

**Diseño de Compiladores (Gpo 2)**

**Documentación del Compilador**

**Luis Eugenio Candelaria Azpilcueta (ITC)**

**A00816826**

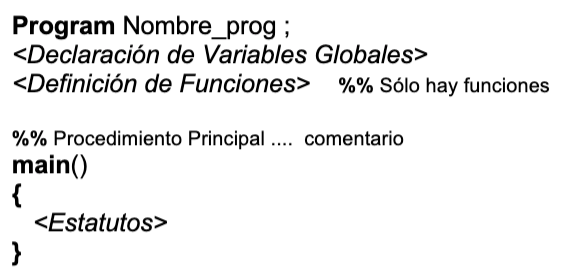
**29 de septiembre de 2021 Monterrey, Nuevo León**

**2. Descripción del lenguaje**

**MyRlike**

Es un lenguaje orientado a jóvenes que buscan aprender los fundamentos de la programación, a través del manejo y maniipulación de conjuntos de datos simples para realizar análiisis estadísticos básico.

La estructura general de un programa escrito en MyRlike es:



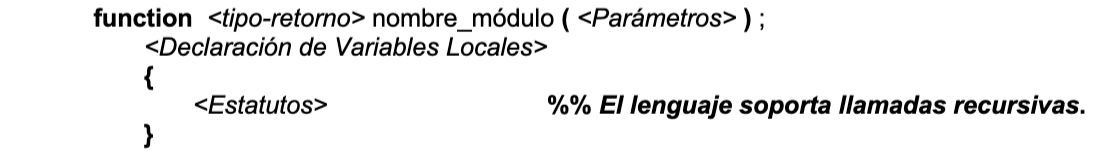
Este lenguaje tiene muchas similitudes con el C++, y considerando la experiencia que uno ha tenido aprendiendo programación desde cero, este parecido es una buena señal para los jóvenes a los que este lengueje es objetivo. Claro que se tienen algunas diferencias críticas, por ejemplo la declaraciión de variables y los diferentes estatutos que tiene este lenguaje, muchos de esos últimos son más cómodos programarlo en un lenguaje como Python.

Para la declaración de variables globales y locales:



Entre los tipos de variables, el lenguaje también soporta los arreglos, estos siendo de cualquier tipo también soportado por el lenguaje, se pueden crear arreglos de números enteros, flotantes y char, y el tamaño dee dicho arreglo será un número entero; aquí nos topamos con la primera restricción del lenguaje, debido a que solo acepta números enteros y el hecho de que solo existen variables globales se puede asumir el hecho de que las expresiónes dentro de los corchetes een un arreglo no podrán contener identificadores de otras variables, puesto que estas no tienen ninún valor een este punto del programa, por ejemplo, se declara una variable i entera, luego un arreglo de números flotantes, y en el tamaño del mismo se pone una expresión como la siguiente 1 + i \* 2, esta expresión no será válida, porque la variable i no tiene ningún valor, y como se acaba de mencionar, solo existen variables globales, no es como en otros lenguajes donde puedes declarar variables dentro del bloque del programa principal, es ilógico agregar un identificador en la expresión que representa el tamaño de un arreglo, para ello, posteriormente se considerará un tipo de expresión especial solo para la declaración de arreglos, es decir, que solo permita una expresión con constantes enteras y flotantes, y claro, como no hay identificadores, se puede calcular en el momento el tamaño del arreglo.

Para la declaración de funciones:



Aquí nos presentamos con el primer reto de este compilador que es las funciones y el uso de recursividad, que en términos generales, la recursividad es poder declarar una función dentro de sí misma, estos casos los vemos en funciones que hacen búsqueda en un árbol binario, o el ordenamiento de una lista, al igual que el lenguaje C++, una función requiere que se le asigne un tipo para saber si tendrá un estatuto de retorno y qué tipo de retorno tendrá, y como las variables solo tienen tres tipos: int, float y char, las funciones también tendrán tres tipos de retorno.

**Estatutos**

Existen 8 tipos de estatutos:

1. Asignación. Que es un ID acompañado por un operador ‘=’ y una expresión que determinará el valor una variable declarada con el ID capturado.
2. Retorno de una función. Como ya lo hablamos en la sección de funciones, las mismas podrán ser de tipo retorno, por lo que necesitarán un estatuto que permita regresar el valor de una expresión a la variable o expresión que la haya llamado.
3. Lectura. Esta si se me permite decir, es el estatuto que más se usa cuando un programador está aprendiendo o análisando un lenguaje y sus algoritmos, este estatuto consiste en imprimir en la terminal el valor de la variable con el ID escrito dentro de los paréntesis, y esto incluye arreglos y sus respectivos índices.
4. Escritura. Parecido al estatuto anterior, este también imprime un dato en la terminal, solo que este consiste en primero resolver una expresión y luego imprimirla, esto o si el usuario lo prefiere, poder imprimir párrafos de texto.
5. Estatuto de decisión. Los famosos estatutos condicionales, consisten en darle al usuario uan herramienta de decisión en el lenguaje, se obtendrán el valor de una expresión con un resultado de tipo booleano, generalmente comparando un dato con otro y con el mismo se decidirá si se cumple el bloque que sigue o los estatutos que vienen después.
   1. En este estatuto se puede tener un comando ‘else’ que permite al usuario ejecutar otro bloque en caso de que la expresión de un resultado falso.
6. Estatutos de repetición. Existen dos tipos de ciclos, los condicionales y no condicionales, ambos evaluando una expresión como condición para entrar al ciclo; simiilar que el estatuto de decisión servirá como decisión para el usuario si su expresión resulta en un booleano verdadero puede entrar al ciclo.

**Expresiones**

Las expresiones en MyRlike son las tradicionales (como en C y en Java). Existen los operadores aritméeticos lógicos y relacionales: +, -, \*, /, & (and), | (or), <, >, ==, ¡=. Se manejan las prioridades tradicionales, se pueden emplear paréntesis para alterarla.

En MyRlike existen identificadores, palabras reservadas, constantes enteras, flotantes, constantes char y string.

**Funciones especiales**

En el lenguaje también existen funciones que resuelven un problema específico, como lo es la media, moda y varianza que tendrán como parámetros una variable de tipo arreglo, regressión y plotXY que recibirán en este caso dos listas, y para evitar problemas semánticos, solo se aceptarán arreglos de tipo entero o flotante, si en los analizadores se encuentra un arreglo tipo char en estas funciones regresará un error.

**2.1 Listado de errores**

Este compilador hará un análisi profunndo de lo que el usuario ha escrito en el lenguaje MyRlike, y para ello se tendrá una seria condiciones por las que tiene que pasar para que se ejecute su programa, el siguiente listado muestra que tipo de errores puede desplegar el compilador:

|  |  |
| --- | --- |
| Sección | Mensajes de error |
| Analizador sintáctico | 1. “Error en la línea ‘x’” |
| Analizador de variables globales | 1. “Se declaró una variable más de una vez” 2. “El tamaño del arreglo debe ser entero” |
| Analizador de funciones | 1. “Se declaró una función más de una vez” 2. “Se repite el ID en las variables locales” 3. “Se repiite el ID en los parámetros” |
| Código Intermedio (Expresiones) | 1. “La expresión del índice de un arreglo debe resultar en entero” 2. “No se puede asignar un valor booleano a una variable” 3. “El resultado de la expresión no es del mismo tipo que la variable que fue asignado” 4. “No se puede cumplir x operación” (Validación por el cubo semántico) 5. “Una función void no puede llamarse dentro de una expresión” 6. “Una función de retorno no puede llamarse fuera de una expresión” 7. “No se puede pedir el valor de un índice a una variable que no es arreglo” |
| Código Intermedio (Estatutos) | 1. “No puedes pedir la media o varianza de un arreglo de tipo char” 2. “No puedes hacer funciones especiales con variables comúnes (no arreglos)” 3. “Se ha pasado el límite de memoria por variables” 4. “Se ha pasado el límiite de memoria por temporales” 5. “No se recibió un valor en el retorno de una función” 6. “Función x no tiene estatuto de retorno” |

**3. Descripción del compilador**

El siguiente capítulo de la documentación constará de una expliicación algunnas veces detalladas, otras general, esto por la razón de que el compilador tiene una metodología de análisis muy extensa, pero también muy directa. En la creación de este compilador se crearon una serie de archivos con el propósito de analizar un fragmento específico del código de prueba, ya sea las variables globales, funciones, parametros, etc. La división de subcapítulos en esta sección se dividió por dichos archivos de análisis, se guiará al lector sobre cómo se llegó a los cuádruplos y la solución del programa, de principio a fin, usando los archivos como punto de partida.

**3.1 Expresiones Regulares**

De acuerdo al documento de requerimientos se llegó a una lista de tokens y palabras reservadas en el lenguaje, cabe recalcar que hay una serie de palabras especiales que no suele tener cualquier lenguaje pero se necesitan en este compilador, tales como la media, moda, varianza, etc.

La lista de tokens es la siguiente (la columna de la derecha no está basada en lo que está en código en al analizadorLexico.py):

|  |  |
| --- | --- |
|  |  |
| Token | **¿Cómo se lee?** |
| ID | [a-zA-Z\_][a-zA-Z\_0-9] |
| CTE\_STRING | [a-zA-z\_] |
| CTE\_INT | [0-9\_] |
| CTE\_FLOAT | [0-9]+.[0-9] |
| CTE\_CHAR | [a, b, c,…] o [A, B, C,…] |
| COLON | : |
| SEMICOLON | ; |
| COMA | , |
| LBRACK | { |
| RBRACK | } |
| LPAREN | ( |
| RPAREN | ) |
| LSBRACK | [ |
| RSBRACK | ] |
| DIFF | <> |
| EQUAL | == |
| GTHAN | > |
| LTHAN | < |
| PLUS | + |
| MINUS | - |
| TIMES | \* |
| DIVIDE | / |
| ASSIGN | = |
| AND | & |
| OR | | |

En cuanto a las palabras reservadas, son las siguientes:

|  |
| --- |
| Palabras reservadas |
| IF |
| THEN |
| ELSE |
| PROGRAM |
| MAIN |
| VARS |
| INT |
| FLOAT |
| CHAR |
| FUNCTION |
| VOID |
| RETURN |
| READ |
| WRITE |
| WHILE |
| DO |
| FOR |
| TO |
| AVERAGE |
| MODE |
| VARIANCE |
| REGRESSION |
| PLOTXY |

**3.2 Gramática**

Para la gramática tuve que considerar una serie de cosas antes de entrar de lleno al esto que es el diseño de mi compilador, aún cuando la sintáxis es solo una parte a analizar del compilador, la complejidad con la que lo quería diseñar me exijía que el analizador sintáctico fuera como una especie de puerta a todo lo demás, ¿qué quiero decir con esto?, las reglas de sintaxis pueden ser claras pero como usuario hay muchas formas de evadirlas; para aclarar este punto, hablaré específicamente de una regla, en las especificaciones se pide el usuario pueda y solamente de esa forma, declarar variables que sean globales, no es como en C++, Python o cualquier otro lenguaje popular que te permite declarar variables dentro de un bloque, ¿cuál es el problema?, de la forma que yo lo veo, es que cuando quiero hacer referencia a una variable arreglo del tipo que sea, quisiera usar un indice en base a una operación matemática, como lo es el clásico:



En el ejemplo anterior vemos cómo el índice no es precisamente una operación matemática, pero sí es derivada de las reglas sobre expresiones, en otras palabras, una regla de qué poner dentro de los corchetes sería permitirle al usuario poner una expresión, y de ahí llegar hasta un ID como una regla derivada de las expresiones. El conflicto que uno tiene con este punto es que en variables globales, poner una regla de expresión genérica sería absurdo, por el camino que esta claramente señalado, en una variable arreglo global no puedes darle como tamaño una operación que hace referencia a un ID, porque el mismo no tiene ningún valor para ese punto, y la misma incongruencia se encuentra con otras reglas derivadas de expresiones, como lo es el poder usar una compuerta lógica OR o AND. Para tratar este problema decidí crear una regla de expresiones específica para la declaración de variables, así cuando se quiera crear un arreglo global, se puede determinar su tamaño con una suma, resta, multiplicación y división, pero solo con números enteros, evadiendo las demás reglas derivadas de la de expresión genérica.

El usar esta metología creará una cantidad considerable de reglas de sintáxis, y se aprovechó esta situación creando un analizador semántico, que tiene como propósito tener un resultado visual y textual de lo que el usuario ha escrito en el lenguaje que representa el compilador. Más detalles se hablarán sobre el analizador semántico en un futuro capítulo. Solo como adelanto, el contenido resultante del analizador semántico permite ir paso a paso en el diagrama de sintaxis creado en cada caso de prueba, con esto, desde el mismo analizador de sintaxis se puede analizar todo lo demás, desde variables globales, hasta llegar a los cuádruplos.

Como ya he mencionado las reglas de sintaxis son bastantes y muy espcíficas, pero puedo asegurar que con estas los errores por parte del compilador son escasos. Las reglas son las siguientes:

program -> PROGRAM ID SEMICOLON programVer1

PROGRAM ID SEMICOLON programVer2

PROGRAM ID SEMICOLON programVer3

programVer1 -> variables functions principal

programVer2 -> variables principal

functions principal

programVer3 -> principal

variables -> VARS varAux

varAux -> oneVar

sameType

newType

oneVar -> type COLON identifierVar SEMICOLON

sameType -> type COLON identifierVar COMA sameTypeFinal

type COLON identifierVar COMA sameTypeRecursive

sameTypeFinal -> identifierVar SEMICOLON

sameTypeRecursive -> identifierVar COMA sameTypeRecursive

identifierVar COMA sameTypeFinal

identifierVar SEMICOLON varAux

newType -> type COLON identifierVar SEMICOLON varAux

functions -> funcAux

recursiveFunc

recursiveFunc -> funcAux functions

funcAux -> FUNCTION typeFunction ID LPAREN withParemeters

FUNCTION typeFunction ID emptyPar RPAREN funcVer1

FUNCTION typeFunction ID emptyPar RPAREN funcVer2

emptyPar -> LPAREN RPAREN

typeFunction -> INT

FLOAT

CHAR

VOID

withParameters -> parameters RPAREN funcVer1

parameters RPAERN funcVer2

funcVer1 -> variablesLoc block

funcVer2 -> block

variablesLoc -> VARS varAuxLoc

varAuxLoc -> oneVarLoc

sameTypeLoc

newTypeLoc

oneVarLoc -> type COLON identifier SEMICOLON

sameTypeLoc -> type COLON identifier COMA sameTypeFinalLoc

type COLON identifier COMA sameTypeRecursiveLoc

sameTypeFinalLoc -> identifier SEMICOLON

sameTypeRecursiveLoc -> identifier COMA sameTypeRecursiveLoc

identifier COMA sameTypeFinalLoc

identifier SEMICOLON varAuxLoc

newTypeLoc -> type COLON identifier SEMICOLON varAuxLoc

parameters -> oneParam

sameTypeParam

newParam

oneParam -> type COLON identifierVar

sameTypeParam -> type COLON identifierVar COMA sameTypeParamFinal

type COLON identifierVar COMA sameTypeParamRecursive

sameTypeParamFinal -> identifierVar

sameTypeParamRecursive -> identifierVar COMA sameTypeParamRecursive

identifierVar COMA sameTypeFinal

identifierVar SEMICOLON parameters

newParam -> type COLON identifierVar SEMICOLON parameters

identifierVar -> indentLonely

identArrayVar

identArrayVar -> ID LSBRACK expressionVar RSBRACK

type -> INT

FLOAT

CHAR

principal -> MAIN LPAREN RPAREN block

block -> LBRACK blockAux

blockAux -> statute RBRACK

blockRecursive

blockFinal

blockFinal -> RBRACK

blockRecursive -> statute blockAux

statute -> assignment SEMICOLON

if\_statement

writing SEMICOLON

reading SEMICOLON

loop

functionCall SEMICOLON

return\_statement SEMICOLON

regressionFunc SEMICOLON

plotXYFunc SEMICOLON

assignment -> identifier ASSIGN expression

plotXYFunc -> PLOTXY LPAREN ID COMA ID RPAREN

regressionFunc -> REGRESSION LPAREN ID COMA ID RPAREN

return\_statement -> RETURN LPAREN expression RPAREN

if\_statement -> shortCondition

longCondition

shortCondition -> IF LPAREN expression RPAREN THEN block

longCondition -> IF LPAREN expression RPAREN THEN block ELSE block

writing -> WRITE LPAREN writingAux

writingAux -> writingFinal

writingRecursive

writingFinal -> writingString RPAREN

expression RPAREN

writiingString -> CTE\_STRING

writingRecursive -> CTE\_STRING COMA writingFinal

expression COMA writingFinal

CTE\_STRING COMA writingRecursive

expression COMA writingRecursive

reading -> READ LPAREN readingAux

readingAux -> readingFinal

readingRecursive

readingFinal -> identifier RPAREN

readingRecursive -> identifier COMA readingRecursive

identifier COMA readingFinal

loop -> conditional

nonconditional

conditional -> WHILE LPAREN expression RPAREN block

nonconditional -> FOR assignment to expression DO block

functionCall -> ID LPAREN callAux

callAux -> expression RPAREN

callRecursive

callFinal

callRecursive -> expression COMA callAux

callFinal -> RPAREN

expression -> term

expressionOperation

expressionOperation -> term PLUS expression

term MINUS expression

term -> factor

termOperation

termOperation -> factor TIMES term

factor DIVIDE term

factor -> idioms

factorOperation

factorOperation -> idioms GTHAN factor

idioms LTHAN factor

idioms EQUAL factor

idioms DIFF factor

idioms -> clause

idiomsOperation

idiomsOperation -> auction OR idioms

auction AND idioms

auction -> constant

otherExpression

identifier

averageFunc

modeFunc

varianceFunc

functionCall

constant -> someInt

someFloat

someChar

someInt -> CTE\_INT

someFloat -> CTE\_FLOAT

someChar -> CTE\_CHAR

otherExpression -> LPAREN expression RPAREN

identifier -> identLonely

identArray

identLonely -> ID

identArray -> ID LSBRACK expression RSBRACK

averageFunc -> AVERAGE LPAREN ID RPAREN

modeFunc -> MODE LPAREN ID RPAREN

varianceFunc -> VARIANCE LPAREN ID RPAREN

expressionVar -> termVar

expressionOperationVar

expressionOperationVar -> termVar PLUS expressionVar

termVar MINUS expressionVar

termVar -> auctionVar

termOperationVar

termOperationVar -> auctionVar TIMES termVar

auctionVar DIVIDE termVar

auctionVar -> constantVar

otherExpressionVar

constantVar -> CTE\_INT

otherExpressionVar -> LPAREN expressionVar RPAREN

**3.3 Analizador Semántico**

Esta estrategia la he aprendido en mi investgación para aprender el uso de PLY para el compilador, de acuerdo con este tutorial, existe una forma de desplegar en forma de árbol (yo lo llamo: árbol semántico) el resultado del analizador sintáctico, esta solución consiste en escribir en un archivo de texto qué regla de sintaxis se usó y sus derivados, por ejemplo, veamos la primera regla: program, esta regla tiene dos hijos, que son el ID del programa y la version del programa (ya sea programVer1, programVer2 o programVer3) y el hijo que representa la regla de sintaxis de la versión del programa principal, tiene otros hijos, y todas estas reglas imprimen en un archivo su propio nombre y sus hijos, con este método se llega a una programación que admito se sintió bastante lenta, pero muy segura, ¿por qué es así?, el hecho de que se impriman los hijos de una regla de sintaxis te asegura que lo que el usuario está haciendo se esté analizando de la forma correcta.

La forma de accesar a imprimir el árbol semántico es creando una serie de clases, todas con las mismas funciones pero diferentes detalles, por ejemplo, la regla varAux tiene un solo hijo, entonces en mi analizador semántico tiene una clase llamada varAux que en su incializador tiene como parametro un solo hijo, y su propio nombre, y dentro de esta clase está la función traducir, que tiene como propósito imprimir el nombre de la clase y un contador junto a su hijo, este contador es el número del nodo o identificador que tiene en el árbol semántico. El siguiente código muestra un ejemplo de estas clases:

class callAux(Node):

def \_\_init\_\_(self, son1, name):

self.name = name

self.son1 = son1

def imprimir(self, ident):

#if str(type(self.son1)) == "<type 'tuple'>":

#elif str(type(self.son1)) == "<type 'instance'>":

if type(self.son1) == type(tuple()):

self.son1[0].imprimir(" " + ident)

else:

self.son1.imprimir(" " + ident)

print(ident + "Nodo: " + self.name)

def traducir(self):

global txt

id = incrementarContador()

if type(self.son1) == type(tuple()):

son1 = self.son1[0].traducir()

else:

son1 = self.son1.traducir()

txt += id + "[label = "+self.name+"]"+"\n"

txt += id + "->" + son1 + "\n"

return id

Como Podemos ver en el código anterior, callAux recibe un solo hijo o nodo, y su propio nombre, con esto traducimos la clase en un formato parecido a una lista, el resultado de traducir es “[label = callAux]” teniendo como id ese contador que funciona como identificador del nodo dentro del árbol semántico y después de “traducir” su propio nombre traduce el del hijo y fuera de esta función, dentro de mi archivo “analizadorSintactico” hago la función de imprimir para que los nodos se desplieguen en un archivo de texto. El siguiente fragmento muestra el árbol semántico en formato de texto:

2[ID = pruebaNueva]

7[Tipo = int]

10[ID = i]

9[label = identLonely]

9->10

8[label = identifierVar]

8->9

14[ID = j]

13[label = identLonely]

13->14

Como pueden ver, a la izquierda está el ID que mencioné y también se imprime un renglón con el mismo ID pero señalando los hijos de la etiqueta, por ejemplo el ID 9 con la etiqueta identLonely, tiene un hijo, que si vemos en las reglas de sintáxis es la función que solo trae un ID para una variable, entonces dentro del archivo de texto el nodo con el ID 10 tiene la nombre de la variable solitaria (no arreglo), y de hecho lo vemos arriba, el ID 10 tiene como valor de etiqueta “i”, ¿esto que significa? Que el usuario inició una variable llamada i.

¿Pero cómo creamos los objetos de las clases? Si uno ya se ha aventurado en mi archivo “analizadorSintactico” puede darse cuenta muy rápido, pero aquíi lo muestro:

def p\_program(p):

'''

program : PROGRAM ID SEMICOLON programVer1

| PROGRAM ID SEMICOLON programVer2

| PROGRAM ID SEMICOLON programVer3

'''

p[0] = program(ID(p[2]), p[4], "program")

Tenemos la regla de sintaxis y en la parte de abajo, tenemos una especie de detonador que crea los objetos con el nombre de la misma regla de sintaxis y mandando sus hijos usando el resultado de la regla como una lista, siendo p[2] la regla de ID (esta solo imprime el valor del ID) y p[4] que es la versión del programa.

Supongo que para este punto se puede intuir qué método se hará con este compilador, cabe mencionar que a partir de este momento no pude encontrar una guía de cómo interpretar el árbol semántico, así que todo el códiigo a partir de ahora fue creado en su totalidad por mí.

**3.4 Analizador del árbol**

Para el análisis del árbol semántico se ha creado un archivo llamado “analizadorArbol” que tiene como propósito leer esta enorme lista de etiquetas regresando que hijos tiene cada regla, la toma de una etiqueta específiica y la obtención del valor de una etiqueta, por ejemplo [label = callAux] regresa “callAux”.

El método de lectura del archivo de texto es tan aburrido como suena, es en pocas palabras leer cada renglón hasta toparse con un ID de un nodo y regresar sus hijos o su etiqueta, por desgracia no me di el tiempo de hacer más eficiente la búsqueda de hijos de un nodo, aunque si es necesario buscar de arriba-abajo, uno si quisiera hacer eficiente este método podría eliminar los nodos ya analizados, porque, por si aún no ha quedado claro el método, lo que se busca es analizar un nodo y sus hijos, pero nunca volver a analizar el padre, todo será lento (para programar) pero muy directo.

La función más importante como ya habrá sospechado es la que trae los hijos de un nodo, y su código es bastante simple de entender, en el siguiente bloque lo muestro:

def gimmeTheChildren(hijo, lista):

lenght = len(lista)

hijos = []

tamañoHijo = len(hijo)

substr = hijo + "-"

for index in range(lenght):

string = lista[index]

sub = string.find(substr, 0, tamañoHijo+1)

if(sub != -1):

newstr = string[tamañoHijo+2:len(string)]

hijos.append(newstr)

return hijos

**3.5 Roma**

Como dice la frase “todos los caminos llegan a roma”, usaré esa idea para este compilador, usando un archivo llamado “roma” como un punto de partida crítico del análisis del árbol semántico, en este archivo se exportaran otros que nos traeran una serie de diccionarios, empezando por el diccionario de variables globales, roma hace un llamado al archivo “analizadorVarGlobales”, de ahí al “analizadorFunciones”, “analizadorBloqueFun” y finalmente “millenium”, todos estos analizadores trabajan con la ayuda del archivo “analizadorArbol” que lee nuestro árbol semántiico y nos dice qué está haciendo el usuario con su código.

Ahora daré un pequeño tour para mostrar como funcionará el analizador del árbol, empezanndo por las variables globales:

def variablesGlobales():

global tree

global dictVariablesGlobales

label = aA.gimmeTheLabel(tree, "3")

value = aA.gimmeTheValue(label)

if(value == "programVer1"):

hijos = aA.gimmeTheChildren("3", tree)

vars = hijos[0]

dictVariablesGlobales = aVG.init(vars, tree)

elif(value == "programVer2"):

hijos = hijos = aA.gimmeTheChildren("3", tree)

hijo = hijos[0]

label = aA.gimmeTheLabel(tree, hijo)

value = aA.gimmeTheValue(label)

if(value == "variables"):

vars = hijo

dictVariablesGlobales = aVG.init(vars, tree)

Aquí podemos ver como se obtienen la etiqueta del nodo “3” ¿por qué el 3? Porque ese nodo siempre es el que tene la versión del programa (esta es una regla de sintaxis obligatoria), y pues la versión 1 y 2 tiene la regla de las variables globales, se saca el ID del nodo de las variables globales y se manda al archivo de “analzadorVarGlobales” a la función “init” junto con el árbol semántico. Esta misma estrategia se usa para las funciones, la versión indica si se necesita analizar el árbol para las funciones que escribió el usuario.

**3.6 Analizador de variables globales**

Como mencioné en un capítulo anterior, el análisis del árbol semántico es lento pero muy directo, sabemos que el nodo que llega a este archivo es el mismo que usó la regla de sintaxis “p\_vars” y en base al nombre de cada regla se redactaron funciones que analizan las mismas. Llegando al init del archivo “analizadorVarGlobales” se tiene el siguiente código:

def init(vars, lista):

global dictVariablesGlobales

global tree

tree = lista

an\_vars(vars)

#printInfo(dictVariablesGlobales)

return dictVariablesGlobales

Es muy sencillo, cada nodo representa una regla, lo que se tiene que hacer es analizar ese nodo de la forma que se lee la regla y así llegar a valores críticos, siguiendo el proceso vamos a analizar el nodo vars:

def an\_vars(vars):

global tree

hijos = aA.gimmeTheChildren(vars, tree)

varAux = hijos[0]

an\_varAux(varAux)

Sabemos por el archivo “analizadorSintactico” que “vars” solo puede tener un hijo, y que ese hijo es la regla “varAux”, y de aquí vamos a analizar ese nodo:

def an\_varAux(varAux):

global tree

hijos = aA.gimmeTheChildren(varAux, tree)

hijo = hijos[0]

label = aA.gimmeTheLabel(tree, hijo)

value = aA.gimmeTheValue(label)

if(value == "oneVar"):

an\_oneVar(hijo)

elif(value == "sameType"):

an\_sameType(hijo)

else:

an\_newType(hijo)

Ahora vemos que la regla tiene diferentes versiones de hijos, o es “oneVar”, “sameType” o “newType”, para reconocer la regla leemos el valor de la etiqueta del hijo, y dependiendo de ese último resultado vamos a analizar la regla que representa el hijo de “varAux”, todos los analizadores trabajan de la misma forma, pero claro, representan diferentes partes del archivo de prueba. Entonces si queremos entender cómo funciona un analizador es simplemente leyendo las reglas de sintaxis del compilador.

La parte importante de este archivo es la creación de variables para meterlas a un diccionario, si se usa una regla de sintaxis recursiva se guardan las variables en una lista junto con su tipo y en casos especiales el tamaño de un arreglo.

Para el tamaño de un arreglo se usa otro analizador que tiene como objetivo calcular el valor de una expresión dentro de los corchetes en la inicialización de una variable.

**3.7 Analizador de expresiones especiales**

¿Por qué estas expresiones son especiales? Cuando se inicializa un arreglo se espera que el tamaño sea un entero, para ello se deben usar una serie de restricciones como no usar un char en la expresión o tener cuidado con el resultado con una operación que usa valores flotantes.

El archivo “analizadorExpresionesEsp” solo hace un seguimiento del árbol semántico y reune toda la expresión en una lista, por ejemplo: [1, +, 2, \*, 3]. Para resolver esta expresión se usa otro archivo llamado “solver”.

**3.7.1 Solver**

La resolución de esta lista se hace índice por índice tomando en cuenta la prioridad del operador, por ejemplo, en la lista escriita en el párrafo anterior, primero se tiene que resolver 2 \* 3, luego se hace la suma 1 + 6 y con esto regresa el valor al archivo anterior para inicializar la variable, en “solver” es donde se tiene ciudado con los flotantes, si el resultado es un número flotante regresa un error.

**3.8 Clase para variables**

En el archivo “classVariables” se tienen las clases para crear los objetos que representan las variables globales, locales y parametros del programa, para ello solo requiere unos parámetros críticos, una vez teniendo el objeto se anexará al diccionario que lo necesita, así se tendrá un diccionario con este formato:

diccionarioVariablesGlobales = {

“IDVariable” : obj <VariableComún>

}

El siguiente fragmento de código representa la clase para una variable común, es decir, no arreglo:

class VariableComun():

especie = "Comun"

def \_\_init\_\_(self, ID, tipo):

self.ID = ID

self.tipo = tipo

if(tipo == "int"):

self.valor = 0

elif(tipo == "float"):

self.valor = 0.0

elif(tipo == "char"):

self.valor = '\_'

def imprimirDatos(self):

print(self.ID + "\t" + self.tipo + "\t" + self.especie + "\t" + str(self.valor))

def asignar(self, nuevo):

self.valor = nuevo

def getValue(self):

return self.valor

Como una nota extra, uno de mis planes era usar los objetos para resolver los cuádruplos del compilador, después se agregó el requerimiento de direcciones de memoria, tema del que se hablará después.

**3.9 Clase para funciones**

Para las funciones también se crearon clases, habrá de dos tipos, funciones de retorno, y funciones void y dentro de las mismas, se tendrá como atributo las variables locales, los parametros y los estatutos, el siguiente código muestra una de estas clases:

class FuncionReturn():

especie = "Retorno"

def \_\_init\_\_(self, ID, tipo, parametros, variablesLocales, estatutos):

self.ID = ID

self.tipo = tipo

self.parametros = parametros

self.variablesLocales = variablesLocales

self.estatutos = estatutos

def imprimirDatos(self):

print(self.ID + "\t" + self.tipo + "\t")

print(self.parametros)

print(self.variablesLocales)

print(self.estatutos)

Con las funciones no se tiene una propósiito específiico mas hayá de guardar su información, posteriormente, cuando se necesiten sus estatutos que estarán en formato de cuádruplos solo se accesarán desde el objeto, al igual que las variables globales, las funciones se guardarán en un diccionario, y dentro de este objeto se tendrán otros diccionarios, representando las variables, parámetros y cuádruplos (estatutos).

Con las variables locales y los parámetros se usan objetos en base a las clases que vimos en el capítulo anterior, entonces, así como el programa principal tiene un diccionario de objetos (variables), cada función tiene dos diccionarios de objetos (variables).

**3.10 Analizador de funciones**

Como ya expliqué en el capítulo de las variables globales, los analizadores usan el árbol semántico para saber que está haciendo el usuario, y es con el mismo por el que el compilador sabe si debe analizar las variables locales y/o los parámetros, una de las funciones más importantes de este archivo “analizadorFunciones” es la siguiente:

def an\_funcAux(funcAux):

global tree

hijos = aA.gimmeTheChildren(funcAux, tree)

typeFunction = hijos[0]

ID = hijos[1]

hijo = hijos[2]

tipo = an\_label(typeFunction)

ID = an\_label(ID)

value = an\_label(hijo)

if(value == "withParameters"):

withParameters = hijo

diccionario = an\_withParameters(withParameters)

parametros = diccionario["parameters"]

variablesLocales = diccionario["varLoc"]

estatutos = diccionario["estatutos"]

if(tipo == "void"):

obj = FunctionVoid(ID, parametros, variablesLocales, estatutos)

dictFunciones[ID] = obj

else:

obj = FuncionReturn(ID, parametros, variablesLocales, estatutos)

dictFunciones[ID] = obj

elif(value == "funcVer1"):

funcVer1 = hijo

diccionario = an\_funcVer1(funcVer1)

variablesLocales = diccionario["varLoc"]

estatutos = diccionario["estatutos"]

parametros = "vacio"

if(tipo == "void"):

obj = FunctionVoid(ID, parametros, variablesLocales, estatutos)

dictFunciones[ID] = obj

else:

obj = FuncionReturn(ID, tipo, parametros, variablesLocales, estatutos)

dictFunciones[ID] = obj

else:

funcVer2 = hijo

diccionario = an\_funcVer2(funcVer2)

estatutos = diccionario["estatutos"]

variablesLocales = "vacio"

parametros = "vacio"

if(tipo == "void"):

obj = FunctionVoid(ID, parametros, variablesLocales, estatutos)

dictFunciones[ID] = obj

else:

obj = FuncionReturn(ID, tipo, parametros, variablesLocales, estatutos)

dictFunciones[ID] = obj

En esta función analizamos qué tipo de función es, si va a tener parámetros y/o variables locales, cuando se tengan se hace el llamado a los archivos “analizadorVarLocales” y “analizadorParametros”. Al final se analiza el bloque de las funciones, para obtener sus estatutos en formato de cuádruplos, cabe mencionar que los mismos llegan también en formato de diccionario para tener el orden bien registrado de los cuádruplos.

**3.11 Variables locales y parámetros**

Los archivos “analizadorVarLocales” y “analizadorParametros” tienen una función similar que el archivo que consigue las variables globales, una de las diferencias es que se verifica que los Ids de los parámetros no coincidan con las variables locales. Así como los demás analizadores, es muy directo, checa qué regla utiliza y con ello sabe qué hijos tiene y los analiza hasta el final.

**3.12 Analizador de bloque**

El archivo “analizadorBloqueFun” analiza desde el árbol semántico los estatutos de las funciones y el programa principal, creando los cuádruplos y reescribiendo los que necesitan un dato extra, por ejemplo, cuando se tiene un cuádruplo para un “if-statement” una vez resuelta una expresión se debe verificar si la condición se cumple o nó para hacer lo que dice el bloque del if o saltarlo, para ello se generan cuádruplos auxiliares, al final de la expresión dentro del if, se genera [‘GotoF’, ID] con ID siendo el temporal resultante de la expresión, cuando se trata de un ciclo no condicional como un for, se genera un [‘GotoT’], como se puede ver en este momento a estos cuádruplos les falta un dato que es el índice del cuádruplo al que debe saltar, este dato se obtiene en otra función:

def remakeStack(pila):

contador = 1

diccionario = {}

pending = []

jumps = []

for index in range(len(pila)):

quad = pila[index]

if(len(quad) == 0):

ignore = ''

else:

inst = quad[0]

if(inst == 'GotoF'):

pending = [contador] + pending

if(inst == 'EndOfIf'):

jumps = jumps + [contador + 1]

pend = pending.pop()

prevQuad = diccionario[pend]

jumpTo = jumps.pop()

prevQuad += [jumpTo]

diccionario[pend] = prevQuad

if(inst == 'BeginElse'):

jumps = jumps + [contador + 1]

pend = pending.pop()

prevQuad = diccionario[pend]

jumpTo = jumps.pop()

prevQuad += [jumpTo]

diccionario[pend] = prevQuad

if(inst == 'GotoT'):

pending = [contador] + pending

if(inst == 'StartLoop'):

jumps = jumps + [contador + 1]

if(inst == 'EndOfLoop'):

jumps = jumps + [contador + 1]

pend = pending.pop()

prevQuad = diccionario[pend]

jumpTo = jumps.pop()

prevQuad += [jumpTo]

diccionario[pend] = prevQuad

pend = pending.pop()

prevQuad = diccionario[pend]

jumpTo = jumps.pop()

prevQuad += [jumpTo]

diccionario[pend] = prevQuad

diccionario[contador] = quad

contador += 1

return diccionario

El diccionario resultante se regresa al analizador de funciones o roma, ya que estos son los que tienen los bloques.

**3.13 Analizador Expresiones**

Parecido al analizador de expresiones especiales, pero este no tiene restricciones que solo permiten enteros o evitando paréntesis, el archivo “analizadorExpresiones” es llamado cuando el analizador del bloque se topa con estatutos que hacen uso de expresiones (que son la mayoría), como resultado regresan una lista que representa la expresión, para activar este archivo se debe mandar el hijo del estatuto que representa la regla “expression” para reglresar la lista, un ejemplo resultante sería, para una asignación: se guarda el id y el operador “=”, y se recibe una lista como esta [1, \*, 2, +, (, ID, -[-, 2, \*, 3, -]-, +, 5, )].

Esta lista se manda al archivo “analizadorCuadruplosExp” que regresa una lista de cuádruplos obtenido por la expresión. Por ejemplo esta función:

def an\_assignment(assignment):

global tree

global pila

global contador

hijos = aA.gimmeTheChildren(assignment, tree)

identifier = hijos[0]

expression = hijos[2]

pilaExp = aE.init(expression, tree)

aE.pila = []

ID = an\_identifier(identifier)

stack = ID + ['='] + pilaExp

quads = aCE.init(pilaExp, contador)

aCE.pila = []

cuadruplos = quads[0]

IDFinal = quads[1]

if(isinstance(IDFinal, list)):

if(len(IDFinal) == 1):

IDFinal = IDFinal

else:

IDFinal = IDFinal[0][2]

IDFinal = [IDFinal]

contador = quads[2]

pila += cuadruplos

finalQuad = ['='] + ID + IDFinal

pila += [finalQuad]

Esta función es proveniente del analizador del bloque de una función o el programa principal, cuando se llega a un estatuto de asignación, se llama al archivo “analizadorExpresiones” con el alias aE, mandándose como parametro el nodo que inicia el análisis de una expresión, esto regresa la liista o pila como el ejemplo que se mostró hace un momento. Esta pila no nos puede servir de mucho ya que el compilador resuelve todo con cuádruplos, para ello se ocupa el archivo que hablaremos a continuación.

**3.14 Analizador de cuádruplos para expresiones**

El archivo “analizadorCuadruplosExp” tiene como propósito traer una pila de cuádruplos que representan la resolución de una expresión, esta expresión puede tener origen un estatuto que haya requerido analizar una expresión, como lo es una asignación, un estatuto “if”, un ciclo, etc. todos estos regresan una pila de la expresión tal y como la escribió el usuario en su archivo de prueba, teniendo la lista se analiza teniendo en cuenta muchas cosas antes de crear un cuádruplo, por ejemplo si hay prioridad de operador, como \* contra +, si hay un símbolo ‘(‘ que indica mayor prioridad, porque los paréntesis son prioridad sobre los operadores, y también el símbolo “-[-“, que es indica como un paréntesis una etiqueta de mayor prioridad, ¿por qué?, cuando se encuentra un corchete, significa que hay un índice para el arreglo de unan variable, para esto se le da prioridad a crear los cuádruplos dentro de los corchetes antes de una operación regular, entre los paréntesis y los corchetes son de la misma priioridad, pero el orden en que se toman en cuenta en este analizador es el que determina la prioridad entre esos dos. Un ejemplo resultante de lo que saca este archivo es:

[[+, Id, Id, t1], [-, t1, Id, t2]]

Como podemos ver, este archivo también genera temporales, pero no se les asigna un objeto como las variables, estos temporales solos ervirán como identificador para el resultado de un cuádruplo, el contador de los temporales, junto con los cuádruplos es regresado al archivo “analizadorExpresiones” con el propósito de ir acumulando este contador en otros estatutos.

**3.15 Analizador de Llamadas**

Las llamadas a funciones son un estatuto especial, porque dentro contienen expresiones de tamaño n (n siendo la cantidad de parametros), es como si fuera un pequeño bloque, por ello se analiza con un archivo especial, pero claro que, se debe aclarar que no se sobreescribe ningún código, si es necesario, el archivo “analizadorLlamada” hará uso del “analizadorExpresiones” para ser más eficiente. Todo el archivo se escribió de esta forma:

def an\_callRecursive(callRecursive):

global tree

global pila

hijos = aA.gimmeTheChildren(callRecursive, tree)

expression = hijos[0]

callAux = hijos[1]

stack = aE.init(expression, tree)

pila += [stack]

an\_callAux(callAux)

def an\_callAux(callAux):

global tree

global pila

hijos = aA.gimmeTheChildren(callAux, tree)

hijo = hijos[0]

value = an\_label(hijo)

if(value == "expression"):

expression = hijo

stack = aE.init(expression, tree)

stack = stack

pila += [stack]

elif(value == "callRecursive"):

callRecursive = hijo

an\_callRecursive(callRecursive)

def init(callAux, lista):

global tree

global pila

tree = lista

an\_callAux(callAux)

stack = pila

pila = []

return stack

Este archivo regresa los cuádruplos de las expresiones dentro de los parámetros en la llamada de una función.

**3.16 Otro analizador de expresiones**

De entre los archivos existe otro que no he mencionado que es el “analizadorExpresionesVar”, si ven estas últiimas letras son lo único que lo diferencia de mi analizador anterior, y eso es porque este archivo analiza las reglas de siintaxis para expresiones que terminan en “Var”, que son una cantidad considerable; esas reglas de sintaxis se hicieron para la creación de variables locales, ya que son a partir de estas que se siguen estas reglas específicas de expresiones, lo que las diferencia es que no tiene restricciones tan estrictas, además de que permite que se repitan los identiificadores con las variables globales.

**3.17 Cubo semántico**

Para la validación semántica de los cuádruplos se genero un cubo semántico que es prácticamente un diccionario tiene diccionarios dentro, la forma en que se escribió fue pensando la manera más sencilla de saber si una operación era válida, esto asumiendo que para cierto punto se tuviera el tipo de variable o temporal que trata, y la forma en que se accesan los diccionarios es rápida y sencilla, lo que se buscaba era que se trajera un valor booleano que me diga que la operación es semánticamente correcta, por ejemplo:

sePuede = cuboSemántico[operador][dato1][dato2][0]

if(sePuede):

resolver expresión

else:

print(“Operación no válida”)

cortar ejecución

El siguiente código muestra un fragmento del cubo semántico:

cuboSemantico = {

'+' : {

"int" : {

"int" : [True, "int"],

"float" : [True, "float"],

"char" : [False],

"string" : [False]

},

"float" : {

"int" : [True, "float"],

"float" : [True, "float"],

"char" : [False],

"string" : [False]

},

"char" : {

"int" : [False],

"float" : [False],

"char" : [True, "string"],

"string" : [True, "string"]

},

"string" : {

"int" : [False],

"float" : [False],

"char" : [True, "string"],

"string" : [True, "string"]

}

},

**3.18 Millenium**