Instituto Tecnológico y de Estudios Superiores de Monterrey

**Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 501)**

**A blue and black logo

Description automatically generated**

**Proyecto de compiladores**

Leonardo Mojica Amézquita - A00571960

14/05/2024

Este proyecto consiste en generar un lenguaje de programación básica.

Se basa en el siguiente diagrama de “Little\_Duck”.

A diagram of a computer program

Description automatically generated**Lista de tokens:**

Símbolos:

* (): paréntesis para encerrar expresiones, y para el uso de funciones.
* {}: llaves que se usan para dar inicio y fin de un “body”.
* []: Corchetes que se usan para encerrar la lógica de una función.
* ;: punto y coma que sirve como un símbolo terminador, el cual se pone cada que termines una asignación de id, llames a una función, declares una función, termines un ciclo, hagas una condición, asignación, etc.
* +: símbolo de suma, se usa para sumar variables.
* -: simbol de resta, se usa para restar variables.
* \*: asterisco que se usa como símbolo de multiplicación.
* /: símbolo que se usa para división.
* =: símbolo de asignación.
* ,: coma, se usa como separador, por ejemplo, para declarar varias variables, en una sola línea, o para poner los parámetros de una función.
* <: menor que, símbolo para comparar dos expresiones.
* >: mayor que, símbolo para comparar dos expresiones.
* !=: símbolo diferente de para comparar dos expresiones
* :: dos puntos que van entre id y su tipado, se usa para especificar el tipado de una variable.

Palabras reservadas:

* program: indica el inicio del programa
* main: indica el inicio del bloque de código principal de tu programa
* end: indica el final del main
* var: indica el inicio de una declaración de variable
* int: indica que tu variable es un entero
* float: indica que tu variable es flotante
* print: función reservada para mostrar información en consola
* if: indica el inicio del condicional if
* else: indica el inisio de lo que se hace en caso de que no se cumpla el if
* void: indica el inicio de una función
* while: indica el inicio de un ciclo while, donde se pondrá la condicional
* do: indica el inicio del cuerpo del while donde se indicará lo que se hará mientras el while se cumpla

Tokens con regex:

En estos tokens uso el separador | como or para especificar las posibles opciones de rangos, pero esto no se suele usar, tengo pensado usar ANTLR4 para desarrollar el compilador, y este puede generar conflictos a la hora de crear las expresiones regulares, así que se deben quitar los |.

* id
  + El id tiene que empezar con una letra (mayúscula o minuscula) y puede seguir con 0 o n letras más (mayúscula o minúscula), o 0 o n números o 0 o n guiones bajo, en cualquier orden.
  + [A-Z|a-z] [A-Z|a-z|0-9|\_]\*
* cte.string
  + una cadena de caracteres (cualquier carácter menos salto de linea), la cual esta encerrada por dos comillas dobles (“”), o dos comillas sencillas (‘’).
  + ("[^\n|"]\*") | ('[^\n|']\*'), el símbolo ^ representa todos los caracteres menos la lista de caracteres a continuación.
  + Esta expresión tiene que ser reescrita para que funcione en antlr4, la cual queda de la siguiente forma: ’ (~[\n]\*) ’ | ” (~[\n]\*) ”, ambos lados del or | son iguales solo cambiando las comillas, para que acepte tanto comillas simples como sobles, adentro cambiamos a ~[\n], el símbolo ~ es un negador, por lo que está diciendo que no acepte lo que esta dentro de los corchetes, que es un salto de línea.
* cte.int
  + Toma valor de números enteros decimales.
  + [0-9]+
* cte.float
  + Únicamente acepta valores numéricos seguidos de un único punto (.) seguido de 1 o más caracteres numéricos
  + [0-9]+ (.) [0-9]+

**Gramática libre de contexto**

<Program> 🡪 program id ; <vars> <a\_funcs> main <BODY> end EOF

<a\_funcs> 🡪 ε | <FUNCS> <a\_funcs>

<vars> 🡪 var <list\_vars> | ε

<list\_vars> 🡪 <list\_id> : <TYPE>; <more\_vars>

<more\_vars> 🡪 ε | <list\_vars>

<list\_id> 🡪 ID <more\_ids>

<more\_ids> 🡪 , list\_id | ε

<TYPE> 🡪 int | float

<Body> 🡪 {<List\_statements>}

<List\_statements> 🡪 ε | <STATEMENT> <List\_statements>

<STATEMENT> 🡪 <ASSIGN> | <CONDITION> | <CYCLE> | <F\_Call> | <Print>

<Print> 🡪 print ( <List\_Expresion> );

<List\_Expresion> 🡪 <exp\_o\_string> |<exp\_o\_string> , <List\_Expresion>

<exp\_o\_string> 🡪 <EXPRESION> | cte.string

<ASSIGN> 🡪 id = <EXPRESION> ;

<CYCLE> 🡪 do <BODY> while ( <EXPRESION> ); este es diferente que en el diagrama, porque está mal

<CONDITION> 🡪 if (<EXPRESION>) <Body> <ELSE>;

<ELSE> 🡪 else <Body> | ε

<EXPRESION> 🡪 <EXP> < comparar\_exp>

<comparar\_exp> 🡪 <comparacion> <EXP>| ε

<comparacion> 🡪 < | > | !=

<EXP> 🡪 <TERMINO> <List\_terminos>

<List\_terminos> 🡪 <sum\_rest> <TERMINO><List\_terminos>| ε

<sum\_rest> 🡪 + | -

<TERMINO> 🡪 <FACTOR><List\_Factor>

<List\_Factor> 🡪 <mult\_div><FACTOR><List\_Factor> | ε

<mult\_div> 🡪 \* | /

<FACTOR> 🡪 (<EXPRESION>) | <sum\_rest><id\_cte> |<id\_cte>

<id\_cte> 🡪 id | <CTE>

<CTE> 🡪 cte\_int | cte\_float

<FUNCS> 🡪 void id (<params >) [<VARS><BODY>];

<params> 🡪 list\_params | ε

<list\_params> 🡪 id : <TYPE> <more\_params>

<more\_params> 🡪 ε | , list\_params

<var\_no\_var> 🡪 ε | <VARS>

<F\_Call> 🡪 id (f\_list\_expresion);

<f\_list\_expresion> 🡪 ε | <EXPRESION> <f\_more\_expresion>

<f\_more\_expresion> 🡪 ε | , <EXPRESION> <f\_more\_expresion>

**Opciones de herramientas de generación automática de compiladores**

Estas herramientas fueron escogidas para poder hacer la implementación en Python.

Encontré el siguiente git, el cual muestra que herramientas de Python puedo usar para todas las etapas de un compilador. <https://gist.github.com/dimitrio-m/cf6cde052787ed97164fe6422a5e4cb0>

3 de las tecnologías que encontré para el análisis léxico y sintáctico es PLY (Python Lex-Yacc), ANTLR y Lark.

PLY:

Es una implementación fiel a lex and yacc. El principal objetivo de PLY es ser lo

más fiel posible la forma en la que herramientas lex/yacc funcionan, como soportar LALR (look-ahead, left to right, rightmost derivation parser) parser y proveer validación de input, reporte de errores y diagnósticos.

PLY cuenta con dos módulos, lex.py y yacc.py dentro del paquete de ply. Lex.ply se usa para el análisis léxico seccionando el input del texto en colecciones de tokens, y el yacc.py se usa para reconocer el lenguaje sintáctico que se debió haber especificado cono una gramática libre de contexto. En general ambos módulos están diseñados para trabajar en conjunto, por ejemplo lex.py tiene una interface en forma de una función token() la cual regresa el próximo token valido del input. Yacc.py normalmente regresa un AST (abstract syntax tree), pero el desarrollador puede cambiarlo para ser implementado como un “simple one-pass compiler”.

ANTLR:

Es un generador para procesar texto estructurado o archivos binarios. Es usado para en base a una gramática, generar un parser que pueda crear y caminar un árbol de parse. ANTLR está dividido en dos partes, el ANTLR tool y la ANTLR runtime libraary. La primera corre en JAVA y genera un parser en base a una gramática. El segundo es la parte que nos ayuda a generar el parser en Python.

LARK:

LARK es una librería de parseo para Python, el cual puede parsear basado en una gramática libre de contexto. Acepta gramáticas basadas en EBNF (Extended Backus-Naur Form), cuenta con algoritmos de parseo de árbol (Earley, LALR and CYK), crea de forma automática un árbol inferido de la gramática, y cuenta con un analizar léxico que soporte expresiones reculares.

**Herramienta ANTLR4 (para python)**

Decidí usar ANTLR4, ya que al haberlo probado un poco y al leer su documentación me di cuenta de que es sencillo de usar y me ofrece muchas herramientas para validar mis gramáticas, por ejemplo, puedo pedirle que me muestre de forma gráfica un árbol de mi input de la gramática.

Para el uso de la herramienta se uso la documentación del git oficial de antlr: <https://github.com/antlr/antlr4/blob/master/doc/getting-started.md>

Para el uso de antlr se necesita tener instalado Python y pip.

En la línea de comandos se debe ejecutar

* pip install antlr4-tools

En Windows puede haber problemas de no poder ejecutar los comandos de antlr4 y es porque la dirección de la herramienta no suele agregarse al path de las variables de sistema, por lo que se debe agregar manualmente. Al intentar ejecutar antlr4 debería salir un error diciendo que agregues la dirección al path, pero si no llega a salir el path que se debe agregar es c:\Users...\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.10\_qbz5n2kfra8p8\LocalCache\local-packages\Python310\Scripts

Después de agregar el path debes reiniciar la computadora.

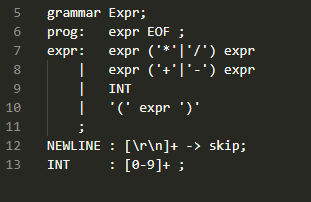
Cuando reinicies ejecuta en terminal.

* Antlr4

Si no tienes java instalado te preguntara si quieres instalarlo, debes dar que si ya que antlr necesita java para correr.

Una ves instale todo lo necesario para usar antlr, intente algunos de los ejemplos de la documentación que me sirvieron para decidir usar esta herramienta.

Lo primero que hice fue crear un documento Expr.g4 el cual contiene el ejemplo de la gramática a analizar. El contenido del documento es el siguiente:



Como se puede observer, se debe inicial con el nombre de tu gramática, luego pones las variables con las reglas de cada una de estas expresiones, al inicio se pone la expresión inicial que en el ejemplo es prog la cual consiste de un “expr” seguido del EOF (end of file). La expresión expr muestra nos deja agregar sumas, restas, multiplicaciones o divisiones entre más expr, además expr, puede ser un entero (INT) o estar encerrado entre paréntesis, ANTLR permite poner las expresiones con ors | para dar las diferentes posbiles expresiones. Al final se ponen los tokesn en formato de expresión regular, como podemos ver en el ejemplo se tiene INT el cual acepta símbolos de 0 a 9 una o más veces, además tenemos el token NEWLINE que lo único que hace es tener la lista de terminadores para ignorarlos, o sea que si encuentra \r \n (tab o newline) haga un “skip”.

Con este documento creado, pude ejecutar el comando:

* antlr4-parse Expr.g4 prog

el cual me analiza la gramática contenida en Expr.g4, además, al final del comando, agregue “-gui”, “-tree”, “-tokens -trace”, dependiendo de que comando use puedo conseguir diferentes outputs, por ejemplo, si uso -gui me muestra gráficamente el árbol que se genera.

Una vez corro el comando, me deja poner el input que quiero analizar y después debo dar ctrl+z (“end of input”) para indicar que termine de poner el input, y a continuación me regresara el resultado.

El ejemplo que corri fue 10\*2+20, con la gramática de prueba presentada anteriormente, usando -gui obtuve como resultado:

A screenshot of a computer

Description automatically generated

Como pueden ver se genera el árbol de la expresión que se analizo y muestra el nombre de los tokens. Este mismo ejemplo con -tree regresa:

(prog:1 (expr:2 (expr:1 (expr:3 10) \* (expr:3 2)) + (expr:3 20)) <EOF>)

Que no es más que una forma escrita de mostrar el árbol.

Hay varias opciones que puedo agregar al comando para que me meustre diferente información sobre el análisis sintáctico.

Algo importante a destacar es que el ejemplo que mostré tiene el problema de tener recursión a la izquierda en expr: expr (‘\*’|’/’) expr | expr: expr (‘+’|’-’) expr, ya que la regla gramatical se llama a si misma en su primer término, sin embargo, no hubo problemas con antlr4, lo que significa que este es capaz de funcionar con algunas ambigüedades y recursividad izquierda, por lo que puedo darle una gramática relativamente “sucia” y puede que funcione sin problemas.

Además, antlr4 me permite generar un parser en el lenguaje que yo quiera, que en mi caso es Python. Para hacer esto debo ejecutar el comando:

Para esto use la documentación de la siguiente página: <https://yetanotherprogrammingblog.medium.com/antlr-with-python-974c756bdb1b>

* antlr4 -Dlanguage=Python3 Grammar.g4

Que lo que hace es crear varios archivos de Python y texto como un proyecto el cual es mi parser diseñado con la gramática contenida en el archivo Grammar.g4

Con esto ya tengo un entorno donde puedo desarrollar mi parser en un archivo main.py (para eso segui la guía que se encuentra en la página mencionada anteriormente).

**Diseño de la gramática para antlr4**

Basándome en la gramática de Little duck, se quiere crear un compilador, para eso primero se debe crear el scanner léxico y sintáctico. Como ya mencioné, este se desarrollará usando antlr4 en Python.

El primer paso para este desarrollo es desarrollar un documento .g4 el cuall contiene todas las reglas de la gramática y las reglas léxicas.

Yo cree un documento llamado Grammar\_duck.g4, que contiene lo siguiente:

grammar Grammar\_duck;

prog    : 'program' ID ';' vars a\_funcs 'main' body 'end' EOF;

a\_funcs : (funcs a\_funcs)?;

vars      : ('var' more\_vars)?;

more\_vars : ( list\_id ':' TYPE ';')+;

list\_id   : ID (',' list\_id)?;

body           : '{' list\_statement '}';

list\_statement : (statement)\*;

statement   : assign

            | condition

            | cycle

            | f\_call

            | print;

print           : 'print(' list\_expresion ')' ';' ;

list\_expresion  : exp\_o\_string

                | exp\_o\_string ',' list\_expresion;

exp\_o\_string    : expresion | CTE\_STRING;

assign : ID '=' expresion';';

cycle : 'do' body 'while' '(' expresion ')' ';';

condition : 'if' '(' expresion ')' body else';';

else      : ('else' body)?;

expresion    : exp comparar?;

comparar     : (comparacion exp);

comparacion  : '<' | '>' | '!='

exp           : termino (sum\_rest termino)\*;

sum\_rest      : '+' | '-';

termino     : factor (mult\_div factor)\*;

mult\_div    : '\*' | '/' ;

factor  : '(' expresion ')'

        | positivo\_negativo id\_cte

        | id\_cte;

positivo\_negativo:

'+' | '-'

id\_cte: ID | cte;

cte: CTE\_INT | CTE\_FLOAT;

funcs       : 'void' ID '(' params ')' '[' vars body ']' ';';

params: (list\_params)?;

list\_params: (ID ':' TYPE) (',' list\_params)? ;

f\_call           : ID '(' f\_list\_expresion ')' ';';

f\_list\_expresion : (expresion f\_more\_expresion)?;

f\_more\_expresion : (',' expresion f\_more\_expresion)?;

// TOKENS

SKIPS : [ \r\t\n]+ -> skip ; // skip -->  espacios, enters y tabs

TYPE       : 'int' | 'float';

ID         : [a-zA-Z][A-Za-z0-9\_]\* ;

CTE\_INT    : [0-9]+ ;

CTE\_FLOAT  : [0-9]+'.'[0-9]+ ;

CTE\_STRING : '\'' (~[\n]\*) '\'' | '"' (~[\n]\*) '"' ;

Esto no es más que la gramática explicada anteriormente en el formato necesario para antlr4.

**Scanner desarrollado**

Se diseño un scanner usando antlr4 en Python basado en la gramática creada anteriormente en este documento, todo el código esta como una carpeta .zip adjunta a este documento, recuerda que para correrlo es recomendable tener instalado todo lo necesario (mencionado en la sección “Herramienta ANTLR4 (para python)”). Si no se tiene el .zip, en una carpeta nueva y crear un archivo “Grammar\_duck.g4”, y ejecutar los comandos para crear el proyecto en Python, después copiar el código explicado más adelante en los archivos necesario.

En una carpeta tengo el documento Grammar\_duck.g4, el cual contiene la gramática especificada en la sección anterior.

En la terminal, estando en la carpeta con el archivo g4, que es donde va a estar todo el proyecto, ejecuto el siguiente comando:

* antlr4 -Dlanguage=Python3 Grammar\_duck.g4

Esto commando genera todo el entorno necesario para crear el proyecto en python3. A continuación se puede ver el resultado, de los archivos que se crean al ejecutar el programa.

A screen shot of a computer

Description automatically generated

Después se debe crear en esta misma carpeta un archivo main.py, donde debe estar el código del parser.

Main.py:

# imports con lo necesario para

# antlr4 y la gramatica

from antlr4 import \*

from Grammar\_duckLexer import Grammar\_duckLexer

from Grammar\_duckParser import Grammar\_duckParser

# Funcion para leer el archivo

# regresa un dict con status que puede ser OK (todo salio bien)

# o ERROR, ademas regresa "content" que tiene el contenido del error

# o el contenido del archivo que se leyo con exito

*def* readFile(*path*):

    fileContent = "" # variable que guardara el contenido del archivo

    if not *path*.endswith('.txt'):

        return {"status":"ERROR", "content": "Must be a .txt file"}

    try:

        with open(*path*, "r") as file: # abrimos el archivo en modo lectura

            fileContent = file.read() # leemos el archivo para guardarlo en la variable

        return {"status":"OK", "content": fileContent}  # se leyo con exito

    except FileNotFoundError:

        return {"status":"ERROR", "content": "ERROR: File not found."} # no se encontro el archivo

    except Exception as e:

        return {"status":"ERROR", "content": "ERROR: " + str(e)} # cualquier otro error

if \_\_name\_\_ == '\_\_main\_\_': # programa main

    # Se pide un path con el archivo a analizar con la gramatica

    print("program path: ")

    input\_path = input("> ")

    file = readFile(input\_path)

    if file["status"] == "ERROR":

        print(file["content"])

    elif file["status"] == "OK":

        content = file["content"]

        lexer = Grammar\_duckLexer(InputStream(content)) # pasamos el doc por el lexer

        stream = CommonTokenStream(lexer) # pasamos el resultado del lexer al stream

        parser = Grammar\_duckParser(stream) # pasamos el stram al parser

        tree = parser.prog()

        print(tree.toStringTree(*recog*=parser))

En este codigo, primero importamos todo lo necesario para usar nuestra gramática, que es la librería de antlr y los archivos Grammar\_duckLexer y Grammar\_duckparser que nos generó el comando “antlr4 -Dlanguage=Python3 Grammar\_duck.g4”.

Después definí una función readfile la cual se encarga de leer un archivo en base a su path, guarda el contenido del archivo para retenerlo y usarlo posteriormente para el procesamiento.

Después tenemos el bloque de main, en el cual pedimos un input del usuario (el path del archivo), llamamos a la función que lee el archivo, revisamos si esta nos regresó un error, para parar el código y mostrar el error, y si no hubo error, ejecutamos las líneas de código que escanean el archivo basando en la gramática.

**Test-plan para el funcionamiento**

A continuación, voy a generar una serie de casos de prueba donde muestro el número de prueba, el texto que se va a probar con mi gramática, el resultado esperado, y el resultado real.

|  |  |  |  |
| --- | --- | --- | --- |
| # | Texto de prueba | Resultado esperado | Resultado real |
| 1 | program Test1 ;  var id1, ID\_2, iD\_\_3 : int ;  var Id4 : float ;  void func1 () [  { }  ];  void func\_2 ( f\_id\_1 : int, fiD02:float)[  var f\_var\_1 : int ;  {  id1 = (ID\_2 + iD\_\_3) \* 43.24 - 4212 ;  f\_var\_1 = 2333 ;  if ( f\_var\_1 != (2 + id1) ) {  x = 3 ;  }  else {  x = 2 ;  };  do{  iD\_\_3 = iD\_\_3 + 1 ;  }while( iD\_\_3 < 4 );  print('Hola 7&76&&-', Id4);  func1();  func1(2,Id4);  }  ];  main{  y = 3 + 4;  }  end | Aceptar todo. (que regrese el árbol sin errores) | Acepto y creo el arbol |
| 2 | program Test2 ;  main{  y = 3 + 4;  }  end | Ejemplo minimo, quitando var y funciones.  Debería ser aceptado | aceptado |
| 3 | program Test3 ;  main{  y = 3 + 4  }  end | Quitamos el ; en el main  Debería der un error | Error  line 6:0 missing ';' at '}' |
| 4 | program Test6 ;  var id1, ID\_2, iD\_\_3 : int ;  void func\_2 ( f\_id\_1 : int, fiD02:float)[  var f\_var\_1 : int ;  {  polas$$ = 3 ;  }  ];  main{  y = 3 + 4;  }  end | Crear en func\_2 un id con símbolos no validos  Se espera un Error | Error  line 13:13 token recognition error at: '$'  line 13:14 token recognition error at: '$' |
| 5 | program Test5 ;  var id1, ID\_2, iD\_\_3 : int ;  main{  print('hola);  }  end | Print sin cerrar comillas  Error en el print | Error  line 6:10 token recognition error at: ''hola);\n'  line 7:0 mismatched input '}' expecting {'+', '-', '(', ID, CTE\_INT, CTE\_FLOAT, CTE\_STRING} |
| 6 | program Test6 ;  var id1, ID\_2, iD\_\_3 : int ;  main{  print('"hola"');  }  end | Print que usa comillas dobles dentro de comillas simples.  Los strings pueden tener comillas diferentes a los que se usen, o sea ‘ ” ” ’, o “ ‘ ‘ ”  Se acepta | Acepta |

**Semántica de variables**

**Creación del cubo semántico**

Un cubo semántico es una estructura de datos que puede ser definido como un arreglo de 3 dimensiones, el propósito de esta estructura de datos es guardar que resultado tiene que dar cada operación según el tipo de dato de sus operadores y de la operación que se está usando.

Para este proyecto solo tenemos 2 tipos de datos, los cuales son entero (int) y flotante (float), y tenemos un total de 7 operaciones, suma, resta, multiplicación, división, mayor que, menor que, y diferente que. En la lógica de mi compilador, cualquier operación no booleana donde al menos uno de sus operandos sea de tipo float, va a dar como resultado un float. Todas las operaciones con dos ints me darán como resultado un int, y todas las operaciones booleanas me darán como resultado un int (que puede ser 0 o 1).

Un ejemplo en dos dimensiones de nuestro cubo es la siguiente:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | + | - | \* | / | < | > | ¡= | = |
| Int, int | int | int | int | int | int | int | Int | int |
| Int, float | Float | Float | Float | Float | Int | int | Int | Int |
| Float, int | Float | Float | Float | Float | Int | Int | Int | float |
| Float, float | Float | Float | Float | Float | int | Int | int | float |

En nuestro código lo voy a tener como una constante global, para que pueda ser de fácil acceso para mi compilador para poder hacer las operaciones. La estructura como tal va a ser un arreglo de tres dimensiones donde cada índice representa una operación o tipo de variable respectivamente. A continuación, podrás ver los índices.

0 = flotante, 1 = entero.

0 = suma, 1 = resta, 2 = multiplicación, 3, = división, 4 = menor que, 5 = mayor que, 6 = diferente que, 7 = asignación.

En el arreglo es el primer y segundo índice son el primer y segundo operando respectivamente, y el tercer índice es el operador.

Teniendo como ejemplo cuboSemantico[0][1][4], es como decir que tenemos un flotante, un entero y un menor que, que es como decir flotante menor que entero.

El código está en un nuevo documento Semántica, el cual contiene una clase donde va a estar la implementación del análisis semántico.

Resumiendo, un poco el código que se tendrá en esta clase. Solo abra una clase llamada Semántica que contendrá todos los atributos y métodos necesarios para el análisis semántico. En esta clase tenemos un cubo semántico el cual lo manejaremos como un arreglo de tres dimensiones, y guardara el valor que se espera al hacer una operación con dos variables diferentes. Como es un arreglo solo podemos acceder a treves de índices (valores numéricos), cada uno de estos representa una operación o tipo de dato, y lo que se guarda en este es el tipo de dato que se espera de la operación.

  # 0 = float, 1 = int, 2 = string

        # 0 = + , 1 = -, 2 = \*, 3 = /, 4 = <, 5 = >, 6 = !=, 7 = =

        # ejemplo cubo[0][1][2] = float, int, multiplicacion

Lo que sigue es definir el directorio de funciones, el cual nos va a guardar las variables tanto globales como locales por cada función.

La estructura de datos que tenemos que usar tiene que ser buena/eficiente para hacer búsquedas, por eso mismo, pienso usar un diccionario. El directorio de función va a hacer un diccionario donde en key, se guardará el nombre de la función y en type lo que retorna, que por como esta diseñado nuestro lenguaje siempre guardará void (ya que solo aceptamos funciones void). Además, guardaremos en una ultima columna un directorio de variables

El directorio de variables es para guardar todas las variables dentro de esa función, este directorio tendrá como key en nombre de la variable y como valor un arreglo, donde por ahora solo guardaremos el tipo de dato (que solo puede ser int o float), lo hago un arreglo por si en un futuro queremos guardar mas información como valor de la variable o dirección de memoria.

Este directorio de funciones es simplemente un diccionario de Python, el cual se vera de la siguiente forma.

        # ==========Directorio de funciones============

*self*.dirFunc = {}

        # dirFunc = { "func" : ["type", {"var1":["type", memoria]}]}

En esta guardaramos como key la función y como valor un arreglo donde en la posición 0 tenemos el tipo de función (que en nuestro programa solo puede ser void o program para global), y en la posición 2 tenemos otro diccionario el cual guardara las variables, en estas variables la key es el nombre de la variable y el valor es un arreglo donde se guarda el tipo de variable (float o int) y el valor en memoria que tendrá.

Además de estas estructuras tendremos las siguientes variables para ayudarnos más adelante en el proceso de llenado del directorio y en general en el análisis semántico y creación de código intermedio.

*self*.currFunc = "" # funcion actual

*self*.tempIDS = []  # arreglo de memoria de lista de vars

*self*.currID = ""   # id actual

*self*.currType = "" # tipo de la variable actual

Ahora hay que definir los puntos neurálgicos donde se ejecutaran nuestras acciones semánticas.

Por ahora solo vamos a definir los puntos y neurálgicos necesarios para hacer nuestro directorio de funciones.

Mostrando primero los puntos neurálgicos, que ocupamos en el programa principal:

A diagram of a type of machine

Description automatically generated

Con las siguientes acciones semánticas:

1. Creamos nuestro directorio de funciones (instanciamos la clase DirectorioFunciones), y guardamos en una variable temporal “currType”, el valor “program”.
2. Revisamos que el id de la función no esté en el directorio de funciones, si esta, regresar error de función ya declarada, si no está, agregar el id con currType.
3. Creamos un arreglo de memoria temporal para guardar la lista de variables, si ya existe lo vaciamos (tempIDS).
4. Checamos si el id está en el directorio (si ya está guardado), si ya está guardado mostramos error de variable ya declarada. Si no, agregamos el id a nuestro arreglo de memoria.
5. Guardamos toda la lista de id (currIDS) y con el tipo “Type”.
6. Borramos todo del directorio de funciones

A diagram of a type of text

Description automatically generated

1. Poner currType = “void”
2. Hacer la acción 2
3. Guardar id en una variable temporar currID, esperamos a intentar guardarla cuando tengamos type.
4. Si el id ya existe regresar error, si no existe, Guardar variable en mi directorio de variables de mi currFunction, con el tipo que tenemos y con currID.
5. Eliminar el directorio de variables de nuestra función (ya terminamos la función)

La implementación de estos puntos neurálgicos esta en el archivo Grammar\_duck.g4 en los anexos, este archivo tiene la gramática y código en Python implementando estos puntos neurálgicos, además todas las funciones que se lleguen a usar en estos puntos, son métodos de la clase semántica la cual se encuentra en el archivo Semantica.py.

Lo que sigue es hacer los puntos neurálgicos para conseguir el código intermedio el cual será representado en cuádruplos.

Recordando, un cuádruplo está organizado de la siguiente forma:

(OP, ARG1, ARG2, RESULT).

Donde OP es la operación, y ARG son los argumentos.

Como tal todo lo guardaremos en formato numérico lo que representa la dirección de memoria virtual de cada variable (ARG y RESULTADO). También es importante tomar en cuenta que podemos manejar constantes, o sea, podemos hacer una suma de A + 2, y en los cuádruplos guardaríamos la dirección de memoria, sin embargo, como las variables ya están en formato numérico, puede haber confusión entre una constante o un espacio de memoria, para esto, también guardaremos las constantes en un espacio de memoria, en el proceso de análisis semántico guardaremos estas constantes en nuestro directorio de variables, de la misma forma que lo hacemos con las variables. Además, editaremos nuestro directorio de variables, ya que ahora necesitamos guardar la dirección de memoria de cada cosa en este diccionario, ya que actualmente solo guardamos el tipo.

Primero definiremos como guardaremos cada elemento del cuádruplo en este formato numérico.

Primero, cada operación debe tener un número. Siguiendo la lógica de nuestro cubo semántico, y agregando las demás operaciones que necesitaremos.

|  |  |  |
| --- | --- | --- |
| **Operación** | **descripción** | **Valor numérico** |
| + | Suma | 0 |
| - | Resta | 1 |
| \* | Multiplicación | 2 |
| / | División | 3 |
| < | Menor que | 4 |
| > | Mayor que | 5 |
| != | Diferente que | 6 |
| = | Asignación | 7 |
| GoTo | Saltar a otro cuadruplo | 8 |
| GoToF | Saltar si falso (valor 0) | 9 |
| GoToT | Saltar si verdadero (valor 1) | 10 |
| Print | Mostrar en consola | 11 |
| EOF | Finalizar programa | 12 |

Como ya mencioné, nuestras variables estarán almacenadas en un espacio de memoria en nuestra máquina virtual, por lo que se definirán rangos de valores dependiendo de con que variable estemos tratando (ya sea constante, variable local, o variable global).

Los rangos de memoria serán:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Variable** | **Rango de memoria** | **Rango de Int** | **Rango de Float** | **Rango de string** |
| Global | [1000, 12999] | [1000, 6499] | [6500, 12999] | NaN |
| Local | [13000, 18999] | [13000, 15999] | [16000, 18999] | NaN |
| Temporal | [19000, 45999] | [19000, 32499] | [32500, 45999] | NaN |
| Constante | [46000, 65999] | [46000, 55999] | [56000, 65999] | [66000, 75999] |

Esto se implementa como nuevos atributos de nuestra variable de la siguiente forma:

        # ==========direcciones de memoria ===========

*self*.rangoMGlobalI = 6499; *self*.rangoMGlobalF = 12999  # valor maximo variable global

*self*.rangoMLocalI  = 15999; *self*.rangoMLocalF = 18999 # valor maximo variable local

*self*.rangoMTempI   = 32499; *self*.rangoMTempF = 45999 # valor maximo variable temporal

*self*.rangoMCteI    = 55999; *self*.rangoMCteF = 65999; *self*.rangoMCteS = 75999 # valor maximo para constantes

        # limite inferior

*self*.minMGlobalI = 1000; *self*.minMGlobalF = 6500

*self*.minMLocalI  = 13000; *self*.minMLocalF = 16000

*self*.minMTempI   = 19000; *self*.minMTempF = 29000

*self*.minMCteI    = 46000; *self*.minMCteF = 56000; *self*.currMCteS = 66000

        # contador para asignar memoria

*self*.currMGlobalI = 1000; *self*.currMGlobalF = 6500

*self*.currMLocalI  = 13000; *self*.currMLocalF = 16000

*self*.currMTempI   = 19000; *self*.currMTempF = 29000

*self*.currMCteI    = 46000; *self*.currMCteF = 56000; *self*.currMCteS = 66000

Además, necesitamos nuevas estructuras de datos para los algoritmos para llenar nuestra lista de cuádruplos. Los cuádruplos los guardaremos en un arreglo, esto es útil ya que en acciones como GoTo podemos poner el índice del arreglo para hacer el salto de forma rápida.

Los cuádruplos se guardarán en forma numérica, donde cada operación tiene un numero asignado y cada operador tiene un valor numérico el cual es su dirección de memoria, por esto mismo no podemos poner constantes numéricas como valores ya que los cuadruplos podrían ser confusos y seria prácticamente imposible distinguir entre la constante 1005 y la dirección de memoria 1005, guardaremos las constantes en direcciones de memoria y para esto usaremos un nuevo diccionario donde la key es la constante (que puede ser entera, flotante o string) y el valor es la memoria.

Además tendremos una pila de variables y una pila de operadores, estas pilas son necesarias para los algoritmos de creación de cuádruplos ya que necesitamos guardar de forma temporal esta información para poder mantener la jerarquía de operaciones de forma correcta (estos algoritmos se explican en los puntos neurálgicos), además necesitamos una pila de tipos la cual ira a la par de la pila de variables ya que esta guarda el tipo de la variable que se encuentra en la pila, esta se usa para hacer la comprobación con el cubo semántico.

Como en Python no existe una estructura de pila como tal, ya que la listas de Python cumplen este papel, por lo que todas mis estructuras de arreglos y pilas serán declarados como listas de Python pero serán manejadas como pilas o arreglo, dependiendo de lo que se necesite.

        # ==========Pilas, operaciones============

*self*.dirCTE         = {} # para guardar constante y direccion de memoria

*self*.currCuadruplo  = 0 # indice del siguiente cuadruplo

*self*.cuadruplos     = [] # arreglo de cuadruplos

*self*.pilaTipos      = [] # pila con los tipos

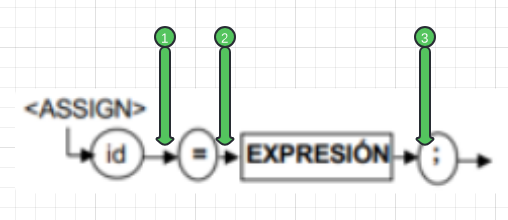
*self*.pilaSaltos     = [] # pila de los saltos

*self*.pilaVariables  = [] # pila con las direcciones de memoria de varibles

*self*.pilaOperadores = [] # pila con operadores

Ahora debemos identificar los puntos neurálgicos y sus acciones para poder crear el cuadruplo de cuádruplos.

Empezaremos con los puntos para expresiones:



1. Revisar que el id existe, si no, regresar error, si sí, guardamos el id en la pila de variables, y su tipo en la pila de tipos.
2. Guardamos el = en la pila de operadores.
3. Sacamos el = de la pila de operadores, sacamos el operando derecho e izquierdo de la pila de variables, y creamos el cuádruplo (=, OPD, ,OPI ).
   1. El valor se guarda en una variable temporal, debemos checar si hay espacio en nuestra memoria temporal, si no, hay que regresar error. Si hay espacio meterlo a la pila de variables.

A diagram of a diagram

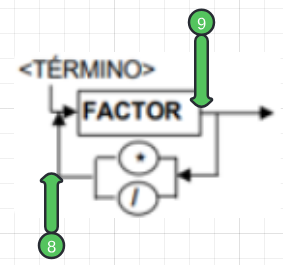
Description automatically generated

1. Guardamos el operador a la pila de operadores.
2. Sacamos el operador de la pila de operadores, junto al operando derecho e izquierdo y creamos el cuádruplo (>, OPI, OPD, temp). Hacer 3.1.

A diagram of a diagram

Description automatically generated

1. Meter el operador (+ -) a la pila de operadores.
2. Revisar si tenemos hasta arriba de la pila de operadores un + o -, si no, continuar, si sí, sacarlo de la pila de operadores, sacar el operando derecho e izquierdo y crear el cuádruplo (+, OPI, OPD, temp). Hacer 3.1.



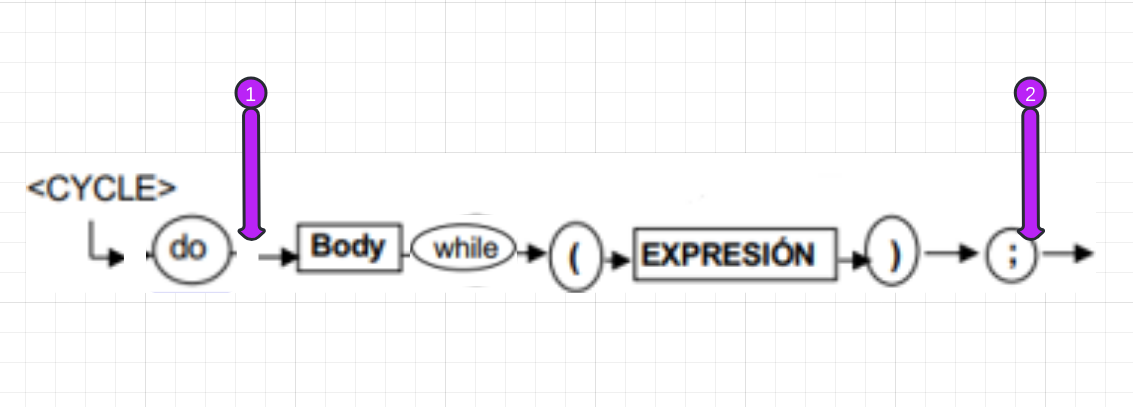
1. Meter operador (\* /) a la pila de operadores.
2. Revisar si tenemos hasta arriba de la pila de operadores un \* p /, si no, continuar, si sí, sacarlo de la pila de operadores, sacar el operador derecho e izquierdo y crear el cuádruplo (\*, OPI, OPD, temp). Hacer 3.1.

A diagram of a complex expression

Description automatically generated with medium confidence

1. Meter barrera (-1) como separador en la pila de operadores.
2. Sacar la barrera de la pila de operadores.
3. Si tenemos un +, no hacer nada, si tenemos un – guardar true en una variable de convertirMenos, esta variable nos dirá más adelante si necesitamos o no multiplicar una variable por -1. ademas agregar una operación \* a mi pila de operaciones, y agregar una constante -1 a mi pila de variables (Antes, debemos guardar el valor -1 como constante en nuestro directorio de memoria si no existe.)
4. Si se tiene un ID, revisar si este existe en nuestro directorio, si no, regresar error, si sí, guardar la dirección de memoria en nuestra pila de variables. Si se tiene una constante, revisar si ya la tenemos declarada en nuestro directorio de variables, si no, crearla y darle una dirección de memoria, una vez exista, meter su dirección de memoria en nuestra pila de variables. Si convertiMenos es true debemos multiplicar por -1 la última variable/constante que metimos a la pila, entonces, sacamos el operador \* de la pila de operadores, y el operador derecho (valor) y el operador izquierdo (-1) de nuestra pila de variables, y crear el cuádruplo (\*,-1,OPD,temp), guardar el valor temporal en nuestra pila de variables.

Ahora sigue crear los puntos neurálgicos para crear los cuadruplos de nuestras funciones no lineales.



1. Metemos el índice del cuádruplo actual a nuestra pila de saltos.
2. Sacamos el índice de la pila de saltos, sacamos la comparación de la pila de variables (el ultimo valor de EXPRESIÓN) y creamos un cuádruplo de GoToT (10), de la siguiente forma: (10, temp, ,indice).

A diagram of a body

Description automatically generated

1. Agregamos el currCuadruplo a la pila de saltos y creamos un nuevo cuádruplo (que estará en esta posicion), de un GoToF (9), para esto hacemos temp = pilaVariables.pop() (conseguimos la condición que tiene que ser falsa), el cuádruplo seria (9,temp, , ), no ponemos el índice del salto aún.
2. Sacamos el valor que tengamos en la pila de saltos y lo guardamos temporalmente. Guardamos el currCuadruplo (else) a la pila de saltos, creamos este cuádruplo GoTo (8), pero aún no lo llenamos con nada (no sabemos a dónde saltar aún), teniendo (8, , , ). Regresamos al cuádruplo que tenemos en el índice de la variable temporal (el if), y lo editamos para que salte a currCuadruplo (que va a ser el cuádruplo que sigue del GoTo que acabamos de hacer) teniendo (9, temp, , salto).
3. Sacamos lo que tengamos en la pila de saltos, regresamos a ese punto (debe ser un GoTo o un GoToF dependiendo si tenemos else o no), y lo editamos para que tenga el salto en currCuadruplo.

A diagram of a function

Description automatically generated

1. Hacemos .pop() a la pila de variables y pila de tipos, hacemos el cuádruplo, print (11), de la siguiente forma (11, , ,temp).
2. Convertimos la cte.string como como dirección de memoria constante de tipo string (revisando que tengamos espacio), y la metemos a la pila de variables, además de meter 2 (string) a la pila de tipos.
3. Hacemos lo mismo que el 6.

Estos puntos neurálgicos se pueden ver en Grammar\_duck.g4 como código de Python, y las funciones usadas se pueden ver como métodos de la clase Semántica.

también vamos a agregar un último cuádruplo al final del programa, justo en el end of file. Este es el cuádruplo end of file el cual es necesario, ya que en casos como whiles o ifs donde no hay nada después de estos, estos dentran GoTo a un cuádruplo que no existe, por lo que agregamos el cuádruplo EOF para que estos GoTo siempre tengan un cuádruplo que agregar.

Este será de la forma:

(EOF,None,None,None), donde la “operación” EOF tomara el valor de 12, Por lo que el cuádruplo será (12,None,None,None)

Para poder correr esto se tiene que volver a correr el comando:

* antlr4 -Dlanguage=Python3 Grammar\_duck.g4

Para actualizar mi entorno de trabajo

Fuentes:

<https://gist.github.com/dimitrio-m/cf6cde052787ed97164fe6422a5e4cb0>

<https://github.com/antlr/antlr4/blob/master/doc/getting-started.md>

<https://yetanotherprogrammingblog.medium.com/antlr-with-python-974c756bdb1b>