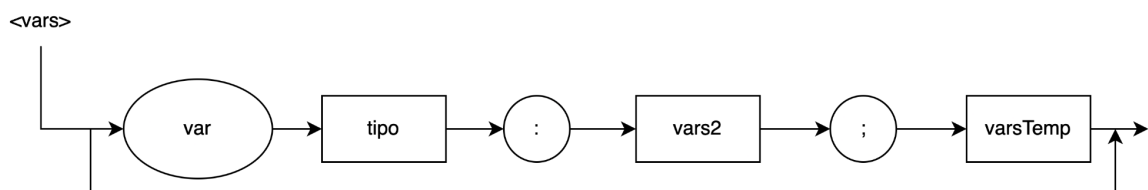
Diagramas de Sintaxis



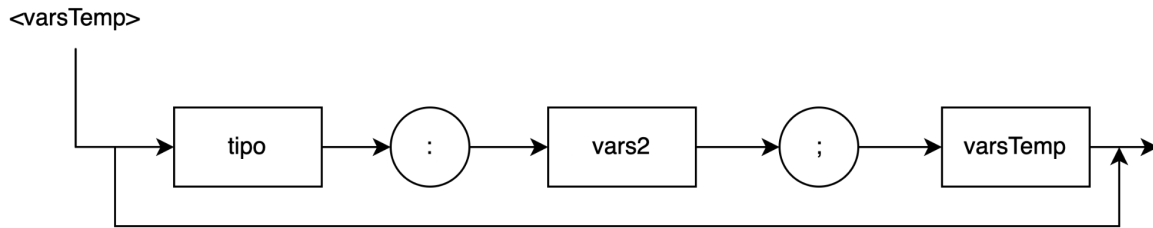
Punto Neurálgico	Función
En este diagrama no se tiene ningún punto neurálgico	



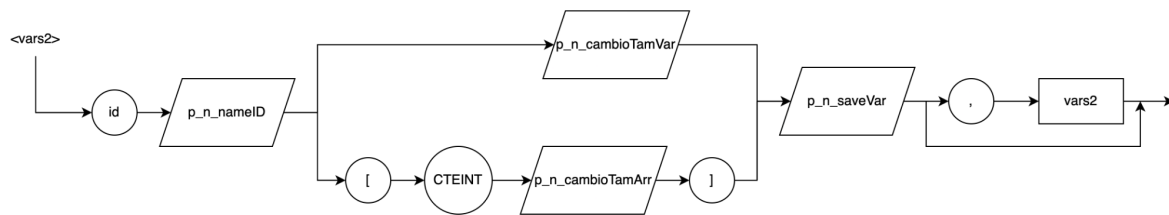
Punto Neurálgico	Función
p_n_scopeNow	Guarda una variable global el scope actual y reinicia los contadores de variables para las funciones
p_n_addFunc	Guarda la función en el directorio de funciones y si no es void, crea una variable con el mismo nombre de la función
p_n_empiezaMain	Guarda en el directorio de funciones en qué cuádruplo empieza el main
p_n_addMemory	Guarda la cantidad de variables que tiene la función por tipo en el directorio de funciones



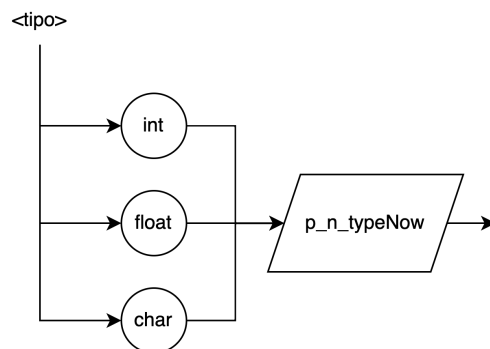
Punto Neurálgico	Función
En este diagrama no se tiene ningún punto neurálgico	



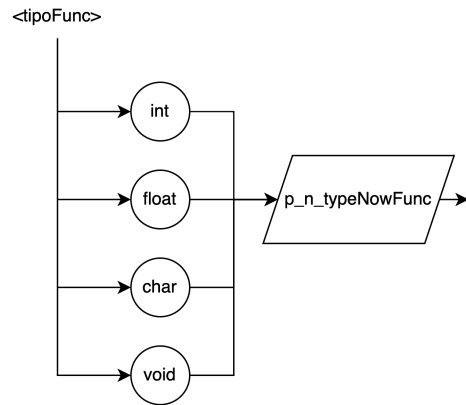
Punto Neurálgico	Función
En este diagrama no se tiene ningún punto neurálgico	



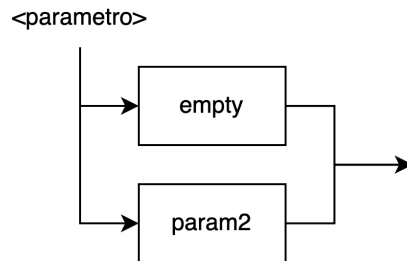
Punto Neurálgico	Función
p_n_nameID	Guarda el nombre del ID que leyó
p_n_cambioTamVar	Guarda el tamaño de la variable
p_n_cambioTamArr	Guarda el tamaño del arreglo
p_n_saveVar	Guarda la variable en la tabla de variables



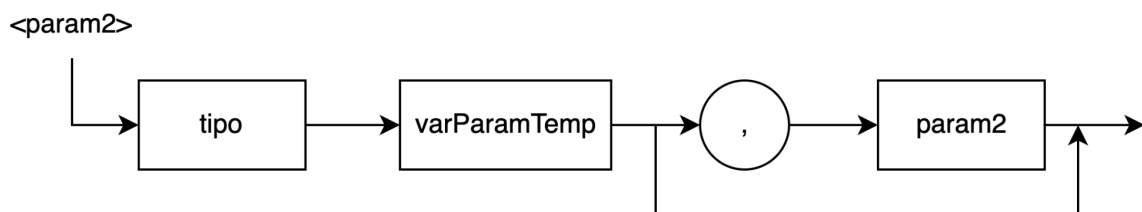
Punto Neurálgico	Función
p_n_typeNow	Guarda el tipo de variable que se leyó



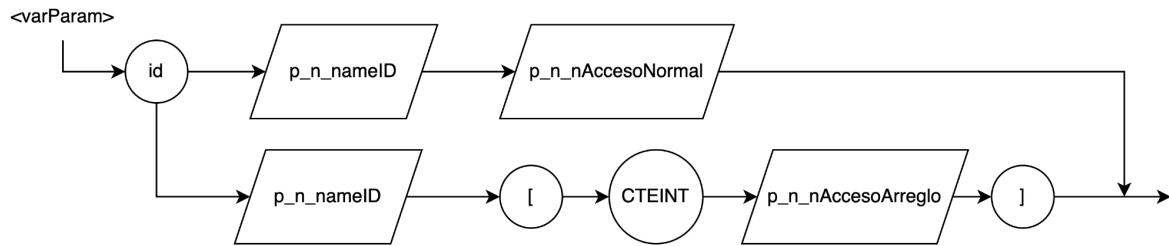
Punto Neurálgico	Función
p_n_typeNowFunc	Guarda el tipo de función que se leyó



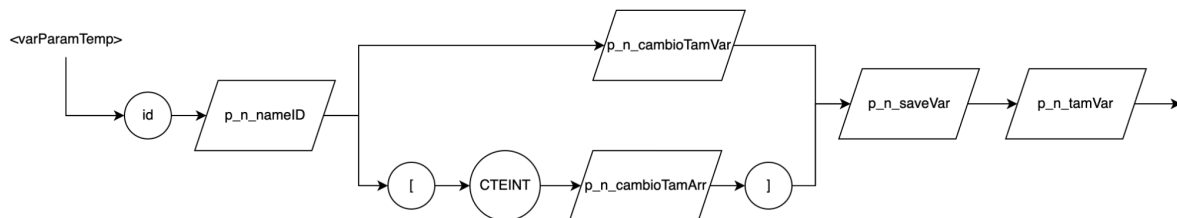
Punto Neurálgico	Función
En este diagrama no se tiene ningún punto neurálgico	



Punto Neurálgico	Función
En este diagrama no se tiene ningún punto neurálgico	



Punto Neurálgico	Función
p_n_nameID	Guarda el nombre del ID que leyó
p_n_nAccesoNormal	Guarda que el índice de una variable es 0
p_n_nAccesoArreglo	Guarda el índice del arreglo para accederlo

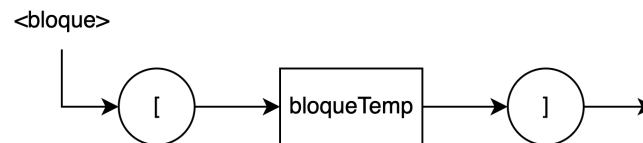


Punto Neurálgico	Función
p_n_nameID	Guarda el nombre del ID que leyó
p_n_cambioTamVar	Guarda el tamaño de la variable
p_n_cambioTamArr	Guarda el tamaño del arreglo
p_n_saveVar	Guarda la variable en la tabla de variables
p_n_tamVar	Suma uno al número de parámetros de la función

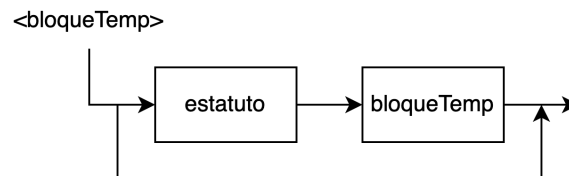


Punto Neurálgico	Función
p_n_scopeNow	Guarda una variable global el scope actual y reinicia los contadores de variables para las funciones
p_n_addFunc	Guarda la función en el directorio de funciones y si no es void, crea una variable con el mismo nombre de la función
p_n_iniciaFunc	Guarda en el directorio de funciones en qué cuádruplo empieza la función

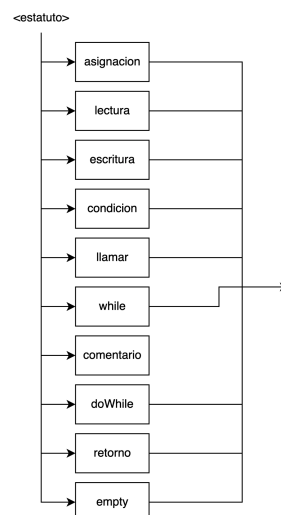
p_n_addMemory	Guarda la cantidad de variables que tiene la función por tipo en el directorio de funciones
p_n_endFunc	Genera el cuádruplo de endfunc y guarda en qué cuádruplo se acaba la función



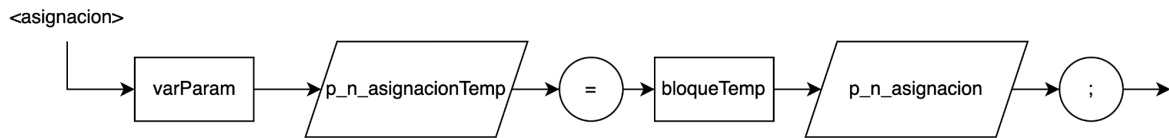
Punto Neurálgico	Función
En este diagrama no se tiene ningún punto neurálgico	



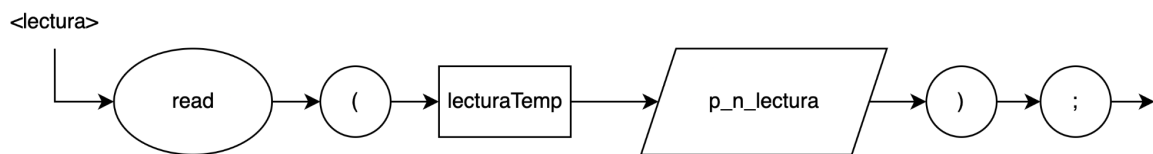
Punto Neurálgico	Función
En este diagrama no se tiene ningún punto neurálgico	



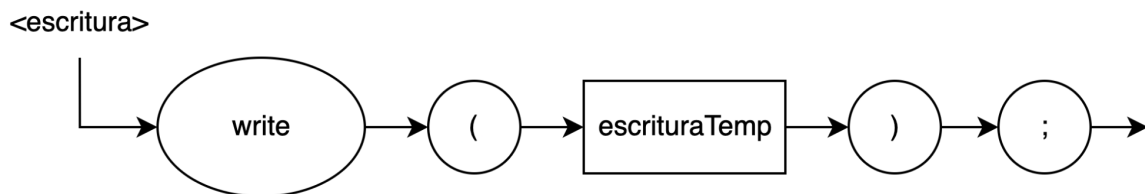
Punto Neurálgico	Función
En este diagrama no se tiene ningún punto neurálgico	



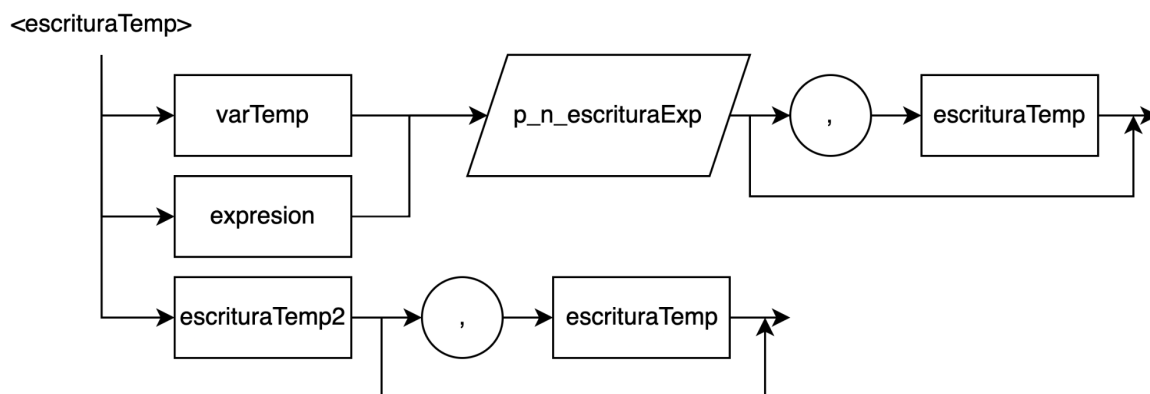
Punto Neurálgico	Función
p_n_asignacionTemp	Introduce la variable varParam a la pilaO y su tipo a la pTipos
p_n_asignacion	Genera el cuádruplo de la asignación y revisa si los tipos son compatibles



Punto Neurálgico	Función
p_n_lectura	Genera el cuádruplo para la lectura

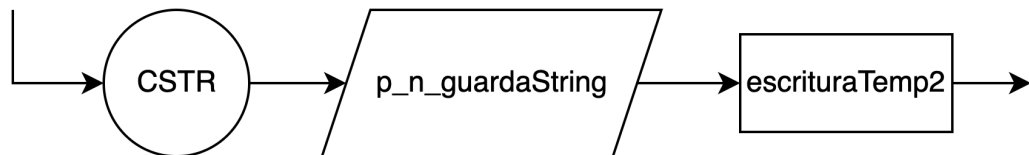


Punto Neurálgico	Función
En este diagrama no se tiene ningún punto neurálgico	



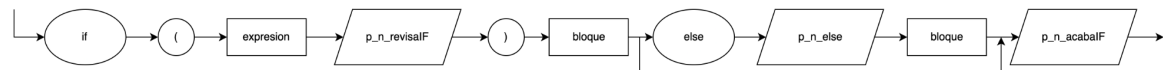
Punto Neurálgico	Función
p_n_escrituraExp	Genera el cuádruplo para la escritura

<escrituraTemp2>



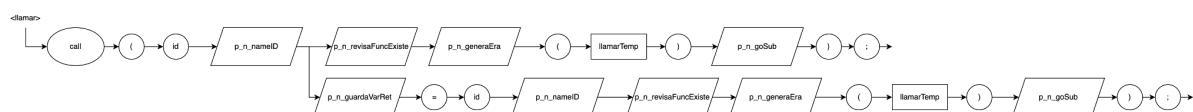
Punto Neurálgico	Función
p_n_guardaString	Genera el cuádruplo para la escritura de un string y lo guarda en una variable temporal

<<condicion>>



Punto Neurálgico	Función
p_n_revisaIF	Revisa que la expresión sea un booleano, genera el cuádruplo booleano y también genera el cuádruplo de gotof
p_n_else	Genera el cuádruplo goto y asigna el valor al gotof para saber a donde ir si la condición es falsa
p_n_acabaIF	Le asigna el valor de goto, para saltar el “else” en caso de que la condición sea verdadera

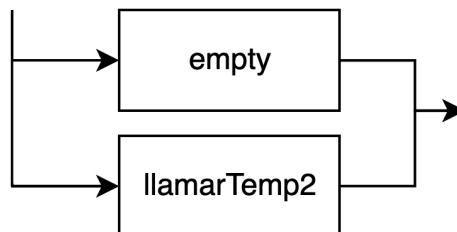
<<llamar>>



Punto Neurálgico	Función
p_n_nameID	Guarda el nombre del ID que leyó
p_n_revisaFuncExiste	Revisa si la función que se quiere llamar existe
p_n_guardaVarRet	Guarda la variable a la que se le va a retornar el valor de la función
p_n_generaEra	Genera el cuádruplo Era y se va sumando la cantidad de variables que se tienen en el programa

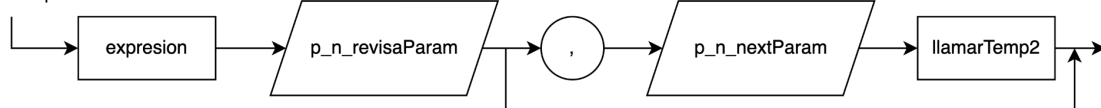
p_n_goSub	Genera el cuádruplo gosub con el valor a donde inicia la función a la cual se está llamando. Además genera el cuádruplo de asignación a la variable a la que se va a retornar un valor (si es que lo hay)
-----------	---

<llamarTemp>



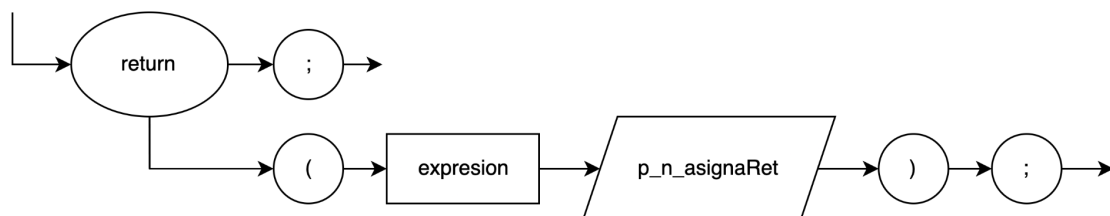
Punto Neurálgico	Función
En este diagrama no se tiene ningún punto neurálgico	

<llamarTemp2>

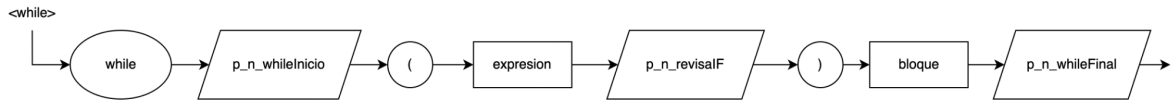


Punto Neurálgico	Función
p_n_revisaParam	Revisa que el tipo de parámetro que se está pasando a la función sea el mismo que tiene asignado
p_n_nextParam	Se toman los datos del siguiente parámetro

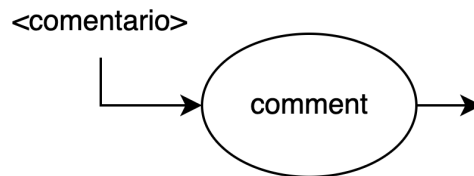
<retorno>



Punto Neurálgico	Función
p_n_revisaParam	Asigna a la variable de retorno lo que regresa la función que se llamó



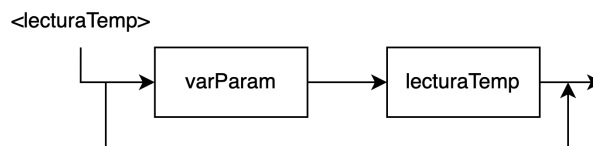
Punto Neurálgico	Función
p_n_whileInicio	Guarda el cuádruplo actual para regresar cuando se acabe el while
p_n_revisaIF	Revisa que la expresión sea un booleano, genera el cuádruplo booleano y también genera el cuádruplo de gotof
p_n_whileFinal	Genera el cuádruplo goto con el valor que se guardó en whileInicio



Punto Neurálgico	Función
En este diagrama no se tiene ningún punto neurálgico	



Punto Neurálgico	Función
p_n_doWhileInicio	Guarda el cuádruplo actual para regresar cuando se acabe el do-while
p_n_doWhileFinal	Genera el cuádruplo gotov con el valor que se guardó en doWhileInicio

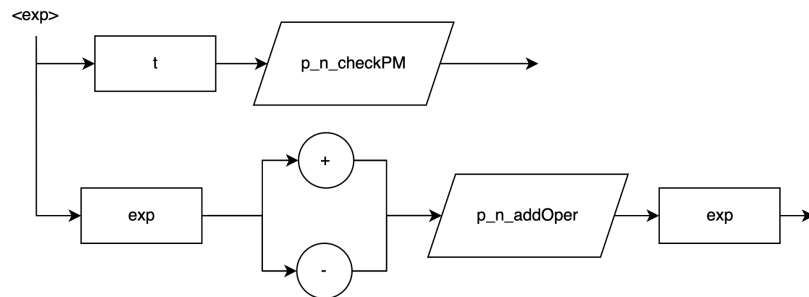


Punto Neurálgico	Función
En este diagrama no se tiene ningún punto neurálgico	

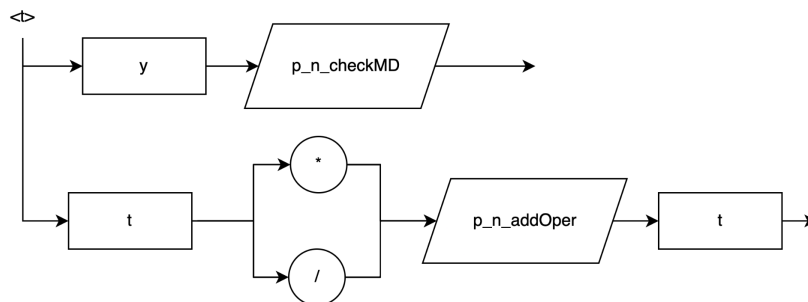
<expresion>



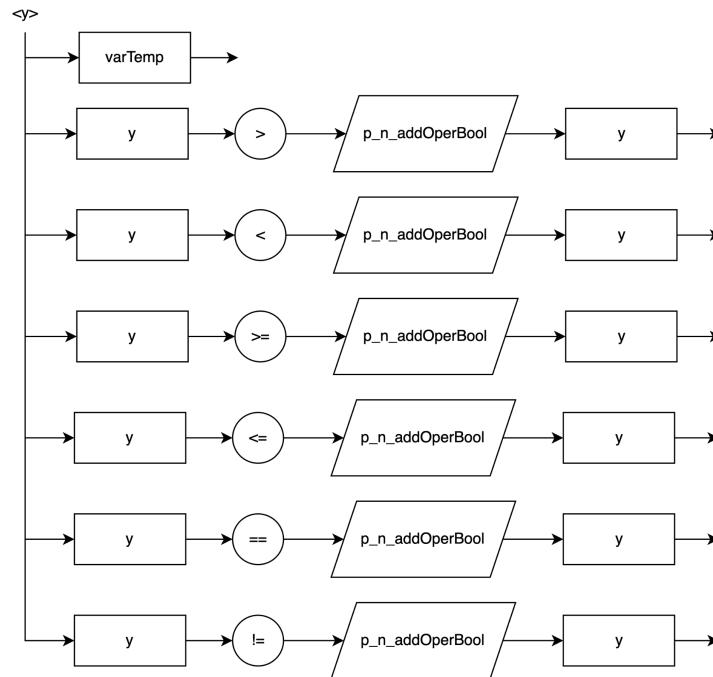
Punto Neurálgico	Función
En este diagrama no se tiene ningún punto neurálgico	



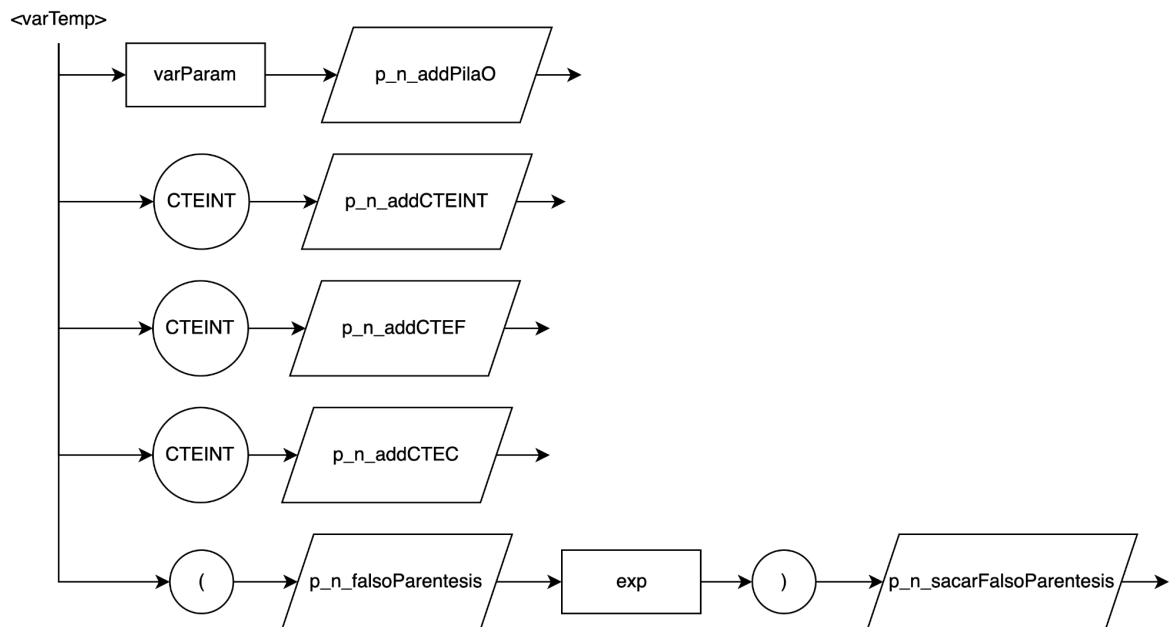
Punto Neurálgico	Función
p_n_checkPM	Revisa si hay una suma o resta en la pila. En caso de que haya se genera el cuádruplo con la operación
p_n_addOper	Se guarda el operador suma o resta en la pila de operadores



Punto Neurálgico	Función
p_n_checkMD	Revisa si hay una multiplicación o división en la pila. En caso de que haya se genera el cuádruplo con la operación
p_n_addOper	Se guarda el operador multiplicación o división en la pila de operadores



Punto Neurálgico	Función
p_n_addOperBool	Guarda el operador Bool en una pila booleana para cuando se necesite hacer la operación



Punto Neurálgico	Función
p_n_addPilaO	Añade la variable a la pila de operandos

p_n_addCTEINT	Añade el número int a la pila de operandos
p_n_addCTEF	Añade el número float a la pila de operandos
p_n_addCTEC	Añade el valor char a la pila de operandos
p_n_falsoParentesis	Añade un paréntesis a la pila de operadores para definir el orden de las operaciones
p_n_sacarFalsoParentesis	Elimina el paréntesis de la pila de operadores

Estructura del Lenguaje

La estructura general del lenguaje de programación LEAD se puede encontrar a continuación:

```
program nombre_del_programa;

<Variables Globales>

<Funciones a Desarrollar>

main() {

<Estatutos>

}
```

En este pequeño código de ejemplo, se ve cómo se programaría en LEAD, donde las palabras subrayadas son palabras reservadas y las secciones en *itálicas* son opcionales.

Para la declaración de variables:

```
var

    tipo: lista_variables;

    <tipo: lista_variables;>
```

En donde tipo puede ser int, float o char y la sección lista_variables son las variables separadas por medio de comas. Ej: int: var1, var2, var3;

Para la declaración de funciones:

```
function <tipo_retorno> nombre_func(<parámetros>)  
  
<Declaración de variables locales>  
  
{  
  
    <Estatutos>  
  
}
```

En donde tipo_retorno puede ser int, float, char o void y los parámetros que puede recibir la función son únicamente variables simples int, float o char. La cantidad de funciones que puede tener un código en LEAD empieza en cero, pero se deben definir antes del main.

Los estatutos del lenguaje de programación que LEAD soporta son los siguientes: asignación, lectura, escritura, condición, llamada, while, do-while y retorno. Su sintaxis y especificaciones se pueden ver a continuación:

Para la asignación:

```
variable = expresion;
```

Donde la variable puede ser un id o id[CTEINT] y recibe como valor el resultado de una expresión aritmética.

Para la lectura:

```
read(variable);
```

Donde la variable puede ser un id o id[CTEINT] y recibe como valor el resultado del input del usuario en la terminal. Únicamente se puede recibir un solo valor por línea.

Para la escritura:

```
write(variable, variable, CSTR, ...);
```

Donde la variable puede ser un id o id[CTEINT] y escribe el valor de la variable en la terminal. Se puede recibir múltiples valores por línea de código separando con comas las variables que se quieren mostrar. Asimismo, se pueden desplegar strings usando comillas. Ej: `write("hola");`

Para la condición:

```
if (<expresion>) {  
    <estatuto>  
}  
else {  
    <estatuto>  
}
```

En donde la expresión debe de ser una variable booleana para revisar a dónde va el código en ejecución y el else es opcional a discreción del usuario.

Para la llamada:

```
call(id = id(<parametro>);
```

En donde la asignación dentro de la llamada es opcional y los parámetros son las variables o números que ya están definidas en el código creado por el usuario.

Para el while:

```
while (<expresion>) {  
    <estatuto>  
}
```

En donde la expresión debe de ser una variable booleana para revisar a dónde va el código en ejecución.

Para el do-while:

```
do{  
    <estatuto>  
}while (<expresion>) ;
```

En donde la expresión debe de ser una variable booleana para revisar a dónde va el código en ejecución.

Para el retorno:

```
return (<expresion>) ;
```

En donde la expresión puede ser tanto una variable definida en el código o una constante int, float o char dependiendo del tipo de función.

Para las expresiones:

Las expresiones en el lenguaje de programación LEAD son tradicionales como en otros lenguajes de programación. Los operadores aritméticos son los siguientes: +, -, *, /, <, >, ==, !=. Las prioridades se manejan como cualquier otro lenguaje de programación, haciendo uso de paréntesis para definir prioridades.

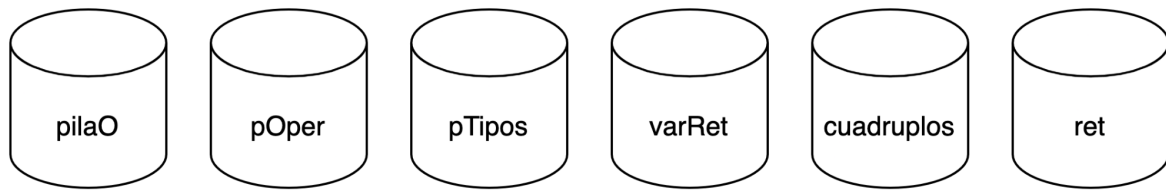
En el lenguaje de programación LEAD existen palabras reservadas, constantes int, char y float, además de constantes string las cuales se definieron en páginas anteriores de este documento.

Consideraciones Semánticas

lOperand	rOperand	+	-	*	/	>	<	==	!=	=	>=	<=
int	int	int	int	int	float	bool	bool	bool	bool	int	bool	bool
int	float	float	float	float	float	bool	bool	bool	bool	float	bool	bool
int	char	error	error	error	error	error	error	error	error	error	error	error
int	bool	error	error	error	error	error	error	error	error	error	error	error
float	float	float	float	float	float	bool	bool	bool	bool	float	bool	bool
float	int	float	float	float	float	bool	bool	bool	bool	float	bool	bool
float	char	error	error	error	error	error	error	error	error	error	error	error
float	bool	error	error	error	error	error	error	error	error	error	error	error
char	char	error	error	error	error	error	error	bool	bool	char	error	error
char	int	error	error	error	error	error	error	error	error	error	error	error
char	float	error	error	error	error	error	error	error	error	error	error	error
char	bool	error	error	error	error	error	error	error	error	error	error	error
bool	bool	error	error	error	error	error	error	bool	bool	bool	error	error
bool	int	error	error	error	error	error	error	error	error	error	error	error
bool	float	error	error	error	error	error	error	error	error	error	error	error
bool	char	error	error	error	error	error	error	error	error	error	error	error

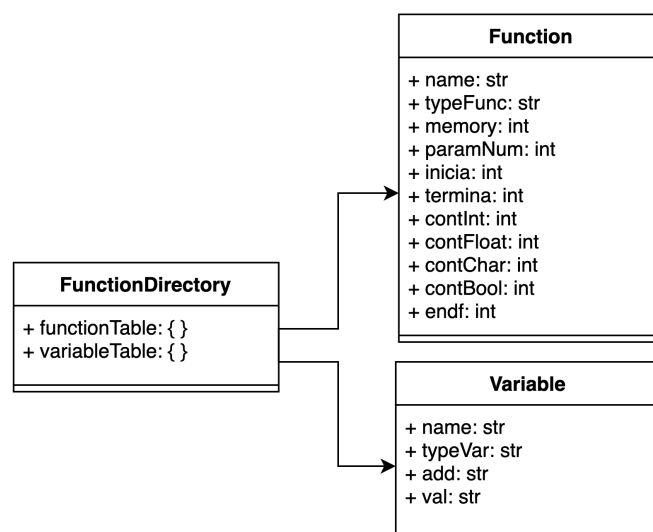
Administración de Memoria

La administración de memoria utilizada en el lenguaje de programación LEAD se compone de estructuras sencillas como pilas y listas. En cuanto a la dependencia de cada una de las pilas, se manejan de manera independiente. Estas pilas tienen como objetivo el guardar valores importantes para la creación de cuádruplos en compilación. Las pilas con sus nombres se pueden encontrar a continuación:



Nombre	Función
pilaO	Almacena los operandos para las operaciones
pOper	Almacena los operadores para las operaciones
pTipos	Almacena los tipos de los operandos para las operaciones
varRet	Almacena la variable de retorno cuando se hace una llamada, es una pila por si hay una recursión
cuádruplos	Almacena los cuádruplos que se generan en compilación
ret	Almacena el número de cuádruplo al cual retornar cuando se hace una llamada a una función

En cuanto a la información sobre las funciones y la información de las variables, se creó una estructura para almacenar información general únicamente. El almacenamiento de los valores de las variables se explicarán después de este diagrama. El diagrama de clases de funciones y variables se puede apreciar a continuación:



Para almacenar la información de las funciones y variables dentro del programa se creó un directorio de funciones. Este directorio de funciones contiene una tabla de funciones y una tabla de variables las cuales son diccionarios que contienen a los objetos Function y Variable respectivamente. Estas tablas son realmente diccionarios dentro de Python. Estos diccionarios nos ayudan a definir una llave (“key”) a la que podemos asignar uno o varios valores. En el caso de este proyecto las funciones se almacenan con una única llave, la cual es el nombre de la función y con la cual se pueden acceder a todos los datos de la clase Function. Por otro lado, para las variables se tienen dos llaves. La primera es el nombre de la función (el “scope”) en donde se encuentra la variable y la segunda llave es el nombre de la variable.

En la clase Function se almacena toda la información de las funciones: el nombre, el tipo de función, la memoria, el número de parámetros general, en qué cuádruplo inicia, en qué cuádruplo termina y el número de variables por tipo que contiene.

La clase Variable almacena la información de las variables: el nombre, el tipo de la variable y la ubicación. En el diagrama se puede observar que también se guarda el valor, ese era el objetivo en un principio pero se decidió cambiar la forma en que se almacenaban los valores usando arreglos y las direcciones de memoria.

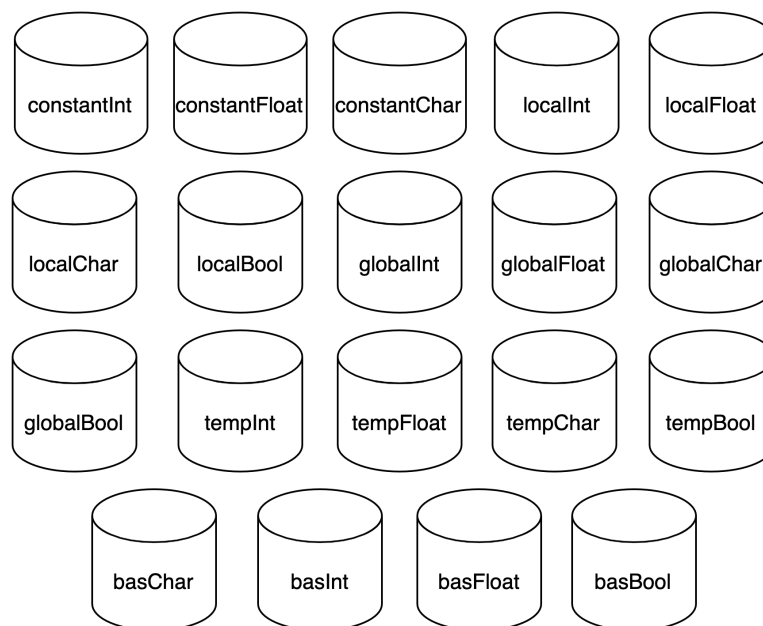
Descripción de la Máquina Virtual

Equipo de Cómputo, Lenguaje y Utilerías

La Máquina Virtual se creó utilizando Python. En cuanto a librerías, no se usó ninguna extra, los cuádruplos y las pilas se crearon usando listas y el equipo de cómputo utilizado para el proyecto fue MacOS con Python 3.9.7, utilizando VSCode como IDE.

Administración de la Memoria

En cuanto a la administración de la memoria y para guardar los valores de las variables en ejecución, como se mencionó anteriormente, se utilizan arreglos. Estos arreglos están separados por tipos y su tamaño es el mismo al rango que tiene cada tipo y alcance de variable (véase la sección [Generación de Código Intermedio y Análisis Semántico](#) en la tabla de variables y rangos). Los arreglos se pudieran visualizar de la siguiente manera:



Nombre	Función
basChar	Almacena la dirección base de las variables char para cuando hay una llamada a una función y poder acceder a los valores correctos dentro de los arreglos

basFloat	Almacena la dirección base de las variables float para cuando hay una llamada a una función y poder acceder a los valores correctos dentro de los arreglos
basInt	Almacena la dirección base de las variables int para cuando hay una llamada a una función y poder acceder a los valores correctos dentro de los arreglos
basBool	Almacena la dirección base de las variables bool para cuando hay una llamada a una función y poder acceder a los valores correctos dentro de los arreglos

Cada vez que se crea una variable, se añade un valor a su arreglo de facto dependiendo de su tipo y si alcance (int: 0, float: 0.0, char: ‘’, bool: False). Estos valores se van modificando cuando existan expresiones que modifiquen la variable. Para llegar a la dirección real de la variable (en caso de que haya llamadas recursivas) se utilizan las pilas de basInt, basFloat, basChar, basBool, las cuales almacenan la dirección real en donde se empiezan a almacenar las variables de la función dentro del arreglo (esto último solo aplica para las variables locales, las constantes, temporales y globales no necesitan un cálculo para revisar su dirección real).

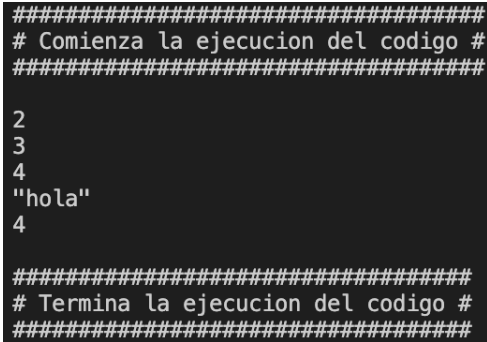
Con esta estructura para guardar los valores de las variables, se puede revisar la cantidad de variables existentes y verificar que el usuario no sobrepase la cantidad de variables permitidas por tipo teniendo así un buen manejo de la memoria del compilador.

Pruebas de Funcionamiento del Lenguaje

Las pruebas de funcionamiento del lenguaje se encuentran a continuación, estas son las mismas pruebas que se definieron en la primera sección del documento (véase

[Requerimientos y Casos de Uso](#))

Test - Vectores
Revisar la asignación de los vectores en el Lenguaje
<pre>program LEAD; var int: t1; main() var int: prob1, vec[5]; { vec[0] = 10; vec[4] = vec[0] + 15; write(vec[4]); }</pre>
Código Intermedio
<pre>[[['goto', '-', '-', 1], ['=', 0, ", 7001"], ['+', 7002, 1, 11000], ['=', 11000, ", 7006"], ['write', ", ", 7006], ['EndFile', ", ", "]]</pre>
Output
<pre>##### # Comienza la ejecucion del codigo # ##### 25 ##### # Termina la ejecucion del codigo # #####</pre>

Test - Recursión y Parámetros Recursivos	
Revisar la funcionalidad de la recursión y el paso de parámetros	
<pre> program LEAD; var int: t1; function int sum(int x){ x = x + 1; write(x); if(x != 4){ call(x = sum(x)); } return(x); } main() var int: main1; { main1 = 1; call(main1 = sum(main1)); write("hola", main1); } </pre>	
Código Intermedio	
<pre> [['goto', '-', '-', 14], ['+', 3001, 0, 11000], ['=', 11000, "", 3001], ['write', "", "", 3001], ['!=', 3001, 1, 26000], ['gotof', 26000, "", 10], ['ERA', "", "", 'sum'], ['parameter', 3001, "", 3001], ['gosub', 'sum', "", 1], ['=', 3000, "", 3001], ['=', 3001, "", 3001], ['=', 3001, "", 3000], ['return', "", "", 'sum'], ['endfunc', "", "", ""], ['=', 2, "", 7001], ['ERA', "", "", 'sum'], ['parameter', 7001, "", 3001], ['gosub', 'sum', "", 1], ['=', 3000, "", 7001], ['write', "", "", 2000], ['write', "", "", 7001], ['EndFile', "", "", "]] </pre>	
Output	
 <pre> ##### # Comienza la ejecucion del codigo # ##### 2 3 4 "hola" 4 ##### # Termina la ejecucion del codigo # ##### </pre>	

Test - Factorial Cíclico

Revisar la funcionalidad de las funciones y ciclos en un caso factorial cíclico

```
program LEAD;
var
    int: t1;

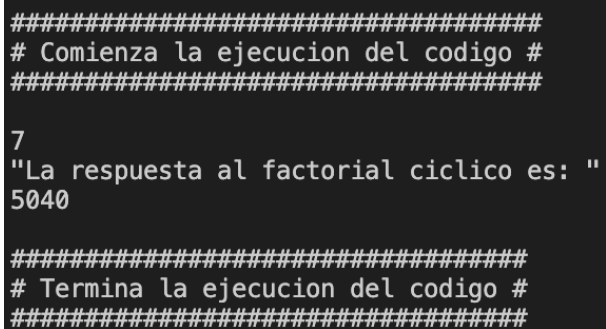
function void fact(int num)
var
    int: fact;
{
    write("La respuesta al factorial ciclico es: ");
    if(num < 0){
        write("No existe el factorial negativo");
    }

    if(num == 0){
        write(1);
    }else{
        fact = 1;
        while(num > 1){
            fact = fact * num;
            num = num - 1;
        }

        write(fact);
    }
}

main()
var
    int: main1;

{
    read(main1);
    call(fact(main1));
}
```

Código Intermedio
<pre> [['goto', '-', '-', 19], ['write', "", "", 2000], ['<', 3000, 0, 26000], ['gotof', 26000, "", 5], ['write', "", ", 2001], ['==', 3000, 1, 26001], ['gotof', 26001, "", 9], ['write', "", "", 2], ['goto', "", "", 18], ['=', 3, ", 3001], ['>', 3000, 4, 26002], ['gotof', 26002, "", 17], ['*', 3001, 3000, 11000], ['=', 11000, "", 3001], ['-', 3000, 5, 11001], ['=', 11001, "", 3000], ['goto', "", "", 10], ['write', "", "", 3001], ['endfunc', "", "", ""], ['Read', "", "", 7001], ['ERA', "", "", 'fact'], ['parameter', 7001, "", 3000], ['gosub', 'fact', "", 1], ['EndFile', "", "", "]] </pre>
Output
 <pre> ##### # Comienza la ejecucion del codigo # ##### 7 "La respuesta al factorial ciclico es: " 5040 ##### # Termina la ejecucion del codigo # ##### </pre>

Test - Factorial Recursivo
Revisar la funcionalidad de las funciones y ciclos en un caso factorial recursivo
<pre> program LEAD; var int: t1; function int fact(int num) var int: ret, n, temp; { n = num; if(num == 1){ return(1); }else{ call(temp = fact(num - 1)); num = n * temp; return(num); } } main() var int: res, fin; { read(res); </pre>


```

if(res < 0){
    write("No hay factorial de numeros menores a 0");
}

if(res == 0){
    write("Factorial Recursivo");
    write(1);
}else{
    write("Factorial Recursivo");
    call(fin = fact(res));
    write(fin);
}
}

```

Código Intermedio

```

[['goto', '-', '-', 19], ['=', 3001, "", 3003], ['==', 3001, 0, 26000], ['gotof', 26000, "", 8], ['=', 1, "", 3000], ['=', 1, "", 3000], ['return', "", "", 'fact'], ['goto', "", "", 18], ['ERA', "", "", 'fact'], ['-', 3001, 2, 11001], ['parameter', 11001, "", 3001], ['gosub', 'fact', "", 1], ['=', 3000, "", 3004], ['*', 3003, 3004, 11002], ['=', 11002, "", 3001], ['=', 3001, "", 3004], ['=', 3001, "", 3000], ['return', "", "", 'fact'], ['endfunc', "", "", ""], ['Read', "", "", 7001], ['<', 7001, 3, 26001], ['gotof', 26001, "", 23], ['write', "", "", 2000], ['==', 7001, 4, 26002], ['gotof', 26002, "", 28], ['write', "", "", 2001], ['write', "", "", 5], ['goto', "", "", 34], ['write', "", "", 2002], ['ERA', "", "", 'fact'], ['parameter', 7001, "", 3001], ['gosub', 'fact', "", 1], ['=', 3000, "", 7002], ['write', "", "", 7002], ['EndFile', "", "", "]]

```

Output

```

#####
# Comienza la ejecucion del codigo #
#####

7
"Factorial Recursivo"
5040

#####
# Termina la ejecucion del codigo #
#####

```

Test - Fibonacci Cíclico

Revisar la funcionalidad de las funciones y ciclos en un caso fibonacci cíclico

```
program LEAD;
var
    int: t1;

function void fact(int num)
var
    int: n1, n2, nth, count;
{
    n1 = 0;
    n2 = 1;
    count = 0;
    if(num <= 0){
        write("Necesita ser un numero positivo");
    }

    if(num == 1){
        write(n1);
    }else{
        write("Secuencia Fibonacci");
        while(count < num){
            write(n1);
            nth = n1 + n2;
            n1 = n2;
            n2 = nth;
            count = count + 1;
        }
    }
}

main()
var
    int: main1;
{
    read(main1);
    write("-----");
    call(fact(main1));
}
```

Código Intermedio
<pre>[[['goto', '-', '-', 23], ['=', 0, "", 3001], ['=', 1, "", 3002], ['=', 2, "", 3004], ['<=', 3000, 3, 26000], ['gotof', 26000, "", 7], ['write', "", "", 2000], ['==', 3000, 4, 26001], ['gotof', 26001, "", 11], ['write', "", "", 3001], ['goto', "", "", 22], ['write', "", "", 2001], ['<', 3004, 3000, 26002], ['gotof', 26002, "", 22], ['write', "", "", 3001], ['+', 3001, 3002, 11000], ['=', 11000, "", 3003], ['=', 3002, "", 3001], ['=', 3003, "", 3002], ['+', 3004, 5, 11001], ['=', 11001, "", 3004], ['goto', "", "", 12], ['endfunc', "", "", ""], ['Read', "", "", 7001], ['write', "", "", 2002], ['ERA', "", "", 'fact'], ['parameter', 7001, "", 3000], ['gosub', 'fact', "", 1], ['EndFile', "", "", "]]</pre>
Output
<pre>##### # Comienza la ejecucion del codigo # ##### 8 "_____" "Secuencia Fibonacci" 0 1 1 2 3 5 8 13 ##### # Termina la ejecucion del codigo # #####</pre>

Test - Fibonacci Recursivo
Revisar la funcionalidad de las funciones y ciclos en un caso fibonacci recursivo
<pre>program LEAD; var int: t1; function int fib(int n) var int: fib1, fib2; { if(n <= 1){ return(n); }else{ call(fib1 = fib(n - 1)); call(fib2 = fib(n - 2)); return(fib1 + fib2); } }</pre>

```

}

main()
var
    int: nTerms, cont, res;
{
    cont = 0;
    read(nTerms);
    if(nTerms <= 0){
        write("Es necesario un numero positivo");
    }else{
        write("Fibonacci recursivo");
        while(nTerms != cont){
            call(res = fib(cont));
            write(res);
            cont = cont + 1;
        }
    }
}

```

Código Intermedio

```

[['goto', '-', '-', 22], ['<=', 3001, 0, 26000], ['gotof', 26000, "", 7], ['=', 3001, "", 3000], ['=', 3001, "", 3000], ['return', "", "", 'fib'], ['goto', "", "", 21], ['ERA', "", "", 'fib'], ['-', 3001, 1, 11001], ['parameter', 11001, "", 3001], ['gosub', 'fib', "", 1], ['=', 3000, "", 3002], ['ERA', "", "", 'fib'], ['-', 3001, 2, 11002], ['parameter', 11002, "", 3001], ['gosub', 'fib', "", 1], ['=', 3000, "", 3003], ['+', 3002, 3003, 11003], ['=', 11003, "", 3003], ['=', 11003, "", 3000], ['return', "", "", 'fib'], ['endfunc', "", "", ""], ['=', 3, "", 7002], ['Read', "", "", 7001], ['<=', 7001, 4, 26001], ['gotof', 26001, "", 28], ['write', "", "", 2000], ['goto', "", "", 39], ['write', "", "", 2001], ['!=', 7001, 7002, 26002], ['gotof', 26002, "", 39], ['ERA', "", "", 'fib'], ['parameter', 7002, "", 3001], ['gosub', 'fib', "", 1], ['=', 3000, "", 7003], ['write', "", "", 7003], ['+', 7002, 5, 11005], ['=', 11005, "", 7002], ['goto', "", "", 29], ['EndFile', "", "", "]]

```

Output

```

#####
# Comienza la ejecucion del codigo #
#####

8
"Fibonacci recursivo"
0
1
1
2
3
5
8
13

#####
# Termina la ejecucion del codigo #
#####

```

Test - Recursión Infinita
Revisar el buen manejo de memoria cuando no hay suficiente
<pre> program LEAD; var int: t1; function int sum(int x){ x = x + 1; if(x != 0){ call(x = sum(x)); } } main() var int: main1; { main1 = 1; call(main1 = sum(main1)); write("hola", main1); } </pre>
Código Intermedio
<pre> [['goto', '-', '-', 10], ['+', 3001, 0, 11000], ['=', 11000, ", 3001], ['!=', 3001, 1, 26000], ['gotof', 26000, ", 9], ['ERA', ", ", 'sum'], ['parameter', 3001, ", 3001], ['gosub', 'sum', ", 1], ['=', 3000, ", 3001], ['endfunc', ", ", "], ['=', 2, ", 7001], ['ERA', ", ", 'sum'], ['parameter', 7001, ", 3001], ['gosub', 'sum', ", 1], ['=', 3000, ", 7001], ['write', ", ", 2000], ['write', ", ", 7001], ['EndFile', ", ", "]] </pre>
Output
<pre> ##### # Comienza la ejecucion del codigo # ##### Error de sintaxis en el input - No hay suficiente memoria </pre>

Referencias

Beazley, David. (2020). PLY (Python Lex-Yacc). Recuperado el 17 de Octubre del 2022 de:
<https://www.dabeaz.com/ply/>