



Tecnológico de Monterrey - Campus Monterrey

BabyDuck - entrega #1 (Léxico y Sintaxis)

Desarrollo de aplicaciones avanzadas de ciencias computacionales

Profa. Elda Guadalupe Quiroga González

Daniela Ramos García A01174259

06 de mayo del 2025

Indice

<i>Indice</i>	<i>2</i>
<i>1. Herramienta de generación de automática de compiladores.</i>	<i>3</i>
<i>2. Expresiones Regulares</i>	<i>3</i>
<i>3. Tokens reconocidos por el lenguaje.</i>	<i>3</i>
<i>4. Reglas gramaticales (CFG).</i>	<i>4</i>
<i>5. Declaración de syntax en BNF.</i>	<i>5</i>
<i>6. Test-Plan</i>	<i>6</i>
Casos de Prueba.	6
Screenshot de ejecución de pruebas.	7
<i>7. Estructuras de datos semánticas</i>	<i>7</i>
<i>8. Puntos neurálgicos en la gramática</i>	<i>8</i>
<i>9. Cubo semántico</i>	<i>8</i>
<i>10. Principales operaciones implementadas</i>	<i>9</i>
<i>11. Plan de pruebas semánticas</i>	<i>9</i>
<i>12. Implementación de pilas y fila para cuádruplos.</i>	<i>10</i>
<i>13. Algoritmo de traducción de cuádruplos.</i>	<i>10</i>
<i>14. Contenido de la Fila de cuádruplos que se genera</i>	<i>10</i>
Screenshot de test cases.	11

1. Herramienta de generación de automática de compiladores.

El compilador se desarrollará usando la herramienta de gocc, el cuál es un generador de compiladores ligero para Go que, a partir de un archivo BNF, produce de forma automática un *scanner* (DFA) y un *parser* (PDA LR-1). Se corre como una herramienta de línea de comandos (gocc <archivo>.bnf) sobre cualquier plataforma compatible con Go.

Gocc basa su *lexer* en autómatas finitos deterministas que reconocen expresiones regulares definidas en la sección léxica del BNF, e implementa un *parser* de tipo LR-1 (pushdown automaton) capaz de resolver automáticamente conflictos *shift/reduce* y *reduce/reduce* y el orden de declaración.

El formato de entrada exige un único archivo que combine la parte léxica y la parte sintáctica con posibilidad de incrustar *action expressions* en << >>. El BNF extendido permite incluir un encabezado con << import "paquete" >> para añadir código Go propio que se invoca en cada producción reconocida .

Gocc se distribuye bajo Apache License 2.0, liberando su uso y modificación en proyectos tanto libres como comerciales. Gocc ofrece un CLI con opciones para habilitar resolución automática de conflictos (-a), logging de depuración (-debug_lexer, -debug_parser), generar solo *parser* (-no_lexer), y personalizar directorio y paquete de salida (-o, -p) .

Para inyectar lógica propia, basta con insertar *action expressions* al final de cada alternativa de producción, devolviendo un (interface{}, error). El texto entre << y >> se evalúa en el momento del *reduce*, permitiendo construir AST o realizar traducciones directas sin escribir código fuera de la gramática . Gocc combina teoría (DFAs para el léxico, PDA LR-1 para la sintaxis) con una interfaz simple de BNF+Go, ofreciendo un flujo ágil para prototipar lenguajes y DSLs en el entorno Go.

2. Expresiones Regulares

id : [a-z][a-z-]*

cte_int : [0-9]+

cte_float : [0-9]+.[0-9]+

cte_string : '['^']

whitespace : [\t\r\n]+

comment : **(?:.|r|n)?**

3. Tokens reconocidos por el lenguaje.

	Nombre del Token	Lexema	Descripción
1	PROGRAM	program	Palabra reservada para iniciar un programa
2	ID	[a-z]([a-z\^-]*)	Identificador de variables o funciones
3	SEMICOLON	;	Fin de instrucción
4	MAIN	main	Función principal del programa
5	END	end	Palabra reservada para terminar bloques
6	LBRACE	{	Llave izquierda (inicio de bloque)
7	RBRACE	}	Llave derecha (fin de bloque)
8	ASSIGN	=	Operador de asignación
9	LT	<	Menor que (comparación)
10	GT	>	Mayor que (comparación)
11	NEQ	!=	Diferente que (comparación)
12	CTE_INT	[0-9]+	Constante entera
13	CTE_FLOAT	[0-9]+\.[0-9]{1,3}	Constante flotante
14	PLUS	+	Operador de suma
15	MINUS	-	Operador de resta
16	VOID	void	Tipo de retorno vacío
17	LPAREN	(Paréntesis izquierdo
18	COLON	:	Dos puntos, común en declaraciones
19	COMMA	,	Separador de elementos
20	RPAREN)	Paréntesis derecho
21	LBRACKET	[Corchete izquierdo (arreglos)
22	RBRACKET]	Corchete derecho (arreglos)
23	MULT	*	Operador de multiplicación
24	DIV	/	Operador de división
25	VAR	var	Palabra clave para declarar variables
26	PRINT	print	Instrucción para imprimir en pantalla
27	CTE_STRING	'[^\\n]*'	Constante de cadena
28	WHILE	while	Palabra clave para ciclo `while`
29	DO	do	Palabra clave que acompaña al `while`
30	IF	if	Palabra clave para condicional
31	ELSE	else	Alternativa al condicional `if`

4. Reglas gramaticales (CFG).

Programa → program id ; <Vars> <Funcs> main <Body> end		
Vars → var <VarList> : <Type> ; <VarsList>	Vars → \emptyset	Func → void id (<ParamList>) [<Vars> <Body>] ;
VarList → id , <VarList>	VarList → id	Body → { <StatementList> }
Type → int	Type → float	Assign → id = <Expression> ;
Funcs → <Func> <Funcs>	Funcs → \emptyset	Cycle → while (Expression) do <Body> ;
StatementList → <Statement> <StatementList>	StatementList → \emptyset	Condition → if (Expression) <Body> <c_else> <Body> ;
Statement → <Assign>	Statement → <Condition>	Condition → if (Expression) <Body> ;
Statement → <Cycle>	Statement → <F_Call>	Expression → <AddExpr> <RelExpr>
Statement → <Print>	ParamList → id : <Type>	RelOp → <
RelExpr → <RelOp> <AddExpr>	RelExpr → \emptyset	MulExpr → <MulExpr> / <Primary>
RelOp → >	RelOp → !=	MulExpr → <MulExpr> * <Primary>
AddExpr → <AddExpr> + <MulExpr>	AddExpr → <MulExpr>	MulExpr → <Primary>
AddExpr → <AddExpr> - <MulExpr>		Primary → (<Expression>)
Primary → cte_int	Primary → cte_float	F_Call → id (ArgList) ;
Primary → cte_string	Primary → id	F_Call → id () ;
ArgList → <Expression> , <ArgList>	ArgList → <Expression>	

5. Declaración de sintax en BNF.

Start : Programa ;	Statement StatementList ;
Programa	Statement
: "program" id ";" Vars Funcs "main" Body "end" ;	: Assign
Vars	Condition
: empty	Cycle
"var" VarList ":" Type ";" Vars ;	F_Call
VarList	Print ;
: id	Assign
id "," VarList ;	: id "=" Expression ";" ;
Type	Expression
: "int"	: AddExpr RelExpr ;
"float" ;	RelExpr
Funcs	: empty
: empty	RelOp AddExpr ;
Func Funcs ;	RelOp
Func	: "<"
: "void" id "(" ")" "{" Vars Body "}" ";"	">"
"void" id "(" ParamList ")" "{" Vars Body "}" ";"	"!=" ;
;	AddExpr
ParamList	: AddExpr "+" MulExpr
: id ":" Type	AddExpr "-" MulExpr
id ":" Type "," ParamList ;	MulExpr ;
Body	MulExpr
: "{" StatementList "}" ;	: MulExpr "*" Primary
StatementList	MulExpr "/" Primary
: empty	Primary ;

Primary	: Expression
: "(" Expression ")"	Expression "," ArgList ;
id	Cycle
cte_int	: "while" "(" Expression ")" "do" Body ";" ;
cte_float	Condition
cte_string ;	: "if" "(" Expression ")" Body "else" Body ";"
Print	"if" "(" Expression ")" Body ";" ;
: "print" "(" ")" ";"	F_Call
"print" "(" ArgList ")" ";" ;	: id "(" ")" ";"
ArgList	id "(" ArgList ")" ";"

6. Test-Plan

Alcance: Incluye pruebas de unidad al parser y al lexer, declaraciones, asignaciones, bloques, funciones, comentarios y errores sintácticos. No incluye generación de AST ni acciones semánticas.

Criterios de Aprobación

- Éxito: $\geq 99\%$ de los TC pasan sin errores; los errores esperados (TC4, TC5) se detectan correctamente.
- Rechazo: fallos inesperados en TC válidos o aceptación de programas sintácticamente incorrectos.

Riesgos y Mitigación

- Cambio en BNF: obliga a reescribir casos \rightarrow mantener versión congelada durante pruebas.
- Conflictos LR(1): usar gocc -a para resolver automáticamente.
- Cobertura insuficiente: planear revisión de casos adicionales tras primera pasada.

Casos de Prueba.

ID	Descripción	Entrada	Resultado Esperado
TC1	Programa vacío	program p; main { } end	Sin errores de parseo
TC2	Asignación y declaración de variables	program p; var x: int; main { x = 5; } end	Sin errores de parseo
TC3	Constante float con asignación	** sample ** program p; var x: float; main { x = 3.14; } end	Sin errores de parseo
TC4	Estatuto if	program p; var x: int; main { if (x < 10) { x = x + 1; }; } end	Sin errores de parseo

TC5	Manejo de comentarios	<code>** Testing comments ** program p; var x: int; main { x = 5; } end</code>	Sin errores de parseo
TC6	Estatuto Print	<code>program p; var x: int; main { x = 5; print(x); } end</code>	Sin errores de parseo
TC7	Whitespace y newlines	<code>*** Testing new line whitespace ** program p;\n\tmain { }\n\tend"</code>	Sin errores de parseo
TC8	Ciclo while en varias lineas	<code>"program p; var x: int; main\n\t{ while (x < 10) do { x = x + 1; }; }\n\tend"</code>	Sin errores de parseo
TC9	Falta punto y coma y id	<code>program p main { } end</code>	Error de parseo (falta ; tras p)
TC10	Falta el token end	<code>program p; main { }</code>	Error de parseo (falta palabra end)
TC11	Punto y coma después de asignación	<code>program p; var x: int; main { x = 5 } end</code>	Error de parseo (falta ; tras 5)
TC12	Texto extra después de end	<code>program p; var x: int; main { x = 5; } end extra</code>	Error de parseo (texto inesperado)
TC13	Declaración de función	<code>program p; var x: int; void f(a: int) [{ b = a + 2; }]; main { } end</code>	Sin errores de parseo

Screenshot de ejecución de pruebas.

```

danielaramosgarcia@Danielas-MacBook-Pro-2 babyduck % go test -v
=== RUN   TestParse
babyduck_test.go:47: === Test #1
babyduck_test.go:47: === Test #2
babyduck_test.go:47: === Test #3
babyduck_test.go:47: === Test #4
babyduck_test.go:47: === Test #5
babyduck_test.go:47: === Test #6
babyduck_test.go:47: === Test #7
babyduck_test.go:47: === Test #8
babyduck_test.go:47: === Test #9
babyduck_test.go:47: === Test #10
babyduck_test.go:47: === Test #11
babyduck_test.go:47: === Test #12
babyduck_test.go:47: === Test #13
--- PASS: TestParse (0.00s)
PASS
ok      babyduck      0.250s
danielaramosgarcia@Danielas-MacBook-Pro-2 babyduck %

```

7. Estructuras de datos semánticas

El actor principal de la semántica de BabyDuck está el objeto Context, que actúa como contenedor global de toda la información necesaria durante el análisis: la tabla de variables globales (GlobalVars), el directorio de funciones (FuncDir) y una referencia puntual a la función en la que nos encontremos (currentFunc). La tabla de variables está implementada como VarTable, un mapa de nombres a entradas de variable junto con un enlace a su contexto padre (en el caso global, el padre es nil). Esta estructura facilita la resolución de identificadores, saltando al padre solo cuando no se encuentra la variable en el ámbito local. Por su parte, cada FuncEntry almacena la firma de una función: nombre, tipo de retorno, lista de parámetros y su propia VarTable para variables locales. Todas las funciones registradas viven en FuncDir, que mantiene un mapa simple de nombre a *FuncEntry y asegura que no haya duplicados. La interacción de estas estructuras permiten lo siguiente:

- Cuando se declara una variable global, Context invoca RegisterGlobalVars, que usa VarTable.Add para asegurarse de que no se re-declare.
- Al registrar una función en la fase sintáctica, Context crea un FuncEntry, lo inserta en FuncDir y prepara la tabla de variables locales enlazada a la tabla global.
- Durante la resolución de tipos en expresiones o la validación de asignaciones, Context usa currentFunc para decidir si busca en la tabla local o, en su defecto, en la global.

8. Puntos neurálgicos en la gramática

Para coordinar la semántica con la generación automática de Gocc, añadimos un no-terminal auxiliar Reset que se reduce antes de procesar el programa completo. Así garantizamos que, en cada llamada a Parse, el Context comience limpio sin residuos de ejecuciones anteriores. La regla inicial Start invoca primero Reset y luego Programa, y al final retorna el Context poblado. A partir de ahí, cada producción relevante en el BNF dispara una acción semántica al final de su alternativa. Por ejemplo, la producción Vars llama a ctx.RegisterGlobalVars usando los atributos generados (X[...]) para añadir todas las variables globales antes de continuar; la producción Func invoca ctx.RegisterFunction con el nombre, tipo de retorno y parámetros de la función recién declarada; la regla Assign valida existencia y compatibilidad de tipos con ctx.ValidateAssign; y la regla Primary obtiene el tipo de un literal o variable mediante ReturnExpression o ctx.ResolveVarType. Estas llamadas quedan encapsuladas en llamadas a funciones auxiliares, lo que mantiene el BNF claro y evita duplicar return en el código generado.

9. Cubo semántico

El cubo semántico reside en `data_structures/semantic_cube.go` y está representado por dos mapas: uno para operadores binarios (`binaryCube`) y otro para unarios (`unaryCube`). Cada entrada de `binaryCube[op][left][right]` devuelve el tipo resultante o produce un error si la combinación no existe. Análogamente, `unaryCube[op][operand]` define operaciones como la negación lógica o aritmética. Sobre estos datos se construyen las funciones `ResultBinary` y `ResultUnary`, que son invocadas por la fase de análisis semántico (fuera del BNF) cuando se recorre el AST. Gracias al cubo, podemos garantizar en tiempo de compilación que expresiones como `x + y` o `-z` cumplen las reglas de tipos de BabyDuck, generando mensajes de error claros cuando no lo hacen.

10. Principales operaciones implementadas

- `ctx.Reset()`: reinicia estado antes de cada parse.
- `ctx.RegisterGlobalVars`, `MakeVarList`, `ConcatVarList`: manejo de variables globales.
- `ctx.RegisterFunction`, `MakeParamList`, `ConcatParamList`: registro de funciones y parámetros.
- `ctx.ValidateAssign`: validación de asignaciones.
- `ctx.ResolveVarType`, `ReturnExpression`: resolución de tipos en expresiones y literales.

11. Plan de pruebas semánticas

<i>ID</i>	<i>Descripción</i>	<i>Entrada</i>	<i>Resultado Esperado</i>
TC14	Redeclaración de variable global	<code>program p; var x: int; var x: float; main { } end</code>	Error semántico: "variable x ya declarada"
TC15	Uso de variable no declarada	<code>program p; main { x = 5; } end</code>	Error semántico: "variable x no declarada"
TC16	Asignación con tipos incompatibles	<code>program p; var x: int; main { x = 3.14; } end</code>	Error semántico: "tipos incompatibles en asignación"
TC17	Llamada a función no declarada	<code>program p; main { f(); } end</code>	Error semántico: "función f no declarada"
TC18	Redeclaración de función	<code>program p; void f(){} void f(){} main { } end</code>	Error semántico: "función f ya declarada"
TC19	Declaración y uso de parámetros en firma de función	<code>program p; void f(a:int,b:float){ } main { } end</code>	Sin errores
TC20	Declaración y uso de variable local	<code>program p; void f(){ var y:int; y=1; } main { } end</code>	Sin errores

TC21	Acceso a variable global dentro de función	program p; var x:int; void f(){ x=2; } main { } end	Sin errores
------	--	---	-------------

12. Implementación de pilas y fila para cuádruplos.

La implementación de las pilas y la fila de cuádruplos enriquece el contexto de compilación con estructuras LIFO y FIFO especializadas. Por un lado, definimos tres pilas en el Context: una de operadores (OperatorStack []string), otra de operandos (OperandStack []string) y una tercera de tipos (TypeStack []Tipo). Cada una expone métodos sencillos de apilar y desapilar que internamente gestionan un slice de Go: PushOperator/PopOperator, PushOperand/PopOperand y PushType/PopType. De este modo garantizamos la preservación del orden de aparición de símbolos durante el análisis sintáctico. Por otro lado, para almacenar las instrucciones intermedias generadas, creamos la estructura Quadruple{Op, Arg1, Arg2, Result} y la cola QuadQueue con su método Enqueue. Esta cola se integra en el Context como Quads QuadQueue, permitiendo acumular cada cuádruplo en el orden exacto en que se deben ejecutar en la etapa posterior de generación de código o interpretación.

13. Algoritmo de traducción de cuádruplos

El algoritmo de traducción a cuádruplos aprovecha esas pilas y la fila para transformar expresiones y sentencias en instrucciones intermedias. En las expresiones aritméticas, cada vez que el parser reconoce un literal o un identificador, ejecuta un helper HandleOperand(lexema, tipo) que coloca el operando en la pila de operandos y su tipo en la pila de tipos. Cuando reduce un operador binario (por ejemplo +, -, *, /), invoca HandleBinary(op), que desapila los dos operandos y tipos correspondientes, consulta el cubo semántico para validar y obtener el tipo resultante, genera un nuevo temporal tN, lo vuelve a apilar y finalmente encola un cuádruplo (op, left, right, tN). Para las expresiones relacionales (<, >, !=), el proceso es análogo: al reconocer el operador el parser llama a HandleBinary(op) y obtiene un booleano como resultado, encolando (op, left, right, tN). En los estatutos lineales del lenguaje, agregamos acciones semánticas que toman el último temporal o identificador de la pila de operandos y generan cuádruplos específicos: (=, rhs, , lhs) para asignaciones, (PRINT, arg, ,) para llamadas a print, y bloques de salto condicional GOTOIF o incondicional GOTO con etiquetas generadas secuencialmente. De esta forma, todo fragmento de programa BabyDuck queda traducido a una secuencia plana de cuádruplos, lista para la siguiente fase del compilador.

14. Contenido de la Fila de cuádruplos que se genera

ID	Descripción	Entrada	Resultado Esperado
1	Simple suma de enteros	2 + 3	OperandStack:['t1'], TypeStack:[Int], Cuádruplos:[(+,2,3,t1)]
2	Simple resta de enteros	05-Feb	OperandStack:['t1'], TypeStack:[Int], Cuádruplos:[(-,5,2,t1)]
3	Simple multiplicación de enteros	3 * 4	OperandStack:['t1'], TypeStack:[Int], Cuádruplos:[(*,3,4,t1)]
4	División de enteros con promoción a float	04-Feb	OperandStack:['t1'], TypeStack:[Float], Cuádruplos:[(/,4,2,t1)]
5	Relacional menor que	5 < 6	OperandStack:['t1'], TypeStack:[Bool], Cuádruplos:[(<,5,6,t1)]
6	Relacional mayor que	7 > 3	OperandStack:['t1'], TypeStack:[Bool], Cuádruplos:[(>,7,3,t1)]
7	Relacional diferente que	4 != 4	OperandStack:['t1'], TypeStack:[Bool], Cuádruplos:[(!=,4,4,t1)]
8	Precedencia mixta	3 * 4 + 2	OperandStack:['t2'], TypeStack:[Int], Cuádruplos:[(*,3,4,t1),(+,t1,2,t2)]
9	Suma encadenada	1 + 2 + 3	OperandStack:['t2'], TypeStack:[Int], Cuádruplos:[(+,1,2,t1),(+,t1,3,t2)]
10	Mismatch de tipos (bool + int)	true + 1	Error semántico esperado por mismatch Bool+Int
11	Expresión compleja	(1 + 2) * (3 - 4) / 2	OperandStack:['t4'], TypeStack:[Float], Cuádruplos:[(+,1,2,t1),(-,3,4,t2),(*,t1,t2,t3),(/,t3,2,t4)]

Screenshot de test cases.

```
7dev/1d/13:18: command not found: compiler
danielaramosgarcia@Danielas-MacBook-Pro-2 babyduck % go test ./data_structures -v
```

```
=== RUN   TestVarTable
--- PASS: TestVarTable (0.00s)
=== RUN   TestFuncDir
--- PASS: TestFuncDir (0.00s)
=== RUN   TestContextHelpers
--- PASS: TestContextHelpers (0.00s)
=== RUN   TestRegisterAndEnterFunction
--- PASS: TestRegisterAndEnterFunction (0.00s)
=== RUN   TestQuadrupleGeneration
stacks_op_test.go:16: === Después de 2+3 ===
stacks_op_test.go:17: OperandStack: [t1]
stacks_op_test.go:18: TypeStack: [0]
stacks_op_test.go:19: Cuádruplos:
stacks_op_test.go:21: 0: (+, 2, 3, t1)
stacks_op_test.go:34: === Después de t1*4 ===
stacks_op_test.go:35: OperandStack: [t1 t2]
stacks_op_test.go:36: TypeStack: [0 0]
stacks_op_test.go:37: Cuádruplos:
stacks_op_test.go:39: 0: (+, 2, 3, t1)
stacks_op_test.go:39: 1: (*, t1, 4, t2)
--- PASS: TestQuadrupleGeneration (0.00s)
=== RUN   TestRelationalQuadruple
stacks_op_test.go:54: === Después de 5<6 ===
stacks_op_test.go:55: OperandStack: [t1]
stacks_op_test.go:56: TypeStack: [2]
stacks_op_test.go:57: Cuádruplos:
stacks_op_test.go:59: 0: (<, 5, 6, t1)
--- PASS: TestRelationalQuadruple (0.00s)
=== RUN   TestMixedPrecedence
stacks_op_test.go:83: OperandStack: [t1 t2]
stacks_op_test.go:84: TypeStack: [0 0]
stacks_op_test.go:85: Cuádruplos:
stacks_op_test.go:87: 0: (*, 3, 4, t1)
stacks_op_test.go:87: 1: (+, t1, 2, t2)
--- PASS: TestMixedPrecedence (0.00s)
=== RUN   TestChainAddition
stacks_op_test.go:108: OperandStack: [t2]
stacks_op_test.go:109: TypeStack: [0]
stacks_op_test.go:110: Cuádruplos:
stacks_op_test.go:112: 0: (+, 1, 2, t1)
stacks_op_test.go:112: 1: (+, t1, 3, t2)
--- PASS: TestChainAddition (0.00s)
```

```
stacks_op_test.go:108: OperandStack: [t2]
stacks_op_test.go:109: TypeStack: [0]
stacks_op_test.go:110: Cuádruplos:
stacks_op_test.go:112: 0: (+, 1, 2, t1)
stacks_op_test.go:112: 1: (+, t1, 3, t2)
--- PASS: TestChainAddition (0.00s)
=== RUN   TestIntFloatPromotion
--- PASS: TestIntFloatPromotion (0.00s)
=== RUN   TestTypeMismatch
stacks_op_test.go:145: Error esperado: operador + no soportado para tipo izquierdo 2
--- PASS: TestTypeMismatch (0.00s)
=== RUN   TestResultBinary_Valid
--- PASS: TestResultBinary_Valid (0.00s)
=== RUN   TestResultBinary_Invalid
--- PASS: TestResultBinary_Invalid (0.00s)
=== RUN   TestResultUnary_Valid
--- PASS: TestResultUnary_Valid (0.00s)
=== RUN   TestResultUnary_Invalid
--- PASS: TestResultUnary_Invalid (0.00s)
=== RUN   TestComplexExpression
stacks_op_test.go:291: OperandStack final: [t1 t2 t3 t4]
stacks_op_test.go:292: TypeStack final: [0 0 0 1]
stacks_op_test.go:293: Cuádruplos generados:
stacks_op_test.go:295: 0: (+, 1, 2, t1)
stacks_op_test.go:295: 1: (-, 3, 4, t2)
stacks_op_test.go:295: 2: (*, t1, t2, t3)
stacks_op_test.go:295: 3: (/, t3, 2, t4)
--- PASS: TestComplexExpression (0.00s)
PASS
ok      babyduck/data_structures      0.319s
```

Liga de Github de la entrega:

https://github.com/danielaramosgarcia/Aplicaciones_avanzadas_TTC3002B/tree/main/Compiladores/babyduck