

Tecnológico de Monterrey

Documentación Patito - Ducklang

Entrega 5 Final

Luis Ángel González Romo A01235962

21 de noviembre del 2024, Campus Monterrey

Desarrollo de aplicaciones avanzadas de ciencias computacionales, TC3002B Grupo 502

Profesora Elda G Quiroga

Profesor Dr. Iván Mauricio Amaya Contreras

Profesor Mauricio González Soto

Investigación de Generación Automática de Analizadores de Léxico y Sintaxis

Las dos herramientas elegidas para esta investigación son Flex & Bison y Python ANTLR4. La primera de estas herramientas consiste en un generador de analizador léxico, Flex (Fast Lexical Analyzer), y un generador de analizador sintáctico, Bison (forma parte del Bison Project de GNU), fue diseñado para ser utilizado con el lenguaje C. La segunda herramienta ANTLR (Another Tool for Language Recognition) es más moderna y maneja ambos analizadores a la vez (léxico y sintáctico) con un solo generador, disponible para muchos lenguajes de programación y en este caso será con Python. A continuación, se desglosan con más detalles las características principales de cada herramienta de esta investigación inicial.

➔ Flex & Bison

Como se mencionó anteriormente, estas herramientas son más antiguas y fueron diseñadas para generar código en C para análisis léxico y sintáctico, respectivamente. Son herramientas open source bajo la licencia GPL (General Public Licence) de GNU. Primero, Flex es utilizado con expresiones regulares para escanear texto y crear los tokens. Después estos tokens, se pasan al análisis sintáctico de Bison, el cual es utilizado con gramáticas libres de contexto que realizan su parsing de manera bottom-up y un algoritmo LALR(1) que crea los árboles parse. Este algoritmo es una versión mejorada y eficiente del algoritmo LR que optimiza el tamaño de las tablas de gramática. Las interfaces de estas herramientas son más complejas y con peor usabilidad comparado con herramientas más modernas por su exclusividad con el lenguaje C, y además maneja archivos de Flex separados de los archivos de Bison para poder crear un proyecto.

➔ Python ANTLR4

Esta otra herramienta más reciente es un generador de analizador léxico y sintáctico compatible con más lenguajes de código (también más modernos que C) como C#, JavaScript, Java y Python. Es una herramienta que integra ambas partes y actualmente está en su versión 4, también es open source bajo licencia BSD

(Berkeley Source Distribution) la cual es aún más permisiva que la de GPL de Flex & Bison. Esta herramienta funciona con un solo archivo de gramática para generar el analizador sintáctico y el analizador léxico. Utiliza el método top-down con el algoritmo LL(*), el cual es también una mejora de los algoritmos LL(k) y LALR(1). Es capaz de manejar gramáticas más complejas y tiene mejor performance. En el archivo de gramática mencionado anteriormente, se definen los tokens del análisis léxico con las propias reglas de léxico que maneja ANTLR pero asemejando expresiones regulares, y también en este archivo van las reglas de la gramática libre de contexto (producciones en notación formal), además su interfaz soporta acciones semánticas customizadas, visitors y listeners. Cabe recalcar que también se analizó un artículo que expone la preferencia actual y la eficacia para trabajar del uso de ANTLR comparado con herramientas más tradicionales como Flex & Bison, por eso se decidió utilizarla para continuar con los siguientes pasos en los entregables del módulo.

➔ Referencias

Levine, J. (2009). *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.".

Ortin, F., Quiroga, J., Rodriguez-Prieto, O., & Garcia, M. (2022). An empirical evaluation of Lex/Yacc and ANTLR parser generation tools. *Plos one, 17(3)*, e0264326.

Test-Plan

Para esta entrega se creó un archivo de texto (input_ducklang_code.txt) con código en lenguaje Ducklang (patito) para poder testear la implementación del analizador léxico y sintáctico, después se le realizaron modificaciones para poder testear el análisis semántico igualmente. Con este archivo se prueban distintos ejemplos importantes de los posibles tokens permitidos, y también se prueban, con distintos ejemplos, todas las reglas gramaticales declaradas en en archivo utilizado por ANTLR para generar el parser y el lexer.

```
programa patito;

vars uno, dos: flotante; tres: entero;

nula funcioncita (x: flotante, y: entero) {
    vars a: flotante;
    {
        a = 1.1;
        escribe("Hola mundo ducklang", a-1+1);
    }
};

nula funcionsota (x: flotante, y: entero) {
    vars a: flotante;
    {
        a = 2.2;
        escribe("Hola mundo ducklang", a+1);
    }
};

inicio {
    uno = 1.11;
    dos = 2.22;
    dos = (uno * -2) - 3.3 / dos;
    si(dos > 0) {
        dos = uno;
    } sino {
        tres = 0;
        mientras(tres != 3) haz {
            tres = tres + 1;
            escribe("Ya casi es tres", tres);
            funcioncita(uno, 1+1);
        };
    };
    funcionsota(5.5, 2);
} fin
```

De los 32 tokens declarados se prueban todos excepto el operador ‘==’ y el operador ‘<’, los cuales se les realizará test más a fondo cuando se implemente el análisis semántico. Los tokens más especiales son los de id, letrero, constante entera y constante flotante, de los cuales se exploraron casos “happy path” para probar la implementación de la herramienta. Cabe recalcar que durante la creación

de este ejemplo, hubo en muchas instancias excepciones y errores que detectó el parser y lexer por errores en el ejemplo por mal uso de Ducklang y no por errores en el uso de ANTLR.

De las 37 reglas gramaticales que se implementaron, también se prueban todas con este ejemplo y deja preparado para los siguientes pasos. Con el análisis semántico, se agregaron acciones por medio de los listeners, junto con el uso de otras estructuras de datos que permiten validar y “adornar” el parsing tree que se genera en esta etapa. Se recorre el árbol, y según se vayan visitando los nodos (reglas sintácticas), ya sea que se ingrese o se salga (asemejando los conceptos de First y Follow), se irán ejecutando funciones de validación y se utilizan en estos puntos neurálgicos las estructuras de apoyo para tener memoria de aspectos del código analizado.

Implementación de análisis de semántica

En la primera implementación de análisis de semántica para Ducklang, se utilizaron tres estructuras principales de apoyo para poder facilitar el proceso de esta etapa. La primera es un diccionario de funciones en la cual tiene como llave el nombre de cada función que será creada en el código Ducklang, y en su valor se encuentra otro diccionario anidado donde (por lo pronto) hay una sola llave de ‘parámetros’ con un valor un diccionario de variables y sus tipos de esa función. Esto será útil para acceder rápidamente a la información necesaria para poder realizar llamadas a cada función y validar que se esté llamando correctamente a una función que ya fue creada y con los parámetros necesarios.

La siguiente estructura que se utilizó fue otro diccionario para la tabla de símbolos donde hay dos llaves, la de variables globales y la de variables locales. El valor de cada una es otro diccionario donde se encuentran nombres de las variables (identificadores) como llaves, y sus valores correspondientes son el tipo de variable (por lo pronto así está, más adelante se tendrá que considerar guardar también el valor que almacena la variable). El diccionario de variables locales se elimina al terminar el parsing de cada función, mientras que las globales se quedan guardadas, eso es para que se pueda respetar el scope de cada variable más adelante y poder validar el error de variable doblemente declarada por scope local

aislado de los otros locales y preservar las globales para que tampoco se vuelvan a declarar en ningún otro lado.

La última estructura que se utilizó es otro diccionario para el cubo semántico donde se encuentran las posibles combinaciones de flotante, entero y booleano con los operadores (+, -, *, /, >, <, !=, ==) de manera anidada, para que más adelante se puedan acceder de manera fácil con cuadruplos y verificar que se estén mandando a llamar correctamente en las expresiones y ver el tipo de dato resultante de estas operaciones. Cabe destacar que se utilizaron los diccionarios por la facilidad que brindan para buscar elementos que ya existen y así poder realizar el análisis semántico correspondiente de manera rápida.

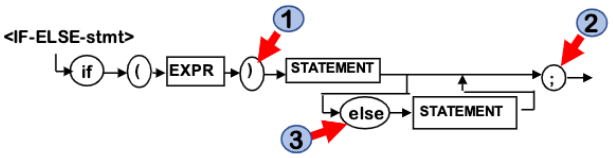
Generación de código intermedio (Cuádruplos)

Para poder generar una representación del código intermedio, que será leído por la máquina virtual del compilador al ser ejecutado el código DuckLang, se utilizan cuádruplos los cuales son formados con distintas estructuras de datos. Son ‘deques’ de la librería de collections que los hace más eficientes que una queue normal, y se le van agregando dentro lists del objeto default de Python para agregar 4 elementos por elemento de la deque. El primer elemento indica el operador o la operación que deberá ejecutar la máquina virtual, el segundo el operando izquierdo de la operación (si lo hay), el tercero el operando derecho (si lo hay), y el cuarto corresponde a la dirección de memoria donde ese resultado será guardado por el procesador con arquitectura de Von Neumann (si lo hay, también). Gracias a los puntos neurálgicos y a otras estructuras previamente preparadas se puede acceder a puntos específicos donde se debe de ir generando los cuádruplos al recorrer el árbol del código parseado, esto último se logra gracias a los listeners de ANTLR extendiendo las clases de enter y exit de cada regla gramatical (nodos).

Se utilizan distintas pilas (que también son creadas con el objeto ‘deque’ pero utilizadas con otros métodos para que se comporten como una stack de manera eficiente) y con esto se puede llevar registro de los operadores, de los operandos, sus tipos, de los saltos que debe realizar el procesador entre las líneas de los cuádruplos para controlar los flujos, y de los espacios de memoria temporal que quedan disponibles. A continuación se muestra la manera con la que se trabajó de manera visual las distintas reglas gramaticales y sus funciones para implementar el pseudocódigo de los algoritmos a partir de puntos neurálgicos que permiten generar

los cuádruplos de las expresiones aritméticas, las asignaciones de variables, la impresión de letreros y expresiones en consola, los flujos condicionales, los ciclos 'while' (mientras - haz), la declaración de funciones, y la llamada de funciones:

TWO-WAYS CONDITIONAL STATEMENT



1.- exp_type = PTypes.Pop()
if (**exp_type** != bool) **ERROR**(Type-mismatch)
else
result = PilaO.Pop()
Generate quad: **GotoF, result , , ___**
PJumps.Push (cont-1)

2.- end=PJumps.Pop()
FILL (end, cont)

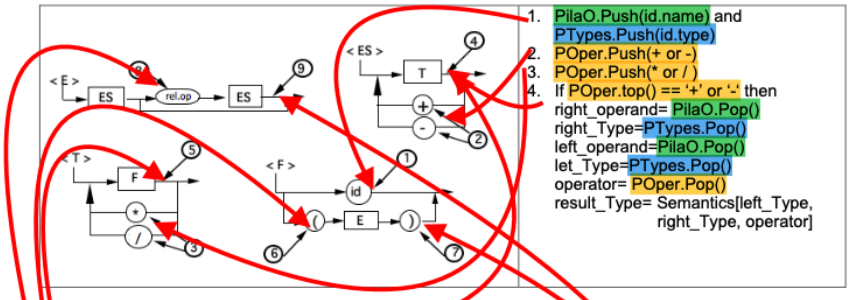
3.- Generate quad: **GOTO ___**
false= PJumps.Pop()
PJumps.Push(**cont-1**)
FILL (falso, cont)

PRE-CONDITION LOOP STATEMENT



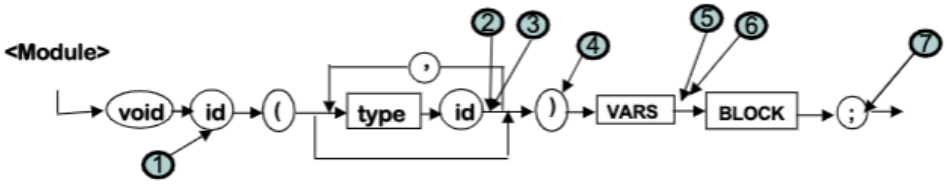
1.- PJumps.Push (cont)
2.- exp_type = PTypes.Pop()
if (**exp_type** != bool) **ERROR**(Type-mismatch)
else
result = PilaO.Pop()
Generate quad: **GotoF, result , , ___**
PJumps.Push (cont-1)
3.- end=PJumps.Pop()
return=PJumps.Pop()
Generate quad: **GOTO return**
FILL (**end**, cont)

We need a Stack for "pending operators" (POper), a Stack for "pending operands" (PilaO), a Stack for corresponding types (PTypes and a Queue (Quad) for output.

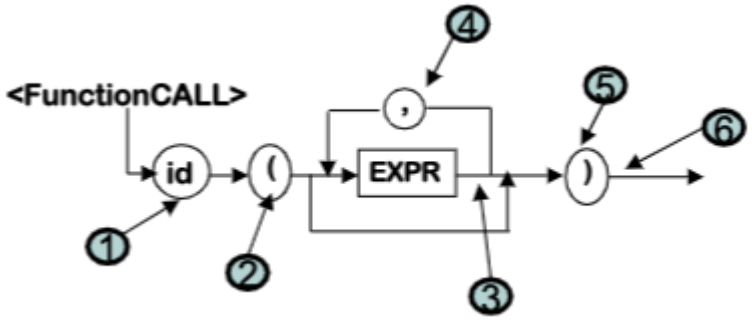


1. PilaO.Push(id.name) and PTypes.Push(id.type)
2. POper.Push(+ or -)
3. POper.Push(* or /)
4. If POper.top() == '+' or '-' then
right_operand= PilaO.Pop()
right_Type=PTypes.Pop()
left_operand=PilaO.Pop()
let_Type=PTypes.Pop()
operator= POper.Pop()
result_Type= Semantics[left_Type, right_Type, operator]

4…continue…
if (result_Type != ERROR)
result ←AVAIL.next()
generate quad= (operator, left_operand, right_operand, result)
Quad.Push(quad)
PilaO.Push(result) PTypes.Push(result_Type)
If any operand were a temporal space, return it to AVAIL
Else
ERROR ("Type mismatch")
5. If POper.top() == '*' or '/' then
= to #4 with *,/
6. POper.Push(False bottom mark)
7. POer.Pop(False bottom mark)
8. POper.Push(rel.op) (like >, <, etc)
9. If POper.top() == rel.op then
= to #4 with >, <,..



1.- Insert Function name into the DirFunc table (and its type, if any), verify semantics.
2.- Insert every parameter into the current (local) VarTable.
3.- Insert the type to every parameter uploaded into the VarTable.
At the same time into the **ParameterTable** (to create the Function's signature)..
4.- Insert into DirFunc the number of parameters defined. ***to calculate the workspace required for execution***
5.- Insert into DirFunc the number of local variables defined. ***to calculate the workspace required for execution***
6.- Insert into DirFunc the current quadruple counter (CONT), ***to establish where the function starts***
7.- Release the current VarTable (local).
Generate an action to end the function (**ENDFunc**).
Insert into DirFunc the number of temporal vars used. ***to calculate the workspace required for execution***



1.- Verify that the function exists into the DirFunc.
2.- Generate action **ERA size** (*Activation Record expansion –NEW—size).*
Start the parameter counter (k) in 1.
Add a pointer to the first parameter type in the ParameterTable.
3.- Argument= PilaO.Pop() ArgumentType= PTypes.Pop().
Verify ArgumentType against current Parameter (#k) in ParameterTable.
Generate action **PARAMETER, Argument, Argument#**k
4.- K = K + 1, move to next parameter.
5.- Verify that the last parameter points to null *(coherence in number of parameters).*
6.- Generate action **GOSUB**, **procedure-name, , initial-address**.

*Obtenido y utilizado directamente desde el material de clase

Ejemplo de ejecución del compilador:

```
ARBOL RESULTANTE:

(programa programa patito ; (pvars (vars vars (vidtipo (vtipo (vid uno
, (vid dos)) : (tipo flotante) ;) (vidtipo (vtipo (vid tres) : (tipo
entero) ;)))))) (ppfuncs (pfuncs (funcs nula funcioncita ( (fidtipo (fid
x : (tipo flotante) , (fid y : (tipo entero)))) ) { (pvars (vars vars
(vidtipo (vtipo (vid a) : (tipo flotante) ;)))) (cuerpo { (ccestatuto
```

```
(cestatuto (estatuto (asigna a = (expresion (exp (termino (factor fsign (faid (cte 1.1))))) eexp) ;)) (cestatuto (estatuto (imprime escribe ( (icont (cont "Hola mundo ducklang") , (icont (cont (expresion (exp (termino (factor fsign (faid a))) (sign -) (exp (termino (factor fsign (faid (cte 1)))) (sign +) (exp (termino (factor fsign (faid (cte 1)))))) eexp)))) ) ;)))))) }) } ;)) (ppfuncs (pfuncs (funcs nula funcionsota ( (fidtipo (fid x : (tipo flotante) , (fid y : (tipo entero)))) ) { (pvars (vars vars (vidtipo (vtipo (vid a) : (tipo flotante) ;)))) (cuerpo { (ccestatuto (cestatuto (estatuto (asigna a = (expresion (exp (termino (factor fsign (faid (cte 2.2))))) eexp) ;)) (cestatuto (estatuto (imprime escribe ( (icont (cont "Hola mundo ducklang") , (icont (cont (expresion (exp (termino (factor fsign (faid a))) (sign +) (exp (termino (factor fsign (faid (cte 1)))))) eexp)))) ) ;)))))) }) } ;)) ppfuncs)) inicio (cuerpo { (ccestatuto (cestatuto (estatuto (asigna uno = (expresion (exp (termino (factor fsign (faid (cte 1.11))))) eexp) ;)) (cestatuto (estatuto (asigna dos = (expresion (exp (termino (factor fsign (faid (cte 2.22))))) eexp) ;)) (cestatuto (estatuto (asigna dos = (expresion (exp (termino (factor ( (expresion (exp (termino (factor fsign (faid uno)) (op *) (termino (factor (fsign (sign -)) (faid (cte 2)))))) eexp) ))) (sign -) (exp (termino (factor fsign (faid (cte 3.3))) (op /) (termino (factor fsign (faid dos)))))) eexp) ;)) (cestatuto (estatuto (condicion si ( (expresion (exp (termino (factor fsign (faid dos)))) (eexp (comp >) (exp (termino (factor fsign (faid (cte 0)))))) ) (cuerpo { (ccestatuto (cestatuto (estatuto (asigna dos = (expresion (exp (termino (factor fsign (faid uno))))) eexp) ;))))) }) (csino sino (cuerpo { (ccestatuto (cestatuto (estatuto (asigna tres = (expresion (exp (termino (factor fsign (faid (cte 0))))) eexp) ;)) (cestatuto (estatuto (ciclo mientras ( (expresion (exp (termino (factor fsign (faid tres)))) (eexp (comp !=) (exp (termino (factor fsign (faid (cte 3)))))) ) haz (cuerpo { (ccestatuto (cestatuto (estatuto (asigna tres = (expresion (exp (termino (factor fsign (faid tres))) (sign +) (exp (termino (factor fsign (faid (cte 1)))))) eexp) ;)) (cestatuto (estatuto (imprime escribe ( (icont (cont "Ya casi es tres") , (icont (cont (expresion (exp (termino (factor fsign (faid tres)))) eexp)))) ) ;)) (cestatuto (estatuto (llamada funcioncita ( (llexpresion (lexpresion (expresion (exp (termino (factor fsign (faid uno)))) eexp) , (lexpresion (expresion (exp (termino (factor fsign (faid (cte 1)))) (sign +) (exp (termino (factor fsign (faid (cte 1)))))) eexp)))) ) ;))))))) }) ;))))) })) ;)) (cestatuto (estatuto (llamada funcionsota ( (llexpresion (lexpresion (expresion (exp (termino (factor fsign (faid (cte 5.5))))) eexp) , (lexpresion (expresion (exp (termino (factor fsign (faid (cte 2))))) eexp)))) ) ;)))))))) }) fin)
```

```
Directorio de funciones al finalizar programa:
{'funcioncita': {'parametros': {'y': 'entero', 'x': 'flotante'},
'numParams': 2, 'inicioQuad': 1, 'numVars': 3, 'numTemps': 2},
'funcionsota': {'parametros': {'y': 'entero', 'x': 'flotante'},
'numParams': 2, 'inicioQuad': 7, 'numVars': 3, 'numTemps': 1},
'__main__': {'numVars': 3, 'numTemps': 3}}

Cuádruplos generados:
0: ['GoTo', None, None, 12]
1: ['=', '1.1', None, 'a']
2: ['print', '"Hola mundo ducklang"', None, None]
3: ['-', 'a', '1', 't1']
4: ['+', 't1', '1', 't2']
5: ['print', 't2', None, None]
6: ['ENDFunc', None, None, None]
7: ['=', '2.2', None, 'a']
8: ['print', '"Hola mundo ducklang"', None, None]
9: ['+', 'a', '1', 't2']
10: ['print', 't2', None, None]
11: ['ENDFunc', None, None, None]
12: ['=', '1.11', None, 'uno']
13: ['=', '2.22', None, 'dos']
14: ['*', 'uno', '-2', 't2']
15: ['/', '3.3', 'dos', 't1']
16: ['-', 't2', 't1', 't3']
17: ['=', 't3', None, 'dos']
18: ['>', 'dos', '0', 't3']
19: ['GoToF', 't3', None, 22]
20: ['=', 'uno', None, 'dos']
21: ['GoTo', None, None, 35]
22: ['=', '0', None, 'tres']
23: ['!=', 'tres', '3', 't3']
24: ['GoToF', 't3', None, 35]
25: ['+', 'tres', '1', 't3']
26: ['=', 't3', None, 'tres']
27: ['print', '"Ya casi es tres"', None, None]
28: ['print', 'tres', None, None]
29: ['ERA', 'funcioncita', None, None]
30: ['+', '1', '1', 't3']
31: ['PARAMETER', 't3', 0, None]
32: ['PARAMETER', 'uno', 1, None]
33: ['GOSUB', 'funcioncita', None, None]
```

```
34: ['GoTo', None, None, 23]
35: ['ERA', 'funcionsota', None, None]
36: ['PARAMETER', '2', 0, None]
37: ['PARAMETER', '5.5', 1, None]
38: ['GOSUB', 'funcionsota', None, None]
39: ['END', None, None, None]
```

Cabe recalcar que esta representación de código intermedio no traduce las variables, constantes y temporales a Direcciones virtuales. Para el diseño de DuckLang, se ha tomado la decisión de dejar la tarea de manejo de memoria con este formato a la máquina virtual encargada de correr el código de representación intermedia del archivo OBJ (directorio de funciones y cuádruplos), ya que es ahí donde se manejará los Activation Records, Data Segment, Execution Segment, y dónde se implementarán los algoritmos que interpretarán y ejecutarán estos códigos de operación.

Máquina Virtual

Para esta fase final del desarrollo de DuckLang, lo que se creó fue el código que ejecuta todo lo que se había compilado, se creó otra clase en un archivo aparte de Python que toma los cuádruplos y el directorio de funciones del paso anterior, y actúa como una máquina virtual. Se declara un diccionario que simula la memoria de la máquina ya que guarda adentro los segmentos (diccionarios) de ejecución de la función main y de cada activation record (cada función que se va mandando a llamar). También tiene un instruction pointer que va recorriendo los cuádruplos, un stack para guardar las posiciones pasadas del instruction pointer, y un stack que guarda las funciones de cada activation record que se fue creando para poder manejar llamadas recursivas y entre funciones para poder saber qué memoria necesita cada llamada y cuando regrese tenga la información necesaria para ejecutar.

También cuenta con una sección encargada de iniciar el espacio de cada variable según los recursos que necesite cada función, y se encarga de crear solo los espacios que son para variables o variables temporales. Después se mete a un ciclo que no termina hasta que llegue al final del programa o hasta que se encuentre

con un run-time error, y dentro de aquí se realizan más validaciones para saber que cada registro es llamado y actualizado en el segmento adecuado de la memoria realizando cada operación aritmética, o de asignación, imprimir, cambiar el instruction pointer, llamar funciones, entre otras tareas que le indican los cuádruplos. Cabe recalcar que se optimizó desde la compilación para usar el número mínimo de variables temporales, y al momento de ejecutarse es donde se nota la diferencia y los temporales se reutilizan mucho, se leen y reescriben constantemente sin necesidad de crear más.

A continuación se muestra un ejemplo de llamadas recursivas para implementar un código en DuckLang que calcula el factorial de un número. Se utilizan las variables globales para conseguir esto y poder ir actualizando el valor según se va ejecutando en la recursión:

```
programa factorial;

vars numero, resultado: entero;

nula factorial (n: entero) {
    {
        si(n > 1) {
            resultado = resultado * n;
            factorial(n - 1);
        };
    }
};

inicio {
    numero = 4;
    resultado = 1;
    factorial(numero);
    escribe("El factorial de", numero, "es");
    escribe(resultado);
} fin
```

Resultados

```
El factorial de
4
es
24
```

```
Memory:
{'__main__': {'numero': 4, 'resultado': 24}, 'factorial1': {'n': 4,
't1': 3}, 'factorial2': {'n': 3, 't1': 2}, 'factorial3': {'n': 2, 't1':
1}, 'factorial4': {'n': 1, 't1': False}}
```

Con esto concluye el alcance de DuckLang, y gracias a esto se pudo desarrollar satisfactoriamente todo lo que conlleva un compilador y una simulación de máquina virtual para ejecutar el código intermedio. Esto fue un gran reto de programación y da una visión diferente con respecto a cómo se puede crear código más adelante considerando cómo funcionan los compiladores para tener en mento como se puede optimizar los recursos y conocer a detalle los pasos que llevan en general para cada lenguaje moderno que se utiliza.

Para poder ver el código del compilador de Ducklang:
[Repositorio del desarrollo en Github]