UNIVERSIDAD AUTÓNOMA DE TAMAULIPAS

Facultad de Ingeniería "Arturo Narro Siller"

Ingeniería en Sistemas Computacionales

PROGRAMACIÓN DE SISTEMAS DE BASE II

Docente: Muñoz Quintero Dante Adolfo

Proyecto Final

Alumnos:
Aldama Trinidad Alfonso René
Domínguez Reyes Pavel Noel

Horario: 13:00-14:00

Semestre y Grupo: 9° “J”

Fecha de entrega: 27 de noviembre del 2025

Índice

Introducción

El presente proyecto tiene como objetivo desarrollar un compilador funcional para un subconjunto específico del lenguaje de programación Python. La finalidad principal del sistema es realizar el análisis léxico, sintáctico y semántico del código fuente ingresado para, posteriormente, traducir y generar su equivalente en Lenguaje Ensamblador para la arquitectura MIPS.

Este compilador no se limita a ser un intérprete, sino que realiza una traducción completa de código de alto nivel a bajo nivel, permitiendo visualizar el proceso de transformación de instrucciones lógicas a instrucciones de máquina.

Dado que se trata de un diseño académico enfocado en la comprensión de las fases de compilación, el lenguaje fuente soportado presenta características específicas que lo diferencian de la implementación estándar de Python:

- **Tipado y Datos:** El compilador está diseñado exclusivamente para el cálculo numérico. Soporta operaciones con números enteros.
- **Restricciones de Tipos:** No se incluye soporte para cadenas de caracteres (*strings*) ni números de punto flotante (*floats*).
- **Sintaxis y Estructura:** A diferencia de Python estándar, este subconjunto no utiliza la indentación como delimitador de bloques de código, simplificando el análisis sintáctico para enfocarse en la generación de código.
- **Arquitectura Objetivo:** El código resultante es compatible con procesadores o simuladores MIPS (como SPIM o MARS).

Para la fase de análisis (frontend del compilador), se ha utilizado la herramienta ANTLR (Another Tool for Language Recognition), la cual facilita la generación de analizadores léxicos y sintácticos a partir de una gramática formal definida.

Instalación y configuración

Requisitos del Sistema

Para la correcta ejecución del compilador, se requiere el siguiente software:

- **Python 3.x:** Lenguaje base del proyecto (confirmado por el entorno virtual venv visible).
- **Java Runtime Environment (JRE):** Necesario para ejecutar la herramienta ANTLR (antlr-4.13.2-complete.jar).
- **Librerías de Python:** Las dependencias se encuentran aisladas en el entorno virtual.

Estructura del Directorio

El proyecto sigue una estructura modular organizada de la siguiente manera:

- **/src**: Contiene el código fuente del compilador.
    - main.py: Punto de entrada principal de la aplicación.
    - PythonSubset.g4: Gramática formal del lenguaje (definición de reglas para ANTLR).
    - VisitanteSemantico.py: Módulo encargado del análisis semántico y validación de tipos.
    - Tabla_de_Simbolos.py: Gestión de variables y ámbitos durante la compilación.
    - CodigoIntermedio.py: Generación de la representación intermedia.
    - Optimizador.py: Módulo de optimización de código.
    - GeneradorMIPS.py: Traducción final a lenguaje ensamblador.
- **/lib**: Contiene las herramientas externas (antlr-4.13.2-complete.jar).
- **/examples**: Casos de prueba para validación.
    - /valid: Ejemplos de código correcto.
    - /invalid: Ejemplos con errores intencionales para probar el manejo de excepciones.
    - /expected: Resultados esperados de la compilación.
- **/tests**: Scripts de pruebas automatizadas (run_tests.py).

Guía de uso

Se debe tener instalado:

- Python 3.8+: Para ejecutar el compilador
- Java Runtime (JRE): Necesario solo para regenerar la gramática con ANTLR
- Simulador Mars (Mars4_5.jar): Para ejecutar el código ensamblador generado.

Se debe tener la estructura correcta del proyecto
proyecto/

lib/ Contiene antlr-4.13.1-complete.jar

src/ Código fuente (Lexer, Parser, Generador, etc.)

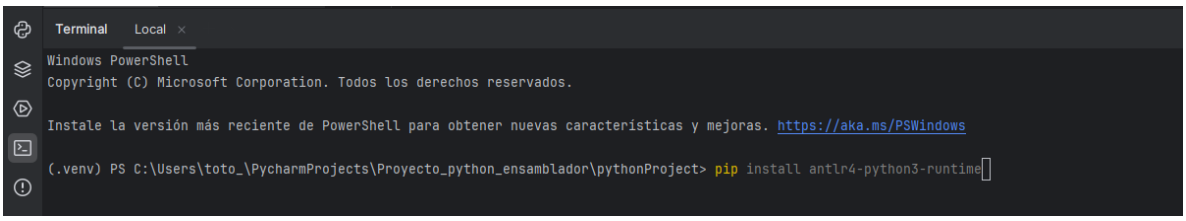
examples/ Ejemplos de prueba (.txt y .asm)

tests/ Scripts de pruebas automatizadas

Makefile Script de construcción

Se debe instalar el runtime de ANTLR

En la terminal del proyecto se ejecuta el siguiente comando: pip install antlr4-python3-runtime



Se debe traducir la gramática .g4 a código python, se ejecuta el siguiente comando en la terminal del proyecto:

java -jar lib/antlr-4.13.2-complete.jar -Dlanguage=Python3 -visitor -no-listener -o src src/PythonSubset.g4

Despues de ejecutar el subset.g4 tendremos todo lo que necesitamos para usar nuestro traductor de código.

Iremos al archivo run_tests.py y correremos el código aquí usara los casos validos e inválidos y mostrara en consola si fue exitoso y los casos de errores que preparamos mostrara el tipo de error.


```
=== PRUEBAS VÁLIDAS (7) ===
Testing 01_prueba_lexico_sintactico.txt     ... DEBUG: Definida variable 'funcion_vacia' de tipo 'FUNCION' en alcance nivel 1
DEBUG: Definida variable 'x' de tipo 'ENTERO' en alcance nivel 1
 PASS
Testing 02_prueba_alcance_semantico.txt     ... DEBUG: Definida variable 'a' de tipo 'ENTERO' en alcance nivel 1
DEBUG: Definida variable 'prueba' de tipo 'FUNCION' en alcance nivel 1
DEBUG: Definida variable 'x' de tipo 'PARAMETRO' en alcance nivel 2
DEBUG: Definida variable 'a' de tipo 'ENTERO' en alcance nivel 2
 PASS
Testing 03_prueba_GeneracionCodigoIntermedio.txt ... DEBUG: Definida variable 'a' de tipo 'ENTERO' en alcance nivel 1
DEBUG: Definida variable 'b' de tipo 'ENTERO' en alcance nivel 1
DEBUG: Definida variable 'c' de tipo 'ENTERO' en alcance nivel 1
DEBUG: Definida variable 'res1' de tipo 'ENTERO' en alcance nivel 1
DEBUG: Definida variable 'res2' de tipo 'ENTERO' en alcance nivel 1
 PASS
Testing 04_prueba_optimizacion.txt          ... DEBUG: Definida variable 'resultado' de tipo 'ENTERO' en alcance nivel 1
 PASS
```

```
Testing 05_prueba_factorial.txt             ... DEBUG: Definida variable 'numero' de tipo 'ENTERO' en alcance nivel 1
DEBUG: Definida variable 'res' de tipo 'ENTERO' en alcance nivel 1
DEBUG: Definida variable 'res' de tipo 'ENTERO' en alcance nivel 1
DEBUG: Definida variable 'numero' de tipo 'ENTERO' en alcance nivel 1
 PASS
Testing 06_prueba_sumatoria.txt             ... DEBUG: Definida variable 'total' de tipo 'ENTERO' en alcance nivel 1
DEBUG: Definida variable 'contador' de tipo 'ENTERO' en alcance nivel 1
DEBUG: Definida variable 'limite' de tipo 'ENTERO' en alcance nivel 1
DEBUG: Definida variable 'total' de tipo 'ENTERO' en alcance nivel 1
DEBUG: Definida variable 'contador' de tipo 'ENTERO' en alcance nivel 1
 PASS
Testing 07_prueba_strings.txt               ... DEBUG: Definida variable 'x' de tipo 'ENTERO' en alcance nivel 1
 PASS
```

Como se puede observar todos los casos validos funcionan y muestra que pasaron las pruebas.

```
=== PRUEBAS INVÁLIDAS (6) ===
Testing 01_error_lexico.txt                 ...  (Error detectado: Error Sintáctico en línea 4:6 - token recognition error at: '@')
Testing 02_error_sintactico.txt             ...  (Error detectado: Error Sintáctico en línea 5:9 - missing ':' at '{')
Testing 03_error_Semantico.txt              ... DEBUG: Definida variable 'x' de tipo 'ENTERO' en alcance nivel 1
 (Error detectado: Error Semántico en línea 5: La variable 'z' no ha sido definida.)
Testing 04_error_semanticoDeAlcance.txt     ... DEBUG: Definida variable 'calcular' de tipo 'FUNCION' en alcance nivel 1
DEBUG: Definida variable 'secreto' de tipo 'ENTERO' en alcance nivel 2
 (Error detectado: Error Semántico en línea 8: La variable 'secreto' no ha sido definida.)
Testing 05_error_semantico_parametros.txt   ... DEBUG: Definida variable 'suma' de tipo 'FUNCION' en alcance nivel 1
DEBUG: Definida variable 'a' de tipo 'PARAMETRO' en alcance nivel 2
DEBUG: Definida variable 'b' de tipo 'PARAMETRO' en alcance nivel 2
 (Error detectado: Error Semántico en línea 9: 'suma' es una función, no una variable. ¿Olvidaste los paréntesis ()?)
Testing 06_error_optimizador.txt            ... DEBUG: Definida variable 'x' de tipo 'ENTERO' en alcance nivel 1
 (Error detectado: float division by zero)
```

En las pruebas de casos inválidos muestra el error detectado.

Ejecucion Exitosa de ejemplos:

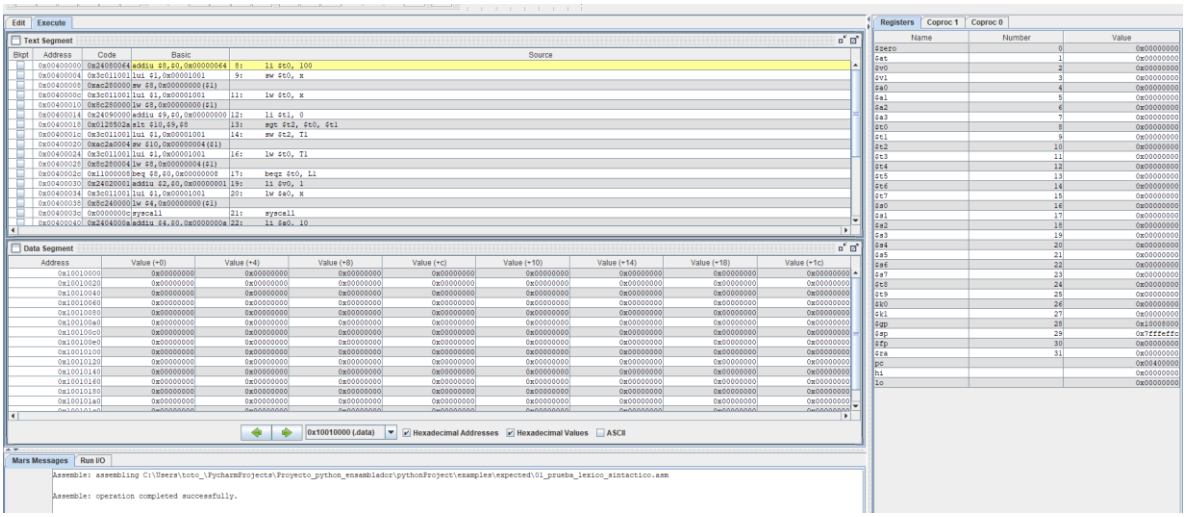
Para las pruebas del código traducido a ensamblador se ocupo Mars 4.5 que se encuentra en la carpeta lib que es un simulador que ejecuta código Ensamblador MIPS y se ejecutaron las salidas validas.

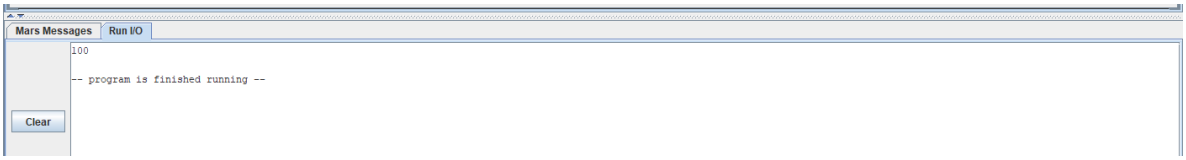
En el archivo de salidas esperadas muestra que debe aparecer en la consola en cada prueba al ejecutarse el código ensamblador.

Salidas_esperadas.txt

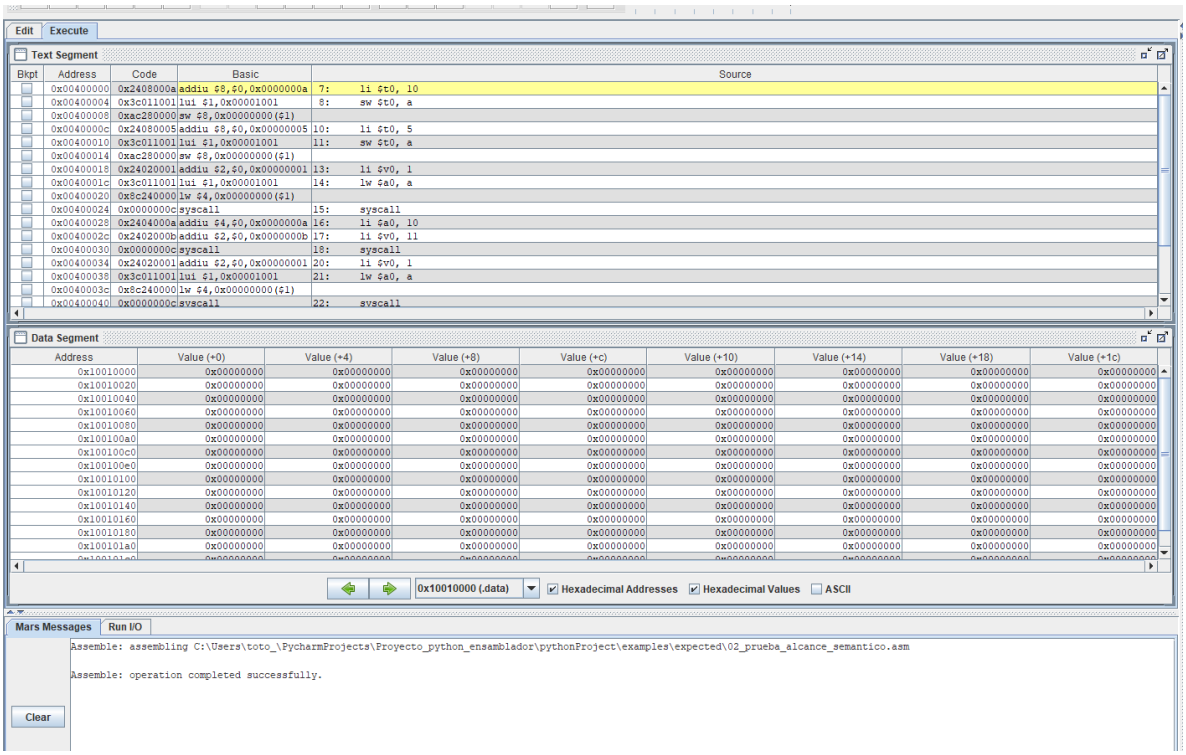
```
Prueba 1
100

prueba 2
10

Prueba 3
110
150

Prueba 4
60

Prueba 5
120

Prueba 6
10
55

Prueba 7
Inicio del programa
El numero es mayor a 5
10
Fin del programa
```

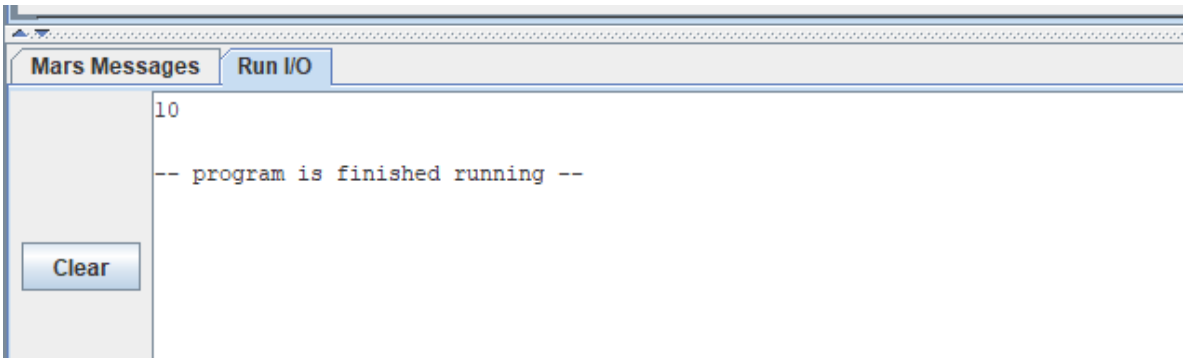
Prueba 1: Ensamblamos el código y todo funciona correctamente y nos da la salida esperada



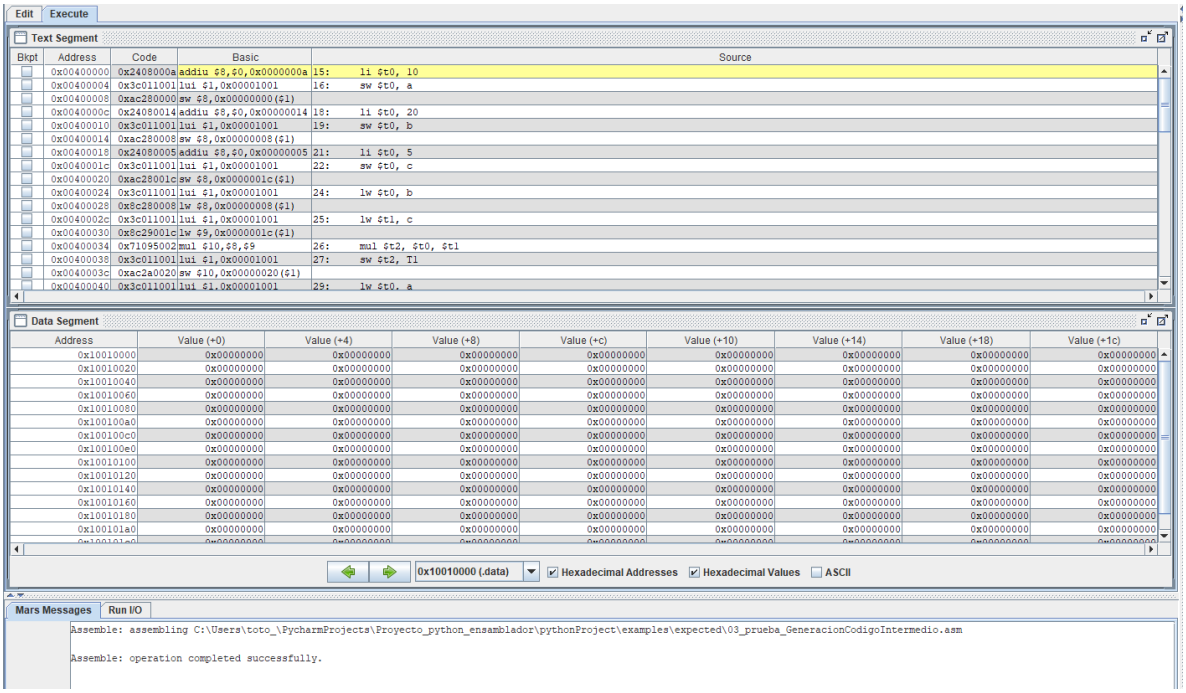


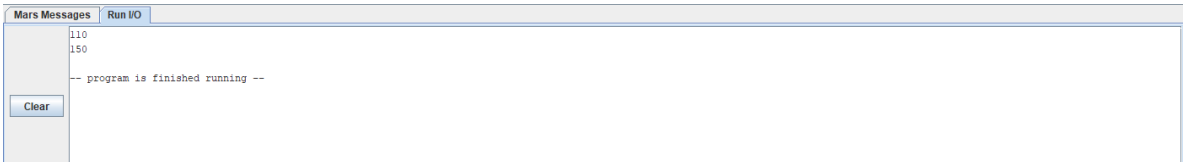
Prueba 2: Ensamblamos el código y funciona correctamente y ejecutamos



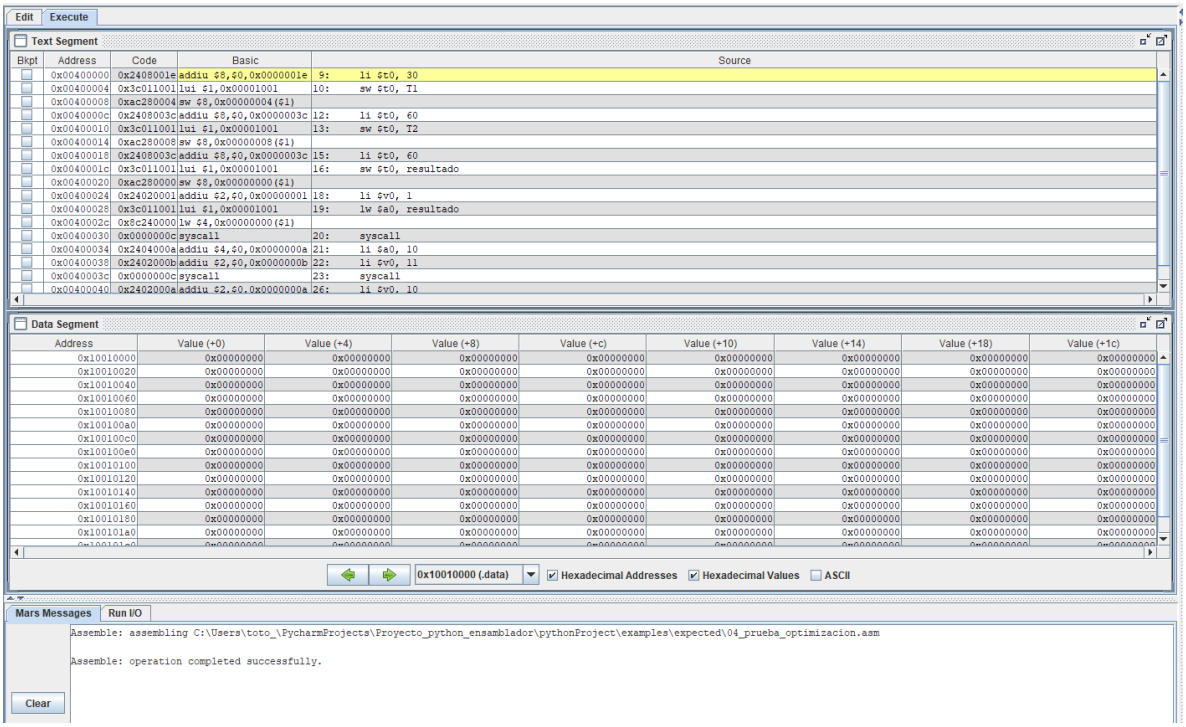


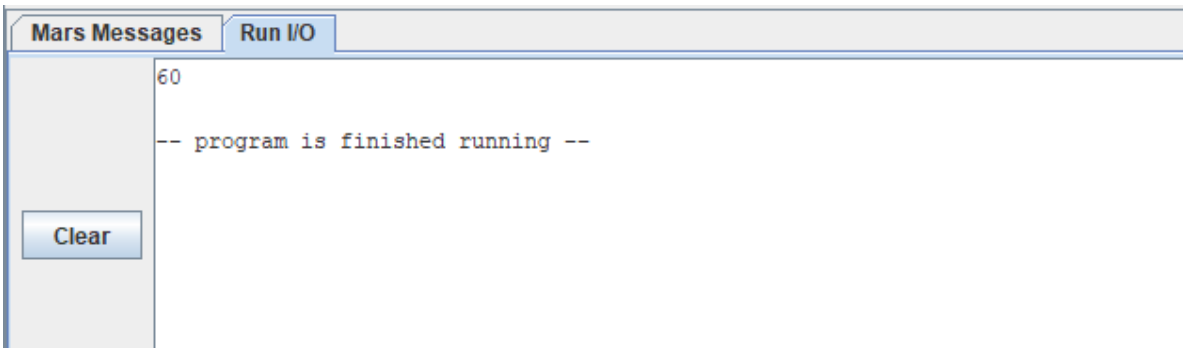
Prueba 3: Ensamblamos el código y funciona correctamente y ejecutamos



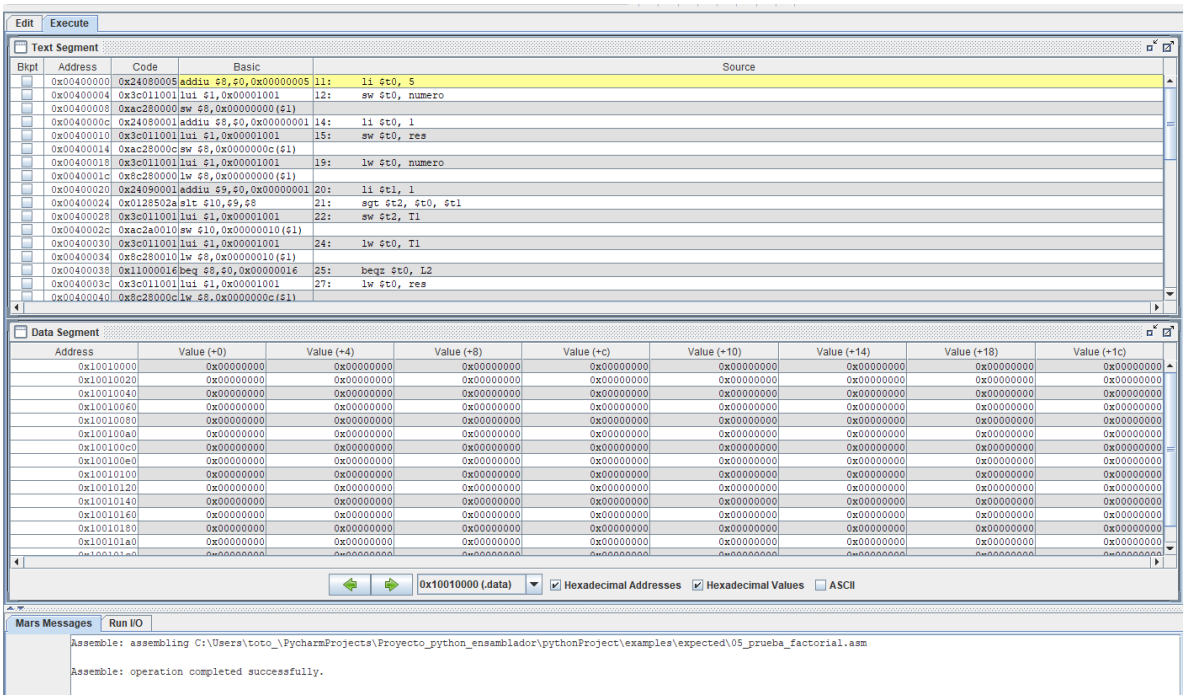


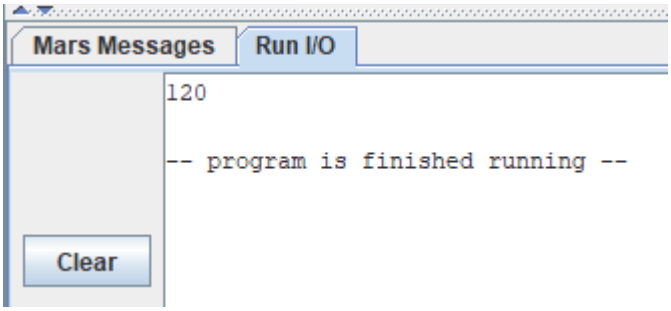
Prueba 4: Ensamblamos el código y funciona correctamente y ejecutamos



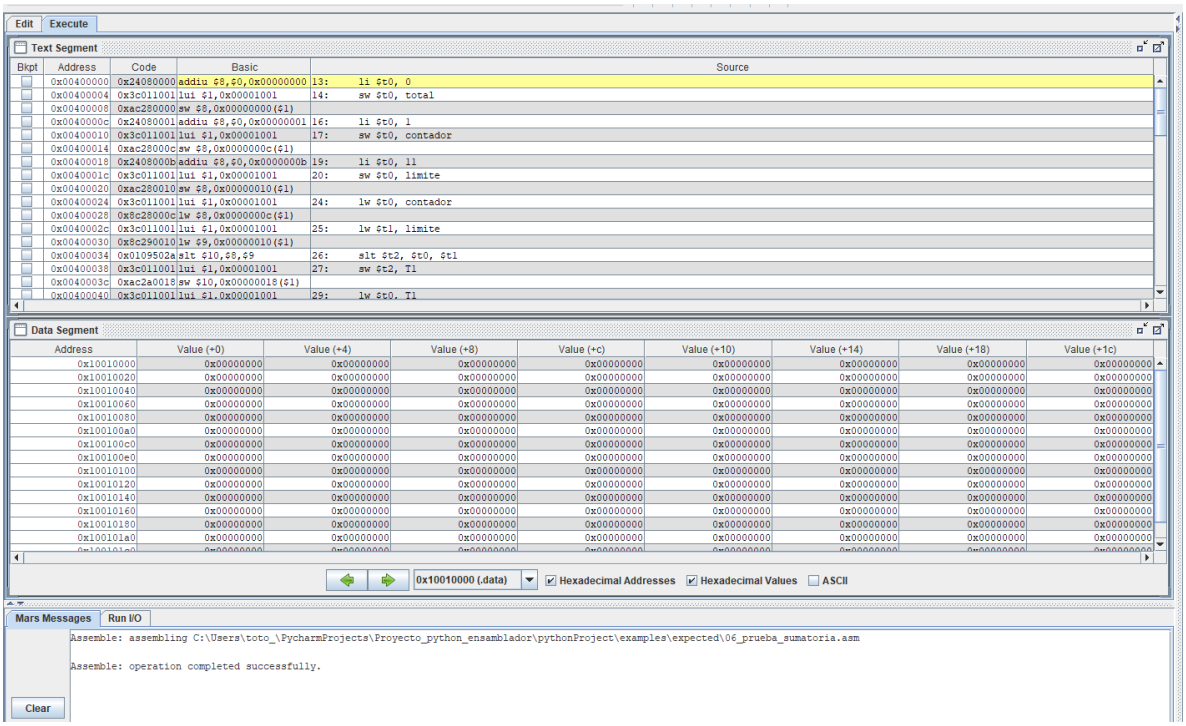


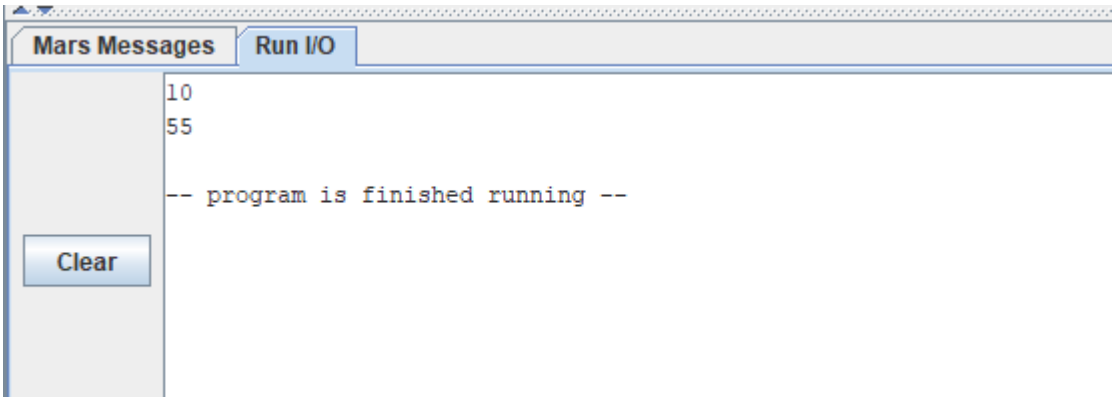
Prueba 5: Ensamblamos el código y funciona correctamente y ejecutamos



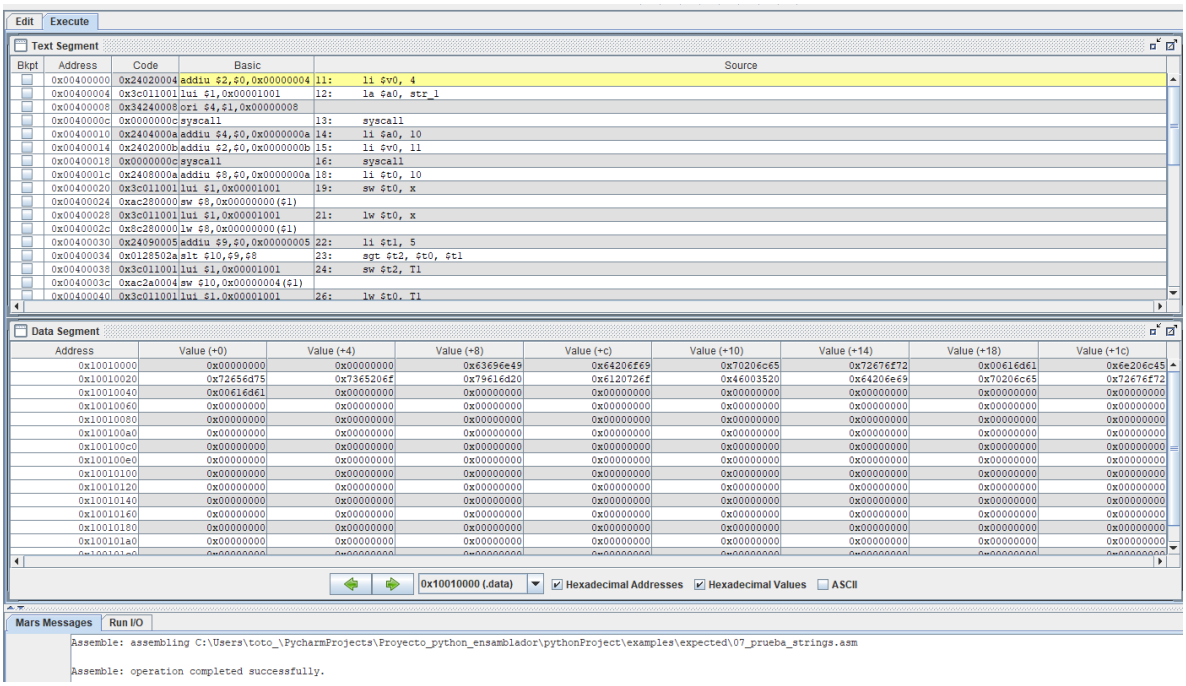


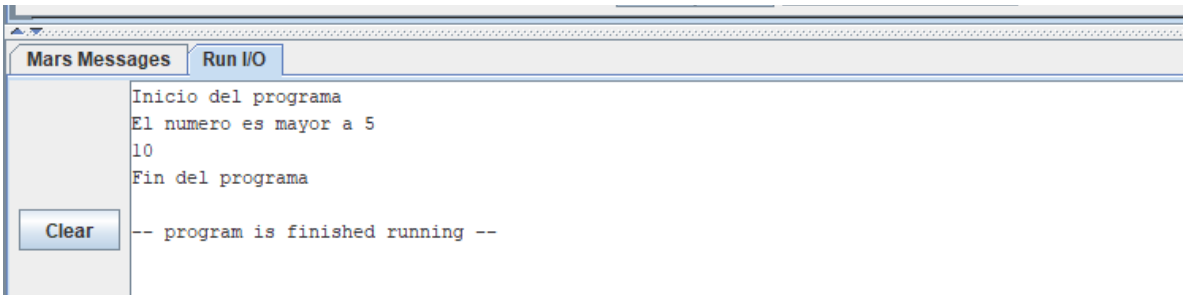
Prueba 6: Ensamblamos el código y funciona correctamente y ejecutamos





Prueba 7: Ensamblamos el código y funciona correctamente y ejecutamos





Como se puede observar todas las salidas esperadas fueron correctas cuando se ejecuto en el código assembler.

Documentación técnica Arquitectura del Compilador

1. Origen del Código y Herramientas Utilizadas

El código fuente de este proyecto se divide en dos categorías según su procedencia:

1. **Módulos Generados Automáticamente:** Son archivos creados por la herramienta ANTLR4 a partir de la gramática definida en el archivo PythonSubset.g4. Estos módulos (Lexer, Parser y Visitor) implementan la teoría de autómatas y gramáticas libres de contexto, proporcionando la base estructural para el análisis del lenguaje.

2. **Módulos de Implementación Lógica:** Son los archivos desarrollados manualmente (VisitanteSemantico, CodigoIntermedio, Optimizador, GeneradorMIPS, etc.). Estos

contienen la lógica específica del compilador para realizar validaciones, gestión de memoria, optimización y traducción a lenguaje ensamblador.

2. Descripción de Módulos por Fase

Fase 1: Análisis Léxico y Sintáctico (Frontend)

Esta fase convierte el código fuente en una estructura jerárquica comprensible para la máquina.

- **PythonSubsetLexer.py (Generado por ANTLR):**
    - **Función:** Es el analizador léxico. Su tarea es leer el flujo de caracteres de entrada y agruparlos en Tokens (identificadores, palabras reservadas como if o while, números, operadores, etc.). Filtra elementos irrelevantes como espacios en blanco y comentarios.
- **PythonSubsetParser.py (Generado por ANTLR):**
    - **Función:** Es el analizador sintáctico. Toma los tokens del lexer y verifica que sigan el orden gramatical definido. Construye el Árbol de Análisis Sintáctico (Parse Tree), una estructura de datos que representa la jerarquía gramatical del programa (ej. una sentencia if contiene una expresión condicional y un bloque de código).
- **PythonSubsetVisitor.py (Generado por ANTLR):**
    - **Función:** Provee la interfaz base del patrón de diseño Visitor. Define los métodos vacíos para recorrer el árbol sintáctico. Las clases lógicas del proyecto heredan de este archivo para implementar sus propias acciones al visitar cada nodo del árbol.

Fase 2: Análisis Semántico

Esta fase valida que el programa tenga sentido lógico, más allá de estar bien escrito gramaticalmente.

- **Tabla_de_Simbolos.py:**
    - **Función:** Estructura de datos que simula la memoria durante la compilación. Utiliza una pila de diccionarios para gestionar los alcances (scopes**)**. Permite registrar variables y funciones, y verificar su existencia respetando la visibilidad local y global.
- **VisitanteSemantico.py:**
    - **Función:** Recorre el árbol sintáctico para aplicar las reglas de tipo y contexto. Verifica que las variables usadas hayan sido declaradas previamente, gestiona la creación de nuevos entornos al entrar en funciones y asegura que no se realicen operaciones inválidas entre tipos incompatibles.

Fase 3: Generación de Código Intermedio (GCI)

Esta fase "aplana" la estructura jerárquica del árbol para facilitar la traducción y optimización.

- **CodigoIntermedio.py:**
    - **Función:** Transforma el árbol sintáctico en una lista lineal de instrucciones conocida como Código de Tres Direcciones (Cuádruplos). Descompone expresiones complejas (ej. x = a + b * c) en pasos simples utilizando variables temporales (T1, T2) y traduce las estructuras de control (if, while) en etiquetas (L1) y saltos (GOTO, IF_FALSE).

Fase 4: Optimización Independiente de la Máquina

Esta fase mejora la eficiencia del código antes de llegar a la arquitectura destino.

- **Optimizador.py:**
    - **Función:** Analiza la lista de cuádruplos para encontrar ineficiencias. Implementa técnicas como la Propagación de Constantes y el Plegado de Constantes (Constant Folding), pre-calculando operaciones aritméticas cuyos valores son conocidos en tiempo de compilación (ej. convertir x = 2 + 3 directamente en x = 5), reduciendo así el número de instrucciones finales.

Fase 5: Generación de Código Objeto (Backend)

Esta fase realiza la síntesis final hacia el lenguaje de la máquina destino.

- **GeneradorMIPS.py:**
    - **Función:** Traduce los cuádruplos optimizados a instrucciones del lenguaje ensamblador MIPS. Gestiona la asignación de memoria estática (sección .data), la traducción de operaciones aritméticas a instrucciones de registro (add, lw, sw), la implementación de saltos (bne, j) y el uso de llamadas al sistema (syscall) para entrada y salida de datos.

Orquestación

- **main.py:**
    - **Función:** Es el punto de entrada del compilador. Integra todas las fases anteriores en un pipeline secuencial: lee el código fuente, invoca al lexer/parser, ejecuta las validaciones semánticas, genera el código intermedio, lo optimiza y finalmente escribe el archivo de salida .asm.

Código fuente

PytonSubset.g4

```
grammar PythonSubset;
program
    : (functionDef | statement)* EOF
    ;
// Definición de función: def nombre(arg1, arg2) { codigo }
functionDef
    : 'def' ID '(' parameterList? ')' block
    ;
// Lista de parámetros: a, b, c
parameterList
    : ID (',' ID)*
    ;
statement
    : assignmentStmt
    | printStmt
    | ifStmt
    | whileStmt
    | returnStmt    // Nueva sentencia para funciones
    | functionCall  // Llamada a función como sentencia
    | block
    ;
assignmentStmt
    : ID '=' expression
    ;
printStmt
    : 'print' '(' expression ')'
    ;
ifStmt
    : 'if' expression ':' statement ('else' ':' statement)?
    ;
whileStmt
    : 'while' expression ':' statement
    ;
returnStmt
    : 'return' expression?
    ;
block
    : '{' statement* '}'
    ;
```

```
expression
    : expression ('*' | '/') expression      # MulDiv
    | expression ('+' | '-') expression      # AddSub
    | expression ('>' | '<' | '==') expression # Relational
    | ID '(' expressionList? ')'             # FunctionCallExpr
    | STRING                                 # StringLiteral
    | INT                                    # Number
    | ID                                     # Id
    | '(' expression ')'                     # Parens
    ;
// Lista de argumentos para llamar función: (1, x, 3)
expressionList
    : expression (',' expression)*
    ;
// Llamada a función aislada (para usarla en statement)
functionCall
    : ID '(' expressionList? ')'
    ;
// LEXER (Tokens)
DEF     : 'def';
RETURN  : 'return';
IF      : 'if';
ELSE    : 'else';
WHILE   : 'while';
PRINT   : 'print';
STRING  : '"' .*? '"' ;
ID      : [a-zA-Z_] [a-zA-Z0-9_]* ;
INT     : [0-9]+ ;
COMMENT : '#' ~[\r\n]* -> skip ;
WS      : [ \t\r\n]+ -> skip ;
// Operadores y símbolos
PLUS    : '+';
MINUS   : '-';
MUL     : '*';
DIV     : '/';
ASSIGN  : '=';
LPAREN  : '(';
RPAREN  : ')';
LBRACE  : '{';
RBRACE  : '}';
COLON   : ':';
COMMA   : ',';
GT      : '>';
```

```
LT      : '<';
EQ      : '==';
```

VisitanteSemántico.py

```
from PythonSubsetVisitor import PythonSubsetVisitor
from Tabla_de_Simbolos import TablaSimbolos

class VisitanteSemantico(PythonSubsetVisitor):
    def __init__(self):
        self.tabla_simbolos = TablaSimbolos()

    # --- Visitante del Programa Principal ---
    def visitProgram(self, ctx):
        self.visitChildren(ctx)

    # --- Visitante de Definición de Funciones ---
    def visitFunctionDef(self, ctx):
        nombre_funcion = ctx.ID().getText()
        self.tabla_simbolos.definir(nombre_funcion, 'FUNCION')

        self.tabla_simbolos.crear_alcance()

        if ctx.parameterList():
            for param in ctx.parameterList().ID():
                nombre_parametro = param.getText()
                self.tabla_simbolos.definir(nombre_parametro, 'PARAMETRO')

        self.visit(ctx.block())
        self.tabla_simbolos.salir_alcance()

    # --- Visitante de Asignaciones ---
    def visitAssignmentStmt(self, ctx):
        nombre_variable = ctx.ID().getText()

        # Primero visitamos la expresión para validar que las variables usadas existan
        self.visit(ctx.expression())

        self.tabla_simbolos.definir(nombre_variable, 'ENTERO')

    # --- Visitante de Identificadores en Expresiones ---
    def visitId(self, ctx):
        nombre_variable = ctx.getText()

        info_tipo = self.tabla_simbolos.buscar(nombre_variable)
```

```
        # 1. Verificamos existencia
        if info_tipo is None:
            linea = ctx.start.line
            raise Exception(f"Error Semántico en línea {linea}: La variable '{nombre_variable}'
no ha sido definida.")

        # 2. Verificamos que NO sea una función usada sin paréntesis
        if info_tipo == 'FUNCION':
            linea = ctx.start.line
            raise Exception(
                f"Error Semántico en línea {linea}: '{nombre_variable}' es una función, no una
variable. ¿Olvidaste los paréntesis ()?")

        return self.visitChildren(ctx)
```

CodigoIntermedio.py

```
from PythonSubsetVisitor import PythonSubsetVisitor


class Cuadruplo:
    def __init__(self, op, arg1, arg2, res):
        self.op = op
        self.arg1 = arg1
        self.arg2 = arg2
        self.res = res

    def __repr__(self):
        return f"({self.op}, {self.arg1}, {self.arg2}, {self.res})"


class VisitanteIntermedio(PythonSubsetVisitor):
    def __init__(self):
        self.cuadruplos = []
        self.contador_temporales = 0
        self.contador_etiquetas = 0

    def nuevo_temporal(self):
        self.contador_temporales += 1
        return f"T{self.contador_temporales}"

    def nueva_etiqueta(self):
        """Genera etiquetas para saltos (GOTO): L1, L2..."""
        self.contador_etiquetas += 1
```

```
    return f"L{self.contador_etiquetas}"

# --- MANEJO DE VALORES ---

def visitNumber(self, ctx):
    # Si vemos un número, simplemente retornamos su texto (ej: "5")
    return ctx.getText()

def visitId(self, ctx):
    # Si vemos una variable, retornamos su nombre (ej: "x")
    return ctx.getText()

def visitStringLiteral(self, ctx):
    # Devuelve el texto con comillas
    return ctx.getText()

# --- OPERACIONES ARITMÉTICAS ---

def visitMulDiv(self, ctx):
    # 1. Visitar hijos recursivamente para obtener sus direcciones
    izq = self.visit(ctx.expression(0))  # Lado izquierdo
    der = self.visit(ctx.expression(1))  # Lado derecho

    # 2. Crear temporal para guardar resultado
    temporal = self.nuevo_temporal()

    # 3. Obtener operador (* o /)
    op = ctx.getChild(1).getText()

    # 4. Generar Cuádruplo
    self.cuadruplos.append(Cuadruplo(op, izq, der, temporal))

    # 5. Retornar el nombre del temporal para que lo use el padre
    return temporal

def visitAddSub(self, ctx):
    izq = self.visit(ctx.expression(0))
    der = self.visit(ctx.expression(1))
    temporal = self.nuevo_temporal()
    op = ctx.getChild(1).getText()
    self.cuadruplos.append(Cuadruplo(op, izq, der, temporal))
    return temporal

# --- ASIGNACIÓN ---
```

```
def visitAssignmentStmt(self, ctx):
    nombre_variable = ctx.ID().getText()

    # Visitamos la expresión de la derecha para obtener el valor/temporal final
    valor = self.visit(ctx.expression())

    # Generamos cuádruplo de asignación
    self.cuadruplos.append(Cuadruplo('=', valor, None, nombre_variable))

# --- PRINT ---
def visitPrintStmt(self, ctx):
    valor = self.visit(ctx.expression())
    self.cuadruplos.append(Cuadruplo('PRINT', valor, None, None))

# --- PARÉNTESIS ---
def visitParens(self, ctx):
    # Simplemente visitamos la expresión de adentro y devolvemos su resultado.
    return self.visit(ctx.expression())

# --- COMPARACIONES (>, <, ==) ---
def visitRelational(self, ctx):
    izq = self.visit(ctx.expression(0))
    der = self.visit(ctx.expression(1))
    temporal = self.nuevo_temporal()
    op = ctx.getChild(1).getText()  # >, <, ==

    self.cuadruplos.append(Cuadruplo(op, izq, der, temporal))
    return temporal

# --- IF STATEMENT ---
def visitIfStmt(self, ctx):
    # if expresion : statement (else : statement)?

    # 1. Evaluar la condición
    condicion = self.visit(ctx.expression())

    # 2. Generar etiqueta para SALIR (o ir al else)
    etiqueta_falso = self.nueva_etiqueta()  # L1

    # 3. Generar salto
    self.cuadruplos.append(Cuadruplo('IF_FALSE', condicion, None, etiqueta_falso))

    # 4. Generar el código del bloque verdadero
```

```
        self.visit(ctx.statement(0))

        # Revisar si hay else
        if len(ctx.statement()) > 1:
            etiqueta_fin = self.nueva_etiqueta()

            # Al terminar el bloque True, saltamos al final para no ejecutar el Else
            self.cuadruplos.append(Cuadruplo('GOTO', None, None, etiqueta_fin))

            self.cuadruplos.append(Cuadruplo('LABEL', None, None, etiqueta_falso))

            self.visit(ctx.statement(1))

            # Ponemos la etiqueta del final
            self.cuadruplos.append(Cuadruplo('LABEL', None, None, etiqueta_fin))
        else:
            # Si no hay else, simplemente ponemos la etiqueta de salida aquí
            self.cuadruplos.append(Cuadruplo('LABEL', None, None, etiqueta_falso))

    # --- WHILE STATEMENT ---
    def visitWhileStmt(self, ctx):
        etiqueta_inicio = self.nueva_etiqueta()
        etiqueta_fin = self.nueva_etiqueta()

        # 1. Poner etiqueta de inicio
        self.cuadruplos.append(Cuadruplo('LABEL', None, None, etiqueta_inicio))

        condicion = self.visit(ctx.expression())

        # 3. Si es falso, salimos del ciclo
        self.cuadruplos.append(Cuadruplo('IF_FALSE', condicion, None, etiqueta_fin))

        # 4. Ejecutar el cuerpo del ciclo
        self.visit(ctx.statement())

        # 5. Volver al inicio automáticamente
        self.cuadruplos.append(Cuadruplo('GOTO', None, None, etiqueta_inicio))

        # 6. Etiqueta de salida
        self.cuadruplos.append(Cuadruplo('LABEL', None, None, etiqueta_fin))
```

Optimizador.py

```
class Optimizador:
    def __init__(self):
```

```
        self.constantes = {}

    def es_numero(self, s):
        try:
            float(s)
            return True
        except ValueError:
            return False

    def es_temporal(self, s):
        return s and s.startswith('T') and s[1:].isdigit()

    def optimizar(self, lista_cuadruplos):
        cuadruplos_optimizados = []

        for cuad in lista_cuadruplos:
            # 1. PROPAGACIÓN DE CONSTANTES
            if cuad.arg1 in self.constantes:
                cuad.arg1 = self.constantes[cuad.arg1]

            if cuad.arg2 in self.constantes:
                cuad.arg2 = self.constantes[cuad.arg2]

            # 2. PLEGADO DE CONSTANTES
            if cuad.op in ['+', '-', '*', '/'] and self.es_numero(cuad.arg1) and
self.es_numero(cuad.arg2):

                resultado = 0
                val1 = float(cuad.arg1)
                val2 = float(cuad.arg2)

                if cuad.op == '+':
                    resultado = val1 + val2
                elif cuad.op == '-':
                    resultado = val1 - val2
                elif cuad.op == '*':
                    resultado = val1 * val2
                elif cuad.op == '/':
                    resultado = val1 / val2

                if resultado.is_integer():
                    resultado = int(resultado)

                cuad.op = '='
```

```
                cuad.arg1 = str(resultado)
                cuad.arg2 = None

                if self.es_temporal(cuad.res):
                    self.constantes[cuad.res] = str(resultado)

            elif cuad.op == '=' and self.es_numero(cuad.arg1):
                if self.es_temporal(cuad.res):
                    self.constantes[cuad.res] = cuad.arg1

            cuadruplos_optimizados.append(cuad)

        return cuadruplos_optimizados
```

GeneradorMIPS.py

```
class GeneradorMIPS:
    def __init__(self):
        self.variables = set()
        self.cadenas = {}
        self.contador_cadenas = 0

    def es_numero(self, s):
        try:
            float(s)
            return True
        except ValueError:
            return False

    def es_cadena(self, s):
        return s and s.startswith("") and s.endswith("")

    def recolectar_variables(self, lista_cuadruplos):

        for cuad in lista_cuadruplos:
            # --- 1. RECOLECCIÓN DE VARIABLES NUMÉRICAS ---
            for arg in [cuad.arg1, cuad.arg2, cuad.res]:
                # Debe existir, NO ser número, NO ser string
                if arg and not self.es_numero(arg) and not self.es_cadena(arg):
                    if cuad.op in ['LABEL', 'GOTO', 'IF_FALSE'] and arg == cuad.res:
                        continue  # Es una etiqueta, ignorar

                    self.variables.add(arg)

            # --- 2. RECOLECCIÓN DE CADENAS (STRINGS) ---
```

```
        if self.es_cadena(cuad.arg1) and cuad.arg1 not in self.cadenas:
            self.contador_cadenas += 1
            etiqueta = f"str_{self.contador_cadenas}"
            self.cadenas[cuad.arg1] = etiqueta

    def generar(self, lista_cuadruplos, nombre_archivo="resultado.asm"):
        self.recolectar_variables(lista_cuadruplos)

        with open(nombre_archivo, 'w') as f:
            # --- SECCIÓN DE DATOS (.data) ---
            f.write(".data\n")

            # 1. Declarar Variables Numéricas (.word)
            for var in self.variables:
                f.write(f"    {var}: .word 0\n")

            # 2. Declarar Cadenas (.asciiz)
            for contenido, etiqueta in self.cadenas.items():
                f.write(f"    {etiqueta}: .asciiz {contenido}\n")

            # --- SECCIÓN DE CÓDIGO (.text) ---
            f.write("\n.text\n")
            f.write("main:\n")

            for cuad in lista_cuadruplos:
                f.write(f"    # {cuad}\n")
                op = cuad.op

                # --- OPERACIONES ARITMÉTICAS ---
                if op in ['+', '-', '*', '/', '<', '>', '==']:
                    # Cargar Operando 1
                    if self.es_numero(cuad.arg1):
                        f.write(f"    li $t0, {cuad.arg1}\n")
                    else:
                        f.write(f"    lw $t0, {cuad.arg1}\n")

                    # Cargar Operando 2
                    if self.es_numero(cuad.arg2):
                        f.write(f"    li $t1, {cuad.arg2}\n")
                    else:
                        f.write(f"    lw $t1, {cuad.arg2}\n")

                    # Operar
                    if op == '+':
```

```
            f.write("    add $t2, $t0, $t1\n")
        elif op == '-':
            f.write("    sub $t2, $t0, $t1\n")
        elif op == '*':
            f.write("    mul $t2, $t0, $t1\n")
        elif op == '/':
            f.write("    div $t0, $t1\n")
            f.write("    mflo $t2\n")
        elif op == '<':
            f.write("    slt $t2, $t0, $t1\n")
        elif op == '>':
            f.write("    sgt $t2, $t0, $t1\n")
        elif op == '==':
            f.write("    seq $t2, $t0, $t1\n")

        # Guardar resultado
        f.write(f"    sw $t2, {cuad.res}\n")

    # --- ASIGNACIÓN (=) ---
    elif op == '=':
        if self.es_numero(cuad.arg1):
            f.write(f"    li $t0, {cuad.arg1}\n")
        else:
            f.write(f"    lw $t0, {cuad.arg1}\n")
        f.write(f"    sw $t0, {cuad.res}\n")

    # --- PRINT (Manejo híbrido String/Int) ---
    elif op == 'PRINT':
        # CASO A: Imprimir Texto
        if self.es_cadena(cuad.arg1):
            f.write("    li $v0, 4\n")  # Syscall 4: Print String
            etiqueta = self.cadenas[cuad.arg1]  # Recuperar etiqueta (str_1)
            f.write(f"    la $a0, {etiqueta}\n")  # Cargar dirección

        # CASO B: Imprimir Número
        else:
            f.write("    li $v0, 1\n")
            if self.es_numero(cuad.arg1):
                f.write(f"    li $a0, {cuad.arg1}\n")
            else:
                f.write(f"    lw $a0, {cuad.arg1}\n")

        f.write("    syscall\n")
```

```
                f.write("    li $a0, 10\n")
                f.write("    li $v0, 11\n")
                f.write("    syscall\n")

            # --- SALTOS Y ETIQUETAS ---
            elif op == 'LABEL':
                f.write(f"{cuad.res}:\n")

            elif op == 'GOTO':
                f.write(f"    j {cuad.res}\n")

            elif op == 'IF_FALSE':
                if self.es_numero(cuad.arg1):
                    f.write(f"    li $t0, {cuad.arg1}\n")
                else:
                    f.write(f"    lw $t0, {cuad.arg1}\n")
                f.write(f"    beqz $t0, {cuad.res}\n")

        f.write("\n    # Fin del programa\n")
        f.write("    li $v0, 10\n")
        f.write("    syscall\n")
```

Tabla_de_Simbolos.py

```
class TablaSimbolos:
    def __init__(self):
        self.pila_alcances = [{}]

    def crear_alcance(self):
        """Crea un nuevo nivel de alcance"""
        self.pila_alcances.append({})

    def salir_alcance(self):
        """Destruye el alcance actual)"""
        self.pila_alcances.pop()

    def definir(self, nombre, tipo_info):
        """Guarda una variable en el alcance actual"""
        scope = self.pila_alcances[-1]
        scope[nombre] = tipo_info
        print(f"DEBUG: Definida variable '{nombre}' de tipo '{tipo_info}' en alcance nivel
{len(self.pila_alcances)}")

    def buscar(self, nombre):
        """Busca una variable. Empieza en el alcance actual y va subiendo hasta el global."""
```

```
        for scope in reversed(self.pila_alcances):
            if nombre in scope:
                return scope[nombre]
        return None # No encontrada
```

Tabla_de_Simbolos.py

```
class TablaSimbolos:
    def __init__(self):
        self.pila_alcances = [{}]

    def crear_alcance(self):
        """Crea un nuevo nivel de alcance"""
        self.pila_alcances.append({})

    def salir_alcance(self):
        """Destruye el alcance actual)"""
        self.pila_alcances.pop()

    def definir(self, nombre, tipo_info):
        """Guarda una variable en el alcance actual"""
        scope = self.pila_alcances[-1]
        scope[nombre] = tipo_info
        print(f"DEBUG: Definida variable '{nombre}' de tipo '{tipo_info}' en alcance nivel
{len(self.pila_alcances)}")

    def buscar(self, nombre):
        """Busca una variable. Empieza en el alcance actual y va subiendo hasta el global."""
        for scope in reversed(self.pila_alcances):
            if nombre in scope:
                return scope[nombre]
        return None # No encontrada
```