Tecnológico de Monterrey

Desarrollo de aplicaciones avanzadas de ciencias computacionales

Actividad Final Mini-Proyecto

Luis Omar Leyva Navarrete A01570367

Campus Monterrey

1. Semántica

1.1. Tabla de variables

La tabla de variables será creada con una estructura de hash table de python, no se incluye el scope debido a que solo hay un nivel.

Symbol	Type
a	int
abc	int
def	float
c	int
mike	float

1.2. Cubo Semántico

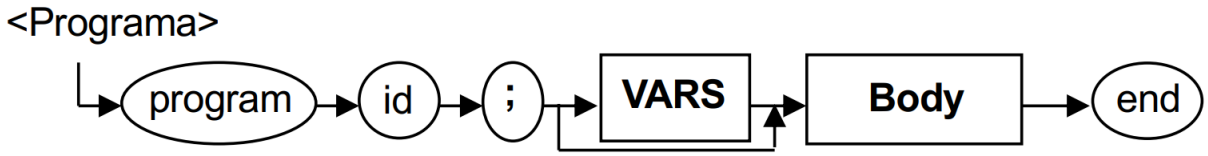
El cubo semántico se creó como un diccionario en el cual podemos consultar el resultado del operador y los tipos de los dos operandos.

l_op	r_op	+	-	*	/	>	<	!=
int	int	int	int	int	float	bool	bool	bool
float	int	float	float	float	float	bool	bool	bool
float	float	float	float	float	float	bool	bool	bool
int	float	float	float	float	float	bool	bool	bool

2. Puntos neurálgicos y diagramas de sintaxis

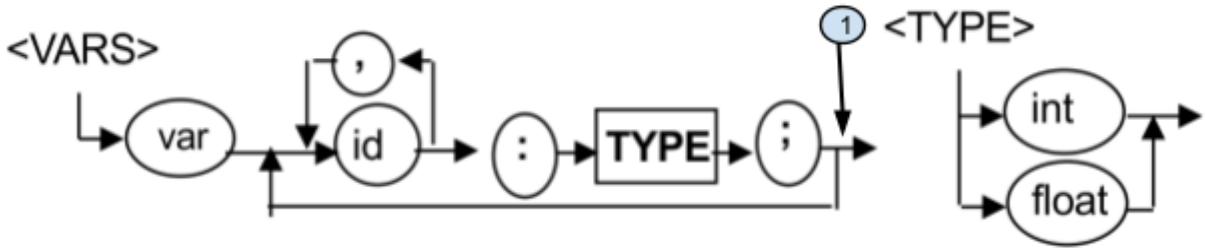
2.1. Programa

El diagrama de *Programa* no contiene ningún punto neurálgico.



2.2. VARS y TYPE

El diagrama de sintaxis de *VARS* y *TYPE* solo contiene un punto neurálgico en el cual se agregan todas las variables a la tabla de variables.



En el siguiente código se muestra el punto neurálgico en el cual después del parsing se recorre el resultado obtenido para guardar todas las variables válidas en la tabla de variables.

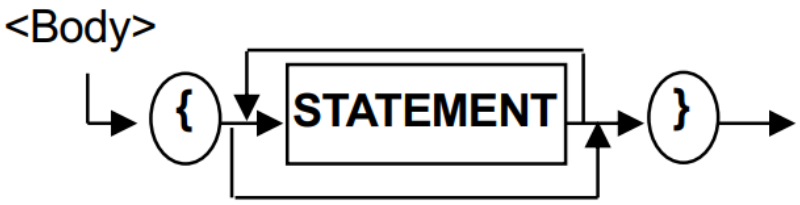
2.2.1. agregar_variables

```python
def agregar_variables(var_list):
    len_list = len(var_list)

    for i in range(0, len_list):
        len_list_i = len(var_list[i])
        var_type = var_list[i][len_list_i - 1]

        for j in range(0, len_list_i - 1):
            var_id = var_list[i][j]

            # Si la variable ya esta declarada
            try:
                if tabla_variables.keys().__contains__(var_id):
                    raise ValueError(
                        "Error: La variable '{}' ya esta declarada".format(
                            var_list[i][j]
                        )
                    )
                    exit()
                else:
```

```
                tabla_variables[var_id] = var_type
            except ValueError as error:
                print(error)
                exit()
```

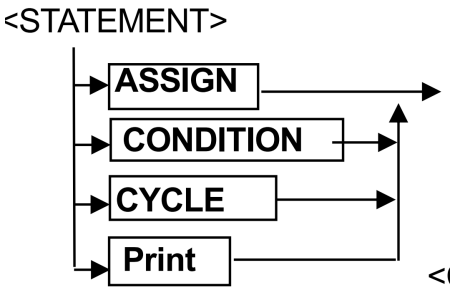
2.3. Body

El diagrama de *Body* no contiene ningún punto neurálgico.



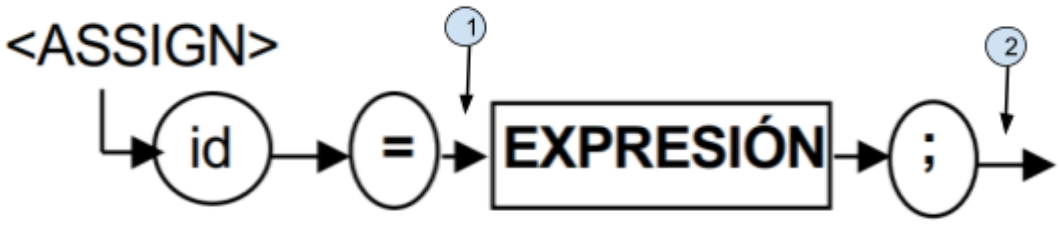
2.4. STATEMENT

El diagrama de *STATEMENT* no contiene ningún punto neurálgico.



2.5. ASSIGN

El diagrama de *ASSIGN* contiene dos puntos neurálgicos.



2.5.1. id_equal

En el primer punto verificamos que la variable exista para despues agregar el operando y el operador a las pilas.

```python
def id_equal(token):
    # Checar que la variable exista
    try:
        if not tabla_variables.keys().__contains__(token[0]):
            raise ValueError(
                "Error: La variable '{}' no esta declarada".format(token[0])
            )
    except ValueError as error:
        print(error)
        exit()

    # Utilizamos las correspondientes
    pila_operandos.put(token[0])
    pila_tipos.put(tabla_variables[token[0]])
    pila_operadores.put("=")
```

2.5.2. cuadruplo_assign

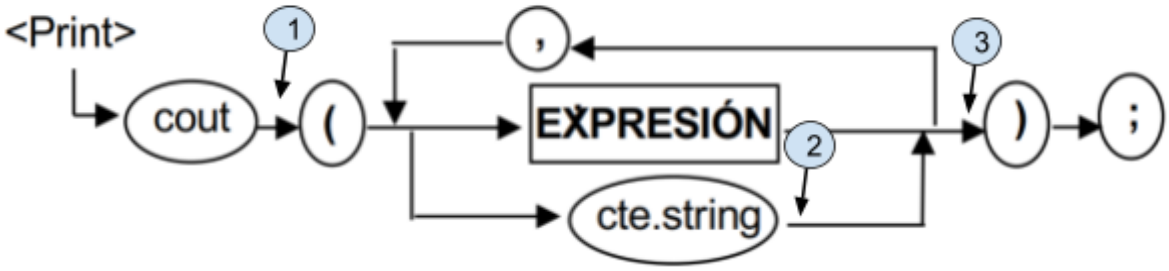
En el segundo punto verificamos que los tipos de todos los operandos sean correctos. Se agregan a las pilas correspondientes y se libera la variable temporal en caso de que exista.

```python
def cuadruplo_assign(token):
    # Verificamos que el tipo de resultado sea = al tipo de la variable
    tipo_resultado = pila_tipos.get()
    try:
        # Utilizamos el cubo semantico
        if tipo_resultado != pila_tipos.get():
            raise ValueError(
                "Error: tipo "
                + tipo_resultado
                + " no puede ser asignado a tipo "
                + tabla_variables[token[0]]
            )
    except ValueError as error:
        print(error)
```

```
        exit()

    # Generamos el cuadruplo de asignacion
    temp = pila_operandos.get()
    tabla_variables[temp] = tipo_resultado
    generar_cuadruplo(pila_operadores.get(), temp, None, pila_operandos.get())

    # Regreamos el temporal a su fila si es que es temporal
    if type(temp) == str:
        if temp[0:4] == "temp":
            fila_temporales.put(temp)
```

2.6. Print

El diagrama de *Print* contiene 3 puntos neurálgicos.



2.6.1. put_operators

Se agrega el operador a la pila para continuar la operación.

```
Python
def put_operators(token):
    pila_operadores.put(token[0])
```

2.6.2. put_string

Si el objeto a imprimir agregamos el tipo string a la pila de tipos.

```
Python
def put_string(token):
    pila_operandos.put(token[0])
```

```
    pila_tipos.put("string")
    tabla_variables[token[0]] = "string"
```

2.6.3. cuadruplo_print

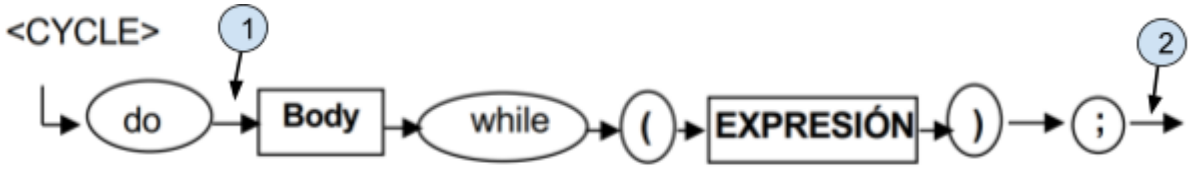
Generamos los cuádruplos necesarios para imprimir todos los objetos en la lista de impresión. Los objetos se concatenan para generar un string grande.

Python

```python
def cuadruplo_print(token):
    # Creamos cuadruplo de print con multiples operandos
    if pila_operadores.queue[-1] == "cout":
        if len(token) == 1:
            pila_tipos.get()
            generar_cuadruplo(
                pila_operadores.get(), None, None, pila_operandos.get()
            )
        else:
            for i in range(len(token) - 1):
                pila_operadores.put("concat")

            # Realizamos una suma de strings
            for i in range(len(token) - 1):
                pila_tipos.get()
                right_op = str(pila_operandos.get())
                pila_tipos.get()
                left_op = str(pila_operandos.get())
                operator = pila_operadores.get()
                result = fila_temporales.get()
                generar_cuadruplo(operator, left_op, right_op, result)
                if left_op[0:4] != "temp" and right_op[0:4] == "temp":
                    fila_temporales.put(right_op)
                pila_operandos.put(result)
                pila_tipos.put("string")
                tabla_variables[result] = "string"

            result = pila_operandos.get()
            generar_cuadruplo(pila_operadores.get(), None, None, result)
            fila_temporales.put(result)
            pila_tipos.get()
```

2.7. Cycle

El diagrama de *Cycle* contiene 2 puntos neurálgicos.



2.7.1. put_do_jump

Revisamos la cantidad de listas en la fila de cuadruplos y agregamos el tamaño actual de la fila a la fila de saltos.

```python
def put_do_jump():
    pila_saltos.put(len(fila_cuadruplos.queue))
```

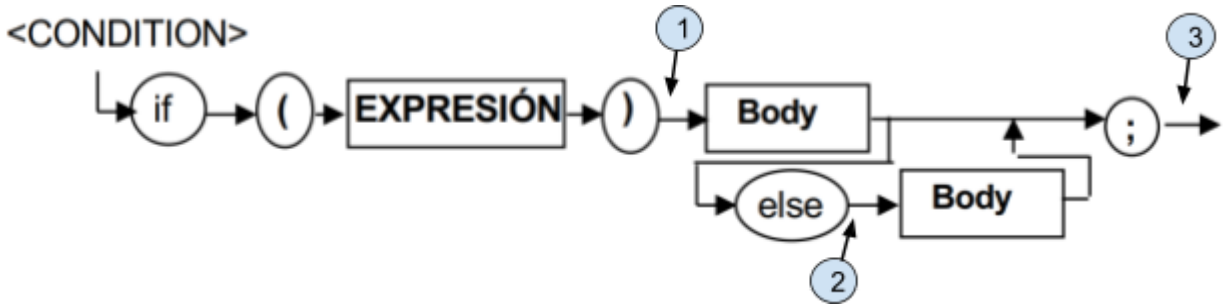
2.7.2. do_while_cuadruplo

Generamos los cuádruplos con los operadores de GOTOV y los agregamos a la fila.

```python
def do_while_cuadruplo():
    tipo_esp = pila_tipos.get()
    try:
        if tipo_esp != "bool":
            raise ValueError("Error: La expresion no es booleana")
        else:
            result = pila_operandos.get()
            generar_cuadruplo("GOTOV", result, None, pila_saltos.get() + 1)
    except ValueError as err:
        print(err)
        exit()
```

2.8. Condition

El diagrama de *Condition* contiene 3 puntos neurálgicos.



2.8.1. if_jumps

Generamos los cuádruplos sin los puntos de salto.

```python
def if_jumps():
    tipo_esp = pila_tipos.get()
    try:
        if tipo_esp != "bool":
            raise ValueError("Error: La expresion no es booleana")
        else:
            result = pila_operandos.get()
            generar_cuadruplo("GOTOF", result, None, None)
            pila_saltos.put(len(fila_cuadruplos.queue) - 1)
    except ValueError as error:
        print(error)
        exit()
```

2.8.2. if_end

Agregamos los puntos de salto al cuádruple pendiente.

```python
def if_end():
    end = pila_saltos.get()
    fila_cuadruplos.queue[end][3] = len(fila_cuadruplos.queue) + 1
```

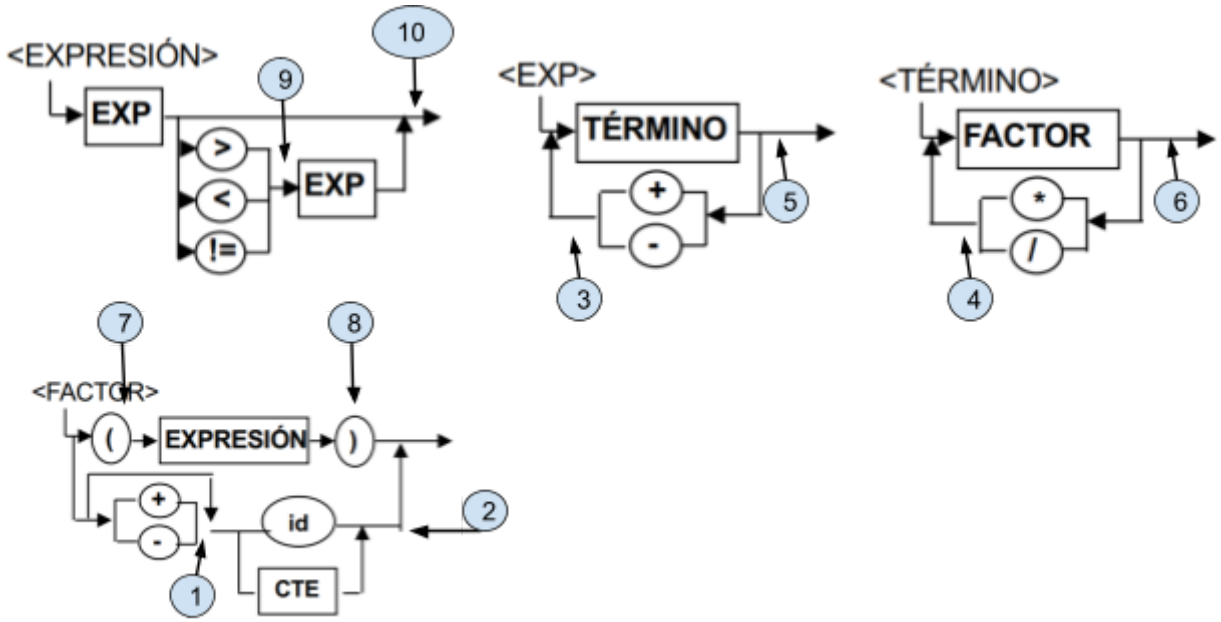
2.8.3. else_jumps

Generamos el cuádruple y agregamos el punto de salto del if inicial.

```python
def else_jumps():
    generar_cuadruplo("GOTO", None, None, None)
    end = pila_saltos.get()
    pila_saltos.put(len(fila_cuadruplos.queue) - 1)
    fila_cuadruplos.queue[end][3] = len(fila_cuadruplos.queue) + 1
```

2.9. Expresion

El diagrama de *Expresion* contiene múltiples puntos neurálgicos debido a que es un conjunto de varias reglas gramaticales.



2.9.1. negative_positive

Para el punto 1 se utilizó una función que agrega el operador de números negativos o positivos de ser necesario.

```python
def negative_positive(token):
    if token[0] == "-":
        pila_operadores.put("negative")
    elif token[0] == "+":
        pila_operadores.put("positive")
```

2.9.2. id_constant

En el punto 2 se agregan las constantes o variables a la pila de operandos junto con su tipo.

```python
def id_constant(token):
    if token[0] == tk.id_:
        try:
            if not tabla_variables.keys().__contains__(token[0]):
                raise ValueError(
                    "Error: La variable " + token[0] + " no esta declarada"
                )
        except ValueError as error:
            print(error)
            exit()
        pila_operandos.put(token[0])
        pila_tipos.put(tabla_variables[token[0]])
    else:
        if token[0] == tk.cte_int:
            pila_tipos.put("int")
            pila_operandos.put(int(token[0]))
        elif token[0] == tk.cte_float:
            pila_tipos.put("float")
            pila_operandos.put(float(token[0]))
```

2.9.3. put_operator

Se utilizó el mismo procedimiento que en el apartado 2.6.1 de este documento, de igual manera se agregan los operadores a la pila. Esta se utilizó en los puntos neurálgicos 4, 3 y 9.

2.9.4. check_precedence

Utilizamos dos funciones que en los puntos 5, 6 y 10 revisara el tipo de operador en la pila y a partir de ahí ejecutar la función de exp_precedence, la cual generar los cuádruplos correspondientes a cada operación. Esta función respeta el orden correcto de operaciones o la precedencia de operaciones. Una vez agregados todos los elementos a las pilas correspondientes se generan los cuádruplos con las variables y constantes necesarias.

Python

```python
def exp_precedence():
    right_op = pila_operandos.get()
    right_type = pila_tipos.get()
    operator = pila_operadores.get()
    left_op = pila_operandos.get()
    left_type = pila_tipos.get()

    try:
        result_type = cubo_s[left_type][right_type][operator]
        if result_type == "error":
            raise ValueError(
                "Error: tipo "
                + left_type
                + " no puede ser operado con tipo"
                + right_type
            )
    except ValueError as error:
        print(error)
        exit()

    temp = fila_temporales.get()
    generar_cuadruplo(operator, left_op, right_op, temp)
    pila_operandos.put(temp)
    pila_tipos.put(result_type)
    tabla_variables[temp] = result_type
    if type(left_op) == str:
        if left_op[0:4] == "temp":
            fila_temporales.put(left_op)
    if type(right_op) == str:
        if right_op[0:4] == "temp":
            fila_temporales.put(right_op)


def check_precedence(token):
    if pila_operadores.queue.__len__() > 0:
        top_operadores = pila_operadores.queue[-1]
        if top_operadores == "negative" or top_operadores == "positive":
            right_op = pila_operandos.get()
            operator = pila_operadores.get()
            temp = fila_temporales.get()
            if operator == "negative":
                generar_cuadruplo("*", right_op, -1, temp)
                tabla_variables[temp] = pila_tipos.queue[-1]
            pila_operandos.put(temp)
```

```
        elif top_operadores == "+" or top_operadores == "-":
            exp_precedence()
        elif top_operadores == "*" or top_operadores == "/":
            exp_precedence()
        elif (
            top_operadores == "<"
            or top_operadores == ">"
            or top_operadores == "!="
        ):
            if fila_bool.get():
                exp_precedence()
                fila_bool.put(False)
            else:
                fila_bool.put(True)
```

2.9.5. put_parenthesis

Para los puntos 7 y 8 se utilizó una función que agregue o quite los paréntesis a la fila de operadores dependiendo de la necesidad. Esto para respetar el orden de operaciones.

```
Python
def put_parenthesis(token):
    if token[0][0] == "(":
        pila_operadores.put("(")
    elif token[0][0] == ")":
        pila_operadores.get()
```