UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
Facultad de Ingeniería

GRUPO No 13

MANUAL TECNICO

DESCRIPCION

Es una Aplicacion con interfaz que funciona como interprete del lenguaje SQL especialmente creado para el curso de Organizacion de Lenguajes y Compiladores 2

Para realizar el interprete se utilizo PLY de Python. Tal como se desarrollara a continuacion

DESARROLLO CON LAS TECNOLOGIAS:

- ***Sistema Operativos***: Ubuntu 20.04LTS, Windows 10
- ***Lenguaje de Programacion***: Python
- ***GUI***: Tkinter
- ***IDE***: Visual Studio Code, PyCharm
- ***Graficas***: graphviz

Construccion del Analizador

PLY: es una herramienta de analisis que esta escrita en el lenguaje de programacion python. Es un generador de analizadores y es una implementacion de lex & yacc. Utiliza el analisis LALR(1) que hace muy potente para gramaticas de todo tipo.

Para este proyecto se implementara una gramatica ascendente y descendente simulada, de la cual se detallara mas adelante.

Estructura de PLY

- Ply.lex para realizar el analisis lexico
- Ply.yacc para creacion de analizadores sintacticos

Definición del Analizador Léxico

Para realizar el analizador léxico se crean una serie de tokens y palabras reservadas que serán reconocidas por el analizador.

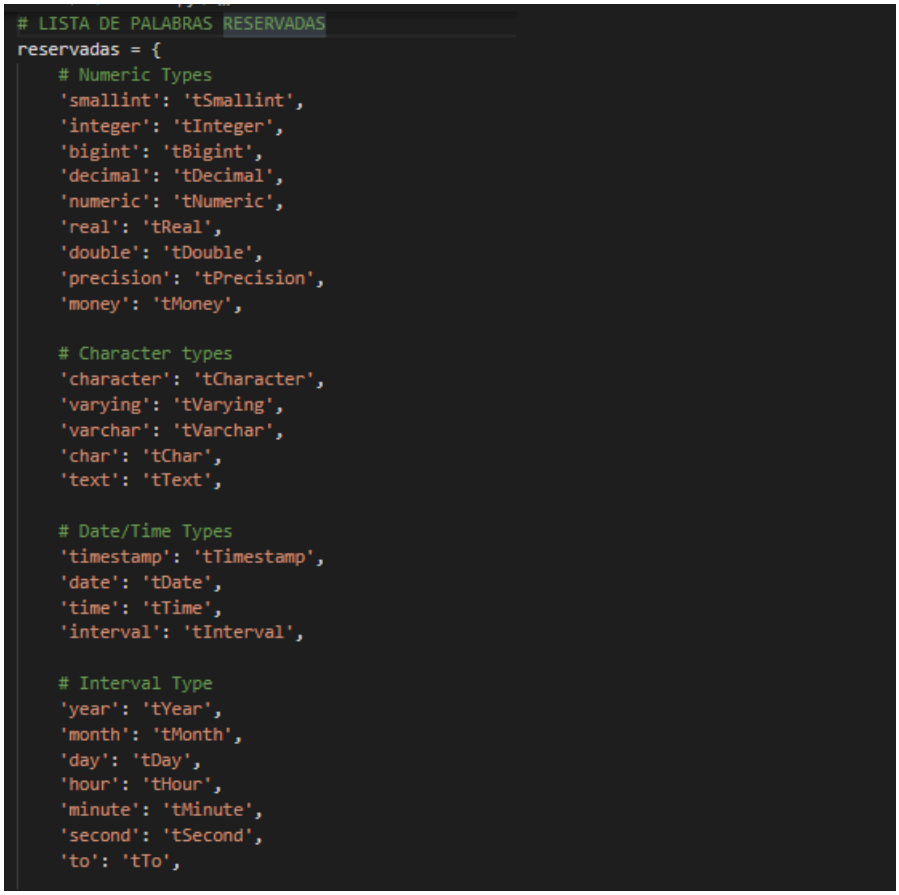
Cualquier otro carácter o expresión no registrada en la definición del analizador, dará como resultado un error y se guardará el error en una lista de errores.

Tokens

Se definen de la siguiente manera. Estos son secuencias de caracteres que serán de utilidad para el análisis sintactico.

En el siguente ejemplo se muestra como se declara los tokens y palabras reservadas
por Ej:

- Aqui se declara palabras reservadas



```
# LISTA DE PALABRAS RESERVADAS
reservadas = {
    # Numeric Types
    'smallint': 'tSmallint',
    'integer': 'tInteger',
    'bigint': 'tBigint',
    'decimal': 'tDecimal',
    'numeric': 'tNumeric',
    'real': 'tReal',
    'double': 'tDouble',
    'precision': 'tPrecision',
    'money': 'tMoney',

    # Character types
    'character': 'tCharacter',
    'varying': 'tVarying',
    'varchar': 'tVarchar',
    'char': 'tChar',
    'text': 'tText',

    # Date/Time Types
    'timestamp': 'tTimestamp',
    'date': 'tDate',
    'time': 'tTime',
    'interval': 'tInterval',

    # Interval Type
    'year': 'tYear',
    'month': 'tMonth',
    'day': 'tDay',
    'hour': 'tHour',
    'minute': 'tMinute',
    'second': 'tSecond',
    'to': 'tTo',
```

- En esta imagen se muestra como se declara los diferentes simbolos

```
# DEFINICIÓN DE TOKENS
t_punto = r'\.'
t_dosPts = r':'
t_corcheI = r'\['
t_corcheD = r'\]'

t_mas = r'\+'
t_menos = r'-'
t_elevado = r'\^'
t_multi = r'\*'
t_divi = r'/'

t_modulo = r'%'
t_igual = r'='
t_menor = r'<'
t_mayor = r'>'
t_menorIgual = r'<='

t_mayorIgual = r'>='
t_diferente = r'<>'

t_parAbre = r'\('
t_parCierra = r'\)'
t_coma = r','
t_ptComa = r';'

# tk_queries
t_barra = r'\|'
t_barraDoble = r'\|\|'
t_amp = r'&'
t_numeral = r'\#'
t_virgulilla = r'~'
t_mayormayor = r'>>'
t_menormenor = r'<<'
```

Otra forma de reconocer Tokens asi como cadenas, numeros, etc .. es de la siguiente forma

- En esta imagen se muestra la forma de declarar cadenas :

```
# DEFINICIÓN DE UNA CADENA PARA LIKE
def t_cadenaLike(t):
    r'\'%.*?%\'|\"%.*?%\"'
    t.value = t.value[2:-2]
    return t


# DEFINICIÓN DE UNA CADENA
def t_cadena(t):
    r'\'.*?\'|\".*?\"'
    t.value = t.value[1:-1]
    return t
```

- En esta imagen se muestra la forma de declarar numeros decimales :

```
# DEFINICIÓN DE UN NÚMERO DECIMAL
def t_decimal(t):
    r'\d+\.\d+'
    try:
        t.value = float(t.value)
    except ValueError:
        print("Floaat value too large %d", t.value)
        t.value = 0
    return t
```

- En este imagen se muestra la otra forma de declarar numero :

```
# DEFINICIÓN DE UN NÚMERO ENTERO
def t_entero(t):
    r'\d+'
    try:
        t.value = int(t.value)
    except ValueError:
        print("Integer value too large %d", t.value)
        t.value = 0
    return t
```

Manejo de Errores Lexicos

Una función de error para manejar los errores léxicos y es donde también se guarda el error en una lista para ser mostrado en el reporte de errores que es generado en la aplicación.

Un error léxico se dispara cuando el o los caracteres no concuerdan con ningún patrón de nuestras expresiones regulares que estamos definiendo.

t.lexer.skip(1) descarta el token.

t.lexer.lineno obtiene el número de línea del token.

```
# HALLAR LA COLUMNA DEL TOKEN ESPECIFICADO
def find_column(token):  # Columna relativa a la fila
    global con
    line_start = con.rfind('\n', 0, token.lexpos) + 1
    return (token.lexpos - line_start) + 1


#  Loading...  ERROR PARA LOS ERRORES LÉXICOS
def t_error(t):
    col = find_column(t)
    # print("Caracter inválido '%s'" % t.value[0], " Línea: '%s'" % str(t.lineno))
    errores_lexicos.append(
        Error(t.value[0], 'Error Léxico', 'El caracter \'' + str(t.value[0]) + '\' no pertenece al lenguaje', col,
              t.lineno))
    t.lexer.skip(1)
```

Construcion de Analizador Lexico

Este analizador es el enviado para realizar el análisis sintáctico.

En la imgen siguiente se muestra como se importan las libreria de analizador Lexico

```
# Construyendo el analizador léxico
import ply.lex as lex
import re
```

Construcion de Analizador Sintactico

Para construir el analizador sintáctico se hará uso del patrón interprete que es mucha utilidad para representar la gramática y realizar la ejecución del código, pero en realida hay dos forma de realizar la ejecucion desde la construccion de arbol con la clase **Node** o con una clase Abstracta, llamada **Sentencias** se mostrara mas adelante de como se puede implementar de la dos formas para poder realizarlo

En este apartado se muestra como se importa la otra libreria para la construcion del analizador sintactico

```
import ply.yacc as yacc
```

Definición de la Gramática en Ply

La forma de declarar producciones en PLY es por medio de una función de Python, seguido de la declaración de

una cadena en comilla simples o triples cuando sea de varias líneas.

Así también se puede incrustar reglas semánticas en cada producción que generarán el AST para posterior ejecución.

Existes dos Tipos de Analisis Sintactico

Analisis Sintactico Ascendente

- Se construye un árbol sintáctico para una cadena de entrada que comienza por las hojas y avanza hacia la raíz
- Se "reduce" una cadena al símbolo inicial de la gramática
- En cada paso de reducción, se sustituye una subcadena determinada que concuerda con el lado derecho de una producción por el símbolo del lado izquierdo de dicha producción (básicamente se busca el lado derecho que corresponde y se reemplaza por el lado izquierdo y se intenta de esa manera llegar a la raíz)
- Se traza una derivación por la derecha en sentido inverso

Analisis Sintactico Descendente

- Se busca una derivación por la izquierda para una cadena de entrada
- Se construye el árbol de análisis sintáctico comenzando desde la raíz y creando los nodos en orden previo
- No hay muchos analizadores sintácticos descendentes con restroceso - casi nunca se necesita el retroceso para analizar las construcciones de lenguajes de programación. El retroceso tampoco es muy eficiente.
- Una gramática recursiva por la izquierda puede causar que el analizador sintáctico entre en un lazo infinito (porque no consume ningún símbolo de entrada)
- Al eliminar la recursividad y factorizar se puede obtener una gramática analizable por un analizador sintáctico predictivo por descenso que no necesite retroceso.

Construcción del Analizador Sintáctico

Para construir el analizador sintáctico se hará uso del patrón interprete con clase abstract llamada Sentencia y de la otra forma es construir el una clase llamada Node sin usar clase abstracta mas adelante se ampliara la informacion .

Definición de la Gramática

La forma de declarar producciones en PLY es por medio de una función de Python, seguido de la declaración de una cadena en comilla simples o triples cuando sea de varias líneas.

Así también se puede incrustar reglas semánticas en cada producción que generarán el AST para posterior ejecución.

Gramatica Ascendente

La gramatica Ascedente son las que se produce por lo general por la Izquierda a diferencia del descedente y aqui no son recomendables usar producciones vacias por lo conflicos que esto puede causar.

```
def p_EXPR_ALTER(t):
    '''EXPR_ALTER : EXPR_ALTER coma alter tColumn id tSet not null ptComa
                  | EXPR_ALTER coma alter tColumn id ttype CHAR_TYPES ptComa
                  | alter tColumn id ttype CHAR_TYPES ptComa
                  | alter tColumn id tSet not null ptComa
                   '''
```

Nota: Se muestra en la figura una porcion de gramatica ASC

Gramatica Descendente

A diferencia de la gramatica ASC sus producciones se van de la derecho y ademas en este tipo de gramatica son permitidas usar producciones vacias, la razon de la misma es para que no lleguen a enciclarse o cause un error de Infinita recursion.

```
def p_EXPR_ALTER(t):
    '''EXPR_ALTER : alter tColumn id tSet not null coma EXPR_ALTER ptComa
                  | alter tColumn id ttype CHAR_TYPES coma EXPR_ALTER  ptComa
                  | alter tColumn id ttype CHAR_TYPES ptComa
                  | alter tColumn id tSet not null ptComa
                  |
                   '''
```

Nota: Se muestra en la figura una porcion de gramatica DESC

Uso de la Pila para Gramática Descendente

Al ser PLY un analizador ascendente, para poder generar una gramática descendente hacemos uso de la Pila que nos provee PLY, accediendo a atributos de la pila por medio de un índice negativo. De esta manera, será una forma de heredar atributos.

```
def p_asignacion_valor(t) :
    'asignacion_valor    :   expresion PTCOMA'
    t[0] = Asignacion(t[-2],t[1])

def p_dec_array_instr(t) :
    'asignacion_valor    : ARRAY PARIZQ PARDER PTCOMA'
    t[0] = Array(t[-2])

def p_asignacion_arr_St(t) :
    'asignacion_arr_st    : def_par lista_parametros  IGUAL expresion PTCOMA'
    t[1].extend(t[2])
    t[0] = AsignacionArrSt(t[-1],t[1],t[4])
```

Construcción del AST

La creación del **AST** requiere que a cada producción se le asocie una regla semántica. En este caso que estamos aplicando el patrón interprete, los nodos serán de un tipo de clase, ademas se puede realizar de varias manera la construcion del **AST** es decir pueden ser con clases abstractas o simplemente una clase

En este caso se muestran dos formar para realizar la construcion del AST

La Primera forma es realizar clases abstractas

```
def p_E(p):
    ''' E : E or E
          | E And        E
          | E diferente  E
          | E notEqual   E
          | E igual      E
          | E mayor      E
          | E menor      E
          | E mayorIgual E
          | E menorIgual E
          | E mas        E
          | E menos      E
          | E multi      E
          | E divi       E
          | E modulo     E
          | E elevado    E
          | E punto      E
    '''
    if p[2].lower() == "or":
        p[0] = SOperacion(p[1], p[3], Logicas.OR)
    elif p[2].lower() == "and":
        p[0] = SOperacion(p[1], p[3], Logicas.AND)
    elif p[2] == "<>":
        p[0] = SOperacion(p[1], p[3], Relacionales.DIFERENTE)
    elif p[2] == "=":
        p[0] = SOperacion(p[1], p[3], Relacionales.IGUAL)
    elif p[2] == ">":
        p[0] = SOperacion(p[1], p[3], Relacionales.MAYOR_QUE)
    elif p[2] == "<":
        p[0] = SOperacion(p[1], p[3], Relacionales.MENOR_QUE)
    elif p[2] == ">=":
        p[0] = SOperacion(p[1], p[3], Relacionales.MAYORIGUAL_QUE)
```

Nota: Se muestra en la figura de como se construye el **AST** por medio de clases abstracta que heredan de sentencias y luego se manda a llamar en las acciones semanticas

La clase SOperacion es la que herera de la clase abstracta **Sentencia** recibe 3 parametros para obtener los valores.

```
class SOperacion(Sentencia):
    def __init__(self, opIzq, opDer, operador):
        self.opIzq = opIzq
        self.opDer = opDer
        self.operador = operador
```

Segunda forma es la realizacion de una simple clase Node

```
def p_E(p):
    ''' E :  E or E
           | E And       E
           | E diferente  E
           | E notEqual   E
           | E igual      E
           | E mayor      E
           | E menor      E
           | E mayorIgual E
           | E menorIgual E
           | E mas        E
           | E menos      E
           | E multi      E
           | E divi       E
           | E modulo     E
           | E elevado    E
           | E punto      E
    '''
    global cont
    if p[2].lower() == "or":
        p[0]  = Node("E","",cont,0,0)
        cont  = cont+1
        nodo1 = Node("or",p[2],cont,p.lineno(2) ,p.lexpos(2))
        cont  = cont+1
        p[0].AddHijos(p[1])
        p[0].AddHijos(nodo1)
        p[0].AddHijos(p[3])
        lista.append(str(recorrerGramatica(p[0],0)))

    elif p[2].lower() == "and":
          p[0]  = Node("E","",cont,0,0)
          cont  = cont+1
          nodo1 = Node("and",p[2],cont,p.lineno(2) ,p.lexpos(2))
          cont  = cont+1
          p[0].AddHijos(p[1])
          p[0].AddHijos(nodo1)
          p[0].AddHijos(p[3])
          lista.append(str(recorrerGramatica(p[0],0)))
```

Nota: Se muestra en la figura de como se construye el **AST** por medio de clase simple llamado Node este es otra forma de como construirlo.

La clase Node contiene atributos para que reciban los valores como se muestra en la siguiente figura

```
class Node():
    def __init__(self,Etiqueta="",Valor="",idNod=0,Fila=0,Columna=0):
        self.Etiqueta=Etiqueta
        self.Valor=Valor
        self.idNod=idNod
        self.Fila=Fila
        self.Columna=Columna
        self.hijos=[]

    def AddHijos(self,son):
        self.hijos.append(son)

    def getHijos(self):
        return self.hijos
```

Recuperación de Errores

A cada producción le asignamos una función de la cual contiene la producción de error, la cual descartará la entrada hasta encontrar el carácter siguiente. En este caso, tenemos el carácter de punto y coma con el cual esperamos que se recupere y continúe el análisis y la posterior ejecución.

También declaramos un nuevo error y lo guardamos en una lista para mostrarla en un reporte cuando sea requerido.

```python
# FUNCIÓN PARA EL MANEJO DE LOS ERRORES SINTÁCTICOS
def p_error(t):
    col = find_column(t)
    # print("Error sintáctico en '%s'" % t.value, " Línea: '%s'" % str(t.lineno), " Columna: '%s'" % str(col) )
    errores_sintacticos.append(Error(t.value, 'Error Sintáctico', 'Símbolo no esperado', col, t.lineno))
```

Precedencia de Operadores

La precedencia de operadores son niveles de jerarquias en una gramatica, en este caso se declaran precedencia de operadores debido a que, la gramatica son ambiguas, y cada herramienta tiene la opcion declarar precedencia como por ejemplo alguno analizadores que son **Irony, Jflex & Jcup, Flex & Bison , Jison**, el analizador ply ademas de los otros analizador tambien puede declarar precedencia como se muestra en la siguiente figura.

```python
precedence = (
    ('right', 'not'),
    ('left', 'And'),
    ('left', 'or'),
    ('left', 'diferente','notEqual', 'igual', 'mayor', 'menor', 'menorIgual', 'mayorIgual'),
    ('left', 'punto'),
    ('right', 'umenos'),
    ('left', 'mas', 'menos'),
    ('left', 'elevado'),
    ('left', 'multi', 'divi', 'modulo'),
    ('nonassoc', 'parAbre', 'parCierra')
)
```

Nota: En este caso si una gramatica es ambigua en cierta operacion y no se declara precedencia el analizador puede causar error

Generación de código intermedio

En el proceso de traducir un programa fuente a código destino, un compilador puede construir una o más representaciones intermedias, las cuales pueden tener una variedad de formas. Los árboles sintácticos son una forma de representación intermedia; por lo general, se utilizan durante el análisis sintáctico y semántico.

Después del análisis sintáctico y semántico del programa fuente, muchos compiladores generan un nivel bajo explícito, o una representación intermedia similar al código máquina, que podemos considerar como un programa para una máquina abstracta. Esta representación intermedia debe tener dos propiedades importantes: debe ser fácil de producir y fácil de traducir en la máquina destino.

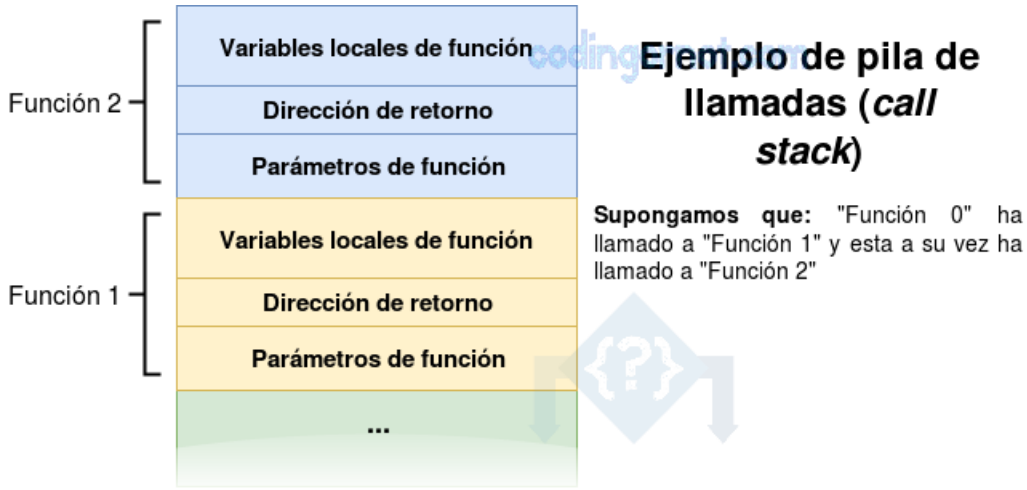
Existe una forma intermedia llamada código de tres direcciones, que consiste en una secuencia de instrucciones similares a ensamblador, con tres operandos por instrucción. ***por Ej***:

```
t1 := b * b
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x := t9
```

Pila

La pila de llamadas, pila de ejecución, pila de función, pila de control, pila de tiempo de ejecución o simplemente call stack es una estructura dinámica de datos que almacena la información sobre las subrutinas activas de un

programa en ejecución. Cuando hacemos una llamada a una función, un bloque en el tope de la pila es reservado para guardar las variables locales junto con algunos datos necesarios para garantizar el funcionamiento adecuado de la estructura (como la dirección a la que tendrá que retornar el hilo de ejecución cuando termine la función). Después de retornar, el bloque de la pila que ocupaba el llamado, se libera para poder utilizarse más adelante de ser necesario.



Heap

El montículo libre, zona libre, almacenamiento libre o heap es una estructura dinámica de datos utilizada para almacenar datos en ejecución. A diferencia de la pila de ejecución que solamente almacena las variables declaradas en los bloques previo a su ejecución, el heap permite reservar memoria dinámicamente, es decir, es el encargado de que la «magia» de la memoria dinámica ocurra.

Optimizacion de Codigo Intermdio

Optimización es un programa transformación técnica, que trata de mejorar el código por lo que consumen menos recursos (es decir CPU, memoria) y ofrecer una alta velocidad.

En la optimización de alto nivel general de programación son sustituidos por construcciones muy eficiente de bajo nivel los códigos de programación. Un código proceso en fase de optimización debe seguir las tres normas que se explican a continuación:

- El código de salida no debe, de ninguna manera, cambiar el sentido del programa.
- Optimización debe aumentar la velocidad del programa y si es posible, el programa debe exigir menos

cantidad de recursos.

- Optimización debe ser rápido y no debe retrasar el proceso de compilación general

Los esfuerzos para un código optimizado puede ser utilizado en los distintos niveles de elaboración del proceso.

- Al principio, los usuarios pueden cambiar o reorganizar el código o utilizar las mejores algoritmos para escribir el código.
- Después de generar código intermedio, el compilador puede modificar el código intermedio por dirección los cálculos y mejorar los lazos.
- Al tiempo que se produce la máquina de destino código, el compilador puede hacer uso de jerarquía de memoria y registros de la CPU.

Optimización independiente de la máquina

En esta optimización, el compilador toma en el código intermedio y transforma una parte del código que no implique un registros de la CPU y/o ubicaciones de memoria absoluta.

por Ej:

```
import principal as p3
from goto import with_goto
useActual=""
@with_goto
def main(stack=[]):
    label .myFuncion
    t9=10*9
    t10=5+t9
    t11=t10-3
    tax=t11
    t12=55*8
    t13=t12-9
    t14=15/3
    t15=t13+t14
    tax2=t15
    t16=tax+0
    tax3=t16
    if 5>3 :
```

Nota: En esta figura se genero un codigo de 3 Direcciones sin Optimizacion.

```
import principal as p3
from goto import with_goto
useActual=""
@with_goto
def main(stack=[]):
    label .myFuncion
    t9=10*9
    t10=5+t9
    t11=t10-3
    tax=t11
    t12=55*8
    t13=t12-9
    t14=15/3
    t15=t13+t14
    tax2=t15
    tax3=tax
    if(5>3): goto .L2
    goto .L3
    label .L2
```

Nota: En esta figura se genero un codigo de 3 Direcciones y esta Optimizado se ve una la diferencia con la figura anterior en la variable tax.

Optimización por mirilla

El método de mirilla consiste en utilizar una ventana que se mueve a través del código de 3 direcciones, la cual se le conoce como mirilla, en donde se toman las instrucciones dentro de la mirilla y se sustituyen en una secuencia equivalente que sea de menor longitud y lo más rápido posible que el bloque original. El proceso de mirilla permite que por cada optimización realizada con este método se puedan obtener mejores beneficios

Existen algunas Reglas para este metodo

Regla 1 Si existe una asignación de valor de la forma a = b y posteriormente existe una asignación de forma b = a, se eliminará la segunda asignación siempre que a no haya cambiado su valor. Se deberá tener la seguridad de que no exista el cambio de valor y no existan etiquetas entre las 2 asignaciones

Regla 2 Si existe un salto condicional de la forma Lx y exista una etiqueta Lx:, todo código contenido entre el goto Lx y la etiqueta Lx, podrá ser eliminado siempre y cuando no exista una etiqueta en dicho código.

Regla 3 Si existe un alto condicional de la forma if *cond* goto Lv; goto Lf; inmediatamente después de sus etiquetas Lv: *instrucciones* Lf: se podrá reducir el número de saltos negando la condición, cambiando el salto condicional hacia la etiqueta falsa Lf: y eliminando el salto condicional innecesario a goto Lf y quitando la etiqueta Lv

Regla 4 Si se utilizan valores constantes dentro de las condiciones de la forma if *cond* goto Lv; goto Lf; y el resultado de la condición es una constante verdadera, se podrá transformar en un salto incondicional y eliminarse el salto hacia la etiqueta falsa Lf.

Regla 5 Si se utilizan valores constantes dentro de las condiciones de la forma if *cond* goto Lv; goto Lf; y el resultado de la condición es una constante falsa, se podrá transformar en un salto incondicional y eliminarse el salto hacia la etiqueta verdadera Lv **Regla 6**

Si existe un salto incondicional de la forma goto Lx donde existe la etiqueta Lx: y la primera instrucción, luego de la etiqueta, es otro salto, de la forma goto Ly; se podrá realizar la modificación al primer salto para que sea dirigido hacia la etiqueta Ly: , para omitir el salto condicional hacia Lx.

Regla 7 Si existe un salto incondicional de la forma if *cond* goto Lx; y existe la etiqueta Lx: y la primera instrucciones luego de la etiqueta es otro salto, de la forma goto Ly; se podrá realizar la modificación al primer salto para que sea dirigido hacia la etiqueta Ly: , para omitir el salto condicional hacia Lx.

Regla 8

```
t11=t1+0
# Se elimna la Instruccion
```

Regla 9

```
t11=t10-0 # Se elimna la Instruccion
```

Regla 10

```
t2=t3*1
# Se elimna la Instruccion
```

Regla 11

```
t2=t3/1
# Se elimna la Instruccion
```

Regla 12

```
t11=t10-0
# Se Optimizo la Instrucion
t11=t10
```

Regla 13

```
t11=t1+0
# Se Optimizo la Instrucion
t11=t1
```

Regla 14

```
t2=t3*1
# Se Optimizo la Instrucion
t2=t3
```

Regla 15

```
t5=t6/1
# Se Optimizo la Instrucion
t5=t6
```

Regla 16

```
t12=t4*2
# Se Optimiza el codigo 3D
t12=t4+t4
```

Regla 17

```
t1=t2*0
# Se Optimiza el codigo 3D
t1=0
```

Regla 18

```
t6=0/t1
# Se Optimiza el codigo 3D
t6=0
```

Archivo Cod3D

En este archivo se muestra el Codigo 3 Direcciones

```
import principal as p3
from goto import with_goto
@with_goto
def main(stack=[]):
    label .myFuncion
    t9=10*9
    t10=5+t9
    t11=t10-3
    tax=t11
    t12=55*8
    t13=t12-9
    t14=15/3
    t15=t13+t14
    tax2=t15
    t16=tax+0
    tax3=t16
    if 5>3 : goto .L2
    goto .L3
    label .L2
    t17=tax+0
    tax=t17
    label .L3
```

Interprete sentencia

Este es el inicio del flujo de la programacion.

```
def interpretar_sentencias(arbol,ejecutar3D):
    # jBase.dropAll()
    global consola
    global texttraduccion, textoptimizado,identacion
    global stack
    global useActual
    global tablaSimbolos
    global contadoresT,contadoresEtiqueta
    global puntero
    if ejecutar3D:
        for nodo in arbol:
            if isinstance(nodo, SCrearBase):
                #crearBase(nodo, tablaSimbolos)
                stack.append(nodo)
                Eti ="t"+str(contadoresT)
                contadoresT+=1
                texttraduccion += identacion+Eti+"= p3.crearBase(stack["+str(puntero)+"],p3.tablaSimbolos)\n"
                puntero += 1
                texttraduccion += identacion+"print("+Eti+")\n"

            elif isinstance(nodo, SShowBase):
                stack.append(nodo)
                Eti ="t"+str(contadoresT)
                contadoresT+=1
                texttraduccion += identacion+Eti+"= p3.showBaseEx(\"stack["+str(puntero)+"]\")\n"
                #texttraduccion += identacion+Eti+"="+nodo.id +"\n"
                texttraduccion += identacion+"print("+Eti+")\n"
```

En algunas partes de nuestro logica existen algunos ejemplos de como se realizo algunas optimizacion de codigo de 3 Direcciones aplicando algunas reglas de optimizacion.

```
        if (str(opDer.valor)=="0" or str(opIzq.valor)=="0"):
            #contadoresT-=1
            vopt="0"
            aux.opt =vopt
            aux.regla ="Regla 17"
            arr_optimizacion.append(aux)
            ropt=17
        elif(str(opDer.valor)=="1"):
            #contadoresT-=1
            vopt=opIzq.valor
            aux.opt =vopt
            aux.regla ="Regla 14"
            arr_optimizacion.append(aux)
            ropt=14
        elif(str(opIzq.valor)=="1"):
            #contadoresT-=1
            vopt=opDer.valor
            aux.opt =vopt
            aux.regla ="Regla 14"
            arr_optimizacion.append(aux)
            ropt=14
        elif(str(opDer.valor)=="2"):
            textoptimizado +=identacion+ Etiqueta + "="+str(opIzq.valor)+"+" +str(opIzq.valor)+"\n"
            vopt=Etiqueta
            aux.opt =vopt
            aux.regla ="Regla 16"
            arr_optimizacion.append(aux)
            ropt=16
        elif(str(opIzq.valor)=="2"):
            textoptimizado +=identacion+ Etiqueta + "="+str(opDer.valor)+"+" +str(opDer.valor)+"\n"
            vopt=Etiqueta
            ropt=16
```