Diego Velázquez Saldaña A01177877

Liga de GitHub: 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.

Test Cases

C:\Users\diego\OneDrive\Escritorio\Tec\8toSemestre\Compilador\RetoCompilador\entrega 1> java -cp ".;antlr-4.13.2-complete.jar" TestWebby

--- Probando archivo: tests/test1.web ---

? Sintaxis válida

Árbol: (programa program myProgram ; vars funcs main (body { statement }) end)

--- Probando archivo: tests/test2.web ---

? Sintaxis válida

Árbol: (programa program testVars ; (vars var : (tipo int) (id_list x , y , z) ; vars) funcs main (body { statement }) end)

--- Probando archivo: tests/test3.web ---

? Sintaxis válida

Árbol: (programa program printProgram ; (vars var : (tipo float) (id_list pi) ; vars) funcs main (body { (statement (print_stmt print ((expresion (exp (termino (factor pi))))) ;)) }) end)

--- Probando archivo: tests/test4.web ---

? Sintaxis válida

Árbol: (programa program mathProgram ; (vars var : (tipo int) (id_list a , b) ; vars) funcs main (body { (statement (assign a = (expresion (exp (termino (factor (cte 5))))) ;)) b = a + 10 * (a - 3) ; }) end)

--- Probando archivo: tests/test5.web ---

? Sintaxis válida

Árbol: (programa program mathProgram ; (vars var : (tipo int) (id_list a , b) ; vars) funcs main (body { (statement (assign a = (expresion (exp (termino (factor (cte 5))))) ;)) b = a + 10 * (a - 3) ; }) end)

--- Probando archivo: tests/test6.web ---

? Sintaxis válida

Árbol: (programa program loopProgram ; (vars var : (tipo int) (id_list counter) ; vars) funcs main (body { (statement (cycle while ((expresion (exp (termino (factor counter))) < (exp (termino (factor (cte 10)))))) do (body { (statement (assign counter = (expresion (exp (termino (factor counter)) + (termino (factor (cte 1))))) ;)) })) }) end)

--- Probando archivo: tests/test7.web ---

? Sintaxis válida

Árbol: (programa program funcProgram ; vars (funcs void myFunction ((tipo int) param) : (vars var : (tipo int) (id_list temp) ; vars) (body { (statement (assign temp = (expresion (exp (termino (factor param)) + (termino (factor (cte 1))))) ;)) }) funcs) main (body { statement }) end)

--- Probando archivo: tests/test8.web ---

? Error en archivo:

Árbol: (programa program errorProgram <missing ';'> vars funcs main (body { statement }) end)

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

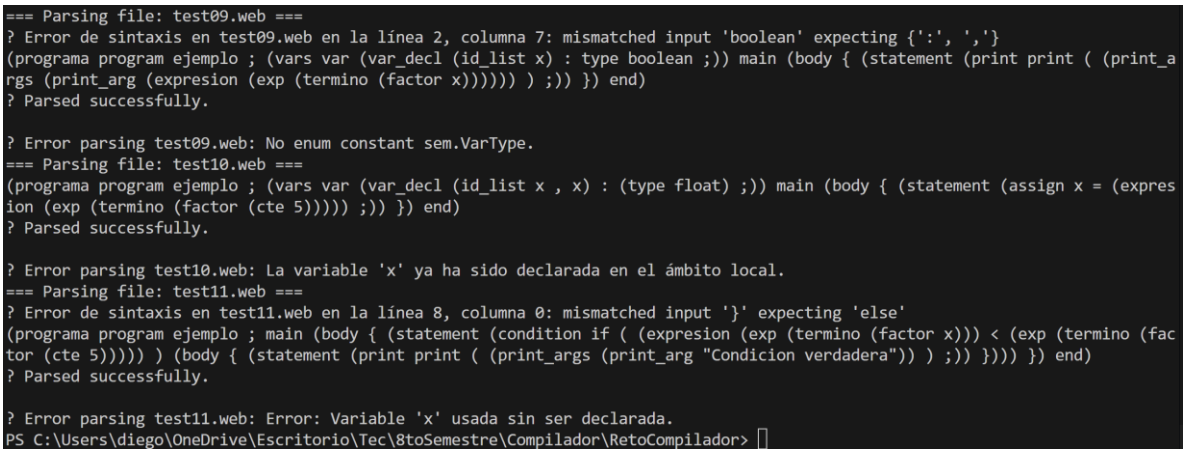
ANTLR para realizar las validaciones semánticas necesarias del lenguaje. Su función principal es verificar que el código fuente cumpla con las reglas semánticas definidas para variables, funciones y estructuras de control, además de registrar la información necesaria para la ejecución o traducción del 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.

Finalmente, se hicieron algunas pruebas de análisis semántico y aquí los resultados:



Para mayor detalle en el GitHub se pueden observar los archivos, pero básicamente en el primero se declaro un tipo de variable que no existe (boolean), que ya previamente falla desde la sintaxis pero que correctamente la semántica también corrige. Para el ejemplo 10 vemos una variable declarada doblemente y para el ejemplo 11 vemos una variable que no había sido previamente declarada.