Seminario de Solución de problemas de Traductores de Lenguajes II

Centro Universitario de Ciencias Exactas en ingenierías

Universidad de Guadalajara

Maestro: LUIS FELIPE MUNOZ MENDOZA

Juan Antonio Pérez Juárez
Diego Andrés Hernandez Rodríguez
Juan Fernando Prieto Gómez

Práctica 4 Generador de código pseudo-ensamblador

Introducción:

Crear un programa que traduzca el árbol de análisis sintáctico/semántico que han construido en ejercicios anteriores a un código intermedio de destino y posteriormente a un ensamblador. En este caso, el código de destino será basado en instrucciones básicas, que incluyan instrucciones de transferencia, de control de flujo, aritméticas y lógicas.

El código debe ser funcional y seguir la lógica del programa original.

Debe cargarse un código fuente y poder pasar todos los analizadores, hasta llegar a un archivo ensamblador y si es posible hasta un ejecutable.

Desarrollo:

Antes de pasar al código, veamos algunas definiciones de los conceptos que veremos más adelante.

El **lenguaje ensamblador** o assembler (en inglés: assembler language y la abreviación asm) es un lenguaje de programación que se usa en los microprocesadores. Implementa una representación simbólica de los códigos de máquina binarios y otras constantes necesarias para programar una arquitectura de procesador y constituye la representación más directa del código máquina específico para cada arquitectura legible por un programador. Cada arquitectura de procesador tiene su propio lenguaje ensamblador que usualmente es definida por el fabricante de hardware, y está basada en los mnemónicos que simbolizan los pasos de procesamiento (las instrucciones), los registros del procesador, las posiciones de memoria y otras características del lenguaje. Un lenguaje ensamblador es por lo tanto específico de cierta arquitectura de computador física (o virtual). Esto está en contraste con la mayoría de los lenguajes de programación de alto nivel, que idealmente son portables.

Un programa utilitario llamado ensamblador es usado para traducir sentencias del lenguaje ensamblador al código de máquina del computador objetivo. El ensamblador realiza una traducción más o menos isomorfa (un mapeo de uno a uno) desde las sentencias mnemónicas a las instrucciones y datos de máquina. Esto está en contraste con los lenguajes de alto nivel, en los cuales una sola declaración generalmente da lugar a muchas instrucciones de máquina.

- El código escrito en lenguaje ensamblador posee una cierta dificultad de ser entendido ya que su estructura se acerca al lenguaje máquina, es decir, es un lenguaje de bajo nivel.
- El lenguaje ensamblador es difícilmente portable, es decir, un código escrito para un microprocesador, puede necesitar ser modificado, para poder ser usado en otra máquina distinta. Al cambiar a una máquina con arquitectura diferente, generalmente es necesario reescribirlo completamente.
- Los programas hechos por un programador experto en lenguaje ensamblador pueden ser más rápidos y consumir menos recursos del sistema (ej: memoria RAM) que el programa equivalente compilado desde un lenguaje de alto nivel. Al programar cuidadosamente en lenguaje ensamblador se pueden crear programas que se ejecutan más rápidamente y ocupan menos espacio que con lenguajes de alto nivel. Conforme han evolucionado tanto los procesadores como los compiladores de lenguajes de alto nivel, esta característica del lenguaje ensamblador se ha vuelto cada vez menos significativa. Es decir, un compilador moderno de lenguaje de alto nivel puede generar código casi tan eficiente como su equivalente en lenguaje ensamblador.
- Con el lenguaje ensamblador se tiene un control muy preciso de las tareas realizadas por un microprocesador por lo que se pueden crear segmentos de código difíciles y/o muy ineficientes de programar en un lenguaje de alto nivel, ya que, entre otras cosas, en el lenguaje ensamblador se dispone de instrucciones del CPU que generalmente no están disponibles en los lenguajes de alto nivel.

Ahora veamos otro concepto importante, el **código intermedio**:
El bytecode o código intermedio es un lenguaje intermedio más abstracto que el lenguaje máquina. Habitualmente, es tratado como un archivo binario que contiene un programa ejecutable similar a un módulo objeto, que es un archivo binario producido por el compilador cuyo contenido es el código objeto o código máquina .

El código intermedio recibe su nombre porque usualmente cada código de operación tiene una longitud de un byte, si bien la longitud del código de las instrucciones varía. Cada instrucción tiene un código de operación entre 0 y 255 seguido de parámetros tales como los registros o las direcciones de memoria. Esta sería la descripción de un caso típico, si bien la especificación del bytecode depende ampliamente del lenguaje.

Como código intermedio, se trata de una forma de salida utilizada por los implementadores de lenguajes para reducir la dependencia respecto del hardware específico y facilitar la interpretación. Menos frecuentemente se utiliza el bytecode como código intermedio en un compilador. Algunos sistemas, llamados traductores dinámicos o compiladores justo a tiempo, traducen el bytecode a código máquina inmediatamente antes de su ejecución para mejorar la velocidad de ejecución.

Ahora pasemos a lo importante.

Código:

```python
import ply.lex as lex
import ply.yacc as yacc
import sys

# Tabla de símbolos para almacenar variables y sus tipos
tabla_simbolos = {}

# Contador para variables temporales
temp_counter = 0
label_counter = 0

# Lista para almacenar código intermedio
codigo_intermedio = []

def new_temp():
    global temp_counter
    temp = f"t{temp_counter}"
    temp_counter += 1
    return temp

def new_label():
    global label_counter
    label = f"L{label_counter}"
    label_counter += 1
    return label

# Tabla de símbolos para almacenar variables y sus tipos
tabla_simbolos = {}

def generar_codigo_intermedio(nodo):
    global temp_counter

    if isinstance(nodo, (int, float)):
        return str(nodo)

    if isinstance(nodo, str) and not nodo.startswith('t'):
        return nodo

    if isinstance(nodo, tuple):
        if len(nodo) == 3:  # Operación binaria
            op = nodo[1]
            izq = generar_codigo_intermedio(nodo[0])
            der = generar_codigo_intermedio(nodo[2])

            temp = f"t{temp_counter}"
```

```
            temp_counter += 1

            if op == '=':  # Asignación
                codigo_intermedio.append(f"ASSIGN {izq} = {der}")
                return izq
            else:  # Otras operaciones
                codigo_intermedio.append(f"{temp} = {izq} {op} {der}")
                return temp

        elif len(nodo) == 2:  # Declaración
            tipo, var = nodo
            codigo_intermedio.append(f"DECLARE {tipo} {var}")
            return var

    return str(nodo)

# Definición del analizador léxico
tokens = (
    'NUMERO',
    'ID',
    'SUMA',
    'RESTA',
    'MULT',
    'DIV',
    'LPAREN',
    'RPAREN',
    'IGUAL',
    'PUNTOCOMA',
    'INT',
    'FLOAT'
)

# Reglas para tokens simples
t_SUMA = r'\+'
t_RESTA = r'-'
t_MULT = r'\*'
t_DIV = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_IGUAL = r'='
t_PUNTOCOMA = r';'

# Palabras reservadas
def t_INT(t):
    r'int'
    return t

def t_FLOAT(t):
```

```
    r'float'
    return t

# Regla para identificadores
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    return t

# Regla para números (enteros y flotantes)
def t_NUMERO(t):
    r'\d+(\.\d+)?'
    if '.' in t.value:
        t.value = float(t.value)
    else:
        t.value = int(t.value)
    return t

# Ignorar espacios y tabulaciones
t_ignore = ' \t'

# Nueva línea
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# Error handling
def t_error(t):
    print(f"Carácter ilegal '{t.value[0]}' en la línea {t.lexer.lineno}")
    t.lexer.skip(1)

# Construcción del lexer
lexer = lex.lex()

# Definición de la gramática
def p_programa(p):
    '''programa : declaracion
                | asignacion
                | expresion'''
    p[0] = p[1]

def p_declaracion(p):
    '''declaracion : INT ID PUNTOCOMA
                   | FLOAT ID PUNTOCOMA'''
    if p[2] in tabla_simbolos:
        print(f"Error semántico: Variable '{p[2]}' ya declarada previamente.")
        p[0] = None
        return
```

```
    tipo = 'int' if p[1] == 'int' else 'float'
    tabla_simbolos[p[2]] = {'tipo': tipo, 'valor': None}
    codigo_intermedio.append(f"DECLARE {tipo} {p[2]}")
    print(f"Variable '{p[2]}' declarada como {tipo}")
    p[0] = None

def p_asignacion(p):
    '''asignacion : ID IGUAL expresion PUNTOCOMA'''
    if p[1] not in tabla_simbolos:
        print(f"Error semántico: Variable '{p[1]}' no declarada.")
        p[0] = None
        return

    valor = p[3]
    tipo_var = tabla_simbolos[p[1]]['tipo']

    if tipo_var == 'int' and isinstance(valor, float):
        print(f"Advertencia: Asignando valor flotante a variable entera '{p[1]}'. Se
truncará el valor.")
        valor = int(valor)

    tabla_simbolos[p[1]]['valor'] = valor
    codigo_intermedio.append(f"ASSIGN {p[1]} = {valor}")
    print(f"Asignación: {p[1]} = {valor}")
    p[0] = valor

def p_expresion_binaria(p):
    '''expresion : expresion SUMA expresion
                 | expresion RESTA expresion
                 | expresion MULT expresion
                 | expresion DIV expresion'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
        temp = new_temp()
        codigo_intermedio.append(f"{temp} = {p[1]} + {p[3]}")
        p[0] = temp
    elif p[2] == '-':
        p[0] = p[1] - p[3]
        temp = new_temp()
        codigo_intermedio.append(f"{temp} = {p[1]} - {p[3]}")
        p[0] = temp
    elif p[2] == '*':
        p[0] = p[1] * p[3]
        temp = new_temp()
        codigo_intermedio.append(f"{temp} = {p[1]} * {p[3]}")
        p[0] = temp
    elif p[2] == '/':
        if p[3] == 0:
```

```
            print("Error: División por cero")
            p[0] = None
            return
        p[0] = p[1] / p[3]
        temp = new_temp()
        codigo_intermedio.append(f"{temp} = {p[1]} / {p[3]}")
        p[0] = temp

def p_expresion_numero(p):
    'expresion : NUMERO'
    p[0] = p[1]

def p_expresion_id(p):
    'expresion : ID'
    if p[1] not in tabla_simbolos:
        print(f"Error semántico: Variable '{p[1]}' no declarada.")
        p[0] = 0
        return

    if tabla_simbolos[p[1]]['valor'] is None:
        print(f"Error semántico: Variable '{p[1]}' no inicializada.")
        p[0] = 0
    else:
        p[0] = tabla_simbolos[p[1]]['valor']

def p_expresion_parentesis(p):
    'expresion : LPAREN expresion RPAREN'
    p[0] = p[2]

def p_expresion_numero(p):
    'expresion : NUMERO'
    p[0] = p[1]

def p_expresion_id(p):
    'expresion : ID'
    # Verificar si la variable existe en la tabla de símbolos
    if p[1] not in tabla_simbolos:
        print(f"Error semántico: Variable '{p[1]}' no declarada.")
        p[0] = 0  # Valor por defecto para continuar la evaluación
        return

    # Verificar si la variable tiene valor asignado
    if tabla_simbolos[p[1]]['valor'] is None:
        print(f"Error semántico: Variable '{p[1]}' no inicializada.")
        p[0] = 0  # Valor por defecto para continuar la evaluación
    else:
        p[0] = tabla_simbolos[p[1]]['valor']
```

```
def p_error(p):
    if p:
        print(f"Error de sintaxis en '{p.value}', línea {p.lineno}")
    else:
        print("Error de sintaxis al final de la entrada")

# Construir el parser
parser = yacc.yacc()

def mostrar_tabla_simbolos():
    print("\n=== TABLA DE SÍMBOLOS ===")
    print("Variable\tTipo\tValor")
    print("-" * 30)
    for var, info in tabla_simbolos.items():
        print(f"{var}\t\t{info['tipo']}\t{info['valor']}")
    print("=" * 30)

def evaluar_ast(expresion):
    """Evalúa una expresión y muestra los resultados"""
    global codigo_intermedio
    codigo_intermedio = []  # Limpiar código intermedio anterior

    try:
        result = parser.parse(expresion)

        print("\n=== CÓDIGO INTERMEDIO ===")
        for i, instr in enumerate(codigo_intermedio):
            print(f"{i}: {instr}")

        print("\n=== CÓDIGO ENSAMBLADOR ===")
        print(generar_asm())

        print("\nValidación semántica completada exitosamente.")
        mostrar_tabla_simbolos()
        return result
    except Exception as e:
        print(f"Error durante el análisis: {e}")
        return None


# Función para generar código ensamblador
def generar_asm():
    asm_code = [
        "section .data",
        "; Variables declaradas"
    ]

    # Declarar variables
```

```
    for var, info in tabla_simbolos.items():
        if info['tipo'] == 'int':
            asm_code.append(f"{var} dd 0")
        else:
            asm_code.append(f"{var} dq 0.0")

    asm_code.extend([
        "\nsection .text",
        "global _start",
        "_start:"
    ])

    # Traducir código intermedio a ensamblador
    for instr in codigo_intermedio:
        if instr.startswith("DECLARE"):
            continue  # Las declaraciones ya se manejaron en .data
        elif "=" in instr:
            partes = instr.split("=")
            destino = partes[0].strip()
            expresion = partes[1].strip()

            if "+" in expresion:
                op1, op2 = expresion.split("+")
                asm_code.extend([
                    f"    mov eax, [{op1.strip()}]",
                    f"    add eax, [{op2.strip()}]",
                    f"    mov [{destino}], eax"
                ])
            elif "-" in expresion:
                op1, op2 = expresion.split("-")
                asm_code.extend([
                    f"    mov eax, [{op1.strip()}]",
                    f"    sub eax, [{op2.strip()}]",
                    f"    mov [{destino}], eax"
                ])

    # Agregar código de salida
    asm_code.extend([
        "\n    ; Salir del programa",
        "    mov eax, 1",
        "    xor ebx, ebx",
        "    int 0x80"
    ])

    return "\n".join(asm_code)

def evaluar_ast(expresion):
    """Evalúa una expresión y muestra los resultados"""
```

```
    global codigo_intermedio
    codigo_intermedio = []  # Limpiar código intermedio anterior

    try:
        result = parser.parse(expresion)
        if result is not None:
            print(f"\nResultado de evaluación: {result}")

        print("\n=== CÓDIGO INTERMEDIO ===")
        for i, instr in enumerate(codigo_intermedio):
            print(f"{i}: {instr}")

        print("\n=== CÓDIGO ENSAMBLADOR ===")
        print(generar_asm())

        print("\nValidación semántica completada exitosamente.")
        mostrar_tabla_simbolos()
        return result
    except Exception as e:
        print(f"Error durante el análisis: {e}")
        return None


# Función principal para probar el analizador
def main():
    print("Analizador Semántico - Evaluador de Expresiones")
    print("Ingrese 'salir' para terminar")

    while True:
        try:
            expresion = input(">> ")
            if expresion.lower() == 'salir':
                break
            evaluar_ast(expresion)
        except EOFError:
            break

if __name__ == "__main__":
    main()
```

Lógica:

Este programa de desprende del programa anterior que es un analizador semántico, por lo que sigue la misma lógica, además que he podido reutilizar la mayor parte del codigo, aqui lo que ha cambiado y lo que se explicará es la generación de código intermedio y la generación de codigo ensamblador.

Python

```python
# Contador para variables temporales y etiquetas
temp_counter = 0
label_counter = 0

def new_temp():
    # Genera nombres únicos para variables temporales (t0, t1, t2...)
    global temp_counter
    temp = f"t{temp_counter}"
    temp_counter += 1
    return temp

def new_label():
    # Genera nombres únicos para etiquetas (L0, L1, L2...)
    global label_counter
    label = f"L{label_counter}"
    label_counter += 1
    return label
```

Estas funciones crean identificadores únicos para variables temporales y etiquetas de salto, esenciales para la generación de código intermedio.

Python

```python
def generar_codigo_intermedio(nodo):
    global temp_counter

    if isinstance(nodo, (int, float)):
        # Si es un valor literal, lo devuelve directamente
        return str(nodo)

    if isinstance(nodo, str) and not nodo.startswith('t'):
        # Si es un identificador (no temporal), lo devuelve
        return nodo

    if isinstance(nodo, tuple):
        if len(nodo) == 3:  # Operación binaria
            op = nodo[1]
            izq = generar_codigo_intermedio(nodo[0])
            der = generar_codigo_intermedio(nodo[2])

            temp = f"t{temp_counter}"
            temp_counter += 1

            if op == '=':  # Asignación
                codigo_intermedio.append(f"ASSIGN {izq} = {der}")
                return izq
```

```
            else:  # Otras operaciones
                codigo_intermedio.append(f"{temp} = {izq} {op} {der}")
                return temp
```

Esta función recorre recursivamente el AST y genera instrucciones de código intermedio según el tipo de nodo. Para operaciones binarias, procesa ambos operandos y crea una variable temporal para almacenar el resultado.

Python

```python
def p_expresion_binaria(p):
    '''expresion : expresion SUMA expresion
                 | expresion RESTA expresion
                 | expresion MULT expresion
                 | expresion DIV expresion'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
        temp = new_temp()
        codigo_intermedio.append(f"{temp} = {p[1]} + {p[3]}")
        p[0] = temp
    elif p[2] == '-':
        p[0] = p[1] - p[3]
        temp = new_temp()
        codigo_intermedio.append(f"{temp} = {p[1]} - {p[3]}")
        p[0] = temp
```

Esta función del parser maneja operaciones binarias, generando tanto el valor calculado como el código intermedio correspondiente para cada operación.

Python

```python
def p_asignacion(p):
    '''asignacion : ID IGUAL expresion PUNTOCOMA'''
    if p[1] not in tabla_simbolos:
        print(f"Error semántico: Variable '{p[1]}' no declarada.")
        p[0] = None
        return

    valor = p[3]
    tipo_var = tabla_simbolos[p[1]]['tipo']

    if tipo_var == 'int' and isinstance(valor, float):
        print(f"Advertencia: Asignando valor flotante a variable entera '{p[1]}'. Se
truncará el valor.")
```

```
        valor = int(valor)

    tabla_simbolos[p[1]]['valor'] = valor
    codigo_intermedio.append(f"ASSIGN {p[1]} = {valor}")
    print(f"Asignación: {p[1]} = {valor}")
    p[0] = valor
```

Verifica que la variable esté declarada antes de la asignación, comprueba compatibilidad de tipos, y genera el código intermedio para la asignación.

Python

```python
def generar_asm():
    asm_code = [
        "section .data",
        "; Variables declaradas"
    ]

    # Declarar variables en la sección de datos
    for var, info in tabla_simbolos.items():
        if info['tipo'] == 'int':
            asm_code.append(f"{var} dd 0")
        else:
            asm_code.append(f"{var} dq 0.0")

    asm_code.extend([
        "\nsection .text",
        "global _start",
        "_start:"
    ])

    # Traducir código intermedio a ensamblador
    for instr in codigo_intermedio:
        if instr.startswith("DECLARE"):
            continue  # Las declaraciones ya se manejaron en .data
        elif "=" in instr:
            partes = instr.split("=")
            destino = partes[0].strip()
            expresion = partes[1].strip()

            if "+" in expresion:
                op1, op2 = expresion.split("+")
                asm_code.extend([
                    f"    mov eax, [{op1.strip()}]",
                    f"    add eax, [{op2.strip()}]",
                    f"    mov [{destino}], eax"
```

```
                ])
            elif "-" in expresion:
                op1, op2 = expresion.split("-")
                asm_code.extend([
                    f"    mov eax, [{op1.strip()}]",
                    f"    sub eax, [{op2.strip()}]",
                    f"    mov [{destino}], eax"
                ])

    # Agregar código de salida
    asm_code.extend([
        "\n    ; Salir del programa",
        "    mov eax, 1",
        "    xor ebx, ebx",
        "    int 0x80"
    ])

    return "\n".join(asm_code)
```

Genera el código ensamblador en dos secciones principales. La sección .data declara las variables, y la sección .text contiene las instrucciones traducidas del código intermedio, utilizando un enfoque de análisis de patrones.

Python

```python
# Tabla de símbolos para almacenar variables y sus tipos
tabla_simbolos = {}

def p_declaracion(p):
    '''declaracion : INT ID PUNTOCOMA
                   | FLOAT ID PUNTOCOMA'''
    if p[2] in tabla_simbolos:
        print(f"Error semántico: Variable '{p[2]}' ya declarada previamente.")
        p[0] = None
        return

    tipo = 'int' if p[1] == 'int' else 'float'
    tabla_simbolos[p[2]] = {'tipo': tipo, 'valor': None}
    codigo_intermedio.append(f"DECLARE {tipo} {p[2]}")
    print(f"Variable '{p[2]}' declarada como {tipo}")
    p[0] = None
```

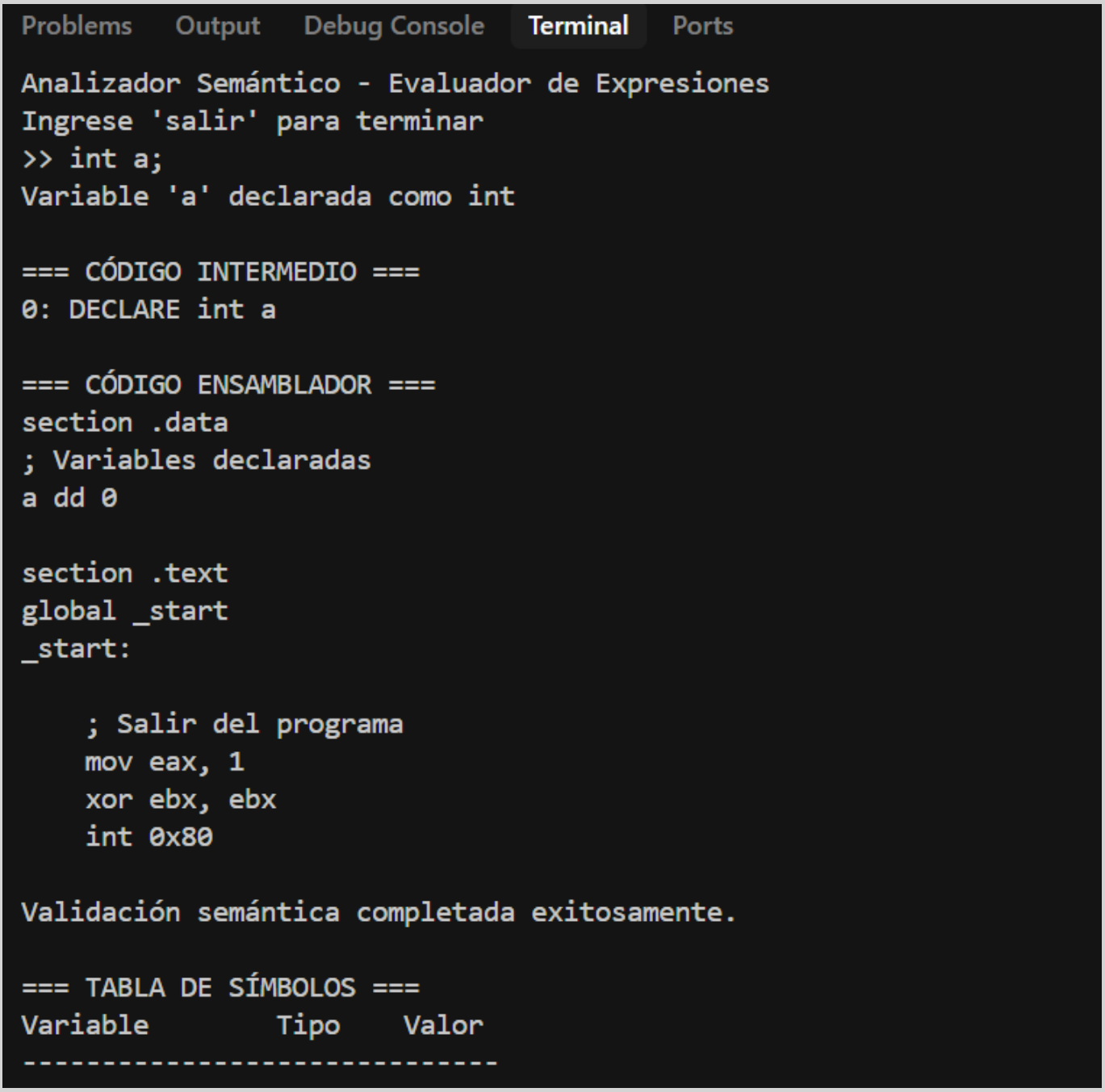
Mantiene la tabla de símbolos para registrar variables, sus tipos y valores, verificando que no haya redeclaraciones y generando código intermedio para las declaraciones.

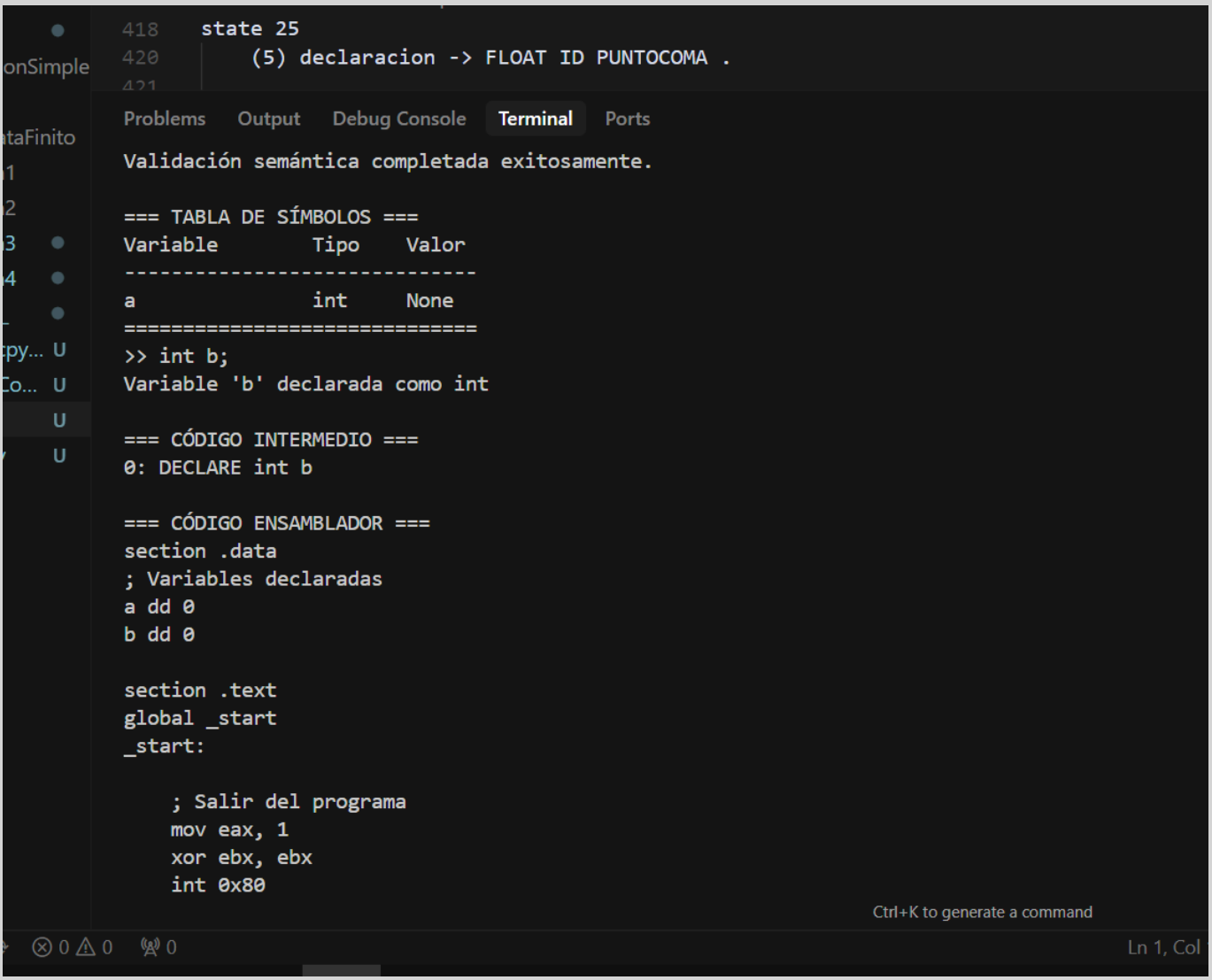
Python

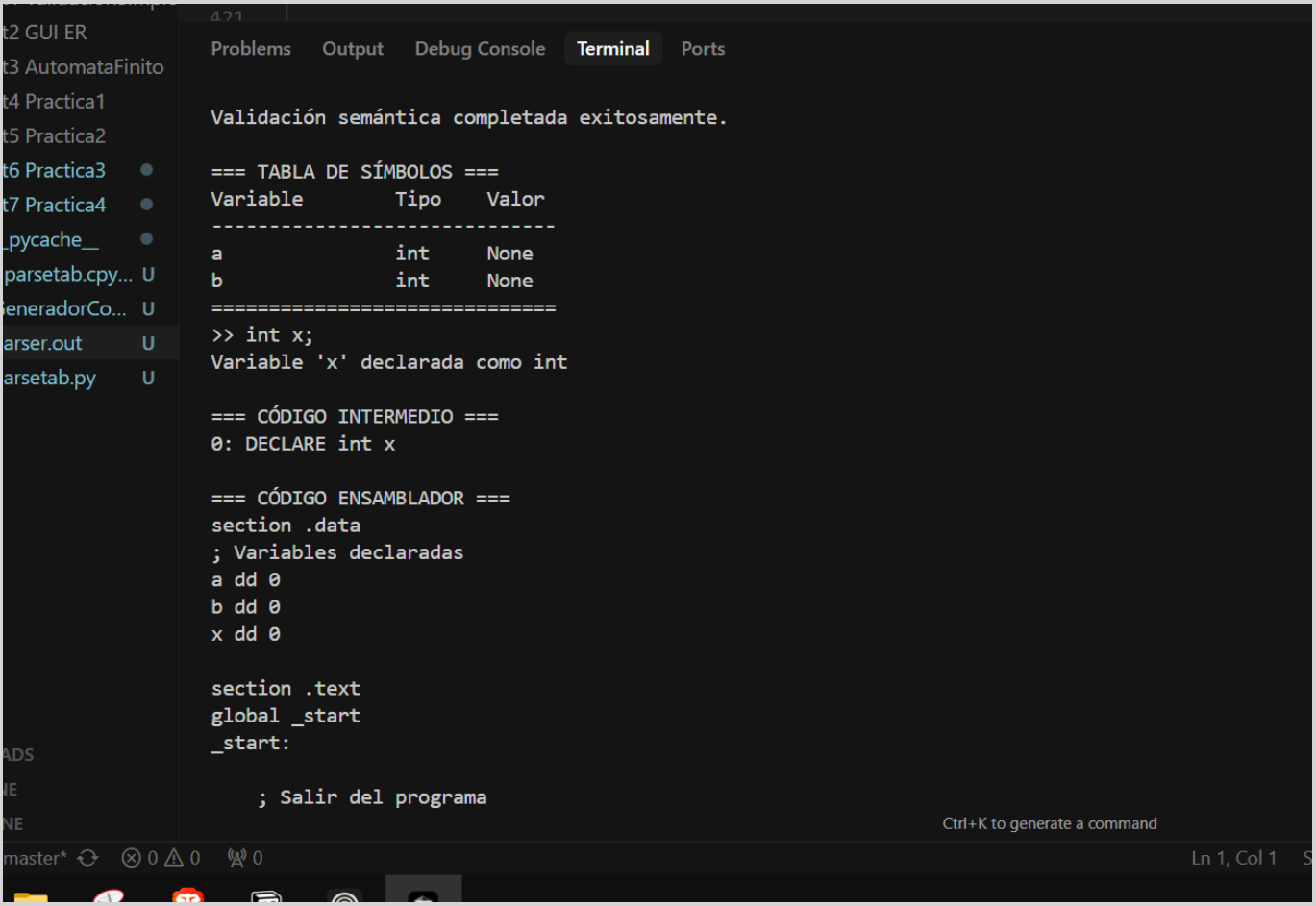
```python
def evaluar_ast(expresion):
    """Evalúa una expresión y muestra los resultados"""
    global codigo_intermedio
    codigo_intermedio = []  # Limpiar código intermedio anterior

    try:
        result = parser.parse(expresion)
        if result is not None:
            print(f"\nResultado de evaluación: {result}")

        print("\n=== CÓDIGO INTERMEDIO ===")
        for i, instr in enumerate(codigo_intermedio):
            print(f"{i}: {instr}")

        print("\n=== CÓDIGO ENSAMBLADOR ===")
        print(generar_asm())

        print("\nValidación semántica completada exitosamente.")
        mostrar_tabla_simbolos()
        return result
    except Exception as e:
        print(f"Error durante el análisis: {e}")
        return None
```

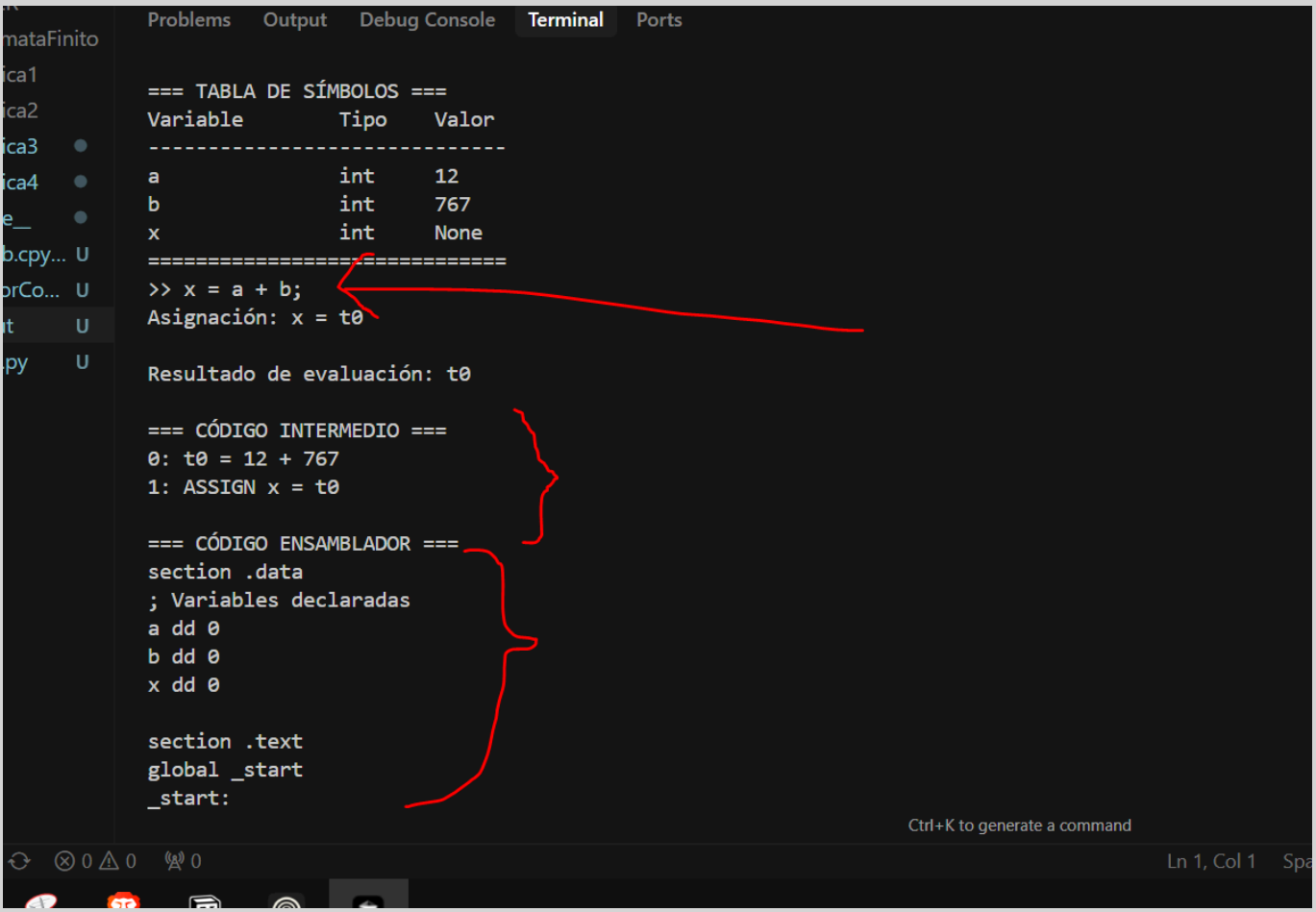
Coordina todo el proceso de análisis y generación de código, limpiando el estado previo, analizando la expresión, y mostrando tanto el código intermedio como el ensamblador generado.

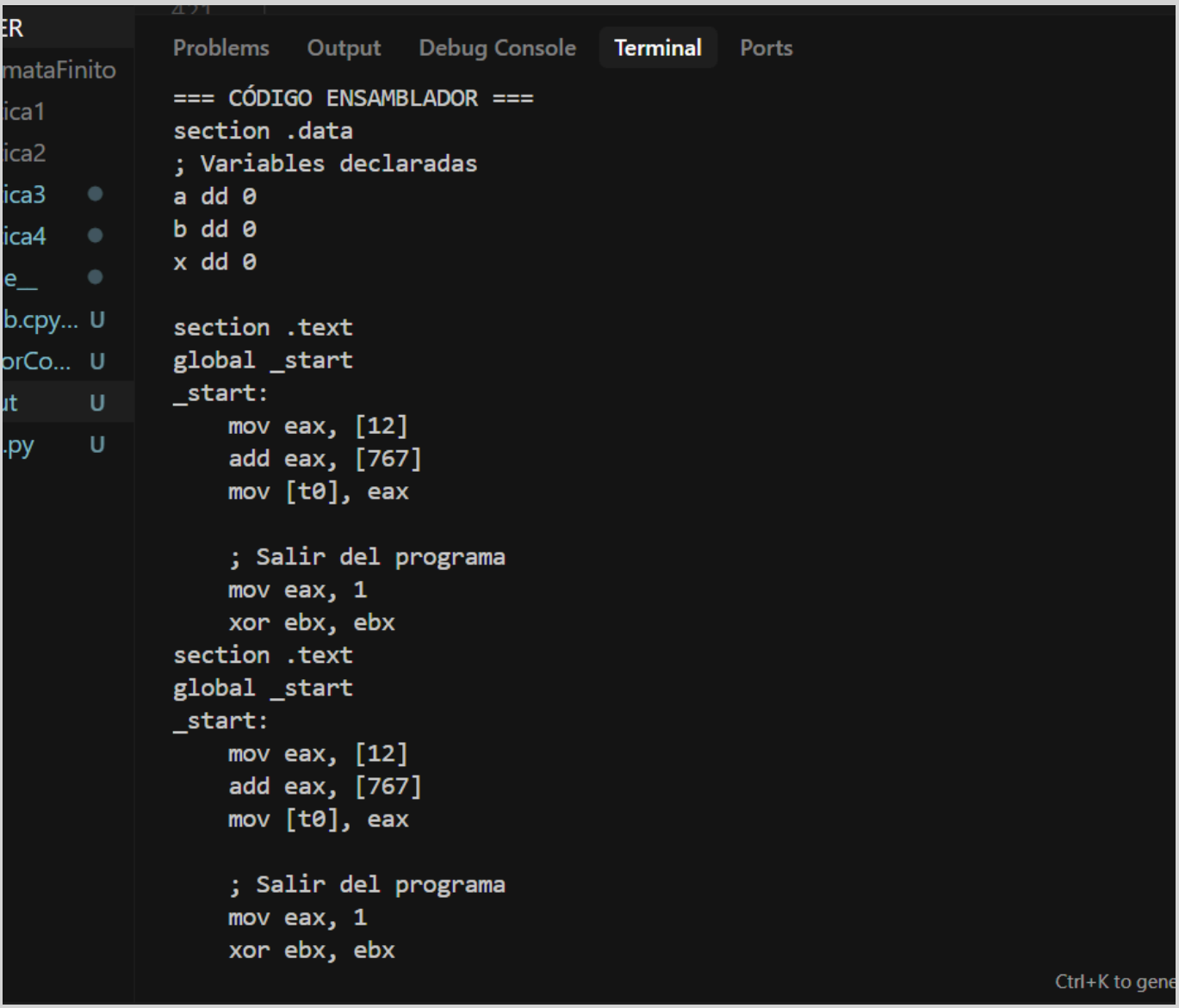
Capturas de pantalla:

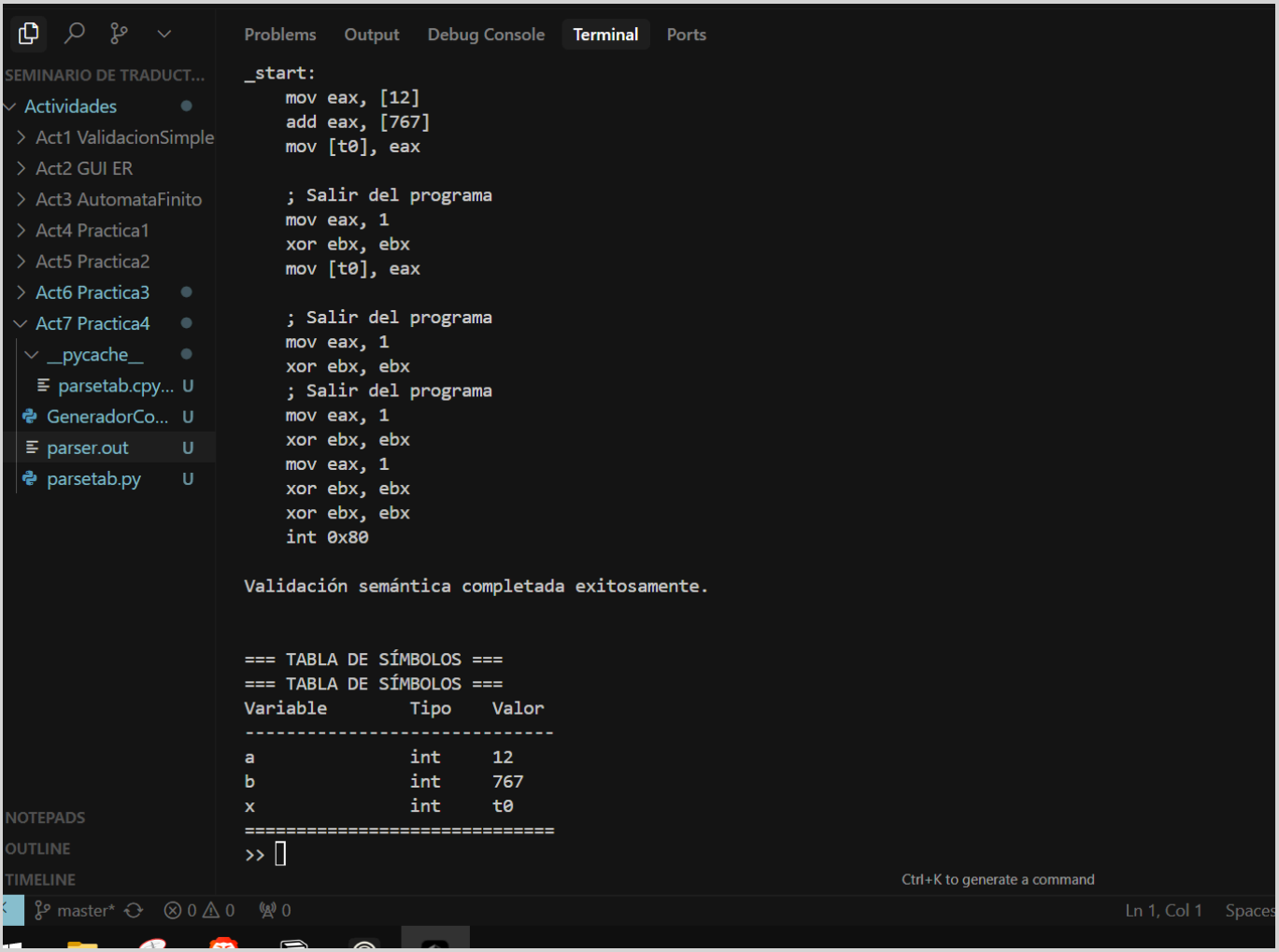



418 state 25
420 (5) declaracion -> FLOAT ID PUNTOCOMA .
Problems
Output
Debug Console
Terminal
Ports
Validación semántica completada exitosamente.
=== TABLA DE SÍMBOLOS ===
Variable Tipo Valor
a int None
>> int b;
Variable 'b' declarada como int
=== CÓDIGO INTERMEDIO ===
0: DECLARE int b
=== CÓDIGO ENSAMBLADOR ===
section .data
; Variables declaradas
a dd 0
b dd 0
section .text
global _start
_start:
; Salir del programa
mov eax, 1
xor ebx, ebx
int 0x80
Ctrl+K to generate a command
Ln 1, Col


t2 GUI ER
t3 AutomataFinito
t4 Practica1
t5 Practica2
t6 Practica3
t7 Practica4
_pycache__
parsetab.cpy... U
eneradorCo... U
arser.out U
arsetab.py U
Problems Output Debug Console Terminal Ports
Validación semántica completada exitosamente.
=== TABLA DE SÍMBOLOS ===
Variable Tipo Valor
a int None
b int None
>> int x;
Variable 'x' declarada como int
=== CÓDIGO INTERMEDIO ===
0: DECLARE int x
=== CÓDIGO ENSAMBLADOR ===
section .data
; Variables declaradas
a dd 0
b dd 0
x dd 0
section .text
global _start
_start:
; Salir del programa
Ctrl+K to generate a command
master* 0 0 0
Ln 1, Col 1


Problems
Output
Debug Console
Terminal
Ports
=== TABLA DE SÍMBOLOS ===
Variable Tipo Valor
a int 12
b int 767
x int None
>> x = a + b;
Asignación: x = t0
Resultado de evaluación: t0
=== CÓDIGO INTERMEDIO ===
0: t0 = 12 + 767
1: ASSIGN x = t0
=== CÓDIGO ENSAMBLADOR ===
section .data
; Variables declaradas
a dd 0
b dd 0
x dd 0
section .text
global _start
_start:
Ctrl+K to generate a command
Ln 1, Col 1


ER
mataFinito
ica1
ica2
ica3
ica4
e__
b.cpy... U
orCo... U
ut U
.py U
Problems
Output
Debug Console
Terminal
Ports
=== CÓDIGO ENSAMBLADOR ===
section .data
; Variables declaradas
a dd 0
b dd 0
x dd 0
section .text
global _start
_start:
mov eax, [12]
add eax, [767]
mov [t0], eax
; Salir del programa
mov eax, 1
xor ebx, ebx
section .text
global _start
_start:
mov eax, [12]
add eax, [767]
mov [t0], eax
; Salir del programa
mov eax, 1
xor ebx, ebx
Ctrl+K to gene



Conclusión:

Que buena actividad, la verdad fue mas o menos sencillo por que ya partía de la actividad anterior, pude reciclar mucho código así que no fue tan difícil como otras actividades anteriores.

Pero la principal estrategia seguida aquí, fue la de dividir y vencer, la idea fue dividir las tareas y resolverlas por separado y juntarlas hasta el final. que también fue un problemon por que es difícil juntar todas las salidas y en lo que si necesite ayuda de IA fue en la generación de ASM, pero ya comprendí cómo es que se hace, que es curioso y medio paradójico, usar un lenguaje de alto nivel como python para generar ASM y Bytecode, mientras que para hacerlo genera su propio ASM y bytecode.

Gracias a esta materia podemos entender de mejor manera las herramientas de programación que usamos todos los días, que no es estrictamente necesario, pero ahora estamos un paso adelante más que el promedio de todos los programadores. Y quizá no sea el área en la que nos queramos especializar pero ahora podemos entender el porqué y el funcionamiento de los compiladores.

Referencias:

colaboradores de Wikipedia. (2025, May 6). Lenguaje ensamblador. Wikipedia, La Enciclopedia Libre. https://es.wikipedia.org/wiki/Lenguaje_ensamblador

colaboradores de Wikipedia. (2024, December 14). Bytecode. Wikipedia, La Enciclopedia Libre. https://es.wikipedia.org/wiki/Bytecode