Tecnológico de Monterrey - Campus Monterrey

BabyDuck - entrega #6 Entrega Final

Desarrollo de aplicaciones avanzadas de ciencias computacionales

Profa. Elda Guadalupe Quiroga González

Daniela Ramos García A01174259

2 de junio del 2025

Índice

Consideraciones de léxico y sintaxis.

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.

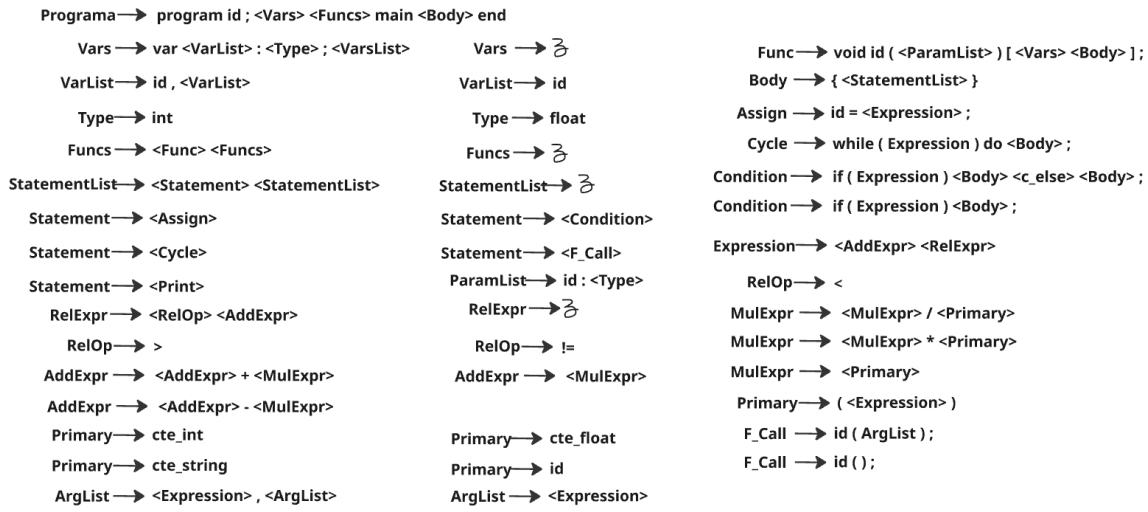
Expresiones Regulares

id : [a-z][a-z-]*
cte_int : [0-9]+
cte_float : [0-9]+.[0-9]+
cte_string : '[^']'
whitespace : [\t\r\n]+
comment : **(?:.|\r|\n)?**

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`

Reglas gramaticales (CFG).



Declaración de sintax en BNF.

La herramienta de generación del lexer y paser (gocc) toma como entrada un archivo de tipo BNF donde se definen las reglas de la gramática para posteriormente generar automáticamente el lexer y parser con el comando de gocc <ruta del archivo>. Dentro de este archivo se definen los tokens junto con las expresiones regulares para su construcción. Las reglas que se agregaron para el proyecto se pueden ver en la sección anterior, sin embargo, se realizaron ligeras modificaciones de partición para poder implementar los puntos neurálgicos debido a que la sintaxis de gocc no permite implementar puntos neurálgicos en medio de las reglas del bnf, si no, hasta su fin.

Liga archivo bnf: Archivo bnf.

Test-Plan

Alcance: En la primera sección se incluyen pruebas de unidad al parser y al lexer, declaraciones, asignaciones, bloques, funciones, comentarios y errores sintácticos.

Criterios de Aprobación

- Éxito: ≥ 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 → mantener versión congelada durante pruebas.
- Conflictos LR(1): usar gocc -a para resolver automáticamente.
- Cobertura insuficiente: planear revisión de casos adicionales tras primera pasada.

Análisis semántico.

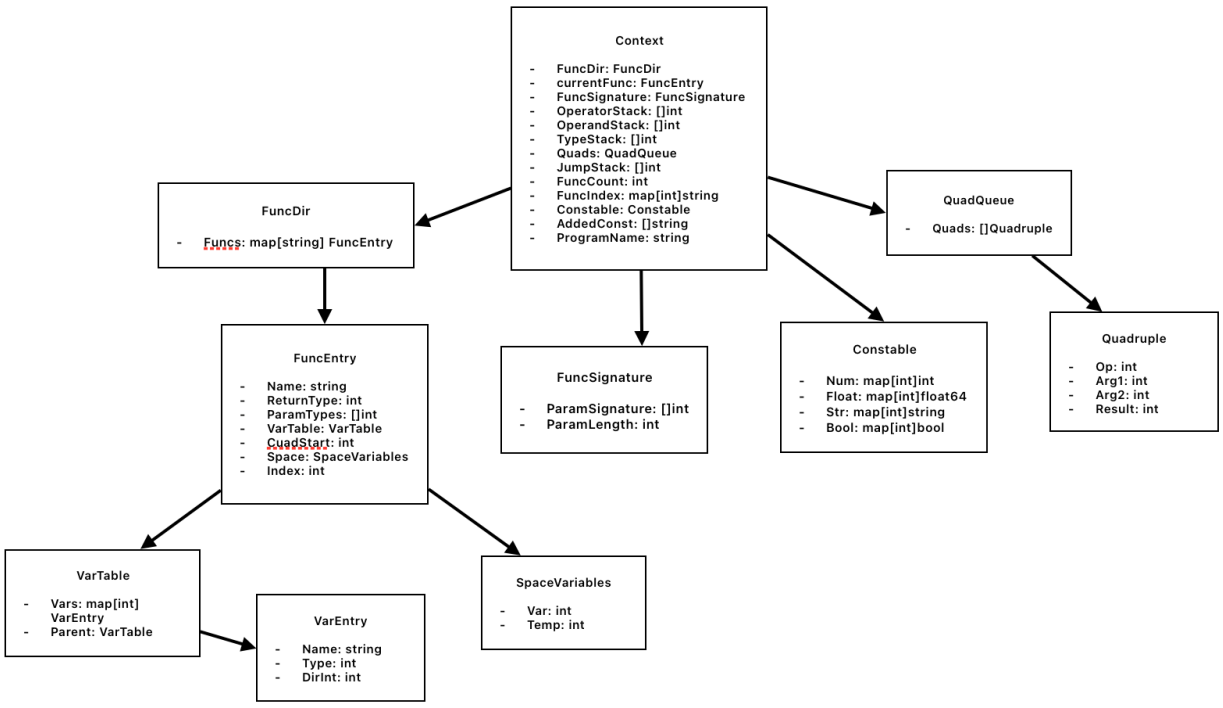
Estructuras de datos.

El actor principal de la semántica de BabyDuck está el objeto ctx de tipo Context, que actúa como contenedor global de toda la información necesaria para el analisis semantico y la generación de código intermedio y el cuál se instancia al inicio de la compilación. A continuación una descripción breve de las propiedades del contexto y sus tipos de datos.

- Directorio de funciones: Representa al directorio de funciones el cual se accesa mediante un mapa con llave de tipo string del nombre de la funcion y valor de entrada de función.
    - Entrada de funciones: Representa una función y contiene propiedades para almacenar el nombre, el tipo de dato de retorno, los tipos de datos de los parametros, la tabla de variables, el cuádruplo donde inicia la función, el indice de la función y el espacio requerido para la funcion representado en numero de variables y número de temporales.
        - Tabla de variables: Almacena las variables locales de cada funcion con un mapa con llave de tipo int y valor de tipo Entrada de variable y apunta a la table padre para almacenar las variables globales.
            - Entrada de variable: Almacenar el nombre, el tipo y el la dirección númerica de cada variable.
- Función actual: Propiedad que se actualiza continuamente durante el análisis semantico para acceder a la tabla de variables perteneciente a la función actual y poder realizar operaciones como agregar y buscar variables.
- Firma de la función: Propiedad que se actualiza continuamente cuando se llama a una función para comprobar que el tipo y cantidad de argumentos que se mandan a la función sea igual que los parámetros que se definieron en la declaracion de la función.
- Stack de operadores: Stack de operadores que se utiliza para la generación de cuadruplos.
- Stack de operandos: Stack de operandos que se utiliza para la generación de cuadruplos.
- Stack de tipos: Stack de tipo de dato que se utiliza para la generación de cuadruplos y validaciones con el cubo semántico.

- Vector de cuádruplos: Vector de tipo Cuadrupo que almacena los cuádruplos que se van generando en el análisis semántico.
    - Cuádruplo: Almacena el operador, argumento izquierdo, argumento derecho y resultado de cada cuádruplo.
- Stack de saltos: Stack de direcciones que se utilizarán en la generación de cuádruplos para indicar saltos entre cuádruplos en estatutos ciclicos y condicionales para el apuntador de la máquina virtual.
- Contador de Funciones: Lleva un contador de las funciones declaradas para asignar un indice en la creación de cada función.
- Index de funciones: Mapa con llave del indice de las funciones y valor de el string con el nombre de la función. Necesario por que los cuádruplos almacenan el indice de la función pero las funciones se acceden medinate strings.
- Tabla de constantes: Almacena la informacion de el valor de las constantes y sus respectivas direcciones de memoria.
- Constantes agregadas: Array que almacena el string de las constantes agregadas para veirificar si la constante ha sido agregada anteriormnete o no.

A continuación una representación gráfica de la definición de los tipos del contexto.



Cubo semántico

El cubo semántico 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, donde op, left y right son enteros que representan al operador y los tipos de los operandos.

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.
Implementación.

```
var binaryCube = map[int]map[int]map[int]int{
	10: {
		0: {0: 0, 1: 1},
		1: {0: 1, 1: 1},
	},
	20: {
		0: {0: 0, 1: 1},
		1: {0: 1, 1: 1},
	},
	30: {
		0: {0: 0, 1: 1},
		1: {0: 1, 1: 1},
	},
	40: {
		0: {0: 1, 1: 1},
		1: {0: 1, 1: 1},
	},
	50: {
		0: {0: 2, 1: 2},
		1: {0: 2, 1: 2},
	},
	60: {
		0: {0: 2, 1: 2},
		1: {0: 2, 1: 2},
	},
	70: {
		0: {0: 2, 1: 2},
		1: {0: 2, 1: 2},
	},
}
```

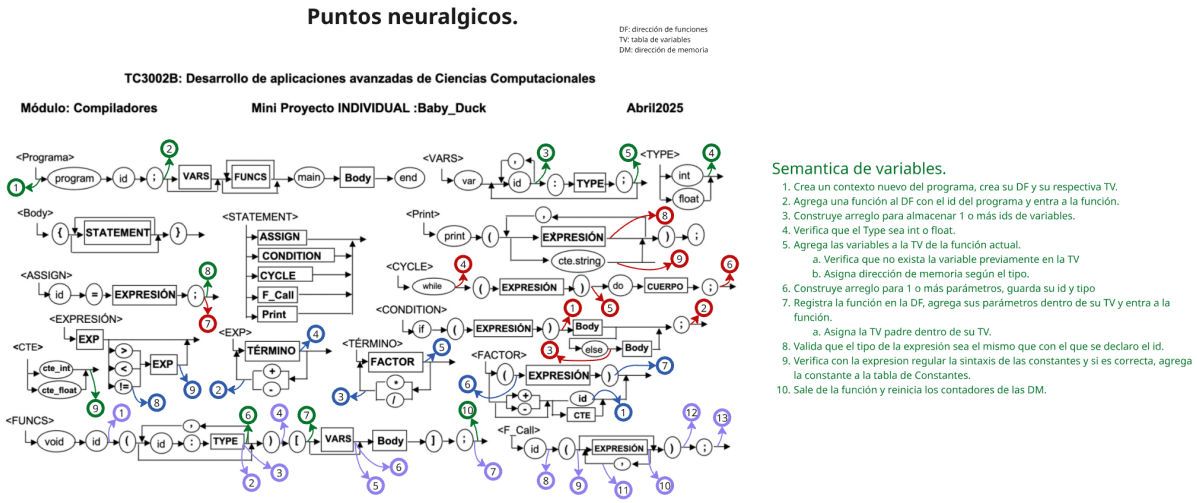
Liga de archivo de cubo semántico: Cubo semántico.

Puntos neurálgicos en la gramática

Para coordinar la semántica con la generación automática de Gocc, se añadió 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.

A continuación, los puntos neurálgicos en la gramática.



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.
- ctx.AddFuncion: Registra una función en el directorio global y crea la entrada con su propia tabla local para variables.

Generación de código intermedio.

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 ejecución de la máquina virtual.

Adicionalmente, se utilizan funciones auxiliares para la traducción de operadores y tipos de datos que se llaman desde el análisis léxico cuando se reconoce un token de operador o un token para los tipos admitidos en el lenguaje. Estas funciones reciben tokens y regresan la representación númerica respectiva para su posterior análisis semántico.

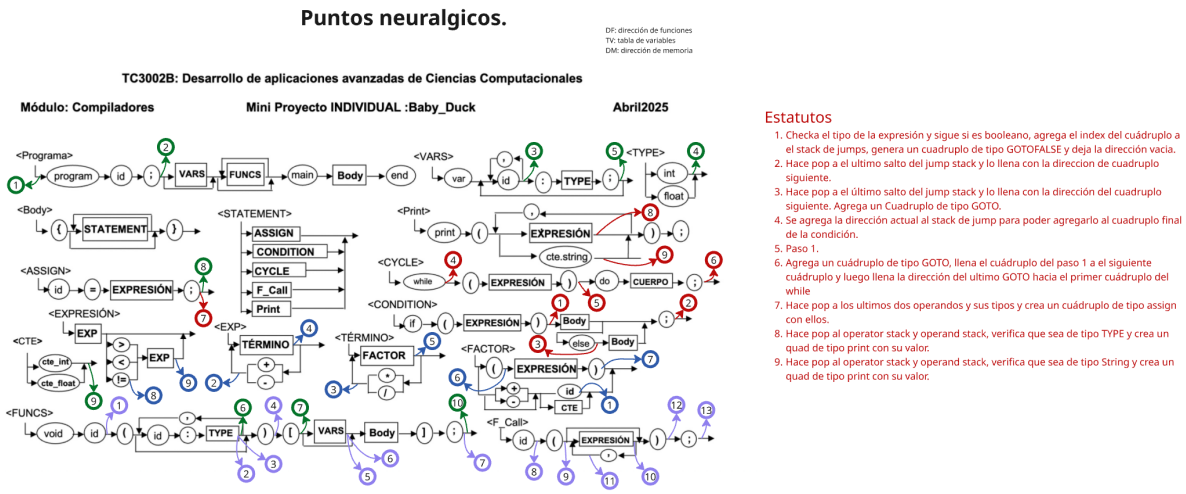
```
func (ctx *Context) TranslateOp(op string) (interface{},
error) {
	switch op {
	case "+":
		ctx.PushOperator(ADD)
		return nil, nil
	case "-":
		ctx.PushOperator(SUB)
		return nil, nil
	case "*":
		ctx.PushOperator(MUL)
		return nil, nil
	case "/":
		ctx.PushOperator(DIV)
		return nil, nil
	case "<":
		ctx.PushOperator(LT)
		return nil, nil
	case ">":
		ctx.PushOperator(GT)
		return nil, nil
	case "!=":
		ctx.PushOperator(NEQ)
		return nil, nil
	case "(":
		ctx.PushOperator(RPAR) // Asumimos
que "(" es un operador especial
		return nil, nil
	case ")":
		return ctx.HandleRightParen()
		// return nil, nil
	default:
		return nil, fmt.Errorf("operador
desconocido: %s", op)
	}
}
```

```
func TranslateType(typ string) Tipo {
	switch typ {
	case "int":
		return 0
	case "float":
		return 1
	case "bool":
		return 2
	default:
		return -1
	}
}
```

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, lo vuelve a apilar y finalmente encola un cuádruplo (op, left, right, temp). 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 GOTOF 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 ejecución y saltos de IP de la máquina virtual.

Puntos neurálgicos para código de Expresiones y Estatutos lineales:



Traduccion de variables a sus direcciones virtuales.

Cada vez que se declara una variable (global o local), el compilador le asigna automáticamente una “dirección virtual” en el espacio de memoria simulado, usando simples contadores que avanzan según el tipo. Por ejemplo, los enteros globales empiezan en 0 y se van incrementando

hasta justo antes del límite para floats globales; los floats globales ocupan el bloque siguiente, y lo mismo sucede para variables locales (enteros desde su base hasta el comienzo de floats locales, luego floats locales). De esta forma, tan pronto como el parser añade la entrada a la tabla, el contador correspondiente (por tipo y ámbito) avanza, reservando esa posición para la variable. Las variables temporales se manejan con contadores exclusivos ubicados en un rango aparte, y las constantes literales, a su vez, van en otro bloque distinto (enteros literales primero, luego floats, luego strings). Si alguno de estos contadores se sale de su subrango, se lanza un error de "overflow de direcciones". Con lo anterior, cada variable o temporal queda indexado por un número único que se usará luego al generar cuádruplos o direccionar lecturas/escrituras en el contexto.

Seudocódigo para ilustración.

```
func (vt *VarTable) Add(name string, typ int) error {
        for _, entry := range variables {
                if nombre en nombre de varialbles,
lanzar error.
        }
        var dir int
        if no tiene tabla padre (es la tabla global) {
                if typ == 0 {
                        if sig direccion >= base del
siguiente segmento de memoria {
                                return
fmt.Errorf("overflow de direcciones globales int")
                        }
                        dir = siguiente direccion
                        nextGlobalIntAddr++
                } else {
                        …repetir logica para tipo
float.
                }
        } else es una tabla local {
                …repetir logica para tabla local
        }
        vt.Vars[dir] = &VarEntry{Name: name, Type:
typ, DirInt: dir}
        return dir
}
```

Funciones.

Cuádruplos para la declaración e invocación de las funciones.

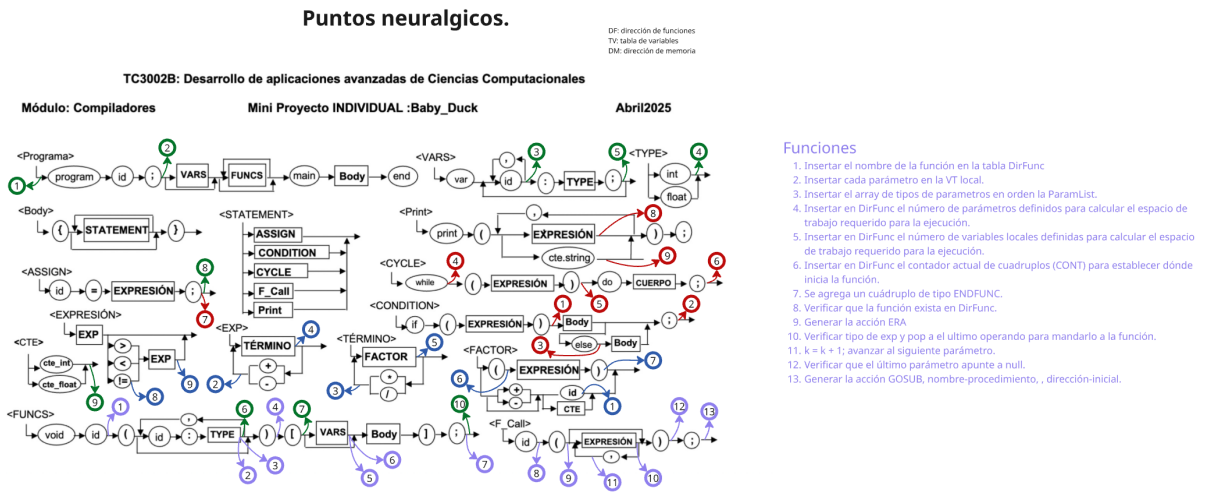
El archivo de generación de cuádruplos quadruples.go hace uso de las estructuras de datos de operadores, operandos y tipos junto con una cola QuadQueue del ctx donde se almacenan los cuádruplos intermedios. Cada vez que el analizador sintáctico reconoce un operando (literal, variable o temporal), llama a HandleOperand, que apila la dirección y el tipo en sus respectivas pilas. Cuando detecta un operador, invoca PushOperator para colocarlo en la pila de operadores. Al encontrar un operador de menor precedencia o un paréntesis de cierre, se desapilan los elementos necesarios y se llama a GenerateQuad o a HandleBinary, que extraen dos operandos y sus tipos, consultan el cubo semántico (ResultBinary) para validar la operación y obtienen el tipo resultante. A continuación, piden un temporal con RegisterTemp, apilan ese nuevo temporal

(junto con su tipo) para futuras operaciones y encolan un cuádruplo con la forma (Op, Arg1, Arg2, Result) en la cola de cuádruplos.

Para las estructuras de control (if, else, while), se utiliza una pila de saltos (JumpStack). Cuando se evalúa la condición de un if, MakeGFQuad genera un cuádruplo GOTOFALSE con destino pendiente y guarda su índice en JumpStack. Al llegar al else o al final del bloque, se llama a FillJump o a ElseJumpIf, que rellenan el campo Result del cuádruplo condicional con la instrucción correcta de salto. En los bucles, CycleJump marca el inicio del ciclo en JumpStack, y WhileJump genera un GOTO incondicional que regresa a ese inicio. De este modo, los saltos condicionales e incondicionales se completan una vez que conoce la dirección de destino.

Las llamadas a funciones se traducen primero con MakeEraQuad, que encola un cuádruplo ERA para preparar el entorno de activación, seguido de tantos MakeParamQuad como parámetros tenga la función, cada uno validando el tipo contra la firma (FuncSignature) y encolando un cuádruplo PARAM. Por último, MakeGOSUBQuad encola un GOSUB que contiene el índice de la función y el punto de inicio de sus cuádruplos. Al finalizar una función, MakeEndFQuad encola un ENDF, y cuando termina todo el programa, MakeEndQuad encola un END. De este modo, al concluir el análisis, la cola de cuádruplos (ctx.Quads.Quads) contiene la secuencia completa de instrucciones intermedias que la máquina virtual debe interpretar.

A continuación, los puntos neurálgicos para el manejo de funciones.



Máquina Virtual.

Mapa de memoria de ejecución.

La máquina virtual usa las siguientes propiedades para almacenar su memoria de ejecución y tener un correcto funcionamiento. Ctx es un puntero al contexto de compilación que agrupa información semántica que fue generada por los puntos neurálgicos de las secciones antermente mencionadas. La pila memStack es un slice de mapas que representa los marcos de memoria, el primer elemento es el mapa global de variables y constantes, y cada vez que se invoca una función se apila un nuevo mapa para sus variables locales. RetStack almacena los índices de instrucción a los que se debe regresar una vez que termine una llamada. El campo Quads contiene la lista completa de cuádruplos que la VM recorrerá, mientras que IP señala el índice actual en ese slice de cuádruplos. Para cada función, ParamAddrs asocia su nombre a un slice ordenado de direcciones que corresponden a sus parámetros. Finalmente, PrintOutput strings.Builder acumula las cadenas que producen los cuádruplos PRINT, de forma que al concluir la ejecución la VM pueda devolver todo el texto generado.

```
type Machine struct {
	Ctx	*data_structures.Context
	memStack	[]map[int]interface{}
	retStack	[]int
	Quads	[]data_structures.Quadruple
	IP	int
	ParamAddrs	map[string][]int // <-- nuevo
	PrintOutput strings.Builder
}
```

Memoria de ejecución.

Al inicializarse con NewMachine, se crea el mapa global (memStack[0]) y se llenan todas las variables globales y constantes (inicializándolas a cero o a su valor literal). Cada vez que se ejecuta un cuádruplo ERA para invocar una función, la VM construye un nuevo mapa vacío con las direcciones de las variables locales de esa función y lo apila arriba. De esta forma, los marcos en la pila representan entornos anidados

Con lo anterior, el tope de memStack es el marco activo, mientras que el fondo es siempre la memoria global. Para poder acceder y asignar valores dento de los marcos de memoria, se implementaron dos funciones principales.

- readMem: Recorre memStack de arriba hacia abajo; en el primer mapa que contenga addr, devuelve su valor. Si ningún marco lo tiene, retorna false.

- writeMem: Recorre memStack de arriba hacia abajo para encontrar si addr ya existe en un marco. Si lo encuentra, actualiza ese valor; si no aparece en ningún marco, lo inserta en el global (memStack[0]).

Cuando la VM ejecuta un cuádruplo PARAM, toma q.Arg1, dirección donde está el valor del argumento, y q.Result (índice 1-basado del parámetro), calcula idx := q.Result - 1 y luego targetDir := ParamAddrs[nombreFunc][idx]. De este modo, el valor se escribe directamente en la dirección correcta del marco local que ya fue apilado por el ERA.

Cada vez que se procesa GOSUB, se empuja m.IP+1 para recordar la instrucción a la que debe volver la VM cuando termine la función (con ENDF). En ENDF, la VM saca de retStack esa dirección, desapila el marco local (memStack) y restaura IP, reanudando la ejecución justo después de la llamada.

Ejecución.

La ejecucion de la máquina virtual se realiza dentro de un ciclo en la funcion Run( ) en el archivo vm_execution.go el cual actualiza el índice del IP dependiendo del cuádruplo en el que se encuentra hasta que el índice del puntero es mayor o igual que el tamaño de la pila de cuádruplos.

- Aritmética (ADD, SUB, MUL, DIV): lee operandos con readMem(q.Arg1) y readMem(q.Arg2), hace la operación sobre enteros, y guarda el resultado con writeMem(q.Result, out). Luego suma uno al IP.
- Comparaciones (LT, GT, NEQ): Produce un entero 1 si la comparación es cierta o 0 si es falsa, que se almacena en q.Result.
- GOTOFALSE: Lee el primer argumento; si es falso, asigna al puntero el index del resultado, sino suma uno al IP.
- GOTO: asigna el resultado al indice del puntero.
- ERA: obtiene la entrada de función con lookupFuncEntry(q.Arg1), crea un mapa local iniciando todas las variables locales en cero, y lo apila en m.memStack. También guarda el nombre de la función en un stack auxiliar (callStack) para asignar parámetros. Luego suma uno al puntero.
- PARAM: Asigna un valor en la dirección addrs[q.Result-1] dentro del marco local activo.
- GOSUB: Agrega la direccion siguiente del puntero actual a la pila del retStack para cuando acabe la función que se va a llamar, obtiene la direccion de cuadruplo de la funcion y asiga ese index al puntero.

- ENDF: Para la función actual, saca el ultimo elemento de la pila de retStack, desapila el marco local, quita el nombre de la función de callStack y asigna el retorno al índice del puntero.
- EQ: lee val := m.readMem(q.Arg1) y lo guarda en q.Result con writeMem(q.Result, val), luego agrega uno al puntero,
- PRINT: lee el valor del resultado, concatena en PrintOutput y hace amento uno al puntero.
- END: Finaliza el ciclo y retorna PrintOutput.String().

Liga de Github de la entrega:
https://github.com/danielaramosgarcia/Aplicaciones_avanzadas_TTC3002B/tree/main/Compiladores/babyduck

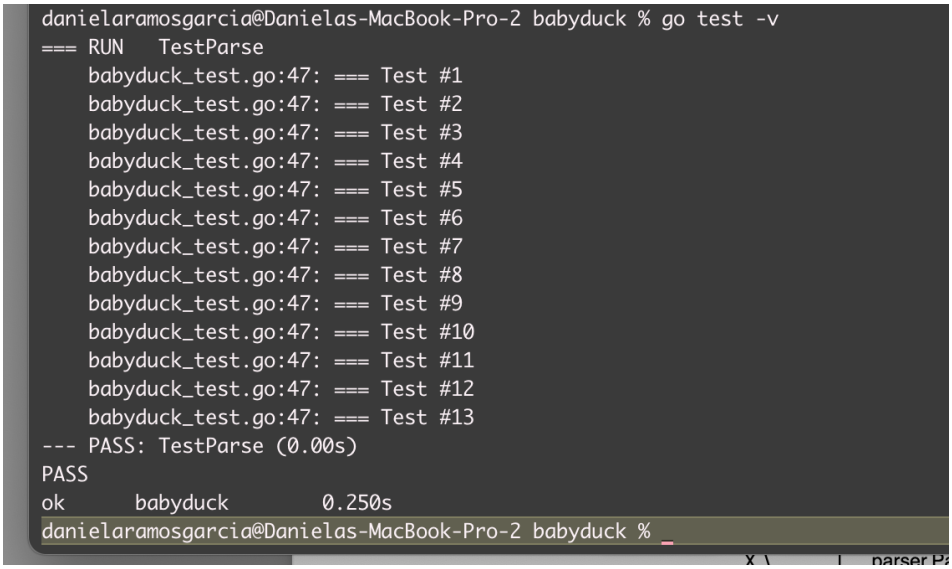
Pruebas.

Análisis léxico y sintáctico.

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	** Testing comments ** program p; var x: int; main { x = 5; } end	Sin errores de parseo
TC6	Estatuto Print	program p; var x: int; main { x = 5; print(x); } end	Sin errores de parseo
TC7	Whitespace y newlines	"** Testing new line whitespace ** program p;\n\tmain { }\n\tend"	Sin errores de parseo
TC8	Ciclo while en varias lineas	"program p; var x: int; main\n\t{ while (x < 10) do { x = x + 1; }; }\n\tend"	Sin errores de parseo
TC9	Falta punto y coma y id	program p main { } end	Error de parseo (falta ; tras p)
TC10	Falta el token end	program p; main { }	Error de parseo (falta palabra end)

TC11	Punto y coma después de asignación	program p; var x: int; main { x = 5 } end	Error de parseo (falta ; tras 5)
TC12	Texto extra después de end	program p; var x: int; main { x = 5; } end extra	Error de parseo (texto inesperado)
TC13	Declaración de función	program p; var x: int; void f(a: int) [{ b = a + 2; }]; main { } end	Sin errores de parseo

Screenshot de ejecución de pruebas.



Plan de pruebas semánticas

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

TC20	Declaración y uso de variable local	program p; void f(){ var y:int; y=1; } main { } end	Sin errores
TC21	Acceso a variable global dentro de función	program p; var x:int; void f(){ x=2; } main { } end	Sin errores

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.

```
/dev/fd/13:18: command not found: compdef
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
```

Máquina Virtual.

Ejecuciones de prueba:

Input	Output

program fibonacci;
  var n, a, b, count, temp: int;
main {
  n = 10;
  if (n < 0) {
  print('Error: n must be greater than 0');
  };
  if (n < 1){
  print ('Fibonacci of', n, 'is', 0);
  };
  if (n < 2) {
  print('Fibonacci of', n, 'is', 1);
  } else {
  a = 0;
  b = 1;
  count = 2;
  while (count < n + 1) do {
  temp = a + b;
  a = b;
  b = temp;
  count = count + 1;
  };
  print('Fibonacci of', n, '1s', b);
  };
} end
	Fibonacci of 20 is 6765

program fib;
  var n, a, b, count, temp: int;
void fibonacci(input: int)
[{
  a = 0;
	Fibonacci of 10 is 55

b = 1;
count = 2;
while (count < input + 1) do {
temp = a + b;
a = b;
b = temp;
count = count + 1;
};
print('Fibonacci of', input, 'is', b);
}
];
 	
main {
n = 10;
if (n < 0) {
print('Error: n must be greater than 0');
};
if (n < 1){
print ('Fibonacci of', n, 'is', 0);
};
if (n < 2) {
print('Fibonacci of', n, 'is', 1);
} else {
fibonacci(n);
};
}
end	

program fact;
 	
var n, result, count :int;
 	
void factorial(input: int)[
{
result = 1;
count = 1;
	Factorial 10 is 3628800

while (count < input + 1) do {
result = result * count;
count = count + 1;
};
print('Factorial of', input, 'is', result);
}
];
 	
main {
n = 10;
if (n < 0) {
print('Error: n must be greater than 0');
};
if (n < 1) {
print('Factorial of', n, 'is', 1);
} else {
factorial(n);
};
}
end	

program fact;
 	
var n, result, count :int;
 	
main {
n = 10;
if (n < 0) {
print('Error: n must be greater than 0');
};
if (n < 1) {
print('Factorial of', n, 'is', 1);
} else {
result = 1;
count = 1;
while (count < n + 1) do {
result = result * count;
	Factorial of 20 is 2432902008176640000

count = count + 1;
};
print('Factorial of', n, 'is', result);
};
}
end	