

Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Monterrey

Compilador: ABC

A01283076 Antonio Díaz Ramos

A01275834 Citlalli Rosario Alonzo Mateos

Diseño de Compiladores
M. C. Elda Guadalupe Quiroga
Dr. Héctor Ceballos

22 de noviembre de 2022

Índice

Descripción y Documentación Técnica del Proyecto	3
Descripción del Proyecto	4
Propósito y Alcance del Proyecto	4
Análisis de Requerimientos	4
Principales Test Cases	5
Proceso General para el Desarrollo del Proyecto	6
Descripción del Lenguaje	15
Nombre del Lenguaje	15
Características del Lenguaje	15
Listado de Errores	16
Descripción del Compilador	19
Herramientas Utilizadas	19
Descripción del Análisis Léxico	19
Descripción del Análisis de Sintaxis	21
Descripción de Generación de Código Intermedio y Análisis Semántico	29
Administración de Memoria	53
Descripción de la Máquina Virtual	55
Herramientas Utilizadas	55
Administración de Memoria	56
Pruebas	58
Documentación del Código del Proyecto	65
Manual de Usuario	72
Configuración del Ambiente	73
Compilación del Programa	73
Ejecución del Programa	73
Estructura General del Programa	74
Declaración de Variables	75
Declaración de Arreglos	76
Declaración de Funciones	76
Declaración del Main	77
Estatutos	78

Descripción y Documentación Técnica del Proyecto

Descripción del Proyecto

Propósito y Alcance del Proyecto

El proyecto final de compiladores tiene como objetivo que nosotros como estudiantes de la carrera de ingeniería en tecnologías computacionales seamos capaces de crear un lenguaje de programación, junto con su compilador, el archivo que se genera en compilación y la máquina virtual. Todo lo anterior haciendo uso de los conocimientos que hemos adquirido a lo largo de nuestro plan de estudios y los temas vistos en esta materia.

Debido al tiempo que tenemos para llevar a cabo esta tarea, el “universo” de nuestro lenguaje es muy pequeño a comparación de los que existen actualmente y de los que hacemos uso (C++, Java, JavaScript, Python, etc.). Sin embargo, si se toman en cuenta los elementos básicos y que son importantes para una programación básica, por lo tanto se incluye el uso y declaración de variables, operaciones aritméticas (suma, resta, multiplicación, división), operaciones lógicas (or, and, not), operaciones relacionales, estatutos secuenciales (read, write), estatutos condicionales (if, while), estatutos cíclicos (for), funciones, arreglos y matrices.

Análisis de Requerimientos

Es necesario hacer una recopilación de los aspectos más importantes que se deben tener en cuenta para medir el éxito del proyecto y saber si nuestro lenguaje ABC cumple con lo que se espera al final del semestre. Los requerimientos que se tomarán en cuenta son los siguientes:

- La gramática para el lenguaje debe ser no ambigua.
- El lenguaje debe soportar la declaración de variables locales.
- El lenguaje debe soportar la declaración de variables globales.
- El lenguaje debe soportar el uso de variables locales.

- El lenguaje debe soportar el uso de variables globales.
- El lenguaje debe soportar estatutos de asignación.
- El lenguaje debe soportar estatutos de lectura.
- El lenguaje debe soportar estatutos de escritura.
- El lenguaje debe soportar operaciones aritméticas.
- El lenguaje debe soportar operaciones lógicas.
- El lenguaje debe soportar operaciones relacionales.
- El lenguaje debe soportar estatutos condicionales.
- El lenguaje debe soportar estatutos cíclicos.
- El lenguaje debe soportar la declaración de funciones.
- El lenguaje debe soportar el uso de funciones.
- El lenguaje debe soportar la llamada recursiva a funciones.
- El lenguaje debe soportar la declaración de elementos estructurados.
- El lenguaje debe soportar el uso de elementos estructurados.
- El lenguaje debe estar orientado a un propósito específico.
- El compilador debe detectar errores léxicos, sintácticos y semánticos.
- El compilador debe generar código intermedio.
- El compilador debe ser capaz de destruir las estructuras que ya no son requeridas.
- El compilador debe ser eficiente en tiempo y memoria.
- La máquina virtual debe leer y almacenar la información del código intermedio.
- La máquina virtual debe manejar de manera dinámica la memoria local, global y de constantes.

Principales Test Cases

Los principales casos de prueba que consideramos para saber si nuestro compilador, máquina virtual y lenguaje estaban funcionando correctamente fueron los que se encuentran en los archivos:

- prueba.txt : Ayudó a comprobar que el léxico y la sintaxis funcionaran correctamente.
- programPatito.txt : Ayudó a empezar a realizar pruebas de cuádruplos. Se tomaron ejemplos de las clases y con eso pudimos corroborar el funcionamiento adecuado.
- test.txt : En este archivo se daba un funcionamiento real a las funciones y código implementado en el main para corroborar que los cuádruplos eran si eran los correctos. Además este archivo igual ya debía funcionar en la máquina virtual.
- test2.txt: Este archivo tiene la función de probar que los arreglos funcionen correctamente tanto en compilación como en ejecución.

Proceso General para el Desarrollo del Proyecto

Nosotros decidimos trabajar de manera presencial en todo el proyecto, es decir, ambos estuvimos pensando en cómo hacer cada especificación del proyecto desde el inicio hasta la finalización del mismo. Lo anterior con el propósito de que ambos conociéramos el código, entendiéramos todo lo que hace cada archivo, la participación fuera activa de ambas partes y de esta manera tener el mejor resultado posible.

Cada semana, desde el inicio del segundo parcial, se trabajó en el proyecto. De igual manera se hizo una entrega con el avance que teníamos del proyecto y lo explicamos en el ReadMe para tener una noción más clara de lo que fuimos logrando. Nuestro plan de trabajo fue el siguiente:

- Diseño de los diagramas de sintaxis.
- Creación del Scanner y Parser.
- Almacenamiento de las variables en la tabla de variables.
- Creación del Cubo Semántico.
- Creación del directorio de Funciones.
- Generación de código intermedio.
- Hacer que el compilador genere un archivo con el código intermedio.

- Creación de la máquina virtual.
- Hacer que la máquina virtual empiece a ejecutar.
- Generación de código intermedio para arreglos y matrices.
- Corrección de errores del compilador y máquina virtual.

Los avances que realizamos cada semana, explicados de manera general, fueron los que se describirán a continuación, también se pueden encontrar en la rama master y main del repositorio [StrictName/Compilador](#):

Avance 1

Sintaxis completa del programa, incluye declaración de clases (con sus atributos y métodos, y herencia simple), variables, funciones y el main. Hemos realizado diversas pruebas y consideramos que toda la sintaxis funciona y está completa, al igual que contempla lo necesario para que el compilador funcione al momento de revisar la sintaxis del lenguaje de programación.

La sintaxis que actualmente el programa maneja es la declaración del nombre de programa, clases, atributos, métodos, funciones, variables y estatutos como if, if-else, for, while-do, read y write.

Código de prueba:

```

program proyectocompilador;

class Animal
{
    attribute:
    public int x;
    private float s;

    method:
    private void prueba (int ys)
    {
        x = 5;
        y[3] = x;

        if(x+2 equal 2)
        {
            p = x;
        }

        else
        {
            metodo(x);
        }
    }
}

public int hello (int y, int z)
{
    y=3;

    while (x+2*3) do
    {
        mato(2);
        y = 2;
    }
    return e;
}

class Dog : public Animal
{
    attribute:
    public int x;
    private float s;

    method:
    private void prueba (int y)
    {
        if(5>3)
        {
            x=w;
        }

        return x*2;
    }
}

public int hello (int y, int z)
{
    x=3;

    return e;
}

var
Dog perro;
int edad;
int edad2;
float peso;
char caracter;
int array[1];
float array2[2][3];

func int suma (int y);
var
int edad3;
{
    x = 5.3;
    y = x;
    z = suma(x) + a - 6;

    read (x);
    read (x[2], x);
    read (x,w,x);
    write ('t', "jojaja");

    x='w';
    return x+2;
}

func void suma (int x, float y);
{
    x=2;
    y = p.ladra(temp1);

    y = p.ladra;

    p.come(x);
    p.duerma(3, y);
    p.juega(4.8);
    p.corre('a');
    p.atril;

    write("buenas", p.edad, p.sum(7, 3.4));

    if (x > p.edad and y < p.anos ) {
        y[1] = p.edad;
        if (v < p.calculo) {
            u = 1 + p;
        }
    }

    while (p.animal) do {
        x = p.corre(3);
    }

    from s = p.t > 2 to p.ala equal 5 do {
        m1(x, 8);
    }
}

func void suma (char z);
{
    y=3;
}

```

```

main()
{
    x=2;
    write(x+2);

    p.come(x);
    p.duerma(3, y);
    p.juega(4.8);
    p.corre('a');
    p.atril;

    from x = 2+3 to y do
    {
        read(w);
    }

    y = p.ladra(temp);

    y = p.ladra;

    write("buenas", p.edad, p.sum(7, 3.4));

    if (x > p.edad and y < p.anos ) {
        y[1] = p.edad;
        if (v < p.calculo) {
            u = 1 + p;
        }
    }

    while (p.animal) do {
        x = p.corre(3);
    }

    from s = p.t > 2 to p.ala equal 5 do {
        m1(x, 8);
    }
}

```

Avance 2

Se agregó el diccionario del cubo semántico, la tabla de variables y el directorio de funciones. El cubo semántico ya tiene los operadores derechos, operadores izquierdos, y operando con su respectivo resultado, además, se hizo la función que devuelve el resultado. La tabla de variables ya guarda las variables de todo el programa junto con su scope y tipo. El directorio de funciones aún no está conectado a la tabla de variables pero ya guarda el nombre de la función.

Avance 3

Se empiezan a generar los cuádruplos de las expresiones de asignación, lectura y escritura. También se generan los cuádruplos de while.

Avance 4

Se realizó la asignación de direcciones virtuales a las variables y constantes. Ya se genera el código intermedio de if y for, además, los cuádruplos se crean tomando en cuenta las direcciones virtuales.

Código de prueba:

<pre>program Compilador; var int i; int a; int b; float c; float d; int e; int f; float g; char x; bool hola; int j; func int suma (int y); var int varF1; { read(varF1); }</pre>	<pre>main() { for j = 1 to i + a * b do { b = i + a; } write ("Terminafor"); if (a > b) { read(a, b); } else { a = e + f; write(e, "Hola", a); } if (c < d) { g = c + d; } while (c < d) do { g = d; } c = d + e + i * a; }</pre>	<pre>1: Goto, 3, , 2: READ, , , 2000 3: =, 6000, , 5 4: =, 5, , VControl 5: *, 1, 2, 2001 6: +, 0, 2001, 2002 7: =, 2002, , VFinal 8: <, VControl, VFinal, 3500 9: GotoF, 3500, , 16 10: +, 0, 1, 2003 11: =, 2003, , 2 12: +, VControl, 1, 2004 13: =, 2004, , VControl 14: =, 2004, , 5 15: GOTO, , , 8 16: WRITE, , , "Terminafor" 17: >, 1, 2, 3501 18: GotoF, 3501, , 22 19: READ, , , 1 20: READ, , , 2 21: GOTO, , , 27 22: +, 3, 4, 2005 23: =, 2005, , 1 24: WRITE, , , 3 25: WRITE, , , "Hola" 26: WRITE, , , 1 27: <, 500, 501, 3502 28: GotoF, 3502, , 31 29: +, 500, 501, 2500 30: =, 2500, , 502 31: <, 500, 501, 3503 32: GotoF, 3503, , 35 33: =, 501, , 502 34: GOTO, , , 31 35: +, 501, 3, 2501 36: *, 0, 1, 2006 37: +, 2501, 2006, 2502 38: =, 2502, , 500</pre>
--	---	--

Avance 5

Se realizaron avances en cuanto a las funciones. Ya se guardan los datos completos de la función, como el cuádruplo en donde inicia su ejecución o la cantidad de recursos que ocupa cada función. De igual manera, se generan de los módulos como ENDFunc y RETURN. Ya se conecta la tabla de variables con su respectiva función.

Avance 6

Corrección de los cuádruplos de las funciones. Agregamos el "parche guadalupano", creación de los cuádruplos para una llamada recursiva. Creación de la máquina virtual: creación de la memoria de las constantes y almacenar valores, creación de la memoria global y local.

Avance 7

Compilación genera cuádruplos de arreglos. Máquina virtual ejecuta expresiones aritméticas, estatutos secuenciales y condicionales, módulos y parte de arreglos. Las pruebas de Fibonacci y Factorial corrieron y funcionaron.

La lista de commits realizadas durante el desarrollo del proyecto es la siguiente:

Fecha	Nombre	Breve descripción
25 Octubre 2022	Actualizado	Ya estaban implementadas las reglas gramaticales, pero no habíamos actualizado nada en el github, y empezamos a generar la tabla de variables, el cubo semántico y el directorio de funciones. Empezamos a guardar, con puntos neurálgicos, los símbolos de expresiones aritméticas.
	Última versión	Eliminamos errores
	Datos funciones (sin variables)	Empezar a guardar datos de la función en el directorio de funciones.
26 Octubre 2022	Direcciones de memoria	Creamos los rangos que llevarán nuestras direcciones de memoria, creación de la función que asigna la dirección de acuerdo el tipo al scope de la variable/temporal.
	Memoria virtual	Agregamos a la función de asignar_direccion_memoria

		los rangos para las funciones.
	Asignar direcciones	Corrección de errores.
31 Octubre 2022	PilaO y PType Funcionando	Hicimos que las pilas PilaO y PType ya guardaran los datos que les corresponden con los puntos neurálgicos para resolver las operaciones aritméticas, lógicas y relacionales.
01 Noviembre 2022	Inicio de cuádruplos	Agregar los puntos neurálgicos que eran necesarios para guardar información en las pilas y así empezar la generación de cuádruplos. También los cuádruplos ya sustituyen el nombre de la variable por su dirección de memoria. Aquí solo se imprimían los cuádruplos pero no se guardaban.
	Clase cuádruplo creada	Creación de la clase cuádruplo que guardaba los cuádruplos. Se editó en el compiler.py que los cuádruplos ya no se imprimieran y ahora se guardarán en la clase.
	Continuación cuádruplos	Creación de la función que guardaba el cuádruplo de asignación en la clase cuádruplos. Cambios en los puntos neurálgicos de asignación.
	Cuádruplos simples	Llamar al cubo semántico en los puntos neurálgicos de las operaciones aritméticas, relacionales y lógicas para comprobar que las operaciones puedan llevarse a cabo. Los cuádruplos de las

		operaciones ya eran guardados con la función de la clase de cuádruplos.
02 Noviembre 2022	Cuádruplo IF	Creación de los puntos neurálgicos gotoMain_np que realiza el primer cuádruplo “GOTO (main)”. Creación de los puntos neurálgicos y las funciones en la clase cuádruplos necesarios para el IF.
03 Noviembre 2022	While, read, write cuádruplos	Generación de cuádruplos de las instrucciones while, read, write, if/else.
	Fill main	El cuádruplo de main ya era rellenado con la línea en donde empieza el main.
04 Noviembre 2022	Ciclo for y asignación de dir virtuales constantes	Generación de cuádruplos del for, asignar dirección a los cuádruplos del for. Creación de la clase constantes que guarda y asigna la dirección de memoria correspondiente a las constantes.
07 Noviembre 2022	Implementación funciones	Editar el directorio de funciones para agregar la línea en donde comienza la función. Editar la tabla de variables para asociar la variable a la dirección de la función que pertenece.
	Cuádruplos de funciones	Creación de los cuádruplos de las funciones con dirección de memoria.
08 Noviembre 2022	Mapa de memoria y cuádruplos de funciones	Creación del archivo tipo txt que exporta cuádruplos, información del directorio de funciones y tabla de constantes. Creación del cuádruplo END. Contar el

		tamaño que tiene la función.
	Arreglos a cuádruplos de módulos	Errores de cuádruplos de funciones arreglados.
11 Noviembre 2022	Inicio de máquina virtual	Empezar la máquina virtual. Lectura del archivo txt generado en compilación dataVirtualMachine.txt
15 Noviembre 2022	Máquina virtual: memoria ctes, globales y temp	Máquina virtual, ya empieza a leer líneas y a separar los elementos de los cuádruplos para poder guardarlos en los diccionarios correspondientes. Creación del “parche guadalupano” en las funciones.
17 Noviembre 2022	Corrección operaciones aritméticas MV	Cambiar la dirección que se le asignaba a los temporales porque tenían un error de scope.
	Corrección op aritmética MV	Creación de la función convert_type en la MV, para mandar la dirección y convertir al tipo que le corresponde de acuerdo al rango de direcciones. Hacer más pequeño el código que realiza las operaciones aritméticas en la MV.
	Asignación con función	Crear la función search_dict que regresa el valor guardado en esa dirección. Ocupar esa función para hacer más pequeño el código de operaciones en la MV.
	Op Aritm cortas	Reducción del código.
	Read, write, op aritm	Ejecución de operaciones aritméticas, write y read en la MV.
	GOTO, GOTO, for?	Ejecución del if y while en

		MV, inicio de ejecución del for.
	For 100%	Ejecución del for al 100.
	Errors included	Exit() en compilación cuando se encuentre un error.
18 Noviembre 2022	Reset de temporales locales	Creación de otro rango de direcciones únicamente para las funciones y así distinguir más fácil los temporales, variables y parámetros.
	Ejecución parcial de funciones	Inicio de ejecución de las funciones. Aún existen errores.
19 Noviembre 2022	Funciones y recursión, eliminar variables	Implementar recursividad de las funciones en la MV. Eliminar las variables que ya no se utilizan al final de la declaración de la función en el compilador.
20 Noviembre 2022	VM fact	Cuádruplos del ejemplo de factorial.
	VM fact	Hacer la prueba de factorial, compilarla y ejecutarla.
	Fibo, arrays intro	Hacer la prueba de fibonacci, compilarla y ejecutarla. Inicio de ejecución de arreglos.
21 Noviembre 2022	Arrays partially working	Modificar el compilador y la máquina virtual para hacer que la búsqueda de un valor en un arreglo funcione.
22 Noviembre 2022	Bloque clase 100%	Eliminar unas reglas de la gramática de clases.

Párrafo de Reflexión: Citlalli

A lo largo de este semestre, gracias a esta materia, puse en práctica varios de mis conocimientos adquiridos a lo largo de lo que llevo estudiando ITC. Fue todo un reto,

tanto personal como de equipo, pero aunque lo sufrí y realmente no sé cómo nos va a ir en cuanto a la calificación, me voy contenta de esta clase y este proyecto porque mis habilidades de programación siento que mejoraron bastante. Pude aprender también cómo es que funciona un lenguaje de programación, desde la parte del léxico y sintaxis, la generación de código intermedio, hasta cómo es que funciona una máquina virtual. Definitivamente aprendí los conceptos de compilación porque los puse en práctica.

Párrafo de Reflexión: Antonio

Personalmente este proyecto me ayudó a comprender el funcionamiento de un compilador y el proceso general que sigue tanto para compilar un programa como para ejecutarlo. Los aspectos principales que se tienen que tomar en cuenta para la realización del mismo. Igualmente conocí mucho más sobre cómo programar en Python, y la manera en la que funcionan los analizadores léxicos y de sintaxis. Por último, al ser un proyecto que engloba una gran cantidad de conceptos vistos y aprendidos durante la carrera, me ayudó mucho a repasar estos mismos conceptos y ponerlos en práctica.

Descripción del Lenguaje

Nombre del Lenguaje

Nuestro lenguaje de programación lleva el nombre **ABC**. Este surgió al inicio de juntar las iniciales de nuestros nombres A+C. Después notamos que si le agregamos la letra b en el medio es como las primeras letras del alfabeto, y pudiera ser una buena analogía ya que como nuestro lenguaje es muy básico solamente podría ser usado para personas que comienzan en el mundo de la programación.

Características del Lenguaje

En el lenguaje ABC es aceptada la declaración y uso de variables de tipo int, float, char y bool. Se pueden hacer operaciones aritméticas, lógicas y relaciones, no obstante, tiene algunas limitaciones nuestro lenguaje ya que deben realizarse de forma jerárquica. De igual manera la declaración y uso de estatutos como if, if/else, while do, for, read y write pueden realizarse. Asimismo, se pueden declarar funciones de tipo int, float, char, bool y void. Para el caso de las funciones void no se regresa nada, mientras que las que son de int, float, char y bool si tienen un return. Las funciones te permiten hacer recursividad lo que facilita el desarrollo de código. Existen también los arreglos, sin embargo, solamente permiten asignar valores e imprimir lo que hay en ese espacio de memoria.

Listado de Errores

En ABC se clasifican los errores en dos etapas: compilación y ejecución.

Compilación

Error	Descripción
Error sintáctico en ‘%s’	Cuando no se estructura o define bien el lenguaje de acuerdo con las reglas gramaticales y de sintaxis.
ERROR: Function undeclared	No se pueden mandar a llamar funciones que no están declaradas.
ERROR: Wrong type of parameter	Avisa cuando mandas como parámetro un tipo diferente al que se usó cuando se definió la función.
ERROR: Wrong quantity of parameters	Avisa cuando mandas un número incorrecto de parámetros a la función llamada.
No se encontró la dirección de la variable	Marca error cuando se realiza un estatuto a una variable no declarada.
No se encontró el type de la variable	Marca error cuando se realiza un estatuto a una variable no declarada.
ERROR: Type mismatch	Marca error cuando se quieren realizar operaciones entre tipos de datos que no son compatibles.
ERROR: Se excedió el máximo de variables	Avisa cuando se han declarado más variables int global de las que soporta el lenguaje.

enteras globales	
ERROR: Se excedió el máximo de variables float globales	Avisa cuando se han declarado más variables float global de las que soporta el lenguaje.
ERROR: Se excedió el máximo de variables char globales	Avisa cuando se han declarado más variables char global de las que soporta el lenguaje.
ERROR: Se excedió el máximo de variables bool globales	Avisa cuando se han declarado más variables bool global de las que soporta el lenguaje.
ERROR: Se excedió el máximo de variables/funciones enteras	Avisa cuando se han declarado más variables int local de las que soporta el lenguaje.
ERROR: Se excedió el máximo de variables/funciones float	Avisa cuando se han declarado más variables float local de las que soporta el lenguaje.
ERROR: Se excedió el máximo de variables/funciones char	Avisa cuando se han declarado más variables char local de las que soporta el lenguaje.
ERROR: Se excedió el máximo de variables/funciones bool	Avisa cuando se han declarado más variables bool local de las que soporta el lenguaje.
ERROR: Se excedió el máximo de variables/clases enteras	Avisa cuando se han declarado más variables int class de las que soporta el lenguaje.
ERROR: Se excedió el máximo de variables/clases flotantes	Avisa cuando se han declarado más variables float class de las que soporta el lenguaje.
ERROR: Se excedió el máximo de variables/clases char	Avisa cuando se han declarado más variables char class de las que soporta el lenguaje.
ERROR: Se excedió el máximo de variables/clases bool	Avisa cuando se han declarado más variables bool class de las que soporta el lenguaje.

ERROR: Se excedió el máximo de constantes int	Avisa cuando se han declarado más constantes int de las que soporta el lenguaje.
ERROR: Se excedió el máximo de constantes float	Avisa cuando se han declarado más constantes float de las que soporta el lenguaje.
ERROR: Se excedió el máximo de constantes char	Avisa cuando se han declarado más constantes char de las que soporta el lenguaje.
ERROR: Se excedió el máximo de funciones int	Avisa cuando se han declarado más funciones int de las que soporta el lenguaje.
ERROR: Se excedió el máximo de funciones float	Avisa cuando se han declarado más funciones float de las que soporta el lenguaje.
ERROR: Se excedió el máximo de funciones char	Avisa cuando se han declarado más funciones char de las que soporta el lenguaje.
ERROR: Se excedió el máximo de funciones bool	Avisa cuando se han declarado más funciones bool de las que soporta el lenguaje.
ERROR: Se excedió el máximo de funciones void	Avisa cuando se han declarado más funciones void de las que soporta el lenguaje.
Archivo no encontrado	Avisa cuando no se encontró el archivo que quiera ser compilado.
The variable {name} already exists	Revisa que el nombre de las variables no se repita.
Variable undeclared	Revisa que no se pueda hacer uso de una variable no declarada.

Ejecución

Error	Descripción
Error: no se encontró el valor de la posición 'address'	Cuando mandas a imprimir una posición vacía del arreglo.

ERROR: out of bounds (‘value_param’) is out of range	Querer asignar un valor, en una posición de un arreglo que es mayor al tamaño del arreglo declarado.
--	---

Descripción del Compilador

Herramientas Utilizadas

Para la creación del compilador de ABC se hizo uso de las siguientes herramientas:

- **Python:** Python es un lenguaje de programación ampliamente utilizado en las aplicaciones web, el desarrollo de software, la ciencia de datos y el machine learning (ML).
- **PLY:** PLY es una implementación de lex y yacc herramientas de análisis para Python.
- **Visual Studio Code:** Visual Studio Code es un editor de código fuente desarrollado por Microsoft para Windows, Linux, macOS y Web.
- **Windows:** Microsoft Windows es el nombre del Sistema Operativo desarrollado por Microsoft para PC de escritorio, servidores y sistemas empujados.
- **MacOS:** MacOS es un sistema operativo diseñado por Apple que está instalado en todos los equipos creados por la compañía Apple Inc., y son conocidos generalmente como Mac.

Descripción del Análisis Léxico

Patrones de Construcción

Las expresiones regulares son patrones que se utilizan para hacer coincidir combinaciones de caracteres en cadenas. En nuestro lenguaje de programación tenemos las siguientes:

- ID: [a-zA-Z_][a-zA-Z_0-9]*
- CTECH: [a-zA-Z_][a-zA-Z_0-9]?
- CTEF: [0-9]+\.[0-9]+
- CTEI: [0-9]+
- LETRERO: "[a-zA-Z_][a-zA-Z_0-9]*"
- newline: \n+

Tokens del Lenguaje

Los tokens son los elementos equivalentes a las palabras y signos de puntuación en el lenguaje natural escrito. Para el caso de ABC son:

PUNTOCOMA	;
PARENTESISIZQ	(
PARENTESIDER)
LLAVEIZQ	{
LLAVEDER	}
CORCHETEIZQ	[
CORCHETEDER]
COMA	,
DOSPUNTOS	:
IGUAL	=
LESSTHAN	<
GREATERTHAN	>
MAS	+
MENOS	-
POR	*
DIV	/
PUNTO	.

Palabras Reservadas

Las palabras reservadas tienen un significado especial para realizar ciertas tareas del compilador, las de nuestro lenguaje son:

'program'	'main'	'class'	'var'	'int'
'float'	'char'	'bool'	'func'	'void'
'return'	'public'	'private'	'protected'	'attribute'
'method'	'read'	'write'	'while'	'do'
'for'	'to'	'and'	'or'	'not'
'equal'	'if'	'else'		

Descripción del Análisis de Sintaxis

Gramática Formal

La gramática formal es un conjunto de reglas para reescribir cadenas, junto con un símbolo de inicio a partir del cual comienza la reescritura. La gramática formal definida para nuestro lenguaje es la siguiente:

program:

```
PROGRAM ID PUNTOCOMA main
| PROGRAM ID PUNTOCOMA clase main
| PROGRAM ID PUNTOCOMA clase var main
| PROGRAM ID PUNTOCOMA clase var funcion main
| PROGRAM ID PUNTOCOMA clase funcion main
| PROGRAM ID PUNTOCOMA var main
| PROGRAM ID PUNTOCOMA var funcion main
| PROGRAM ID PUNTOCOMA funcion main
```

main:

```
MAIN PARENTESISIZQ PARENTESISDER LLAVEIZQ LLAVEDER
```

| MAIN PARENTESISIZQ PARENTESISDER LLAVEIZQ estatuto LLAVEDER

clase:

CLASS ID DOSPUNTOS tipo_clase ID LLAVEIZQ bloque_clase LLAVEDER
PUNTOCOMA

| CLASS ID DOSPUNTOS tipo_clase ID LLAVEIZQ bloque_clase LLAVEDER
PUNTOCOMA clase

| CLASS ID LLAVEIZQ bloque_clase LLAVEDER PUNTOCOMA

| CLASS ID LLAVEIZQ bloque_clase LLAVEDER PUNTOCOMA clase

tipo_clase:

PUBLIC

| PROTECTED

| PRIVATE

var:

VAR varp

varp:

tipo_compuesto ID PUNTOCOMA

| tipo_compuesto ID PUNTOCOMA varp

| tipo_simple ID PUNTOCOMA

| tipo_simple ID PUNTOCOMA varp

| tipo_simple ID CORCHETEIZQ CTEI CORCHETEDER PUNTOCOMA

| tipo_simple ID CORCHETEIZQ CTEI CORCHETEDER PUNTOCOMA varp

| tipo_simple ID CORCHETEIZQ CTEI CORCHETEDER CORCHETEIZQ CTEI
CORCHETEDER PUNTOCOMA

| tipo_simple ID CORCHETEIZQ CTEI CORCHETEDER CORCHETEIZQ CTEI
CORCHETEDER PUNTOCOMA varp

tipo_simple:

INT
| FLOAT
| CHAR
| BOOL

tipo_simple_func:

INT
| FLOAT
| CHAR
| BOOL

tipo_compuesto:

ID

funcion:

FUNC tipo_simple_func ID PARENTESISIZQ parametros PARENTESISDER
PUNTOCOMA dec_var cuerpo
| FUNC tipo_simple_func ID PARENTESISIZQ parametros PARENTESISDER
PUNTOCOMA dec_var cuerpo funcion
| FUNC tipo_simple_func ID PARENTESISIZQ parametros PARENTESISDER
PUNTOCOMA cuerpo
| FUNC tipo_simple_func ID PARENTESISIZQ parametros PARENTESISDER
PUNTOCOMA cuerpo funcion
| FUNC VOID ID PARENTESISIZQ parametros PARENTESISDER PUNTOCOMA
dec_var cuerpo_void
| FUNC VOID ID PARENTESISIZQ parametros PARENTESISDER PUNTOCOMA
dec_var cuerpo_void funcion
| FUNC VOID ID PARENTESISIZQ parametros PARENTESISDER PUNTOCOMA
cuerpo_void
| FUNC VOID ID PARENTESISIZQ parametros PARENTESISDER PUNTOCOMA
cuerpo_void funcion

dec_var:

VAR dec_varp

dec_varp:

tipo_simple ID PUNTOCOMA dec_varp

| tipo_simple ID PUNTOCOMA

| tipo_simple ID CORCHETEIZQ CTEI CORCHETEDER PUNTOCOMA dec_varp

| tipo_simple ID CORCHETEIZQ CTEI CORCHETEDER PUNTOCOMA

| tipo_simple ID CORCHETEIZQ CTEI CORCHETEDER CORCHETEIZQ CTEI
CORCHETEDER PUNTOCOMA dec_varp

| tipo_simple ID CORCHETEIZQ CTEI CORCHETEDER CORCHETEIZQ CTEI
CORCHETEDER PUNTOCOMA

parametros:

INT ID

| INT ID COMA parametros

| FLOAT ID

| FLOAT ID COMA parametros

| CHAR ID

| CHAR ID COMA parametros

| BOOL ID

| BOOL ID COMA parametros

cuerpo:

LLAVEIZQ estatuto RETURN exp PUNTOCOMA LLAVEDER

| LLAVEIZQ estatuto LLAVEDER

cuerpo_void:

LLAVEIZQ estatuto LLAVEDER

bloque_clase:

ATTRIBUTE DOSPUNTOS atributo METHOD DOSPUNTOS metodo

atributo:

tipo_clase tipo_simple ID PUNTOCOMA

| tipo_clase tipo_simple ID PUNTOCOMA atributo

metodo:

tipo_clase tipo_simple ID PARENTESISIZQ parametros PARENTESISDER cuerpo

| tipo_clase tipo_simple ID PARENTESISIZQ parametros PARENTESISDER cuerpo
metodo

| tipo_clase VOID ID PARENTESISIZQ parametros PARENTESISDER cuerpo

| tipo_clase VOID ID PARENTESISIZQ parametros PARENTESISDER cuerpo
metodo

estatuto:

asignacion PUNTOCOMA

| asignacion PUNTOCOMA estatuto

| llamada PUNTOCOMA

| llamada PUNTOCOMA estatuto

| lee PUNTOCOMA

| lee PUNTOCOMA estatuto

| escribe PUNTOCOMA

| escribe PUNTOCOMA estatuto

| condicion

| condicion estatuto

| ciclo_w

| ciclo_w estatuto

| ciclo_f

| ciclo_f estatuto

| llamada_metodo PUNTOCOMA estatuto

| llamada_atributo PUNTOCOMA estatuto

asignacion:

variable IGUAL exp

llamada:

ID PARENTESISIZQ llamadap PARENTESISDER

llamadap:

exp

| exp COMA llamadap

lee:

READ PARENTESISIZQ leep PARENTESISDER

leep:

variable

| variable COMA leep

variable:

ID

| ID CORCHETEIZQ exp CORCHETEDER

| ID CORCHETEIZQ exp CORCHETEDER CORCHETEIZQ exp CORCHETEDER

escribe:

WRITE PARENTESISIZQ escribep PARENTESISDER

escribep:

exp

| exp COMA escribep

| LETRERO

| LETRERO COMA escribep

condicion:

IF PARENTESISIZQ exp PARENTESISDER LLAVEIZQ estatuto LLAVEDER
| IF PARENTESISIZQ exp PARENTESISDER LLAVEIZQ estatuto LLAVEDER
ELSE LLAVEIZQ estatuto LLAVEDER

ciclo_w:

WHILE PARENTESISIZQ exp PARENTESISDER DO LLAVEIZQ estatuto
LLAVEDER'

ciclo_f:

FOR variable_for IGUAL exp TO exp DO LLAVEIZQ estatuto LLAVEDER

variable_for:

ID

exp:

t_exp
| t_exp OR exp

t_exp:

g_exp
| g_exp AND t_exp

g_exp:

m_exp
| m_exp EQUAL m_exp
| m_exp NOT m_exp
| m_exp GREATERTHAN m_exp
| m_exp LESSTHAN m_exp

m_exp:

t

| t MAS m_exp

| t MENOS m_exp

t:

f

| f POR t

| f DIV t

f:

PARENTESISIZQ exp PARENTESISDER

| CTEI

| CTEF

| CTECH

| variable

| llamada

| llamada_metodo

| llamada_atributo

llamada_metodo:

ID PUNTO ID PARENTESISIZQ llamada_metodop PARENTESISDER

llamada_metodop:

CTEI

| CTEI COMA llamada_metodop

| CTEF

| CTEF COMA llamada_metodop

| CTECH

| CTECH COMA llamada_metodop

| ID

| ID COMA llamada_metodop

llamada_atributo:

ID PUNTO ID

Descripción de Generación de Código Intermedio y Análisis Semántico

Código Intermedio

El compilador de ABC logra transformar el código que le da el usuario en código intermedio usando el método de cuádruplos. El código intermedio se encuentra en el archivo dataVirtualMachine.txt una vez que termina de compilarse y no cuenta con errores.

Los cuádruplos que se generan tienen la siguiente estructura:

Operador	Argumento1	Argumento2	Temporal
----------	------------	------------	----------

Las direcciones asociadas a los temporales dependen de si son locales o globales.

Códigos de Operación

Un código de operación es la parte de una instrucción en lenguaje máquina que especifica la operación a realizar. Los códigos de operación que pueden encontrarse en los cuádruplos que ABC genera son los siguientes:

GOTO	Te indica a qué número de línea debes moverte, puede ser una anterior o una posterior. La línea está indicada en la última posición del cuádruplo.
GOTOIF	Te indica a qué línea debes moverte si el valor del temporal es falso.
=	Realiza la asignación del argumento 1 al temporal.

-	Realiza la resta del argumento 1 y argumento 2 y lo guarda en el temporal.
+	Realiza la suma del argumento 1 y argumento 2 y lo guarda en el temporal.
*	Realiza la multiplicación del argumento 1 y argumento 2 y lo guarda en el temporal.
/	Realiza la división del argumento 1 y argumento 2 y lo guarda en el temporal.
>	Realiza la comparación de argumento 1 > argumento 2 y lo guarda en el temporal.
<	Realiza la comparación de argumento 1 < argumento 2 y lo guarda en el temporal.
equal	Realiza la comparación de argumento 1 == argumento 2 y lo guarda en el temporal.
not	Realiza la comparación de argumento 1 != argumento 2 y lo guarda en el temporal.
and	Realiza la comparación de argumento 1 && argumento 2 y lo guarda en el temporal.
or	Realiza la comparación de argumento 1 argumento 2 y lo guarda en el temporal.
WRITE	Imprime lo que se le manda en el temporal.
READ	Lee lo que se guarda en el temporal.
ERA	Crea el tamaño de memoria necesario para la función que se va a llamar.
PARAMETER	Manda como parámetro el argumento 1, a la función que previamente se llamó en ERA. La posición de temporal indica el número de parámetro que es.
GOSUB	Te indica en la posición del temporal en que línea empieza la función que se va a mandar a llamar.
RETURN	Regresa el valor que está indicado en la posición del temporal.
ENDFunc	Indica la línea en donde termina la función.

Direcciones Virtuales

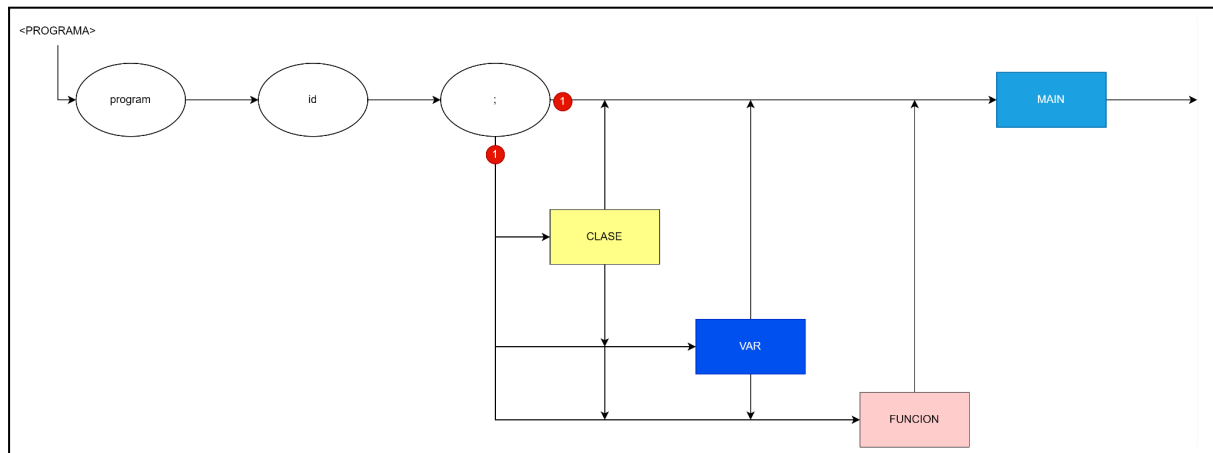
Una dirección virtual no representa la ubicación física real de un objeto en memoria, en cambio, en la memoria es un puntero para un espacio. Las direcciones que ABC asigna están divididas por scope y tipo de dato. Los rangos de direcciones son de la siguiente manera:

- 0: programa
- 1 - 999: variables enteras globales
- 1000 - 1999: variables flotantes globales
- 2000 - 2999: variables char globales
- 3000 - 3999: variables bool globales
- 4000 - 4999: variables enteras locales
- 5000 - 5999: variables flotantes locales
- 6000 - 6999: variables char locales
- 7000 - 7999: variables bool locales
- 8000 - 8999: funciones void
- 9000 - 9999: variables int clase
- 10000 - 10999: variables float clase
- 11000 - 11999: variables char clase
- 12000 - 12999: variables bool clase
- 13000 - 13999: constantes enteras
- 14000 - 14999: constantes flotantes
- 15000 - 15999: constantes char
- 16000 - 16999: funciones int
- 17000 - 17999: funciones float
- 18000 - 18999: funciones char
- 19000 - 19999: funciones bool

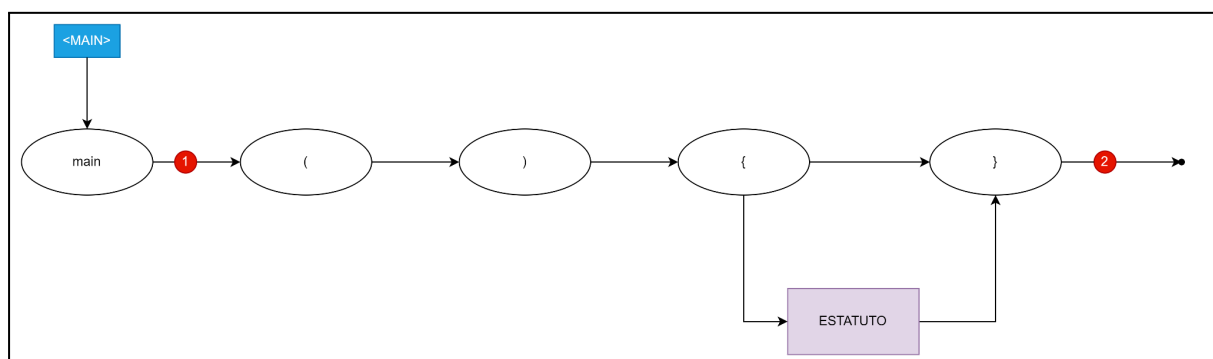
Diagramas de Sintaxis y Acciones Semánticas

Los diagramas sintácticos son una forma de representar una gramática libre de contexto. Los que fueron usados para la implementación de nuestro lenguaje son los

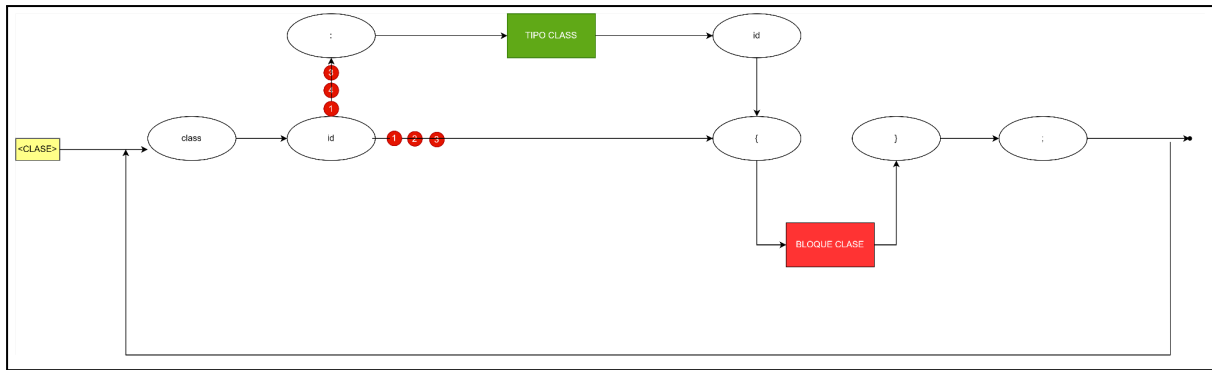
que se mostraran a continuación y se dará una breve explicación de las acciones semánticas.



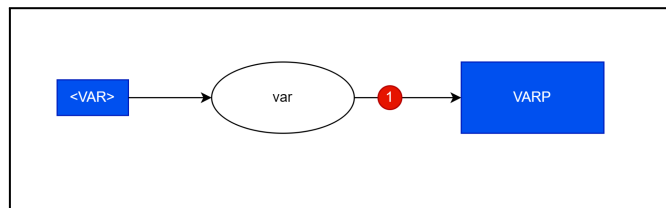
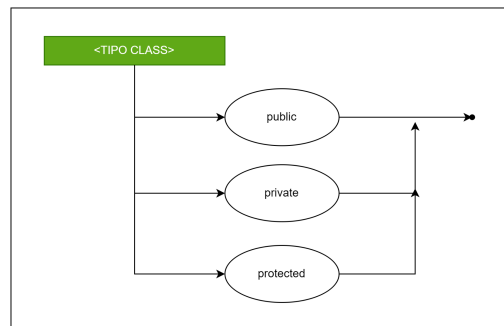
Punto	Descripción
1	Crea el primer cuádruplo necesario, es decir, el GOTO main.



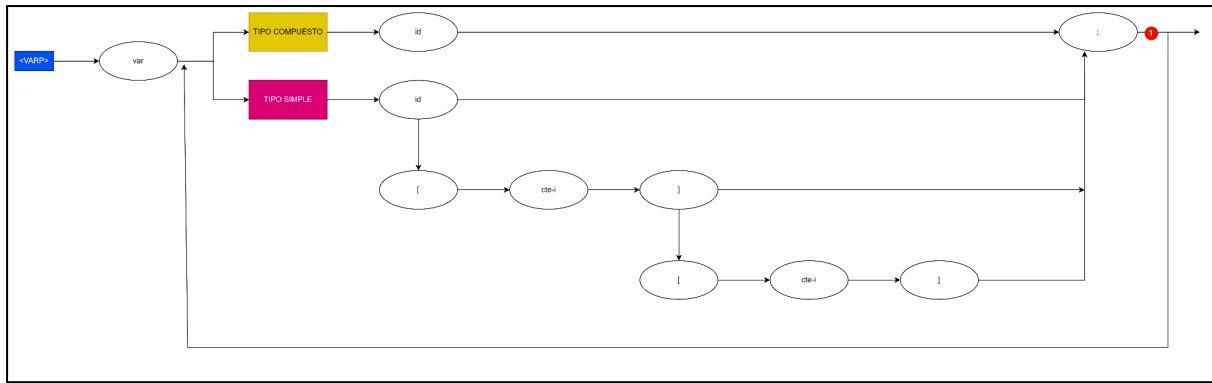
Punto	Descripción
1	Rellena el cuádruplo de main con la línea en donde empieza el main.
2	Crea el último cuádruplo del programa. También escribe en el archivodataVirtualMachine.txt la tabla de constantes, los cuádrupos y el directorio de funciones.



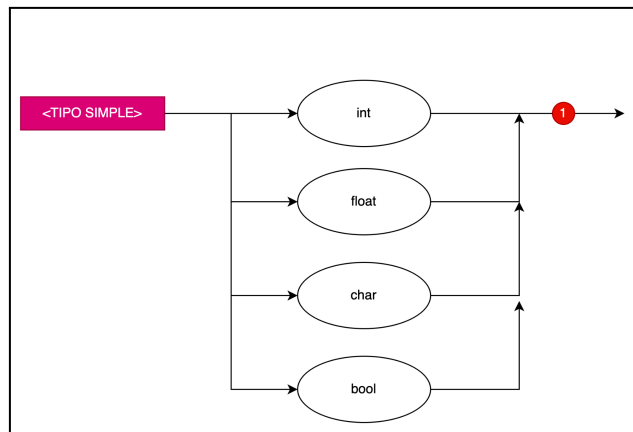
Punto	Descripción
1	Guarda el nombre de la clase
2	Asigna que la clase es de tipo padre
3	Guarda el nombre de la clase y el tipo de clase en la tabla de clases
4	Asigna que la clase es de tipo hijo



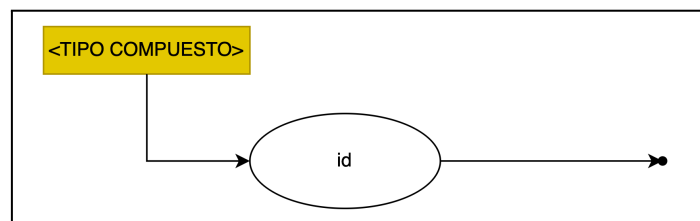
Punto	Descripción
1	Asigna un scope global a esa variable

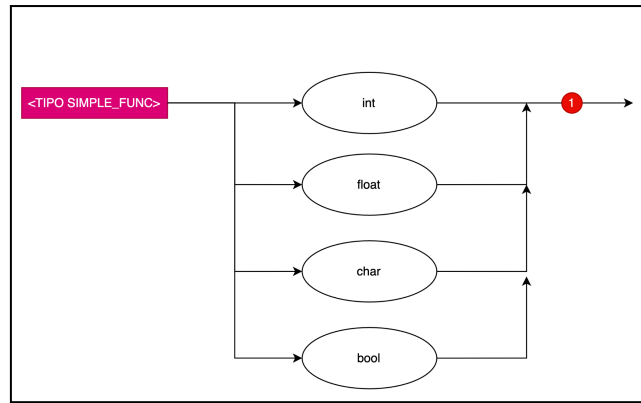


Punto	Descripción
1	Asigna la dirección de memoria a la variable y la guarda en el diccionario correspondiente.

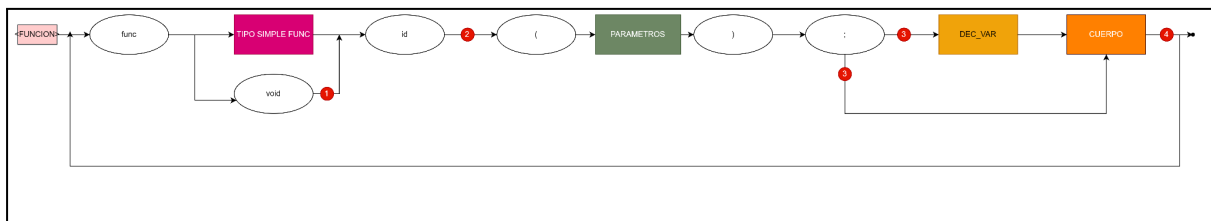


Punto	Descripción
1	Guarda el tipo en la variable 'current_var_type' y le suma uno al contador del tipo correspondiente para tomarlo en cuenta en el tamaño de la función.

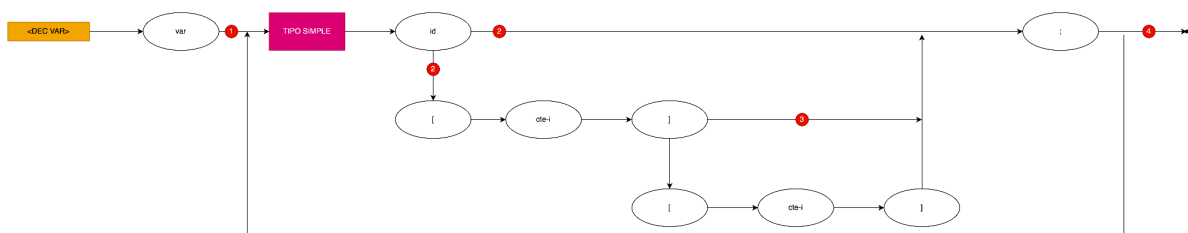




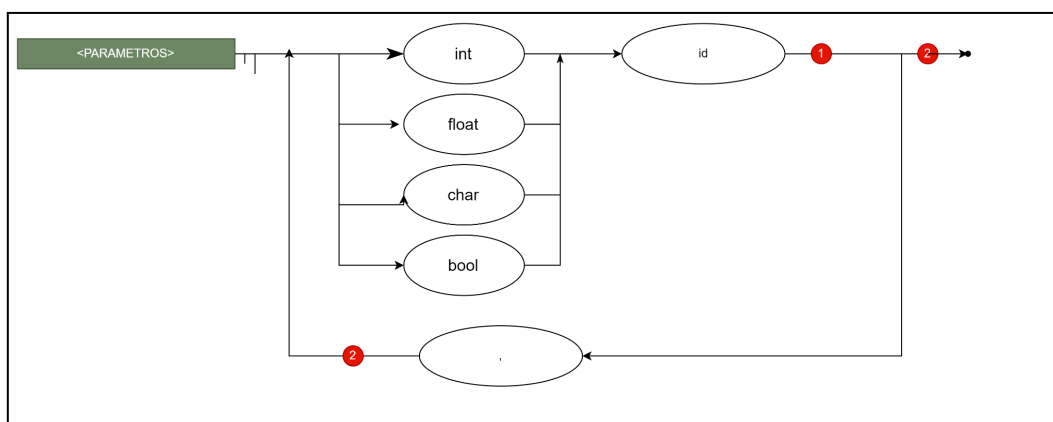
Punto	Descripción
1	Asigna una dirección de memoria a la función dependiendo de su tipo y scope.



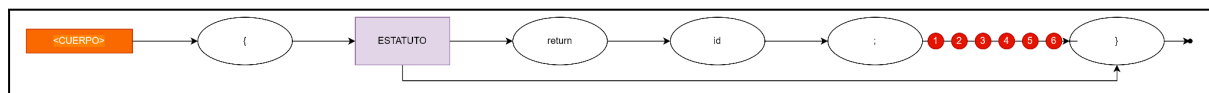
Punto	Descripción
1	Asigna una dirección de memoria a la función dependiendo de su tipo y scope.
2	Se reinician los contadores de los tipos de variables, se guarda el ID de la función y el número de cuádruplo donde inicia.
3	Se agrega la función con todos sus datos a la tabla de funciones. En caso de ser una función de un tipo diferente de void, crea una variable global.
4	Se agrega el tamaño de la función a la tabla de funciones.



Punto	Descripción
1	Asigna un scope global a esa variable
2	Se guarda el id de la variable en una función
3	Se verifica que el arreglo sea de tipo entero y se guarda el tamaño del arreglo en una variable.
4	Se le asigna una dirección de memoria a la variable. En caso de ser un arreglo el apuntador de dirección de memoria se desplaza los números necesarios y se agregan todos los datos a la tabla de variables.

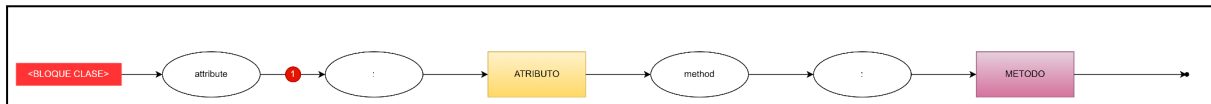


Punto	Descripción
1	Se guarda el id, tipo y scope del parámetro y se le suma uno al contador del tipo correspondiente (para el tamaño de la función).
2	Se le asigna una dirección de memoria al parámetro y se guarda en la tabla de variables con todos sus datos.

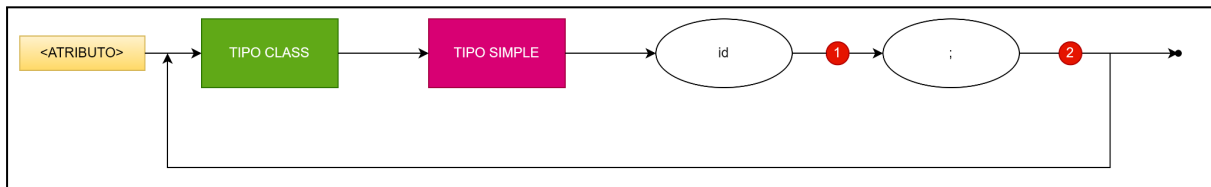


Punto	Descripción
1	Si en el top de la pila hay algún '*' o '/' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
2	Si en el top de la pila hay algún '+' o '-' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
3	Si en el top de la pila hay algún '<', '>', 'equal' o 'not' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.

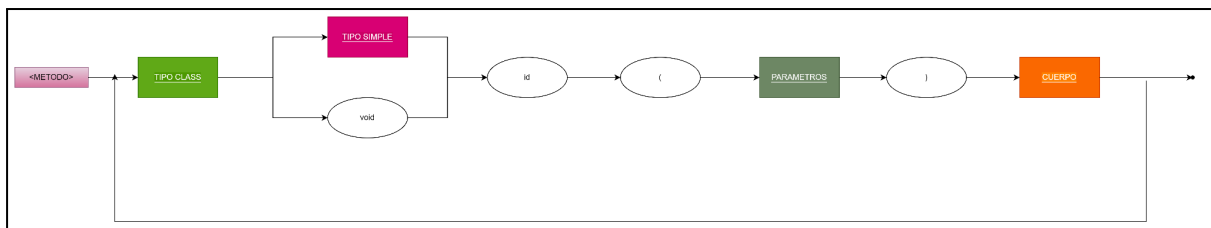
4	Si en el top de la pila hay algún ‘and’ genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
5	Si en el top de la pila hay algún ‘or’ genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
6	Saca de la pila el resultado de la expresión del ‘return’.

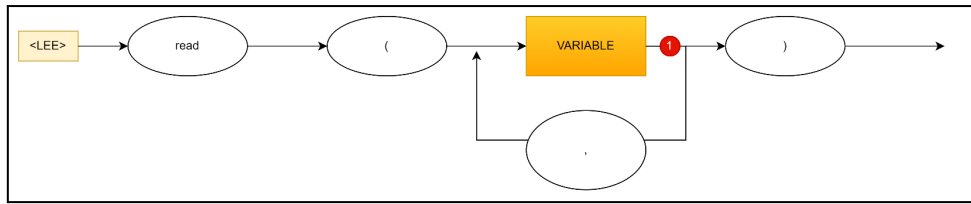


Punto	Descripción
1	Guarda el scope ‘class’ en una variable.

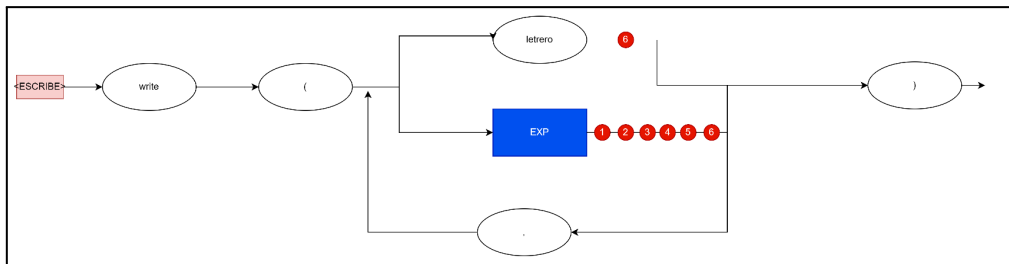


Punto	Descripción
1	Guarda el id del atributo en una variable.
2	Se le asigna una dirección de memoria a la variable. En caso de ser un arreglo el apuntador de dirección de memoria se desplaza los números necesarios y se agregan todos los datos a la tabla de variables.

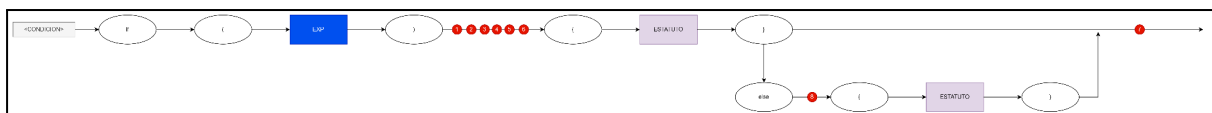




Punto	Descripción
1	Saca la variable de la pila de operandos y genera el cuádruplo READ.

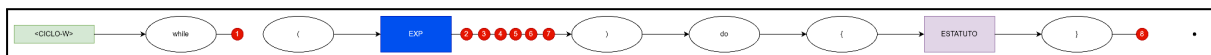


Punto	Descripción
1	Si en el top de la pila hay algún '*' o '/' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
2	Si en el top de la pila hay algún '+' o '-' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
3	Si en el top de la pila hay algún '<', '>', 'equal' o 'not' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
4	Si en el top de la pila hay algún 'and' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
5	Si en el top de la pila hay algún 'or' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
6	Saca la variable de la pila de operandos y genera el cuádruplo WRITE.



Punto	Descripción
1	Si en el top de la pila hay algún '*' o '/' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.

2	Si en el top de la pila hay algún ‘+’ o ‘-’ genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
3	Si en el top de la pila hay algún ‘<’, ‘>’, ‘equal’ o ‘not’ genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
4	Si en el top de la pila hay algún ‘and’ genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
5	Si en el top de la pila hay algún ‘or’ genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
6	Verifica que el resultado de sea de tipo bool, genera el cuádruplo de GOTO del if y guarda el número de cuádruplo en la pila de saltos.
7	Se completa el cuádruplo de if/else con el número de salto correspondiente.
8	Se genera el cuádruplo de GOTO del else, guarda el número de cuádruplo en la pila de saltos y completa el cuádruplo de if con el salto correspondiente.

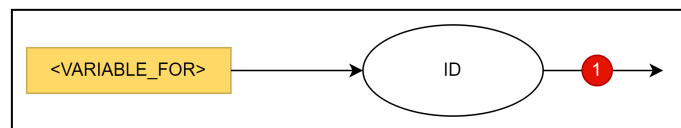


Punto	Descripción
1	Se guarda el número del cuádruplo actual en la pila de saltos.
2	Si en el top de la pila hay algún ‘*’ o ‘/’ genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
3	Si en el top de la pila hay algún ‘+’ o ‘-’ genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
4	Si en el top de la pila hay algún ‘<’, ‘>’, ‘equal’ o ‘not’ genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
5	Si en el top de la pila hay algún ‘and’ genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
6	Si en el top de la pila hay algún ‘or’ genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
7	Verifica que el resultado de la condición del while sea de tipo bool, guarda el resultado en la pila de operandos, genera el cuádruplo GOTO y guarda el número de cuádruplo actual en la pila de saltos.
8	Se agrega el cuádruplo de GOTO del while y se completa el GOTO con el

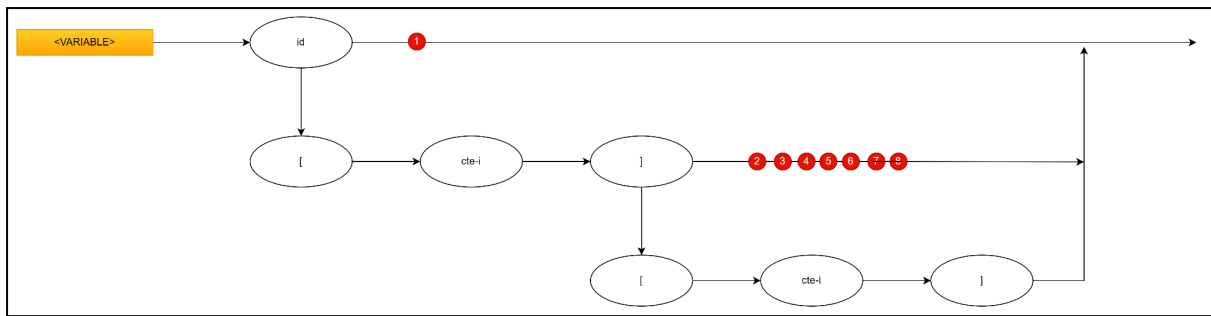
	salto correspondiente.
--	------------------------



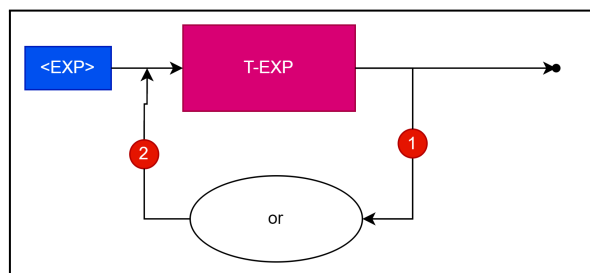
Punto	Descripción
1	Si en el top de la pila hay algún '*' o '/' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
2	Si en el top de la pila hay algún '+' o '-' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
3	Si en el top de la pila hay algún '<', '>', 'equal' o 'not' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
4	Si en el top de la pila hay algún 'and' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
5	Si en el top de la pila hay algún 'or' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
6	Valida que el resultado sea de tipo entero y genera el par de cuádruplos donde igualas el resultado a la variable de control.
7	Valida que el resultado sea de tipo entero y genera el cuádruplo que iguala el resultado a la variables final, el cuádruplo que compara la variable de control con la final y el GOTOF.
8	Genera los cuádruplos de sumarle 1 a la variable de control, el GOTO y llena el GOTOF creado anteriormente.



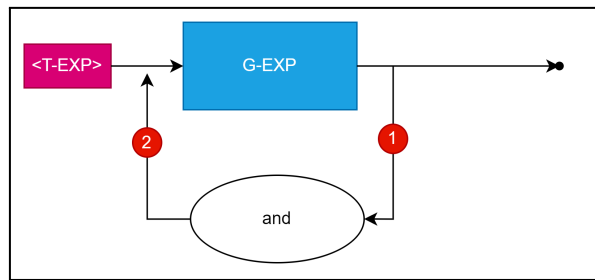
Punto	Descripción
1	Valida que la variable sea de tipo entero.



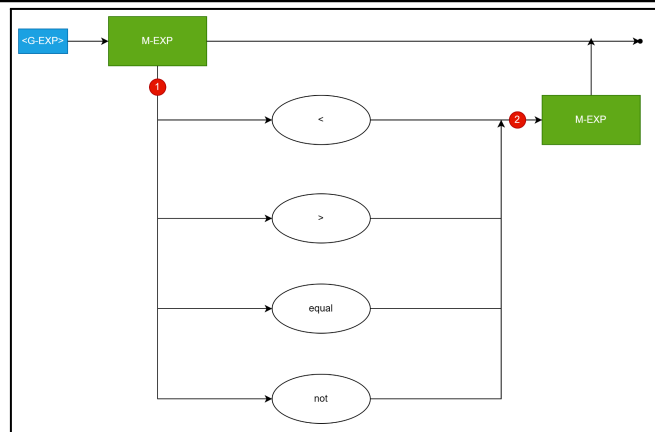
Punto	Descripción
1	Guarda la variable y su tipo en las pilas correspondientes.
2	Si en el top de la pila hay algún '*' o '/' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
3	Si en el top de la pila hay algún '+' o '-' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
4	Si en el top de la pila hay algún '<', '>', 'equal' o 'not' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
5	Si en el top de la pila hay algún 'and' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
6	Si en el top de la pila hay algún 'or' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
7	Si en el top de la pila hay algún '=' genera el cuádruplo correspondiente.
8	Se agregan los cuádruplos relacionados a los arreglos, como la verificación de los límites, la suma de -k y la suma a la dirección virtual del arreglo.



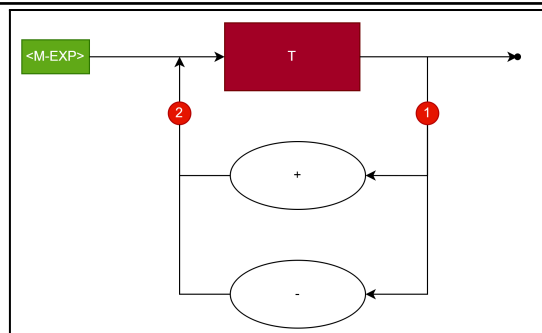
Punto	Descripción
1	Si en el top de la pila hay algún 'or' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
2	Se guarda el operando 'or' en la pila de operandos.



Punto	Descripción
1	Si en el top de la pila hay algún ‘and’ genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
2	Se guarda el operando ‘and’ en la pila de operandos.



Punto	Descripción
1	Si en el top de la pila hay algún ‘<’, ‘>’, ‘equal’ o ‘not’ genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
2	Se guarda el operando ‘<’, ‘>’, ‘equal’ o ‘not’ en la pila de operandos.



Punto	Descripción
1	Si en el top de la pila hay algún ‘+’ o ‘-’ genera el cuádruplo

	correspondiente y agrega el resultado a la pila de operandos.
3	Si en el top de la pila hay algún '<', '>', 'equal' o 'not' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
4	Si en el top de la pila hay algún 'and' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
5	Si en el top de la pila hay algún 'or' genera el cuádruplo correspondiente y agrega el resultado a la pila de operandos.
6	Se agrega la constante 'int' a la tabla de constantes.
7	Se agrega la constante 'float' a la tabla de constantes.
8	Se agrega la constante 'char' a la tabla de constantes.

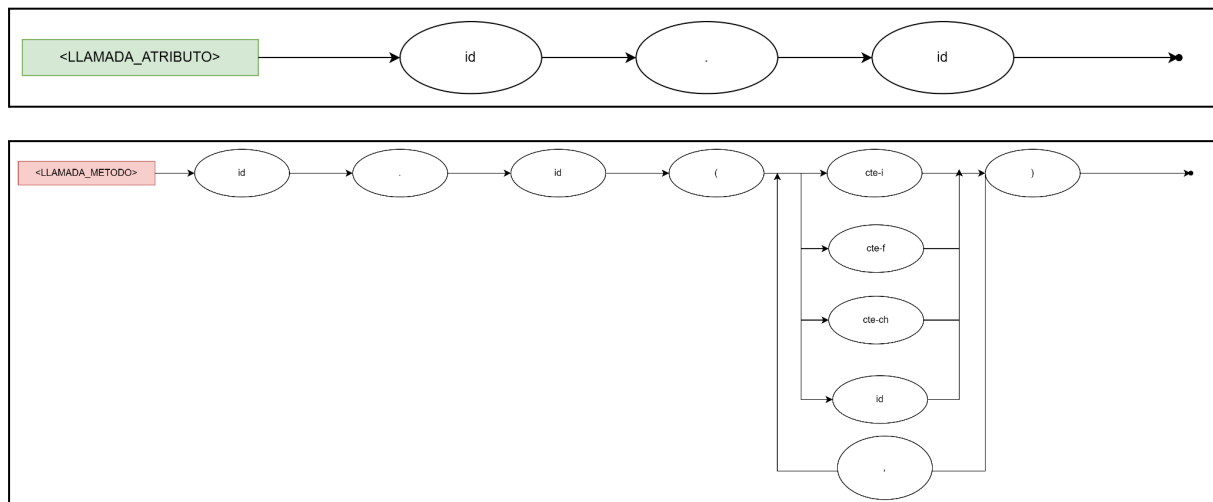


Tabla de Consideraciones Semánticas

La tabla de consideraciones semánticas nos indica qué operaciones están permitidas entre las variables de acuerdo con el tipo de dato que tiene cada una. La del lenguaje ABC es la siguiente:

Primer tipo	Operación	Segundo tipo	Resultado
int	+	int	int
	-		int
	/		float

	*		int
	<		bool
	>		bool
	and		bool
	or		bool
	not		bool
	equal		bool
	=		True
int	+	float	float
	-		float
	/		float
	*		float
	<		bool
	>		bool
	and		bool
	or		bool
	not		bool
	equal		bool
	=		False
int	+	bool	err
	-		err
	/		err
	*		err
	<		err
	>		err
	and		err

	or		err
	not		err
	equal		err
	=		False
int	+	char	err
	-		err
	/		err
	*		err
	<		err
	>		err
	and		err
	or		err
	not		err
	equal		err
	=		False
float	+	int	float
	-		float
	/		float
	*		float
	<		bool
	>		bool
	and		bool
	or		bool
	not		bool
	equal		bool
	=		True

float	+	float	float
	-		float
	/		float
	*		float
	<		bool
	>		bool
	and		bool
	or		bool
	not		bool
	equal		bool
	=		True
float	+	bool	err
	-		err
	/		err
	*		err
	<		err
	>		err
	and		err
	or		err
	not		err
	equal		err
	=		False
float	+	char	err
	-		err
	/		err
	*		err

	<		err
	>		err
	and		err
	or		err
	not		err
	equal		err
	=		False
bool	+	int	err
	-		err
	/		err
	*		err
	<		err
	>		err
	and		err
	or		err
	not		err
	equal		err
	=		False
bool	+	float	err
	-		err
	/		err
	*		err
	<		err
	>		err
	and		err
	or		err

	not		err
	equal		err
	=		False
bool	+	bool	err
	-		err
	/		err
	*		err
	<		err
	>		err
	and		bool
	or		bool
	not		bool
	equal		bool
	=		True
bool	+	char	err
	-		err
	/		err
	*		err
	<		err
	>		err
	and		err
	or		err
	not		err
	equal		err
	=		False
char	+	int	err

	-		err
	/		err
	*		err
	<		err
	>		err
	and		err
	or		err
	not		err
	equal		err
	=		False
char	+	float	err
	-		err
	/		err
	*		err
	<		err
	>		err
	and		err
	or		err
	not		err
	equal		err
	=		False
char	+	bool	err
	-		err
	/		err
	*		err
	<		err

	>		err
	and		err
	or		err
	not		err
	equal		err
	=		False
char	+	char	err
	-		err
	/		err
	*		err
	<		err
	>		err
	and		err
	or		err
	not		bool
	equal		bool
	=		True

Administración de Memoria

Decidimos usar en su mayoría diccionarios ya que nos permitió ordenar de manera más fácil diversos datos. Al darle una 'key' y que cada una de estas tuviera asignado una serie de valores nos facilitó mucho el acceso e inserción de los datos a lo largo del código, lo anterior lo hicimos por ejemplo en la tabla de variables, directorio de funciones, etc.

De igual manera, optamos por usar pilas ya que nos dimos cuenta de que para cierto tipo de datos solo era necesario guardar un valor o una pila asociada a ese dato y las

pilas facilitaban el push y pop de los mismos datos, un ejemplo es en la generación de cuádruplos.

Directorio de funciones: Utilizamos un diccionario en donde su ‘key’ es el nombre de cada función y el contenido de cada ‘key’ es una lista con los datos de la función: tipo, dirección, cuádruplo donde inicia, tamaño (lista de cuatro enteros, un entero por tipo de variable) y parámetros (lista que contenía el tipo de cada parámetro).

Dir.Func = {‘nombre_func’ : [tipo, dir, cuad, [cant_ints, cant_floats, cant_chars, cant_bools], [tipo de cada param]] }

Tabla de variables: Utilizamos un diccionario en donde su ‘key’ es el nombre de cada variable y el contenido de cada ‘key’ es una lista con los datos de la variable: tipo, scope, dirección, dirección de la función a la que está asociada, dimensión en caso de ser arreglo (pila que guarda el límite superior del arreglo).

VarsTable = {‘nombre_var’: [tipo, scope, dir, dir_func, [lim_sup]] }

Tabla de clases: Utilizamos un diccionario en donde su ‘key’ es el nombre de cada clase y el contenido de cada ‘key’ es una lista con el tipo de la clase.

ClassTable = {‘nombre_clase’: [tipo] }

Cubo semántico: Utilizamos un diccionario, el cual, tiene cuatro ‘keys’ una por cada tipo de dato (int, float, char, bool). Dentro de cada una de las cuatro ‘keys’ existen cuatro diccionarios y cada ‘key’ hace referencia a un tipo de dato con el propósito de compararlos, y dentro de cada una de estas cuatro ‘keys’ existen 11 diccionarios donde hay una ‘key’ por cada tipo de operador (+, -, *, /, etc.). A su vez cada una de estas 11 ‘keys’ contiene el resultado de aplicarle el operador a los dos tipos de datos, como: int, float, bool, True, False o error.

SemanticCube = {tipo : { tipo : { operador : resultado } } }

Lista de cuádruplos: Utilizamos una pila, la cual, contiene a su vez una lista dividida en cuatro que contiene la información de cada cuádruplo, como el operador, operando izquierdo, operando derecho y resultado. Aunque cada cuádruplo puede estar estructurado de manera diferente se manejaron los mismos nombres para identificar cada uno de los cuatro datos del cuádruplo.

QuadruplesList = [operator, left_oper, right_oper, result]

Tabla de constantes: Utilizamos un diccionario en donde su 'key' es la dirección virtual de la constante y el contenido de cada 'key' es una lista que incluye el valor de la constante y su tipo.

CtesTable = { direccion : [value, type] }

Operandos, operadores, tipos de variables, saltos de cuádruplos, parámetros de funciones, tamaño de la función: Todos estos datos se guardan en su respectiva pila.

Pila = [dato#1, dato#2, ... , dato#n]

Descripción de la Máquina Virtual

Herramientas Utilizadas

Para la creación de la máquina virtual de ABC se hizo uso de las siguientes herramientas:

- **Python:** Python es un lenguaje de programación ampliamente utilizado en las aplicaciones web, el desarrollo de software, la ciencia de datos y el machine learning (ML).
- **Linecache:** El módulo linecache permite obtener cualquier línea de un archivo fuente Python, mientras se intenta optimizar internamente, usando una caché, el caso común en el que se leen muchas líneas de un solo archivo.
- **Visual Studio Code:** Visual Studio Code es un editor de código fuente desarrollado por Microsoft para Windows, Linux, macOS y Web.
- **Windows:** Microsoft Windows es el nombre del Sistema Operativo desarrollado por Microsoft para PC de escritorio, servidores y sistemas empujados.
- **MacOS:** MacOS es un sistema operativo diseñado por Apple que está instalado en todos los equipos creados por la compañía Apple Inc., y son conocidos generalmente como Mac.

Administración de Memoria

Decidimos usar diccionarios ya que nos permitió ordenar de manera más fácil diversos datos. Al darle una 'key' y que cada una de estas llaves tuviera asignado una serie de valores nos facilitó mucho el acceso e inserción de los datos a lo largo del código, es decir nos ayudó a la manipulación de los datos.

De igual manera, optamos por usar pilas ya que nos dimos cuenta de que para cierto tipo de datos solo era necesario guardar un valor o una pila asociada a ese dato y las pilas facilitaban el push y pop de los mismos datos, lo cual, a su vez también facilitó el saber el último dato insertado, aspecto muy ventajoso para la recursión de funciones y poder acceder al valor correcto.

Constantes: Decidimos usar un diccionario donde su llave es la dirección virtual de cada constante y contienen su valor y su tipo. Esto nos facilitó el acceso a las constantes y sus datos.

Dict_Ctes = {dir_vir : [valor, tipo] }

Funciones: Decidimos usar un diccionario donde su llave es la dirección virtual de cada función y contienen su nombre, tipo de la función en caso de ser void o la dirección virtual de la variable global que guarda el resultado de la función, el cuádruplo donde inicia, su tamaño y los tipos de los parámetros. Esto nos facilitó el acceso a los datos de las funciones.

Dict_Funcs = {dir_vir : [id, tipo/dir_vir, cuad, ints, floats, chars, bools, type_params] }

Memoria global: Decidimos usar un diccionario donde su llave es la dirección virtual de cada variable global y contienen su valor. Esto nos ayudó a acceder a los valores de cada una de las variables manipuladas en el main.

Mem_global = {dir_vir : valor }

Memoria local: Decidimos usar una pila, ya que nos dimos cuenta que al momento de hacer recursividad de funciones era mucho más fácil hacerle ‘pop’ a la última función, la cual, será la primera en ser destruida al momento de acabar su ejecución. Dentro de esta pila se agregan más que nada diccionarios (‘dict_local’), los cuales tienen como primera ‘key’ el nombre de la función y contiene un contador que hace referencia al número de función, en caso de ser recursiva. Seguido de esta ‘pareja’ de datos se encuentran todas las variables locales que se utilizan en la función. Inicialmente solo se crean los ‘espacios de memoria’ sin tener algún valor y conforme se ejecutan los estatutos de la función se le va dando los valores a estas variables.

Mem_local = [{ nombre_func : cont, var_local : valor }]

Pilas de funciones y retornos (migajitas de pan): Decidimos usar pilas para guardar el nombre de las funciones que se iban ejecutando y saber que funciones ‘mandar a

dormir' y a que función necesitábamos acceder a sus valores. Del mismo modo se hizo uso de una pila para guardar los valores de retorno, a los cuales nos necesitamos mover cada vez que se acababa la ejecución de una función.

Pile_funcs = [nombre_funcion#1, nombre_funcion#2, ... nombre_funcion#n]

Pile_returns = [#quad1, ... #quadrn]

Pruebas

Al realizar pruebas podemos verificar y comprobar que el lenguaje funciona de manera correcta.

Factorial Cíclico

Código

```
program Compilador;

var
  int i;
  int fact;

main() {
  read(i);
  if (i equal 0) {
    write(1);
  }
  else {
    fact = 1;
    while (i > 1) do {
      fact = fact * i;
      i = i - 1;
    }
    write(fact);
  }
}
```

Compilación

```
#
13000,0,int
13001,1,int
13002,1,int
13003,1,int
13004,1,int
#
1,GOTO,-1,-1,2
2,READ,-1,-1,1
3,equal,1,13000,3000
4,GOTOF,3000,-1,7
5,WRITE,-1,-1,13001
6,GOTO,-1,-1,16
7,=,13002,-1,2
8,>,1,13003,3001
9,GOTOF,3001,-1,15
10,*,2,1,3
11,=,3,-1,2
12,-,1,13004,4
13,=,4,-1,1
14,GOTO,-1,-1,8
15,WRITE,-1,-1,2
16,END,-1,-1,-1
#
```

Ejecución

```
PS C:\Users\1110077013\Documents\Diseño de compiladores\Compilador> & C:/Users/1110077013/AppData/Local/Microsoft/windowsApps/python3.10.exe "c:/Users/1110077013/Documents/Diseño de compiladores/Compilador/MaquinaVirtual.py"
4
24
```

Factorial Recursivo

Código

```
program Compilador;

var
  int i;
  int f;

func int factorial (int n);
var
  int resultado;
{
  if (n equal 1)
  {
    resultado = 1;
  }

  else
  {
    resultado = n * factorial(n - 1);
  }

  return resultado;
}

main() {
  read(i);
  f = factorial(i);
  write(f);
}
```

Compilación

```
16000,factorial,3,2,5,0,0,1,int,
#
13000,1,int
13001,1,int
13002,1,int
#
1,GOTO,-1,-1,15
2,equal,4000,13000,7000
3,GOTO,7000,-1,6
4,=,13001,-1,4001
5,GOTO,-1,-1,13
6,ERA,factorial,-1,-1
7,-,4000,13002,4002
8,PARAMETER,4002,-1,1
9,GOSUB,factorial,-1,2
10,=,3,-1,4
11,*,4000,4,4003
12,=,4003,-1,4001
13,RETURN,-1,-1,4001
14,ENDFunc,-1,-1,-1
15,READ,-1,-1,1
16,ERA,factorial,-1,-1
17,PARAMETER,1,-1,1
18,GOSUB,factorial,-1,2
19,=,3,-1,5
20,=,5,-1,2
21,WRITE,-1,-1,2
22,END,-1,-1,-1
#
```

Ejecución

```
PS C:\Users\1110077013\Documents\Diseño de compiladores\Compilador> & C:/Users/1110077013/AppData/Local/Microsoft/windowsApps/python3.
10.exe "c:/Users/1110077013/Documents/Diseño de compiladores/Compilador/MaquinaVirtual.py"
4
24
```

Fibonacci Cíclico

Código

```

program Compilador;

var
  int i;
  int a;
  int x1;
  int y1;
  int total;

main() {

  read(a);
  x1 = 1;
  y1 = 1;

  if (a > 2)
  {
    write(0);
    write(x1);
    write(y1);
    for i = 0 to a-2 do
    {
      total = x1 + y1;
      y1 = x1;
      x1 = total;
      write(total);
    }
  }
}

```

Compilación

```

#
13000,1,int
13001,1,int
13002,2,int
13003,0,int
13004,0,int
13005,2,int
#
1,GOTO,-1,-1,2
2,READ,-1,-1,2
3,=,13000,-1,3
4,=,13001,-1,4
5,>,2,13002,3000
6,GOTO,3000,-1,25
7,WRITE,-1,-1,13003
8,WRITE,-1,-1,3
9,WRITE,-1,-1,4
10,=,13004,-1,1
11,=,1,-1,VControl
12,-,2,13005,6
13,=,6,-1,VFinal
14,<,VControl,VFinal,3001
15,GOTO,3001,-1,25
16,+,3,4,7
17,=,7,-1,5
18,=,3,-1,4
19,=,5,-1,3
20,WRITE,-1,-1,5
21,+,VControl,1,8
22,=,8,-1,VControl
23,=,8,-1,1
24,GOTO,-1,-1,14
25,END,-1,-1,-1
#

```

Ejecución

```
PS C:\Users\1110077013\Documents\Diseño de compiladores\Compilador> & C:/Users/1110077013/AppData/Local/Microsoft/WindowsApps/python3.10.exe "c:/Users/1110077013/Documents/Diseño de compiladores/Compilador/MaquinaVirtual.py"
5
0
1
1
2
3
5
```

Fibonacci Recursivo

Código

```
program Compilador;

var
    int a;
    int j;

func int fibo (int n);
var
    int resultado;
    int res1;
    int res2;
{
    if (n < 2) {
        resultado = n;
    }
    else {
        res1 = fibo(n - 1);
        res2 = fibo(n - 2);
        resultado = res1 + res2;
    }
    return resultado;
}

main() {
    read(a);
    for j = 0 to a + 1 do {
        write(fibo(j));
    }
}
```

Compilación

```

16000,fibo,3,2,9,0,0,1,int,
#
13000,2,int
13001,1,int
13002,2,int
13003,0,int
13004,1,int
#
1,GOTO,-1,-1,22
2,<,4000,13000,7000
3,GOTO,7000,-1,6
4,=,4000,-1,4001
5,GOTO,-1,-1,20
6,ERA,fibo,-1,-1
7,-,4000,13001,4004
8,PARAMETER,4004,-1,1
9,GOSUB,fibo,-1,2
10,=,3,-1,4
11,=,4,-1,4002
12,ERA,fibo,-1,-1
13,-,4000,13002,4005
14,PARAMETER,4005,-1,1
15,GOSUB,fibo,-1,2
16,=,3,-1,5
17,=,5,-1,4003
18,+,4002,4003,4006
19,=,4006,-1,4001
20,RETURN,-1,-1,4001
21,ENDFunc,-1,-1,-1

```

```

20,RETURN,-1,-1,4001
21,ENDFunc,-1,-1,-1
22,READ,-1,-1,1
23,=,13003,-1,2
24,=,2,-1,VControl
25,+,1,13004,6
26,=,6,-1,VFinal
27,<,VControl,VFinal,3000
28,GOTO,3000,-1,38
29,ERA,fibo,-1,-1
30,PARAMETER,2,-1,1
31,GOSUB,fibo,-1,2
32,=,3,-1,7
33,WRITE,-1,-1,7
34,+,VControl,1,8
35,=,8,-1,VControl
36,=,8,-1,2
37,GOTO,-1,-1,27
38,END,-1,-1,-1
#

```

Ejecución

```

PS C:\Users\1110077013\Documents\Diseño de compiladores\Compilador> & C:/Users/1110077013/AppData/Local/Microsoft/WindowsApps/python3.
10.exe "c:/Users/1110077013/Documents/Diseño de compiladores/Compilador/MaquinaVirtual.py"
5
0
1
1
2
3
5

```

Acceder a un elemento que está fuera del rango del arreglo: Error

Código

```

program Compilador;

var
    int a;
    int b;
    int arra[2];
    int arrab[5];
    int arrac[10];

main()
{
    a = 2;
    b = 3;
    arrab[2+3] = 4 + a * b;
    arrab[2] = 9;
    write(arra[3]);
}

```

Compilación

```
#
13000,2,int
13001,3,int
13002,2,int
13003,3,int
13004,4,int
13005,2,int
13006,9,int
13007,3,int
#
1,GOTO,-1,-1,2
2,=,13000,-1,1
3,=,13001,-1,2
4,+,13002,13003,20
5,VER,20,-1,5
6,+,20,-1,21
7,+,21,5,20000
8,*,1,2,22
9,+,13004,22,23
10,=,23,-1,20000
11,VER,13005,-1,5
12,+,13005,-1,24
13,+,24,5,20001
14,=,13006,-1,20001
15,VER,13007,-1,2
16,+,13007,-1,25
17,+,25,3,20002
18,WRITE,-1,-1,20002
19,END,-1,-1,-1
#
```

Ejecución

```
0.exe "c:/Users/1110077013/Documents/Diseño de compiladores/Compilador/MaquinaVirtual.py"
ERROR: out of bounds (3 is out of range 1...2)
```

Acceder a un elemento que está dentro del rango del arreglo y tiene valor

Código

```
program Compilador;

var
    int a;
    int b;
    int arra[2];
    int arrab[5];
    int arrac[10];

main()
{
    a = 2;
    b = 3;
    arrab[2+3] = 4 + a * b;
    arrab[2] = 9;
    write(arrab[5]);
}
```

Compilación


```
#
13000,2,int
13001,3,int
13002,2,int
13003,3,int
13004,4,int
13005,2,int
13006,9,int
13007,5,int
#
1,GOTO,-1,-1,2
2,=,13000,-1,1
3,=,13001,-1,2
4,+,13002,13003,20
5,VER,20,-1,5
6,+,20,-1,21
7,+,21,5,20000
8,*,1,2,22
9,+,13004,22,23
10,=,23,-1,20000
11,VER,13005,-1,5
12,+,13005,-1,24
13,+,24,5,20001
14,=,13006,-1,20001
15,VER,13007,-1,5
16,+,13007,-1,25
17,+,25,5,20002
18,WRITE,-1,-1,20002
19,END,-1,-1,-1
#
```

Ejecución

```
0.exe "c:/Users/1110077013/Documents/Diseño de compiladores/Compilador/MaquinaVirtual.py"
10
```

Documentación del Código del Proyecto

Estas funciones se encargan de guardar todos los datos de las clases (nombre y tipo) y al final se agregan a la tabla de clases.

```
##### Guarda datos de la clase #####
def p_getnameClass_np(p):
    '''getnameClass_np : empty'''
    global current_name_class
    current_name_class = p[-1]

def p_geFatherClass_np(p):
    '''getFatherClass_np : empty'''
    global current_class_type
    current_class_type = 'father'

def p_getSonClass_np(p):
    '''getSonClass_np : empty'''
    global current_class_type
    current_class_type = 'son'

def p_saveClass_np(p):
    '''saveClass_np : empty'''
    claseTable.add(current_name_class, current_class_type)
```

Funciones que guardan los datos de las variables (id, tipo, dirección, scope) y las guarda en la tabla de variables.

```
##### Guarda datos de las variables#####
def p_getID_np(p):
    '''getID_np : empty'''
    global current_var_id
    current_var_id = p[-1]

def p_getType_np(p):
    '''getType_np : empty'''
    global current_var_type, cont_int, cont_float, cont_char, cont_bool
    current_var_type = p[-1]
    if (p[-1] == 'int'):
        cont_int += 1
    elif (p[-1] == 'float'):
        cont_float += 1
    elif (p[-1] == 'char'):
        cont_char += 1
    elif (p[-1] == 'bool'):
        cont_bool += 1

def p_saveVar_np(p):
    '''saveVar_np : empty'''
    global address_func, is_array, tam_array, dir_global_int, dir_funcion_int, dir_global_float, dir_funcion_float, dir_global_char, dir_funcion_char
    address = asignar_direccion_memoria()
    if is_array == True:
        if current_var_type == 'int':
            if current_var_scope == 'global':
                dir_global_int += tam_array - 1
            elif current_var_scope == 'funcion':
                dir_funcion_int += tam_array - 1
        elif current_var_type == 'float':
            if current_var_scope == 'global':
                dir_global_float += tam_array - 1
            elif current_var_scope == 'funcion':
                dir_funcion_float += tam_array - 1
        elif current_var_type == 'char':
            if current_var_scope == 'global':
                dir_global_char += tam_array - 1
            elif current_var_scope == 'funcion':
                dir_funcion_char += tam_array - 1
    pile_dim.append(tam_array)
    varsTable.add(current_var_id, current_var_type, current_var_scope, address, address_func)
    pile_dim = []
    is_array = False
```

Funciones que guardan los datos de las constantes y las agregan a la tabla de constantes.

```
def p_saveConstantInt_np(p):
    '''saveConstantInt_np : empty'''
    global cte_type
    cte_type = 'int'
    address = asignar_direccion_memoriaCtes()
    Pila0.append(address)
    PilaTipos.append(cte_type)
    constantsTable.add(p[-1], cte_type, address)

def p_saveConstantFloat_np(p):
    '''saveConstantFloat_np : empty'''
    global cte_type
    cte_type = 'float'
    address = asignar_direccion_memoriaCtes()
    Pila0.append(address)
    PilaTipos.append(cte_type)
    constantsTable.add(p[-1], cte_type, address)

def p_saveConstantChar_np(p):
    '''saveConstantChar_np : empty'''
    global cte_type
    cte_type = 'char'
    address = asignar_direccion_memoriaCtes()
    Pila0.append(address)
    PilaTipos.append(cte_type)
    constantsTable.add(p[-1], cte_type, address)
```

Función que genera los cuádruplos de los arreglos.

```

def p_quadsArray_np(p):
    '''quadsArray_np : empty'''
    global current_var_scope
    id_array = p[-10]
    limit = varsTable.size_array(id_array)
    cuadruplo.addQuadruple('VER', Pila0[-1], -1, limit)
    temp = asignar_direccion_memoria()
    cuadruplo.addQuadruple('+', Pila0[-1], -1, temp)
    Pila0.pop()
    PilaTipos.pop()
    if main == False:
        current_var_scope = 'funcion'
    else:
        current_var_scope = 'global'
    temp_pointer = asignar_direccion_temp_pointers()
    cuadruplo.addQuadruple('+', temp, varsTable.find_address(id_array), temp_pointer)
    Pila0.append(temp_pointer)
    PilaTipos.append('int')

```

Funciones que guardan los datos de las funciones en la tabla de funciones y generan sus cuádruplos respectivos.

```

#####Guarda datos de las funciones y genera cuádruplos generados de las funciones#####
def p_getIDFunc_np(p):
    '''getIDFunc_np : empty'''
    global current_func_id, parameters_list, cont_char, cont_int, cont_float, cont_bool, inicio_cuadruplo
    cont_int = 0
    cont_float = 0
    cont_char = 0
    cont_bool = 0
    current_func_id = str(p[-1])
    inicio_cuadruplo = cuadruplo.cont

def p_getTypeFunc_np(p):
    '''getTypeFunc_np : empty'''
    global current_func_type, address_func, current_var_scope, current_var_type
    current_func_type = str(p[-1])
    current_var_type = current_func_type
    current_var_scope = 'funcion'
    address_func = asignar_direccion_funciones()

def p_getParameters_np(p):
    '''getParameters_np : empty'''
    global current_var_id, current_var_type, current_var_scope, cont_int, cont_float, cont_char, cont_bool
    current_var_id = p[-1]
    current_var_type = p[-2]
    current_var_scope = 'funcion'
    parameters_list.append(p[-2])
    if (p[-2] == 'int'):
        cont_int += 1
    elif (p[-2] == 'float'):
        cont_float += 1
    elif (p[-2] == 'char'):
        cont_char += 1
    elif (p[-2] == 'bool'):
        cont_bool += 1

```

```

def p_saveParameter_np(p):
    '''saveParameter_np : empty'''
    global parameters_list, address_func
    address = asignar_direccion_memoria()
    varsTable.add(current_var_id, current_var_type, current_var_scope, address, address_func, [])

def p_saveFuncSign_np(p):
    '''saveFuncSign_np : empty'''
    global parameters_list, current_func_type, current_func_id, tam_func, current_var_type, current_var_scope
    functionsTable.add(current_func_id, current_func_type, address_func, inicio_cuadruplo, tam_func, parameters_list)
    if current_func_type != 'void':
        current_var_type = current_func_type
        current_var_scope = 'global'
        address_var = asignar_direccion_memoria()
        varsTable.add(current_func_id, current_func_type, current_var_scope, address_var, 0, [])
        functionsTable.fill_address(current_func_id, address_var)

def p_saveFunc_np(p):
    '''saveFunc_np : empty'''
    global parameters_list, current_var_type, current_var_scope, address_func, inicio_cuadruplo, cont_int, cont_float, cont_char, cont_bool
    tam_func.append(cont_int)
    tam_func.append(cont_float)
    tam_func.append(cont_char)
    tam_func.append(cont_bool)
    functionsTable.fillTam(current_func_id, tam_func)

```

Clase para la tabla de funciones que recibe como parámetros los datos relacionados a la función, crea un diccionario con el nombre de la función como 'key' y agrega una pila con sus datos. Así mismo existen diferentes funciones dentro de la misma clase con tareas específicas como llenar el diccionario con la dirección de memoria o el tamaño, o buscar una función específica y regresa algún dato en particular.

```
class funcTable:
    def __init__(self):
        self.table = {}

    def add(self, name, type, address, inicio_cuad, tam, parameters):
        currentFunc = Function(type, address, inicio_cuad, tam, parameters)
        if name in self.table:
            print(f"The function {name} is already declared")
        else:
            self.table[name] = currentFunc
            #print(f"Function {name} saved successfully")

    def fill_address(self, name, address):
        if name in self.table:
            self.table[name].type = address

    def fillTam(self, name, tam):
        if name in self.table:
            self.table[name].tam = tam

    def search(self, name):
        if name in self.table:
            return self.table[name]
        else:
            return "Function undeclared"
```

Clase para la lista de cuádruplos que contiene diversas funciones que reciben diferente cantidad de parámetros para ir insertando un cuádruplo de algún tipo en específico.

```
class quadruplesList:
    def __init__(self):
        self.quadsList = []
        self.cont = 1

    def addQuadruple(self, operador, left_operand, right_operand, result):
        if operador != '=':
            cuadruplo = quadruple(self.cont, operador, left_operand, right_operand, result)
            self.quadsList.append(cuadruplo)
            self.cont += 1
            #print("contador", self.cont)

    def addQuadrupleIgual(self, operador, left_operand, right_operand):
            cuadruplo = quadruple(self.cont, operador, right_operand, -1, left_operand)
            self.quadsList.append(cuadruplo)
            self.cont += 1
            #print(cuadruplo.printQuad())

    def addGotoMain(self):
            cuadruplo = quadruple(self.cont, 'GOTO', -1, -1, 'main')
            self.quadsList.append(cuadruplo)
            self.cont += 1
            #print(cuadruplo.printQuad())
```

Clase para la tabla de variables que recibe como parámetros los datos relacionados a la variable y crea un diccionario con el nombre de la variable como 'key' y asigna sus

datos a una lista la cual está dentro del mismo diccionario. Igualmente existen diversas funciones dentro de la misma clase con tareas específicas como buscar cierto nombre de una variable, o buscar su dirección, tipo, etc. También hay una función que se encarga de eliminar la variable con sus datos al momento de que la función a la que esta variable pertenecía acaba su ejecución.

```
class varTable:
    def __init__(self):
        self.table = {}

    def add(self, name, type, scope, direccion, direccion_funcion, dim_nonatomics):
        currentVar = Var(type, scope, direccion, direccion_funcion, dim_nonatomics)
        if name in self.table:
            print(f"The variable {name} already exists")
        else:
            self.table[name] = currentVar
            #print(f"Variable {name} saved successfully, direccion {direccion}")

    def search(self, name):
        if name in self.table:
            return name
        else:
            return "Variable undeclared"

    def find_address(self, name):
        if name in self.table:
            return self.table[name].direccion

    def deleteKey(self, name):
        del self.table[name]

    def find_type(self, name):
        if name in self.table:
            return self.table[name].type

    def size_array(self, name):
        if name in self.table:
            print(self.table[name].dim_nonatomics)
            return self.table[name].dim_nonatomics[-1]
```

Función que regresa el valor convertido al tipo que debe ser de acuerdo con la dirección asignada.

```
def convert_type(address, valor):
    val = None
    #Globales
    if address >= 1 and address < 1000:
        val = int(valor)
    elif address >= 1000 and address < 2000:
        val = float(valor)
    elif address >= 2000 and address < 3000:
        val = str(valor)
    elif address >= 3000 and address < 4000:
        val = int(valor)

    #Locales
    elif address >= 4000 and address < 5000:
        val = int(valor)
    elif address >= 5000 and address < 6000:
        val = float(valor)
    elif address >= 6000 and address < 7000:
        val = str(valor)
    elif address >= 7000 and address < 8000:
        val = int(valor)

    #Constantes
    elif address >= 13000 and address < 14000:
        val = int(valor)
    elif address >= 14000 and address < 15000:
        val = float(valor)
    elif address >= 15000 and address < 16000:
        val = str(valor)

    elif address >= 20000 and address < 22000:
        val = int(valor)

    return val
```

Función que regresa las operaciones aritméticas correspondientes.

```
def operaciones_arit(oper, valueIzq, valueDer):
    result = None
    if oper == '+':
        result = valueIzq + valueDer
    elif oper == '-':
        result = valueIzq - valueDer
    elif oper == '*':
        result = valueIzq * valueDer
    elif oper == '/':
        result = valueIzq / valueDer

    elif oper == '>':
        if valueIzq > valueDer:
            return 1
        else:
            return 0
    elif oper == '<':
        if valueIzq < valueDer:
            return 1
        else:
            return 0
    elif oper == 'igual':
        if valueIzq == valueDer:
            return 1
        else:
            return 0
    elif oper == 'not':
        if valueIzq != valueDer:
            return 1
        else:
            return 0
    elif oper == 'and':
        if valueIzq == 1 and valueDer == 1:
            return 1
        else:
            return 0
    elif oper == 'or':
```

Función que busca la variable que se le manda en alguno de los diccionarios.

```
def search_dict(address):
    if address in ctes:
        return ctes[address]
    elif address in global_mem:
        return global_mem[address]
    elif address in local_mem:
        return local_mem[address]
    elif contador == 1:
        print(f'ERROR: no se encontró el valor de la posicion {address}')
        exit()
    elif contador == 2:
        return local_mem[-1][address]
    elif (contador-1) == local_mem[len(local_mem)-1][pile_funcs[-1]]:
        if local_mem[len(local_mem)-1][address] == '':
            return local_mem[len(local_mem)-2][address]
        else:
            return local_mem[len(local_mem)-1][address]
```

Función que agrega la dirección con su valor en la memoria que le corresponde, ya sea global, local o de constantes.

```
def add_value(address, val):  
    if address > 0 and address < 4000:  
        global_mem[address] = val  
    elif address > 3999 and address < 8000:  
        if address in local_mem:  
            local_mem[address] = val  
        else:  
            local_mem[-1][address] = val  
    elif address > 19999 and address < 21000:  
        global_mem[address] = val  
    elif address > 20999 and address < 22000:  
        if address in local_mem:  
            local_mem[address] = val  
        else:  
            local_mem[-1][address] = val
```

Manual de Usuario

Este manual de usuario explica todo lo necesario para saber como desarrollar programas en ABC. Se muestran ejemplos y los comandos necesarios para correr el compilador y la máquina virtual.

Configuración del Ambiente

Para poder utilizar ABC solo se necesita tener instalado Python en tu computadora, de preferencia versión 2.7 en adelante.

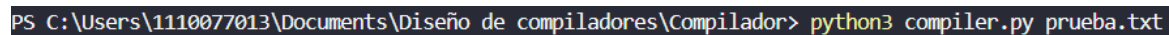
Para verificar la versión de Python que se tiene instalada solo basta con correr el siguiente comando en la terminal:

```
python --version
```

Compilación del Programa

El código que se quiera compilar debe estar escrito en un archivo txt. Seguido de eso necesitas el archivo compiler.py para poder compilar. Se abre la terminal y se escribe el comando:

```
python3 compiler.py prueba.txt
```



```
PS C:\Users\1110077013\Documents\Diseño de compiladores\Compilador> python3 compiler.py prueba.txt
```

*prueba.txt puede cambiar de acuerdo al nombre que tenga tu archivo que contenga el código.

Ejecución del Programa

Una vez que se compila el archivo se va a generar un archivo txt llamado dataVirtualMachine.txt. Para ejecutar solo debes abrir el programa llamado MaquinaVirtual.py y correrlo, no necesitas darle el nombre del archivo que se genera en compilación porque ya lo tiene. ¡Listo! Ya debes poder visualizar los resultados de tu programa en consola.



Estructura General del Programa

Explicando un poco la estructura del código podemos ver cómo los programas en el lenguaje ABC comienzan su primera línea de código dándole un nombre al programa de la siguiente manera: **program** '**nombre_programa**;',. Seguido de esto viene la declaración de variables globales, para esto es necesario escribir '**var**' seguido de las variables con su tipo, id y punto y coma. Después viene la declaración de todas las funciones donde tenemos que poner '**func**' seguido del tipo de la función, su id y sus parámetros entre paréntesis, para finalizar con un punto y coma. En la línea siguiente de la declaración de la función tenemos que declarar las variables locales de la misma y para esto usamos la misma estructura de la declaración de las variables globales. Cuando se terminan de declarar las variables locales procedemos a incluir los estatutos de la función entre llaves.

En la parte final del código siempre viene el main seguido de paréntesis y entre las llaves se encuentran los estatutos del mismo. En este ejemplo específico hacemos uso del 'write' el cual imprime en consola el valor dado entre paréntesis.

```
program writeDemo;

var
    int a;

func void func1 ();
var
    float b;
{
    read (a);
}

main() {
    write("HolaMundo");
}
```

Declaración de Variables

La estructura para declarar variables necesita la palabra var (una vez) antes de empezar a declarar. Los tipos que se permiten para declarar son int, float, char y bool. Haciendo énfasis en que los nombres de las variables no se pueden repetir. Dentro del main no se pueden definir variables, únicamente fuera o dentro de funciones. Además, cuando se declaran no pueden asignarse valores o realizar otras operaciones. Ejemplo:

```
program Compilador;

var
    int a;
    float b;
    char c;
    bool d;
```

Variables globales: Estas pueden ser utilizadas para funciones y el main, pero necesitan estar definidas fuera de las funciones y el main.

```
program patito;

var
    int a;
    int b;
    float f;

main() {
    a=3;
    b=a+1;
    write(a, b);
}
```

Variables locales: Son las declaradas dentro de las funciones pero únicamente existen ahí, es decir solo funcionan dentro de donde son declaradas. Van después de la firma de la función y antes de las llaves que inician el cuerpo de la función.

```
func void dos(int a1, int b1, float g);
var
    int i;
{
    i = 5;
    while (i>0) do {
        a=a+b*i;
        uno(i*2);
        write(a);
        i=i-1;
    }
}
```

Declaración de Arreglos

La estructura para declarar arreglos es parecida a la de una variable. Lo que cambia es que únicamente hay de tipo int y entre corchetes va el tamaño del cual se quiere que sea el arreglo. Del mismo modo que las variables pueden ser globales o locales.

```
var
    int arra[2];
    int arrab[5];
```

A los arreglos se les puede asignar un valor entero y también pueden imprimir un valor que se esté buscando.

```
arrab[2+3] = 4 + a * b;
arrab[2] = 9;
```

Declaración de Funciones

La estructura para declarar funciones necesita la palabra func antes de empezar a declarar, seguido del tipo de función + el id de la función + (parámetros). Los tipos que se permiten para declarar son int, float, char, bool y void. Cada parámetro debe ser declarado con tipo y id seguido de coma si es más de uno. Void no regresa nada, por ende no se puede hacer uso de la palabra **return** en el cuerpo de la función. Los demás tipos de funciones si aceptan la palabra **return** y esta va dentro del cuerpo de la función y al final. El **return** siempre va a regresar una expresión o variable. Se pueden declarar variables propias o utilizar las variables globales.

Función Void:

```
func void dos(int a1, int b1, float g);
var
    int i;
{
    i = 5;
    while (i>0) do {
        a=a+b*i;
        uno(i*2);
        write(a);
        i=i-1;
    }
}
```

Funciones != Void:

```
func int fibo (int n);
var
    int resultado;
    int res1;
    int res2;
{
    if (n < 2) {
        resultado = n;
    }
    else {
        res1 = fibo(n - 1);
        res2 = fibo(n - 2);
        resultado = res1 + res2;
    }
    return resultado;
}
```

Declaración del Main

El main se declara usando la palabra reservada **main** seguido de () y cuando se va a comenzar a codificar se hace el uso de {}.

```
main() {
    a=3;
    b=a+1;
    write(a, b);
    f=3.14;
    dos(a+b*2, b, f*3);
    write(a,b,f+2*1);
}
```

Estatutos

Asignación: Para asignar valores a las variables se debe escribir el nombre de la variable seguido del signo = , el valor que se le quiera asignar y punto y coma. Debe coincidir el tipo de dato de la variable con el dato asignado.

```
resultado = n;
```

```
resultado = res1 + res2;
```

```
res1 = fibo(n - 1);
```

```
x = 5;  
y[3] = x;
```

Llamada a una función: Debe escribirse el nombre de la función que se quiere mandar a llamar y entre paréntesis los valores que se van a mandar como parámetros. Los parámetros deben coincidir tanto en número como en tipo de la firma de la función.

```
func void dos(int a1, int b1, float g);
```

```
dos(a+b*2, b, f*3);
```

Lectura de un valor: Debe escribirse la palabra reservada **read** y entre paréntesis la variable que va a guardar el valor del input que haga el usuario desde la consola.

```
read(a);
```

Impresión de un valor o letrero: Debe escribirse la palabra reservada **write** y entre paréntesis la variable que va a imprimirse o si es letrero la oración entre comillas.

```
write(0);  
write("hola");  
write(y1);
```

Condicional if/else: La estructura del if/else inicia con la palabra reservada if y entre paréntesis la condición que se desea verificar. Después, entre llaves va el código que se desea ejecutar. El else lleva la misma estructura pero con la palabra reservada else.

```

if (n < 2) {
    resultado = n;
}
else {
    res1 = fibo(n - 1);
    res2 = fibo(n - 2);
    resultado = res1 + res2;
}

```

Condicional While: La estructura del while inicia con la palabra reservada while y entre paréntesis la condición que se desea verificar, seguido de eso la palabra reservada do. Después, entre llaves va el código que se desea ejecutar.

```

while (i>0) do {
    a=a+b*i;
    uno(i*2);
    write(a);
    i=i-1;
}

```

Ciclo For: La estructura del for inicia con la palabra reservada for, seguido del id de una variable asignando un valor inicial, seguido la palabra reservada to, después la condición hasta la cual se quiere llegar y la palabra reservada do. Lo que se va a ejecutar hasta que se cumpla esa condición va entre llaves.

```

for i = 0 to a-2 do
{
    total = x1 + y1;
    y1 = x1;
    x1 = total;
    write(total);
}

```