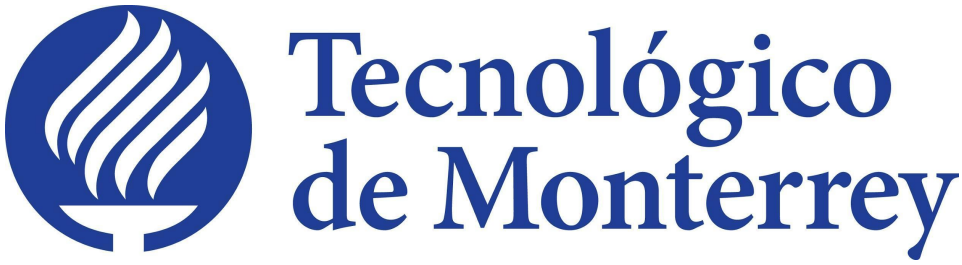
Instituto Tecnológico y de Estudios Superiores de Monterrey



Diseño de Compiladores

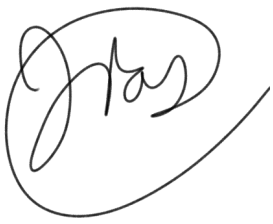
Elda G. Quiroga, M.C.

Dr. Héctor Ceballos, PhD

Compilador PePa–

Patricio Salazar Ríos **A01193833**

José Roberto Adame Sosa **A01176646**



06/6/2022

Índice

1 Descripción del Proyecto

1.1 Propósito y Alcance del Proyecto:

1.1.1 Propósito del Proyecto:

El propósito de PePa– es principalmente producir un lenguaje fuertemente tipado el cual pueda hacer las operaciones básicas de cualquier lenguaje. También queremos que la estructura estricta de nuestro programa ayude a quienes quieran utilizar este lenguaje a entenderlo y aprenderlo fácilmente.

1.1.2 Alcance del Proyecto:

Queremos que, como mínimo, nuestro lenguaje pueda hacer todo lo básico que haría cualquier otro lenguaje como hacer operaciones aritméticas, utilizar estatutos de decisión, condicionales y no-condicionales.

1.2 Análisis de Requerimientos y Descripción de los Principales Casos de Uso:

1.2.1 Análisis de Requerimientos:

1. RF01 El lenguaje debe leer el código fuente de un archivo .txt
2. RF02 El lenguaje debe tener los tipos primitivos básicos (enteros, flotantes, booleanos y strings)
3. RF03 El lenguaje debe poder manejar tipos estructurados de una o dos dimensiones
4. RF04 El lenguaje debe poder declarar y asignar variables de los tipos primitivos
5. RF05 El lenguaje debe poder manejar estatutos de decisión
6. RF06 El lenguaje debe poder manejar estatutos condicionales
7. RF07 El lenguaje debe poder manejar estatutos no-condicionales
8. RF08 El lenguaje debe poder declarar en contexto global y llamar funciones, aunque sea recursivamente, en contexto local
9. RF09 El lenguaje debe poder manejar tantos contextos como haya funciones, además de global y principal
10. RF10 El lenguaje debe poder manejar funciones con tipo de retorno void
11. RF11 El lenguaje debe regresar errores informativos al usuario

1.2.2 Descripción de los Principales Casos de Uso:

1. **Funcionamiento Normal:** Se prueba el funcionamiento normal del programa, es decir se lee un archivo .txt el cual contiene estatutos los cuales: declaran variables de todos los tipos primitivos, declaran estatutos de decisión, estatutos condicionales y no-condicionales, declaran funciones en contexto global y local, funciones con tipo de retorno void.

2. **Funcionamiento con Error:** Se prueban los límites del programa al leer un archivo .txt el cual contiene estatutos los cuales contienen errores que deberían parar ejecución del programa y arrojar un mensaje de error apropiado.

3. **Funcionamiento de Funciones Recursivas:** Se prueban los límites del programa al leer un archivo .txt el cual contiene estatutos los cuales contienen una función recursiva la cual puede llegar a hacer miles de llamadas.

1.3 Descripción del Proceso:

En general, se trabajó mayormente juntos haciendo "pair-programming" procurando estar en llamada en Discord todo el tiempo que se trabajaba para poder desarrollar las funciones más rápidamente y poder encontrar errores que a lo mejor uno no había visto. La mayoría del tiempo uno programaba y compartía pantalla mientras el otro comentaba y ayudaba.

1.3.1 Bítacora General

Semana	Descripción de Avance
1	● Diagramas de Análisis Léxico ● Diagramas de Análisis Sintáctico
Semana Santa	
2	
3	
4	
5	● Semántica básica de expresiones
6	● Se agregan Funciones ● Se agregan variables en base a su Scope
7	● Se genera Cubo Semántico Base
8	● Se genera Cubo Semántico Completo ● Generación de Código de Expresiones Aritméticas ● Generación de Código de Estatutos Secuenciales ● Generación de Código de Estatutos Condicionales ● Generación de Código de Funciones ● Mapa de Memoria de Ejecución para la Máquina Virtual ● Generación de Código de Arreglos/Tipos Estructurados (Declaración) ● Máquina Virtual: Ejecución de Expresiones Aritméticas ● Máquina Virtual: Ejecución de Estatutos Secuenciales

	● Máquina Virtual: Ejecución de Estatutos Condicionales ● Máquina Virtual: Declaración y Llamada de Funciones ● Documentación

1.3.2 Lista de Commits

Autór	Título	Descripción	Fecha
José Adame	Initial Commit	-	16/05/2022
José Adame	lol funny	haha	16/05/2022
Patricio Salazar	agregarFunc	Se programó la funcionalidad de agregarFunc	17/05/2022
José Adame	ya se agregan las variables basándose en su scope	-	17/05/2022
José Adame	added the basic structure of the semantic cube	-	26/05/2022
José Adame	finished semantic cube?	-	31/05/2022
José Adame	added a better print for the vartable	-	31/05/2022
José Adame	yolo	push momentario	31/05/2022
José Adame	ya se guardan los cuádruplos de +-*/ y los de asignación	-	31/05/2022
José Adame	escritura+lectura	-	1/06/2022
Patricio Salazar	ya se guardan cuádruplos ifs	-	1/06/2022
Patricio Salazar	ya se guardan cuádruplos if/else	-	1/06/2022
José Adame	some changes	-	1/06/2022

José Adame	Merge branch 'main' of https://github.com/pepeadame1/compilador	-	1/06/2022
José Adame	fixed index of fill()	-	1/06/2022
José Adame	jumps in while quadruples	-	1/06/2022
José Adame	for loops	-	1/06/2022
José Adame	better print function for fundir	-	1/06/2022
José Adame	modules falta void call	-	1/06/2022
Patricio Salazar	Llamada a función tentativa	Error: funciones en main.py son unreachable	2/06/2022
José Adame	finished void method call	-	2/06/2022
José Adame	Merge pull request #1 from pepeadame1/pato2	-	2/06/2022
José Adame	agregar tabla de constantes	-	2/06/2022
José Adame	started virtual memory	-	2/06/2022
José Adame	finished converting quadruples into virtual memory numbers	-	3/06/2022
Patricio Salazar	Declaración de arrays + manejo de memoria	-	3/06/2022
José Adame	parte de la máquina virtual ya funcional	-	4/06/2022
José Adame	for loops now work as intended	-	5/06/2022
José Adame	medio arreglado las llamadas a funciones (no compila)	-	5/06/2022

José Adame	funciona?? :O	-	5/06/2022
José Adame	fixed jump index	-	5/06/2022
José Adame	funciona recursividad (corre)	-	6/06/2022
José Adame	limpia de prints + llamada de archivo desde terminal	-	6/06/2022
José Adame	algunas pruebas	-	6/06/2022
José Adame	recursion funcional!!!!	-	6/06/2022
José Adame	mejores comentarios	-	6/06/2022
José Adame	minor fix	-	6/06/2022

https://github.com/pepeadame1/compilador/commits/main

1.3.3 Párrafos de Reflexión:

1.3.3.1 Patricio:
Estamos acabando el documento en la noche del día de la entrega, con funcionalidades que queríamos implementar todavía no completadas, lo que lleva a mi mayor aprendizaje: Un compilador es mucho más complejo de lo que me pude haber imaginado. Al pasar días seguidos trabajando por horas y horas intentando sacar cada funcionalidad, me di cuenta que no nos daban 8 semanas para hacerlo por nada. En verdad los lenguajes modernos nos dejan demasiadas cosas ya regaladas que no tomaba en cuenta, como poder programar con una sintaxis bastante laxa a comparación a la que desarrollamos. Nosotros no programamos objetos, y me da miedo pensar en hacerlo. Fuera de broma, el proyecto en verdad me ayudó a aterrizar muchos de los conocimientos vistos en clase, como el manejo de los cuádruplos, el manejo de memoria y los cálculos de los tamaños de memoria de arreglos y matrices, aunque no las terminamos usando. Además mi compañero Pepe me ayudó bastante a entender varios conceptos, pues muchas la mayoría del tiempo andábamos haciendo “peer programming” y nos podíamos hacer preguntas al medio del desarrollo de las diversas funciones.

1.3.3.2 José:

Claramente nos faltó tiempo, empezamos muy tarde y sufrimos las consecuencias, por una parte claramente estoy triste porque no pudimos terminar toda la funcionalidad que queriamos, aun así estoy orgulloso con lo que logramos, aprendimos muy rápido y el saber ya básicamente todos los temas de la materia antes de programar ciertas partes del compilador nos ayudó mucho a entender las partes más difíciles y a prevenir ciertas partes para luego poder cambiarlas fácilmente. De haber tenido más tiempo hubiéramos trabajado primero en un mejor manejo de memoria en la ejecución del programa ya que solo se logró la memoria para las variables locales que sean dinámicas y aun así no confiamos del todo en ellas. Elda nos dijo lo que iba a pasar y sucedió, aún así terminó satisfecha con nuestro esfuerzo. Un tema que no había entendido hasta hacer el compilador es como funciona bien la memoria al correr, yo pensaba que llamábamos a mem[120000] y después de pegar nuestras cabezas con una pared entendimos el uso de los offsets para sacar el valor real.

2 Descripción del Lenguaje

2.1 Nombre del Lenguaje:

El lenguaje se llama PePa--, inspirado en los apodos de los autores de este lenguaje (Pepe y Pato). Se agregó el '--' pues C++ es el lenguaje con que se empezó la carrera y siempre estará en nuestros corazones.

2.2 Descripción Genérica de las Principales Características del Lenguaje:

PePa-- es un lenguaje de programación altamente tipado el cual soporta datos primitivos como: enteros, flotantes, strings y booleanos. Permite la entrada y salida de datos por el usuario. Se pueden usar estatutos de decisión, estatutos condicionales y estatutos no-condicionales. Además se pueden declarar funciones y llamar a estas recursivamente.

2.3 Listado de Errores:

Razón de Error	**Mensaje de error**
Cuando el parser lee un condicional pero la expresión no resulta en booleano	*error: La expresion no es bool*
Cuando el parser lee un no-condicional pero la expresión no resulta en int o float	*error: tipos de datos no iguales*
Cuando el parser lee un no-condicional pero los tipos de variables no son iguales	*error: los tipos de variable no son iguales*
Cuando se intenta inicializar una variable pero esta ya existe.	*error: el id ya esta en uso*
Cuando se intenta agregar un límite pero este es igual o menor a zero.	*error: limite debe ser mayor a 0*
Cuando se intenta inicializar una función pero esta ya existe.	*error: la funcion ya fue declarada previamente*
Cuando se trata de guardar un temporal que no es int, float o booleano	*error: se trato de guardar un temp que no es int,float o bool*
Cuando se intenta regresar una dirección que no está en la lista de funciones.	*error: no se encontro la direccion*
Cuando se intenta llamar una función que no ha sido declarada	*error: la funcion no ha sido declarada*
Cuando el tipo del parámetro en una función está mal	*error: el tipo de parametros esta mal*
Cuando se intenta hacer una operación con tipos de parámetros no compatibles	*error: tipo de datos no compatibles*

3 Descripción del Compilador

3.1 Equipo de Cómputo: Para el desarrollo del compilador se utilizaron equipos Windows 11 y Linux (4.4.0/22000/Microsoft)

3.2 Lenguaje: El lenguaje utilizado fue Python

3.3 Utilerías Especiales: Se utilizó Sly como utilería especial

3.4 Descripción del Análisis de Léxico:

3.4.1 Patrones de Construcción:

Patrón de Construcción	Expresion Regular
newline	r’/n’
espacio	r‘ ‘
CTESTRING	r’\”([^””]+)\”’
CTEFLOAT	r’\d+\.\d+’
CTEINT	r’\d+’
CTECHAR	r’\’.*\’’
ID	r’[a-zA-Z_][a-zA-Z0-9_]*’

3.4.2 Enumeración de los “tokens”:

Número	Token	Descripción
1	SI	Palabra reservada que representa la palabra antes de la condición en un estatuto de decisión
2	ENTONCES	Palabra reservada que representa la palabra antes de la expresión cuando la condición resulta en Verdadero
3	SINO	Palabra reservada que representa la palabra antes de la expresión cuando la condición resulta en Falso
4	PROGRAMA	Palabra reservada que representa la inicialización de un

		programa, seguida de su identificador
5	VAR	Palabra reservada que representa la inicialización de variables en contexto global
6	INT	Palabra reservada que representa el tipo CTEINT
7	FLOAT	Palabra reservada que representa el tipo CTEFLOAT
8	STRING	Palabra reservada que representa el tipo CTESTRING
9	CHAR	Palabra reservada que representa el tipo CTECHAR
10	DATAFRAME	Palabra reservada que representa el tipo DATAFRAME
11	ID	Palabra reservada que representa el identificador
12	CTEINT	Palabra reservada que representa el tipo de variable entero
13	CTEFLOAT	Palabra reservada que representa el tipo de variable flotante
14	CTESTRING	Palabra reservada que representa el tipo de variable string
15	CTECHAR	Palabra reservada que representa el tipo de variable char
16	FUNCION	Palabra reservada que representa la palabra antes de inicializar una función, seguida de su identificados
17	RETURN	Palabra reservada que llama la función de regresar una variable
18	LEE	Palabra reservada que llama la función de leer uno o más identificadores
19	ESCRIBE	Palabra reservada que llama la función de escribir alguna expresión o string
20	CARGAARCHIVO	Palabra reservada que llama la función de cargar un archivo
21	RUTA	Palabra reservada que representa la ruta de donde se va a obtener el archivo a cargar
22	MIENTRAS	Palabra reservada que representa la palabra antes de la expresión en un estatuto condicional

23	HAZ	Palabra reservada que representa la palabra antes de la expresión a repetir
24	DESDE	Palabra reservada que representa la palabra antes de la variable inicial en un estatuto no-condicional
25	HASTA	Palabra reservada que representa la palabra antes de la variable de control en un estatuto no-condicional
26	HACER	Palabra reservada que representa la palabra antes de la expresión a repetir
27	PRINCIPAL	Palabra reservada que representa el inicio de la función principal del programa
28	VOID	Palabra reservada que representa que no hay tipo de retorno para la función
Literales		
29	;	Punto y coma
30	,	Coma
31	:	Dos puntos
32	{	Llave izquierda
33	}	Llave derecha
34	=	Igual a
35	(	Paréntesis izquierdo
36	)	Paréntesis derecho
37	+	Suma
38	-	Resta
39	*	Multiplica
40	/	División
41	<	Menor a
42	>	Mayor a

43	[	Corchete derecho
44	]	Corchete izquierdo

3.5 Descripción del Análisis de Sintaxis:

3.5.1 Gramática Formal Empleada:

Número	Nombre	Gramática
1	preprograma	*catcherprograma* “;” *programa*
2	catcherprograma	PROGRAMA ID
3	programa	*programa2* PRINCIPAL *contextoprograma* “(” “)” *bloque fin*
4	programa	PRINCIPAL *contextoprograma* “(” “)” *bloque fin*
5	contextoprograma	
6	programa2	*vars programa3*
7	programa2	*programa3*
8	programa3	*funcs programa3*
9	programa3	funcs
10	vars	*varshelper var2*
11	varshelper	VAR
12	var2	*tipo var3* “;” *var2*
13	var2	*tipo var3* “;”
14	var3	*varhelp2* “[” *var4 arrCalc*
15	var3	*varhelp2* “[” *var4 arrCalc* “,” *var3*
16	var3	*varhelp var3*
17	var3	*varhelp*

18	var4	CTEINT “]” “[” CTEINT “]”
19	var4	CTEINT “]”
20	varhelp	ID “,”
21	varhelp	ID
22	varhelp2	ID
23	arrCalc	
24	tipo	INT
25	tipo	FLOAT
26	tipo	STRING
27	tipo	CHAR
28	tipo	DATAFRAME
29	tipo	VOID
30	param	*paramhelp “,” param*
31	param	*paramhelp*
32	paramhelp	*tipo* ID
33	funcs	*funcshelper funcs2*
34	funcshelper	FUNCION *tipo* ID
35	funcs2	“(” *param* “)” *countparam funcs3*
36	funcs2	“(” “)” *countparam funcs3*
37	countparam	
38	funcs3	*vars bloque funcs4*
39	funcs3	*bloque funcs4*
40	funcs4	
41	bloque	“{” “}”

42	bloque	“{” *bloque2*
43	bloque2	*estatuto* “}”
44	bloque2	*estatuto bloque2*
45	estatuto	*asignacion*
46	estatuto	*escritura*
47	estatuto	*void*
48	estatuto	*retorno*
49	estatuto	*lectura*
50	estatuto	*cargadatos*
51	estatuto	*condicional*
52	estatuto	*nocondicional*
53	asignacion	ID “=” *expresion* “;”
54	void	*verifyid* “(” “)” “;”
55	void	*verifyid* “(” *void2* “;”
56	verifyid	ID
57	void2	*expression void4* “,” *void5 void2*
58	void2	*Expression void4* “)” *void6*
59	void4	
60	void5	
61	void6	
62	retorno	REGRESA “(” *exp* “)” “,”
63	lectura	LEE “(” *lectura2* “;”
64	lectura2	*lectura3* “)”
65	lectura2	*lectura3* “,” *lectura2*

66	lectura3	ID
67	escritura	ESCRIBE “(” *escritura2*
68	escritura2	*escritura4* “)” “;”
69	escritura2	*escritura4* “,” *escritura2*
70	escritura2	*escritura3* “)” “;”
71	escritura2	*escritura3* “,” *escritura2*
72	escritura4	CTESTRING
73	escritura3	*expresion*
74	cargadatos	CARGAARCHIVOS “(” ID “,” RUTA “,” INT “,” INT “)” “;”
75	expresion	*exp expresion2*
76	expresion	*exp*
77	expresion2	“<” *exp*
78	expresion2	“>” “=” *exp*
79	expresion2	“<” “=” *exp*
80	expresion2	“>” *exp*
81	expresion2	“<” “>” *exp*
82	expresion2	“=” “=” *exp*
83	decision	*decision1* ENTONCES *bloque decision2* SINO *bloque*
84	decision	*decision1 ENTONCES bloque*
85	decision1	SI “(” *expresion* “)”
86	decision2	
87	condicional	MIENTRAS *condicional1* “(” *expresion* “)” *condicional2* HAZ *bloque condicional3*
88	condicional1	
89	condicional2	

90	condicional3	
91	nocondicional	DESDE *nocondicional1* “=” *exp nocondicional2* HASTA *exp nocondicional3* HACER *bloque nodoncional4*
92	nocondicional1	ID
93	nocondicional2	
94	nocondicional3	
95	nocondicional4	
96	exp	*termino validatipos1*
97	validatipos1	
98	validatipos2	
99	exp	*pushomas exp*
100	pushomas	*termino validatipos1* “+”
101	exp	*pushomin exp*
102	pushomin	*termino validatipos1* “-”
103	termino	*factor validatipos2*
104	termino	*pushomult termino*
105	pushomult	*factor validatipos2* “*”
106	termino	*pushodiv termino*
107	pushodiv	*factor validatipos2* “/”
108	factor	“(” *expresion* “)”
109	factor	“+” *varcte*
110	factor	“-” *varcte*
111	factor	*varcte*
112	varcte	*ID*
113	varcte	*CTEINT*

114	varcte	*CTEFLOAT*
115	varcte	*CTESTRING*
116	varcte	*CTECGAR*
117	fin	

3.6 Descripción de Generación de Código Intermedio y Análisis Semántico:

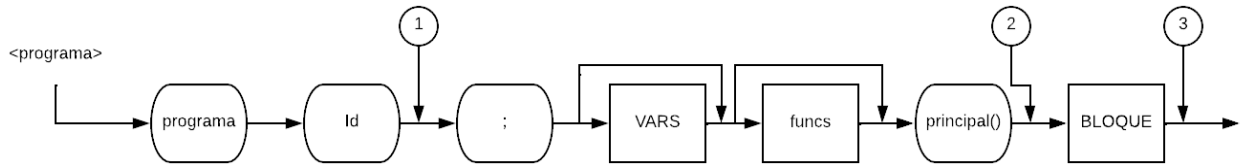
3.6.1 Código de Operación y Direcciones Virtuales:

```
#rangos
#global
self.intG = 0
self.floatG = 2500
self.charG = 5000
self.stringG = 7500
self.dataFrameG = 10000
#local
self.intL = 12500
self.floatL = 15000
self.charL = 17500
self.stringL = 20000
self.dataFrameL = 22500
#temp
self.intT = 25000
self.floatT = 27500
self.boolT = 30000
#const
self.intC = 32500
self.floatC = 35000
self.charC = 37500
self.stringC = 40000

#rango de tipos
#orden = [global, local, temp, const]
self.intRango = [2500, 15000, 27500, 35000]
self.floatRango = [5000, 17500, 30000, 37500]
self.charRango = [7500, 20000, -1, 40000]
self.string = [10000, 22500, -1, 42500]
self.bool = [-1, -1, 32500, -1]
```

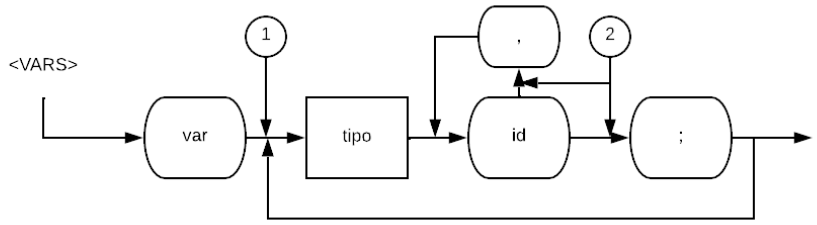
3.6.2 Diagramas de Sintaxis

Programa:



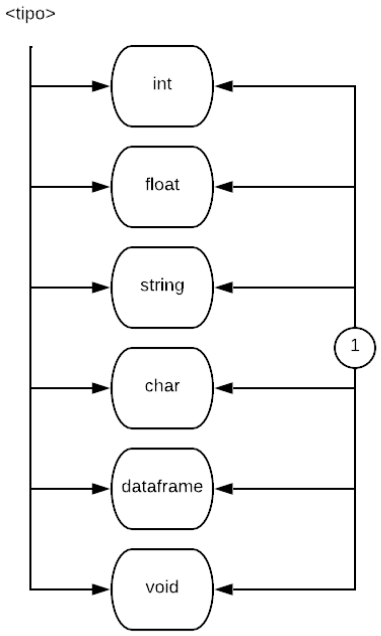
1. Inicializa funDir, genera cuádruplo de GOTO y agrega ‘1’ a la tabla de constantes
2. Cambia el scope a principal
3. Manda datos a la máquina virtual y empieza a correr

Vars:



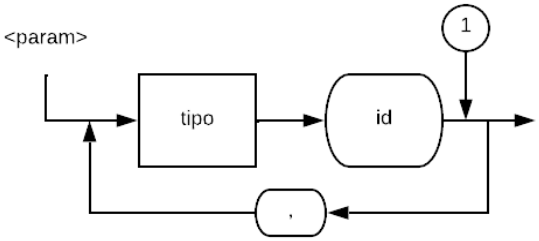
1. Se checa el scope actual
2. Si no existe ya, se agrega la variable al funDir. Si es un arreglo se valida que sea correcto

Tipo:



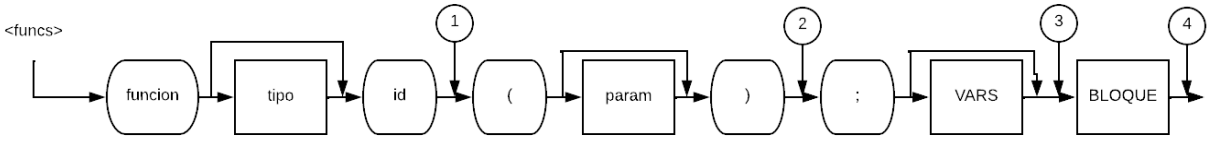
1. Guarda el tipo de la variable en funDir

Param:



1. Si no existe ya, se agrega la variable a funDir, luego se agrega a la tabla de parámetros.

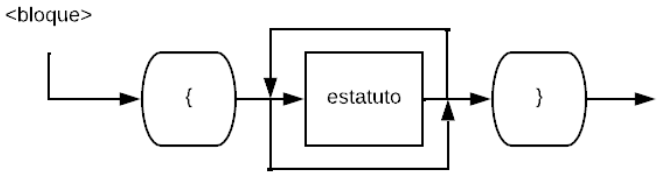
Funcs:



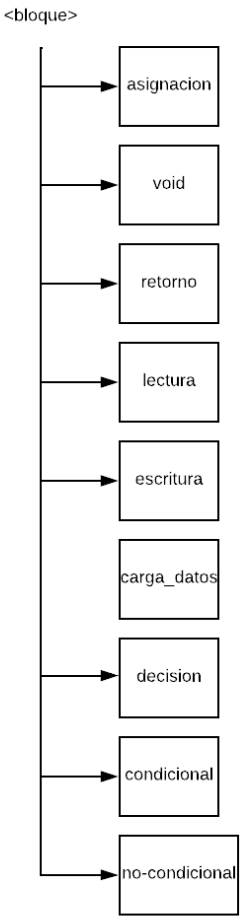
1. Se guarda el scope de la función, se guarda el tipo de retorno y si no está ya declarada se agrega la función a funDir

2. Se cuenta el número de parámetros que tiene la función y se guarda
3. Se cuenta el número de variables que contiene la función y se guarda
4. Se genera el cuádruplo de ENDFunc

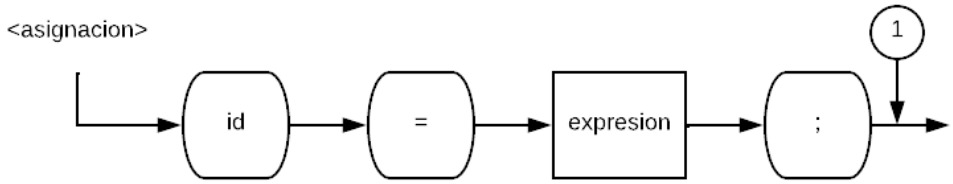
Bloque:



Estatuto:

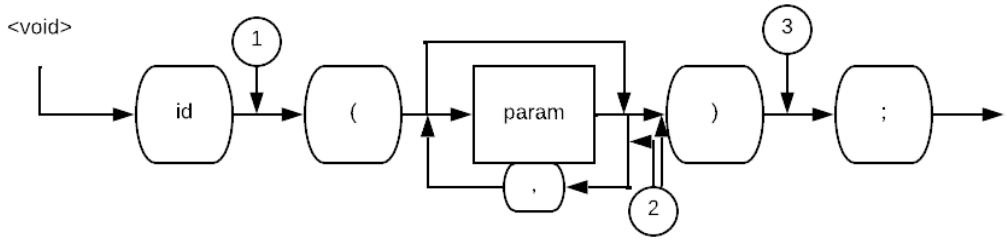


Asignacion:



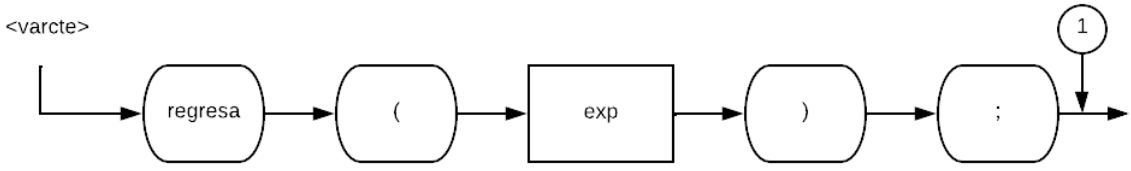
1. Si los tipos de la expresión son correctos, se genera el cuadrupolo de asignación

Void:



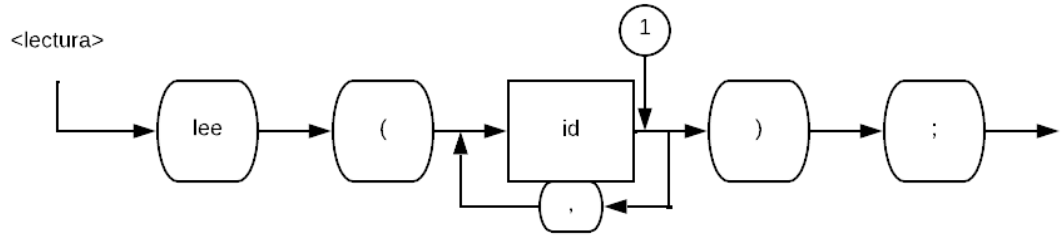
1. Se verifica si la función ya existe, si no se agrega un nuevo scope y se genera un cuádruplo de ERA
2. Se valida el parámetro, si es correcto se genera un cuádruplo de PARAMETER y se agrega al contador
3. Se genera el cuadruplo de GOSUB

Retorno:



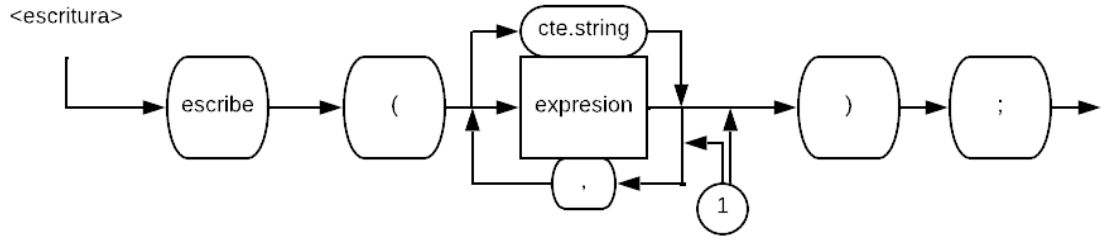
1. Se mete el valor y el tipo resultante de la expresión en la pila de operadores/tipos, se genera el cuádruplo de REGV y se genera el cuádruplo de ENDFUNC

Lectura:



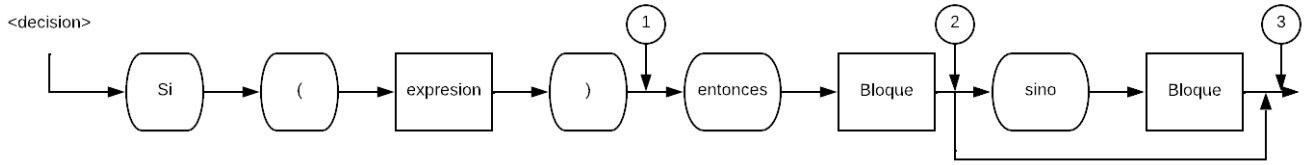
1. Se verifica que exista la variable y si existe se genera el cuádruplo de LEE

Escritura:



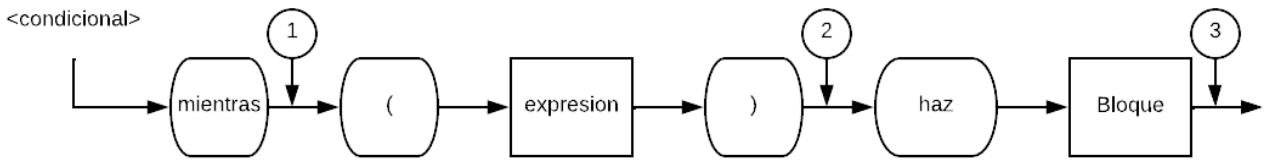
1. Si es un string se valida si existe o no, si no existe se agrega la variable, luego se genera el cuádruplo de ESCRIBE. Si es una expresión se resuelve y se genera el cuádruplo de ESCRIBE

Decision:



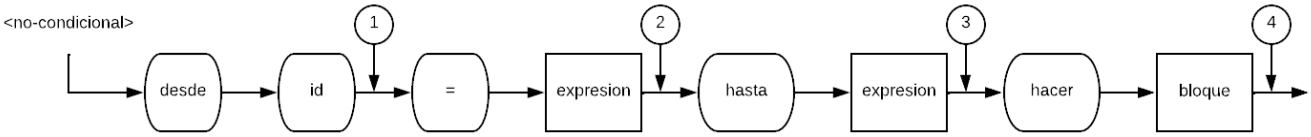
1. Si el resultado de la expresión no es tipo bool se genera error, si lo es se genera el cuádruplo de GoToF y se hace manejo de memoria para los saltos
2. Se genera el cuádruplo de GOTO y se hace manejo de memoria para los saltos
3. Se hace manejo de memoria para los saltos

Condicional:



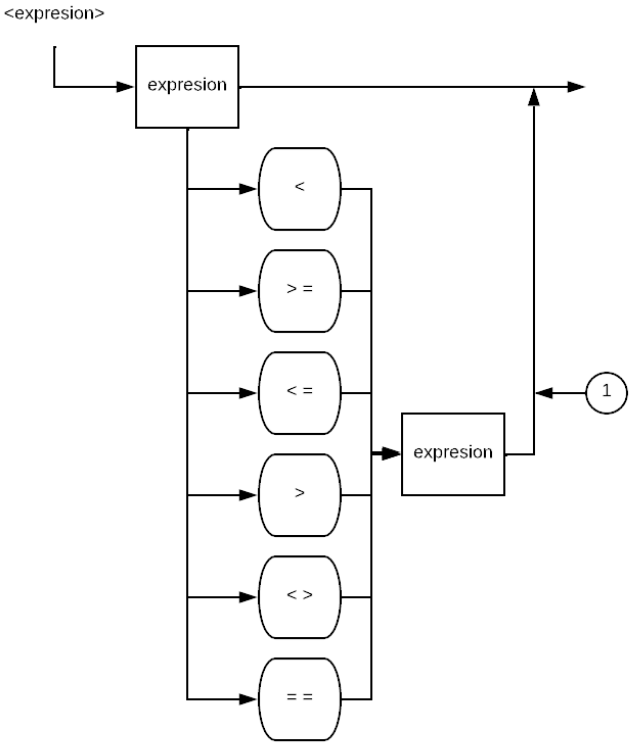
1. Se hace manejo de memoria para los saltos
2. Si el resultado de la expresión no es bool se genera error, si lo es se genera el cuádruplo GoToF y se hace manejo de memoria para los saltos.
3. Se genera el cuádruplo de GOTO y se hace manejo de memoria para los saltos

No-Condicional:



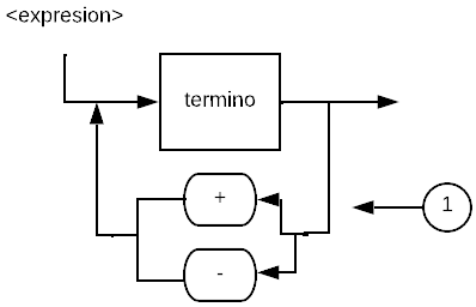
1. Si el tipo de variable del id is int o float, se empuja a la pila de operadores y de tipos
2. Si el resultado de la expresión es int o float se guarda la variable de control, pero si el tipo de la la variable de control no es igual a el tipo de la variable id resulta en error
3. Si el resultado de la expresión es int o float se hace la operación para validar que se entre en el ciclo, se genera el cuádruplo de GoToF y se hace manejo de memoria para los saltos
4. Se agrega 1 a la variable de control, se genera el cuádruplo de GOTO y se hace manejo de memoria para los saltos

Expresion:



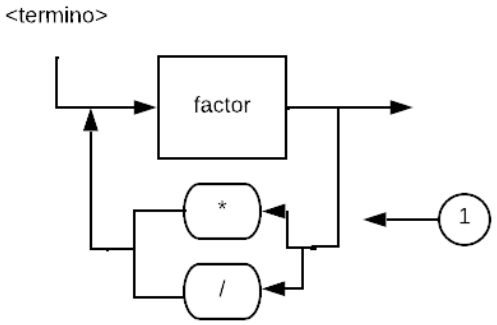
1. Se valida que la expresión sea válida, si es válida se genera el cuádruplo perteneciente a la operación y se guarda el resultado

Exp:



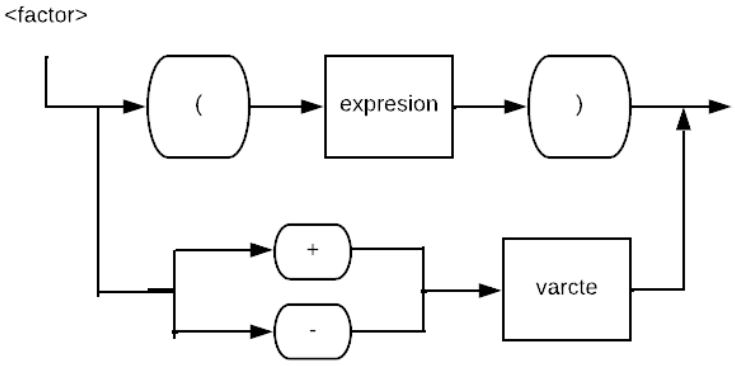
1. Se validan los tipos de los términos, si son compatibles se genera el cuádruplo de la operación

Termino:

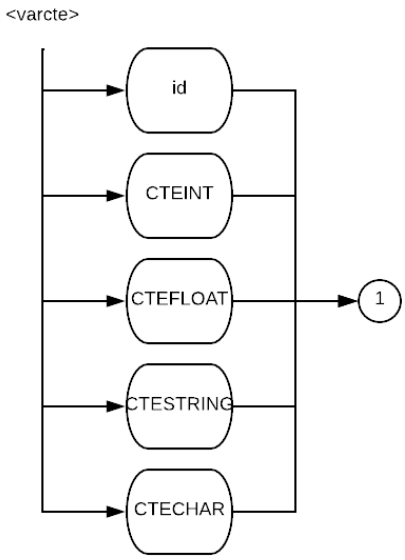


1. Se validan los tipos de los factores, si son compatibles se genera el cuádruplo de la operación

Factor:



VarCTE:



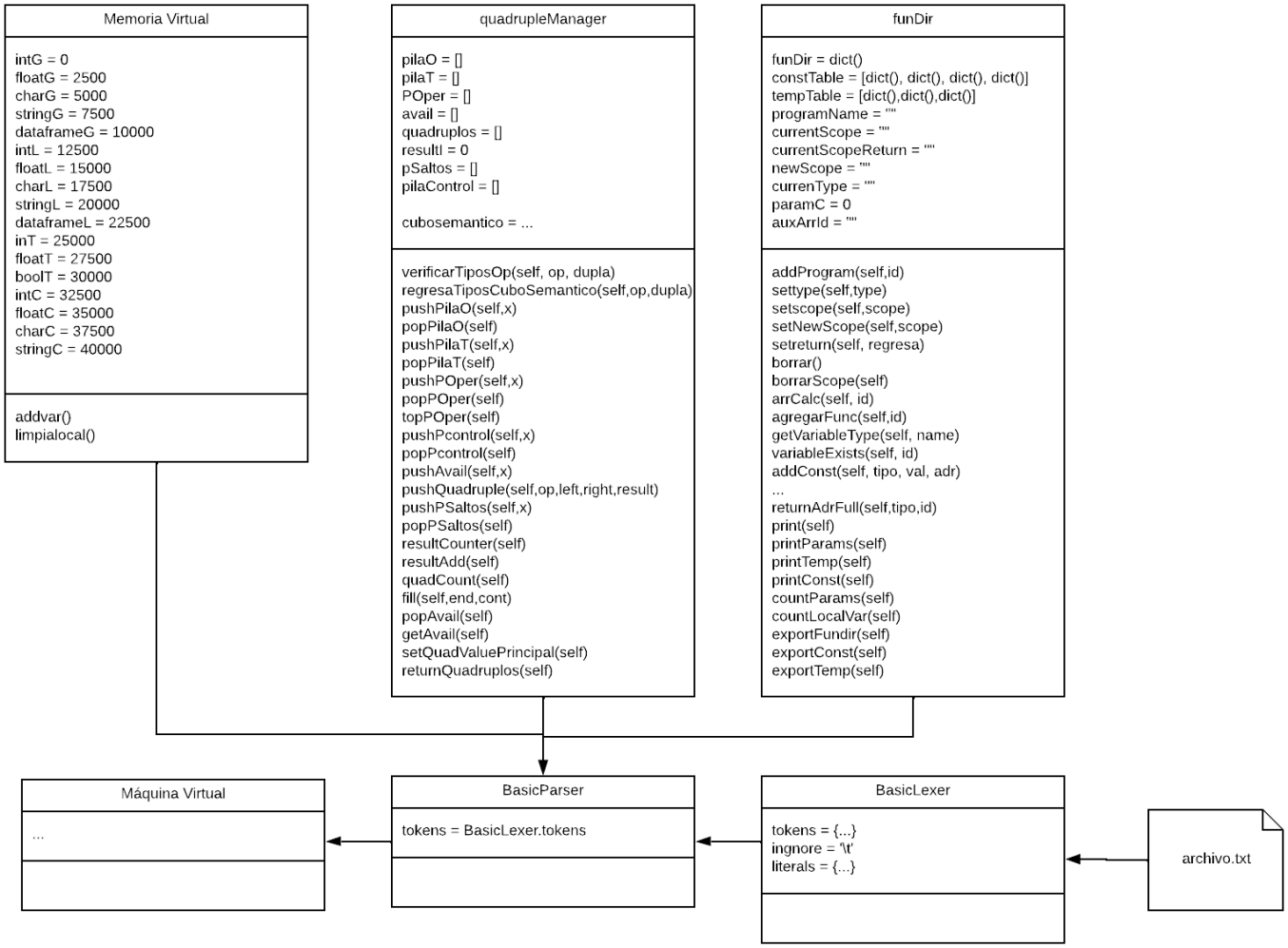
1. Se guarda el valor y el tipo en la pila de operadores/tipos, se valida si está en la tabla de constantes y si no está se guarda en ella.

3.6.3 Tabla de Consideraciones Semánticas

Tipos		Operación												
		-	+	/	*	<	>	<=	>=	!=	==	&&	||	!
int	**int**	int	int	float	int	bool	bool	bool	bool	bool	bool	-	-	-
int	**float**	float	float	float	float	bool	bool	bool	bool	bool	bool	-	-	-
float	**int**	float	float	float	float	bool	bool	bool	bool	bool	bool	-	-	-
float	**float**	float	float	float	float	bool	bool	bool	bool	bool	bool	-	-	-
string	**string**	-	-	-	-	-	-	-	-	bool	bool	-	-	-
bool	**bool**	-	-	-	-	-	-	-	-	-	-	bool	bool	-
bool	**-**	-	-	-	-	-	-	-	-	-	-	-	-	bool

3.7 Descripción Detallada del Proceso de Administración de Memoria:

3.7.1 Especificación Gráfica:



Es importante la estructura de datos que tenemos, pues ya todo funciona como engranajes en un reloj. Si se le mueve a algo para de funcionar. Las pilas las necesitamos para poder hacer operaciones de Pop y Push, las cuales son necesarias para el manejo de muchos datos a través de diferentes funciones. Las listas de python nos dejan hacer Push, Pop entre otros sin problemas. En funDir se utilizaron diccionarios porque ya que estamos usando los nombres de las variables, el diccionario es la manera más rápida de encontrar valores si se tiene su llave. Además con esta estructura de datos pudimos esconder arreglos bajo valores de los diccionarios y crear una estructura de datos eficiente. Finalmente la Memoria Virtual es necesaria para facilitar el uso de módulos o funciones sin tener que andar pasando valores entre diversas funciones o dejándolos como globales. Esta estructura es relativamente más difícil de implementar pero es muy eficiente si se hace y se usa correctamente.

4 Descripción de la Máquina Virtual

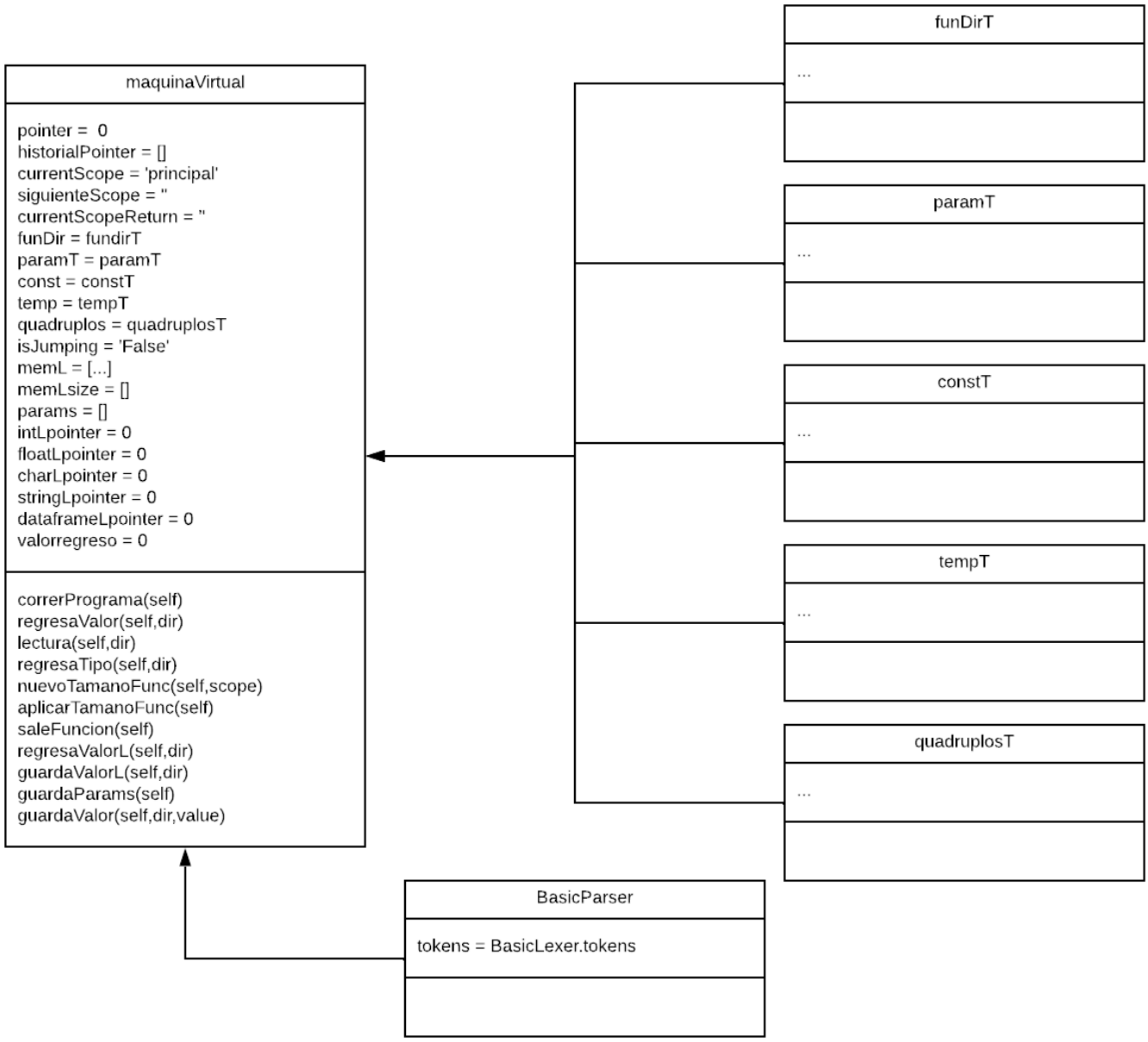
4.1 Descripción General de Recursos:

4.1.1 Equipo de Cómputo: La máquina virtual fue desarrollada en equipos Windows 11 y Linux Linux (4.4.0/22000/Microsoft)

4.1.2 Lenguaje Usado: El lenguaje utilizado fue Python

4.1.3 Utilerías Especiales Usadas: No se utilizó ninguna utilería especial

4.2 Descripción Detallada del Proceso de Administración de Memoria:



Primero se crean las direcciones virtuales usando nuestra clase de memoria virtual durante la etapa de parsing. Al momento de usar la máquina virtual el uso de memoria se divide en tipos.

Memoria local: Esta es la parte de memoria que tiene que poder tener nuevos contextos para poder usar recursividad sin arruinar las variables, se usa un sistema de offsets donde una variable tiene una dirección 12001 y para acceder en memoria primero se le resta la base de el tipo de datos ej 12000 entonces la dirección nueva seria 1, lugea se le suma el apuntador para mandar el numero a el contexto correcto, finalmente se hace la operación necesaria.

Memoria global: Como la memoria global no planea incrementar mágicamente esta se dejó en un diccionario para acceder más rápido a sus valores, las llaves del diccionario fueron cambiadas a direcciones virtuales al exportar la tabla

Memoria temporal: A falta de tiempo no logramos implementar la memoria como debería, similar a la local, esta se guarda en un diccionario y se accede usando la dirección virtual.

Variables Constantes: Igualmente como las variables constantes no cambian estas se guardan en un diccionario (para poder usar for loops siempre se guarda el numero 1 aunque el usuario no lo introduzca en su código)

5 Pruebas de Comprobación de Funcionalidad:

5.1.1 Codificación de Prueba 1:

patito.txt

```
programa covid19;

var
	int i,j,p,maxVariables,maxRenglones;
	string o;
	float f1;
	dataframe ash;

funcion int fact(int j, string k)
var int i;
string r;
{
	r = "test";
	p = 3;
	i = j + p-j*2+j;
	f1 = 2.0*1;
	escribe(r);
	si(j == 1) entonces{
	lee(i);
	i = 3;
	}sino{
	lee(j);
```

```
        p = 5;
        }
}

funcion void yolo(int a, float j)
var char y;
int l;
{
        lee(y);
}


principal(){
        i = 5+3;
        o = "test";
        fact(i,o);
        lee(p);
        escribe(p);
        escribe(1+2);
        escribe("HOLA");
        escribe('c');
        f1 = 20.5;
        yolo(5,f1);
        mientras(i > 0) haz {
        escribe("test");
        i = i-1;
        }

        desde i = 3 hasta 10 hacer{
        escribe(i*2);
        }

        j = p*2;

}
```

5.1.2 Resultados Arrojados por Generación de Código Intermedio y Ejecución:

```
------------------------
0| GOTO | | |20
------------------------
1| = |32501 | |2
------------------------
2| + |12500 |2 |25000
```

```
-----------------------
3| * |12500 |32502 |25001
-----------------------
4| - |25000 |25001 |25002
-----------------------
5| + |25002 |12500 |25003
-----------------------
6| = |25003 | |12501
-----------------------
7| * |35000 |32503 |27500
-----------------------
8| = |27500 | |2500
-----------------------
9| ESCRIBE | | |20001
-----------------------
10| == |12500 |32503 |30000
-----------------------
11| GotoF |30000 | |15
-----------------------
12| LEE | | |12501
-----------------------
13| = |32501 | |12501
-----------------------
14| GOTO | | |17
-----------------------
15| LEE | | |12500
-----------------------
16| = |32504 | |2
-----------------------
17| ENDFunc | | |
-----------------------
18| LEE | | |20000
-----------------------
19| ENDFunc | | |
-----------------------
20| + |32504 |32501 |25004
-----------------------
21| = |25004 | |0
-----------------------
22| ERA | | |fact
-----------------------
23| PARAMETER |0 | |param0
-----------------------
24| PARAMETER |7500 | |param1
```

```
------------------------
25| GOSUB |principal | |1
------------------------
26| LEE | | |2
------------------------
27| ESCRIBE | | |2
------------------------
28| + |32503 |32502 |25005
------------------------
29| ESCRIBE | | |25005
------------------------
30| ESCRIBE | | |40001
------------------------
31| ESCRIBE | | |37500
------------------------
32| = |35001 | |2500
------------------------
33| ERA | | |yolo
------------------------
34| PARAMETER |32504 | |param0
------------------------
35| PARAMETER |2500 | |param1
------------------------
36| GOSUB |principal | |18
------------------------
37| > |0 |32505 |30001
------------------------
38| GotoF |30001 | |43
------------------------
39| ESCRIBE | | |40000
------------------------
40| - |0 |32503 |25006
------------------------
41| = |25006 | |0
------------------------
42| GOTO | | |37
------------------------
43| = |32501 | |0
------------------------
44| = |32501 | |25007
------------------------
45| = |32506 | |25008
------------------------
46| < |25007 |25008 |30002
```

```
------------------------
47| GotoF |30002 | |54
------------------------
48| * |0 |32502 |25009
------------------------
49| ESCRIBE | | |25009
------------------------
50| + |25007 |32500 |25010
------------------------
51| = |25010 | |25007
------------------------
52| = |25010 | |0
------------------------
53| GOTO | | |46
------------------------
54| * |2 |32502 |25011
------------------------
55| = |25011 | |1
------------------------
codigo valido
valor
>5
>5
5
3
"HOLA"
'c'
>5
"test"
"test"
"test"
"test"
"test"
"test"
"test"
"test"
6
8
10
12
14
16
18
```

5.2.1 Codificación de Prueba 2:

factorial.txt

```
programa covid19;

var
        int i,k,j;

principal(){
        escribe("Que numero quires para factorial?");
        lee(i);
        i = i+1;
        j = 1;
        desde k = 1 hasta i hacer{
        j = j*k;
        }
        escribe(j);
}
```

5.2.2 Resultados Arrojados por Generación de Código Intermedio y Ejecución:

```
------------------------
0| GOTO | | |1
------------------------
1| ESCRIBE | | |40000
------------------------
2| LEE | | |0
------------------------
3| + |0 |32501 |25000
------------------------
4| = |25000 | |0
------------------------
5| = |32501 | |2
------------------------
6| = |32501 | |1
------------------------
7| = |32501 | |25001
------------------------
8| = |0 | |25002
------------------------
9| < |25001 |25002 |30000
------------------------
10| GotoF |30000 | |17
------------------------
```

```
11| * |2 |1 |25003
-----------------------
12| = |25003 | |2
-----------------------
13| + |25001 |32500 |25004
-----------------------
14| = |25004 | |25001
-----------------------
15| = |25004 | |1
-----------------------
16| GOTO | | |9
-----------------------
17| ESCRIBE | | |2
-----------------------
codigo valido
"Que numero quires para factorial?"
>5
120
```

5.3.1 Codificación de Prueba 3:

fib.txt

```
programa covid19;

var
        int i,k,f;

funcion int fib(int j)
var int a,b;
{
        si(j <= 1)entonces{
        regresa(j);
        }
        a = fib(j-1);
        b = fib(j-2);
        f = a+b;
        regresa(f);
}

principal(){
        escribe("Numero de niveles de fibonacci");
        lee(i);
        k = fib(i);
        escribe(k);
```

}

5.3.2 Resultados Arrojados por Generación de Código Intermedio y Ejecución:

```
-----------------------
0| GOTO | | |20
-----------------------
1| <= |12500 |32501 |30000
-----------------------
2| GotoF |30000 | |5
-----------------------
3| REGV |12500 | |
-----------------------
4| ENDFunc | | |
-----------------------
5| ERA | | |fib
-----------------------
6| - |12500 |32501 |25000
-----------------------
7| PARAMETER |25000 | |param0
-----------------------
8| GOSUB |fib | |1
-----------------------
9| =RET | | |12501
-----------------------
10| ERA | | |fib
-----------------------
11| - |12500 |32502 |25001
-----------------------
12| PARAMETER |25001 | |param0
-----------------------
13| GOSUB |fib | |1
-----------------------
14| =RET | | |12502
-----------------------
15| + |12501 |12502 |25002
-----------------------
16| = |25002 | |2
-----------------------
17| REGV |2 | |
-----------------------
18| ENDFunc | | |
-----------------------
```

```
19| ENDFunc | | |
-----------------------
20| ESCRIBE | | |40000
-----------------------
21| LEE | | |0
-----------------------
22| ERA | | |fib
-----------------------
23| PARAMETER |0 | |param0
-----------------------
24| GOSUB |principal | |1
-----------------------
25| =RET | | |1
-----------------------
26| ESCRIBE | | |1
-----------------------
codigo valido
"Numero de niveles de fibonacci"
>19
4181
```

5.4.1 Codificación de Prueba 4:

forloops.txt

```
programa forLoops;
var
        int i,j,p;

principal(){
        escribe("cuantos loops quieres?");
        lee(i);

        desde j = 0 hasta i hacer{
        escribe(j);
        }
}
```

5.4.2 Resultados Arrojados por Generación de Código Intermedio y Ejecución:

```
-----------------------
0| GOTO | | |1
-----------------------
1| ESCRIBE | | |40000
-----------------------
```

```
2| LEE | | |0
------------------------
3| = |32501 | |1
------------------------
4| = |32501 | |25000
------------------------
5| = |0 | |25001
------------------------
6| < |25000 |25001 |30000
------------------------
7| GotoF |30000 | |13
------------------------
8| ESCRIBE | | |1
------------------------
9| + |25000 |32500 |25002
------------------------
10| = |25002 | |25000
------------------------
11| = |25002 | |1
------------------------
12| GOTO | | |6
------------------------
codigo valido
"cuantos loops quieres?"
>10
0
1
2
3
4
5
6
7
8
9
```

5.5.1 Codificación de Prueba 5:

recursion.txt

```
programa covid19;
var
        int i,p;

funcion int inicia(int y)
```

```
var int x;
{
        x = 3;
        y = y*2;
        escribe(y);
        si(y<100)entonces{
        inicia(y);
        }
}

principal(){
        inicia(5);
}
```

5.5.2 Resultados Arrojados por Generación de Código Intermedio y Ejecución:

```
------------------------
0| GOTO | | |11
------------------------
1| = |32501 | |12501
------------------------
2| * |12500 |32502 |25000
------------------------
3| = |25000 | |12500
------------------------
4| ESCRIBE | | |12500
------------------------
5| < |12500 |32503 |30000
------------------------
6| GotoF |30000 | |10
------------------------
7| ERA | | |inicia
------------------------
8| PARAMETER |12500 | |param0
------------------------
9| GOSUB |inicia | |1
------------------------
10| ENDFunc | | |
------------------------
11| ERA | | |inicia
------------------------
12| PARAMETER |32504 | |param0
------------------------
13| GOSUB |principal | |1
------------------------
```

```
codigo valido
10
20
40
80
160
```

5.6.1 Codificación de Prueba 6:

while.txt

```
programa covid19;

var
        int i,k,f;


principal(){
        i = 1;
        k = 100;
        mientras(i < k) haz{
        escribe(i);
        i = i*2;
        }
}
```

5.6.2 Resultados Arrojados por Generación de Código Intermedio y Ejecución

```
------------------------
0| GOTO | | |1
------------------------
1| = |32501 | |0
------------------------
2| = |32502 | |1
------------------------
3| < |0 |1 |30000
------------------------
4| GotoF |30000 | |9
------------------------
5| ESCRIBE | | |0
------------------------
6| * |0 |32503 |25000
------------------------
7| = |25000 | |0
------------------------
8| GOTO | | |3
```

```
------------------------
codigo valido
1
2
4
8
16
32
64
```

5.7.1 Codificación de Prueba 7:

error.txt

```
programa covid19;

var
        int i,k,f;


principal(){
        escribe(a);
}
```

5.7.2 Resultados Arrojados por Generación de Código Intermedio y Ejecución

```
------------------------
0| GOTO | | |1
------------------------
1| ESCRIBE | | |-1
------------------------
codigo valido
no existe este valor
-1
```

6 Manual de Usuario

6.1 Guía de Referencia Rápida

Primero que nada se tiene que instalar la librería de Python SLY corriendo el comando de "*pip install sly*" en la terminal del proyecto para poder utilizar el lenguaje.

Luego, para correr un programa se pone en la terminal "*python3 main.py programa*" programa siendo el nombre del programa que deseas correr.

¡Ya con eso estás listo para programar con PePa–!

A continuación se presenta una lista de todo lo que se podría necesitar para aprender a programar en PePa–.

Las secciones en bold son palabras reservadas, las secciones en itálicas son opcionales y el "//" indica comentario.

6.1.1 La Estructura General de un programa escrito en PePa– es:

```
Programa Nombre_Programa ;

//Declaración de variables globales
var
        //variables
//Declaración de funciones
funcion nombreFuncion(){
}

//Procedimiento principal
principal(){
        //Estatutos
}
```

6.1.2 Para la Declaración de Variables Globales o Locales:

```
var
        tipo listaIds ;
        < tipo listaIds; >...
```

Donde:

tipo = **int, float, string, char**

listaIds = Uno o más identificadores separados por comas. Se pueden inicializar con un valor.

Ejemplo:

```
var
    int a, b = 4;
```

En el ejemplo se definen dos variables enteras (a, no inicializada y b, inicializada con un valor de 4).

6.1.3 Para la Declaración de Funciones: //Se pueden definir 0 o más funciones

funcion *tipo_retorno* nombreFuncion (*parámetros*)
declaración de variables locales
{
    estatutos
}

Donde:

tipo_retorno: Puede ser cualquier tipo soportado (**int, float, string, char**) o **void** si no regresa valor.
parámetros: Los parámetros siguen la sintaxis de declaración de variables sin valores asignados.

6.1.4 Para la Declaración de Estatutos**:**
La sintaxis básica de cada uno de los estatutos en el lenguaje PePa++ es:

6.1.4.1 Asignación:

Id = Expresión ;

o

Id<dimensiones> = nombreFuncion(*parametros*) *;*

Donde:

A un identificador se le asigna el valor de una expresión o de el valor del retorno de una función o una combinación de ambas.

6.1.4.2 Llamada a Funcion Void:

nombreFuncion(*parametros*) ;

Donde:

Se manda a llamar una función que no regresa valor.

6.1.4.3 Retorno de una función:

regresa(exp)

Donde:

Se indica y se regresa el valor de retorno de la función.

6.1.4.4 Lectura:

lee(id*, id...*) ;

Donde:

Se pueden leer uno o más identificadores separados por comas.

6.1.4.5 Escritura:

escribe(CTESTRING o expresión, *<CTESTRING o expresión>...*) ;

Donde:

Se pueden escribir CTESTRINGS y/o resultados de expresiones separados por comas.

6.1.4.6 Estatuto de Decisión:

si(expresion) **entonces** {
estatutos
}
sino { //opcional
estatutos
}

Donde:

Se puede escribir con o sin "sino". Si no se escribe y la expresión resulta en falso se salta a la siguiente instrucción después del estatuto de decisión.

6.1.4.7 Estatuto de Repetición (Condicional):

mientras (expresión) **haz** {
estatutos
}

Donde:

Se repiten los estatutos mientras la expresión sea verdadera.

6.1.4.8 Estatuto de Repetición (No-Condicional):

desde Id = exp **hasta** exp **hacer** {
estatutos
}

Donde:
Se repiten los estatutos hasta que id sea igual a exp.

6.1.4.9 Expresiones:
Las expresiones en PePa++ consisten de operadores aritméticos, lógicos y relacionales como: **+, -, *, /, <, >, ==.** Se manejan las prioridades de PEMDAS tradicionales. Existen identificadores, palabras reservadas, constantes: enteras, flotantes, char y string.

6.1.4.10 Ejemplo de Programa Completo:
:-)

6.2 Video Demo

https://youtu.be/ACbl2SufGLI