Tecnológico de Monterrey - Campus Monterrey

BabyDuck - entrega #5 (Código de Funciones y Ejecución de Expresiones)

Desarrollo de aplicaciones avanzadas de ciencias computacionales

Profa. Elda Guadalupe Quiroga González

Daniela Ramos García A01174259

27 de mayo del 2025

Indice

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.

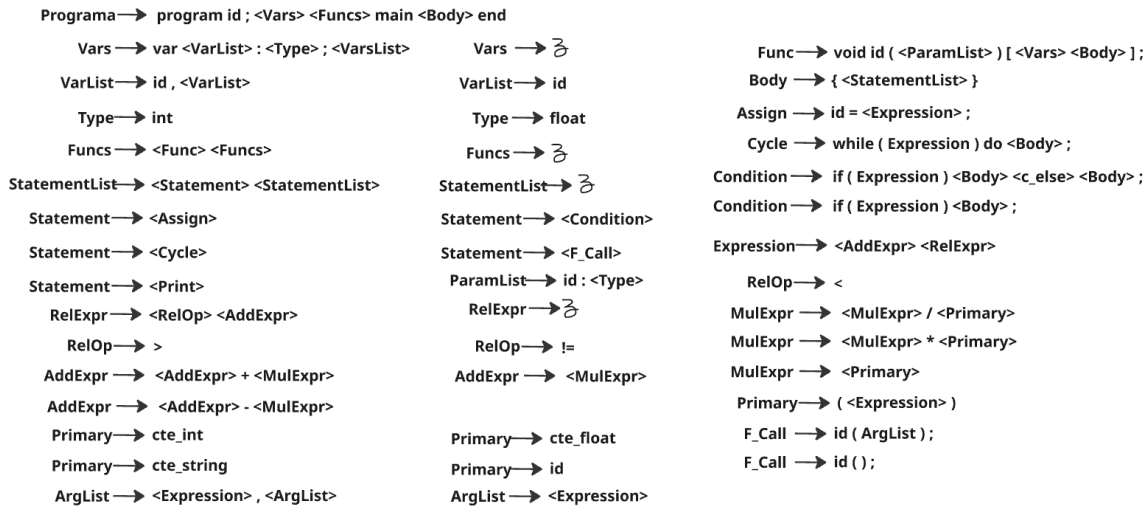
2. Expresiones Regulares

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).



5. Declaración de sintax en BNF.

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

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

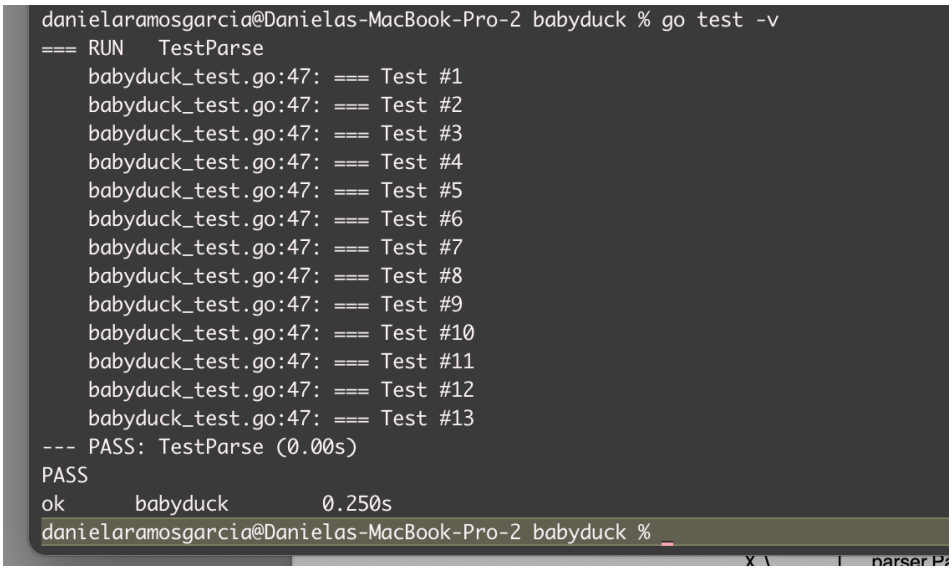
- Cambio en BNF: obliga a reescribir casos → mantener versión congelada durante pruebas.
- Conflictos LR(1): usar gocc -a para resolver automáticamente.
- Cobertura insuficiente: planear revisión de casos adicionales tras primera pasada.

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	** 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.



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

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

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 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 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.

```
/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
```

15. Traduce las variables, constantes y temporales a sus respectivas Direcciones virtuales.

```
// Simulación de direcciones de memoria (stack) por segmentos:
const (
    // Variables globales [0,500]
    GlobalIntBase   = 1
    GlobalFloatBase = 250
    GlobalLimit     = 500

    // Variables locales [501,1000]
    LocalIntBase   = 501
    LocalFloatBase = 751
    LocalLimit     = 1000

    // Variables temporales [1001,1600]
    TempIntBase   = 1001
    TempFloatBase = 1201
    TempBoolBase  = 1401
    TempLimit     = 1600

    // Constantes [1601,2200]
    ConstIntBase    = 1601
    ConstFloatBase  = 1801
    ConstStringBase = 2001
    ConstLimit      = 2200
)
```

```
// TanslateOp recibe un operador como string y devuelve su representación numérica
func TanslateOp(op string) (int, error) {
    switch op {
    case "+":
        return 10, nil
    case "-":
        return 20, nil
    case "*":
        return 30, nil
    case "/":
        return 40, nil
    case "<":
        return 50, nil
    case ">":
        return 60, nil
    case "!=":
        return 70, nil
    default:
        return -1, fmt.Errorf("operador desconocido: %s", op)
    }
}
```

```go
/*
Traduccion de tipo a su representacion en numero

    int -> 0
    float -> 1
    bool -> 2
*/
func TranslateType(typ string) (Tipo, error) {
    switch typ {
    case "int":
        return 0, nil
    case "float":
        return 1, nil
    case "bool":
        return 2, nil
    default:
        return -1, fmt.Errorf("tipo desconocido: %s", typ)
    }
}
```

16. Cuádruplos para la declaración e invocación de las funciones de BabyDuck

La función PrintQuad extrae primero de la pila de operandos (PilaO) la dirección o el valor que debe imprimirse y, a continuación, obtiene de la pila de tipos (PTypes) el tipo asociado a ese operando. Realiza un chequeo semántico para asegurar que el tipo sea uno de los imprimibles permitidos (entero, flotante o cadena) y, de no cumplirse esta condición, lanza un error de desajuste de tipo. Si el tipo es correcto, genera y encola un cuádruplo con la operación PRINT, dejando los campos Arg1 y Arg2 sin uso (marcados con –1) y almacenando en Result la dirección u operando extraído, de modo que el motor de ejecución sepa qué dato volcar en la salida estándar.

La función MakeEndFQuad no interactúa con las pilas de operandos ni de tipos; su única responsabilidad es marcar el punto de terminación de la función actual. Para ello, construye y encola un cuádruplo cuyo operador es ENDF y cuyos tres argumentos (Arg1, Arg2 y Result) se establecen en –1, lo que señala al generador de código intermedio y al back-patcher que a partir

de este punto debe finalmente cerrar el registro de activación y preparar los saltos de retorno correspondientes.

La función MakeEraQuad, al recibir como parámetro el identificador de la función destino, consulta la tabla directora de funciones (FuncDir) para obtener el índice interno asociado a ese nombre. Con dicho índice construye un cuádruplo ERA (Activation Record Expansion) en el que asigna el índice a Arg1 y deja Arg2 y Result en –1. Este cuádruplo reserva el espacio necesario para el registro de activación de la función llamada, calculando allí la memoria para variables locales, temporales y parámetros antes de que se ejecute cualquier instrucción de llamada real.

La función MakeParamQuad comienza sacando de las pilas el operando del argumento y su tipo, y compara este último con el tipo esperado según la firma de la función almacenada en FuncSignature.ParamSignature. Si hay discrepancia, arroja un error de “type mismatch”. Tras validar la coincidencia, elimina de la firma el tipo ya procesado y calcula la posición ordinal del parámetro restando el tamaño restante de la firma al número inicial de parámetros. Finalmente, encola un cuádruplo PARAM cuyas entradas incluyen el operando extraído en Arg1, un –1 en Arg2, y el índice calculado en Result, de modo que cada argumento quede correctamente ubicado en su slot dentro del registro de activación.

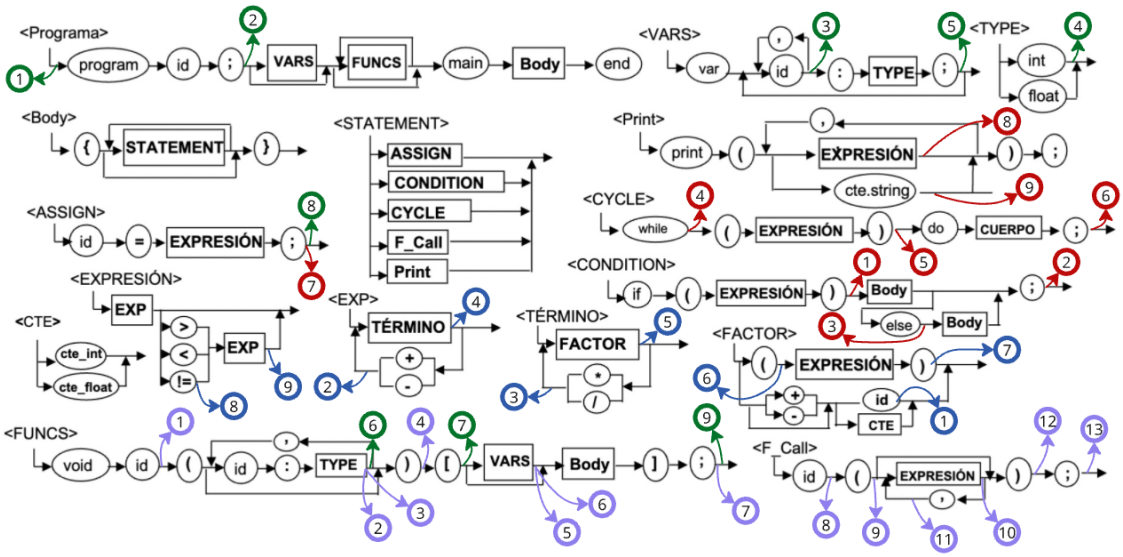
17. Agregar en los diagramas del lenguaje BabyDuck los puntos neurálgicos.

Puntos neuralgicos.

DF: dirección de funciones
TV: tabla de variables
DM: dirección de memoria

TC3002B: Desarrollo de aplicaciones avanzadas de Ciencias Computacionales

Módulo: Compiladores **Mini Proyecto INDIVIDUAL :Baby_Duck** **Abril2025**



Semantica de variables.

1. Crea un contexto nuevo del programa, crea su DF y su respectiva TV.
2. Agrega una función al DF con el id del programa y entra a la función.
3. Construye arreglo para almacenar 1 o más ids de variables.
4. Verifica que el Type sea int o float.
5. Agrega las variables a la TV de la función actual.
    a. Verifica que no exista la variable previamente en la TV
    b. Asigna dirección de memoria según el tipo.
6. Construye arreglo para 1 o más parámetros, guarda su id y tipo
7. Registra la función en la DF, agrega sus parámetros dentro de su TV y entra a la función.
    a. Asigna la TV padre dentro de su TV.
8. Valida que el tipo de la expresión sea el mismo que con el que se declaro el id.
9. Sale de la función y reinicia los contadores de las DM.

Código de Expresiones y Estatutos lineales

1. Encuentra la variable en la VT, agrega el operando y el tipo a sus respectivas pilas.
2. Traduce el Operador de suma o resta en número y lo agrega a la pila de operadores.
3. Traduce el Operador de multiplicación o división en número y lo agrega a la pila de operadores.
4. Se genera el cuádruplo.
5. Se genera el cuádruplo.
6. Se agrega el operador de "(" como fondo falso
7. Se agrega la lógica para hacer pop a los operadores y operandos hasta topar con el fondo falso.
8. Agregar operador de expresiones booleanas
9. Generar cuádruplo.

Estatutos

1. Checka el tipo de la expresión y sigue si es booleano, agrega el index del cuádruplo a el stack de jumps, genera un cuadruplo de tipo GOTOFALSE y deja la dirección vacia.
2. Hace pop a el ultimo salto del jump stack y lo llena con la direccion de cuadruplo siguiente.
3. Hace pop a el último salto del jump stack y lo llena con la dirección del cuadruplo siguiente. Agrega un Cuadruplo de tipo GOTO.
4. Se agrega la dirección actual al stack de jump para poder agregarlo al cuadruplo final de la condición.
5. Paso 1.
6. Agrega un cuádruplo de tipo GOTO, llena el cuádruplo del paso 1 a el siguiente cuádruplo y luego llena la dirección del ultimo GOTO hacia el primer cuádruplo del while
7. Hace pop a los ultimos dos operandos y sus tipos y crea un cuádruplo de tipo assign con ellos.
8. Hace pop al operator stack y operand stack, verifica que sea de tipo TYPE y crea un quad de tipo print con su valor.
9. Hace pop al operator stack y operand stack, verifica que sea de tipo String y crea un quad de tipo print con su valor.

Estatutos

1. Insertar el nombre de la función en la tabla DirFunc
2. Insertar cada parámetro en la VT local.
3. Insertar el array de tipos de parametros en orden la ParamList.
4. Insertar en DirFunc el número de parámetros definidos para calcular el espacio de trabajo requerido para la ejecución.
5. Insertar en DirFunc el número de variables locales definidas para calcular el espacio de trabajo requerido para la ejecución.
6. Insertar en DirFunc el contador actual de cuadruplos (CONT) para establecer dónde inicia la función.
7. Se agrega un cuádruplo de tipo ENDFUNC.
8. Verificar que la función exista en DirFunc.
9. Generar la acción ERA
10. Verificar tipo de exp y pop a el ultimo operando para mandarlo a la función.
11. k = k + 1; avanzar al siguiente parámetro.
12. Verificar que el último parámetro apunte a null.
13. Generar la acción GOSUB, nombre-procedimiento, , dirección-inicial.

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