

Liga de GitHub: [https://github.com/diegovelsal/webby\\_compilador](https://github.com/diegovelsal/webby_compilador)

## BabyDuck – Entrega #0

1.- Diseñar las Expresiones Regulares que representan a los diferentes elementos de léxico que ahí aparecen.

Token	Expresión Regular
ID	[a-z ][a-zA-Z0-9 ]*
CTE INT	[0-9] +
CTE FLOAT	[0-9] +\.[0-9] +
CTE STRING	“(.*?””
ADD	+
SUB	-
MUL	*
DIV	/
ASSIGN	=
NOTEQUAL	!=
LESS	<
GREATER	>
SEMICOLON	;
COLON	:
COMMA	,
LPAREN	(
RPAREN	)
LBRACE	{
RBRACE	}
LBRACK	[
RBRACK	]
PROGRAM	program
MAIN	main
END	end
VAR	var
PRINT	print
IF	if
ELSE	else
WHILE	while
DO	do
VOID	void
INT	int
FLOAT	float

2.- Listar todos los Tokens que serán reconocidos por el lenguaje

### Palabras Clave

- program
- main
- end
- if
- else
- while
- do
- print
- void
- type (int o float)

## Variables

- id
- cte\_int
- cte\_float
- cte\_string

## Operadores aritméticos

- +
- -
- \*
- /

## Operadores de comparación

- =
- !=
- <
- >

## Delimitadores

- (
- )
- {
- }
- [
- ]
- ;
- ,
- :

3.- Diseñar las reglas gramaticales (Context Free Grammar) equivalentes a los diagramas.

Regla	Definición
programa	PROGRAM ID SEMICOLON (vars)? (funcs_list)? MAIN body END
vars	VAR (var_decl)+
var_decl	id_list COLON type SEMICOLON
id_list	ID (COMMA ID)*
type	INT   FLOAT
params	ID COLON type (COMMA ID COLON type)*
funcs_list	funcs+
funcs	VOID ID LPAREN (params)? RPAREN LBRACK (vars)? body RBRACK SEMICOLON
body	LBRACE (statement)* RBRACE
statement	assign   condition   cycle   f_call   print
assign	ID ASSIGN expresion SEMICOLON
print	PRINT LPAREN print_args RPAREN SEMICOLON
print_args	print_arg (COMMA print_arg)*
print_arg	expression   CTE_STRING
condition	IF LPAREN expresion RPAREN body ELSE body SEMICOLON
cycle	WHILE LPAREN expresion RPAREN DO body SEMICOLON
expresion	exp ( LESS   GREATER   NOTEQUAL ) exp )*
exp	termino ( (ADD   SUB) termino )*
termino	factor ( (MUL   DIV) factor )*
factor	(ADD   SUB)? (ID   cte   LPAREN expresion RPAREN)
cte	CTE_INT   CTE_FLOAT
f_call	ID LPAREN args? RPAREN SEMICOLON
args	expresion (COMMA expresion)*

## BabyDuck – Entrega #1

Como parte del desarrollo del compilador para el lenguaje BabyDuck, esta primera entrega tuvo como objetivo comprender y aplicar herramientas automáticas para la generación de analizadores léxicos (scanners) y sintácticos (parsers). Para ello, se realizó una investigación sobre distintas opciones existentes, evaluando su documentación, facilidad de uso, compatibilidad con el lenguaje de programación que elegí para aprender (que en mi caso fue Java) y su potencial para integrarse en futuras etapas del compilador.

Las principales herramientas consideradas fueron ANTLR, Lex/Yacc, Flex/Bison y JavaCC. Lex/Yacc y Flex/Bison, aunque ampliamente utilizados en entornos de bajo nivel como C/C++, presentan una curva de aprendizaje más pronunciada y una sintaxis menos intuitiva. JavaCC, por otro lado, es una alternativa viable en Java, pero tiene una comunidad más reducida y menor cantidad de recursos actualizados. En cambio, ANTLR (Another Tool for Language Recognition) resultó ser la opción más adecuada debido a su amplia documentación, soporte activo, sintaxis clara y modular mediante archivos .g4, así

como su compatibilidad directa con Java. Por estas razones, se seleccionó ANTLR 4 como la herramienta principal para esta y futuras entregas del compilador BabyDuck.

La implementación del lenguaje comenzó con la separación del léxico y la sintaxis en archivos diferentes. En el archivo BabyDuckLexer.g4 se definieron todas las reglas gramaticales. Aquí se dividieron dos archivos, el primero siendo **WebbyLexer.g4** donde se definieron los tokens tal y como se escribieron en la entrega #0 (primera tabla). Por otra parte, en el archivo **WebbyParser.g4** se definió la gramática y fue redactada tal y como se especificó en la entrega #0 (segunda tabla). La implementación respetó el uso de reglas y ANTLR se encargó de generar automáticamente las clases Java correspondientes al lexer y parser.

Para validar el funcionamiento del compilador, se desarrolló un conjunto de once casos de prueba que cubren los elementos esenciales del lenguaje. Estos casos incluyen desde un programa mínimo con solo program y main, hasta declaraciones de variables, impresiones, operaciones aritméticas, estructuras de control de flujo y funciones. También se incluyó un caso con un error intencional de sintaxis (una instrucción sin punto y coma) para comprobar la detección de errores. Estos archivos se guardaron con la extensión .web y se procesaron mediante un programa Java que instanciaba el lexer y parser, y aplicaba un listener personalizado para detectar errores sintácticos de forma clara y confiable.

## Pruebas Realizadas

Se diseñaron cinco archivos .web con distintos escenarios para verificar que el análisis sintáctico del lenguaje funcione correctamente. Las pruebas se ejecutaron usando el archivo Test.java que recorre automáticamente los archivos .web del directorio de pruebas y reporta errores de sintaxis mediante un listener de ANTLR personalizado.

### test01.web – Programa mínimo

Este archivo representa el caso más simple: un programa con solo la estructura main vacía. Sirve para comprobar que la gramática acepta correctamente un programa sin declaraciones ni instrucciones.

- Sintaxis aceptada sin errores.

```
== Parsing file: test01.web ==
(programa program myProgram ; main (body { }) end)
? Sintaxis correcta.
```

### test02.web – Declaraciones de variables globales

Incluye varias declaraciones de variables globales de tipo int y float. Permite validar que la sección vars y la sintaxis de listas de identificadores y tipos estén correctamente definidos.

- Sintaxis aceptada sin errores.

```
== Parsing file: test02.web ==
(programa program myProgram ; (vars var (var_decl (id_list a , b , c) : (type int) ;) (var_decl (id_list x , y) : (type float) ;)) main (body { }) end)
? Sintaxis correcta.
```

### test03.web – Funciones con parámetros y cuerpo

Contiene dos funciones, una con parámetros y una sin ellos. Dentro de las funciones hay declaraciones locales, expresiones de asignación y un print. Esta prueba valida la sintaxis de funciones completas con cuerpo.

- Sintaxis aceptada sin errores.

```
== Parsing file: test03.web ==
(programa program testFuncs ; (vars var (var_decl (id_list a) : (type int) ;)) (funcs_list (funcs void suma ( (params a : (type int) , b : (type int)) ) [ (vars var (var_decl (id_list res) : (type int) ;)) (body { (statement (assign res = (expresion (expr (termino (factor a)) + (termino (factor b)))) ;)) } ) ] ;) (funcs void mensaje ( ) [ (body { (statement (print print ( (print_args (print_arg "Hola mundo") ) ;)) } ) ] ;) ) main (body { }) end)
? Sintaxis correcta.
```

### test04.web – Error por token no reconocido

Introduce un carácter inválido (~) dentro de una expresión. Este caso prueba que el lexer identifique correctamente tokens ilegales y que el parser reporte adecuadamente el error de sintaxis.

- Error detectado por token no reconocido.

```
== Parsing file: test04.web ==
line 3:8 token recognition error at: '~'
? Error de sintaxis en test04.web en la línea 3, columna 10: mismatched input '3' expecting {'+', '-', '*', '/', '!=', '<', '>', ';' }
(programa program exprProg ; main (body { (statement (assign a = (expresion (exp (termino (factor (cte 5)))) 3 * ( 2 - 1 ) ;)
) }) end)
? Sintaxis incorrecta.
```

### test05.web – Condicional incompleto

Contiene una estructura if válida en forma, pero sin incluir la rama else, que es obligatoria según la gramática definida. Esta prueba asegura que el parser exija todas las partes requeridas de una estructura condicional.

 Error de sintaxis por ausencia de else.

```
== Parsing file: test05.web ==
? Error de sintaxis en test05.web en la línea 6, columna 0: mismatched input '}' expecting 'else'
(programa program conditionTest ; main (body { (statement (condition if ( (expresion (exp (termino (factor a))) < (exp (termino (factor b)))) ) (body { (statement (print print ( (print_args (print_arg "menor")) ) ;)) })) } end)
? Sintaxis incorrecta.
```

## BabyDuck – Entrega #2

### Tablas de Consideraciones Semánticas

#### 1. Operaciones aritméticas (+ , - , \* , / )

Izquierda \ Derecha	INT	FLOAT
INT	INT	FLOAT
FLOAT	FLOAT	FLOAT

#### 2. Operaciones relacionales (< , > , != )

Izquierda \ Derecha	INT	FLOAT
INT	BOOL	BOOL
FLOAT	BOOL	BOOL

#### 3. Asignación (= )

Izquierda \ Derecha	INT	FLOAT
INT	INT	ERROR
FLOAT	FLOAT	FLOAT

### DirFunc y VarTable

Para gestionar las funciones del lenguaje y sus respectivos ámbitos de variables, se diseñó la clase DirFunc. Esta clase utiliza como estructura principal un `HashMap<String, VarTable>` llamado `functions`, donde cada llave representa el nombre de una función (incluyendo el ámbito global) y el valor es una instancia de `VarTable`, que contiene las variables locales a dicha función. Esta estructura permite realizar operaciones como búsquedas, inserciones y validaciones de manera eficiente, en tiempo constante promedio ( $O(1)$ ), lo cual es ideal para un lenguaje donde las funciones pueden ser accedidas frecuentemente por nombre.

Además de la tabla de funciones, DirFunc mantiene dos atributos auxiliares: globalScopeName, que almacena el nombre del ámbito global (típicamente el nombre del programa), y currentFunction, que indica en qué función se encuentra actualmente el compilador o el analizador semántico. Esto permite que las operaciones sobre variables, como inserciones o validaciones, se realicen directamente sobre el contexto correcto sin necesidad de pasarlo explícitamente en cada operación.

Las operaciones principales de esta clase incluyen la creación de nuevas funciones (addFunction), el cambio de contexto (setCurrentFunction), la inserción de variables en la función actual (addVariable), y la validación de existencia de variables tanto en el ámbito local como global. Esta organización permite mantener una jerarquía clara y un control estricto sobre los ámbitos, además de facilitar la verificación semántica en múltiples fases del compilador.

Por otra parte, la clase VarTable se encarga de representar el conjunto de variables declaradas dentro del ámbito de una función o del ámbito global. Internamente utiliza un `HashMap<String, VarInfo>` llamado variables, donde cada llave es el nombre de una variable y su valor es una instancia de VarInfo, clase interna que encapsula tanto el tipo (VarType) como el valor actual de la variable. Esta estructura permite realizar búsquedas, inserciones y actualizaciones de forma eficiente, en tiempo constante promedio, ideal para entornos donde se espera acceder frecuentemente a las variables por nombre.

El uso de la clase auxiliar VarInfo proporciona flexibilidad y escalabilidad, ya que permite almacenar más información relacionada a la variable en el futuro (por ejemplo, dirección de memoria, tipo de acceso, etc.), sin modificar la lógica principal de la tabla. Entre las operaciones principales de VarTable se encuentran: agregar una nueva variable (addVariable), obtener su tipo (getVariableType), leer o modificar su valor (getVariableValue y setVariableValue), y verificar su existencia (hasVariable). Todas estas funciones están diseñadas para asegurar la consistencia del ámbito, evitando redefiniciones y permitiendo un control estricto sobre las declaraciones y usos válidos de variables en el lenguaje.

En conjunto con DirFunc, esta clase permite modelar de manera clara y eficiente los diferentes niveles de ámbito que existen en un lenguaje con funciones, diferenciando entre variables locales y globales, y facilitando la implementación de reglas semánticas como la detección de dobles declaraciones o el uso de variables no definidas.

## Puntos neurálgicos y Validaciones

Para establecer ahora si las funciones previas al revisar el código se uso un validador semántico de ANTLR. El SemanticVisitor es una clase que extiende de WebbyParserBaseVisitor<Void> y se encarga de recorrer el árbol sintáctico generado por ANTLR para realizar las validaciones semánticas necesarias del lenguaje. Su función principal es verificar que el código fuente cumpla con las reglas semánticas definidas para variables, funciones y estructuras de control, además de registrar la información necesaria para la ejecución o traducción del programa. Para ello, utiliza una instancia de DirFunc, la cual encapsula toda la información de ámbito y declaración de variables y funciones, separando claramente el contexto global del contexto local de cada función.

Durante el recorrido del árbol, el SemanticVisitor primero registra el nombre del programa como función principal y crea una nueva instancia del directorio de funciones. Después, visita las declaraciones de variables globales, funciones y finalmente el bloque principal del programa. Una de las primeras validaciones importantes es asegurar que no haya variables duplicadas dentro del mismo ámbito (local o global). En el caso de funciones, también se verifica que no existan funciones duplicadas. Cuando se detecta una redefinición de variable o función, se lanza una excepción con un mensaje claro de error.

En cuanto a las funciones, SemanticVisitor revisa que todos los parámetros estén correctamente definidos y que no se repitan. Al registrar parámetros o variables locales, se realiza una verificación del tipo de dato utilizando el enumerador VarType, y se almacenan en la tabla de variables del ámbito correspondiente. Esto asegura que cada variable esté correctamente tipada y declarada antes de ser utilizada.

Dentro de las expresiones, como las asignaciones o factores (por ejemplo, variables usadas en cálculos), se valida que todas las variables hayan sido declaradas previamente, ya sea en el ámbito local o global. En caso contrario, se lanza un error señalando el uso indebido de una variable no declarada. Este control permite prevenir errores de ejecución y mejora la robustez del lenguaje.

## Pruebas Realizadas

Se llevaron a cabo tres pruebas específicas para validar la capacidad del compilador de detectar errores semánticos, pese a que el análisis sintáctico fue exitoso en todos los casos. Los archivos .web fueron construidos sin errores de gramática, pero con problemas relacionados con las reglas del contexto, como declaraciones duplicadas o usos indebidos de identificadores.

### test01.web – Función doblemente declarada

Este archivo define dos funciones con el mismo identificador hola. Ambas funciones tienen firmas idénticas y se encuentran en el mismo ámbito global.

- ✗ Error semántico: Función 'hola' ya fue declarada.

```
== Parsing file: test01.web ==
(programa program programDuplicado ; (funcs_list (funcs void hola ( ) [ (body { (statement (print print ( (print_args (print_arg "Hola mundo")) ) ;)) }) ] ;) (funcs void hola ( ) [ (body { (statement (print print ( (print_args (print_arg "Otra versión"
)) ) ;)) }) ] ;)) main (body { }) end)
? Sintaxis correcta.

? Error parsing test01.web: Error: Función 'hola' ya fue declarada.
```

### test02.web – Variable doblemente declarada

Se intenta declarar la variable a dos veces como variable global, primero como int y luego como float.

- ✗ Error semántico: La variable 'a' ya ha sido declarada en el ámbito local.

```
== Parsing file: test02.web ==
(programa program varDuplicada ; (vars var (var_decl (id_list a) : (type int) ;) (var_decl (id_list a) : (type float) ;)) main
(body { }) end)
? Sintaxis correcta.

? Error parsing test02.web: La variable 'a' ya ha sido declarada en el ámbito local.
```

### test03.web – Uso de variable no declarada

Dentro del bloque main, se realiza una asignación a la variable a sin haberla declarado previamente en ninguna parte del programa.

- ✗ Error semántico: Variable 'a' usada en asignación sin ser declarada.

```
== Parsing file: test03.web ==
(programa program sinDeclarar ; main (body { (statement (assign a = (expresion (exp (termino (factor (cte 5))))))) }) end)
? Sintaxis correcta.

? Error parsing test03.web: Error: Variable 'a' usada en asignación sin ser declarada.
```

Estas pruebas confirman que el análisis semántico del compilador identifica correctamente violaciones al contexto del lenguaje, incluso si el programa es sintácticamente válido.

## BabyDuck – Entrega #3

### Estructuras para Traducción a Cuádruplos

#### Clase Quadruple

La clase Quadruple representa la estructura base utilizada para traducir instrucciones del lenguaje BabyDuck a una forma intermedia conocida como cuádruplos. Cada cuádruplo consiste en cuatro elementos: una operación (op), dos operandos (arg1, arg2) y un resultado (res). Esta representación simplifica la generación de código intermedio al permitir una forma uniforme y flexible para describir operaciones aritméticas, asignaciones,

condiciones, impresiones y más. Esta clase también sobrescribe el método `toString()` para permitir una impresión legible del cuádruplo, útil para pruebas e inspección manual.

## Clase **SemanticStackContext**

Durante la generación de cuádruplos, es necesario manejar múltiples pilas de trabajo para mantener el estado semántico de manera estructurada. La clase **SemanticStackContext** encapsula las siguientes pilas:

- **operandStack**: almacena los operandos (nombres de variables o constantes) usados en las expresiones.
- **operatorStack**: guarda los operadores aritméticos o relacionales en espera de evaluación.
- **typeStack**: lleva el seguimiento del tipo de cada operando (por ejemplo, INT, FLOAT) para realizar validaciones semánticas usando las tablas de consideraciones.

El uso de estas pilas permite transformar expresiones infijas en cuádruplos, generando instrucciones intermedias paso a paso conforme se recorren los nodos del árbol de análisis sintáctico.

## Cambios al **SemanticVisitor** para Generación de Cuádruplos

Además del análisis semántico previamente implementado, el **SemanticVisitor** fue extendido para traducir expresiones del lenguaje BabyDuck a cuádruplos.

Primeramente, se introdujeron variables nuevas donde se destacan:

- **quadruples**: lista que acumula todos los cuádruplos generados hasta el momento.
- **tempVarCounter**: contador para nombrar variables temporales generadas durante la evaluación de expresiones complejas.

Además, se instancia la clase **SemanticStackContext** para tener nuestras pilas listas para utilizar.

Posterior a eso se hicieron cambios en los métodos que conforman este. Este proceso ocurre dentro de los métodos relacionados con:

- Asignaciones (`visitAssign`)
- Factores y términos (`visitFactor`, `visitTermino`, `visitExp`)
- Prints (`visitPrint_arg`)

Cada vez que se detecta una operación aritmética o una asignación, se realiza lo siguiente:

- Se colocan operandos y operadores en sus respectivas pilas.

- Al finalizar la subexpresión, se "resuelve" la operación, generando un cuádruplo.
- Se asigna una variable temporal (por ejemplo, t0, t1, etc.) para guardar el resultado.
- Este resultado se vuelve a colocar en la pila de operandos para su uso en operaciones futuras.

Finalmente, en los cuadruplos también se incluyen los prints de el resultado final que se manda a imprimir (o por consiguiente el letrero).

El resultado final es una lista ordenada de cuádruplos que puede utilizarse para interpretación directa o generación de código más adelante. La estructura generada es clara, lineal y respeta las reglas de precedencia de operadores del lenguaje.

## Pruebas Realizadas

Se llevaron a cabo tres archivos de prueba para verificar la correcta generación de cuádruplos en distintos contextos del lenguaje. A continuación, se describen brevemente:

### test01.web – Asignaciones complejas

Este archivo contiene varias asignaciones que mezclan operaciones aritméticas simples y compuestas. Se genera correctamente una secuencia de cuádruplos que reflejan el orden y jerarquía de operaciones.

```
*** Parsing file: test01.web ***
(programa program asignacionCompleja ; (vars var (var_decl (id_list a , b , c , d , resultado) : (type int) ;)) main (body { (statement (assign a = (exp (termino (factor (cte 5)))) ;))
(statement (assign b = (exp (termino (factor (cte 10)))) ;))) (statement (assign c = (exp (termino (factor (cte 2)))) ;))) (statement (assign d = (exp (termino (factor (cte 3)))) ;))) (statement (assign resultado = (exp (exp (termino (factor a)) + (termino (factor b) * (factor c)) - (termino (factor (cte 1)))))))) ;)) end
? Parsed successfully.

Cuádruplos generados:
(=, 5, _, a)
(=, 10, _, b)
(=, 2, _, c)
(=, 3, _, d)
(+, b, c, t0)
(+, a, t0, t1)
(-, a, 1, t2)
(/, d, t2, t3)
(-, t1, t3, t4)
(=, t4, _, resultado)
```

### test02.web – Print con operaciones

Este ejemplo incluye una instrucción print que imprime tanto texto como el resultado de una expresión aritmética. Se verifica que el cuádruplo PRINT se pueda usar tanto para cadenas como para resultados de operaciones.

```
*** Parsing file: test02.web ***
(programa program printComplejo ; (vars var (var_decl {id_list x , y , z} : (type int) ;)) main (body { (statement (assign x = (expresion (exp (termino (factor (cte 4))))))) ;) (statement (assign y = (expresion (exp (termino (factor (cte 8))))))) ;) (statement (assign z = (expresion (exp (termino (factor (cte 2))))))) ;) (statement (print print ( (print_args (print_arg "Resultado: ") , (print_arg (expression (exp (termino (factor x) * (factor y))) + (termino (factor z)) / (factor ( (expresion (exp (termino (factor x)) - (termino (factor (cte 2))))))) )))))) ) ;) }) end)
? Parsed successfully.

Cuádruplos generados:
(=, 4, _, x)
(=, 8, _, y)
(=, 2, _, z)
(PRINT, _, _, "Resultado: ")
(*, x, y, t0)
(-, x, 2, t1)
(/, z, t1, t2)
(+, t0, t2, t3)
(PRINT, _, _, t3)
```

### test03.web – Condicional con comparación

Este archivo prueba la evaluación de una condición con operadores relacionales y dos bloques if / else. Se generan correctamente los cuádruplos para ambas ramas, evaluando primero la condición y después cada cuerpo.

```
*** Parsing file: test03.web ***
(programa program ifComplejo ; (vars var (var_decl {id_list m , n , p} : (type int) ;)) main (body { (statement (assign m = (expresion (exp (termino (factor (cte 6))))))) ;) (statement (assign n = (expresion (exp (termino (factor (cte 3))))))) ;) (statement (assign p = (expresion (exp (termino (factor (cte 10))))))) ;) (statement (condition if ( (expresion (exp (termino (factor ( (expresion (exp (termino (factor m) * (factor n)) + (termino (factor (cte 4))))))) )))) > (exp (termino (factor ( (expresion (exp (termino (factor p))) - (termino (factor n) / (factor (cte 2))))))) )))) ;) (body { (statement (print print ( (print_args (print_arg (expression (exp (termino (factor m)) + (termino (factor n)) / (termino (factor p))))))) ;) )) ;) else (body { (statement (print print ( (print_args (print_arg No cumple)))) ;) )) ;) }) end)
? Parsed successfully.

Cuádruplos generados:
(=, 6, _, m)
(=, 3, _, n)
(=, 10, _, p)
(*, m, n, t0)
(+, t0, 4, t1)
(/, n, 2, t2)
(-, p, t2, t3)
(>, t1, t3, t4)
(+, m, n, t5)
(+, t5, p, t6)
(PRINT, _, _, t6)
(PRINT, _, _, "No cumple")
```

## ¿Cómo ejecutar las pruebas?

Las pruebas son ejecutadas usando:

- javac -cp ".;libs/antlr-4.13.2-complete.jar" src/lex\_par/\*.java src/sem/exp/\*.java src/sem/funcs\_vars/\*.java src/sem/\*.java src/\*.java
- java -cp ".;libs/antlr-4.13.2-complete.jar;src" Test

Cabe aclarar que en el archivo Test.java es necesario cambiar el path de la línea 19 a las pruebas que se deseen ejecutar.