Tecnológico de Monterrey

Instituto Tecnológico y de Estudios

Superiores de Monterrey

Desarrollo de aplicaciones avanzadas de ciencias computacionales

(TC3002B.505)

Prof. Elda Quiroga

“Desarrollo de lenguaje BabyDuck”

Ana Carolina Ramírez González

A00833324

01/06/2025
Monterrey, N.L.

En este documento se detalla el diseño e implementación de BabyDuck, un lenguaje de programación desarrollado en Java con ANTLR v4.1. Se cubrirán desde el análisis léxico y sintáctico hasta la semántica y la generación de código intermedio (cuádruplos).

Se eligió ANTLR v4 por su integración nativa con Java, su amplia documentación y la capacidad de definir léxico y sintaxis en un único archivo.

1. Formato de implementación

Toda la gramática del lenguaje (análisis léxico y sintáctico) se concentra en BabyDuck.g4.

a. **Sección léxica (tokens):**
   Definida en mayúsculas, usando expresiones regulares compatibles con ANTLR. Por ejemplo:

```
ID       : [a-zA-Z_][a-zA-Z0-9_]* ;
CTE_INT   : [0-9]+ ;
CTE_FLOAT : [0-9]+ '.' [0-9]+ ;
CTE_STRING: "" (~["\\] | '\\' .)* "" ;
```

b. **Sección sintáctica (reglas EBNF):**
   Definida en minúsculas, refleja los diagramas del lenguaje. Por ejemplo:

```
programa : 'program' ID ';' vars funcs 'main' body 'end' ;
vars     : ( 'var' idList ':' type ';' )* ;
body     : '{' statement* '}' ;
```

   La gramática sigue una estructura modular, esto es declaraciones de variables, definición de funciones, cuerpo del programa y sentencias, tal como se especificó en los diagramas de BabyDuck.

2. Expresiones Regulares y Tokens

A continuación se presenta la gramática completa del lenguaje, acompañada de las tablas de tokens y de las reglas sintácticas.

Identificadores y constantes:

TOKEN	Expresión Regular
id	[a-zA-Z_][a-zA-Z0-9_]*

cte_int	[0-9]+
cte_float	[0-9]+\.[0-9]+
cte_string	"([^"\\]|\\.)*"

Palabras reservadas:

TOKEN	Expresión Regular
program	‘program’
main	‘main’
end	‘end’
var	‘var’
int	‘int’
float	‘float’
print	‘print’
while	‘while’
do	‘do’
if	‘if’
else	‘else’
void	‘void’

Operaciones y símbolos:

TOKEN	Expresión Regular
plus	‘\+’
minus	‘-’
times	‘*’
divided	‘/’
semicolon	‘;’
colon	‘:’
comma	‘,’
lthan	‘<’

gthan	‘>’
lparen	‘\(‘
rparen	‘\)’
lcurly	‘\{’
rcurly	‘\}’
neq	‘!=’

➔ **Gramática**

Program start	<programa> ::= "program" ID ";" <vars> <funcs> "main" <body> "end"
Variable declaration	<vars> ::= (VAR <idList> COLON <type> SEMICOLON)*
	<idList> ::= ID (COMMA ID)*
	<type> ::= INT | FLOAT
Functions	<paramList> ::= ID COLON <type> (COMMA ID COLON <type>)*
	<funcs> ::= (VOID ID LPAREN <paramList>? RPAREN <vars> <body> SEMICOLON)*
Instructions	<body> ::= LCURLY <statement>* RCURLY
	<statement> ::= <assign> | <print> | <condition> | <cycle> | <f_call>
Assigning	<assign> ::= ID ASSIGN <expresion> SEMICOLON
Print	<print> ::= PRINT LPAREN (<expresion> | CTE_STRING) (COMMA (<expresion> | CTE_STRING))* RPAREN SEMICOLON
Conditionals	<condition> ::= IF LPAREN <expresion> RPAREN <body> (ELSE <body>)?
	<cycle> ::= WHILE LPAREN <expresion> RPAREN DO <body>
Function call	<f_call> ::= ID LPAREN (<expresion> (COMMA <expresion>)*)? RPAREN SEMICOLON

Expressions	<expresion> ::= <exp> (<oprel> <exp>)?
	<oprel> ::= NEQ | LTHAN | GTHAN
Arithmetic Operations	<exp> ::= <termino> ((PLUS | MINUS) <termino>)*
	<termino> ::= <factor> ((TIMES | DIVIDED) <factor>)*
Factors & Constants	<factor> ::= LPAREN <expresion> RPAREN | (PLUS | MINUS)? (ID | <cte>)
	<cte> ::= CTE_INT | CTE_FLOAT

3. Casos de Prueba:

A continuación se presenta una tabla con ejemplos de entrada (programa BabyDuck), los tokens principales que se esperan y si debe pasar o fallar el análisis léxico/sintáctico. Estos casos reflejan exactamente la gramática y el comportamiento actual del parser:

Nº	Descripción	Entrada	Tokens principales esperados	¿Pasa?
1	Programa mínimo válido	program foo; main { } end	PROGRAM ID SEMICOLON MAIN LCURLY RCURLY END	✓
2	Declaración de variables globales	program p; var x,y : int; main{} end	PROGRAM ID SEMICOLON VAR ID COMMA ID COLON INT SEMICOLON MAIN LCURLY RCURLY END	✓
3	Declaración de float y enteros	program p; var a,b:float;c:int; main{} end	VAR ID COLON FLOAT SEMICOLON VAR ID COLON INT SEMICOLON	✓
4	Asignación simple	x = 42;	ID ASSIGN CTE_INT SEMICOLON	✓

5	Constante float	x = 3.14;	ID ASSIGN CTE_FLOAT SEMICOLON	✓
6	Print de expresiones y strings	print(x, "hola", 2.0);	PRINT LPAREN ID COMMA CTE_STRING COMMA CTE_FLOAT RPAREN SEMICOLON	✓
7	If-else básico	if(x<0){print(x);}else{print(0);}	IF LPAREN ID LTHAN CTE_INT RPAREN LCURLY ... ELSE LCURLY ...	✓
8	While básico	while(i!=n) do { i=i-1; }	WHILE LPAREN ID NEQ ID RPAREN DO LCURLY ID ASSIGN ID MINUS ID SEMICOLON RCURLY	✓
9	Función sin parámetros	void f() var x:int;{print(x);} ;	VOID ID LPAREN RPAREN VAR ID COLON INT SEMICOLON LCURLY PRINT LPAREN ID RPAREN SEMICOLON RCURLY SEMICOLON	✓
10	Función con parámetros	void g(a:float,b:int){}	VOID ID LPAREN ID COLON FLOAT COMMA ID COLON INT RPAREN LCURLY RCURLY	✓
11	Llamada a función	foo(1,2.0);	ID LPAREN CTE_INT COMMA CTE_FLOAT RPAREN SEMICOLON	✓
12	Expresión compleja	x = (a+3)*b/2;	LPAREN ID PLUS CTE_INT RPAREN TIMES ID DIVIDED CTE_INT	✓

13	Falta de punto y coma	x = 5	—	✗
14	Paréntesis sin cerrar	print((x);	—	✗

4. Estructuras de Datos para el Análisis Semántico

4.1. FunctionDirectory

- Propósito: Mantener un directorio de todas las funciones declaradas (incluida "global") y, para cada función, su lista de parámetros y tabla de variables locales.
- Métodos principales:
  - boolean addFunction(String name, String returnType): agrega una nueva FunctionInfo a dir. Si ya existe, devuelve false.
  - boolean addVariableToCurrent(String name, String type, int address): inserta una variable en la tabla de variables de la función actual (currentFunction).
  - FunctionInfo getFunction(String name): devuelve la instancia FunctionInfo asociada a name o null si no existe.
  - void setCurrentFunction(String name): cambia el contexto actual (afecta dónde se insertarían nuevas variables).

4.2. FunctionInfo

- Propósito: Modelar la información de cada función individual: tipo de retorno, lista de parámetros (nombres y tipos) y su propia tabla de variables locales.
- Métodos principales:
  - void addParam(String name, String type, int address): registra el parámetro name (con tipo type) en paramNames, paramTypes y añade new VariableInfo(type, address, "local") a varTable.
  - boolean addVariable(String name, VariableInfo info): inserta VariableInfo para name en varTable. Devuelve false si ya existía.
  - VariableInfo getVariable(String name): obtiene la VariableInfo o null si no existe.

4.3. VariableInfo

- Propósito: Encapsular los datos asociados a cada variable (global o local).

5. Generación de Objetos Temporales y Constantes

5.1. VirtualMemoryManager

- Propósito: Asignar rangos de direcciones virtuales a distintas categorías:
- Globales: enteros [1000..1999], floats [2000..2999].
- Locales: enteros [3000..3999], floats [4000..4999].
- Temporales: enteros [5000..5999], floats [6000..6999].
- Constantes: enteros [7000..7999], floats [8000..8999].

- Métodos principales:
  - int allocGlobal(String type): devuelve la siguiente dirección global según type.
  - int allocLocal(String type): devuelve la siguiente dirección local según type.
  - int allocTemp(String type): devuelve nueva dirección temporal según type.
  - int allocConst(String value, String type): si type:value no está en constTable, asigna nueva dirección constante; devuelve siempre el mismo address para esa constante.

6. Análisis Semántico y Generación de Cuádruplos

6.1. SemanticVisitor (extiende BabyDuckBaseVisitor<Void>)

Objetivos principales:

1. Registrar funciones y variables en FunctionDirectory.
2. Verificar existencia de variables (si no están declaradas, arrojar RuntimeException).
3. Invocar generación de cuádruplos en QuadrupleGenerator durante el recorrido de expresiones, asignaciones, impresiones, llamadas a funciones, condicionales y ciclos.

6.1.1. visitPrograma

- Genera un GOTO L0 para saltar a la sección main.
- Procesa declaraciones de variables globales (llama a visitVars).
- Procesa funciones (llama a visitFuncs).
- Marca la etiqueta L0 y recorre el cuerpo de main.

6.1.2. visitVars

- Cada variable global se reserva con vmm.allocGlobal(type).
  Registra la variable (nombre, tipo y dirección) en el directorio bajo “global”.

6.1.3. visitFuncs

- Genera LABEL funcName al inicio de la función.
- Registra la nueva función en FunctionDirectory.
- Cambia el contexto a currentFunction = funcName.
  Para cada parámetro:
    - Reserva dirección local con vmm.allocLocal(type).
    - Registra el parámetro en la tabla de variables de la función.
- Procesa variables locales y cuerpo (llama a visit(vars) y visit(body)).
- Al cerrar la función:
    - Genera ENDPROC.
    - Restaura el contexto a “global”.

6.1.4. visitAssign

- Verifica que la variable destino exista en el contexto actual o en “global”.
- Llama a visit(expresion) para generar cuádruplos de la expresión derecha; deja en la pila la dirección del resultado.
- Invoca qg.generateAssignmentAddress(destAddr), que consume esa dirección y encola el cuádruplo
  ("=", dirExpr, "-", destAddr).

6.1.5. visitPrint

- Para cada expresión numérica: llama a visit(expresion) y luego a qg.generatePrint(), que encola ("PRINT", dirExpr, "-", "-").
- Para cada literal de cadena (CTE_STRING): llama a qg.generatePrintString(text), que encola ("PRINT_STR", text, "-", "-").

6.1.6. visitCondition

- Llama a visit(expresion) y obtiene la dirección del resultado con qg.peekOperand().
- Genera GOTOF condTemp, Lf.
- Recorre el bloque “then” con visit(body(0)).
- Si existe else:
    1. Crea etiqueta Le.
    2. Genera GOTO Le.
    3. Marca Lf.
    4. Recorre el bloque “else” con visit(body(1)).
    5. Marca Le.
- Si no hay else, marca directamente Lf.

6.1.7. visitCycle

- Crea y marca etiqueta start.

- Llama a visit(expresion) y obtiene la dirección de la condición.
- Genera GOTOF condTemp, end.
- Recorre el cuerpo del bucle con visit(body).
- Al finalizar, genera GOTO start.
- Marca la etiqueta end.

6.1.8. visitF_call

- Genera ERA funcName para preparar el activation record.
- Para cada argumento en la llamada:
    1. Llama a visit(e).
    2. Llama a qg.generateParam(), que extrae la dirección y encola ("PARAM", dirArg, "-", "-").
- Finalmente, genera GOSUB funcName para saltar a la función.

6.1.9. visitExp

- Visita termino(0).
- Para cada término adicional:
    1. Obtiene el operador (“+” o “−”) y hace qg.pushOperator(op).
    2. Llama a visit(termino(i)).
    3. Invoca qg.generateBinaryQuad() para generar el cuádruplo.

6.1.10. visitTermino

- Visita factor(0).
- Para cada factor adicional:
    1. Obtiene el operador (“*” o “/”) y hace qg.pushOperator(op).
    2. Llama a visit(factor(i)).
    3. Invoca qg.generateBinaryQuad().

6.1.11. visitFactor

- Si hay paréntesis: llama a visit(expresion).
- Si es una constante numérica:
    1. Determina el tipo (“int” o “float”).
    2. Reserva dirección con vmm.allocConst(val, type).
    3. Llama a qg.pushOperandAddress(addr, type).
- Si es identificador:
    1. Busca VariableInfo en el contexto actual o en “global”.
    2. Llama a qg.pushOperandAddress(vi.address, vi.type).
- Esto deja en la pila la dirección lista para operaciones posteriores.

7. Generación de Cuádruplos

En *QuadrupleGenerator* se emplea una lógica basada en tres pilas (operandos, operadores y tipos) y una cola FIFO (quads) para almacenar las instrucciones intermedias. A continuación se listan de manera resumida los métodos más relevantes:

1. **pushOperandAddress(int address, String type)**
   Empuja en la pila de operandos la dirección address (representa un literal, variable o temporal) y en la pila de tipos su type (“int” o “float”).
   - Se invoca cada vez que *SemanticVisitor* encuentra un literal numérico o una variable, para preparar su uso en operaciones posteriores.

2. **pushOperator(String op)**
   - Empuja en la pila de operadores el símbolo op (“+”, “−”, “*”, “/”, “<”, “>”, “!=”, etc.).
   - Se llama justo antes de visitar la segunda parte de una subexpresión binaria (aritmética o relacional).

3. **generateBinaryQuad()**
   - Extrae de las pilas:
     - El operando derecho y su tipo.
     - El operando izquierdo y su tipo.
       El operador op.
   - Determina el tipo de resultado (resType = "float" si alguno es float; de lo contrario, “int”).
   - Reserva una nueva dirección temporal: tempAddr = vmm.allocTemp(resType).
   - Empuja tempAddr y resType en las pilas de operandos/tipos para futuras operaciones.
   - Encola en quads un objeto Cuadruplo(op, dirIzq, dirDer, dirTemp).

4. **generateAssignmentAddress(int destAddr)**
   - Extrae de la pila de operandos la dirección del valor a asignar.
     Encola en quads un cuádruplo

5. **generatePrint() y generatePrintString(String text)**
   - **generatePrint():** extrae de la pila de operandos la dirección con el valor numérico a imprimir y genera
   - **generatePrintString(text):** encola directamente para literales de cadena.

8. Máquina Virtual

La Máquina Virtual constituye el componente final del compilador y se encarga de interpretar la lista de cuádruplos generados durante la fase de generación de código intermedio. Para ello, gestiona tres espacios de memoria diferenciados: global, local y temporal, cada uno con su propio rango de direcciones. Las variables globales y constantes persisten a lo largo de toda la ejecución; las variables locales son específicas de cada función; y las temporales almacenan resultados intermedios.

Previo a la ejecución, la Máquina Virtual realiza un primer recorrido sobre la lista de cuádruplos para identificar y registrar todas las etiquetas (LABEL) asociadas a saltos de control (GOTO, GOTOF). Esto permite construir un mapa de etiquetas que se utilizará durante la ejecución para resolver los saltos.

Durante la ejecución (segundo recorrido), cada cuádruplo es interpretado en función de su operador:

- Operaciones aritméticas y relacionales (como +, -, *, /, <, >, !=) recuperan los operandos desde las direcciones de memoria indicadas, realizan la operación correspondiente y almacenan el resultado en una dirección temporal.
- Asignaciones (=) copian valores de una dirección a otra.
- Cuádruplos de impresión (PRINT, PRINT_STR) muestran valores numéricos o cadenas de texto directamente al usuario.
- Saltos incondicionales (GOTO) y condicionales (GOTOF) alteran el contador de programa dependiendo de condiciones evaluadas en tiempo de ejecución.
- Llamadas a función utilizan una secuencia de cuádruplos especializados:
    - ERA prepara el entorno de ejecución
    - PARAM transfiere argumentos
    - GOSUB realiza el salto a la función, guardando el estado actual del programa
    - ENDPROC indica el fin de la función, restaurando los entornos de memoria y el punto de retorno

Este modelo de ejecución acepta el manejo de múltiples llamadas a funciones anidadas, manteniendo una pila para memorias locales, temporales y direcciones de retorno. El uso de estructuras auxiliares como pendingParams, returnStack y labelMap permite simular de manera eficiente una pila de activación funcional.

En cuanto a la gestión de memoria virtual, se emplea un asignador que otorga direcciones según el tipo y ámbito de la variable:

- Las direcciones para enteros y flotantes globales comienzan en 1000 y 2000, respectivamente.
- Las direcciones locales inician en 3000 (int) y 4000 (float).
- Las temporales se asignan desde 5000 y 6000.
- Las constantes son almacenadas a partir de 7000 (int) y 8000 (float), usando un mecanismo de tabla de literales que evita duplicados.

Durante la inicialización, todas las constantes identificadas durante la compilación son cargadas en memoria a través de una rutina específica, garantizando que sus valores estén disponibles antes de ejecutar cualquier cuádruplo.

Finalmente, el flujo de ejecución completo se inicia desde el método main, que construye el árbol sintáctico a partir del código fuente, invoca al visitante semántico para generar los cuádruplos y, posteriormente, inicializa la Máquina Virtual para ejecutar el programa. Este diseño modular y jerárquico permite separar claramente las fases de análisis, generación y ejecución, facilitando el mantenimiento y escalabilidad del compilador.

A continuación se muestran las tablas que definen el comportamiento de las operaciones aritméticas y relacionales tal como las interpreta la Máquina Virtual.

Aritmética:

Operando Izquierdo	**Operando Derecho**	**+**	**-**	*****	**/**
int	int	int	int	int	float
int	float	float	float	float	float
float	int	float	float	float	float
float	float	float	float	float	float
char	cualquier	error	error	error	error
cualquier	char	error	error	error	error

Relacional:

Operando Izquierdo	**Operando Derecho**	**=**	**!=**	**<**	**>**
int	int	bool	bool	bool	bool
int	float	error	error	error	error
float	int	error	error	error	error
float	float	bool	bool	bool	bool
char	char	bool	bool	bool	bool

char	distinto	error	error	error	error

9. Puntos Neurálgicos

A continuación se presenta una versión condensada con los **puntos más esenciales** del compilador/interprete de BabyDuck. Cada punto identifica la rutina o llamada crítica y describe brevemente su función:

1. **SemanticVisitor() (Constructor)**
   - Crea la entrada "global" en el FunctionDirectory y fija currentFunction = "global".
   - Inicializa QuadrupleGenerator y VirtualMemoryManager.

2. **visitPrograma(...)**
   - Genera una etiqueta mainLabel con qg.newLabel().
   - Emite qg.generateGoto(mainLabel) para saltar al inicio del bloque main.
     3. Marca la etiqueta mainLabel (qg.markLabel(mainLabel)) y luego visita body (cuerpo de main).

4. **visitVars(...) (contexto global o de función)**
   - Para cada VAR idList : type ; recorre los IDs,
     1. Llama a vmm.allocGlobal(type) (o allocLocal(type) si está dentro de una función) para obtener una dirección.
     2. Llama a dir.addVariableToCurrent(id, type, address) para registrar la variable en la tabla semántica del contexto actual.

visitFuncs(...)

- Para cada función void ID (params?) vars body ; hace:
  1. **(5) qg.generateFuncBegin(funcName) → emite LABEL funcName.**
  2. dir.addFunction(funcName, "void") y dir.setCurrentFunction(funcName).
  3. **(6) Si hay paramList, para cada parámetro:**
     - Reserva dirección con vmm.allocLocal(ptype).
     - Llama a dir.addVariableToCurrent(paramName, ptype, paddr) y a dir.getFunction(funcName).addParam(...).
  4. **(7) Llama a qg.generateFuncEnd() → emite ENDPROC y luego dir.setCurrentFunction("global").**

visitAssign(...)

- **(8) Verifica que la variable ID exista (primero en currentFunction, luego en "global").**
- **(9) Llama a qg.generateAssignmentAddress(destAddr) → emite (=, <origen>, -, <destAddr>).**

visitPrint(...)

- **(10) Para cada expresion:**
  1. Llama a visit(expresion) → deja un operando en la pila.
  2. Llama a qg.generatePrint() → emite (PRINT, <addr>, -, -).
- **(11) Para cada CTE_STRING:**
  1. Llama a qg.generatePrintString(text) → emite (PRINT_STR, "<texto>", -, -).

visitCondition(...) (if (expresion) body (else body)?)

- **(12) Obtiene cond = qg.peekOperand(), crea etiqueta Lf = qg.newLabel().**
- Llama a qg.generateGotoF(cond, Lf) → (GOTOF, <cond>, -, Lf).
- **(13) Si existe else:**
  1. Crea etiqueta Le = qg.newLabel().
  2. qg.generateGoto(Le) → (GOTO, -, -, Le).
  3. qg.markLabel(Lf), visita body “false” y finalmente qg.markLabel(Le).
- (13) Si no hay else: simplemente marca Lf con qg.markLabel(Lf).

visitCycle(...) (while (expresion) do body)

- **(14) Crea start = qg.newLabel(), marca con qg.markLabel(start).**
- (**15) Crea end = qg.newLabel() y emite qg.generateGotoF(cond, end).**
- **(16) Emite qg.generateGoto(start) y marca end con qg.markLabel(end).**

visitF_call(...) (llamada a función)

- **(17) Toma fname = ctx.ID().getText().**
- Emite qg.generateERA(fname) → (ERA, fname, -, -).
- **(18) Si hay argumentos (expresion), para cada uno:**
  1. Llama a qg.generateParam() → (PARAM, <addrArg>, -, -).
- **(19) Finalmente emite qg.generateGOSUB(fname) → (GOSUB, fname, -, -).**

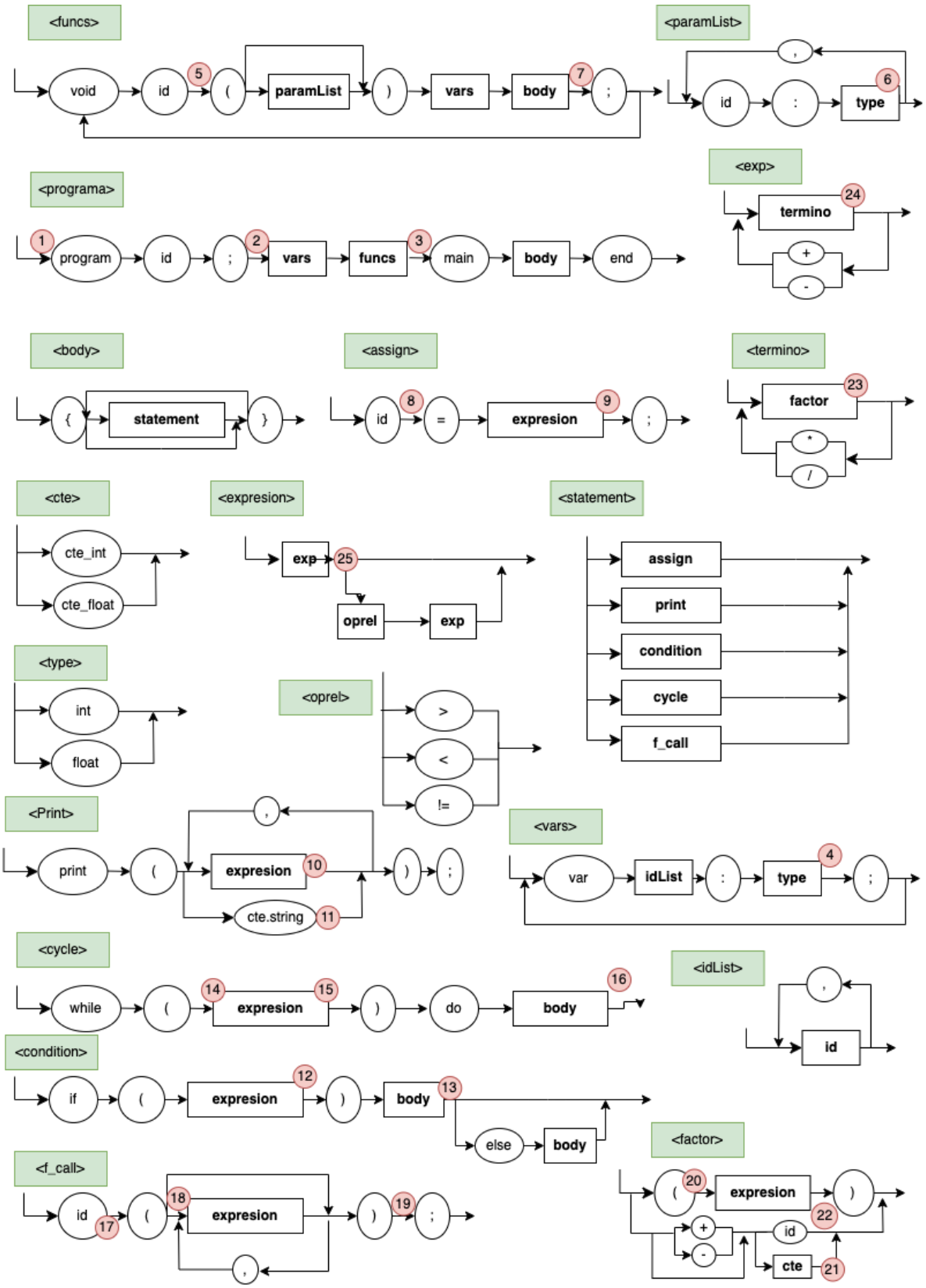
- visitFactor:
  1. **(20) Si es (expresion), reingresa recursivamente.**
  2. **(21) Si es cte:**
     - Determina type ("int" o "float").

        - Llama a vmm.allocConst(value, type) → obtiene una dirección constante única.
        - Llama a qg.pushOperandAddress(addr, type).
    3. **(22) Si es ID:**
        - Busca VariableInfo vi en tabla semántica (función o global).
        - Llama a qg.pushOperandAddress(vi.address, vi.type).

- visitTermino**:**
    1. **(23) Si (* | /) factor(i):**
        - qg.pushOperator("*" o "/").
        - Llama a qg.generateBinaryQuad().

- visitExp:
    1. **(24) Si (+ | -) termino(i):**
        - qg.pushOperator("+" o "-").
        - Llama a qg.generateBinaryQuad().

- visitExpresion:
    1. **(25) Si hay comparador (< | > | !=) exp(1):**
        - qg.pushOperator(oprel).
        - Llama a qg.generateBinaryQuad().


<funcs>
void
id
5
(
paramList
)
vars
body
7
;
<paramList>
,
id
:
type
6
<programa>
1
program
id
;
2
vars
funcs
3
main
body
end
<exp>
termino
24
+
-
<body>
{
statement
}
<assign>
id
8
=
expresion
9
;
<termino>
factor
23
*
/
<cte>
cte_int
cte_float
<expresion>
exp
25
oprel
exp
<statement>
assign
print
condition
cycle
f_call
<type>
int
float
<oprel>
>
<
!=
<Print>
print
(
,
expresion
10
cte.string
11
)
;
<vars>
var
idList
:
type
4
;
<cycle>
while
(
14
expresion
15
)
do
body
16
<idList>
,
id
<condition>
if
(
expresion
12
)
body
13
else
body
<factor>
(
20
expresion
)
+
-
id
22
cte
21
<f_call>
id
17
(
18
expresion
)
19
;
,

10. Conclusión

El compilador BabyDuck integra todas las etapas fundamentales del proceso de compilación: desde el análisis léxico y sintáctico hasta la ejecución con una máquina virtual. La gramática diseñada en ANTLR, junto con el uso de estructuras como FunctionDirectory y VirtualMemoryManager, permitió construir un sistema modular capaz de interpretar programas con funciones, ciclos, condiciones y operaciones aritméticas.

El proyecto consolidó conocimientos clave sobre el diseño de lenguajes y ejecución simbólica. Como trabajo futuro, se pudiera contemplar extender el lenguaje con arreglos, más validaciones, valores de retorno en funciones y un sistema más completo de manejo de errores en tiempo de ejecución.