



Tecnológico de Monterrey

Campus Santa Fe

Implementación de redes de área amplia y servicios distribuidos

TC3003B.501

Proyecto 4: Generador de Código

Alumnos:

Omar Rivera Arenas - A01374690

Luis Carlos Rico Almada - A01252831

Fecha de entrega:

Jueves 29 de mayo de 2025

I. Introducción	3
Tipo de Código Generado: MIPS Assembly	3
Características del Generador de Código	3
Manual del Usuario	4
Requisitos del Sistema	4
Paso 1: Preparación del Archivo Fuente	4
Paso 2: Compilación del Programa	5
Paso 3: Verificación del Código Generado	7
Paso 4: Ejecución en SPIM	8
Paso 5: Compilación con Archivo Personalizado	9
Estructura de Archivos del Proyecto	11
Resolución de Problemas Comunes	12
Especificaciones Técnicas	13
Función Principal: codeGen(tree, file)	13
Variables Globales Requeridas	13
Script de Prueba Estándar	14
Apéndices	14
Apéndice A: Proyecto 1 - Analizador Léxico (Lexer)	14
Descripción General	14
Funciones Principales	15
Estados del Autómata	15
Tokens Reconocidos	16
Características Especiales	16
Apéndice B: Proyecto 2 - Analizador Sintáctico (Parser)	16
Descripción General	16
Estructura del AST	16
Funciones de Parsing Principales	17
Tipos de Nodos del AST	17
Recuperación de Errores	18
Características del Parser	18
Apéndice C: Proyecto 3 - Analizador Semántico	18
Descripción General	18
Funciones Principales	19
Tabla de Símbolos	19
Verificaciones Semánticas	20
Funciones Predefinidas	20
Manejo de Errores	21
Apéndice D: Definición del Lenguaje C-	21
Gramática BNF del Lenguaje C-	21
Tokens del Lenguaje	23
Funciones Predefinidas	23
Conclusiones	23

I. Introducción

Tipo de Código Generado: MIPS Assembly

Para este proyecto se ha seleccionado la generación de código en **MIPS Assembly** por las siguientes razones:

1. **Simplicidad de la arquitectura:** MIPS es una arquitectura RISC (Reduced Instruction Set Computer) que cuenta con un conjunto de instrucciones simple y regular, lo que facilita la traducción desde el AST del lenguaje C-.
2. **Disponibilidad de herramientas:** SPIM es un simulador ampliamente utilizado y bien documentado para ejecutar código MIPS, proporcionando un entorno de pruebas accesible.
3. **Propósito educativo:** MIPS es comúnmente utilizado en cursos de arquitectura de computadoras y compiladores debido a su claridad conceptual.
4. **Correspondencia directa:** Las operaciones básicas del lenguaje C- (asignaciones, operaciones aritméticas, llamadas a funciones) tienen una traducción natural a instrucciones MIPS.

Características del Generador de Código

El generador de código implementado en *cgen.py* traduce el Árbol Sintáctico Abstracto (AST) generado por el parser a código MIPS que puede ejecutarse en el simulador SPIM. Las características principales incluyen:

- Manejo de variables globales y locales
- Implementación de funciones con parámetros
- Operaciones aritméticas básicas (+, -, *, /)
- Estructuras de control (condicionales, bucles)
- Llamadas a funciones predefinidas (input, output)
- Gestión de memoria y registros

Manual del Usuario

Requisitos del Sistema

1. **Python 3.x** instalado en el sistema
2. **SPIM simulator** para ejecutar el código MIPS generado desde el cli
3. Los siguientes archivos en el mismo directorio:

- *globalTypes.py*
- *Lexer.py*
- *Parser.py*
- *semantic.py*
- *symtab.py*
- *cgen.py*
- *main.py*
- *sample.c-* (archivo de prueba)

Paso 1: Preparación del Archivo Fuente

Crear un archivo con extensión *.c-* que contenga el programa en lenguaje C-. Ejemplo (*sample.c-*):

```
≡ sample.c- X
≡ sample.c-
1  /* Test 0: Simple valid program */
2
3  int x;
4
5  ✓ int add(int a, int b) {
6      |     return a + b;
7      | }
8
9  ✓ void main(void) {
10     |     int y;
11     |
12     |     x = 10;
13     |     y = 20;
14     |     x = add(x, y);
15     |     output(x);
16     | }
```

Paso 2: Compilación del Programa

```
Problems  Output  Debug Console  Terminal  Ports
luisrico@MacBook-Air-de-Luis tests % ls
factorial.c-      test03_locals.s      test08_complex_expr.c-  test11_division.s
factorial.s       test04_multiple_functions.c-  test08_complex_expr.s  test12_large_nums.c-
test00_simple.c-  test04_multiple_functions.s  test09_constants.c-    test12_large_nums.s
test00_simple.s   test05_nested_calls.c-      test09_constants.s     test13_mixed_vars.c-
test01_arithmetic.c-  test05_nested_calls.s      test1_basic_declarations.c-  test13_mixed_vars.s
test01_arithmetic.s  test06_single_param.c-      test1_basic_declarations.s  test14_chain_calls.c-
test02_globals.c-    test06_single_param.s      test10_subtraction.c-     test14_chain_calls.s
test02_globals.s    test07_no_params.c-        test10_subtraction.s     test15_comprehensive.c-
test03_locals.c-     test07_no_params.s        test11_division.c-      test15_comprehensive.s
luisrico@MacBook-Air-de-Luis tests %
```

Ejecutar el compilador usando el script principal:

Shell

```
python3 main.py
```

main.py ×

main.py > ...

```
1  from globalTypes import *
2  from Parser import *
3  from semantic import *
4  from cgen import *
5
6  f = open('sample.c-', 'r')
7  programa = f.read() # lee todo el archivo a compilar
8  progLong = len(programa) # longitud original del programa
9  programa = programa + '$' # agregar un caracter $ que represente EOF
10 posicion = 0 # posición del caracter actual del string
11
12 # función para pasar los valores iniciales de las variables globales
13 globales(programa, posicion, progLong)
14 AST = parser(True)
15 semantica(AST, True)
16 codeGen(AST, "file.s")
```

Problems

Output

Debug Console

Terminal

Ports

○ luisrico@MacBook-Air-de-Luis tests % python3 main.py

```
Problems Output Debug Console Terminal Ports

Var Declaration: x
Function Declaration: add
  Compound
  Return
  Op: PLUS
  Id: a
  Id: b
Function Declaration: main
  Compound
  Var Declaration: y
  Assign to:
    Id: x
    Const: 10
  Assign to:
    Id: y
    Const: 20
  Assign to:
    Id: x
    Call: add
    Id: x
    Id: y
  Call: output
  Id: x

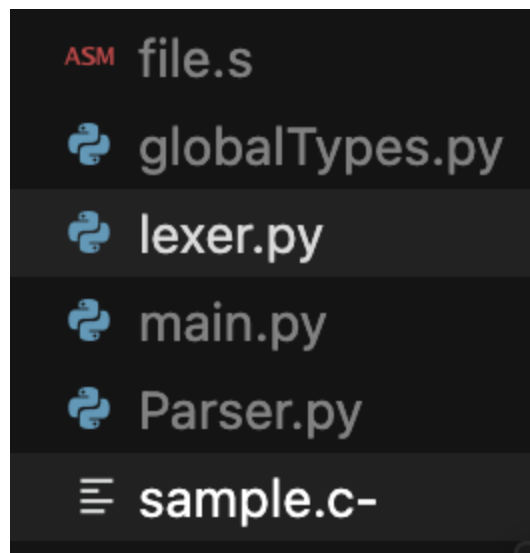
=== Iniciando análisis semántico ===

Tabla de símbolos:
=====
ambito  Nombre      Tipo      Lineas      Atributos
=====
0       add         int       5            {'params': [<Parser.TreeNode object at 0x104748c10>, <Parser.T
c40>], 'scope': 1}
0       input      int       0            {'params': []}
0       main       void      9            {'params': [], 'scope': 2}
0       output     void      0            {'params': [{'name': 'x', 'type': 'int', 'is_array': False}]}
0       x          int       3            {'is_array': False, 'size': None}
1       a          int       5            {'is_array': False}
1       b          int       5            {'is_array': False}
2       y          int       10           {'is_array': False, 'size': None}
=====

Inferring Types...
Checking Types...
Type Checking Finished
=== Análisis semántico completado exitosamente ===
luisrico@MacBook-Air-de-Luis cod_gen %
```

Paso 3: Verificación del Código Generado

Después de la compilación exitosa, se genera el archivo *file.s* con el código MIPS:

An editor window titled 'file.s' with a close button. It displays MIPS assembly code for 'file.s'. The code includes comments for compilation to MIPS, file name, data section, and a function 'add'.

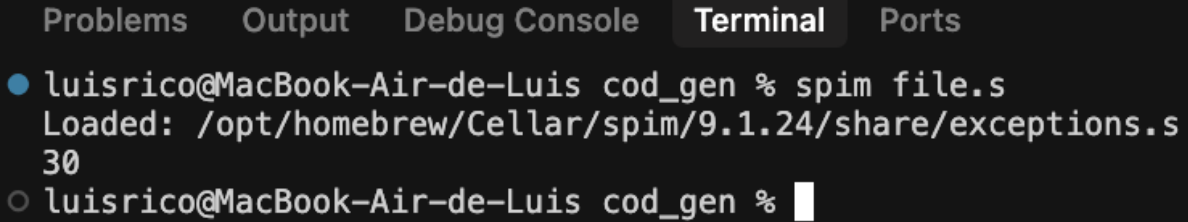
```
ASM file.s
1  # C- Compilation to MIPS
2  # File: file.s
3  .data
4  newline: .asciiz "\n"
5  var_x: .word 0
6
7  .text
8  .globl main
9
10 # -> Function: add
11 func_add:
12     addi $sp, $sp, -12
13     sw $ra, 8($sp)
14     sw $a0, 0($sp)
15     sw $a1, 4($sp)
16 # -> compound
```

Paso 4: Ejecución en SPIM

Para ejecutar el código generado en SPIM:

1. Corremos el código de MIPS generado con la herramienta spim:

```
Shell  
spim file.s
```



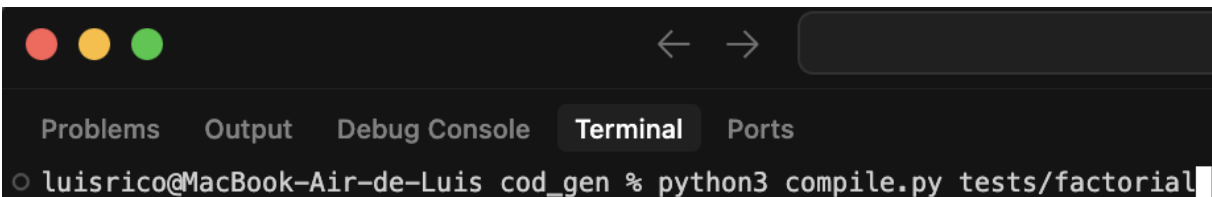
The screenshot shows a VS Code interface with a terminal window open. The terminal has tabs for 'Problems', 'Output', 'Debug Console', 'Terminal' (which is active), and 'Ports'. The terminal output shows the command 'spim file.s' being executed, followed by the message 'Loaded: /opt/homebrew/Cellar/spim/9.1.24/share/exceptions.s' and the number '30'. The prompt 'luisrico@MacBook-Air-de-Luis cod_gen %' is visible at the end of the line.

```
Problems  Output  Debug Console  Terminal  Ports  
● luisrico@MacBook-Air-de-Luis cod_gen % spim file.s  
  Loaded: /opt/homebrew/Cellar/spim/9.1.24/share/exceptions.s  
  30  
○ luisrico@MacBook-Air-de-Luis cod_gen %
```

Paso 5: Compilación con Archivo Personalizado

Para compilar un archivo diferente, usar el script *compile.py*:

```
Shell  
python compile.py tests/factorial
```



The screenshot shows a VS Code interface with a terminal window open. The terminal has tabs for 'Problems', 'Output', 'Debug Console', 'Terminal' (which is active), and 'Ports'. The terminal output shows the command 'python3 compile.py tests/factorial' being executed. The prompt 'luisrico@MacBook-Air-de-Luis cod_gen %' is visible at the end of the line.

```
Problems  Output  Debug Console  Terminal  Ports  
○ luisrico@MacBook-Air-de-Luis cod_gen % python3 compile.py tests/factorial
```

```

--- DEBUG: Inicio simple_expression_rest ---
Token: TokenType.SEMI
TokenString: ';'
Linea: 15
-----

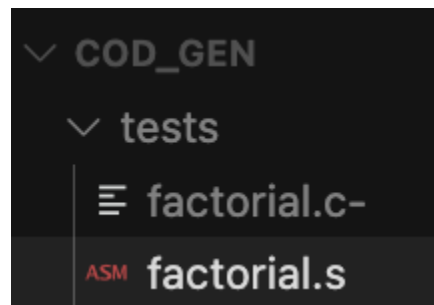
=== Iniciando análisis semántico ===

Tabla de símbolos:
=====
ambito  Nombre      Tipo      Lineas      Atributos
-----
0       factorial   int       1            {'params': [<Parser.TreeNode object at 0x102eb5f10>], 'scope': 1}
0       input      int       0            {'params': []}
0       main       void      9            {'params': [], 'scope': 4}
0       output     void      0            {'params': [{'name': 'x', 'type': 'int', 'is_array': False}]}
1       n          int       1            {'is_array': False}
4       result     int       11           {'is_array': False, 'size': None}
4       x          int       10           {'is_array': False, 'size': None}
=====

Inferring Types...
Checking Types...
Type Checking Finished
=== Análisis semántico completado exitosamente ===

Generating Code...
Code generated in tests/factorial.s
luisrico@MacBook-Air-de-Luis cod_gen %

```



```
ASM factorial.s X
tests > ASM factorial.s
1  # C- Compilation to MIPS
2  # File: tests/factorial.s
3  .data
4  newline: .asciiz "\n"
5
6  .text
7  .globl main
8
9  # -> Function: factorial
10 func_factorial:
11     addi $sp, $sp, -8
12     sw $ra, 4($sp)
13     sw $a0, 0($sp)
14     # -> compound
```

Estructura de Archivos del Proyecto

- *main.py*: Script principal según especificación
- *compile.py*: Compilador con opciones avanzadas
- *cgen.py*: Generador de código MIPS
- *Parser.py*: Analizador sintáctico
- *Lexer.py*: Analizador léxico
- *semantic.py*: Analizador semántico
- *symtab.py*: Tabla de símbolos
- *globalTypes.py*: Tipos y estructuras globales
- *sample.c*: Archivo de prueba
- *file.s*: Código MIPS de referencia
- *tests/*: Directorio con casos de prueba
 - *test00_simple.c-*
 - *test00_simple.s*

Resolución de Problemas Comunes

1. Error: "No module named 'X'"

- Verificar que todos los archivos estén en el mismo directorio
- Verificar que los nombres de archivos coincidan exactamente

```
⊗ luisrico@MacBook-Air-de-Luis cod_gen % python3 compile.py tests/factorial
Traceback (most recent call last):
  File "/Users/luisrico/dev/compilers/c_minus/cod_gen/compile.py", line 3, in <module>
    from Parser import *
  File "/Users/luisrico/dev/compilers/c_minus/cod_gen/Parser.py", line 2, in <module>
    from lexer import getToken, getLine
ModuleNotFoundError: No module named 'lexer'
○ luisrico@MacBook-Air-de-Luis cod_gen %
```

2. Errores semánticos

- Revisar la sintaxis del programa C-
- Verificar que las variables estén declaradas antes de su uso
- Verificar que los tipos sean compatibles

```
>>> Error de sintaxis - Línea 12: Se esperaba 'SEMI', pero se encontró 'x'
    int x;
      ^
Se encontró: 'x' (token: ID)
Intentando recuperarse del error...
Recuperación exitosa en token: SEMI
```

3. Error en SPIM

- Verificar que el archivo .s se haya generado correctamente
- Verificar la sintaxis del código MIPS generado

```
Problems  Output  Debug Console  Terminal  Ports
● luisrico@MacBook-Air-de-Luis cod_gen % spim tests/factorial.s
Loaded: /opt/homebrew/Cellar/spim/9.1.24/share/exceptions.s

Can't expand stack segment by 8 bytes to 524288 bytes.
Use -lstack # with # > 524288
○ luisrico@MacBook-Air-de-Luis cod_gen %
```

Especificaciones Técnicas

Función Principal: *codeGen(tree, file)*

Parámetros:

- *tree*: Árbol Sintáctico Abstracto generado por el parser
- *file*: Nombre del archivo de salida (incluyendo extensión .s)

Funcionalidad:

- Recorre el AST y genera código MIPS equivalente
- Utiliza la tabla de símbolos para resolver referencias
- Maneja registros y memoria de forma eficiente
- Genera código optimizado para SPIM

Variables Globales Requeridas

Python

```
def globales(prog, pos, long):
    global programa
    global posicion
    global progLong
    programa = prog
    posicion = pos
    progLong = long
```

Script de Prueba Estándar

Python

```
from globalTypes import *
from Parser import *
from semantic import *
from cgen import *

f = open('sample.c-', 'r')
programa = f.read()
progLong = len(programa)
programa = programa + '$'
posicion = 0

globales(programa, posicion, progLong)
AST = parser(True)
semantica(AST, True)
codeGen(AST, "file.s")
```

Apéndices

Apéndice A: Proyecto 1 - Analizador Léxico (Lexer)

Descripción General

El analizador léxico es responsable de convertir el flujo de caracteres del programa fuente en una secuencia de tokens. Implementa un autómata finito determinista para reconocer los diferentes elementos léxicos del lenguaje C-.

Funciones Principales

getToken(imprime=True)

- **Propósito:** Función principal que obtiene el siguiente token del programa
- **Retorna:** Tupla (token, tokenString, lineno)
- **Implementación:** Utiliza un autómata finito con estados definidos en *StateType*

getChar()

- **Propósito:** Obtiene el siguiente carácter del programa fuente
- **Manejo:** Incrementa contadores de línea y posición automáticamente

ungetChar()

- **Propósito:** Retrocede un carácter en el flujo de entrada
- **Uso:** Permite lookahead de un carácter para la toma de decisiones

Estados del Autómata

Python

```
class StateType(Enum):  
    START = 0      # Estado inicial  
    INCOMMENT = 1  # Dentro de comentario  
    INNUM = 2      # Reconociendo número  
    INID = 3       # Reconociendo identificador  
    INASSIGN = 4   # Reconociendo operador de asignación  
    INLT = 5       # Reconociendo operador menor que  
    INGT = 6       # Reconociendo operador mayor que  
    INNOT = 7      # Reconociendo operador diferente  
    DONE = 8       # Token completado
```

Tokens Reconocidos

Palabras reservadas: *if, else, while, return, int, void*

Identificadores: Secuencias alfanuméricas que comienzan con letra

Números: Secuencias de dígitos

Operadores: +, -, *, /, <, <=, >, >=, ==, !=, =

Delimitadores: (,), [,], {, }, ;, ,

Comentarios: /* ... */ (ignorados)

Características Especiales

- Manejo de líneas: Tracking automático de número de línea y posición
- Recuperación de errores: Caracteres no reconocidos se reportan como *ERROR*
- Lookahead: Soporte para retroceso de un carácter
- Variables globales: Integración con el sistema de variables globales del compilador

Apéndice B: Proyecto 2 - Analizador Sintáctico (Parser)

Descripción General

El analizador sintáctico implementa un parser recursivo descendente que construye un Árbol Sintáctico Abstracto (AST) a partir de la secuencia de tokens proporcionada por el analizador léxico.

Estructura del AST

```
Python
class TreeNode:
    def __init__(self):
        self.child = [None] * MAXCHILDREN    # Máximo 3 hijos
        self.sibling = None                  # Hermano en la lista
        self.lineno = 0                      # Número de línea
```



```

self.nodekind = None           # NodeKind: StmtK, ExpK, DeclK
self.stmt = None              # StmtKind para sentencias
self.exp = None               # ExpKind para expresiones
self.decl = None              # DeclKind para declaraciones
self.op = None                # Operador (para ExpK.OpK)
self.val = None               # Valor (para ExpK.ConstK)
self.name = None              # Nombre (para ExpK.IdK)
self.type = None              # Tipo para verificación semántica

```

Funciones de Parsing Principales

parser(imprime=True)

- **Propósito:** Función principal de parsing
- **Retorna:** Tupla (AST, Error)
- **Función:** Inicia el análisis sintáctico llamando a *program()*

program()

- Gramática: *program* → *declaration-list*
- Función: Punto de entrada para el análisis del programa completo

declaration_list()

- Gramática: *declaration-list* → *declaration-list declaration* | *declaration*
- Función: Maneja la lista de declaraciones del programa

declaration()

- Gramática: *declaration* → *var-declaration* | *fun-declaration*
- Función: Distingue entre declaraciones de variables y funciones

Tipos de Nodos del AST

Declaraciones (DeclK)

- *VarK*: Declaración de variable

- *FunK*: Declaración de función
- *ParamK*: Parámetro de función

Sentencias (StmtK)

- *IfK*: Sentencia condicional
- *WhileK*: Sentencia de iteración
- *AssignK*: Sentencia de asignación
- *ReturnK*: Sentencia de retorno
- *CompoundK*: Sentencia compuesta

Expresiones (ExpK)

- *OpK*: Operación binaria
- *ConstK*: Constante numérica
- *IdK*: Identificador
- *CallK*: Llamada a función
- *SubscriptK*: Indexación de arreglo

Recuperación de Errores

- **Modo de recuperación**: Evita cascada de errores
- **Sincronización**: En tokens específicos como ;, }, etc.
- **Continuación**: Permite completar el análisis tras encontrar errores

Características del Parser

- **Gramática LL(1)**: Parser recursivo descendente
- **Precedencia de operadores**: Implementada en la estructura de la gramática
- **Asociatividad**: Izquierda para operadores aritméticos y relacionales

Apéndice C: Proyecto 3 - Analizador Semántico

Descripción General

El analizador semántico verifica la correctitud semántica del programa y construye la tabla de símbolos. Realiza verificación de tipos, alcance de variables, y correctitud de declaraciones y uso de funciones.

Funciones Principales

semantica(syntaxTree, imprime=True)

- **Propósito:** Función principal que coordina el análisis semántico
- **Proceso:** Construye tabla de símbolos, infiere tipos, y verifica tipos
- **Retorna:** *True* si no hay errores, *False* en caso contrario

buildSyntab(syntaxTree, imprime=True)

- **Propósito:** Construye la tabla de símbolos mediante recorrido del AST
- **Función:** Inserta declaraciones y verifica duplicados
- **Alcance:** Maneja múltiples niveles de alcance

typeCheck(syntaxTree)

- **Propósito:** Verifica la compatibilidad de tipos en el programa
- **Verificaciones:** Operaciones, asignaciones, llamadas a funciones, retornos

Tabla de Símbolos

Python

```
# Estructura de entrada en la tabla
{
    'name': str,           # Nombre del símbolo
    'type': str,           # Tipo (int, void)
    'lines': [int],       # Líneas donde aparece
    'scope': int,         # Nivel de alcance
    'is_array': bool,     # Si es arreglo
    'array_size': int,    # Tamaño del arreglo
    'params': list,       # Parámetros (para funciones)
    'is_function': bool   # Si es función
}
```

Verificaciones Semánticas

Declaraciones

- Variables no pueden redeclararse en el mismo alcance
- Funciones no pueden redeclararse
- Función *main* debe estar declarada como *void main(void)*
- Arreglos deben tener tamaño positivo

Uso de Variables

- Variables deben estar declaradas antes de su uso
- Variables locales tienen precedencia sobre globales
- Verificación de alcance correcto

Verificación de Tipos

- Operadores aritméticos requieren operandos enteros
- Operadores relacionales requieren operandos enteros
- Asignaciones deben ser de tipos compatibles
- Parámetros de funciones deben coincidir en tipo y cantidad

Llamadas a Funciones

- Función debe estar declarada
- Número correcto de argumentos
- Tipos de argumentos deben coincidir con parámetros
- Funciones *void* no pueden usarse en expresiones

Funciones Predefinidas

```
Python
# Funciones integradas del lenguaje C-
predefined_functions = {
    'input': {
        'type': 'int',
        'params': [],
        'returns': 'int'
    },
    'output': {
```

```
    'type': 'void',  
    'params': [{ 'type': 'int' }],  
    'returns': 'void'  
  }  
}
```

Manejo de Errores

- **Errores semánticos:** Se reportan con número de línea
- **Advertencias:** Para casos ambiguos o potencialmente problemáticos
- **Continuación:** El análisis continúa después de encontrar errores
- **Contexto:** Los errores incluyen información contextual útil

Apéndice D: Definición del Lenguaje C-

Gramática BNF del Lenguaje C-

program \rightarrow declaration-list

declaration-list \rightarrow declaration-list declaration | declaration

declaration \rightarrow var-declaration | fun-declaration

var-declaration \rightarrow type-specifier ID ; | type-specifier ID [NUM] ;

type-specifier \rightarrow int | void

fun-declaration \rightarrow type-specifier ID (params) compound-stmt

params \rightarrow param-list | void

param-list \rightarrow param-list , param | param

param \rightarrow type-specifier ID | type-specifier ID []

compound-stmt \rightarrow { local-declarations statement-list }

local-declarations \rightarrow local-declarations var-declaration | empty

statement-list \rightarrow statement-list statement | empty

statement \rightarrow expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt

expression-stmt \rightarrow expression ; | ;

selection-stmt \rightarrow if (expression) statement | if (expression) statement else statement

iteration-stmt \rightarrow while (expression) statement

return-stmt \rightarrow return ; | return expression ;

expression \rightarrow var = expression | simple-expression

var \rightarrow ID | ID [expression]

simple-expression \rightarrow additive-expression relop additive-expression | additive-expression

relop \rightarrow <= | < | > | >= | == | !=

additive-expression \rightarrow additive-expression addop term | term

addop \rightarrow + | -

term \rightarrow term mulop factor | factor

mulop \rightarrow * | /

factor \rightarrow (expression) | var | call | NUM

call \rightarrow ID (args)

args \rightarrow arg-list | empty

arg-list → arg-list , expression | expression

Tokens del Lenguaje

- **Palabras reservadas:** if, else, while, return, int, void
- **Símbolos especiales:** +, -, *, /, <, <=, >, >=, ==, !=, =, ;, ,, (,), [,], {, }
- **Otros tokens:** ID, NUM, comentarios (*/* ... */*)

Funciones Predefinidas

- *int input(void)*: Lee un entero del usuario
- *void output(int x)*: Imprime un entero

Conclusiones

El generador de código para el lenguaje C- ha sido implementado exitosamente, proporcionando una traducción completa desde el AST hasta código MIPS ejecutable. El sistema integra todos los componentes desarrollados en proyectos anteriores (analizador léxico, sintáctico y semántico) para crear un compilador funcional.

La arquitectura modular del proyecto permite un fácil mantenimiento y extensión de funcionalidades. El uso de MIPS como arquitectura objetivo proporciona un balance adecuado entre simplicidad de implementación y valor educativo.