MANUAL TÉCNICO

Ruth Nohemy Ardón Lechuga
Carnet: 201602975

Descripción de aplicación

MinorC es un subconjunto del lenguaje C, creado con el fin de poner en práctica los conceptos del proceso de compilación.

Augus es un lenguaje de programación, basado en PHP y en MIPS. Su principal funcionalidad es ser un lenguaje intermedio, ni de alto nivel como PHP ni de bajo nivel como el lenguaje ensamblador de MIPS.

El lenguaje tiene dos restricciones: la primera, es que cada instrucción es una operación simple; y la segunda, es que en cada instrucción hay un máximo de dos operandos y su asignación.

Es un lenguaje débilmente tipado, sin embargo, si se reconocen cuatro tipos de datos no explícitos: entero, punto flotante, cadena de caracteres y arreglo. Para manejar el flujo de control se proporciona la declaración de etiquetas.

La aplicación consiste que en base al ingreso de un código como MinorC, la aplicación traduce dicho código a un lenguaje de bajo nivel como lo es Augus. Una vez realizado el código este es optimizado y ejecutado.

Requerimientos mínimos

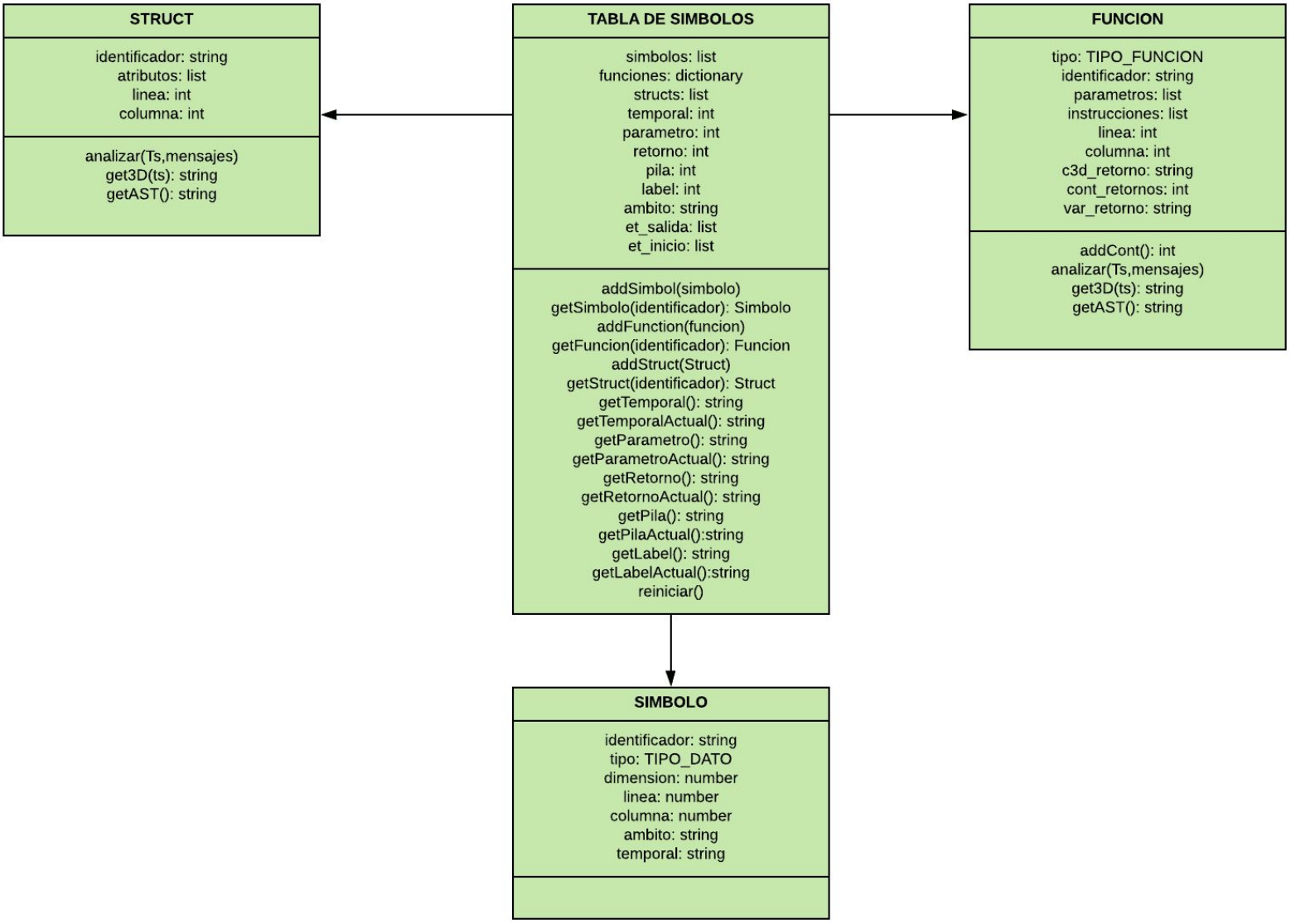
Definición de los requerimientos técnicos del sistema: Procesador I3, espacio disponible de al menos 5 MB, sistema operativo de Windows 10.

Aplicaciones/Librerías necesarias

- Visual Studio code (O cualquier editor de texto)
- Librería PLY
- Graphviz 2.28
- Librería Magicsticklibs
- Módulo tkinter
- Módulo easygui
- PyPI

Diagrama de clases

Clases Tabla de Símbolos



Clases instrucciones

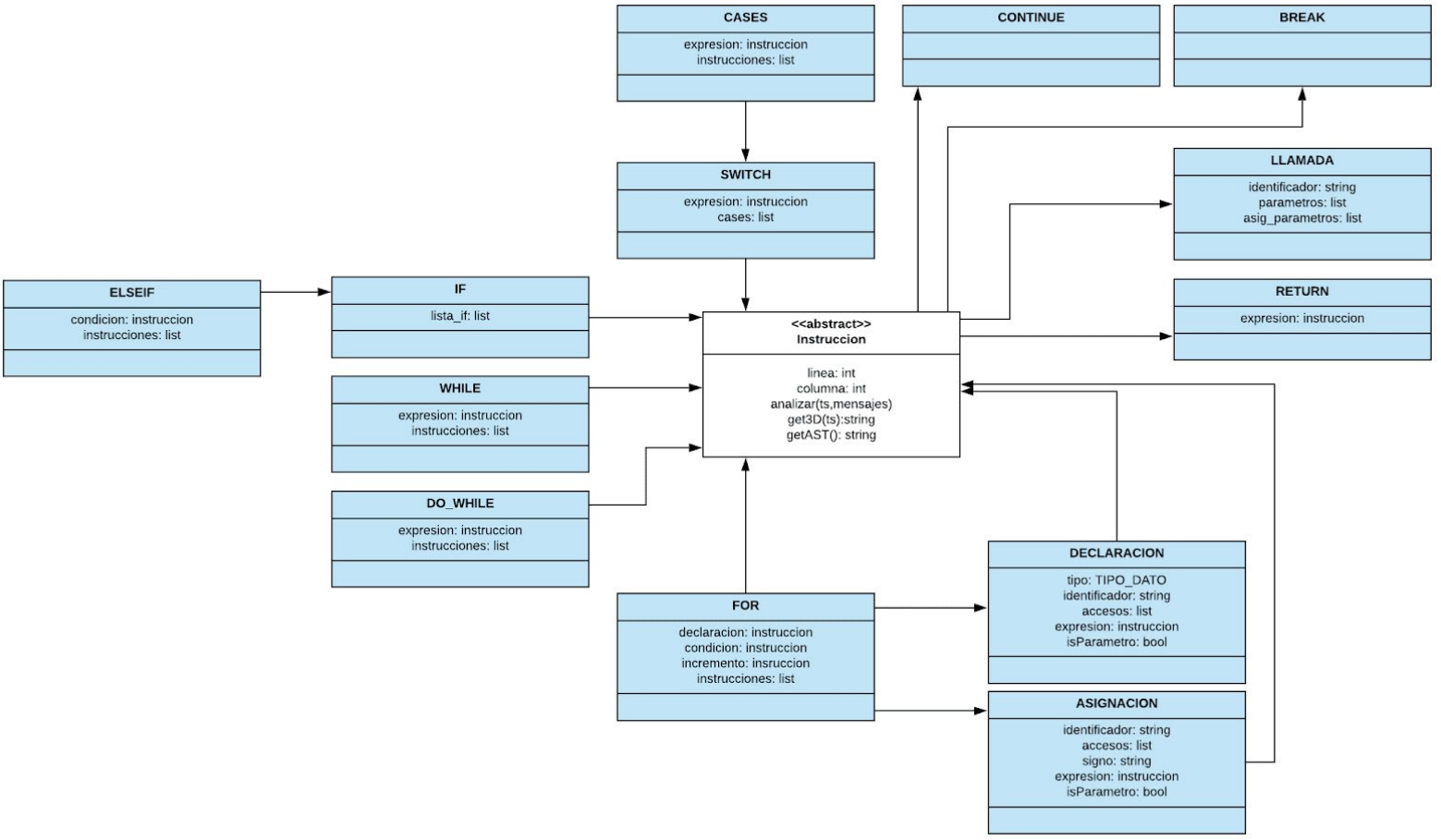
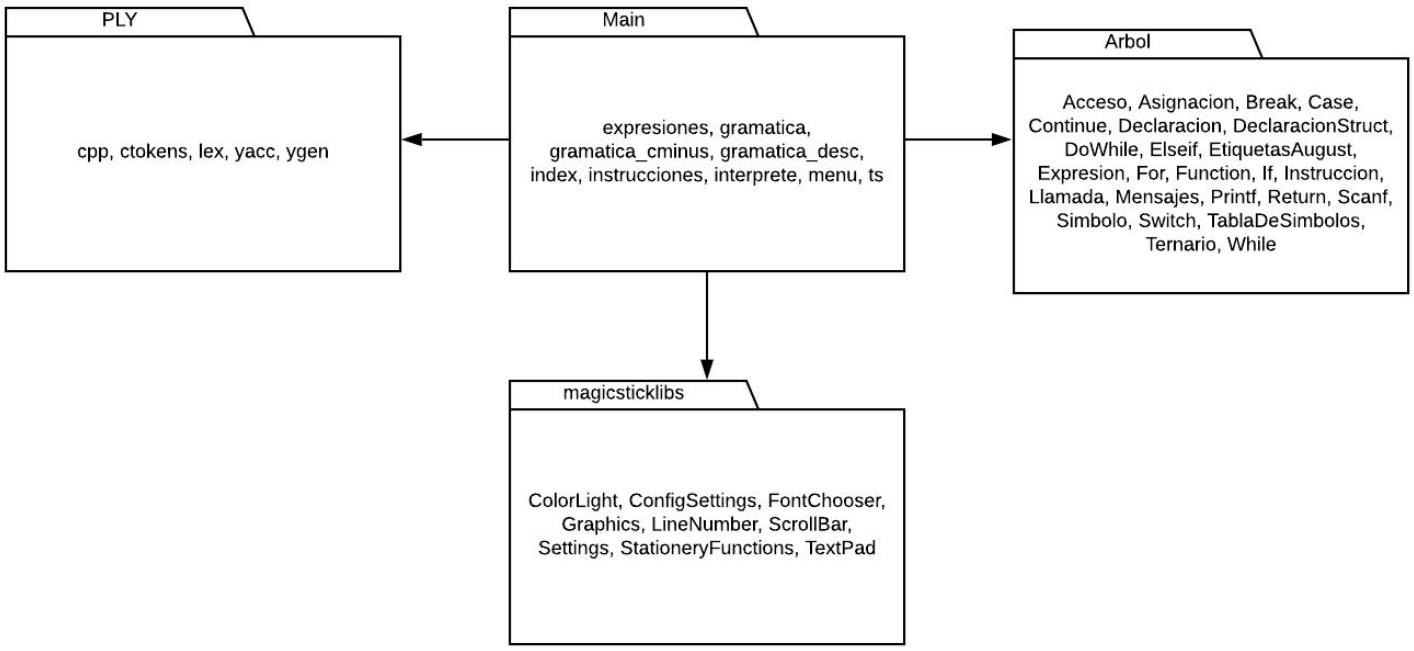


Diagrama de paquetes



Descripción de herramientas utilizadas

- PLY: Generador de Python de analizadores léxicos y sintácticos.

- Tkinter: Estándar para la interfaz gráfica de usuario para Python.

- Magicsticklibs: Editor simple en Python, repositorio público en: https://github.com/surajsinghbisht054/MagicStick_Editor. Se extrajeron las clases: ColorLight, ConfigSetting, FontChooser, Graphics, LineNumber, ScrollBar, Settings, StationeryFunctions y TextPad para poder realizar la interfaz gráfica.

- EasyGUI: Librería para producir mensajes y alertas.

- Graphviz: conjunto de herramientas de software para el diseño de diagramas definido en el lenguaje descriptivo DOT.

Descripción de clases y métodos

- gramatica_cminus: Contiene el analizador léxico y sintáctico, este análisis se realiza mediante atributos sintetizados. Se lleva una variable global para los errores léxicos y sintácticos. Además un arreglo para producir el reporte gramatical mientras se realiza el análisis. Y una variable tipo para poder guardar el valor en algunas producciones de declaración donde este es necesario.

- index: Clase en la cual empieza la ejecución, contiene toda la interfaz gráfica del IDE y los métodos de acceso para la ejecución correcta de los archivos de entrada.

- Instrucción: Interfaz con tres métodos: analizar, get3D y getAST. Todas las clases que implementen esta interfaz, deberán de implementar dichos métodos, todas esas clases se considerarán de tipo interfaz.
    - analizar: Método que se encargará de analizar todas las instrucciones del código para buscar errores semánticos que eviten la correcta traducción del código, además en esta fase se podrán establecer datos importantes como el tipo de dato de los símbolos o las funciones.
    - getAST: Método que construirá el árbol ascendente de análisis sintáctico de la clase.
    - get3D: Método se que se encargará de producir el código 3D de la instrucción en la que se encuentre..

- Símbolo: Clase que contiene todos los atributos esenciales y característicos de un símbolo en un compilador.

- Función: Clase que contiene todos los atributos esenciales y característicos de una función, además contendrá su código de retornos, variables de retorno y temporales que corresponden a sus parámetros.

- Struct: Clase que contiene todos los atributos esenciales y característicos de un struct.

- Tabla de símbolos: Estructura donde se almacenarán todos los símbolos que se encuentren en la ejecución del programa.
    - getSimbolo: Recibe como parámetro la cadena de identificador del símbolo y retorna símbolo en caso exista.
    - addSimbolo: Recibe como parámetro el nuevo símbolo y lo inserta en el diccionario.
    - getFuncion: Recibe como parámetro la cadena de identificador del la función y retorna función en caso exista.
    - addFuncion: Recibe como parámetro la nueva función y lo inserta en el diccionario.
    - getStruct: Recibe como parámetro la cadena de identificador del struct y retorna la definición del struct en caso exista.
    - addStruct: Recibe como parámetro el nuevo struct y lo inserta en el diccionario.
    - getTemporal, getTemporalActual, getParametro, getParametroActual, getRetorno, getRetornoActual, getPila, getPilaActual, getLabel, getLabelActual: Retorna la cadena representativa del identificador correspondiente.
    - Reiniciar: Restablece los valores de la tabla de símbolos

- Mensaje: Clase correspondiente a un nuevo mensaje o error. Contiene el tipo de error, la línea, columna y descripción del error que se encontró.

Explicación de acciones semánticas

GRAMÁTICA

init → instrucciones_globales
//se sintetiza el valor de instrucciones globales

instrucciones_globales → instrucciones_globales instruccion_global
//se agrega instruccion_global a la lista de instrucciones_globales
| instruccion_global
//se sintetiza el valor de instruccion_global como una lista de un elemento

instruccion_global → declaracion ;
| def_struct
| declaracion_struct ;
| metodo
//se sintetiza el valor de la instruccion

declaracion → tipo lista_asignaciones_dec
//se restablece la variable global Tipo a None y se sintetiza el valor de list_asignaciones_dec

lista_asignaciones_dec → lista_asignaciones_dec , asignacion_dec
//se agrega asignacion_dec a la lista de lista_asignaciones_dec
| asignacion_dec
//se sintetiza el valor de asignacion_dec y se guarda el tipo de la declaración en base a la pila

asignacion_dec → IDENTIFICADOR
| IDENTIFICADOR sig_asig expresion
| IDENTIFICADOR accesos
| IDENTIFICADOR accesos sig_asig expresion
//se crea el nodo de declaración

sig_asig → = | &= | <<= | >>= | /= | += | -= | %= | |= | *= | ^=
//se sintetiza el signo

tipo → int | float | double | char
//se crea el TIPO_DATO en base a la opción

accesos → accesos acceso
//se agrega el acceso a la lista de accesos
| acceso
//se sintetiza acceso como una lista

acceso → [expresion]
//se sintetiza expresión

acceso → [ ]
//se sintetiza como None

def_struct → STRUCT IDENTIFICADOR { struct_list_decl }
//se crea la definición del struct

struct_list_decl → struct_list_decl struct_decl
//se agrega struct_decl a la lista de struct_list_decl
| struct_decl
//se sintetiza struct_decl como una lista

struct_decl → tipo lista_id_struct ;
//se restablece tipo a None y se sintetiza lista_id_struct

lista_id_struct → lista_id_struct COMA id_struct
//se agrega id_struct a la lista de lista_id_struct
| id_struct
//se sintetiza id_struct como una lista de un elemento

id_struct → IDENTIFICADO
| IDENTIFICADOR accesos
//se crea la declaración

declaracion_struct → STRUCT IDENTIFICADOR IDENTIFICADOR
| STRUCT IDENTIFICADOR accesos IDENTIFICADOR
//se crea el nodo de declaracion del struct y se sintetiza

metodo → VOID IDENTIFICADOR (lista_parametros) { instrucciones }
| VOID IDENTIFICADOR ( ) { instrucciones }
| tipo IDENTIFICADOR (lista_parametros) { instrucciones }
| tipo IDENTIFICADOR ( ) { instrucciones }
//se crea el nodo metodo y se sintetiza

lista_parametros → lista_parametros parametro
//se agrega parametro a la lista de parametros
| parametro
//se sintetiza parametro como una lista

parametro → tipo IDENTIFICADOR
//se crea el parametro como una declaracion

instrucciones → instrucciones instruccion
//se agrega instruccion a instrucciones
| instruccion
//se sintetiza instruccion como una lista

instruccion → declaracion ;
| def_struct
| declaracion_struct ;
| printf ;
| asignacion ;
| if
| switch
| while
| do_while
| for
| break;
| continue;
| return
| llamada
| inc_dec;
//se sintetiza la instruccion

printf → PRINTF (CADENA , lista_param_printf) ;
| PRINTF (CADENA) ;
//se crea el nodo de la instruccion print

lista_param_printf → lista_param_print COMA expresion
//se agrega expresion a la lista de lista_param_print
| expresion
//se sintetiza expresion como una lista

asignacion → IDENTIFICADOR sig_asig expresion
| IDENTIFICADOR accesos sig_asig expresion
| IDENTIFICADOR lista_punto sig_asig expresion
| IDENTIFICADOR accesos lista_punto sig_asig expresion
//se crea el nodo de asignacion

lista_punto → lista_punto . valor
//se agrega valor a a la lista_punto
| .valor
//se sintetiza valor como una lista

valor → IDENTIFICADOR
| IDENTIFICADOR accesos
//se sintetiza el valor

if → IF (expresion) { instrucciones }
//se crea el una lista de if en un nodo if y se agrega el if principal
| IF (expresion) { instrucciones } ELSE { instrucciones }
//se crea una lista de if en un nodo if y se agrega el if principal y el else
| IF (expresion) { instrucciones } ELSE if
//a la lista de ifs del if, se le agrega el if

switch → SWITCH (expresion) { lista_cases }
//se crea el nodo de la instruccion switch
| SWITCH (expresion) { lista_cases default }
//se agrega a la lista de lista_cases el default y se sintetiza

lista_cases → lista_cases case
//se agrega el case a la lista_cases
| case
//se sintetiza el case como una lista

case → CASE expresion : instrucciones
//se crea un nodo case

default → DEFAULT : instrucciones
//se crea un nodo case

while → WHILE (expresion) { instrucciones }
//se crea el nodo de la instruccion while

do_while → DO { instrucciones } WHILE (expresion) ;
//se crea el nodo de la instruccion do while

for → FOR (for_ini ; for_exp ; for_inc) { instrucciones }
//se crea el nodo de la instruccion for

for_ini → identificador
| declaracion
| asignacion
| epsilon
//se sintetiza el valor para for_ini

for_exp → expresion
| epsilon
//se sintetiza el valor para for_exp

for_inc → asignacion
| inc_dec
| epsilon
//se sintetiza el valor para for_inc

inc_dec → ++ IDENTIFICADOR
| -- IDENTIFICADOR
| IDENTIFICADOR ++
| IDENTIFICADOR --
//se crea una instrucción de asignación en base a si es incremento o decremento

expresion → expresion + expresion
| expresion - expresion
| expresion * expresion
| expresion / expresion
| expresion % expresion
| expresion && expresion
| expresion || expresion
| expresion & expresio
| expresion | expresion
| expresion ^ expresion
| expresion << expresion
| expresion >> expresion
| expresion == expresion
| expresion != expresion
| expresion < expresion
| expresion > expresion
| expresion <= expresion
| expresion >= expresion
| ~expresion
| !expresion
| ++IDENTIFICADOR
| --IDENTIIFCADOR
| IDENTIFICADOR ++
| IDENTIFICADOR --
| IDENTIFICADOR accesos
| IDENTIFICADOR lista_punto
| IDENTIFICADOR accesos lista_punto
| ENTERO
| CADENA
| DECIMAL
| CARACTER
| IDENTIFICADOR
| llamada
| (expresion)
| expresion ? expresion : expresion
| sizeof (tipo)
| (tipo) expresion
//se crea el nodo de expresión en base a la cantidad de tokens que contiene la producción y del signo de la producción

llamada → IDENTIFICADOR ( )
| IDENTIFICADOR (lista_expresiones)
//se crea el nodo de la instruccion llamada

lista_expresiones → lista_expresiones COMA expresion
//se agrega expresion a la lista_expresiones
| expresion
//se sintetiza expresion como una lista