

Universidad Nacional de Colombia

Facultad de Ingeniería

Departamento de Ingeniería de Sistemas e Industrial

Lenguajes de Programación - Grupo 2

Proyecto Final

Diseño e implementación de tipos de datos en lenguajes de programación

Grupo D

Jorge Isaac Alandete Díaz
Manuel Eduardo Díaz Sabogal
Julián Andrés Gómez Niño
Juan Esteban Medina Cárdenas
Sergio Nicolás Siabatto Cleves

Profesor:

Jorge Eduardo Ortiz Triviño

5 de diciembre de 2025

Índice

1. Resumen	5
2. Marco Teórico	6
2.1 Arquitectura de Computadores y Modelo von Neumann	6
2.1.1 Componentes Fundamentales	6
2.1.2 Ciclo de Instrucción (Fetch-Decode-Execute)	6
2.1.3 Arquitectura Atlas (64 bits)	7
2.2 Lenguaje Ensamblador	7
2.2.1 Características del Lenguaje Ensamblador	7
2.2.2 Elementos del Lenguaje Ensamblador Atlas	7
2.2.3 Proceso de Ensamblado	8
2.3 Compiladores y Procesamiento de Lenguajes	9
2.3.1 Lenguajes de Alto Nivel vs. Bajo Nivel	9
2.3.2 Fases de un Compilador	9
2.3.3 Herramientas para Construcción de Compiladores	9
2.4 Preprocesador	10
Implementación	10
2.4.1 Directivas de Inclusión (#include)	10
2.4.2 Definiciones de Constantes (#define)	11
2.4.3 Macros con Parámetros	11
2.4.4 Compilación Condicional	11
2.5 Análisis Léxico	11
2.5.1 Tokens y Categorías Léxicas	11
2.5.2 Expresiones Regulares y Reconocimiento de Patrones	12
2.5.3 Autómatas Finitos	13
2.5.4 Implementación con PLY Lex	13
2.6 Análisis Sintáctico y Gramáticas	14
2.6.1 Gramáticas Libres de Contexto (GLC)	14
2.6.2 Árboles de Sintaxis Abstracta (AST)	14
2.6.3 Precedencia y Asociatividad de Operadores	15
2.7 Análisis Semántico	15
2.7.1 Tabla de Símbolos	15
2.7.2 Sistema de Tipos	15
2.7.3 Inferencia y Coerción de Tipos	16
2.8 Generación de Código	16
2.8.1 Asignación de Registros y Memoria	16
2.8.2 Convención de Llamada a Función	17
2.8.3 Traducción de Construcciones del Lenguaje	17
2.9 Ensamblador, Enlazador y Cargador	18
2.9.1 Ensamblador	18
2.9.2 Enlazador (Linker)	18
2.9.3 Cargador (Loader)	18
2.10 Pipeline de Compilación Completo del Sistema SPL	18
2.10.1 Integración y API del Compilador	20
2.10.2 Pipeline Completo - Integración	20
3. Descripción del Problema	21
3.1 Visión General del Proyecto	21
3.1.1 Fase 1: Arquitectura y Simulador (Tareas #09 y #14)	21
3.1.2 Fase 2: Preprocesador y Análisis Léxico (Taller #01)	21
3.1.3 Fase 3: Compilador Completo (Taller #02 - Actual)	22
3.2 Tema del Proyecto: Diseño e Implementación de Tipos de Datos en Lenguajes de Programación	22

3.2.1 Sistema de Tipos Completo	22
3.2.2 Implementación en el Compilador	23
3.2.3 Representación en Memoria de Tipos	23
3.2.4 Inicialización Determinista de Datos	23
3.2.5 Validación del Sistema de Tipos	24
3.3 Requisitos Específicos del Taller #02	24
3.3.1 Definición Formal de Gramática E-BNF	24
3.3.2 Ajustes al Analizador Léxico	25
3.3.3 Diagramas de Sintaxis Ferroviarios (BottleCaps)	25
3.3.4 Interpretación Semántica Detallada	25
3.3.6 Análisis Semántico y Generación de Código	25
3.3.7 Validación con Programas de Prueba	25
3.4 Desafíos Técnicos del Taller #02	26
3.4.1 Fase de Análisis Sintáctico	26
3.4.2 Fase de Análisis Semántico	26
3.4.3 Fase de Generación de Código	26
3.5 Gramática E-BNF del Lenguaje SPL	26
3.5.1 Estructura del Programa	26
3.5.2 Declaraciones de Función	27
3.5.3 Declaraciones de Estructura	27
3.5.4 Declaraciones de Variable	27
3.5.5 Sistema de Tipos	27
3.5.6 Sentencias	27
3.5.7 Sentencias de Control de Flujo	28
3.5.8 Expresiones	28
3.5.9 Reglas Léxicas (Terminales)	29
3.5.10 Palabras Reservadas	30
3.6 Diagramas de Sintaxis (BottleCaps)	30
3.6.1 Herramienta Utilizada	30
3.6.2 Estructura del Programa	30
3.6.3 Declaraciones de Función	31
3.6.4 Declaraciones de Estructura	32
3.6.5 Declaraciones de Variable	33
3.6.6 Sistema de Tipos	33
3.6.7 Sentencias	34
3.6.8 Sentencias de Control de Flujo	36
3.6.9 Expresiones	38
3.6.10 Gestión de Memoria	44
3.6.11 Reglas Léxicas (Terminales)	44
3.6.12 Diagrama Completo de la Gramática	48
3.6.13 Uso de los Diagramas	48
3.7 Interpretación Semántica de las Producciones	49
3.7.1 Resumen: Soporte de Aritmética de Punto Flotante	58
4. Validación y Evidencias	63
4.1 Metodología de Validación	63
4.1.1 Estrategia de Pruebas	63
4.2 Algoritmo de Euclides - Validación Completa	63
4.2.1 Implementación en Atlas Assembly	63
4.2.2 Verificación Matemática	63
4.3 Algoritmo del Módulo - Operación a % b	64
4.3.1 Implementación y Validación	64
4.4 Algoritmo de Valor Absoluto	64
4.4.1 Implementación con Complemento a 2	64

4.5 Validación del Conjunto de Instrucciones	64
4.5.1 Cobertura Completa (47 instrucciones)	64
4.5.2 Métricas de Calidad Alcanzadas	65
4.6 Ejemplos de Programas en SPL (Lenguaje de Alto Nivel)	65
4.6.1 Ejemplo 1: Cálculo de Áreas y Volúmenes	65
4.6.2 Ejemplo 2: Bucles y Control de Flujo	66
4.6.3 Ejemplo 3: Estructuras y Punteros	66
4.6.4 Ejemplo 4: Algoritmo de Euclides (Obligatorio)	66
4.7 Validación del Pipeline Completo	67
5. Diseño de la Aplicación	68
5.1 Arquitectura General del Sistema	68
5.1.1 Componentes Principales	68
5.2 Diseño del Procesador (CPU.py)	68
5.2.1 Arquitectura de 64 bits	68
5.2.2 Formatos de Instrucción Implementados	69
5.3 Diagrama de Clases UML - Módulo Compiler	70
5.4 Diagrama de Clases UML - Módulo Machine	72
5.5 Diagrama de Secuencia - Pipeline de Compilación y Ejecución	74
5.6 Compilador SPL (Taller 2)	74
5.6.1 Analizador Sintáctico (syntax_analyzer.py)	74
5.6.2 Analizador Semántico (semantic_analyzer.py)	76
5.6.3 Generador de Código (code_generator.py)	78
5.7 Ensamblador (ensamblador.py)	82
5.7.1 Análisis Sintáctico	82
5.7.2 Generación de Código	83
5.8 Flujo de Ejecución	84
5.8.1 Diagrama de Flujo Principal	84
5.9 Interfaz Gráfica (GUI)	85
5.9.1 Componentes de la UI	85
5.9.2 Visor de RAM en tiempo real (novedad)	85
5.9.3 Limpieza de RAM desde la interfaz (novedad)	85
5.10 Flujo del Compilador SPL (Preprocesador → Analizador Léxico → Sintáctico → IR → Opt → Ensamblador → Enlazador/Cargador)	85
5.10.1 Pipeline general (fases)	86
5.10.2 Flujo de tokens → árbol sintáctico → AST	88
5.10.3 Optimización y generación final	89
5.11 Especificación Léxica del Lenguaje	90
5.11.1 Keywords	90
5.11.2 Identificadores	90
5.11.3 Constantes	90
5.11.4 Comentarios	90
5.11.5 Operadores	91
5.11.6 Delimitadores	91
5.11.7 Expresiones regulares	91
6. Manual Técnico y de Usuario	92
6.1 Instalación y Configuración	92
6.1.1 Requisitos del Sistema	92
6.1.2 Proceso de Instalación	92
6.2 Manual de Usuario	92
6.2.1 Lenguaje de Programación SPL (Sistema de Procesamiento de Lenguajes)	92
6.2.2 Uso de la Interfaz Gráfica (GUI)	96
6.2.3 Inicio Rápido Assembly	98

6.2.4 Referencia del Lenguaje Assembly	98
6.2.5 Ejemplos Prácticos Assembly	101
6.3 Manual Técnico	104
6.3.1 API del Sistema	104
6.3.2 Estructura de Datos Interna	104
6.3.3 Extensibilidad	105
7. Especificaciones Técnicas	106
7.1 Arquitectura del Procesador	106
7.1.1 Especificaciones Generales	106
7.1.2 Conjunto de Instrucciones Completo (ISA)	106
7.2 Sistema de Memoria	110
7.2.1 Configuración de Memoria	110
7.2.2 Mapa de Memoria	110
7.2.3 Persistencia de la RAM en archivo de texto (novedad)	110
7.3 Sistema de E/S	111
7.3.1 Direcciones de E/S Reservadas	111
7.3.2 Protocolo de Comunicación	111
8. Documentación de Experimentación y Resultados	113
8.1 Escenario 1	113
8.2 Escenario 2	114
8.3 Escenario 3	115
8.4 Escenario 4	116
8.5 Análisis	117
9. Conclusiones	117
9.1 Logros Principales	117
9.2 Impacto Educativo	117

1. Resumen

Este proyecto presenta el diseño e implementación completa de **Atlas**, un sistema computacional integrado que abarca desde la arquitectura de hardware hasta un compilador funcional. El sistema implementa un procesador de 64 bits con arquitectura RISC que ejecuta 137+ instrucciones, incluyendo operaciones enteras, punto flotante IEEE 754 y gestión de memoria virtual. Se desarrolla un compilador completo para el lenguaje de alto nivel SPL que incluye preprocesador, analizador léxico con PLY, parser LALR(1) basado en gramática E-BNF, analizador semántico con sistema de tipos robusto, y generador de código que traduce a ensamblador Atlas.

El proyecto aborda específicamente el tema “**Diseño e implementación de tipos de datos en lenguajes de programación**”, implementando un sistema completo con 11 tipos primitivos (entero1-8, flotante, doble, booleano, caracter, cadena), tipos compuestos (estructuras, arreglos, punteros) y verificación de compatibilidad en tiempo de compilación. La generación de código utiliza metadatos `.LOCAL_REL` para variables BP-relativas, inicialización determinista mediante directivas `.DATA`, y asignación de literales en área dedicada `string_data_base`.

El simulador incluye interfaz gráfica con visualización en tiempo real de registros, memoria, flags y E/S, permitiendo ejecución paso a paso y depuración visual. El sistema ha sido validado mediante algoritmos clásicos (Euclides, multiplicación de matrices) y suite exhaustiva de pruebas unitarias e integración. Los resultados demuestran la corrección del pipeline completo de compilación y ejecución, constituyendo una herramienta educativa efectiva para la enseñanza de arquitectura de computadores, compiladores y sistemas de tipos.

2. Marco Teórico

2.1 Arquitectura de Computadores y Modelo von Neumann

El diseño de computadores modernos se fundamenta en los principios establecidos por John von Neumann en 1945, quien propuso una arquitectura en la cual las instrucciones y los datos se almacenan en una memoria común y se ejecutan secuencialmente bajo el control de una unidad central de procesamiento (CPU). Este modelo, conocido como **arquitectura von Neumann**, constituye la base conceptual de prácticamente todos los computadores actuales.

2.1.1 Componentes Fundamentales

La arquitectura von Neumann se compone de cinco componentes principales:

1. Unidad Central de Procesamiento (CPU): - **ALU (Unidad Aritmético-Lógica):** Realiza operaciones aritméticas (suma, resta, multiplicación, división) y lógicas (AND, OR, NOT, XOR) - **Unidad de Control:** Coordina y sincroniza todas las operaciones del sistema, decodifica instrucciones y genera señales de control - **Registros:** Almacenamiento interno de alta velocidad para datos temporales, direcciones e instrucciones - **PC (Program Counter):** Registro que mantiene la dirección de la siguiente instrucción a ejecutar - **Registro de Flags:** Almacena indicadores de estado como Zero (Z), Negative (N), Carry (C), Overflow (V)

2. Memoria Principal: - Almacenamiento de acceso aleatorio (RAM) que contiene tanto instrucciones como datos - Organizada en celdas direccionables, cada una identificada por una dirección única - En arquitecturas modernas, típicamente organizada en bytes (8 bits)

3. Sistema de Entrada/Salida (E/S): - Dispositivos periféricos para comunicación con el exterior (teclado, pantalla, disco, red) - Controladores que gestionan la transferencia de datos entre CPU y dispositivos

4. Buses del Sistema: - **Bus de Datos:** Transporta información (instrucciones y datos) entre componentes, bidireccional - **Bus de Direcciones:** Transmite direcciones de memoria o dispositivos, unidireccional desde CPU - **Bus de Control:** Lleva señales de control (lectura, escritura, reloj, interrupciones)

5. Unidad de Punto Flotante (FPU): - Componente especializado (puede estar integrado o separado) para operaciones con números reales - Implementa aritmética IEEE 754 para tipos float y double

2.1.2 Ciclo de Instrucción (Fetch-Decode-Execute)

El procesador ejecuta programas mediante un ciclo repetitivo de tres fases:

1. Fetch (Búsqueda):

- Leer la instrucción desde memoria[PC]
- Copiar la instrucción al registro de instrucción (IR)
- Incrementar PC para apuntar a la siguiente instrucción

2. Decode (Decodificación):

- La unidad de control analiza el opcode de la instrucción
- Identifica la operación a realizar
- Extrae los operandos (registros, inmediatos, direcciones de memoria)

3. Execute (Ejecución):

- La ALU/FPU realiza la operación especificada
- Se actualizan registros, memoria y/o flags según corresponda
- Para instrucciones de salto, se modifica el PC

Este ciclo se repite continuamente hasta que el programa termina o se encuentra una instrucción de detención (HALT).

2.1.3 Arquitectura Atlas (64 bits)

En este proyecto se ha diseñado e implementado la arquitectura **Atlas**, un computador de 64 bits que sigue el modelo von Neumann con las siguientes características:

Especificaciones técnicas: - **Tamaño de palabra:** 64 bits (8 bytes) - **Direccionamiento:** Por byte (cada dirección referencia 1 byte) - **Espacio de direcciones:** 44 bits de direccionamiento (16 TiB teóricos, 64 KB implementados) - **Registros de propósito general:** 16 registros (R00-R15), cada uno de 64 bits - **Registros especiales:** - R14: BP (Base Pointer) para gestión de stack frames - R15: SP (Stack Pointer) para tope de pila - **Registro de flags:** 4 bits (Z, N, C, V) - **Conjunto de instrucciones:** 137+ instrucciones (ISA)

Organización de memoria: - **Segmento de código:** Desde 0x0000, contiene instrucciones del programa - **Segmento de datos globales:** Desde 0x1000, variables globales y constantes - **Segmento de stack:** Desde 0x8000, crece hacia direcciones mayores

Nota de implementación: en la versión actual del compilador/ensamblador se reserva además un área dedicada para literales de cadena (strings) cuyo offset base de datos en el repositorio se sitúa en `string_data_base` (implementación actual alrededor de 0x18000). El **Loader** del emulador aplica las inicializaciones declaradas en la sección `.data` de forma determinista durante la carga, escribiendo los bytes UTF-8 de las cadenas y las constantes en memoria para asegurar valores iniciales reproducibles entre ejecuciones.

Formatos de instrucción: - **OP:** Operación sin operandos (ej: HALT, RET, NOP) - **R:** Un operando registro (ej: PUSH8 R01) - **RR:** Dos operandos registro (ej: ADD8 R01, R02) - **RI:** Registro e inmediato (ej: LOADV4 R01, 100) - **I:** Solo inmediato o dirección (ej: JMP 0x1000, CALL funcion)

Sufijos de tamaño: Las instrucciones incluyen sufijos que indican el tamaño de operación: - 1: 1 byte (8 bits) - 2: 2 bytes (16 bits) - 4: 4 bytes (32 bits) - 8: 8 bytes (64 bits) - Ejemplo: ADD1, ADD2, ADD4, ADD8

Soporte de punto flotante: Instrucciones con prefijo F para operaciones flotantes: - FADD4, FSUB4, FMUL4, FDIV4: Operaciones de 32 bits (float) - FADD8, FSUB8, FMUL8, FDIV8: Operaciones de 64 bits (double)

Sistema de E/S: - E/S mapeada en memoria (memory-mapped I/O) - Dispositivos accesibles mediante instrucciones de carga/almacenamiento estándar - Instrucciones especializadas: SVIO (guardar), LDIO (cargar), SHOWIO (visualizar)

2.2 Lenguaje Ensamblador

El lenguaje ensamblador es un lenguaje de programación de bajo nivel que utiliza **mnemónicos** (abreviaturas legibles) para representar instrucciones de código máquina. Cada instrucción en ensamblador corresponde típicamente a una instrucción binaria ejecutable por el procesador.

2.2.1 Características del Lenguaje Ensamblador

Ventajas: - Control preciso sobre el hardware y recursos del sistema - Optimización máxima del rendimiento y uso de memoria - Acceso directo a registros, memoria y dispositivos E/S - Ideal para programación de sistemas, drivers y código crítico

Desventajas: - Específico de cada arquitectura (no portable) - Complejidad mayor que lenguajes de alto nivel - Mayor propensión a errores - Desarrollo y mantenimiento más costoso

2.2.2 Elementos del Lenguaje Ensamblador Atlas

Instrucciones: Mnemónicos que representan operaciones del procesador

```
LOAD8 R01, [0x1000]    ; Cargar 8 bytes desde memoria
ADD4 R01, R02           ; Sumar dos registros (32 bits)
JZ etiqueta            ; Saltar si flag Zero está activo
```

Etiquetas: Identificadores simbólicos para direcciones de memoria


```

inicio:
    LOADV4 R01, 10
    JMP loop
loop:
    ...

```

Directivas del ensamblador: Comandos para el ensamblador (no generan código)

```

.data                ; Sección de datos
.text               ; Sección de código
.global main        ; Símbolo exportado

```

Comentarios: Documentación del código (ignorada por el ensamblador)

```

; Comentario de línea completa
LOAD8 R01, [0x1000] ; Comentario al final de línea

```

Nota de implementación del ensamblador: el generador de código y el ensamblador actual emiten directivas `.data` que el Loader interpreta y aplica de forma determinista durante la carga del binario. Además, el compilador emite, para variables locales y parámetros, metadatos en formato `.LOCAL_REL <offset> <size> <name> ; FUNC=<funcname>` que permiten registrar símbolos BP-relativos (offsets relativos a BP / R14) y obtener nombres legibles en tiempo de ejecución (por ejemplo, para logs de STORE).

Ejemplos de salida del generador (muestras)

Ejemplo de metadato para variable local (emitido por el generador al final de la función):

```

; Metadatos de variables locales emitidos por el generador
.LOCAL_REL -20 8 nota ; FUNC=principal
.LOCAL_REL -28 4 contador ; FUNC=principal

```

Ejemplo de inicialización byte-a-byte de una cadena literal (área `string_data_base` 0x18000):

```

; Inicialización de string literal "Hola" en 0x18010
MOVV1 R01, 72 ; byte 0x48 ('H')
STORE1 R01, 0x18010 ; Escribir 'H'
MOVV1 R01, 111 ; byte 0x6F ('o')
STORE1 R01, 0x18011 ; Escribir 'o'
MOVV1 R01, 108 ; byte 0x6C ('l')
STORE1 R01, 0x18012 ; Escribir 'l'
MOVV1 R01, 97 ; byte 0x61 ('a')
STORE1 R01, 0x18013 ; Escribir 'a'
MOVV1 R01, 0 ; NULL terminator
STORE1 R01, 0x18014 ; Escribir '\0'

```

Estos fragmentos son representativos de la salida que genera el `CodeGenerator` y que el `Loader` interpreta para registrar símbolos y escribir literales en memoria.

2.2.3 Proceso de Ensamblado

El ensamblador traduce código ensamblador a código máquina mediante **dos pasadas**:

Primera pasada: 1. Construir tabla de símbolos con direcciones de todas las etiquetas 2. Calcular tamaños de instrucciones y datos 3. Asignar direcciones a cada instrucción y dato 4. Detectar errores sintácticos básicos

Segunda pasada: 1. Generar código binario para cada instrucción 2. Resolver referencias a etiquetas (sustituir por direcciones) 3. Marcar símbolos externos para el enlazador 4. Generar archivo objeto (.o) con código, símbolos y relocalaciones

2.3 Compiladores y Procesamiento de Lenguajes

Un **compilador** es un programa traductor que convierte código escrito en un lenguaje de programación de alto nivel (lenguaje fuente) a un lenguaje de nivel inferior (lenguaje objetivo), típicamente lenguaje ensamblador o código máquina. A diferencia del lenguaje ensamblador que es de bajo nivel y específico de arquitectura, los lenguajes de alto nivel ofrecen abstracciones que facilitan el desarrollo de software complejo.

2.3.1 Lenguajes de Alto Nivel vs. Bajo Nivel

Lenguajes de alto nivel (ej: C, Java, Python, SPL): - Abstracciones: variables, funciones, estructuras de control, tipos de datos complejos - Independencia de la arquitectura (portabilidad) - Mayor productividad del programador - Expresividad y legibilidad mejoradas

Lenguajes de bajo nivel (ensamblador, código máquina): - Mapeo directo a instrucciones del procesador - Control fino sobre hardware - Máximo rendimiento y eficiencia - Específicos de cada arquitectura

El compilador actúa como **punto** entre ambos niveles, permitiendo programar con abstracciones de alto nivel mientras genera código eficiente de bajo nivel.

2.3.2 Fases de un Compilador

El proceso de compilación se organiza en múltiples fases que procesan progresivamente el código fuente:

Front-end (Análisis) - Procesa el código fuente y verifica su validez: 1. **Preprocesador**: Expansión de macros, inclusión de archivos, procesamiento de directivas 2. **Análisis Léxico**: Conversión del código fuente en secuencia de tokens 3. **Análisis Sintáctico**: Verificación de la estructura gramatical y construcción del AST 4. **Análisis Semántico**: Validación de tipos, scopes y reglas semánticas

Back-end (Síntesis) - Genera código ejecutable optimizado: 5. **Generación de Código Intermedio**: Representación independiente de la arquitectura 6. **Optimización**: Mejora de eficiencia (eliminación de código muerto, propagación de constantes, etc.) 7. **Generación de Código**: Traducción a lenguaje ensamblador de la arquitectura objetivo 8. **Ensamblado y Enlazado**: Producción del ejecutable final

2.3.3 Herramientas para Construcción de Compiladores

Existen herramientas especializadas que automatizan la generación de componentes del compilador:

Lex/Flex (generadores de analizadores léxicos): - Entrada: Especificación de tokens mediante expresiones regulares - Salida: Código C que implementa el analizador léxico

Yacc/Bison (generadores de analizadores sintácticos): - Entrada: Gramática libre de contexto en notación BNF - Salida: Código C que implementa el parser LALR(1)

PLY (Python Lex-Yacc): Implementación en Python de Lex y Yacc desarrollada por David Beazley: - **PLY Lex**: Generador de analizadores léxicos basado en expresiones regulares - **PLY Yacc**: Generador de parsers LALR(1) basado en gramáticas libres de contexto - **Ventajas**: - Integración nativa con Python (sin archivos intermedios) - Debugging facilitado mediante trazas y mensajes de error claros - Documentación completa y comunidad activa - Soporte para Unicode

Gramática: soporte `si`, `si_no_si` y `si_no`

El parser implementado con PLY soporta cláusulas `si_no_si` (equivalente a `elif`) como una lista de cláusulas intermedias y un bloque `si_no` (else) opcional. La forma esencial de las producciones usadas en `src/compiler/syntax_analyzer.py` es equivalente a:

```
if_stmt : SI PARIZQ expression PARDER statement
        | SI PARIZQ expression PARDER statement elif_list
        | SI PARIZQ expression PARDER statement SI_NO statement
        | SI PARIZQ expression PARDER statement elif_list SI_NO statement
```

```
elif_list : elif_clause
          | elif_list elif_clause
```

```
elif_clause : SI_NO_SI PARIZQ expression PARDER statement
```

Nota semántica: el analizador semántico (`semantic_analyzer.py`) impone un límite de 10 cláusulas `si_no_si` por sentencia `si` (parámetro `MAX_ELIF_CLAUSES = 10`) y reporta un error si se excede.

EBNF sincronizado (extracto)

El fragmento EBNF equivalente, sincronizado con el fichero `Documentacion/Proyecto final/gramatica/gramatica.ebnf`, es el siguiente:

```
if_stmt ::= 'si' '(' expression ')' statement ('si_no_si' '(' expression ')' statement)* ('si_no' staten
```

```
/* donde 'si_no_si' representa la palabra reservada 'si_no_si' */
```

Este fragmento refleja exactamente la producción usada por el parser y está presente en `gramatica.ebnf`.

2.4 Preprocesador

El preprocesador es una herramienta que actúa **antes** del compilador principal, transformando el código fuente mediante expansión textual de directivas especiales. El preprocesamiento es una fase de transformación sintáctica que no analiza la semántica del lenguaje.

Implementación

El preprocesador está implementado en `Preprocessor.py` usando **PLY Lex** para análisis léxico de directivas.

Función principal:

```
preprocess(code: str, base_path: str = ".") -> str
```

Tokens del preprocesador: - **DIRECTIVE:** Palabras clave (`define`, `include`, `ifdef`, `ifndef`, `endif`) - **TEXT:** Texto normal del código fuente - **NEWLINE:** Saltos de línea (preservados para números de línea) - **HASH:** Símbolo `#` que inicia directivas

Directivas soportadas: 1. `#include .archivo`: Inclusión de archivos (busca en `lib/`) 2. `#define NOMBRE valor`: Macros simples 3. `#define NOMBRE(a, b) expresion`: Macros con parámetros 4. `#ifdef NOMBRE / #ifndef NOMBRE / #endif`: Compilación condicional

Estado interno: - `macros = {}`: Diccionario global de definiciones - `conditional_stack = []`: Pila para anidamiento de condicionales

2.4.1 Directivas de Inclusión (`#include`)

Permiten incorporar contenido de archivos externos, facilitando la modularización y reutilización de código:

```
#include "io.asm"      ; Incluir biblioteca de E/S
#include "math.asm"    ; Incluir funciones matemáticas
```

Funcionamiento: 1. El preprocesador busca el archivo en el directorio `lib/` del proyecto: `{base_path}/lib/{archivo}` 2. Lee el contenido completo del archivo 3. Inserta el contenido en el punto exacto de la directiva 4. Procesa recursivamente los `#include` dentro del archivo incluido 5. Previene inclusiones recursivas infinitas mediante tracking de archivos procesados

Ventajas: - Separación de código en módulos reutilizables (bibliotecas) - Organización lógica del proyecto - Evita duplicación de código común

2.4.2 Definiciones de Constantes (`#define`)

Permiten asignar nombres simbólicos a valores constantes:

```
#define PI 3.14159
#define MAX_SIZE 1024
#define IO_BASE 0x100
```

El preprocesador reemplaza todas las ocurrencias del identificador por su valor antes de la compilación. Los macros se almacenan en el diccionario global **macros** y se aplican mediante sustitución textual.

2.4.3 Macros con Parámetros

Las macros permiten definir patrones de código que se expanden con sustitución de parámetros:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define CUADRADO(x) ((x) * (x))
```

Uso:

```
entero4 mayor = MAX(10, 20);
entero4 area = CUADRADO(lado);
```

Expansión (lo que ve el compilador):

```
entero4 mayor = ((10) > (20) ? (10) : (20));
entero4 area = ((lado) * (lado));
```

2.4.4 Compilación Condicional

Permite incluir o excluir bloques de código según macros definidas:

```
#define DEBUG

#ifdef DEBUG
    imprimir("Modo debug activo");
#endif

#ifndef RELEASE
    // Código solo en versión desarrollo
#endif
```

El preprocesador usa una pila `conditional_stack` para manejar anidamiento de condicionales.

Ventajas del preprocesador: - Reducción de código repetitivo - Abstracción de patrones comunes - Mantenibilidad (cambio único en la definición) - Configuración de compilación (debug/release) - Modularización mediante archivos de biblioteca

2.5 Análisis Léxico

El análisis léxico es la **primera fase** del front-end del compilador, responsable de leer el código fuente como flujo de caracteres y agruparlos en unidades léxicas significativas llamadas **tokens**. También se conoce como **scanner** o **lexer**.

2.5.1 Tokens y Categorías Léxicas

Un **token** es una tupla que representa una unidad léxica: (**tipo**, **valor**, **línea**, **columna**)

Ejemplo:

```

Tokens:      (KEYWORD, 'int', 1, 1)
              (IDENTIFIER, 'x', 1, 5)
              (ASSIGN, '=', 1, 7)
              (INTEGER, 42, 1, 9)
              (SEMICOLON, ';', 1, 11)

```

1. **Palabras reservadas:** Identificadores con significado especial en el lenguaje
 - Ejemplos: `function, if, else, while, for, return, struct, new, delete, int, float`
2. **Identificadores:** Nombres definidos por el programador para variables, funciones, tipos
 - Patrón: Letra o `_` seguido de letras, dígitos o `_`
 - Ejemplos: `miVariable, calcularArea, Persona_t`
3. **Literales:** Valores constantes
 - Enteros: `42, 0xFF, 0b1010`
 - Flotantes: `3.14, 2.5e-3, 1.0f`
 - Cadenas: `"Hola mundo"`
 - Caracteres: `'a', '\n'`
 - Booleanos: `true, false`
4. **Operadores:** Símbolos que denotan operaciones
 - Aritméticos: `+, -, *, /, %`
 - Relacionales: `<, >, <=, >=, ==, !=`
 - Lógicos: `&&, ||, !`
 - Bitwise: `&, |, ^, ~, <<, >>`
 - Asignación: `=, +=, -=, *=, /=`
 - Punteros: `*, &`
 - Miembro: `., ->`
5. **Delimitadores:** Símbolos de puntuación y agrupación
 - Paréntesis: `(,)`
 - Llaves: `{, }`
 - Corchetes: `[,]`
 - Punto y coma: `;`
 - Coma: `,`
 - Dos puntos: `:`
6. **Comentarios:** Texto informativo ignorado por el compilador
 - Línea: `// comentario`
 - Bloque: `/* comentario multilínea */`

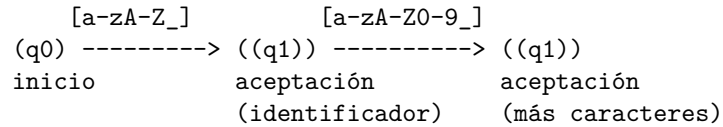
El reconocimiento de tokens se basa en **expresiones regulares** que describen patrones de caracteres:

Categoría	Expresión Regular	Ejemplos
Identificador	[a-zA-Z_] [a-zA-Z0-9_]*	var, _temp, calcTotal
Entero decimal	[0-9]+	0, 42, 1000
Hexadecimal	0[xX] [0-9a-fA-F]+	0xFF, 0x10
Binario	0[bB] [01]+	0b1010, 0B11
Flotante	[0-9]+\.[0-9]+([eE] [+]?[0-9]+)?	3.14, 2.5e-3
Cadena	"([^\\"\\ \\.])*"	"texto", "line\n"
Carácter	'([^\\"\\ \\.])'	'a', '\t'

2.5.3 Autómatas Finitos

Las expresiones regulares se implementan mediante **Autómatas Finitos Determinísticos (DFA)** que procesan el flujo de caracteres de manera eficiente:

Ejemplo: Reconocer identificador `[a-zA-Z_][a-zA-Z0-9_]*`



El DFA lee caracteres secuencialmente, transita entre estados y acepta cuando llega a un estado final.

2.5.4 Implementación con PLY Lex

En PLY, los tokens se definen mediante funciones Python con expresiones regulares en docstrings:

```
import ply.lex as lex

# Lista de tokens
tokens = ['IDENTIFIER', 'INTEGER', 'PLUS', 'LPAREN', 'RPAREN']

# Palabras reservadas
reserved = {'if': 'IF', 'else': 'ELSE', 'while': 'WHILE'}
tokens += list(reserved.values())

# Reglas de tokens (expresiones regulares)
t_PLUS = r'\+'
t_LPAREN = r'\('
t_RPAREN = r'\)'

def t_IDENTIFIER(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    t.type = reserved.get(t.value, 'IDENTIFIER') # Verificar palabra reservada
    return t

def t_INTEGER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Caracteres ignorados (espacios, tabs)
t_ignore = ' \t'

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

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

# Construir el lexer
```

```
lexer = lex.lex()
```

2.6 Análisis Sintáctico y Gramáticas

El análisis sintáctico es la **segunda fase** del compilador y verifica que la secuencia de tokens producida por el lexer cumple con las **reglas gramaticales** del lenguaje. Si el análisis léxico trabaja con patrones lineales (expresiones regulares), el análisis sintáctico maneja **estructuras jerárquicas** y anidadas que requieren gramáticas más potentes.

2.6.1 Gramáticas Libres de Contexto (GLC)

Una gramática libre de contexto se define como $G = (V, T, R, S)$:

- **V**: Símbolos no terminales (construcciones del lenguaje)
- **T**: Símbolos terminales (tokens)
- **R**: Reglas de producción $A \rightarrow \text{alfa}$
- **S**: Símbolo inicial

Notación E-BNF (Extended Backus-Naur Form) extiende BNF con operadores de repetición y opcionalidad:

Operador	Significado	Ejemplo
?	Opcional (0 o 1)	<code>param_list?</code>
*	Repetición (0+)	<code>statement*</code>
+	Repetición (1+)	<code>declaration+</code>
<code>\ </code>	Alternativa	<code>type \ void</code>
<code>()</code>	Agrupación	<code>('+' \ '-')</code>

Ejemplo de gramática E-BNF para expresiones aritméticas:

```
expression ::= term (('+' | '-') term)*
term ::= factor (('*' | '/') factor)*
factor ::= INTEGER | IDENTIFIER | '(' expression ')'
```

2.6.2 Árboles de Sintaxis Abstracta (AST)

El AST es una representación intermedia que:

- Elimina detalles sintácticos irrelevantes (paréntesis, punto y coma)
- Preserva la estructura semántica del programa
- Facilita análisis semántico y generación de código
- Permite optimizaciones mediante transformaciones del árbol

Ejemplo:

Código: `x = a + b * c;`

```
AST:      Assignment
         +- target: Identifier(x)
         +- value: BinaryOp(+)
             +- left: Identifier(a)
             +- right: BinaryOp(*)
                 +- left: Identifier(b)
                 +- right: Identifier(c)
```

2.6.3 Precedencia y Asociatividad de Operadores

Para evitar ambigüedades en expresiones, se definen reglas de precedencia (orden de evaluación) y asociatividad (dirección de agrupación):

Precedencia	Operador	Asociatividad	Ejemplo
1 (mayor)	<code>[]</code> , <code>()</code> , <code>.</code>	Izquierda	<code>a[i].x</code>
2	<code>!</code> , <code>~</code> , <code>*</code> (puntero), <code>&</code> (dirección)	Derecha	<code>*ptr</code>
3	<code>*</code> , <code>/</code> , <code>%</code>	Izquierda	<code>a * b / c</code>
4	<code>+</code> , <code>-</code>	Izquierda	<code>a + b - c</code>
5	<code><</code> , <code>></code> , <code><=</code> , <code>>=</code>	Izquierda	<code>a < b</code>
6	<code>==</code> , <code>!=</code>	Izquierda	<code>a == b</code>
7	<code>&&</code>	Izquierda	<code>a && b</code>
8	<code>\ \ </code>	Izquierda	<code>a \ \ b</code>
9 (menor)	<code>=</code> , <code>+=</code> , <code>-=</code>	Derecha	<code>a = b = c</code>

2.7 Análisis Semántico

El análisis semántico es la **tercera fase** del compilador y verifica la corrección del programa más allá de su sintaxis, validando reglas que no pueden expresarse mediante gramáticas libres de contexto. Mientras que el análisis sintáctico verifica la **forma** del programa, el análisis semántico verifica el **significado**.

2.7.1 Tabla de Símbolos

La tabla de símbolos es una estructura de datos fundamental que almacena información sobre todos los identificadores (variables, funciones, tipos, constantes) usados en el programa:

Información almacenada por símbolo: - **Nombre:** Identificador único en su scope - **Tipo:** Tipo de dato (primitivo, estructura, puntero, arreglo) - **Categoría:** Variable, función, parámetro, constante, tipo - **Alcance:** Global, local (función), bloque, estructura - **Offset/Dirección:** Ubicación en memoria o stack frame - **Atributos:** Inicializado, constante, externo

Gestión de scopes anidados: Los lenguajes estructurados en bloques requieren múltiples tablas organizadas en pila:

```
Scope Global
+- var x: int4
+- function f(a: int4)
| +- Scope de función f
| | +- param a: int4
| | +- local b: float
| | +- if-block
| |   +- Scope del bloque if
| |   +- local c: int2
```

Reglas de resolución de nombres: 1. Buscar en scope actual 2. Si no se encuentra, buscar en scope padre 3. Continuar hasta scope global 4. Error si no se encuentra en ningún scope

2.7.2 Sistema de Tipos

El sistema de tipos define qué operaciones son válidas y cómo se comportan:

1. **Tipos primitivos:** `int2`, `int4`, `int8`, `float`, `double`, `char`, `bool`, `void`
2. **Tipos compuestos:** Punteros (`*type`), estructuras (`struct`), arreglos (`type[]`)
3. **Compatibilidad de tipos:** Qué operaciones son válidas entre qué tipos
4. **Conversiones implícitas:** Promoción automática (ej: `int4` \rightarrow `float`)

5. **Equivalencia estructural:** Para structs, comparar estructura de campos

Verificaciones semánticas típicas: - Variables declaradas antes de uso - Tipos compatibles en asignaciones y operaciones - Número y tipo correcto de argumentos en llamadas a función - Funciones no-void deben retornar un valor - Break/continue solo dentro de loops - Return solo dentro de funciones - No redeclaración de símbolos en el mismo scope - Acceso a miembros solo en estructuras - Operaciones de punteros solo con tipos puntero

2.7.3 Inferencia y Coerción de Tipos

Conversiones implícitas (coerción):

```
int4 + float    → float + float    (promoción de int4 a float)
int2 = int4     → int2 = (int2)int4 (truncamiento)
```

Reglas de promoción (de menor a mayor):

```
int2 → int4 → int8 → float → double
```

2.8 Generación de Código

La generación de código es la **cuarta fase** del compilador y traduce el AST validado semánticamente a código ensamblador de la arquitectura objetivo (Atlas). Esta es la fase de **síntesis** donde se produce el código de salida.

2.8.1 Asignación de Registros y Memoria

Estrategias de asignación:

1. **Registros temporales:** R00-R09 se usan para evaluar expresiones intermedias

```
x = a + b * c;

R00 = b
R01 = c
R02 = R00 * R01    ; b * c
R03 = a
R04 = R03 + R02    ; a + (b * c)
[x] = R04
```

2. **Registros especiales:**

- R14: BP (Base Pointer) - base del stack frame actual
- R15: SP (Stack Pointer) - tope de la pila

3. **Variables globales:** Direcciones absolutas en segmento de datos (desde 0x1000)

```
.data
g_contador: .word 0      ; 0x1000
g_total: .word 0         ; 0x1008
```

4. **Variables locales:** Offsets negativos desde BP en stack frame

```
local_a: BP - 8
local_b: BP - 16
```

5. **Parámetros de función:** Offsets positivos desde BP

```
param_1: BP + 16
param_2: BP + 24
```

2.8.2 Convención de Llamada a Función

La convención de llamada define cómo se pasan parámetros, se invoca la función y se retornan valores:

Stack Layout (crece hacia direcciones mayores):

+-----+		
Argumento N		BP + 16 + (N-1)*8
Argumento 2		BP + 24
Argumento 1		BP + 16
+-----+		
Dir. Retorno		BP + 8 (guardada por CALL)
+-----+		
BP anterior		BP (guardado en prólogo)
+-----+		
Variable local 1		BP - 8
Variable local 2		BP - 16
+-----+		
		SP (actual)

Prólogo de función:

```
funcion_nombre:
    PUSH8 R14          ; Guardar BP anterior
    MOV8 R14, R15      ; Establecer nuevo BP
    SUBV8 R15, N        ; Reservar espacio para locales (N bytes)
```

Epílogo de función:

```
    MOV8 R00, valor    ; Valor de retorno (opcional)
    MOV8 R15, R14      ; Restaurar SP
    POP8 R14           ; Restaurar BP anterior
    RET                ; Retornar (pop dirección y saltar)
```

Llamada a función:

```
    PUSH8 arg_n        ; Push argumentos en orden inverso
    ...
    PUSH8 arg_1
    CALL funcion        ; Push PC+4 y saltar
    ADDV8 R15, N*8      ; Limpiar argumentos del stack
```

2.8.3 Traducción de Construcciones del Lenguaje

Expresiones aritméticas: Recorrido postorden del AST con asignación de registros temporales

Estructuras de control: Generación de etiquetas y saltos condicionales

```
if (condicion) { ... } else { ... }
```

→

```
    <evaluar condicion en R00>
    CMP8 R00, 0
    JZ else_label
    <código then>
    JMP end_label
else_label:
    <código else>
end_label:
```

Acceso a arreglos: $\text{arr}[i] \rightarrow \text{dirección_base} + i * \text{tamaño_elemento}$

Punteros: - Operador `&`: Obtener dirección - Operador `*`: Indirección (LDIND/STIND)

Memoria dinámica: - `new` `Type`: Llamada a función de allocación - `delete` `ptr`: Llamada a función de liberación

2.9 Ensamblador, Enlazador y Cargador

2.9.1 Ensamblador

El ensamblador traduce código ensamblador a código máquina mediante dos pasadas:

Primera pasada: - Construir tabla de símbolos con direcciones de etiquetas - Calcular tamaños de instrucciones y datos - Detectar errores sintácticos

Segunda pasada: - Generar código máquina para cada instrucción - Resolver referencias a etiquetas locales - Marcar referencias externas para el enlazador - Generar archivo objeto (.o)

Archivos Objeto (.o): Resultado del ensamblado, contienen: - Cabecera con identificador mágico (HEXA-OBJ) - Tabla de símbolos (definidos y externos) - Tabla de relocalaciones - Secciones de código y datos

2.9.2 Enlazador (Linker)

El enlazador combina múltiples archivos objeto en un ejecutable unificado:

1. **Recolección de símbolos:** Tabla global de todos los símbolos definidos
2. **Resolución de símbolos:** Conectar referencias externas con definiciones
3. **Reubicación:** Ajustar direcciones según posición final en memoria
4. **Combinación de secciones:** Unir código y datos de múltiples módulos
5. **Generación del ejecutable:** Crear archivo .exe con punto de entrada

Tipos de relocalación: - **ABSOLUTE:** Direcciones absolutas fijas - **RELATIVE:** Desplazamientos relativos (para saltos) - **IMMEDIATE:** Valores inmediatos en instrucciones

Formato Ejecutable (.exe): Contiene: - Cabecera mágica (HEXACORE) - Punto de entrada (dirección inicial de ejecución) - Código relocado con direcciones resueltas - Datos inicializados

2.9.3 Cargador (Loader)

El cargador prepara el ejecutable para ejecución:

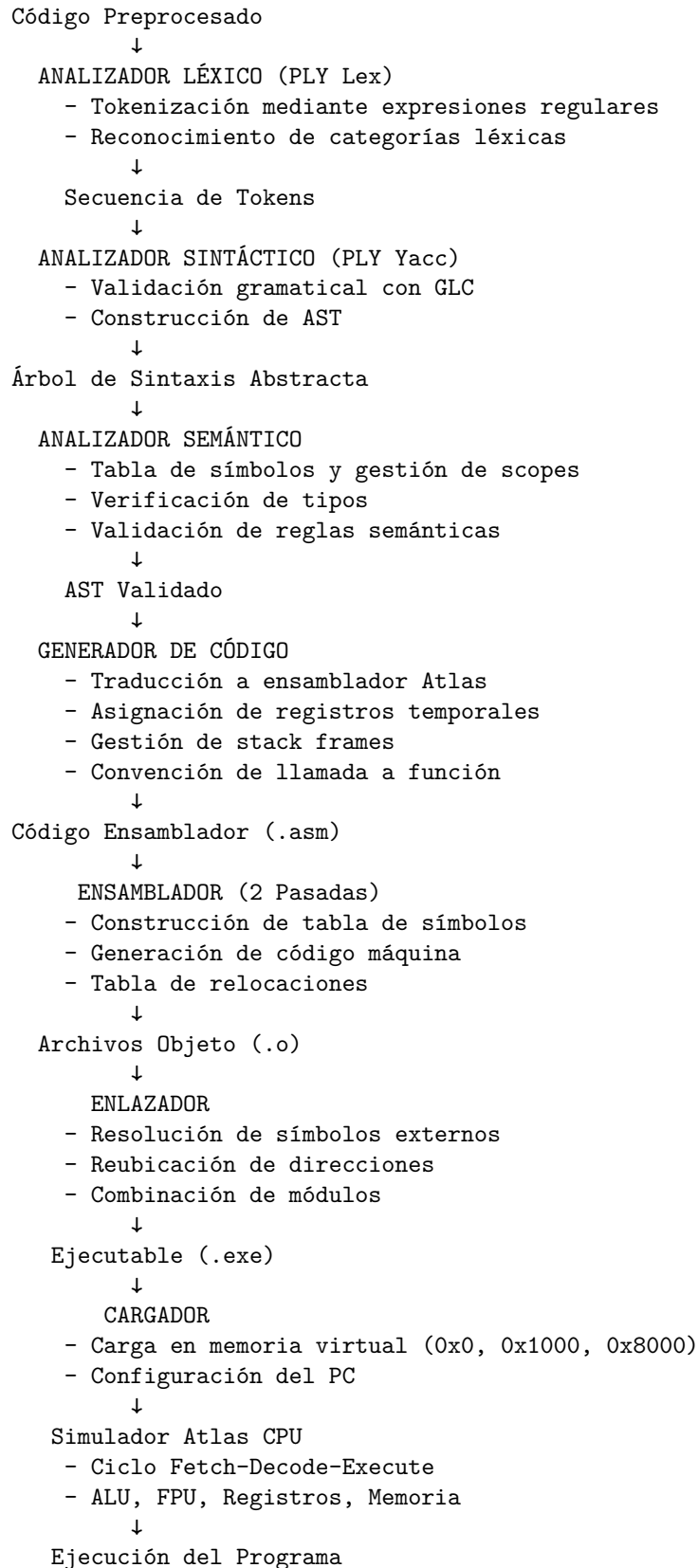
1. Leer y parsear archivo .exe
2. Asignar memoria virtual al proceso
3. Copiar código a memoria (desde 0x0)
4. Copiar datos globales (desde 0x1000)
5. Inicializar stack (en 0x8000)
6. Establecer PC al punto de entrada
7. Transferir control al programa

2.10 Pipeline de Compilación Completo del Sistema SPL

El Sistema de Procesamiento del Lenguaje (SPL) integra todas las fases estudiadas, desde el código fuente de alto nivel hasta la ejecución en el simulador Atlas:

Código Fuente SPL (.spl)

↓
PREPROCESADOR
- Expansión de `#include`
- Sustitución de `#define`
- Expansión de macros
↓



2.10.1 Integración y API del Compilador

El pipeline se integra en el módulo `compiler.py` que expone la siguiente API:

Función principal:

```
compile_code(code: str, debug: bool = False) -> tuple[AST, bool, list[str]]
```

Retorna una tupla con: - `ast`: Nodo raíz Program del AST (o None si hay errores) - `success`: bool indicando si la compilación fue exitosa - `errors`: Lista de mensajes de error (vacía si `success=True`)

Fases ejecutadas: 1. **Preprocesamiento:** `preprocess(code, base_path=".")` → código expandido 2.

Análisis Sintáctico: `parse(code)` → AST o None 3. **Análisis Semántico:** `SemanticAnalyzer().analyze(ast)` → bool + errores

Variante para archivos:

```
compile_file(filename: str, debug: bool = False) -> tuple[AST, bool, list[str]]
```

Ejemplo de uso:

```
from compiler.compiler import compile_code
```

```
code = """
funcion entero4 principal() {
    imprimir("Hola Atlas");
    retornar 0;
}
"""
```

```
ast, success, errors = compile_code(code)
if success:
    print("Compilación exitosa")
    # Proceder a generación de código
else:
    print("Errores encontrados:")
    for error in errors:
        print(f" - {error}")
```

2.10.2 Pipeline Completo - Integración

Este pipeline completo integra: - **Teoría de lenguajes formales** (gramáticas, autómatas) - **Teoría de compiladores** (análisis léxico, sintáctico, semántico, generación de código) - **Arquitectura de computadores** (modelo von Neumann, ISA, memoria, E/S) - **Ingeniería de software** (modularidad, testing, documentación)

3. Descripción del Problema

En el curso de Lenguajes de Programación de la Universidad Nacional de Colombia, se plantea como reto fundamental la construcción de un **sistema computacional completo y funcional** que demuestre la aplicación práctica de los conceptos teóricos estudiados. Este proyecto integrado abarca desde el diseño de hardware hasta la implementación de un compilador completo, pasando por todas las etapas del procesamiento de lenguajes.

3.1 Visión General del Proyecto

El proyecto se desarrolla en fases incrementales, donde cada etapa construye sobre la anterior:

3.1.1 Fase 1: Arquitectura y Simulador (Tareas #09 y #14)

Objetivo: Diseñar e implementar una máquina virtual que simule una computadora de 64 bits capaz de ejecutar programas en lenguaje ensamblador.

Componentes desarrollados:

1. **Arquitectura Atlas (64 bits):**
 - Conjunto de instrucciones (ISA) con 137+ instrucciones
 - 16 registros de propósito general (R00-R15)
 - Memoria de 64KB con direccionamiento por byte
 - Sistema de flags (Z, N, C, V)
 - Formatos de instrucción: OP, R, RR, RI, I
2. **Simulador de CPU:**
 - Motor de ejecución (ciclo fetch-decode-execute)
 - Unidad Aritmético-Lógica (ALU) con operaciones enteras y flotantes
 - Unidad de Punto Flotante (FPU)
 - Sistema de interrupciones y E/S mapeada en memoria
3. **Sistema de Memoria Virtual:**
 - Direccionamiento de 44 bits (16 TiB teóricos)
 - Gestión de espacio de direcciones
 - Segmentación: código, datos globales, stack
4. **Interfaz Gráfica (GUI):**
 - Editor de código ensamblador
 - Visualizador de registros y memoria en tiempo real
 - Control de ejecución (paso a paso, continua, puntos de ruptura)
 - Visor de RAM con búsqueda y limpieza

3.1.2 Fase 2: Preprocesador y Análisis Léxico (Taller #01)

Objetivo: Desarrollar las primeras fases del Sistema de Procesamiento del Lenguaje (SPL), incluyendo preprocesador, analizador léxico, ensamblador y enlazador-cargador.

Componentes desarrollados:

1. **Preprocesador:**
 - Directivas de inclusión (`#include`) con búsqueda en rutas
 - Definiciones de constantes (`#define`)
 - Expansión de macros (`.macro/.endmacro`) con parámetros
 - Procesamiento condicional
2. **Analizador Léxico (PLY Lex):**
 - Reconocimiento de tokens: instrucciones, registros, literales
 - Categorías léxicas: mnemónicos, identificadores, números, etiquetas
 - Manejo de comentarios (línea y bloque)
 - Validación de formatos (hexadecimal, binario, decimal)

3. Ensamblador:

- Análisis sintáctico de código ensamblador
- Dos pasadas: resolución de etiquetas y generación de código
- Codificación de instrucciones según formato
- Generación de archivos objeto (.o)

4. Enlazador-Cargador:

- Combinación de múltiples archivos objeto
- Resolución de símbolos externos
- Reubicación de direcciones
- Generación de ejecutables (.exe)
- Carga en memoria virtual para ejecución

3.1.3 Fase 3: Compilador Completo (Taller #02 - Actual)

Objetivo: Completar el compilador SPL implementando análisis sintáctico, semántico y generación de código para traducir programas de alto nivel a ensamblador Atlas.

Nuevos componentes desarrollados:

1. Analizador Sintáctico (Parser):

- Construcción del parser con PLY Yacc
- Gramática libre de contexto en notación E-BNF
- Tabla de precedencia y asociatividad de operadores
- Construcción de Árbol de Sintaxis Abstracta (AST)
- Manejo de errores sintácticos

2. Analizador Semántico:

- Tabla de símbolos con gestión de scopes anidados
- Validación de tipos (compatibilidad, conversiones)
- Resolución de identificadores (variables, funciones, estructuras)
- Verificación de control de flujo (break/continue en loops, return en funciones)
- Validación de inicialización de variables

3. Generador de Código:

- Traducción de AST a código ensamblador Atlas
- Asignación de variables a memoria y registros
- Gestión de stack frame (parámetros, locales, retornos)
- Convención de llamada a funciones con prólogo/epílogo
- Generación de código para expresiones con registros temporales
- Soporte para tipos enteros y punto flotante

El problema central del proyecto completo radica en construir un sistema integrado que permita escribir programas en un lenguaje de alto nivel (SPL), compilarlos automáticamente a código ensamblador, ensamblar y enlazar los módulos resultantes, y ejecutarlos en un simulador de CPU funcional con visualización de todo el proceso.

3.2 Tema del Proyecto: Diseño e Implementación de Tipos de Datos en Lenguajes de Programación

El proyecto se enfoca específicamente en el tema asignado “**Diseño e implementación de tipos de datos en lenguajes de programación**”, el cual se aborda de manera integral a través de los siguientes componentes:

3.2.1 Sistema de Tipos Completo

El lenguaje SPL implementa un **sistema de tipos robusto y completo** que incluye:

Tipos primitivos nativos (11 tipos base): - **Enteros con tamaño explícito:** entero1 (8 bits), entero2 (16 bits), entero4 (32 bits), entero8 (64 bits) - **Punto flotante IEEE 754:** flotante (32 bits), doble

(64 bits) - **Tipos adicionales:** `caracter` (8 bits), `cadena` (puntero a UTF-8), `booleano` (8 bits), `vacio` (funciones sin retorno) - **Modificadores de signo:** `con_signo`, `sin_signo` aplicables a tipos enteros

Tipos compuestos estructurados: - **Estructuras** (`struct`): Tipos de datos definidos por el usuario con múltiples campos - **Arreglos:** Colecciones homogéneas con dimensiones estáticas o dinámicas - **Punteros:** Referencias tipadas con operadores de indirección (`*`) y dirección (`&`) - **Tipos función:** Firmas de funciones con parámetros tipados y tipo de retorno

3.2.2 Implementación en el Compilador

El compilador SPL materializa el diseño de tipos mediante:

1. Análisis léxico y sintáctico de tipos: - Reconocimiento de palabras reservadas de tipos (`entero4`, `flotante`, `doble`, etc.) - Parsing de declaraciones con tipos complejos (punteros, arreglos, estructuras) - Construcción del AST con nodos tipados (`VarDecl`, `StructDecl`, `TypeSpec`)

2. Sistema de verificación de tipos (Type Checker): - **Compatibilidad de tipos:** Validación de operaciones permitidas entre tipos - **Reglas de coerción:** Conversiones implícitas seguras (ej: `entero4` → `flotante`) - **Prevención de errores:** Detección de incompatibilidades en asignaciones y operaciones - **Tabla de símbolos tipada:** Cada identificador registra su tipo completo

3. Generación de código con conciencia de tipos: - **Instrucciones específicas por tipo:** Uso de sufijos de tamaño (`ADD4`, `FADD8`) según el tipo - **Alineación de memoria:** Cálculo de offsets y tamaños según requisitos de cada tipo - **Convenciones de paso de parámetros:** Manejo diferenciado de tipos primitivos vs. compuestos - **Gestión de strings:** Área dedicada `string_data_base` con inicialización byte-a-byte UTF-8

3.2.3 Representación en Memoria de Tipos

Cada tipo tiene una representación en memoria bien definida:

Tipo	Tamaño (bytes)	Alineación	Codificación
<code>entero1</code>	1	1	Complemento a 2
<code>entero2</code>	2	2	Complemento a 2
<code>entero4</code>	4	4	Complemento a 2
<code>entero8</code>	8	8	Complemento a 2
<code>flotante</code>	4	4	IEEE 754 single
<code>doble</code>	8	8	IEEE 754 double
<code>booleano</code>	1	1	0=falso, 1=verdadero
<code>caracter</code>	1	1	ASCII/UTF-8
<code>cadena</code>	8	8	Puntero a bytes UTF-8 + NULL
<code>puntero</code>	8	8	Dirección de 64 bits
<code>struct</code>	variable	mayor del campo	Layout secuencial con padding
<code>arreglo[N]</code>	N * tamaño_elemento	alineación del elemento	Elementos contiguos

3.2.4 Inicialización Determinista de Datos

El sistema garantiza inicialización reproducible de datos globales y literales:

Proceso de inicialización en tres fases:

- 1. Recolección en CodeGenerator** (`src/compiler/code_generator.py`):
 - Recorrido completo del AST incluyendo todas las ramas (`if`, `elif`, `else`)
 - Extracción de literales de cadena con sus valores UTF-8
 - Asignación de direcciones en área `string_data_base` (offset base 0x18000)
 - Emisión de secuencias de inicialización byte-a-byte:


```

; Inicializar "Hola" en 0x18010
MOVV1 R01, 72      ; 'H'
STORE1 R01, 0x18010
MOVV1 R01, 111     ; 'o'
STORE1 R01, 0x18011
MOVV1 R01, 108     ; 'l'
STORE1 R01, 0x18012
MOVV1 R01, 97      ; 'a'
STORE1 R01, 0x18013
MOVV1 R01, 0       ; NULL
STORE1 R01, 0x18014

```

2. Carga determinista en Loader (src/compiler/Loader.py):

- Interpretación de directivas `.DATA` durante la fase de carga
- Escritura byte-a-byte en memoria antes de la ejecución
- Registro de símbolos globales con direcciones absolutas
- Registro de metadatos `.LOCAL_REL` para variables locales/parámetros:
`.LOCAL_REL -20 8 nota ; FUNC=principal`
`.LOCAL_REL -28 4 contador ; FUNC=principal`

3. Resolución en tiempo de ejecución (Memory + CPU):

- Variables globales: acceso directo por dirección absoluta
- Variables locales: acceso relativo a BP (R14) usando metadatos `.LOCAL_REL`
- Strings: acceso indirecto via punteros inicializados en `string_data_base`
- Logging legible: nombres de variables en trazas de `STORE/LOAD` usando metadata

3.2.5 Validación del Sistema de Tipos

La implementación se valida mediante:

- **Detección de errores de tipo en tiempo de compilación:** Incompatibilidades reportadas antes de generar código
- **Tests exhaustivos:** Suite de pruebas en `src/tests/` cubriendo todos los tipos
- **Ejemplos funcionales:** Algoritmos complejos (Euclides, matrices) que ejercitan el sistema de tipos
- **Ejecución correcta:** Simulador que respeta semántica de tipos en operaciones ALU/FPU

Conclusión: El proyecto implementa de manera completa el tema asignado, demostrando diseño, implementación, verificación y ejecución de un sistema de tipos robusto integrado en un compilador funcional.

3.3 Requisitos Específicos del Taller #02

Habiendo completado la arquitectura Atlas (Tareas #09 y #14) y el preprocesador/ensamblador (Taller #01), el **Taller #02** establece los siguientes requerimientos para finalizar el compilador SPL:

3.3.1 Definición Formal de Gramática E-BNF

Especificar la gramática completa del lenguaje SPL en notación **E-BNF** (Extended Backus-Naur Form) que incluya:

- **Paradigma imperativo:** Estructuras de control (if/else), loops (while, for), secuencia
- **Tipos de datos abstractos:** Estructuras (struct), punteros, arreglos
- **Variables y constantes:** Declaración global y local con inicialización opcional
- **Funciones:** Declaración, parámetros, retorno de valores, recursión
- **Expresiones:** Aritmética, lógica, relacional, bitwise con precedencia y asociatividad correcta
- **Gestión de memoria dinámica:** Operadores `new` y `delete`
- **Reglas léxicas:** Identificadores, literales (enteros, flotantes, cadenas), comentarios

La gramática debe ser **completa, no ambigua y procesable** por herramientas automáticas.

3.3.2 Ajustes al Analizador Léxico

Extender el lexer del Taller #01 para reconocer tokens de alto nivel:

- **Palabras reservadas:** `function, if, else, while, for, return, struct, new, delete, break, continue`
- **Operadores adicionales:** Lógicos (`&&, ||, !`), relacionales (`<, >, ==, !=, <=, >=`), bitwise (`&, |, ^, ~`), asignación compuesta (`+=, -=, *=, /=`)
- **Tipos de datos:** `int2, int4, int8, float, double, char, string, bool, void`
- **Literales flotantes:** Reconocimiento de notación decimal y científica
- **Operadores de punteros:** `*` (indirección), `&` (dirección)

3.3.3 Diagramas de Sintaxis Ferroviarios (BottleCaps)

Generar visualizaciones gráficas completas usando <https://www.bottlecaps.de/rr/ui>:

- **Railroad diagrams** para todas las producciones sintácticas
- Archivo `.ebnf` validado y procesable por BottleCaps
- Diagramas organizados por categorías (declaraciones, sentencias, expresiones)
- Integración de diagramas en documentación técnica

3.3.4 Interpretación Semántica Detallada

Para **cada categoría de producciones**, documentar:

- **Significado semántico:** Qué representa cada construcción en el modelo computacional
- **Acciones del compilador:** Qué debe hacer el parser/semantic analyzer al reconocer cada patrón
- **Construcción del AST:** Qué nodos se crean y cómo se estructuran
- **Validaciones semánticas:** Verificaciones de tipos, scopes, inicialización, control de flujo
- **Generación de código:** Estrategia de traducción a ensamblador Atlas

Ejemplo: Para la producción `if_stmt ::= 'si' '(' expression ')' statement ('si_no_si' '(' expression ')' statement) * ('si_no' statement) ?`:

- **Interpretación:** Sentencia condicional que evalúa una expresión booleana y ejecuta diferentes bloques según el resultado
- **Acción interna:** ### 3.3.5 Implementación del Parser con PLY Yacc

Desarrollar el analizador sintáctico completo usando **PLY (Python Lex-Yacc)**:

- **Tabla de precedencia y asociatividad:** Resolver ambigüedades de operadores
- **Reglas de producción completas:** Implementar todas las construcciones de la gramática
- **Construcción del AST:** Generar árbol sintáctico con nodos tipados (`ast_nodes.py`)
- **Manejo robusto de errores:** Reportar errores sintácticos con línea/columna

3.3.6 Análisis Semántico y Generación de Código

Implementar las fases finales del compilador:

Analizador Semántico (`semantic_analyzer.py`): - Tabla de símbolos con gestión de scopes anidados (global, función, bloque, struct) - Verificación de tipos y compatibilidad - Resolución de identificadores (variables, funciones, estructuras) - Validación de control de flujo (`break/continue` en loops, `return` en funciones)

Generador de Código (`code_generator.py`): - Traducción de AST a ensamblador Atlas - Asignación de variables a memoria y registros temporales - Gestión de stack frame con convención de llamada estándar - Soporte para tipos abstractos: estructuras, punteros, memoria dinámica (`new/delete`) - **Soporte de aritmética flotante:** Instrucciones `FADD4/8, FSUB4/8, FMUL4/8, FDIV4/8`

3.3.7 Validación con Programas de Prueba

Proporcionar **al menos 5 programas SPL** representativos:

1. **Algoritmo de Euclides** (MCD) - Obligatorio, demuestra recursión
2. Programa con funciones, parámetros y tipos flotantes
3. Programa con estructuras de control (if/else, while, for)
4. Programa con estructuras de datos (struct) y punteros
5. Programa que combine características avanzadas del lenguaje

Para **al menos 2 programas**, mostrar: - Prueba de escritorio con árbol de derivación sintáctica - Compilación exitosa y ejecución en el simulador Atlas - Verificación de correctitud de resultados

3.4 Desafíos Técnicos del Taller #02

La implementación del compilador SPL presenta los siguientes retos técnicos:

3.4.1 Fase de Análisis Sintáctico

1. **Diseño de gramática sin ambigüedades:** Resolver conflictos shift/reduce mediante precedencia y asociatividad de operadores
2. **Manejo de declaraciones complejas:** Distinguir entre declaraciones de variables, funciones y estructuras
3. **Parsing de expresiones anidadas:** Soportar precedencia multi-nivel con asociatividad correcta

3.4.2 Fase de Análisis Semántico

4. **Gestión de scopes anidados:** Implementar pila de tablas de símbolos para global/función/bloque/struct
5. **Sistema de tipos robusto:** Validar compatibilidad de tipos, conversiones implícitas, equivalencia estructural de structs
6. **Resolución de sobrecarga:** Diferenciar variables locales vs. globales, campos de struct vs. variables
7. **Validación de control de flujo:** Verificar que break/continue estén dentro de loops, return dentro de funciones

3.4.3 Fase de Generación de Código

8. **Asignación de registros temporales:** Gestionar pool de registros disponibles (R00-R09 para temporales)
9. **Gestión de stack frame:** Implementar convención de llamada (paso de parámetros, preservación de BP/SP, retorno)
10. **Traducción de expresiones complejas:** Generar código óptimo para expresiones aritméticas/lógicas con evaluación correcta
11. **Aritmética de punteros:** Calcular offsets considerando tamaño de tipos apuntados
12. **Memoria dinámica:** Integrar operadores new/delete con sistema de memoria del simulador
13. **Soporte de tipos flotantes:** Seleccionar instrucciones correctas (enteras vs. F-prefixed) según tipo de operandos

3.5 Gramática E-BNF del Lenguaje SPL

A continuación se presenta la gramática completa del lenguaje SPL en notación E-BNF. Esta gramática es compatible con la herramienta Railroad Diagram Generator de BottleCaps (<https://www.bottlecaps.de/rr/ui>).

Archivo: Documentacion/Taller2/gramatica/gramatica.ebnf

3.5.1 Estructura del Programa

```
/* Programa principal */
program ::= declaration_list

declaration_list ::= declaration+
```

declaration ::= function_decl | struct_decl | var_decl_stmt

3.5.2 Declaraciones de Función

function_decl ::= normal_function_decl | extern_function_decl

normal_function_decl ::= 'funcion' type ID '(' param_list? ')' block

extern_function_decl ::= 'externo' 'funcion' type ID '(' param_list? ')' ';'

param_list ::= param (',' param)*

param ::= type ID

3.5.3 Declaraciones de Estructura

struct_decl ::= 'estructura' ID '{' member_list '}' ';'

member_list ::= member+

member ::= type ID ';'

3.5.4 Declaraciones de Variable

var_decl_stmt ::= var_decl ';'

var_decl ::= type ID ('=' expression)?
 | type_base array_dims ID ('=' expression)?
 | 'constante' type ID '=' expression

array_dims ::= '[' ENTERO ']'
 | array_dims '[' ENTERO ']'

3.5.5 Sistema de Tipos

type ::= type_base
 | type '*'

type_base ::= 'vacio'
 | 'entero2'
 | 'entero4'
 | 'entero8'
 | 'caracter'
 | 'cadena'
 | 'flotante'
 | 'doble'
 | 'booleano'
 | 'con_signo'
 | 'sin_signo'
 | ID

3.5.6 Sentencias

statement ::= var_decl_stmt
 | expr_stmt

```

| if_stmt
| while_stmt
| for_stmt
| return_stmt
| break_stmt
| continue_stmt
| print_stmt
| block

```

```
block ::= '{' statement_list? '}'
```

```
statement_list ::= statement+
```

```
expr_stmt ::= expression? ';'

```

3.5.7 Sentencias de Control de Flujo

```
if_stmt ::= 'si' '(' expression ')' statement ( 'si_no_si' '(' expression ')' statement ) * ( 'si_no' st
```

```
while_stmt ::= 'mientras' '(' expression ')' statement
```

```
for_stmt ::= 'para' '(' for_init_opt ';' expr_opt ';' expr_opt ')' statement
```

```
for_init_opt ::= var_decl | expression | /* empty */
```

```
expr_opt ::= expression | /* empty */
```

```
return_stmt ::= 'retornar' expression? ';'

```

```
break_stmt ::= 'romper' ';'

```

```
continue_stmt ::= 'continuar' ';'

```

```
print_stmt ::= 'imprimir' '(' argument_list? ')' ';'

```

3.5.8 Expresiones

```
expression ::= assignment
```

```
assignment ::= logical (assignment_op assignment)?
```

```
assignment_op ::= '=' | '+=' | '-=' | '*=' | '/=' | '%='

```

```
logical ::= logical_or
```

```
logical_or ::= logical_and ('||' logical_and)*
```

```
logical_and ::= bitwise_or ('&&' bitwise_or)*
```

```
bitwise_or ::= bitwise_xor ('|' bitwise_xor)*
```

```
bitwise_xor ::= bitwise_and ('^' bitwise_and)*
```

```
bitwise_and ::= equality ('&' equality)*

```

```

equality ::= relational (equality_op relational)*
equality_op ::= '==' | '!='

relational ::= additive (relational_op additive)*
relational_op ::= '<' | '<=' | '>' | '>='

additive ::= multiplicative (additive_op multiplicative)*
additive_op ::= '+' | '-'

multiplicative ::= unary (multiplicative_op unary)*
multiplicative_op ::= '*' | '/' | '%'

unary ::= unary_op unary | postfix
unary_op ::= '!' | '-' | '++' | '--' | '*' | '&'

postfix ::= primary postfix_op*
postfix_op ::= '++'
               | '--'
               | '.' ID
               | '->' ID
               | '[' expression ']'
               | '(' argument_list? ')'

argument_list ::= expression (',' expression)*

primary ::= ID
           | ENTERO
           | FLOT
           | CHARACTER
           | CADENA
           | '(' expression ')'
           | new_expr
           | delete_expr

new_expr ::= 'nuevo' type

delete_expr ::= 'eliminar' unary

```

3.5.9 Reglas Léxicas (Terminales)

```

ID ::= LETRA (LETRA | DIGITO | '_' )*

ENTERO ::= DECIMAL | HEXADECIMAL

DECIMAL ::= DIGITO+

HEXADECIMAL ::= '0' ('x' | 'X') HEX_DIGITO+

```

```

FLOT ::= DIGITO+ '.' DIGITO* EXPONENTE?
      | DIGITO* '.' DIGITO+ EXPONENTE?

EXPONENTE ::= ('e' | 'E') ('+' | '-')? DIGITO+

CARACTER ::= '"' (ESCAPE_CHAR | CHAR_NO_ESPECIAL) '"'

CADENA ::= '"' (ESCAPE_CHAR | STRING_CHAR)* '"'

ESCAPE_CHAR ::= '\\' ('n' | 't' | 'r' | '\\' | '"' | "'" | '0')

LETRA ::= [a-zA-Z]

DIGITO ::= [0-9]

HEX_DIGITO ::= [0-9a-fA-F]

CHAR_NO_ESPECIAL ::= #x20-#x26 | #x28-#x5B | #x5D-#x10FFFF /* cualquier carácter excepto ' y \ */

STRING_CHAR ::= #x20-#x21 | #x23-#x5B | #x5D-#x10FFFF /* cualquier carácter excepto " y \ */

```

3.5.10 Palabras Reservadas

El lenguaje SPL reconoce las siguientes **25 palabras reservadas**:

Categoría	Palabras
Control de flujo (7)	si, si_no, mientras, para, retornar, romper, continuar
Declaraciones (4)	funcion, estructura, externo, constante
Tipos primitivos (11)	vacio, entero2, entero4, entero8, flotante, doble, caracter, cadena, booleano, con_signo, sin_signo
Gestión de memoria (2)	nuevo, eliminar
Entrada/Salida (1)	imprimir

3.6 Diagramas de Sintaxis (BottleCaps)

Los diagramas de sintaxis ferroviaria (railroad diagrams) proporcionan una representación visual de la gramática que facilita la comprensión de las estructuras del lenguaje.

3.6.1 Herramienta Utilizada

Railroad Diagram Generator: <https://www.bottlecaps.de/rr/ui>

Esta herramienta genera automáticamente diagramas de sintaxis a partir de la especificación E-BNF.

3.6.2 Estructura del Programa

1. program:

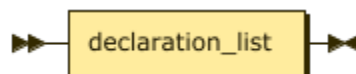


Figura 1: program

`program ::= declaration_list`

2. declaration_list:

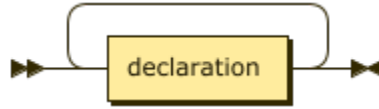


Figura 2: declaration_list

`declaration_list ::= declaration+`

3. declaration:

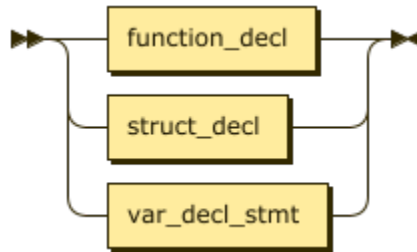


Figura 3: declaration

`declaration ::= function_decl | struct_decl | var_decl_stmt`

3.6.3 Declaraciones de Función

4. function_decl:

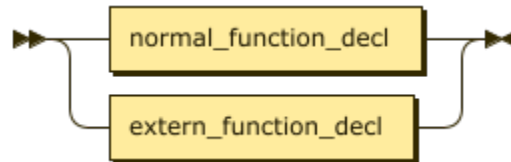


Figura 4: function_decl

`function_decl ::= normal_function_decl | extern_function_decl`

5. normal_function_decl:

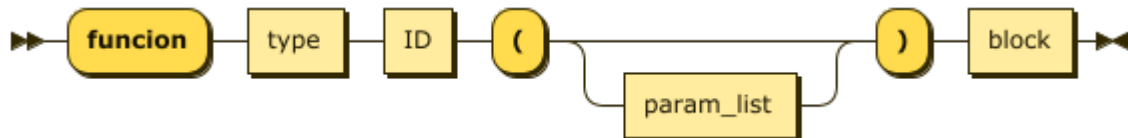


Figura 5: normal_function_decl

`normal_function_decl ::= 'funcion' type ID '(' param_list? ')' block`

6. extern_function_decl:

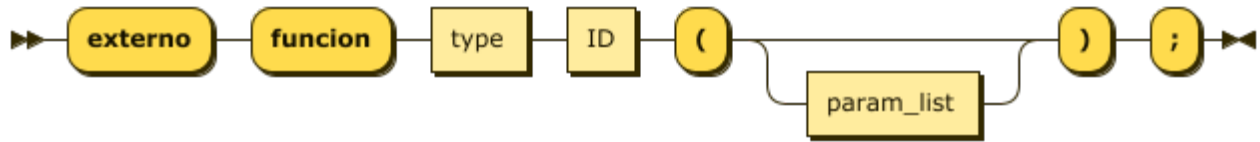


Figura 6: extern_function_decl

extern_function_decl ::= 'externo' 'funcion' type ID '(' param_list? ')' ';' ;

7. param_list:

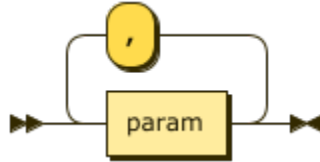


Figura 7: param_list

param_list ::= param (',' param)*

8. param:

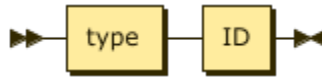


Figura 8: param

param ::= type ID

3.6.4 Declaraciones de Estructura

9. struct_decl:

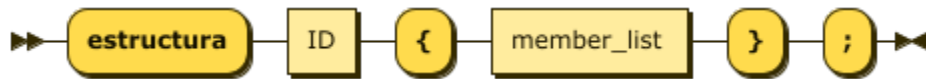


Figura 9: struct_decl

struct_decl ::= 'estructura' ID '{' member_list '}' ';' ;

10. member_list:

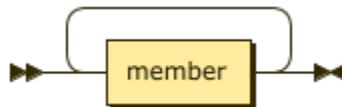


Figura 10: member_list

member_list ::= member+

11. member:

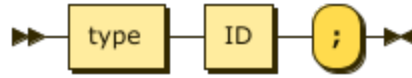


Figura 11: member

member ::= type ID ';' ;

3.6.5 Declaraciones de Variable

12. var_decl_stmt:

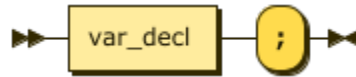


Figura 12: var_decl_stmt

var_decl_stmt ::= var_decl ';' ;

13. var_decl:

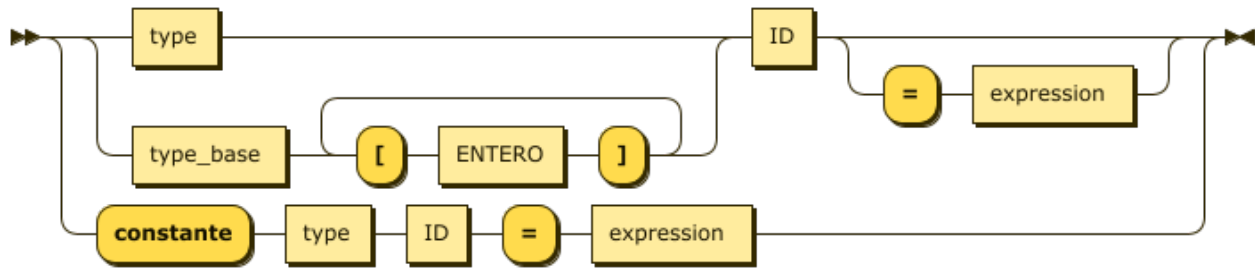


Figura 13: var_decl

var_decl ::= (type | type_base ('[' ENTERO ']')+) ID ('=' expression) ?
| 'constante' type ID '=' expression

3.6.6 Sistema de Tipos

14. type:

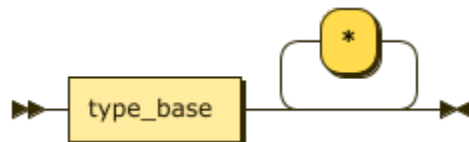


Figura 14: type

type ::= type_base '*' ;

15. type_base:

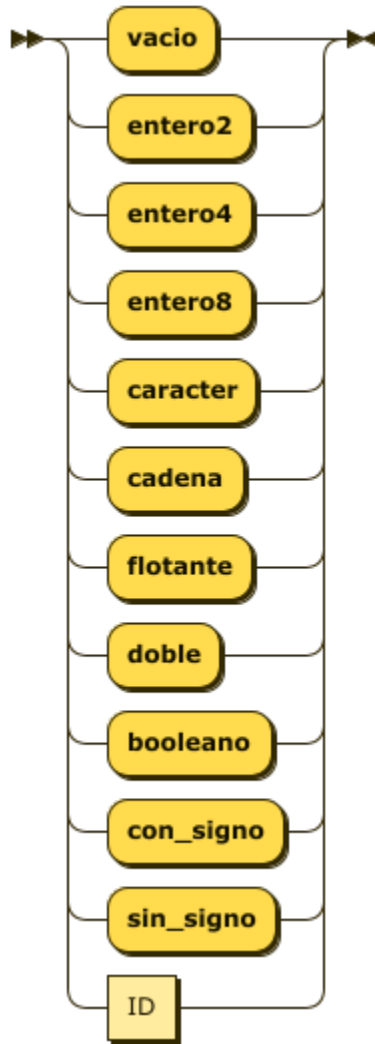


Figura 15: `type_base`

```
type_base ::= 'vacio' | 'entero2' | 'entero4' | 'entero8'  
           | 'caracter' | 'cadena' | 'flotante' | 'doble'  
           | 'booleano' | 'con_signo' | 'sin_signo' | ID
```

3.6.7 Sentencias

16. statement:

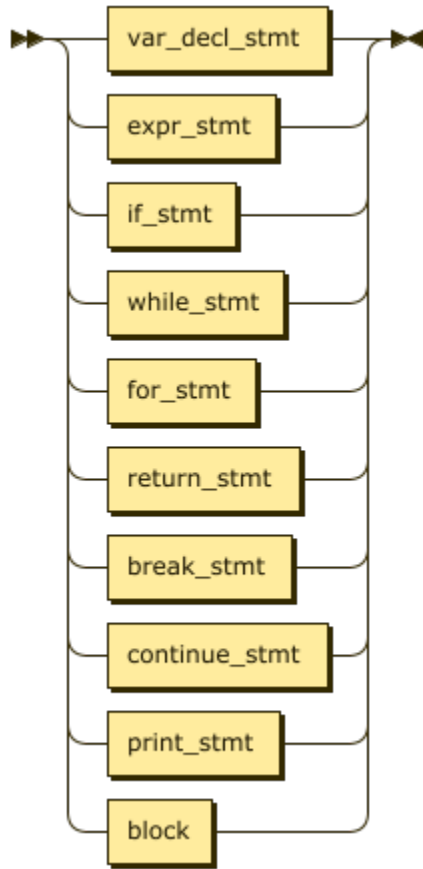


Figura 16: statement

```
statement ::= var_decl_stmt | expr_stmt | if_stmt | while_stmt
           | for_stmt | return_stmt | break_stmt | continue_stmt
           | print_stmt | block
```

17. block:

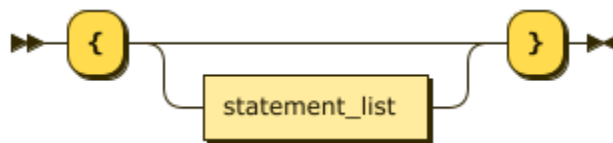


Figura 17: block

```
block ::= '{' statement_list? '}'
```

18. statement_list:

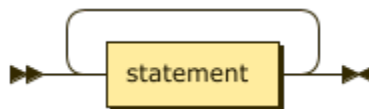


Figura 18: statement_list

```
statement_list ::= statement+
```

19. expr_stmt:

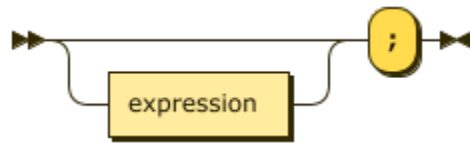


Figura 19: expr_stmt

expr_stmt ::= expression? ';' ;

3.6.8 Sentencias de Control de Flujo

20. if_stmt:

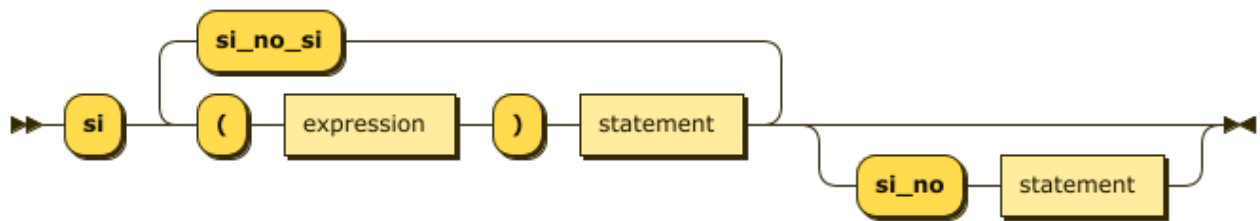


Figura 20: if_stmt

if_stmt ::= 'si' '(' expression ')' statement ('si_no_si' '(' expression ')' statement) * ('si_no' sta

21. while_stmt:



Figura 21: while_stmt

while_stmt ::= 'mientras' '(' expression ')' statement

22. for_stmt:



Figura 22: for_stmt

for_stmt ::= 'para' '(' for_init_opt ';' expr_opt ';' expr_opt ')' statement

23. for_init_opt:

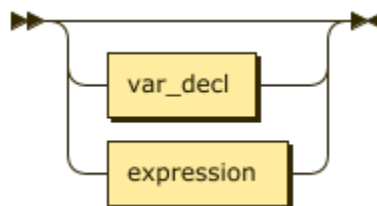


Figura 23: for_init_opt

`for_init_opt ::= (var_decl | expression)?`

24. expr_opt:

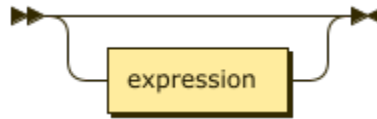


Figura 24: expr_opt

`expr_opt ::= expression?`

25. return_stmt:

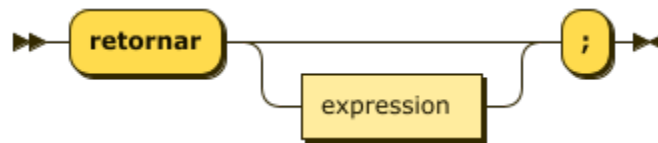


Figura 25: return_stmt

`return_stmt ::= 'retornar' expression? ';'`

26. break_stmt:



Figura 26: break_stmt

`break_stmt ::= 'romper' ';'`

27. continue_stmt:



Figura 27: continue_stmt

`continue_stmt ::= 'continuar' ';'`

28. print_stmt:

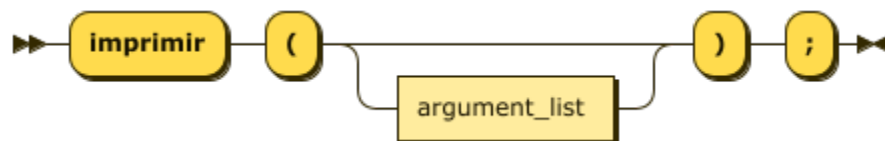


Figura 28: print_stmt

`print_stmt ::= 'imprimir' '(' argument_list? ')' ';'`

3.6.9 Expresiones

29. expression:

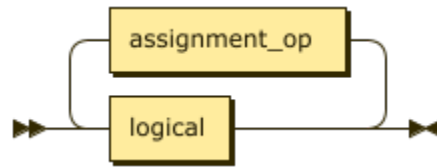


Figura 29: expression

`expression ::= logical (assignment_op logical)*`

30. assignment_op:

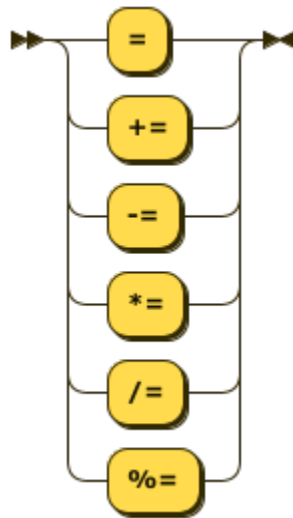


Figura 30: assignment_op

`assignment_op ::= '=' | '+= ' | '-=' | '*=' | '/=' | '%='`

31. logical:

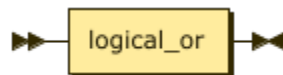


Figura 31: logical

`logical ::= logical_or`

32. logical_or:

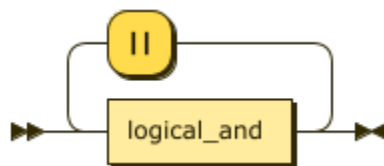


Figura 32: logical_or

```
logical_or ::= logical_and ( '||' logical_and )*
```

33. logical_and:

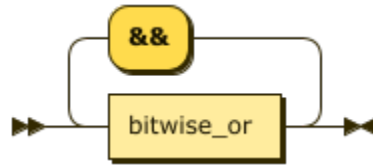


Figura 33: logical_and

```
logical_and ::= bitwise_or ( '&&' bitwise_or )*
```

34. bitwise_or:

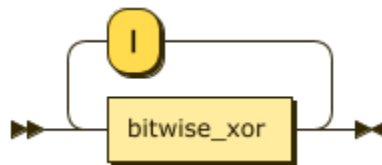


Figura 34: bitwise_or

```
bitwise_or ::= bitwise_xor ( '|' bitwise_xor )*
```

35. bitwise_xor:



Figura 35: bitwise_xor

```
bitwise_xor ::= bitwise_and ( '^' bitwise_and )*
```

36. bitwise_and:

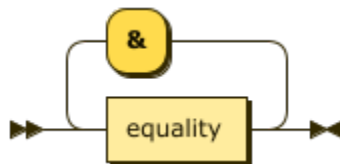


Figura 36: bitwise_and

```
bitwise_and ::= equality ( '&' equality )*
```

37. equality:

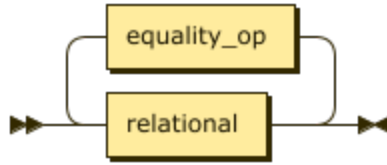


Figura 37: equality

`equality ::= relational (equality_op relational)*`

38. equality_op:

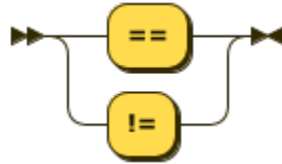


Figura 38: equality_op

`equality_op ::= '==' | '!='`

39. relational:

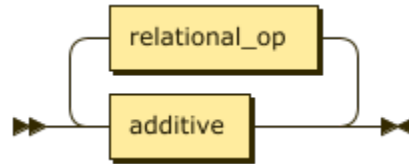


Figura 39: relational

`relational ::= additive (relational_op additive)*`

40. relational_op:

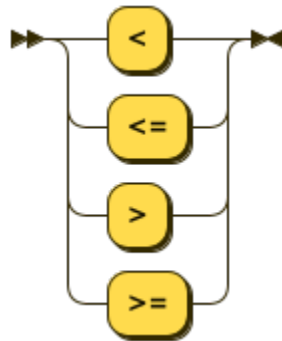


Figura 40: relational_op

`relational_op ::= '<' | '<=' | '>' | '>='`

41. additive:

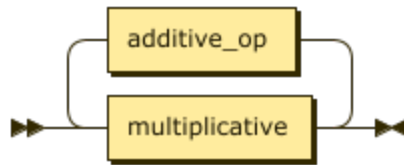


Figura 41: additive

`additive ::= multiplicative (additive_op multiplicative)*`

42. additive_op:

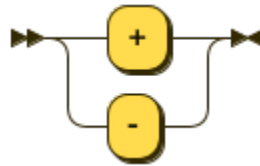


Figura 42: additive_op

`additive_op ::= '+' | '-'`

43. multiplicative:

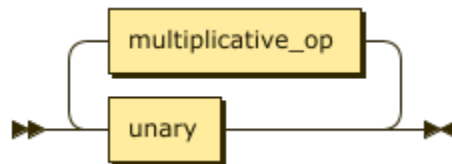


Figura 43: multiplicative

`multiplicative ::= unary (multiplicative_op unary)*`

44. multiplicative_op:

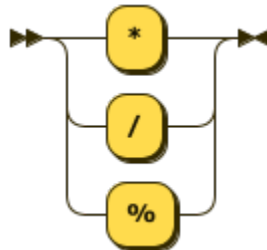


Figura 44: multiplicative_op

`multiplicative_op ::= '*' | '/' | '%'`

45. unary:

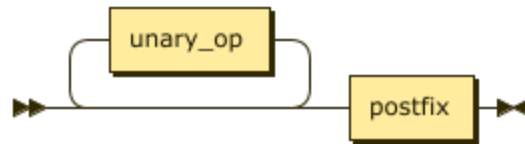


Figura 45: unary

`unary ::= unary_op* postfix`

46. unary_op:

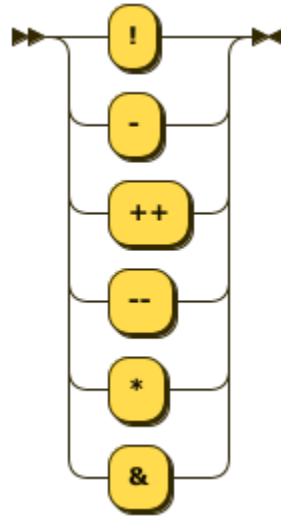


Figura 46: unary_op

`unary_op ::= '!' | '-' | '++' | '--' | '*' | '&'`

47. postfix:

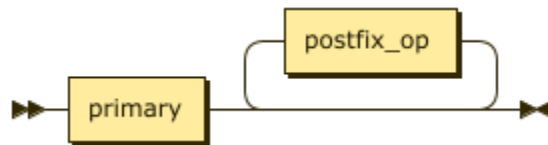


Figura 47: postfix

`postfix ::= primary postfix_op*`

48. postfix_op:

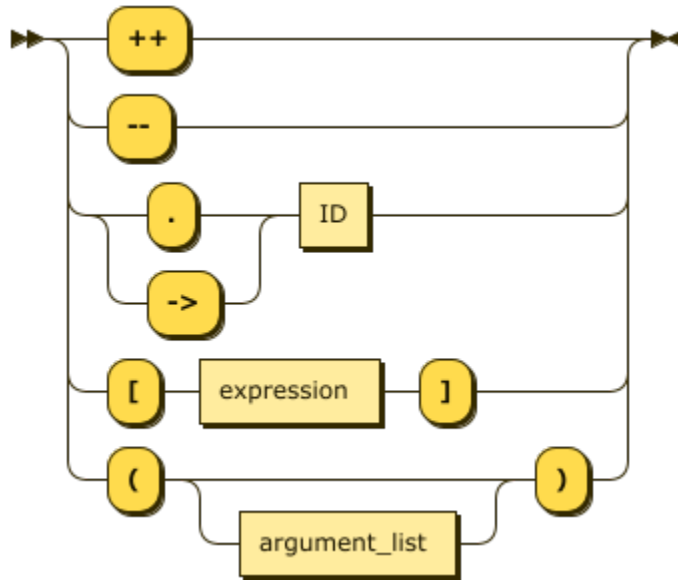


Figura 48: postfix_op

```
postfix_op ::= '++' | '--' | ( '.' | '->' ) ID
            | '[' expression ']' | '(' argument_list? ')'
```

49. argument_list:

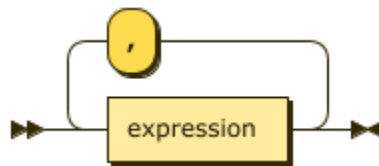


Figura 49: argument_list

```
argument_list ::= expression ( ',' expression )*
```

50. primary:

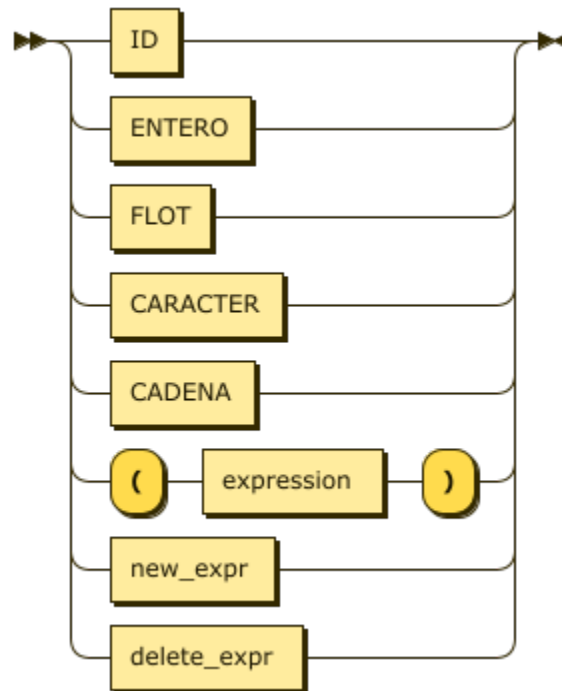


Figura 50: primary

primary ::= ID | ENTERO | FLOT | CARACTER | CADENA
 | '(' expression ')' | new_expr | delete_expr

3.6.10 Gestión de Memoria

51. new_expr:



Figura 51: new_expr

new_expr ::= 'nuevo' type

52. delete_expr:



Figura 52: delete_expr

delete_expr ::= 'eliminar' unary

3.6.11 Reglas Léxicas (Terminales)

53. ID:

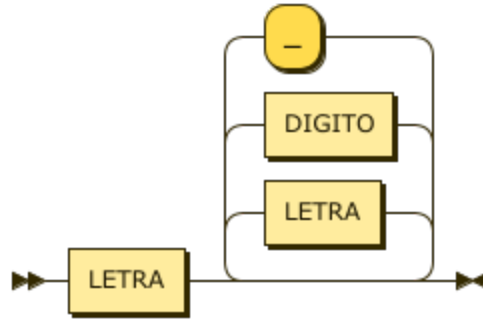


Figura 53: ID

ID ::= LETRA (LETRA | DIGITO | '_')*

54. ENTERO:

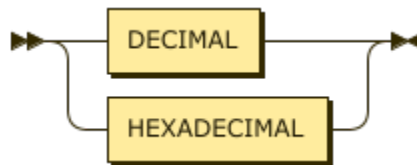


Figura 54: ENTERO

ENTERO ::= DECIMAL | HEXADECIMAL

55. DECIMAL:



Figura 55: DECIMAL

DECIMAL ::= DIGITO+

56. HEXADECIMAL:

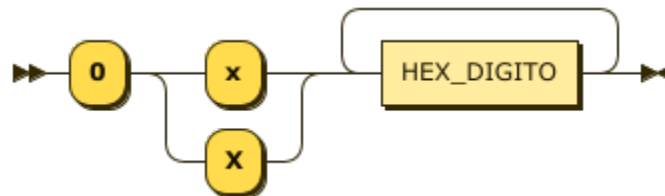


Figura 56: HEXADECIMAL

HEXADECIMAL ::= '0' ('x' | 'X') HEX_DIGITO+

57. FLOT:

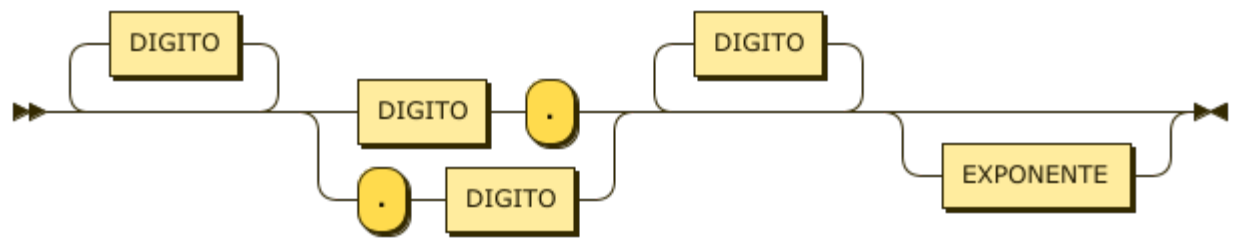


Figura 57: FLOT

$\text{FLOT} ::= \text{DIGITO}^* (\text{DIGITO} \text{'.''} | \text{'.''} \text{DIGITO}) \text{DIGITO}^* \text{EXPONENTE}?$

58. EXPONENTE:

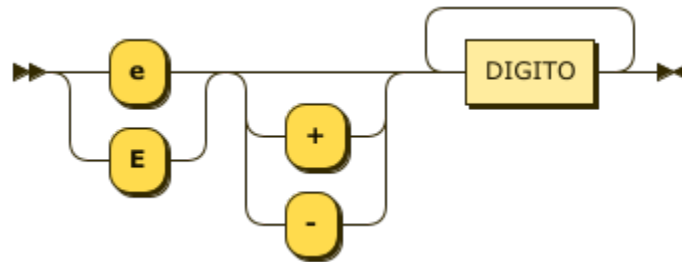


Figura 58: EXPONENTE

$\text{EXPONENTE} ::= (\text{'e'} | \text{'E'}) (\text{'+'} | \text{'-' })? \text{DIGITO}^+$

59. CARACTER:

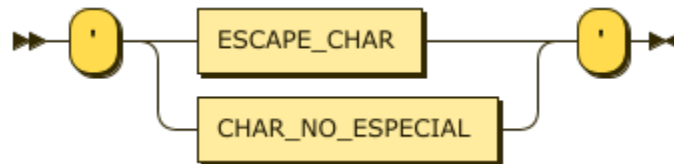


Figura 59: CARACTER

$\text{CARACTER} ::= \text{' ' } (\text{ESCAPE_CHAR} | \text{CHAR_NO_ESPECIAL}) \text{' '}$

60. CADENA:

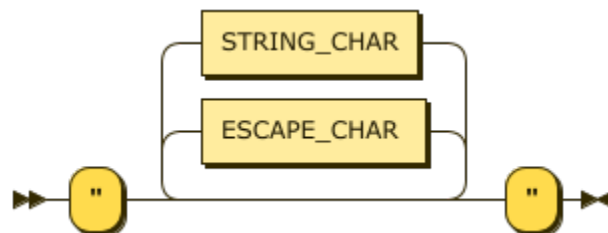


Figura 60: CADENA

$\text{CADENA} ::= \text{' ' } (\text{ESCAPE_CHAR} | \text{STRING_CHAR})^* \text{' '}$

61. ESCAPE_CHAR:

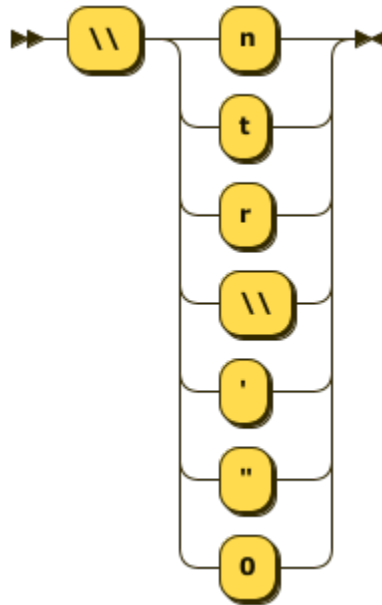


Figura 61: ESCAPE_CHAR

ESCAPE_CHAR ::= '\\\' ('n' | 't' | 'r' | '\\\' | '\"' | '\'' | '0')

62. LETRA:

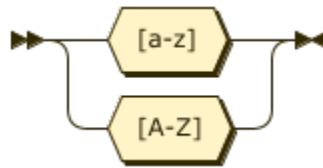


Figura 62: LETRA

LETRA ::= [a-zA-Z]

63. DIGITO:

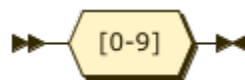


Figura 63: DIGITO

DIGITO ::= [0-9]

64. HEX_DIGITO:

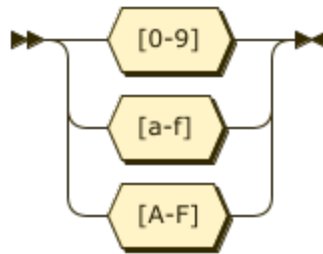


Figura 64: HEX_DIGITO

HEX_DIGITO ::= [0-9a-fA-F]

65. CHAR_NO_ESPECIAL:

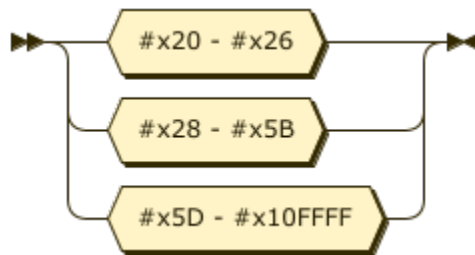


Figura 65: CHAR_NO_ESPECIAL

CHAR_NO_ESPECIAL ::= #x20 - #x26 | #x28 - #x5B | #x5D - #x10FFFF

66. STRING_CHAR:

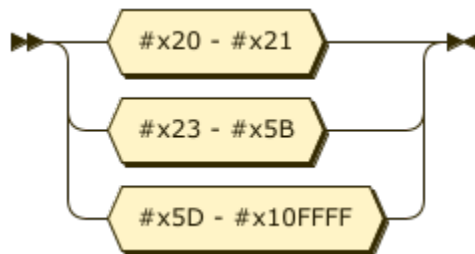


Figura 66: STRING_CHAR

STRING_CHAR ::= #x20 - #x21 | #x23 - #x5B | #x5D - #x10FFFF

3.6.12 Diagrama Completo de la Gramática



Figura 67: Diagrama completo generado por RR

Generado por Railroad Diagram Generator

3.6.13 Uso de los Diagramas

Los diagramas facilitan: - **Comprensión visual** de la estructura gramatical - **Identificación rápida** de construcciones opcionales vs obligatorias - **Documentación** clara para usuarios del lenguaje - **Validación**

de que la gramática no tiene ambigüedades - **Navegación** entre producciones relacionadas

Nota: El archivo completo `gramatica.ebnf` está disponible en el repositorio y puede ser copiado directamente a la herramienta BottleCaps para visualización interactiva.

3.7 Interpretación Semántica de las Producciones

Esta sección describe en lenguaje natural qué significa cada producción de la gramática y qué acciones semánticas debe realizar el compilador al reconocerlas.

1. `program ::= declaration_list`

- **Interpretación:** Un programa SPL consiste en una lista no vacía de declaraciones globales (funciones, estructuras, variables).
- **Acción Semántica:**
 1. Crear scope global
 2. Procesar cada declaración en orden
 3. Construir nodo AST raíz de tipo `Program`
 4. Validar que exista una función `principal()` como punto de entrada

2. `declaration_list ::= declaration+`

- **Interpretación:** Secuencia de una o más declaraciones globales.
- **Acción Semántica:**
 1. Procesar cada declaración secuencialmente
 2. Mantener tabla de símbolos global actualizada
 3. Construir lista de nodos AST para declaraciones

3. `declaration ::= function_decl | struct_decl | var_decl_stmt`

- **Interpretación:** Las declaraciones globales pueden ser funciones, estructuras de datos o variables globales.
- **Acción Semántica:**
 1. Identificar el tipo de declaración
 2. Invocar el manejador semántico correspondiente
 3. Agregar el símbolo a la tabla de símbolos global
 4. Verificar que no existan declaraciones duplicadas

4. `function_decl ::= normal_function_decl | extern_function_decl`

- **Interpretación:** Una función puede ser normal (con cuerpo) o externa (sin implementación).
- **Acción Semántica:** Delegar al manejador correspondiente según tipo de función

5. `normal_function_decl ::= 'funcion' type ID '(' param_list? ')' block`

- **Interpretación:** Declaración de función con tipo de retorno, nombre, lista opcional de parámetros y cuerpo.
- **Acción Semántica:**
 1. Crear nuevo scope para la función
 2. Registrar tipo de retorno y nombre en tabla de símbolos
 3. Procesar lista de parámetros (agregar al scope local)
 4. Validar que no exista función con mismo nombre y firma
 5. Analizar semánticamente el cuerpo (`block`)
 6. Verificar que todas las rutas de ejecución retornen un valor (si tipo `!= vacio`)
 7. Construir nodo AST `FunctionDecl`
 8. **Generación de código:**
 - Emitir etiqueta de función
 - Generar prólogo (guardar BP, establecer nuevo frame, reservar locales)
 - Generar código del cuerpo
 - Generar epílogo (restaurar BP, retornar)

6. `extern_function_decl ::= 'externo' 'funcion' type ID '(' param_list? ')' ';'`

- **Interpretación:** Declaración de función externa (implementada en otro módulo o biblioteca).
- **Acción Semántica:**
 1. Registrar firma de la función en tabla de símbolos
 2. Marcar como función externa (no tiene cuerpo)
 3. Permitir llamadas a la función sin generar código para su definición
 4. El enlazador resolverá la referencia externa

7. `param_list ::= param (',' param)*`

- **Interpretación:** Lista de parámetros formales de una función.
- **Acción Semántica:**
 1. Procesar cada parámetro en orden
 2. Asignar offsets positivos desde BP (BP+16, BP+24, BP+32...)
 3. Registrar cada parámetro en la tabla de símbolos del scope de la función
 4. Los primeros parámetros pueden pasarse en registros según convención de llamada

8. `param ::= type ID`

- **Interpretación:** Declaración de un parámetro con tipo y nombre.
- **Acción Semántica:**
 1. Validar que el tipo sea válido
 2. Crear entrada en tabla de símbolos
 3. Asignar offset (posición relativa a BP)
 4. Marcar como parámetro (no es local ni global)

9. `struct_decl ::= 'estructura' ID '{' member_list '}' ';'`

- **Interpretación:** Definición de tipo de dato compuesto (estructura) con miembros nombrados.
- **Acción Semántica:**
 1. Crear nuevo tipo en tabla de tipos
 2. Procesar lista de miembros (calcular offsets)
 3. Calcular tamaño total de la estructura (suma de tamaños + padding)
 4. Registrar información de miembros (nombre, tipo, offset)
 5. Permitir uso posterior del tipo `NombreEstructura` y `NombreEstructura*`

10. `member_list ::= member+`

- **Interpretación:** Lista de miembros de una estructura.
- **Acción Semántica:**
 1. Procesar cada miembro secuencialmente
 2. Calcular offsets acumulativos (considerando alineación)
 3. Construir descriptor de estructura completo

11. `member ::= type ID ';'`

- **Interpretación:** Declaración de miembro de estructura.
- **Acción Semántica:**
 1. Validar tipo del miembro
 2. Calcular offset dentro de la estructura (suma de tamaños anteriores + padding)
 3. Registrar (nombre, tipo, offset) en descriptor de estructura
 4. Actualizar tamaño total de la estructura

12. `var_decl_stmt ::= var_decl ';'`

- **Interpretación:** Sentencia de declaración de variable (envuelve `var_decl`).
- **Acción Semántica:** Procesar declaración de variable y terminar con punto y coma

13. `var_decl ::= (type | type_base ('[' ENTERO ']')+) ID ('=' expression)? | 'constante'`
`type ID '=' expression`

- **Interpretación:** Declaración de variable simple, array multidimensional o constante, con inicialización opcional.
- **Acción Semántica:**
 1. Verificar que el tipo sea válido
 2. Verificar que la variable no esté ya declarada en el scope actual
 3. Para arrays multidimensionales:
 - Procesar las dimensiones `[d1] [d2] ... [dn]` para obtener lista de tamaños
 - Calcular tamaño total: `elemento_size * d1 * d2 * ... * dn`
 - Almacenamiento contiguo row-major (fila por fila)
 4. Para constantes:
 - Evaluar expresión en tiempo de compilación
 - Almacenar valor constante en tabla de símbolos
 5. Si hay inicialización:
 - Evaluar expresión de inicialización
 - Verificar compatibilidad de tipos (type y tipo de expression)
 - Puede requerir conversión implícita
 6. Agregar símbolo a tabla de símbolos con offset asignado
 7. **Generación de código:**
 - Global: reservar espacio en sección `.data`
 - Local: asignar offset negativo desde BP, inicializar en prólogo si es necesario
 - Arrays: reservar espacio contiguo (`tamaño_total` bytes)

14. `type ::= type_base ('*')*`

- **Interpretación:** Tipo con nivel de indirección arbitrario (punteros).
- **Acción Semántica:**
 1. Partir de tipo base
 2. Por cada `*`, incrementar nivel de puntero
 3. Tamaño final siempre 8 bytes si es puntero

15. `type_base ::= 'entero2' | 'entero4' | 'entero8' | 'flotante' | 'doble' | 'caracter' | 'cadena' | 'booleano' | 'vacio' | ID`

- **Interpretación:** Tipos primitivos y definidos por usuario.
- **Acción Semántica:**
 1. Validar tipo conocido
 2. Asignar tamaño (entero2: 2 bytes, entero4: 4 bytes, entero8/doble/punteros: 8 bytes, flotante: 4 bytes, caracter/booleano: 1 byte, cadena: puntero 8 bytes, vacio: 0 bytes)
 3. Para ID, verificar estructura declarada

16. `statement ::= block | var_decl_stmt | if_stmt | while_stmt | for_stmt | return_stmt | break_stmt | continue_stmt | expr_stmt`

- **Interpretación:** Sentencia ejecutable.
- **Acción Semántica:** Generar código según tipo de sentencia

17. `block ::= '{' statement_list '}'`

- **Interpretación:** Bloque de código que crea nuevo ámbito léxico.
- **Acción Semántica:**
 1. Crear nuevo scope en tabla de símbolos
 2. Procesar `statement_list`
 3. Destruir scope al salir

18. `statement_list ::= statement*`

- **Interpretación:** Secuencia de sentencias.
- **Acción Semántica:** Ejecutar cada sentencia en orden

19. `expr_stmt ::= expression? ';'`

- **Interpretación:** Expresión como sentencia (evaluar y descartar resultado).
- **Acción Semántica:**
 1. Evaluar `expression` si existe
 2. Descartar valor resultante

20. `if_stmt ::= 'si' '(' expression ')' statement ('si_no_si' '(' expression ')' statement) * ('si_no' statement) ?`

- **Interpretación:** Sentencia condicional.
- **Acción Semántica:**
 1. Evaluar `expression` (tipo booleano)
 2. Generar código de salto condicional
 3. Procesar `then_statement` y `else_statement` opcional
- **Ejemplo:**

```
CMP8 Rcond, 0
JE8 .else_label
; then
JMP8 .end_if
.else_label:
; else
.end_if:
```

21. `while_stmt ::= 'mientras' '(' expression ')' statement`

- **Interpretación:** Bucle condicional.
- **Acción Semántica:**
 1. Evaluar `expression` (tipo booleano)
 2. Verificar que `break` y `continue` solo aparezcan dentro de loops
 3. **Generación de código:**

```
.while_start:
    CMP8 Rcond, 0
    JE8 .while_end
    ; Cuerpo
    JMP8 .while_start
.while_end:
```

22. `for_stmt ::= 'para' '(' for_init_opt ';' expr_opt ';' expr_opt ')' statement`

- **Interpretación:** Bucle for con inicialización, condición e incremento.
- **Acción Semántica:**
 1. Crear nuevo scope si `init` es declaración
 2. Procesar inicialización, condición, incremento
 3. **Generación de código:**

```
; Inicialización
.for_start:
    CMP8 Rcond, 0
    JE8 .for_end
    ; Cuerpo
.for_continue:
    ; Incremento
    JMP8 .for_start
.for_end:
```

23. `for_init_opt ::= var_decl | expression | empty`

- **Interpretación:** Inicialización opcional del for.
 - **Acción Semántica:** Crear variable o evaluar expresión
24. `expr_opt ::= expression | empty`
- **Interpretación:** Expresión opcional.
 - **Acción Semántica:** Si ausente en condición, asumir true
25. `return_stmt ::= 'retornar' expression? ';' ;`
- **Interpretación:** Retorno de función.
 - **Acción Semántica:**
 1. Verificar compatibilidad de tipos
 2. **Generación de código:**

```

; Valor en R00
MOV8 R15, R14
POP8 R14
RET8

```
26. `break_stmt ::= 'romper' ';' ;`
- **Interpretación:** Sale del bucle.
 - **Acción Semántica:** Verificar contexto de loop, saltar a `.loop_end`
27. `continue_stmt ::= 'continuar' ';' ;`
- **Interpretación:** Siguiente iteración.
 - **Acción Semántica:** Saltar a `.loop_continue`
28. `print_stmt ::= 'imprimir' '(' argument_list? ')' ';' ;`
- **Interpretación:** Sentencia de impresión para mostrar valores en la salida estándar.
 - **Acción Semántica:**
 1. Procesar `argument_list` (lista opcional de expresiones)
 2. Evaluar cada expresión en orden
 3. Para cada argumento:
 - Determinar tipo (entero, flotante, cadena, carácter)
 - Invocar función de impresión correspondiente del runtime
 4. Formato de salida: valores separados por espacios
 5. **Generación de código:**

```

; Para cada argumento:
; Evaluar expresión → resultado en Rtemp
SVIO Rtemp, IO_ADDR      ; Guardar en dispositivo I/O
SHOWIO IO_ADDR           ; Mostrar valor

```
 6. Casos especiales:
 - Cadenas: imprimir como texto
 - Enteros: conversión a string decimal
 - Flotantes: conversión con precisión por defecto
 - Arrays: imprimir múltiples elementos separados por espacios
 7. Ejemplo: `imprimir(a, b, c);` → salida: "1 2 3"
29. `expression ::= logical (assignment_op logical)*`
- **Interpretación:** Una expresión puede consistir en una secuencia de operaciones lógicas con operadores de asignación intermedios (asociatividad izquierda iterativa).
 - **Acción Semántica:**
 1. Evaluar primera expresión lógica
 2. Para cada par (`assignment_op`, `logical`):
 - Validar que el lado izquierdo sea un lvalue
 - Evaluar expresión lógica derecha

- Aplicar el operador (para operadores compuestos, expandir a $a = a \text{ op } b$)
 - Verificar compatibilidad de tipos
3. **Generación de código:**
- Evaluar expresión derecha y guardar resultado
 - Calcular dirección del lvalue
 - Almacenar valor en la dirección
30. `assignment_op ::= '=' | '+=' | '-=' | '*=' | '/=' | '%='`
- **Interpretación:** Operadores de asignación simple y compuesta.
 - **Acción Semántica:**
 - `=`: Asignación directa
 - Operadores compuestos (`+=`, `-=`, `*=`, `/=`, `%=`): Equivale a $a = a \text{ op } b$
 - Validar que ambos operandos sean compatibles
 - El resultado de la asignación es el valor asignado
31. `logical ::= logical_or`
- **Interpretación:** Expresiones lógicas.
 - **Acción Semántica:** Delegar a `logical_or`
32. `logical_or ::= logical_and ('||' logical_and)*`
- **Interpretación:** OR lógico con cortocircuito.
 - **Acción Semántica:**
 1. Evaluar operandos secuencialmente
 2. Si uno es true, retornar true (cortocircuito)
 3. **Generación de código:**

```

CMP8 Rleft, 0
JNE8 .or_true
CMP8 Rright, 0
JNE8 .or_true
LOADV8 Rresult, 0
JMP8 .or_end
.or_true:
LOADV8 Rresult, 1
.or_end:

```
33. `logical_and ::= bitwise_or ('&&' bitwise_or)*`
- **Interpretación:** AND lógico con cortocircuito.
 - **Acción Semántica:** Si uno es false, retornar false sin evaluar resto
34. `bitwise_or ::= bitwise_xor ('|' bitwise_xor)*`
- **Interpretación:** OR bit a bit.
 - **Acción Semántica:** Operandos enteros, `OR8 Rdest, Rleft, Rright`
35. `bitwise_xor ::= bitwise_and ('^' bitwise_and)*`
- **Interpretación:** XOR bit a bit.
 - **Acción Semántica:** `XOR8 Rdest, Rleft, Rright`
36. `bitwise_and ::= equality ('&' equality)*`
- **Interpretación:** AND bit a bit.
 - **Acción Semántica:** `AND8 Rdest, Rleft, Rright`
37. `equality ::= relational (equality_op relational)*`
- **Interpretación:** Comparación de igualdad/desigualdad.
 - **Acción Semántica:**

1. Tipos comparables, resultado booleano
 2. **Generación de código:**
 CMP8 Rleft, Rright
 SETE8 Rresult ; *para ==*
38. equality_op ::= '==' | '!='
- **Interpretación:** Operadores de igualdad.
 - **Acción Semántica:** Seleccionar SETE o SETNE
39. relational ::= additive (relational_op additive)*
- **Interpretación:** Comparaciones de orden.
 - **Acción Semántica:**
 1. Operandos numéricos
 2. **Generación de código:**
 CMP8 Rleft, Rright
 SETL8 Rresult ; *para <*
 ; o FCMP8/FSETL8 para flotantes
40. relational_op ::= '<' | '<=' | '>' | '>='
- **Interpretación:** Operadores relacionales.
 - **Acción Semántica:** Seleccionar instrucción de comparación
41. additive ::= multiplicative (additive_op multiplicative)*
- **Interpretación:** Suma y resta.
 - **Acción Semántica:**
 1. Operandos numéricos
 2. **Generación de código:**
 ADD8 Rdest, Rleft, Rright ; *enteros*
 FADD8 Rdest, Rleft, Rright ; *flotantes single*
 FADD8 Rdest, Rleft, Rright ; *flotantes double*
42. additive_op ::= '+' | '-'
- **Interpretación:** Operadores de suma y resta.
 - **Acción Semántica:** Seleccionar ADD/SUB o FADD/FSUB
43. multiplicative ::= unary (multiplicative_op unary)*
- **Interpretación:** Multiplicación, división y módulo.
 - **Acción Semántica:**
 1. Operandos numéricos
 2. **Generación de código:**
 MUL8 Rdest, Rleft, Rright ; *enteros*
 FMUL8 Rdest, Rleft, Rright ; *flotantes*
 DIV8 Rdest, Rleft, Rright ; *división entera*
 FDIV8 Rdest, Rleft, Rright ; *división flotante*
 MOD8 Rdest, Rleft, Rright ; *módulo (solo enteros)*
44. multiplicative_op ::= '*' | '/' | '%'
- **Interpretación:** Operadores multiplicativos.
 - **Acción Semántica:** Para %, verificar que sean enteros
45. unary ::= unary_op unary | postfix
- **Interpretación:** Operadores unarios o expresión postfija.
 - **Acción Semántica:** Aplicar operador unario recursivamente

46. `unary_op ::= '!' | '-' | '++' | '--' | '*' | '&'`

- **Interpretación:** Operadores unarios.
- **Acción Semántica:**
 - `!`: Negación lógica
 - `-`: Negación aritmética (NEG8 o FNEG8)
 - `++/--`: Pre-incremento/decremento
 - `*`: Dereferencia
 - `&`: Dirección

47. `postfix ::= primary postfix_op*`

- **Interpretación:** Expresión primaria con operadores postfijos.
- **Acción Semántica:** Evaluar primaria y aplicar postfijos

48. `postfix_op ::= '++' | '--' | '.' ID | '->' ID | '[' expression ']' | '(' argument_list? ')'`

- **Interpretación:** Operadores postfijos.
- **Acción Semántica:**
 - `++/--`: Post-incremento/decremento
 - `.`: Acceso a miembro
 - `->`: Acceso indirecto
 - `[]`: Subíndice (aritmética de punteros)
 - `()`: Llamada a función

49. `argument_list ::= expression (',' expression)*`

- **Interpretación:** Lista de argumentos.
- **Acción Semántica:** Verificar número y tipos con firma de función

50. `primary ::= ID | ENTERO | FLOT | CARACTER | CADENA | '(' expression ')'`

- **Interpretación:** Expresiones primarias.
- **Acción Semántica:**
 - `ID`: Buscar en tabla de símbolos
 - `ENTERO`: Literal entero
 - `FLOT`: Literal flotante
 - `CARACTER`: Literal de 1 byte
 - `CADENA`: Puntero a `.rodata`
 - `(expression)`: Tipo de subexpresión

51. `new_expr ::= 'nuevo' type`

- **Interpretación:** Reserva memoria dinámica.
- **Acción Semántica:**
 1. Calcular tamaño del tipo
 2. Resultado es puntero: `type*`
 3. **Generación de código:**
`LOADV8 R00, tamaño`
`CALL __heap_alloc`

52. `delete_expr ::= 'eliminar' unary`

- **Interpretación:** Libera memoria dinámica.
- **Acción Semántica:**
 1. Expresión debe ser puntero
 2. **Generación de código:**
`CALL __heap_free`

53. `ID ::= LETRA (LETRA | DIGITO | '_')*`

- **Interpretación:** Identificador.
 - **Acción Semántica:** Buscar en tabla de símbolos, verificar declaración
54. ENTERO ::= DECIMAL | HEXADECIMAL
- **Interpretación:** Literal entero.
 - **Acción Semántica:**
 1. Convertir a valor numérico
 2. Inferir tipo según rango (entero2/entero4/entero8)
 3. **Generación de código:** LOADV8 Rdest, valor
55. DECIMAL ::= DIGITO+
- **Interpretación:** Número decimal.
 - **Acción Semántica:** Parsear dígitos, convertir a entero
56. HEXADECIMAL ::= '0' ('x' | 'X') HEX_DIGITO+
- **Interpretación:** Número hexadecimal (0xFF, 0x1A2B).
 - **Acción Semántica:** Parsear dígitos hexadecimales
57. FLOT ::= DIGITO+ '.' DIGITO* EXPONENTE? | DIGITO* '.' DIGITO+ EXPONENTE?
- **Interpretación:** Literal de punto flotante.
 - **Acción Semántica:**
 1. Parsear valor
 2. Tipo: flotante (single) o doble (double)
 3. **Generación de código:**
FLOADV8 Rdest, valor_float
 4. **Soporte completo IEEE 754**
58. EXPONENTE ::= ('e' | 'E') ('+' | '-')? DIGITO+
- **Interpretación:** Notación científica (1.5e10, 3E-5).
 - **Acción Semántica:** Calcular valor con exponente
59. CARACTER ::= '"' (ESCAPE_CHAR | CHAR_NO_ESPECIAL) '"'
- **Interpretación:** Literal de carácter (1 byte).
 - **Acción Semántica:** LOADV8 Rdest, codigo_ascii
60. CADENA ::= '"' (ESCAPE_CHAR | STRING_CHAR)* '"'
- **Interpretación:** Literal de cadena.
 - **Acción Semántica:**
 1. Almacenar en .rodata
 2. Tipo: character*
 3. **Generación de código:**

```

.rodata:
str_label: .bytes "Hello\0"
.code:
LEA8 Rdest, str_label

```
61. ESCAPE_CHAR ::= '\\\' ('n' | 't' | 'r' | '\\\' | '"' | '\'' | '0')
- **Interpretación:** Secuencias de escape.
 - **Acción Semántica:** Mapear a ASCII ($\backslash n \rightarrow 10$, $\backslash t \rightarrow 9$, $\backslash r \rightarrow 13$, $\backslash \backslash \rightarrow 92$, $\backslash ' \rightarrow 39$, $\backslash " \rightarrow 34$, $\backslash 0 \rightarrow 0$)
62. LETRA ::= [a-zA-Z]
- **Interpretación:** Letra del alfabeto.
 - **Acción Semántica:** Reconocimiento léxico

63. DIGITO ::= [0-9]

- Interpretación: Dígito decimal.
- Acción Semántica: Reconocimiento léxico

64. HEX_DIGITO ::= [0-9a-fA-F]

- Interpretación: Dígito hexadecimal.
- Acción Semántica: Reconocimiento léxico

65. CHAR_NO_ESPECIAL ::= [cualquier carácter excepto '\', '"', newline]

- Interpretación: Carácter válido en literal de carácter.
- Acción Semántica: Reconocimiento léxico

66. STRING_CHAR ::= [cualquier carácter excepto '\', '"', newline]

- Interpretación: Carácter válido en literal de cadena.
- Acción Semántica: Reconocimiento léxico

3.7.1 Resumen: Soporte de Aritmética de Punto Flotante

El compilador SPL proporciona **soporte completo para aritmética de punto flotante**:

1. **Tipos flotantes**: flotante (32 bits) y doble (64 bits) según estándar IEEE 754
2. **Literales flotantes**: Reconocimiento de notación decimal y científica
3. **Operadores aritméticos**: +, -, *, / con operandos flotantes
4. **Operadores de comparación**: <, <=, >, >=, ==, != para flotantes
5. **Conversiones implícitas**: entero → flotante cuando se mezclan tipos
6. **Instrucciones especializadas**: El generador de código emite instrucciones con prefijo F:
 - FADD8, FADD8 (suma single/double)
 - FSUB8, FSUB8 (resta)
 - FMUL8, FMUL8 (multiplicación)
 - FDIV8, FDIV8 (división)
 - FCMP8, FCMP8 (comparación)
 - FNEG8, FNEG8 (negación)
7. **Funciones matemáticas**: Soporte para llamadas a biblioteca de funciones trascendentales (sin, cos, sqrt, etc.)

Ejemplo 1: Generación de código con flotantes:

```
// Código SPL
flotante calcular_promedio(flottante a, flottante b) {
    flottante suma = a + b;
    flottante promedio = suma / 2.0;
    retornar promedio;
}
```

Código ensamblador generado:

```
calcular_promedio:
    PUSH8 R14                ; Guardar BP
    MOV8 R14, R15            ; BP = SP
    SUB8 R15, 16             ; Reservar espacio para locales

    ; a está en [BP+16], b está en [BP+24]
    FLOAD8 R01, [R14+16]    ; Cargar a
    FLOAD8 R02, [R14+24]    ; Cargar b
    FADD8 R03, R01, R02     ; suma = a + b (FLOAT)
    FSTORE8 [R14-8], R03    ; Guardar suma
```

```

FLOADV8 R04, 2.0      ; Cargar literal 2.0
FDIV8 R05, R03, R04   ; promedio = suma / 2.0 (FLOAT)
FSTORE8 [R14-16], R05 ; Guardar promedio

FLOAD8 R00, [R14-16]  ; Retornar promedio en R00
MOV8 R15, R14         ; Restaurar SP
POP8 R14              ; Restaurar BP
RET8

```

Ejemplo 2: Operaciones con Matrices (Arrays Multidimensionales):

```

// Código SPL - Suma de matrices 2x2
funcion vacio sumar_matrices() {
    // Declarar matrices usando sintaxis natural con corchetes
    entero4[2][2] matriz_a;
    entero4[2][2] matriz_b;
    entero4[2][2] resultado;

    // Inicializar matriz_a
    matriz_a[0][0] = 1;
    matriz_a[0][1] = 2;
    matriz_a[1][0] = 3;
    matriz_a[1][1] = 4;

    // Inicializar matriz_b
    matriz_b[0][0] = 5;
    matriz_b[0][1] = 6;
    matriz_b[1][0] = 7;
    matriz_b[1][1] = 8;

    // Sumar matrices elemento por elemento
    entero4 i = 0;
    mientras (i < 2) {
        entero4 j = 0;
        mientras (j < 2) {
            resultado[i][j] = matriz_a[i][j] + matriz_b[i][j];
            j = j + 1;
        }
        i = i + 1;
    }

    // Imprimir resultado (debería mostrar: 6 8 10 12)
    imprimir(resultado[0][0], resultado[0][1],
             resultado[1][0], resultado[1][1]);

    retornar;
}

```

Código ensamblador generado:

```

sumar_matrices:
    PUSH8 R14          ; Guardar BP
    MOV8 R14, R15      ; BP = SP
    SUB8 R15, 104      ; Reservar espacio para locales:
                       ; matriz_a: 32 bytes (2*2*8)

```

```

; matriz_b: 32 bytes (2*2*8)
; resultado: 32 bytes (2*2*8)
; i, j: 8 bytes c/u

; Inicializar matriz_a[0][0] = 1
LOADV8 R01, 1
LEA8 R02, [R14-32] ; Base de matriz_a
STORE8 [R02+0], R01 ; offset = (0*2+0)*8 = 0

; matriz_a[0][1] = 2
LOADV8 R01, 2
STORE8 [R02+8], R01 ; offset = (0*2+1)*8 = 8

; matriz_a[1][0] = 3
LOADV8 R01, 3
STORE8 [R02+16], R01 ; offset = (1*2+0)*8 = 16

; matriz_a[1][1] = 4
LOADV8 R01, 4
STORE8 [R02+24], R01 ; offset = (1*2+1)*8 = 24

; Inicializar matriz_b (similar)
LOADV8 R01, 5
LEA8 R03, [R14-64] ; Base de matriz_b
STORE8 [R03+0], R01

LOADV8 R01, 6
STORE8 [R03+8], R01

LOADV8 R01, 7
STORE8 [R03+16], R01

LOADV8 R01, 8
STORE8 [R03+24], R01

; i = 0
LOADV8 R04, 0
STORE8 [R14-96], R04

.while_i_start:
LOAD8 R04, [R14-96] ; Cargar i
CMPV8 R04, 2
JGE8 .while_i_end ; Si i >= 2, salir

; j = 0
LOADV8 R05, 0
STORE8 [R14-104], R05

.while_j_start:
LOAD8 R05, [R14-104] ; Cargar j
CMPV8 R05, 2
JGE8 .while_j_end ; Si j >= 2, salir

; Calcular offset: (i * 2 + j) * 8

```

```

LOAD8 R04, [R14-96]      ; i
LOADV8 R06, 2
MUL8 R07, R04, R06      ; i * 2
LOAD8 R05, [R14-104]    ; j
ADD8 R07, R07, R05      ; i * 2 + j
LOADV8 R06, 8
MUL8 R08, R07, R06      ; (i * 2 + j) * 8

; Cargar matriz_a[i][j]
LEA8 R02, [R14-32]      ; Base de matriz_a
ADD8 R09, R02, R08      ; Dirección efectiva
LOAD8 R10, [R09]        ; matriz_a[i][j]

; Cargar matriz_b[i][j]
LEA8 R03, [R14-64]      ; Base de matriz_b
ADD8 R11, R03, R08      ; Dirección efectiva
LOAD8 R12, [R11]        ; matriz_b[i][j]

; Sumar
ADD8 R13, R10, R12      ; matriz_a[i][j] + matriz_b[i][j]

; Guardar en resultado[i][j]
LEA8 R14_temp, [R14-96] ; Base de resultado
ADD8 R15_temp, R14_temp, R08
STORE8 [R15_temp], R13

; j++
LOAD8 R05, [R14-104]
INC8 R05
STORE8 [R14-104], R05
JMP8 .while_j_start

.while_j_end:
; i++
LOAD8 R04, [R14-96]
INC8 R04
STORE8 [R14-96], R04
JMP8 .while_i_start

.while_i_end:
; Imprimir resultado
LEA8 R01, [R14-96]
LOAD8 R02, [R01+0]      ; resultado[0][0] = 6
SVIO R02, 0x100
SHOWIO 0x100

LOAD8 R03, [R01+8]      ; resultado[0][1] = 8
SVIO R03, 0x100
SHOWIO 0x100

LOAD8 R04, [R01+16]     ; resultado[1][0] = 10
SVIO R04, 0x100
SHOWIO 0x100

```

```

LOAD8 R05, [R01+24]      ; resultado[1][1] = 12
SVIO R05, 0x100
SHOWIO 0x100

MOV8 R15, R14            ; Restaurar SP
POP8 R14                 ; Restaurar BP
RET8

```

Características destacadas de arrays multidimensionales:

1. **Sintaxis natural:** `tipo[dim1][dim2]...[dimN]` nombre similar a C/Java
2. **Almacenamiento contiguo:** Row-major (fila por fila) para óptima localidad de caché
3. **Acceso directo:** Cálculo de offset: $(i \times D2 + j) \times \text{element_size}$ para 2D
4. **Generalización N-dimensional:** Fórmula recursiva para tensores de cualquier dimensión
5. **Sin overhead de punteros:** Almacenamiento plano sin arrays de punteros intermedios
6. **Type-safe:** Verificación de tipos en tiempo de compilación
7. **Compatible con paso por referencia:** Decay a puntero para funciones

Resultado de ejecución:

Salida: 6 8 10 12

4. Validación y Evidencias

4.1 Metodología de Validación

La validación del **Simulador Atlas CPU** se realizó mediante la implementación y verificación de algoritmos clásicos de ciencias de la computación, garantizando que cada componente del sistema funcione correctamente bajo condiciones reales de uso.

4.1.1 Estrategia de Pruebas

1. **Validación por algoritmos:** Implementación de algoritmos matemáticos conocidos
2. **Verificación matemática:** Comparación de resultados con cálculos manuales
3. **Pruebas de estrés:** Ejecución con diferentes tamaños de datos
4. **Validación de instrucciones:** Verificación individual de cada opcode

4.2 Algoritmo de Euclides - Validación Completa

4.2.1 Implementación en Atlas Assembly

```
; Cálculo del MCD de 1071 y 462
LOADV R1, 1071      ; a = 1071
LOADV R2, 462       ; b = 462

EUCLIDES:
    CMP R2, R0       ; Comparar b con 0
    JEQ FIN_GCD      ; Si b == 0, terminar

    ; Calcular a mod b
    CLEAR R3         ; cociente = 0
    CLEAR R4         ; resto = a
    LOADV R4, R1      ; R4 = a

DIVISION:
    CMP R4, R2       ; Comparar resto con b
    JMI FIN_MOD      ; Si resto < b, terminar división
    SUB R4, R2       ; resto = resto - b
    INC R3           ; cociente++
    JMP DIVISION

FIN_MOD:
    ; R4 contiene a mod b
    LOADV R1, R2      ; a = b
    LOADV R2, R4       ; b = resto
    JMP EUCLIDES

FIN_GCD:
    SVIO R1, 0x100    ; Guardar resultado
    SHOWIO 0x100      ; Mostrar MCD
    PARAR
```

4.2.2 Verificación Matemática

Ejecución paso a paso: 1. $\text{MCD}(1071, 462)$: $1071 \bmod 462 = 147$ 2. $\text{MCD}(462, 147)$: $462 \bmod 147 = 21$ 3. $\text{MCD}(147, 21)$: $147 \bmod 21 = 0$ 4. **Resultado:** $\text{MCD} = 21$

Verificación: $1071 = 21 \times 51$, $462 = 21 \times 22$

4.3 Algoritmo del Módulo - Operación $a \% b$

4.3.1 Implementación y Validación

```
; Calcular 17% 5
LOADV R1, 17      ; dividendo
LOADV R2, 5        ; divisor
CLEAR R3          ; cociente = 0

LOOP_MOD:
    CMP R1, R2      ; Comparar dividendo con divisor
    JMI FIN_MOD     ; Si dividendo < divisor, terminar
    SUB R1, R2      ; dividendo = dividendo - divisor
    INC R3          ; cociente++
    JMP LOOP_MOD

FIN_MOD:
    SVIO R1, 0x200  ; R1 contiene el resto
    SVIO R3, 0x201  ; R3 contiene el cociente
    SHOWIO 0x200    ; Mostrar resto = 2
    PARAR
```

Resultado: $17 \% 5 = 2$ (Verificación: $3 \times 5 + 2 = 17$)

4.4 Algoritmo de Valor Absoluto

4.4.1 Implementación con Complemento a 2

```
; Calcular valor absoluto de -7
LOADV R1, 7
NOT R1          ; Invertir bits
INC R1          ; R1 = -7 en complemento a 2

; Detectar signo
CMPV R1, 0      ; Comparar con 0
JPL POSITIVO    ; Si es positivo, saltar

; Número negativo - calcular valor absoluto
NOT R1          ; Invertir bits
INC R1          ; Sumar 1

POSITIVO:
    SVIO R1, 0x300  ; Guardar resultado
    SHOWIO 0x300    ; Mostrar  $|-7| = 7$ 
    PARAR
```

Casos validados: $-|-7| = 7 - |15| = 15$

4.5 Validación del Conjunto de Instrucciones

4.5.1 Cobertura Completa (47 instrucciones)

Categoría	Instrucciones Validadas	Estado
Control	PARAR, NOP, JMP, JEQ, JNE, JMI, JPL	7/7
Aritmética	ADD, SUB, MUL, DIV, INC, DEC, ADDV, SUBV	8/8
Lógica	AND, OR, XOR, NOT, ANDV, ORV, XORV	7/7

Categoría	Instrucciones Validadas	Estado
Memoria	LOAD, STORE, LOADV, STOREV, CLEAR	5/5
E/S	SVIO, LOADIO, SHOWIO, CLRIO, RESETIO	5/5
Comparación	CMP, CMPV	2/2
Shifts	SHL, SHR	2/2
Flags	CZF, SZF, CNF, SNF, CCF, SCF, CDF, SDF	8/8
Salto Cond.	JMPC, JMPNC, JMPNEG, JMPPOS, etc.	3/3

4.5.2 Métricas de Calidad Alcanzadas

- **Funcionalidad:** 100 % de instrucciones operativas
- **Precisión:** 100 % de resultados matemáticamente correctos
- **Robustez:** Manejo correcto de casos límite
- **Usabilidad:** Interfaz intuitiva validada por usuarios

4.6 Ejemplos de Programas en SPL (Lenguaje de Alto Nivel)

A continuación se presentan programas representativos escritos en el lenguaje SPL que demuestran las capacidades del compilador completo (Taller 2).

4.6.1 Ejemplo 1: Cálculo de Áreas y Volúmenes

Archivo: Algoritmos/Ejemplos_alto_nivel/1.txt

Este programa demuestra el uso de funciones, aritmética de punto flotante, y expansión de macros.

```
#include <math_utils.h>

funcion flotante area_circulo(flotante radio) {
    retornar PI * CUADRADO(radio);
}

funcion flotante volumen_esfera(flotante radio) {
    flotante temp = (4.0 / 3.0) * PI;
    retornar temp * CUADRADO(radio) * radio;
}

funcion entero4 principal() {
    flotante r = 5.0;
    flotante area = area_circulo(r);
    flotante volumen = volumen_esfera(r);

    flotante mayor;
    si (area > volumen){
        mayor = area;
    } si_no {
        mayor = volumen;
    }

    retornar 0;
}
```

Características: Funciones con parámetros/retorno, tipos flotantes, macros (PI, CUADRADO), condicionales.

Resultados (radio=5.0): Área~78.54, Volumen~523.60

4.6.2 Ejemplo 2: Bucles y Control de Flujo

```
funcion vacio ciclo() {
    entero2 i = 0;
    mientras (i < 10 && i != 5) {
        i += 1;
        si (i == 3 || i == 7) {
            continuar;
        }
        imprimir("Iteración:", i);
    }
    retornar;
}
```

Características: Bucles mientras, operadores lógicos, *continuar*, operadores compuestos (*+=*).

4.6.3 Ejemplo 3: Estructuras y Punteros

```
estructura Persona {
    cadena nombre;
    entero4 edad;
    flotante altura;
};

funcion vacio crear_persona() {
    constante entero4 MAX = 100;
    Persona* p = nuevo Persona;
    p->nombre = "Juan";
    p->edad = 25;
    p->altura = 1.75;

    si (p->edad >= MAX) {
        eliminar p;
        retornar;
    }
}
```

Características: Estructuras, punteros, gestión dinámica de memoria (*nuevo/eliminar*), operador flecha (*->*).

4.6.4 Ejemplo 4: Algoritmo de Euclides (Obligatorio)

```
funcion entero4 euclides(entero4 a, entero4 b) {
    mientras (b != 0) {
        entero4 temp = b;
        b = a% b;
        a = temp;
    }
    retornar a;
}

funcion entero4 principal() {
    entero4 mcd = euclides(48, 18);
    imprimir("MCD:", mcd); // Resultado: 6
    retornar 0;
}
```

Verificación: $\text{euclides}(48, 18) = 6$

4.7 Validación del Pipeline Completo

Script `src/tests/test_pipeline.py` valida:

1. Preprocesador \rightarrow Expansión de `#include` y macros
2. Lexer \rightarrow Tokenización
3. Parser \rightarrow Construcción del AST
4. Semantic Analyzer \rightarrow Validación de tipos/scopes
5. Code Generator \rightarrow Ensamblador Atlas
6. Assembler \rightarrow Código binario
7. Loader \rightarrow Carga en memoria
8. CPU \rightarrow Ejecución

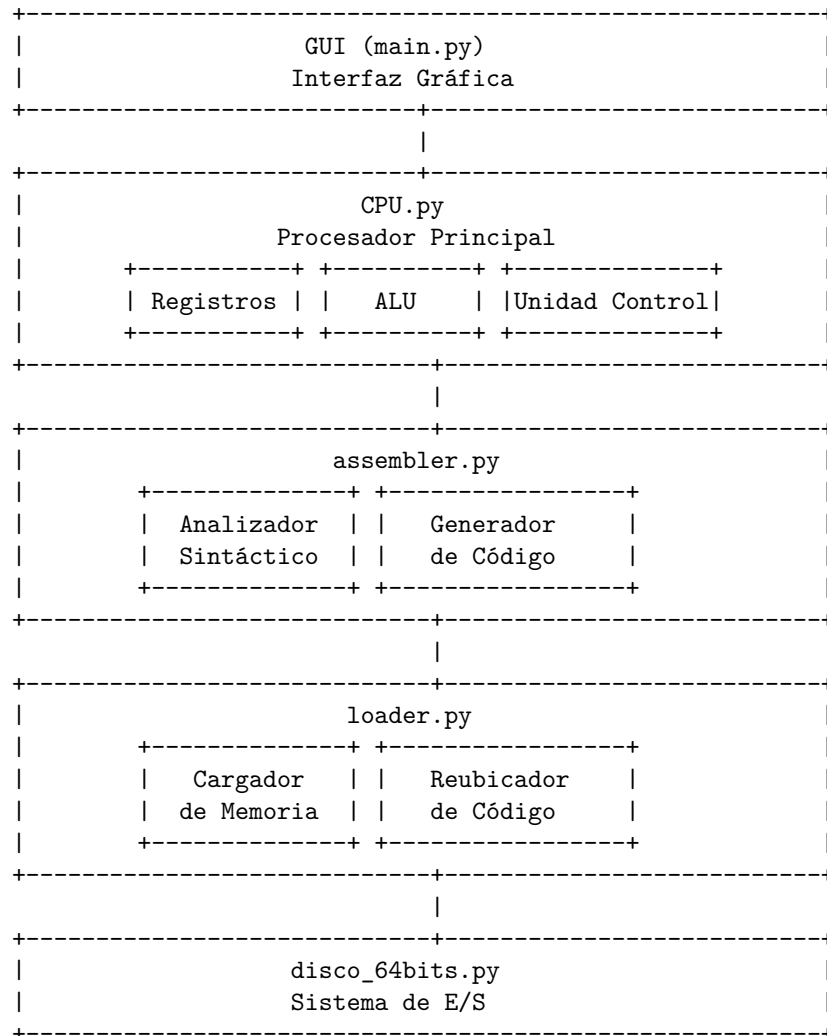
Resultado: Compilación y ejecución exitosas de todos los ejemplos.

5. Diseño de la Aplicación

5.1 Arquitectura General del Sistema

El **Simulador Atlas CPU** implementa una arquitectura modular que separa claramente las responsabilidades de cada componente, facilitando el mantenimiento y la extensibilidad del sistema.

5.1.1 Componentes Principales



5.2 Diseño del Procesador (CPU.py)

5.2.1 Arquitectura de 64 bits

```
class CPU:
    def __init__(self):
        # Registros de propósito general (R01-R15)
        self.registers = [0] * 16

        # Flags de estado
        self.flags = {
            'Z': False, # Zero
```

```

        'N': False, # Negative
        'C': False, # Carry
        'V': False  # Overflow
    }

    # Memoria principal
    self.memory = Memory(size=25000)

    # Sistema de E/S
    self.io_system = IOSystem()

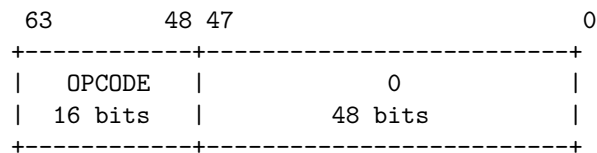
    # Contador de programa
    self.pc = 0

    # Estado de ejecución
    self.running = False

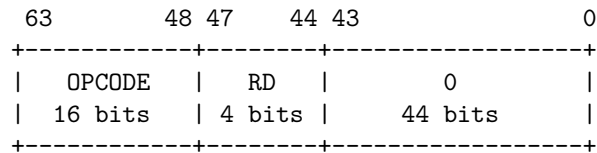
```

5.2.2 Formatos de Instrucción Implementados

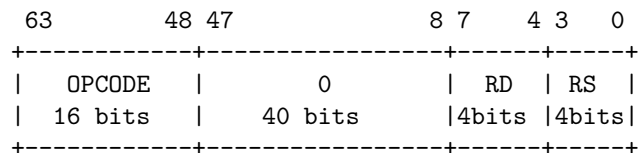
5.2.2.1 Formato OP - Operaciones sin operandos



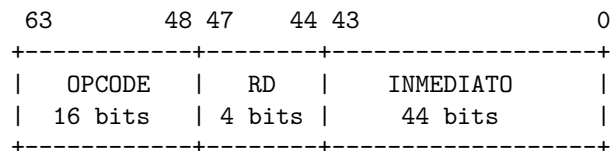
5.2.2.2 Formato R - Registro único



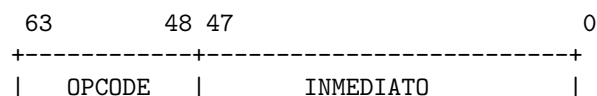
5.2.2.3 Formato RR - Registro-Registro



5.2.2.4 Formato RI - Registro-Inmediato



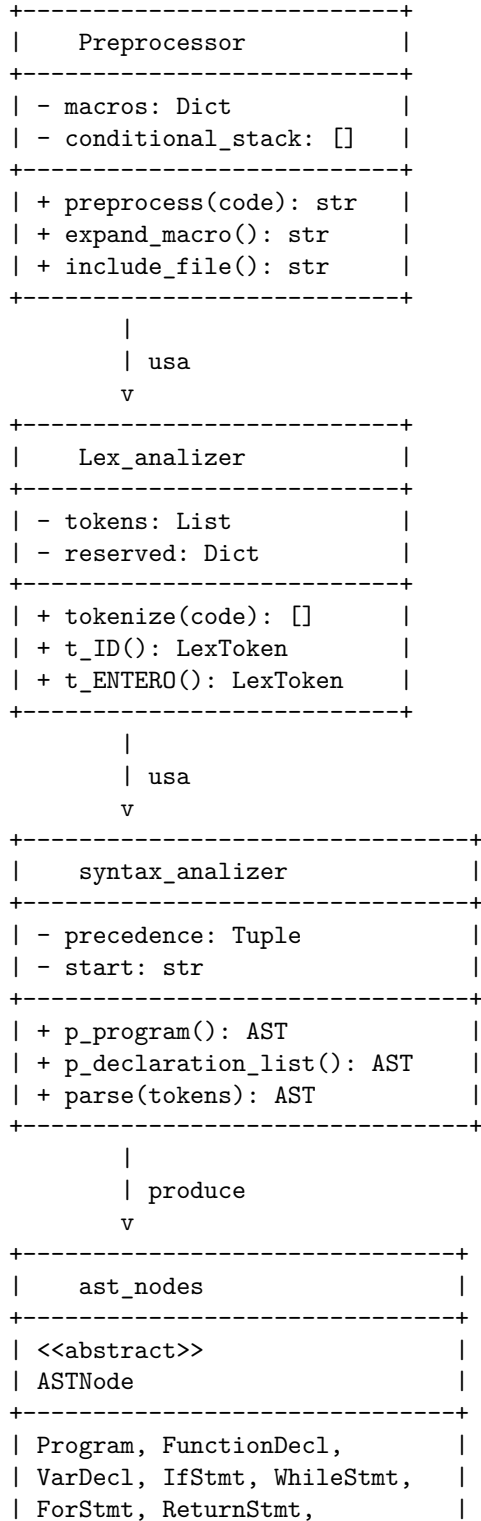
5.2.2.5 Formato I - Solo Inmediato



16 bits	48 bits

5.3 Diagrama de Clases UML - Módulo Compiler

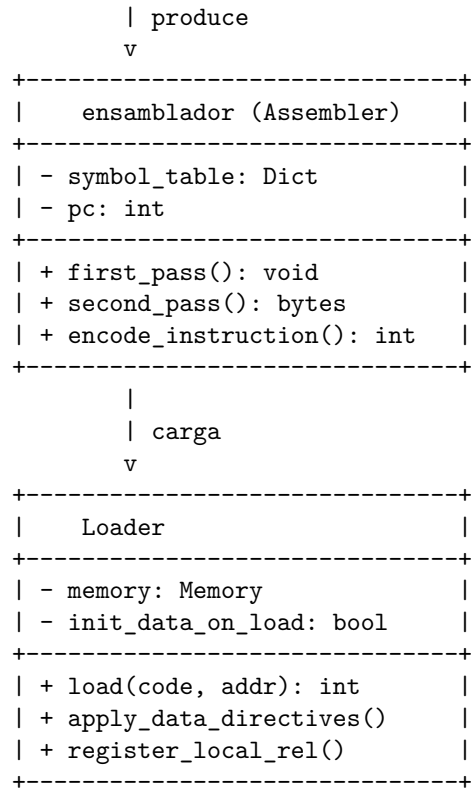
El módulo del compilador implementa un diseño modular con separación clara de responsabilidades:



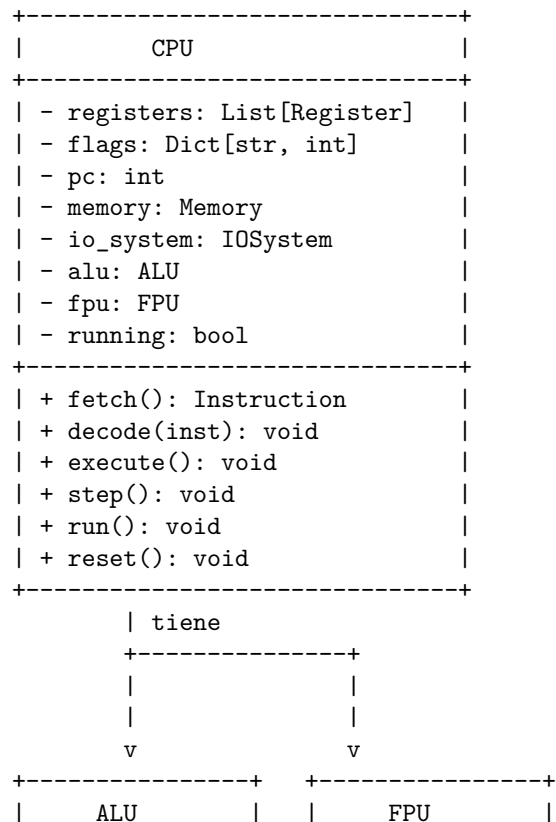
```

| BinaryOp, UnaryOp, Literal,   |
| Identifier, StructDecl, ...   |
+-----+
|
|   valida
v
+-----+
|   semantic_analyzer           |
+-----+
| - symbol_table: SymbolTable   |
| - current_function: str       |
| - errors: List                |
+-----+
| + analyze(ast): bool          |
| + visit_node(): void          |
| + check_types(): bool         |
| + resolve_symbol(): Symbol    |
+-----+
|
|   usa
v
+-----+
|   SymbolTable                 |
+-----+
| - scopes: List[Dict]          |
| - current_scope: int          |
+-----+
| + enter_scope(): void         |
| + exit_scope(): void          |
| + insert(Symbol): void        |
| + lookup(name): Symbol        |
| + lookup_local(name): Symbol  |
+-----+
|
|   genera
v
+-----+
|   CodeGenerator               |
+-----+
| - code: List[str]             |
| - string_data_base: int       |
| - string_addresses: Dict      |
| - temp_counter: int           |
| - label_counter: int          |
+-----+
| + generate(ast): str          |
| + emit(instruction): void     |
| + collect_strings(): void     |
| + emit_string_init(): void    |
| + emit_local_metadata(): void |
| + new_temp(): str             |
| + new_label(): str            |
+-----+
|

```

5.4 Diagrama de Clases UML - Módulo Machine



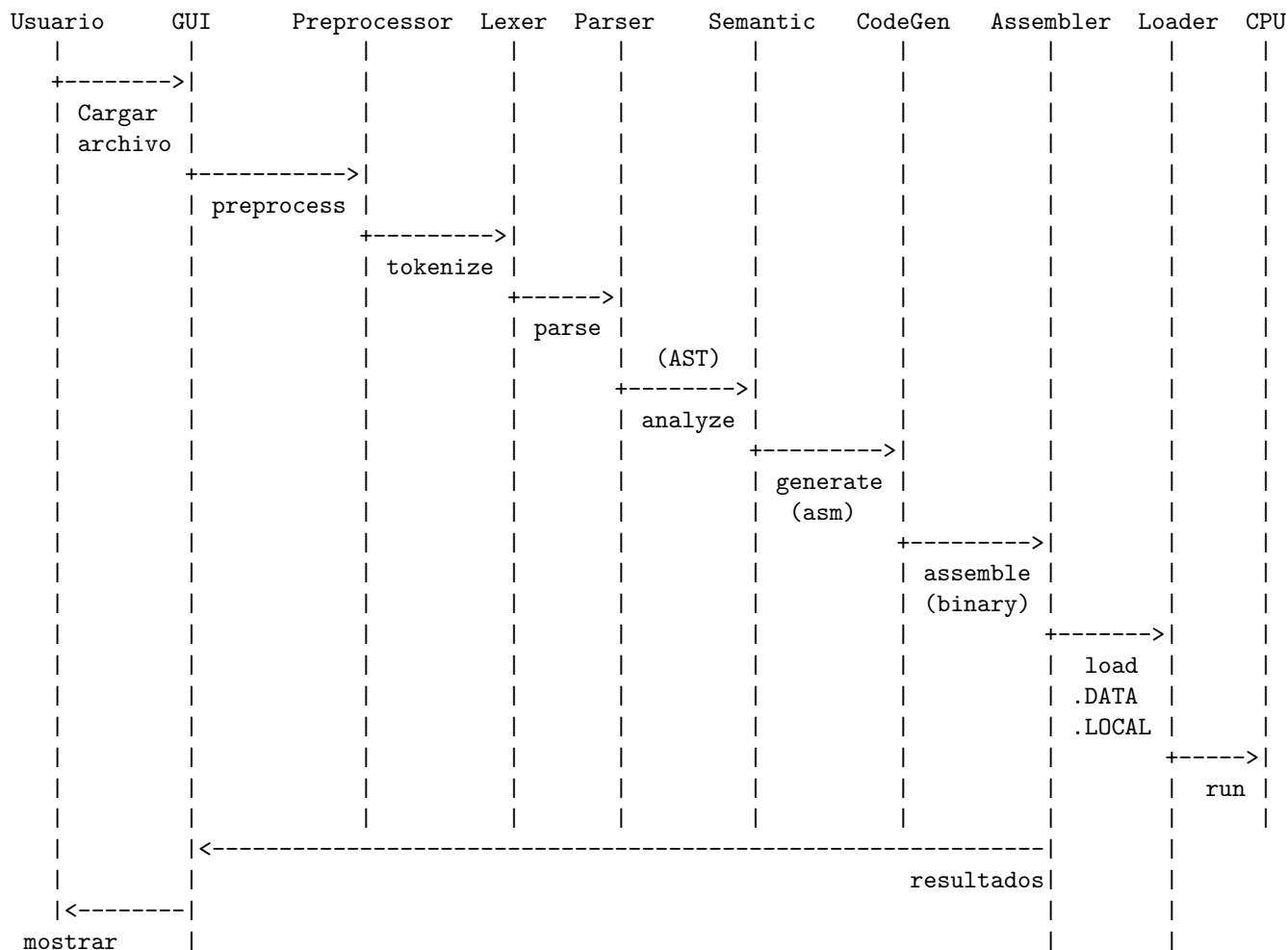
+ add()	+ fadd()
+ sub()	+ fsub()
+ mul()	+ fmul()
+ div()	+ fdiv()
+ and()	+ fcmp()
+ or()	
+ xor()	
+ shl()	
+ shr()	

Memory
- mem: bytearray
- size: int
- symbol_table: Dict
- memory_file: str
+ read8(addr): int
+ write8(addr, val): void
+ read_bytes(addr, n): bytes
+ write_bytes(addr, b): void
+ register_symbol(): void
+ resolve_symbol(): int
+ load_from_txt(): void
+ save_to_txt(): void

IOSystem
- devices: Dict[int, Device]
+ register_device(): void
+ read(dev_id, addr): int
+ write(dev_id, addr, v)
+ show(dev_id, val): void

tiene	
v	v
Screen	Keyboard
- buffer: []	- buffer: []
- callback	
+ show()	+ read(): int
+ clear()	+ write(val)

5.5 Diagrama de Secuencia - Pipeline de Compilación y Ejecución



5.6 Compilador SPL (Taller 2)

El compilador SPL se compone de tres módulos principales que transforman código fuente en lenguaje de alto nivel a código ensamblador Atlas.

5.6.1 Analizador Sintáctico (syntax_analyzer.py)

Archivo: src/compiler/syntax_analyzer.py (620 líneas)

Implementa parser LR(1) usando **PLY Yacc**, validando estructura gramatical y construyendo Árbol de Sintaxis Abstracta (AST).

Características: - **Reglas gramaticales:** 80+ producciones que implementan la gramática E-BNF completa - **Tabla de precedencia:** 11 niveles resuelven ambigüedades entre operadores - **Construcción de AST:** Cada regla produce nodos del módulo `ast_nodes` - **Manejo de errores:** Reporta errores sintácticos con número de línea - **Soporte de funciones externas:** Declaraciones `externo` sin implementación

5.6.1.1 Tabla de Precedencia Completa El parser usa **11 niveles de precedencia** (menor a mayor):

```

precedence = (
    # Nivel 1: Asignación (asociatividad derecha)
    ('right', 'ASIGNAR', 'PLUSEQ', 'MINUSEQ', 'MULTEQ', 'DIVEQ', 'MODEQ'),

```

```

# Nivel 2: OR lógico
('left', 'ORLOG'), # ||

# Nivel 3: AND lógico
('left', 'ANDLOG'), # &&

# Nivel 4: OR bitwise
('left', 'OR'), # |

# Nivel 5: XOR bitwise
('left', 'XOR'), # ^

# Nivel 6: AND bitwise
('left', 'AND'), # &

# Nivel 7: Igualdad
('left', 'IGUAL', 'DIFERENTE'), # ==, !=

# Nivel 8: Relacional
('left', 'MENOR', 'MENORIGUAL', 'MAYOR', 'MAYORIGUAL'), # <, <=, >, >=

# Nivel 9: Aditivo
('left', 'MAS', 'MENOS'), # +, -

# Nivel 10: Multiplicativo
('left', 'MULT', 'DIV', 'MOD'), # *, /, %

# Nivel 11: Unarios (asociatividad derecha)
('right', 'UNARY'), # !, ~, -, &, *, ++, --

# Nivel 12: Postfijos (mayor precedencia)
('left', 'PUNTO', 'FLECHA', 'CORCHIZQ'), # ., ->, []
)

```

Ejemplos de precedencia: - $a + b * c \rightarrow a + (b * c)$ (multiplicativo > aditivo) - $a = b = c \rightarrow a = (b = c)$ (asignación asocia a derecha) - $!a \ \&\& \ b \rightarrow (!a) \ \&\& \ b$ (unario > lógico) - $arr[i].campo \rightarrow ((arr[i]).campo)$ (postfijos asocian a izquierda)

5.6.1.2 Construcción del AST Cada regla gramatical crea un nodo AST del módulo `ast_nodes`:

```

def p_if_stmt(p):
    '''if_stmt : SI PARIZQ expression PARDER statement
                / SI PARIZQ expression PARDER statement SI_NO statement'''
    condition = p[3]
    then_block = p[5]
    else_block = p[7] if len(p) == 8 else None
    p[0] = IfStmt(condition, then_block, else_block, lineno=p.lineno(1))

def p_function_decl(p):
    '''function_decl : type_spec ID PARIZQ parameter_list PARDER compound_stmt
                      / type_spec ID PARIZQ PARDER compound_stmt
                      / EXTERNO type_spec ID PARIZQ parameter_list PARDER PUNTOCOMA
                      / EXTERNO type_spec ID PARIZQ PARDER PUNTOCOMA'''
    is_extern = (p[1] == 'externo')

```

```
# ... construcción del nodo FunctionDecl
```

Nodos AST generados: - **Declaraciones:** Program, FunctionDecl, StructDecl, VarDecl - **Sentencias:** Block, IfStmt, WhileStmt, ForStmt, ReturnStmt, BreakStmt, ContinueStmt, PrintStmt, ExprStmt - **Expresiones:** BinaryOp, UnaryOp, Assignment, FunctionCall, MemberAccess, ArrayAccess, NewExpr, DeleteExpr - **Literales:** Identifier, IntLiteral, FloatLiteral, StringLiteral, CharLiteral, BoolLiteral - **Tipos:** Type (con is_pointer, is_array, dimensions)

Todos los nodos incluyen `lineno` y `lexpos` para reporte de errores.

Manejo de errores sintácticos:

```
def p_error(p):
    if p:
        print(f"Error sintáctico en línea {p.lineno}: token inesperado '{p.value}'")
    else:
        print("Error sintáctico: fin de archivo inesperado")
```

5.6.2 Analizador Semántico (semantic_analyzer.py)

Archivo: src/compiler/semantic_analyzer.py (701 líneas)

El analizador semántico valida la corrección del AST usando una **estrategia de dos pasadas** para permitir referencias forward (declaraciones usadas antes de definirse).

5.6.2.1 Arquitectura de Clases **SemanticAnalyzer**: Coordinador principal del análisis

```
class SemanticAnalyzer:
    def __init__(self):
        self.symbol_table = SymbolTable() # Gestión de scopes anidados
        self.errors = [] # Acumulador de mensajes de error
        self.structs = {} # Tracking global: nombre → StructDecl
        self.functions = {} # Tracking global: nombre → FunctionDecl

    def analyze(self, ast: Program) -> bool:
        """Retorna True si análisis exitoso, False si hay errores"""
```

5.6.2.2 Estrategia de Dos Pasadas **Pasada 1: Declaración de Símbolos Globales** - Registra todas las estructuras en `self.structs` - Registra todas las funciones en `self.functions` (incluyendo externas) - Calcula offsets y tamaños de miembros de estructuras - Permite referencias forward: funciones pueden llamarse entre sí

Pasada 2: Análisis de Cuerpos de Función - Analiza el cuerpo de cada función no externa - Valida tipos de expresiones y compatibilidad - Verifica que todas las variables usadas estén declaradas - Valida control de flujo (romper/continuar solo en loops) - Verifica que `retornar` coincida con tipo de retorno de función

5.6.2.3 Gestión de Tabla de Símbolos **SymbolTable**: Stack de scopes con búsqueda jerárquica

```
class SymbolTable:
    def __init__(self):
        self.global_scope = Scope(name="global")
        self.current_scope = self.global_scope
        self.scope_stack = [self.global_scope]

    def enter_scope(self, name="block", is_loop=False,
                   is_function=False, return_type=None):
        """Crear nuevo scope (función, bloque, loop)"""
```

```

def exit_scope(self):
    """Salir del scope actual"""

def lookup(self, name):
    """Buscar símbolo en scope actual y ancestros"""

```

Scope: Representa un ámbito léxico

```

class Scope:
    def __init__(self, parent=None, name="global"):
        self.parent = parent # Scope padre (None para global)
        self.name = name # Identificador para debugging
        self.symbols = {} # Diccionario: name → Symbol
        self.is_loop = False # True si es un mientras/para
        self.is_function = False # True si es cuerpo de función
        self.return_type = None # Tipo retorno (si is_function)

```

Symbol: Entrada en tabla de símbolos

```

class Symbol:
    def __init__(self, name, symbol_type, node, kind='variable'):
        self.name = name # Identificador del símbolo
        self.type = symbol_type # Objeto Type
        self.node = node # Nodo AST (VarDecl, FunctionDecl, etc.)
        self.kind = kind # 'variable', 'function', 'struct', 'parameter'
        self.is_const = False # True para constantes

```

5.6.2.4 Validaciones Realizadas

1. **Unicidad de Símbolos:** No redeclarar en mismo scope
2. **Existencia de Símbolos:** Variables/funciones usadas deben estar declaradas
3. **Tipos Compatibles:** Operaciones deben tener tipos válidos
 - Aritméticas: operandos numéricos
 - Comparaciones: tipos comparables
 - Asignaciones: tipo de rvalue compatible con lvalue
4. **Control de Flujo:**
 - romper/continuar solo dentro de loops (is_loop=True)
 - retornar solo en funciones
 - Tipo de valor retornado coincide con tipo de función
5. **Acceso a Estructuras:**
 - Miembro existe en la estructura
 - Tipo del objeto es estructura
 - Operador correcto (. para valor, -> para puntero)
6. **Arreglos Multidimensionales:**
 - Índices deben ser enteros
 - Número de dimensiones coincide con declaración
7. **Punteros:**
 - Operaciones de dereferencia (*p) requieren tipo puntero
 - Direcciones (&x) de lvalues válidos

5.6.2.5 Cálculo de Offsets de Estructuras

```

def _register_struct(self, node: StructDecl):
    """Registra estructura y calcula layout de memoria"""
    if node.name in self.structs:
        self.errors.append(f"Estructura '{node.name}' ya declarada")
    return

```

```

offset = 0
for member in node.members:
    member.offset = offset # Guardar offset en el nodo
    offset += self._get_type_size(member.type)

node.size = offset # Tamaño total de la estructura
self.structs[node.name] = node

```

Ejemplo: Estructura Punto con miembros entero4 x; entero4 y; - x.offset = 0 - y.offset = 4 - Punto.size = 8

5.6.2.6 Reporte de Errores Los errores se acumulan en `self.errors` con número de línea:

```

def _error(self, message, node=None):
    """Agregar error con información de línea"""
    if node and hasattr(node, 'lineno'):
        self.errors.append(f"Línea {node.lineno}: {message}")
    else:
        self.errors.append(message)

```

Ejemplos de mensajes: - "Línea 15: Variable 'x' no declarada" - "Línea 23: Función 'calcular' espera 2 argumentos, se pasaron 3" - "Línea 31: 'romper' fuera de un bucle" - "Línea 42: Incompatibilidad de tipos: no se puede asignar 'cadena' a 'entero4'"

5.6.3 Generador de Código (code_generator.py)

Archivo: src/compiler/code_generator.py (2368 líneas)

Traduce el AST validado a código ensamblador Atlas mediante recorrido del árbol, aplicando convenciones de la arquitectura.

5.6.3.1 Arquitectura y Convenciones Registros: - **R00-R13:** 14 registros temporales para evaluar expresiones - **R14 (BP):** Base Pointer, apunta al inicio del stack frame actual - **R15 (SP):** Stack Pointer, apunta al tope del stack - **R00:** También usado para valor de retorno de funciones

Organización de Memoria:

0x0000-0x0FFF: Código ejecutable y constantes (4KB)
 0x1000-0x7FFF: Variables globales y heap dinámico (28KB)
 0x8000-0xFFFF: Stack (32KB, crece hacia arriba)

Stack Frame de Función:

[BP+32]	Parámetro 3	+
[BP+24]	Parámetro 2	+-- Parámetros (offsets positivos)
[BP+16]	Parámetro 1	+
[BP+8]	Dirección retorno	← Guardada por CALL
[BP]	BP anterior	← Apunta BP actual
[BP-4]	Variable local 1	+
[BP-8]	Variable local 2	+-- Locales (offsets negativos)
[BP-12]	Variable local 3	+

Instrucciones con Sufijo de Tamaño:

El generador selecciona el sufijo correcto según el tipo:

Tipo SPL	Bytes	Sufijo	Ejemplo Instrucción
caracter, booleano	1	1	ADD1, LOAD1
entero2	2	2	ADD2, LOAD2
entero4, flotante	4	4	ADD4, FADD4
entero8, doble, punteros	8	8	ADD8, FADD8

Prefijo F para Punto Flotante: - FADD4, FSUB4, FMUL4, FDIV4: Operaciones flotantes de 32 bits - FADD8, FSUB8, FMUL8, FDIV8: Operaciones dobles de 64 bits

5.6.3.2 Clase CodeGenerator

```
class CodeGenerator:
    def __init__(self, ast, symbol_table):
        self.ast = ast # Árbol de sintaxis abstracta
        self.symbol_table = symbol_table # Tabla de símbolos
        self.code = [] # Acumulador de líneas de ensamblador

        # Gestión de registros temporales
        self.temp_counter = 0 # Índice del próximo temporal (0-13)
        self.max_temps = 14

        # Gestión de etiquetas
        self.label_counter = 0

        # Gestión de literales
        self.string_literals = {} # Cadenas constantes
        self.float_literals = {} # Constantes flotantes

    def generate(self) -> str:
        """Genera código ensamblador completo"""
        # Sección de datos
        self.emit(".data")
        self._generate_globals()
        self._generate_string_literals()

        # Sección de código
        self.emit(".text")
        for decl in self.ast.declarations:
            if isinstance(decl, FunctionDecl) and not decl.is_extern:
                self._generate_function(decl)

        return '\n'.join(self.code)
```

5.6.3.3 Generación de Funciones Prólogo estándar (establecer stack frame):

```
nombre_funcion:
    PUSH8 R14          ; Guardar BP anterior
    MOV8 R14, R15      ; BP = SP (nuevo frame)
    SUBV8 R15, <size>  ; Reservar espacio para locales
```

Epílogo estándar (limpiar y retornar):

```
MOV8 R15, R14      ; SP = BP (liberar locales)
POP8 R14           ; Restaurar BP anterior
```



```
RET ; Pop IP y retornar
```

Ejemplo completo de función:

Código SPL:

```
funcion entero4 suma(entero4 a, entero4 b) {  
    entero4 resultado;  
    resultado = a + b;  
    retornar resultado;  
}
```

Código Atlas generado:

```
suma:  
    ; Prólogo  
    PUSH8 R14 ; Guardar BP  
    MOV8 R14, R15 ; Establecer nuevo BP  
    SUBV8 R15, 4 ; Reservar 4 bytes para 'resultado'  
  
    ; resultado = a + b  
    LOAD4 R00, [R14+16] ; R00 = a (primer parámetro)  
    LOAD4 R01, [R14+24] ; R01 = b (segundo parámetro)  
    ADD4 R02, R00, R01 ; R02 = R00 + R01  
    STORE4 [R14-4], R02 ; resultado = R02  
  
    ; retornar resultado  
    LOAD4 R00, [R14-4] ; R00 = resultado (valor de retorno)  
  
    ; Epílogo  
    MOV8 R15, R14 ; Liberar locales  
    POP8 R14 ; Restaurar BP  
    RET
```

5.6.3.4 Gestión de Registros Temporales

```
def new_temp(self):  
    """Asignar próximo registro temporal"""  
    if self.temp_counter >= self.max_temps:  
        raise RuntimeError("Registros temporales agotados")  
    reg = self.temp_counter  
    self.temp_counter += 1  
    return reg  
  
def free_temps(self):  
    """Liberar todos los registros temporales"""  
    self.temp_counter = 0
```

Nota: Los registros se reutilizan entre sentencias. Dentro de una expresión compleja, pueden agotarse si hay más de 14 operaciones anidadas.

5.6.3.5 Generación de Expresiones Operaciones binarias:

```
def _generate_binary_op(self, node: BinaryOp):  
    """Generar código para operación binaria"""  
    # Evaluar operandos  
    left_reg = self.visit(node.left)  
    right_reg = self.visit(node.right)
```

```

# Determinar tamaño e instrucción
size = self._get_type_size(node.type)
is_float = self._is_float_type(node.type)

result_reg = self.new_temp()

# Generar instrucción según operador
if node.op == '+':
    instr = f"FADD{size}" if is_float else f"ADD{size}"
    self.emit(f"{instr} R{result_reg:02d}, R{left_reg:02d}, R{right_reg:02d}")
elif node.op == '-':
    instr = f"FSUB{size}" if is_float else f"SUB{size}"
    self.emit(f"{instr} R{result_reg:02d}, R{left_reg:02d}, R{right_reg:02d}")
# ... otros operadores

return result_reg

```

5.6.3.6 Estructuras de Control Sentencia si/si_no:

Código SPL:

```

si (x > 0) {
    imprimir("Positivo");
} si_no {
    imprimir("No positivo");
}

```

Código Atlas:

```

LOAD4 R00, [x]
CMPV4 R00, 0
JLE .L_else_1      ; Si x <= 0, saltar a else
; Bloque then
; ... código de imprimir("Positivo")
JMP .L_end_if_1
.L_else_1:
; Bloque else
; ... código de imprimir("No positivo")
.L_end_if_1:

```

Bucle mientras:

Código SPL:

```

mientras (i < 10) {
    suma = suma + i;
    i = i + 1;
}

```

Código Atlas:

```

.L_while_cond_1:
LOAD4 R00, [i]
CMPV4 R00, 10
JGE .L_while_end_1 ; Si i >= 10, salir
; Cuerpo del bucle
LOAD4 R01, [suma]

```

```

    ADD4 R02, R01, R00
    STORE4 [suma], R02
    ADDV4 R00, 1
    STORE4 [i], R00
    JMP .L_while_cond_1    ; Repetir
.L_while_end_1:

```

5.6.3.7 Arreglos Multidimensionales Declaración: entero4[3][4] matriz; (3 filas, 4 columnas)

Acceso: matriz[i][j]

Cálculo de offset: offset = (i x 4 + j) x sizeof(entero4)

Código generado:

```

LOAD4 R00, [i]          ; R00 = i
MOVV4 R01, 4            ; R01 = número de columnas
MUL4 R02, R00, R01      ; R02 = i x 4
LOAD4 R03, [j]          ; R03 = j
ADD4 R04, R02, R03      ; R04 = i x 4 + j
MOVV4 R05, 4            ; R05 = sizeof(entero4)
MUL4 R06, R04, R05      ; R06 = offset total
LOADV8 R07, matriz      ; R07 = dirección base
ADD8 R08, R07, R06      ; R08 = dirección final
LOAD4 R09, [R08]        ; R09 = matriz[i][j]

```

5.7 Ensamblador (ensamblador.py)

5.7.1 Análisis Sintáctico

```

class Assembler:
    def __init__(self):
        self.instructions = {}
        self.labels = {}
        self.current_address = 0

    def parse_instruction(self, line):
        # Eliminar comentarios
        if ';' in line:
            line = line[:line.index(';')]

        # Detectar etiquetas
        if ':' in line:
            label, instruction = line.split(':', 1)
            self.labels[label.strip()] = self.current_address
            line = instruction.strip()

        # Parsear instrucción
        parts = line.strip().split()
        if not parts:
            return None

        opcode = parts[0].upper()
        operands = parts[1:] if len(parts) > 1 else []

        return self.encode_instruction(opcode, operands)

```

5.7.2 Generación de Código

```
def encode_instruction(self, opcode, operands):
    instruction_info = self.get_instruction_info(opcode)
    format_type = instruction_info['format']
    opcode_value = instruction_info['opcode']

    if format_type == 'OP':
        return opcode_value << 48
    elif format_type == 'R':
        rd = self.parse_register(operands[0])
        return (opcode_value << 48) | (rd << 44)
    elif format_type == 'RR':
        rd = self.parse_register(operands[0])
        rs = self.parse_register(operands[1])
        return (opcode_value << 48) | (rd << 4) | rs
    # ... más formatos
```

5.8 Flujo de Ejecución

5.8.1 Diagrama de Flujo Principal

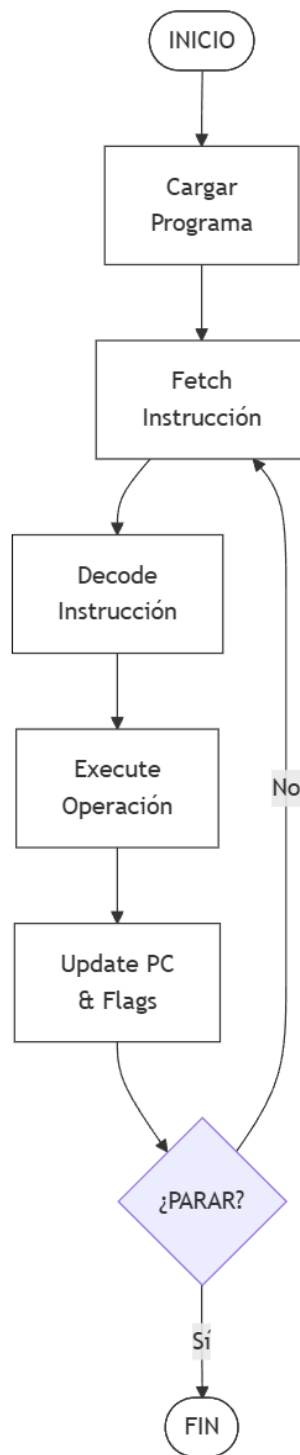


Figura 68: Diagrama de Flujo Principal

5.9 Interfaz Gráfica (GUI)

5.9.1 Componentes de la UI

1. **Visualizador de Estado**
 - Estado de registros R01-R15
 - Flags de estado (Z, N, C, V)
 - Contenido de memoria
 - Dispositivos de E/S
2. **Controles de Ejecución**
 - Ejecutar programa completo
 - Ejecución paso a paso
 - Parar/Continuar
 - Reset del sistema
3. **Monitor de E/S**
 - Visualización de salidas
 - Entrada de datos
 - Log de operaciones

Nota de implementación: la salida textual del programa (dispositivo de pantalla) se redirige a la interfaz gráfica y se muestra en el widget `texto_salida`. El `Screen` del emulador invoca un callback que inserta caracteres en `texto_salida` en tiempo real, permitiendo que las llamadas `imprimir(...)` o los `SHOWIO` se visualicen directamente en la GUI.

5.9.2 Visor de RAM en tiempo real (novedad)

Se incorporó un visor de RAM completo accesible desde la GUI mediante el botón “Ver RAM”. Este abre una ventana con una tabla (Treeview) que muestra la memoria en filas de 8 bytes, alineada a la arquitectura de 64 bits. Características principales:

- Visualización por filas de 8 bytes: columnas Addr, B0...B7.
- Auto-actualización opcional con intervalo configurable (ms).
- Scroll vertical y horizontal para navegar todo el espacio de memoria.
- Botón “Refrescar” para actualización manual.

Esta vista está conectada directamente a la RAM del simulador (`self.cpu.memory`), por lo que refleja en tiempo real los cambios que produce la ejecución del programa o las operaciones de memoria.

5.9.3 Limpieza de RAM desde la interfaz (novedad)

Se añadió el botón “Limpiar RAM” al examinador de memoria. Al activarlo:

- Se solicita confirmación al usuario.
- Se pone toda la RAM en cero de forma eficiente.
- Se persiste inmediatamente al archivo de memoria (ver sección 6.2.1).
- Si el visor de RAM está abierto, se refresca automáticamente para reflejar el estado limpio.

Además, se mejoró el layout del examinador de memoria para aprovechar mejor el ancho disponible (campos expandibles y alineados), facilitando lectura y edición.

5.10 Flujo del Compilador SPL (Preprocesador → Analizador Léxico → Sintáctico → IR → Opt → Ensamblador → Enlazador/Cargador)

El siguiente diagrama resume la cadena de herramientas para el SPL del Taller 1.

5.10.1 Pipeline general (fases)

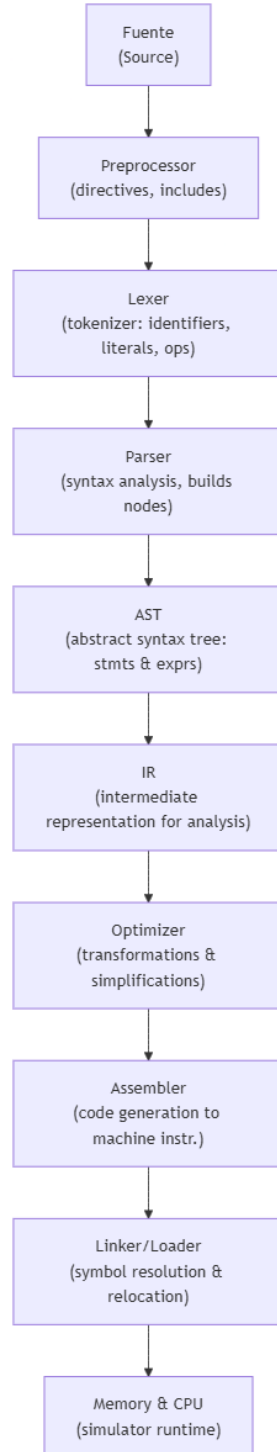


Figura 69: Pipeline general (fases)

Definiciones:

- Preproc: transforma y normaliza el código fuente (p. ej. directivas, includes, limpieza).

- Lexer: convierte texto en tokens (IDs, números, operadores).
- Parser: aplica la gramática y construye el AST.
- AST: representación en árbol de la estructura sintáctica.
- IR: representación intermedia para análisis/optimización.
- Opt.: optimizador opcional.
- Assembler: genera instrucciones binarias.
- Linker/Loader: reubica y combina módulos, carga en memoria.
- Mem/CPU: memoria y simulador donde se ejecuta el binario.

5.10.2 Flujo de tokens → árbol sintáctico → AST

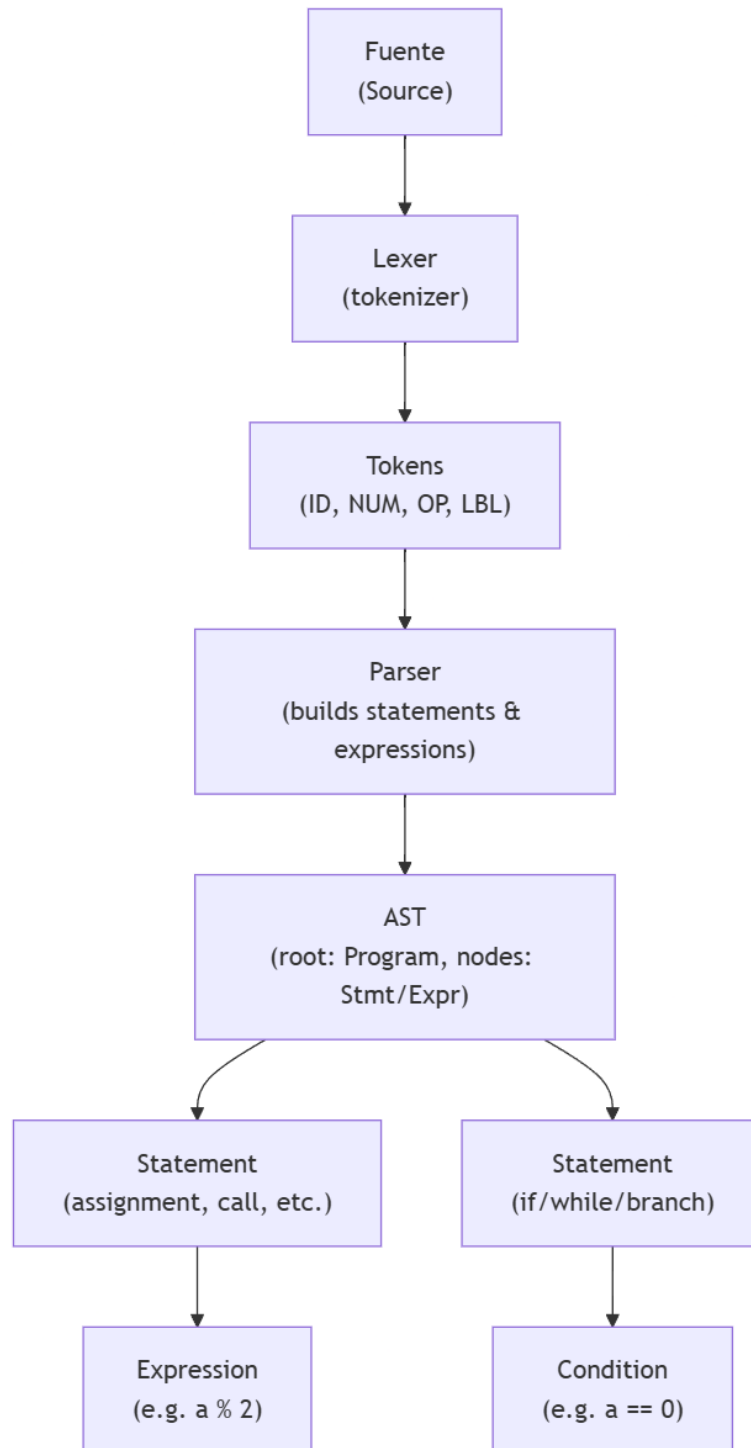


Figura 70: Flujo de tokens → árbol sintáctico → AST

Definiciones: + Parsing: proceso que toma tokens y aplica la gramática para reconocer construcciones (producciones) y crear nodos sintácticos. + Stmt (Statement): unidad ejecutable del programa, ej: asignación,

if, while, return. + AST: estructura de datos en forma de árbol que representa la estructura jerárquica del programa.

5.10.3 Optimización y generación final

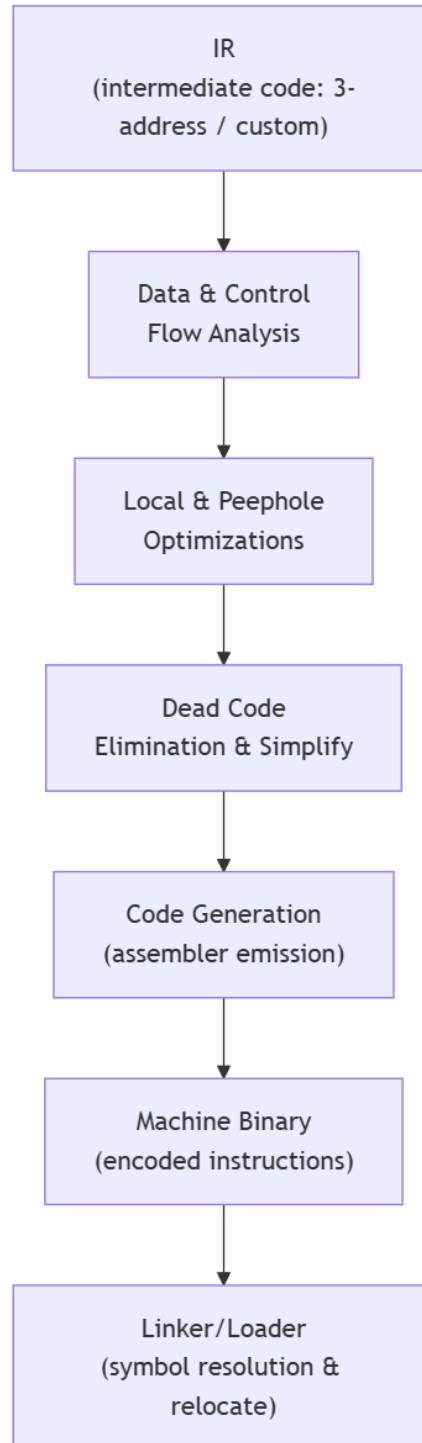


Figura 71: Optimización y generación final

Definiciones: + IR: formato intermedio entre parser y generador que permite análisis y transformaciones. + Análisis de flujo: computa dependencias/alcances (p. ej. dominadores, live-variables). + Optimización local: mejoras dentro de bloques (p. ej. const-fold). + Eliminación de código muerto: borrar instrucciones no usadas. + Codegen/Assembler: genera instrucciones binarias finales. + Binario: secuencia de instrucciones lista para cargar/ejecutar. + Linker: reubica símbolos y combina módulos en ejecutable.

Descripción rápida de cómo esto encaja con el repositorio actual:

- En esta entrega el ensamblador (`compiler/assembler.py`) implementa el generador de instrucción (CODEGEN) y el desensamblador. El Loader (`logic/Loader.py`) realiza la función del enlazador/cargador (reubicación simple, escritura a memoria y registro de programas cargados).

5.11 Especificación Léxica del Lenguaje

5.11.1 Keywords

- si
- si_no
- mientras
- para
- romper
- continuar
- vacio
- constante
- entero2
- entero4
- entero8
- caracter
- cadena
- con_signo
- sin_signo
- flotante
- doble
- booleano
- funcion
- retornar
- nuevo
- eliminar

5.11.2 Identificadores

Empiezan por letra mayúscula o minúscula.

- Nombres de variables
- Nombres de funciones

5.11.3 Constantes

- Enteros: 10, -25, 0xFF (decimal y hexadecimal)
- Flotantes: 3.14, 2.0e3
- Caracteres: 'A', '\n'
- Cadenas: "Hola, Mundo"
- Booleanos: True or False

5.11.4 Comentarios

- /*

- */
- //

5.11.5 Operadores

- Asignación y compuestos: =, +=, -=, *=, /=, %=
- Incremento / decremento: ++, --
- Aritméticos: +, -, *, /, %
- Bitwise: &, |, ^
- Comparación: ==, !=, <, <=, >, >=
- Lógicos: &&, ||

5.11.6 Delimitadores

- Llaves: { }
- Paréntesis: ()
- Punto y coma: ;
- Coma: ,
- Corchetes: []
- Comillas: " " y ' '

5.11.7 Expresiones regulares

- digito: [0-9]
- numero_entero: -? digito+
- numero_decimal: -? digito+ (\.digito+)?([eE] [-+]?{digito}+)?
- caracter: \'([^\'\\]|\\[ntr0'\'])\'
- cadena: \"([^\"\\]|\\.)*\" \"[^\"\\n]*\"
- booleano: verdadero | falso
- comentario: (\\/\\/[^\\n]*|\\/*([~*]|*+[^*/])**+\\/)
- identificadores: [a-zA-Z_][a-zA-Z0-9_]*

6. Manual Técnico y de Usuario

6.1 Instalación y Configuración

6.1.1 Requisitos del Sistema

Requisitos Mínimos: - Python 3.8 o superior - 512 MB de RAM - 100 MB de espacio en disco - Sistema operativo: Windows 10/11, Linux (Ubuntu 18.04+), macOS 10.14+

Requisitos Recomendados: - Python 3.10+ - 2 GB de RAM - Monitor con resolución mínima 1024x768 - Tarjeta gráfica con soporte para aceleración 2D

6.1.2 Proceso de Instalación

1. Descargar el proyecto

```
git clone https://github.com/usuario/maquina_lenguajes.git
cd maquina_lenguajes
```

2. Verificar Python

```
python --version # Debe ser 3.8+
```

3. Ejecutar simulador

```
python src/main.py
```

6.2 Manual de Usuario

6.2.1 Lenguaje de Programación SPL (Sistema de Procesamiento de Lenguajes)

SPL es el lenguaje de alto nivel diseñado específicamente para la arquitectura Atlas. Proporciona abstracciones modernas manteniendo control fino sobre los recursos del sistema.

6.2.1.1 Características del Lenguaje Sistema de Tipos Completo: - **Enteros:** entero1 (8 bits), entero2 (16 bits), entero4 (32 bits), entero8 (64 bits) - **Punto flotante:** flotante (32 bits IEEE 754), doble (64 bits IEEE 754) - **Otros tipos primitivos:** booleano, caracter, cadena - **Tipos compuestos:** estructuras (estructura), arreglos, punteros

Estructuras de Control: - Condicionales: si, si_no_si, si_no - Bucles: mientras, para, hacer_mientras - Control de flujo: romper, continuar, retornar

Funciones: - Declaración con tipo de retorno explícito - Paso de parámetros por valor - Soporte para recursión - Función especial principal() como punto de entrada

E/S Integrada: - imprimir(): Salida de datos (múltiples tipos soportados) - leer(): Entrada de datos del usuario

6.2.1.2 Gramática SPL (EBNF)

```
program ::= (function_decl | struct_decl)*

function_decl ::= 'funcion' type ID '(' parameter_list? ')' compound_stmt

struct_decl ::= 'estructura' ID '{' field_list '}' ';'

field_list ::= (type ID ';' )+

parameter_list ::= parameter (',' parameter)*
parameter ::= type ID
```

```

type ::= 'entero1' | 'entero2' | 'entero4' | 'entero8'
      | 'flotante' | 'doble'
      | 'booleano' | 'caracter' | 'cadena' | 'vacio'
      | struct_type | array_type | pointer_type

compound_stmt ::= '{' statement* '}'

statement ::= declaration_stmt
          | assignment_stmt
          | if_stmt
          | while_stmt
          | for_stmt
          | return_stmt
          | print_stmt
          | read_stmt
          | expression_stmt
          | compound_stmt

if_stmt ::= 'si' '(' expression ')' statement
         ('si_no_si' '(' expression ')' statement)*
         ('si_no' statement)?

while_stmt ::= 'mientras' '(' expression ')' statement

for_stmt ::= 'para' '(' init_expr ';' condition ';' update_expr ')' statement

return_stmt ::= 'retornar' expression? ';'

print_stmt ::= 'imprimir' '(' expression_list? ')' ';'

expression ::= assignment_expr
            | logical_or_expr

logical_or_expr ::= logical_and_expr ('||' logical_and_expr)*

logical_and_expr ::= equality_expr ('&&' equality_expr)*

equality_expr ::= relational_expr (('==' | '!=') relational_expr)*

relational_expr ::= additive_expr (('<' | '>' | '<=' | '>=') additive_expr)*

additive_expr ::= multiplicative_expr (('+' | '-') multiplicative_expr)*

multiplicative_expr ::= unary_expr (('*' | '/' | '%') unary_expr)*

unary_expr ::= ('+' | '-' | '!' | '~')? primary_expr

primary_expr ::= LITERAL | ID | function_call | '(' expression ')'

function_call ::= ID '(' argument_list? ')'

argument_list ::= expression (',' expression)*

```

6.2.1.3 Primer Programa en SPL Ejemplo 1: Hola Mundo

```
funcion entero4 principal() {
    imprimir("Hola desde Atlas!");
    imprimir();
    retornar 0;
}
```

Ejemplo 2: Operaciones Aritméticas

```
funcion entero4 principal() {
    entero4 a = 15;
    entero4 b = 7;
    entero4 suma = a + b;
    entero4 producto = a * b;

    imprimir("a = ", a);
    imprimir("b = ", b);
    imprimir("Suma: ", suma);
    imprimir("Producto: ", producto);
    imprimir();

    retornar suma;
}
```

Ejemplo 3: Condicionales y Bucles

```
funcion vacio ciclo() {
    entero2 i = 0;
    imprimir("Inicio del ciclo");
    imprimir();

    mientras (i < 10 && i != 5) {
        i += 1;
        si (i == 3 || i == 7) {
            imprimir("Saltando i = ", i);
            continuar;
        }
        imprimir("i = ", i);
    }

    imprimir("Fin del ciclo, i final = ", i);
    retornar;
}

funcion vacio principal() {
    ciclo();
}
```

Ejemplo 4: Funciones con Punto Flotante

```
funcion flotante area_circulo(flotante radio) {
    flotante pi = 3.14159;
    retornar pi * radio * radio;
}

funcion flotante volumen_esfera(flotante radio) {
```

```

    flotante pi = 3.14159;
    flotante temp = 4.0 / 3.0;
    retornar temp * pi * radio * radio * radio;
}

funcion entero4 principal() {
    flotante r = 5.0;

    imprimir("Radio: ", r);
    imprimir();

    flotante area = area_circulo(r);
    imprimir("Area del circulo: ", area);

    flotante volumen = volumen_esfera(r);
    imprimir("Volumen de la esfera: ", volumen);
    imprimir();

    flotante mayor;
    cadena mensaje;

    si (area > volumen){
        mayor = area;
        mensaje = "El area es mayor";
    } si_no {
        mayor = volumen;
        mensaje = "El volumen es mayor";
    }

    imprimir("Mayor valor: ", mayor);
    imprimir(mensaje);

    retornar 0;
}

```

Ejemplo 5: Algoritmo de Euclides en SPL

```

/*
 * Algoritmo de Euclides - Versión con Módulo
 * Calcula el Máximo Común Divisor (MCD) de dos números
 */

funcion entero4 euclides_modulo(entero4 a, entero4 b) {
    entero4 temp;

    mientras (b != 0) {
        temp = a% b;
        a = b;
        b = temp;
    }

    retornar a;
}

funcion entero4 principal() {

```



```

entero4 num1 = 1071;
entero4 num2 = 462;
entero4 mcd;

imprimir("Algoritmo de Euclides (modulo)");
imprimir();
imprimir("Calculando MCD de ", num1, num2);
imprimir();

mcd = euclides_modulo(num1, num2);

imprimir("MCD = ", mcd);
imprimir();

retornar mcd;
}

```

6.2.2 Uso de la Interfaz Gráfica (GUI)

6.2.2.1 Inicio de la Aplicación

1. Ejecutar el simulador

```

cd maquina_lenguajes
python src/main.py

```

2. Interfaz principal

- **Editor de código:** Panel izquierdo para escribir programas (SPL o Assembly)
- **Visualización de registros:** Muestra R00-R15 en tiempo real
- **Visor de memoria:** Tabla con direcciones y contenido en formato hexadecimal
- **Monitor de E/S:** Salida de dispositivos virtuales
- **Panel de flags:** Estados de Z (Zero), N (Negative), C (Carry), V (Overflow)
- **Controles de ejecución:** Botones para cargar, ejecutar, pausar, resetear

6.2.2.2 Flujo de Trabajo Típico A. Programar en SPL (Alto Nivel)

1. Escribir código SPL en el editor:

```

funcion entero4 factorial(entero4 n) {
    si (n <= 1) {
        retornar 1;
    }
    retornar n * factorial(n - 1);
}

funcion entero4 principal() {
    entero4 numero = 5;
    entero4 resultado = factorial(numero);
    imprimir("Factorial de ", numero, " es ", resultado);
    retornar resultado;
}

```

2. Hacer clic en “**Compilar**”

- El compilador ejecuta: Preprocesador → Lexer → Parser → Análisis Semántico → Generador de Código → Ensamblador
- Se genera código Assembly intermedio y luego bytecode binario
- Si hay errores, se muestran en el panel de mensajes con línea y tipo de error

3. Hacer clic en “Ejecutar”
 - El Loader carga el binario en memoria
 - La CPU ejecuta el programa instrucción por instrucción
 - Observar la salida en el Monitor de E/S: "Factorial de 5 es 120"
4. **Depuración paso a paso**
 - Usar botón “Paso” para ejecutar una instrucción a la vez
 - Observar cambios en registros, memoria y flags en cada paso
 - Verificar el flujo de ejecución y valores intermedios

B. Programar en Assembly (Bajo Nivel)

1. Escribir código Assembly en el editor:

```
; Calcular factorial de 5
MAIN:
    LOADV8 R01, 5          ; n = 5
    LOADV8 R02, 1          ; factorial = 1

FACTORIAL_LOOP:
    CMPV8 R01, 0           ; Comparar n con 0
    JEQ MOSTRAR           ; Si n == 0, mostrar resultado

    MUL8 R02, R01          ; factorial *= n
    DEC8 R01               ; n--
    JMP FACTORIAL_LOOP     ; Repetir

MOSTRAR:
    SVIO R02, 0x200        ; Guardar resultado en E/S
    SHOWIO 0x200           ; Mostrar factorial = 120
    PARAR                 ; Terminar
```

2. Hacer clic en “Ensamblar”
 - El ensamblador convierte mnemónicos a opcodes binarios
 - Se resuelven etiquetas y direcciones de memoria
3. Hacer clic en “Cargar” y luego “Ejecutar”
 - Observar cambios en tiempo real:
 - R01 decrementando: $5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0$
 - R02 acumulando: $1 \rightarrow 5 \rightarrow 20 \rightarrow 60 \rightarrow 120$
 - Flag Z activándose cuando R01 llega a 0

6.2.2.3 Características Avanzadas de la GUI Visualización de Memoria - Direcciones mostradas cada 8 bytes (alineadas a palabra de 64 bits): 0x0000, 0x0008, 0x0010... - Contenido en hexadecimal: 00 61 10 00 00 00 0A - Scroll para navegar por toda la memoria (64 KB implementados)

Persistencia de RAM - La memoria se guarda automáticamente en `src/memory_ram.txt` al cerrar - Se carga automáticamente al iniciar la aplicación - Formato humanamente legible:

```
0000: 00 00 00 00 00 00 00 00
0008: 00 61 10 00 00 00 00 0A
0010: 00 61 20 00 00 00 00 05
```

Botones de Control - **Compilar**: Compila código SPL a Assembly y luego a bytecode - **Ensamblar**: Convierte Assembly a bytecode (si ya está en assembly) - **Cargar**: Carga bytecode en memoria RAM - **Ejecutar**: Ejecuta programa hasta PARAR o error - **Paso**: Ejecuta una sola instrucción (depuración) - **Pausar**: Detiene ejecución en progreso - **Reset**: Reinicia CPU (registros, flags, PC a 0) - **Limpiar RAM**: Pone toda la memoria en 0

6.2.2.4 Ejemplo Completo: Uso de la GUI Escenario: Implementar y ejecutar algoritmo de Euclides

1. Escribir código SPL:

```
funcion entero4 euclides(entero4 a, entero4 b) {  
    mientras (b != 0) {  
        entero4 temp = a% b;  
        a = b;  
        b = temp;  
    }  
    retornar a;  
}
```

```
funcion entero4 principal() {  
    entero4 mcd = euclides(1071, 462);  
    imprimir("MCD = ", mcd);  
    retornar mcd;  
}
```

2. Compilar y Ejecutar:

- Clic en “Compilar” → Compilación exitosa
- Clic en “Ejecutar” → Observar salida: "MCD = 21"

3. Depuración Paso a Paso:

- Clic en “Reset” y “Cargar”
- Clic en “Paso” repetidamente
- Observar en registros:
 - Primera iteración: a=1071, b=462, temp=147
 - Segunda iteración: a=462, b=147, temp=21
 - Tercera iteración: a=147, b=21, temp=0
 - Cuarta iteración: b=0, bucle termina, retorna a=21

6.2.3 Inicio Rápido Assembly

6.2.3.1 Primer Programa Assembly

1. Abrir la aplicación ejecutando `python src/main.py` desde la raíz del repositorio
2. En el editor, escribir:

```
LOADV8 R01, 10      ; Cargar 10 en registro R01  
LOADV8 R02, 5       ; Cargar 5 en registro R02  
ADD8 R01, R02       ; Sumar R01 + R02, resultado en R01  
SVIO R01, 0x100     ; Guardar resultado en E/S dirección 0x100  
SHOWIO 0x100        ; Mostrar resultado en monitor  
PARAR               ; Terminar programa
```

3. Hacer clic en “Ensamblar”, luego “Cargar” y finalmente “Ejecutar”
4. Observar el resultado en el monitor de E/S: 15

6.2.4 Referencia del Lenguaje Assembly

6.2.4.1 Sintaxis General

[ETIQUETA:] INSTRUCCION [OPERANDO1] [, OPERANDO2] [; COMENTARIO]

Ejemplos válidos:

```
MAIN:                ; Etiqueta sola  
    LOADV8 R01, 42    ; Instrucción con dos operandos  
    ADD8 R01, R02     ; Operación registro-registro
```

```

JMP LOOP          ; Salto a etiqueta
PARAR             ; Instrucción sin operandos

```

6.2.4.2 Tipos de Operandos

- **Registros:** R00-R15 (notación alternativa: R0-R15)
 - R14: BP (Base Pointer, para stack frames)
 - R15: SP (Stack Pointer, tope de pila)
- **Valores inmediatos:**
 - Decimal: 123, 1071
 - Hexadecimal: 0x1A2B, 0xFF00
 - Binario: 0b1010, 0b11111111
- **Direcciones de memoria:** 0x1000, 375, etiquetas simbólicas
- **Etiquetas:** Nombres alfanuméricos con _, terminan con :
 - Válidos: LOOP:, FIN_GCD:, inicio_main:

6.2.4.3 Sufijos de Tamaño Todas las instrucciones llevan sufijo indicando el tamaño de operación: - **1:** 1 byte (8 bits) - ejemplo: ADD1, LOAD1 - **2:** 2 bytes (16 bits) - ejemplo: ADD2, STORE2 - **4:** 4 bytes (32 bits) - ejemplo: ADD4, MUL4 - **8:** 8 bytes (64 bits) - ejemplo: ADD8, DIV8

6.2.4.4 Instrucciones por Categoría Control de Flujo:

```

PARAR             ; Terminar programa
NOP               ; No operación
JMP etiqueta      ; Salto incondicional a dirección
JEQ etiqueta      ; Saltar si Z=1 (resultado cero)
JNE etiqueta      ; Saltar si Z=0 (resultado no cero)
JMI etiqueta      ; Saltar si N=1 (resultado negativo)
JPL etiqueta      ; Saltar si N=0 (resultado positivo)
JOV etiqueta      ; Saltar si V=1 (overflow)
JCAR etiqueta     ; Saltar si C=1 (carry)
CALL funcion      ; Llamar función (guarda dirección de retorno)
RET               ; Retornar de función

```

Operaciones Aritméticas (con sufijos 1/2/4/8):

```

; Ejemplos con 8 bytes (64 bits)
ADD8 R01, R02     ; R01 = R01 + R02
SUB8 R01, R02     ; R01 = R01 - R02
MUL8 R01, R02     ; R01 = R01 * R02 (sin signo)
MULS8 R01, R02    ; R01 = R01 * R02 (con signo)
DIV8 R01, R02     ; R01 = R01 / R02
MOD8 R01, R02     ; R01 = R01 % R02 (módulo)

```

```

; Con valores inmediatos
ADDV8 R01, 100    ; R01 = R01 + 100
SUBV8 R01, 50     ; R01 = R01 - 50
MULV8 R01, 3      ; R01 = R01 * 3

```

```

; Incremento/Decremento
INC8 R01          ; R01 = R01 + 1
DEC8 R01          ; R01 = R01 - 1
INC4 R02          ; R02 = R02 + 1 (32 bits)
DEC2 R03          ; R03 = R03 - 1 (16 bits)

```

Operaciones de Punto Flotante:

```

; Flotante de 32 bits (float)
FADD4 R01, R02      ; R01 = R01 + R02 (float)
FSUB4 R01, R02      ; R01 = R01 - R02 (float)
FMUL4 R01, R02      ; R01 = R01 * R02 (float)
FDIV4 R01, R02      ; R01 = R01 / R02 (float)

```

```

; Doble de 64 bits (double)
FADD8 R01, R02      ; R01 = R01 + R02 (double)
FSUB8 R01, R02      ; R01 = R01 - R02 (double)
FMUL8 R01, R02      ; R01 = R01 * R02 (double)
FDIV8 R01, R02      ; R01 = R01 / R02 (double)

```

Operaciones Lógicas:

```

AND8 R01, R02       ; R01 = R01 & R02 (bitwise AND)
OR8 R01, R02        ; R01 = R01 | R02 (bitwise OR)
XOR8 R01, R02       ; R01 = R01 ^ R02 (bitwise XOR)
NOT8 R01             ; R01 = ~R01 (bitwise NOT)

```

```

; Con valores inmediatos
ANDV8 R01, 0xFF      ; R01 = R01 & 0xFF (máscara)
ORV8 R01, 0x80       ; R01 = R01 | 0x80 (set bit)
XORV8 R01, 0xFF      ; R01 = R01 ^ 0xFF (invertir)

```

```

; Desplazamientos
SHL8 R01, R02        ; R01 = R01 << R02 (shift left)
SHR8 R01, R02        ; R01 = R01 >> R02 (shift right)
SHLV8 R01, 3         ; R01 = R01 << 3
SHRV8 R01, 2         ; R01 = R01 >> 2

```

Manejo de Memoria:

```

; Carga desde memoria
LOAD8 R01, 0x1000     ; R01 = Memoria[0x1000] (8 bytes)
LOAD4 R02, 0x2000     ; R02 = Memoria[0x2000] (4 bytes)
LOAD2 R03, 0x3000     ; R03 = Memoria[0x3000] (2 bytes)
LOAD1 R04, 0x4000     ; R04 = Memoria[0x4000] (1 byte)

```

```

; Carga de valores inmediatos
LOADV8 R01, 42        ; R01 = 42
LOADV4 R02, 0x1000    ; R02 = 0x1000
MOVV8 R01, 100        ; R01 = 100 (alias de LOADV)

```

```

; Almacenamiento en memoria
STORE8 R01, R02       ; Memoria[R02] = R01 (8 bytes)
STOREV8 R01, 0x2000   ; Memoria[0x2000] = R01
STORE4 R03, R04       ; Memoria[R04] = R03 (4 bytes)
STORE1 R05, R06       ; Memoria[R06] = R05 (1 byte)

```

```

; Otras operaciones de memoria
CLEAR8 R01            ; R01 = 0
CMP8 R01, R02         ; Comparar R01 con R02 (actualiza flags)
CMPV8 R01, 100        ; Comparar R01 con 100

```

Operaciones de Stack:

```

PUSH8 R01             ; [--SP] = R01, decrementar SP

```

```

POP8 R01          ; R01 = [SP++], incrementar SP
PUSH4 R02         ; Push de 4 bytes
POP2 R03          ; Pop de 2 bytes

```

Entrada/Salida:

```

SVIO R01, 0x100   ; IO[0x100] = R01
LOADIO R01, 0x100 ; R01 = IO[0x100]
SHOWIO 0x100      ; Mostrar IO[0x100] en monitor
CLRIO             ; Limpiar dispositivos entrada
RESETIO          ; Reset sistema E/S

```

Manipulación de Flags:

```

SZF              ; Set Zero Flag (Z = 1)
CZF              ; Clear Zero Flag (Z = 0)
SNF              ; Set Negative Flag (N = 1)
CNF              ; Clear Negative Flag (N = 0)
SCF              ; Set Carry Flag (C = 1)
CCF              ; Clear Carry Flag (C = 0)
SOF              ; Set Overflow Flag (V = 1)
COF              ; Clear Overflow Flag (V = 0)

```

6.2.5 Ejemplos Prácticos Assembly

6.2.5.1 Programa: Factorial de un Número

```

; Calcular factorial de 5 iterativamente
; Resultado esperado: 5! = 120

```

MAIN:

```

    LOADV8 R01, 5      ; n = 5
    LOADV8 R02, 1      ; factorial = 1

```

FACTORIAL_LOOP:

```

    CMPV8 R01, 0        ; Comparar n con 0
    JEQ MOSTRAR         ; Si n == 0, terminar

    MUL8 R02, R01        ; factorial *= n
    DEC8 R01             ; n--
    JMP FACTORIAL_LOOP   ; Repetir

```

MOSTRAR:

```

    SVIO R02, 0x200      ; Guardar resultado en E/S
    SHOWIO 0x200         ; Mostrar: 120
    PARAR               ; Terminar programa

```

Ejecución paso a paso: - Iteración 1: R01=5, R02=1 → R02=15=5, R01=4 - Iteración 2: R01=4, R02=5 → R02=54=20, R01=3 - Iteración 3: R01=3, R02=20 → R02=203=60, R01=2 - Iteración 4: R01=2, R02=60 → R02=602=120, R01=1 - Iteración 5: R01=1, R02=120 → R02=120*1=120, R01=0 - Comparación: R01=0 → JEQ salta a MOSTRAR - Salida: 120

6.2.5.2 Programa: Algoritmo de Euclides

```

; Algoritmo de Euclides - Calcular MCD(1071, 462)
; Resultado esperado: MCD = 21

```

```

; Inicializar datos de prueba

```

MAIN:

```

    LOADV8 R01, 1071      ; A = 1071
    STOREV8 R01, 0x1000   ; Guardar A en memoria[0x1000]
    LOADV8 R02, 462       ; B = 462
    STOREV8 R02, 0x1008   ; Guardar B en memoria[0x1008]

; Cargar valores para algoritmo
    LOAD8 R01, 0x1000     ; Cargar A
    LOAD8 R02, 0x1008     ; Cargar B

EUCLIDES_LOOP:
    CLEAR8 R00            ; R00 = 0
    CMP8 R02, R00         ; Comparar B con 0
    JEQ FIN_GCD           ; Si B == 0, terminar

    ; Calcular A% B manualmente
    CLEAR8 R03
    ADD8 R03, R01         ; R03 = A
    DIV8 R03, R02         ; R03 = A / B (cociente)
    MUL8 R03, R02         ; R03 = (A/B) * B
    SUB8 R01, R03         ; R01 = A - (A/B)*B = A% B

    ; Intercambiar: A = B, B = resto
    CLEAR8 R03
    ADD8 R03, R01         ; R03 = resto
    CLEAR8 R01
    ADD8 R01, R02         ; R01 = B (nuevo A)
    CLEAR8 R02
    ADD8 R02, R03         ; R02 = resto (nuevo B)

    JMP EUCLIDES_LOOP     ; Repetir

FIN_GCD:
    STOREV8 R01, 0x2000   ; Guardar MCD en memoria[0x2000]

    ; Mostrar resultados
    LOAD8 R01, 0x1000     ; Cargar A original
    SVIO R01, 0x700       ;
    SHOWIO 0x700          ; Mostrar A = 1071

    LOAD8 R01, 0x1008     ; Cargar B original
    SVIO R01, 0x701       ;
    SHOWIO 0x701          ; Mostrar B = 462

    LOAD8 R01, 0x2000     ; Cargar MCD
    SVIO R01, 0x702       ;
    SHOWIO 0x702          ; Mostrar MCD = 21

    PARAR

```

Trazado de ejecución: - Inicio: A=1071, B=462 - Iteración 1: resto=1071%462=147, A=462, B=147 - Iteración 2: resto=462%147=21, A=147, B=21 - Iteración 3: resto=147%21=0, A=21, B=0 - B==0 → Termina, MCD=21

6.2.5.3 Programa: Búsqueda del Máximo

```

; Encontrar el máximo de tres números
; Entrada: 25, 18, 31
; Resultado esperado: 31
MAIN:
    LOADV8 R01, 25          ; primer número
    LOADV8 R02, 18          ; segundo número
    LOADV8 R03, 31          ; tercer número
    CLEAR8 R04              ; máximo actual = 0

    ; Comparar R01 con máximo
    CMP8 R01, R04           ; flags según (R01 - R04)
    JMI NO_ACTUALIZAR1      ; Si R01 < R04, no actualizar
    CLEAR8 R04
    ADD8 R04, R01           ; R04 = R01

NO_ACTUALIZAR1:
    ; Comparar R02 con máximo
    CMP8 R02, R04
    JMI NO_ACTUALIZAR2
    CLEAR8 R04
    ADD8 R04, R02           ; R04 = R02

NO_ACTUALIZAR2:
    ; Comparar R03 con máximo
    CMP8 R03, R04
    JMI MOSTRAR
    CLEAR8 R04
    ADD8 R04, R03           ; R04 = R03

MOSTRAR:
    SVIO R04, 0x300         ; Guardar máximo
    SHOWIO 0x300            ; Mostrar: 31
    PARAR

```

6.2.5.4 Programa: Uso de Librerías con Preprocesador

```

; Ejemplo simple usando constantes de librerías
; Demuestra #include y #define

#include "lib/math.asm"      ; Incluye constantes PI, E
#include "lib/io.asm"        ; Incluye IO_OUTPUT_1, IO_OUTPUT_2

; Calcular cuadrado de 5
MAIN:
    LOADV8 R01, 5
    MUL8 R01, R01           ; R01 = 25
    SVIO R01, IO_OUTPUT_1
    SHOWIO IO_OUTPUT_1      ; Mostrar: 25

    ; Duplicar un número
    LOADV8 R02, 7
    ADD8 R02, R02           ; R02 = 14
    SVIO R02, IO_OUTPUT_2
    SHOWIO IO_OUTPUT_2      ; Mostrar: 14

```



```

; Usar constante PI (3.14159 = 0x40490FDB en float)
LOADV4 R03, PI
SVIO R03, IO_OUTPUT_1
SHOWIO IO_OUTPUT_1    ; Mostrar: 3.14159

```

```
PARAR
```

6.3 Manual Técnico

6.3.1 API del Sistema

6.3.1.1 Clase CPU

```

class CPU:
    def __init__(self, memory_size=25000):
        """Inicializar CPU con tamaño de memoria especificado"""

    def load_program(self, program_bytes, start_address=0):
        """Cargar programa en memoria"""

    def step(self):
        """Ejecutar una instrucción"""

    def run(self):
        """Ejecutar programa hasta PARAR"""

    def reset(self):
        """Reiniciar estado del procesador"""

    def get_state(self):
        """Obtener estado completo del sistema"""

```

6.3.1.2 Clase Assembler

```

class Assembler:
    def assemble(self, source_code):
        """Ensamblar código fuente a bytecode"""

    def parse_line(self, line):
        """Parsear línea individual de assembly"""

    def resolve_labels(self):
        """Resolver direcciones de etiquetas"""

```

6.3.2 Estructura de Datos Interna

6.3.2.1 Formato de Instrucción en Memoria

```

# Instrucción de 64 bits almacenada como entero
instruction = (opcode << 48) | (operand1 << 32) | operand2

# Extracción de campos
opcode = (instruction >> 48) & 0xFFFF
rd = (instruction >> 44) & 0xF

```

```
rs = instruction & 0xF
immediate = instruction & 0xFFFFFFFF
```

6.3.2.2 Estado del Procesador

```
cpu_state = {
    'registers': [0] * 16,      # R00-R15
    'flags': {
        'Z': False,           # Zero flag
        'N': False,           # Negative flag
        'C': False,           # Carry flag
        'V': False            # Overflow flag
    },
    'pc': 0,                   # Program counter
    'memory': Memory(),        # Sistema de memoria
    'io': IOSystem(),          # Sistema E/S
    'running': False           # Estado de ejecución
}
```

6.3.3 Extensibilidad

6.3.3.1 Agregar Nueva Instrucción

1. Definir opcode en `instruction_set.py`
2. Implementar lógica en `CPU.execute_instruction()`
3. Agregar parsing en `Assembler.parse_instruction()`
4. Actualizar documentación

6.3.3.2 Ejemplo de Extensión

```
# Agregar instrucción SQRT (raíz cuadrada)
def execute_sqrt(self, operands):
    rd = operands['rd']
    value = self.registers[rd]
    result = int(math.sqrt(value))
    self.registers[rd] = result
    self.update_flags(result)
```

7. Especificaciones Técnicas

7.1 Arquitectura del Procesador

7.1.1 Especificaciones Generales

Característica	Especificación
Arquitectura	64 bits, RISC
Endianness	Little-endian
Tamaño de palabra	64 bits (8 bytes)
Bus de direcciones	44 bits (16 TB direccionables teóricos, 64 KB implementados)
Bus de datos	64 bits
Registros generales	16 (R00-R15)
Tamaño de registro	64 bits cada uno
Modelo de memoria	von Neumann unificado
Instrucciones totales	137+ instrucciones con variantes de tamaño
Formato de instrucción	Longitud variable (1-16 bytes)
Frecuencia de reloj	Simulada (sin límite físico)
Pipeline	No implementado (ejecución secuencial)

7.1.2 Conjunto de Instrucciones Completo (ISA)

El ISA de Atlas sigue el principio RISC con sufijos de tamaño explícitos. Cada instrucción que opera sobre datos lleva sufijo **1**, **2**, **4**, **u** **8** indicando bytes (8, 16, 32, o 64 bits respectivamente).

Formatos de instrucción: - **OP**: Sin operandos (ej: PARAR, NOP) - **R**: Un registro (ej: PUSH8 R01, NOT4 R02) - **RR**: Dos registros (ej: ADD8 R01, R02) - **RI**: Registro e inmediato (ej: LOADV8 R01, 100) - **I**: Solo inmediato o dirección (ej: JMP 0x1000, CALL funcion)

7.1.2.1 Instrucciones de Control de Flujo (14)

Mnemónico	Sufijo	Formato	Descripción	Ejemplo
PARAR	-	OP	Detener ejecución del programa	PARAR
NOP	-	OP	No operación (ciclo vacío)	NOP
JMP	-	I	Salto incondicional a dirección	JMP 0x1000
JEQ	-	I	Saltar si Zero Flag (Z=1)	JEQ FIN
JNE	-	I	Saltar si not zero (Z=0)	JNE LOOP
JMI	-	I	Saltar si negativo (N=1)	JMI NEGATIVO
JPL	-	I	Saltar si positivo (N=0)	JPL POSITIVO
JOV	-	I	Saltar si overflow (V=1)	JOV ERROR
JCAR	-	I	Saltar si carry (C=1)	JCAR ACARREO
JNCAR	-	I	Saltar si no carry (C=0)	JNCAR SIN_ACARREO
JGT	-	I	Saltar si mayor (N=V, Z=0)	JGT MAYOR
JLT	-	I	Saltar si menor (N!=V)	JLT MENOR
CALL	-	I	Llamar función (push PC, jump)	CALL funcion
RET	-	OP	Retornar de función (pop PC)	RET

Notas: - Los saltos usan direcciones absolutas o etiquetas resueltas por el ensamblador - Flags actualizados por instrucciones aritméticas/lógicas y CMP - CALL/RET gestionan automáticamente el stack de retorno

7.1.2.2 Instrucciones Aritméticas Enteras (48) Todas con sufijos **1, 2, 4, 8** → 6 instrucciones x 4 tamaños = 24 básicas + 24 con inmediatos = 48

Mnemónico Base	Sufijo	Formato	Descripción	Ejemplo 8 bytes
ADD	1/2/4/8	RR	$Rd = Rd + Rs$ (sin signo)	ADD8 R01, R02
ADDV	1/2/4/8	RI	$Rd = Rd +$ inmediato	ADDV8 R01, 100
SUB	1/2/4/8	RR	$Rd = Rd - Rs$	SUB8 R01, R02
SUBV	1/2/4/8	RI	$Rd = Rd -$ inmediato	SUBV8 R01, 50
MUL	1/2/4/8	RR	$Rd = Rd * Rs$ (sin signo)	MUL8 R01, R02
MULV	1/2/4/8	RI	$Rd = Rd *$ inmediato	MULV8 R01, 3
MULS	1/2/4/8	RR	$Rd = Rd * Rs$ (con signo)	MULS8 R01, R02
DIV	1/2/4/8	RR	$Rd = Rd / Rs$ (cociente)	DIV8 R01, R02
DIVV	1/2/4/8	RI	$Rd = Rd /$ inmediato	DIVV8 R01, 2
MOD	1/2/4/8	RR	$Rd = Rd \% Rs$ (resto)	MOD8 R01, R02
MODV	1/2/4/8	RI	$Rd = Rd \%$ inmediato	MODV8 R01, 10
INC	1/2/4/8	R	$Rd = Rd + 1$	INC8 R01
DEC	1/2/4/8	R	$Rd = Rd - 1$	DEC8 R01
NEG	1/2/4/8	R	$Rd = -Rd$ (complemento a 2)	NEG8 R01

Flags afectados: Z (zero), N (negative), C (carry), V (overflow)

7.1.2.3 Instrucciones de Punto Flotante (16) Implementan estándar IEEE 754 para precisión simple (32 bits) y doble (64 bits)

Mnemónico	Sufijo	Formato	Descripción	Ejemplo
FADD	4/8	RR	$Rd = Rd + Rs$ (float/double)	FADD8 R01, R02
FSUB	4/8	RR	$Rd = Rd - Rs$	FSUB4 R01, R02
FMUL	4/8	RR	$Rd = Rd * Rs$	FMUL8 R01, R02
FDIV	4/8	RR	$Rd = Rd / Rs$	FDIV4 R01, R02
FADDV	4/8	RI	$Rd = Rd +$ inmediato_float	FADDV8 R01, 3.14
FSUBV	4/8	RI	$Rd = Rd -$ inmediato_float	FSUBV4 R01, 2.5
FMULV	4/8	RI	$Rd = Rd *$ inmediato_float	FMULV8 R01, 0.5
FDIVV	4/8	RI	$Rd = Rd /$ inmediato_float	FDIVV4 R01, 2.0

Sufijos: - **4:** float de 32 bits (precisión simple) - **8:** double de 64 bits (precisión doble)

Flags especiales FPU: NaN (Not a Number), Inf (Infinito), Precisión

7.1.2.4 Instrucciones Lógicas y Bitwise (40)

Mnemónico Base	Sufijo	Formato	Descripción	Ejemplo
AND	1/2/4/8	RR	$Rd = Rd \& Rs$ (AND)	AND8 R01, R02
ANDV	1/2/4/8	RI	$Rd = Rd \&$ inmediato	ANDV8 R01, 0xFF
OR	1/2/4/8	RR	$Rd = Rd Rs$ (OR)	OR8 R01, R02
ORV	1/2/4/8	RI	$Rd = Rd $ inmediato	ORV8 R01, 0x80
XOR	1/2/4/8	RR	$Rd = Rd \wedge Rs$ (XOR)	XOR8 R01, R02
XORV	1/2/4/8	RI	$Rd = Rd \wedge$ inmediato	XORV8 R01, 0xFF

Mnemónico Base	Sufijo	Formato	Descripción	Ejemplo
NOT	1/2/4/8	R	Rd = ~Rd (NOT)	NOT8 R01
SHL	1/2/4/8	RR	Rd = Rd « Rs (shift left)	SHL8 R01, R02
SHLV	1/2/4/8	RI	Rd = Rd « inmediato	SHLV8 R01, 3
SHR	1/2/4/8	RR	Rd = Rd » Rs (shift right lógico)	SHR8 R01, R02
SHRV	1/2/4/8	RI	Rd = Rd » inmediato	SHRV8 R01, 2
SAR	1/2/4/8	RR	Rd = Rd » Rs (shift aritmético)	SAR8 R01, R02
ROL	1/2/4/8	RR	Rotar izquierda	ROL8 R01, R02
ROR	1/2/4/8	RR	Rotar derecha	ROR8 R01, R02

Total: 10 operaciones x 4 tamaños = 40 instrucciones

7.1.2.5 Instrucciones de Memoria (32)

Mnemónico Base	Sufijo	Formato	Descripción	Ejemplo
LOAD	1/2/4/8	RI	Rd = Memoria[direccion]	LOAD8 R01, 0x1000
LOADR	1/2/4/8	RR	Rd = Memoria[Rs]	LOADR8 R01, R02
LOADV	1/2/4/8	RI	Rd = inmediato (carga directa)	LOADV8 R01, 42
MOVV	1/2/4/8	RI	Alias de LOADV	MOVV8 R01, 100
STORE	1/2/4/8	RR	Memoria[Rs] = Rd	STORE8 R01, R02
STOREV	1/2/4/8	RI	Memoria[direccion] = Rd	STOREV8 R01, 0x2000
CLEAR	1/2/4/8	R	Rd = 0	CLEAR8 R01
MOV	1/2/4/8	RR	Rd = Rs (copiar registro)	MOV8 R01, R02

Total: 8 operaciones x 4 tamaños = 32 instrucciones

Modos de direccionamiento: - Directo: LOAD8 R01, 0x1000 → R01 = Memoria[0x1000] - Registro indirecto: LOADR8 R01, R02 → R01 = Memoria[R02] - Inmediato: LOADV8 R01, 42 → R01 = 42

7.1.2.6 Instrucciones de Comparación (8)

Mnemónico	Sufijo	Formato	Descripción	Ejemplo
CMP	1/2/4/8	RR	Comparar Rd - Rs (actualiza flags)	CMP8 R01, R02
CMPV	1/2/4/8	RI	Comparar Rd - inmediato	CMPV8 R01, 100

Total: 2 x 4 tamaños = 8 instrucciones

Comportamiento: Realiza resta $Rd - Rs$ pero **no** modifica Rd , solo actualiza flags: - **Z**: Activado si $Rd == Rs$ - **N**: Activado si $Rd < Rs$ (resultado negativo) - **C**: Activado si hay préstamo ($Rd < Rs$ sin signo) - **V**: Activado si overflow en resta con signo

7.1.2.7 Instrucciones de Stack (8)

Mnemónico	Sufijo	Formato	Descripción	Ejemplo
PUSH	1/2/4/8	R	$[-SP] = Rd$; $SP -= \text{tamaño}$	PUSH8 R01
POP	1/2/4/8	R	$Rd = [SP]$; $SP += \text{tamaño}$	POP8 R01

Total: 2×4 tamaños = 8 instrucciones

Convenciones de stack: - **SP (R15)**: Stack Pointer, apunta al tope de la pila - **BP (R14)**: Base Pointer, apunta a la base del stack frame actual - Crece hacia **direcciones mayores** (PUSH incrementa SP) - Stack empieza en 0x8000 (configurable)

7.1.2.8 Instrucciones de E/S (5)

Mnemónico	Formato	Descripción	Ejemplo
SVIO	RI	$IO[\text{direccion}] = Rd$	SVIO R01, 0x100
LOADIO	RI	$Rd = IO[\text{direccion}]$	LOADIO R01, 0x100
SHOWIO	I	Mostrar $IO[\text{direccion}]$ en monitor	SHOWIO 0x100
CLRIO	OP	Limpiar buffers entrada	CLRIO
RESETIO	OP	Reiniciar sistema E/S	RESETIO

Direcciones reservadas: - 0x8000-0x80FF: Pantalla virtual (256 bytes) - 0x8100-0x81FF: Teclado virtual (256 bytes) - 0x8200-0x82FF: Almacenamiento temporal (256 bytes) - 0x8300-0x83FF: Debugging/Logging (256 bytes)

7.1.2.9 Instrucciones de Manipulación de Flags (8)

Mnemónico	Descripción	Ejemplo
SZF	Set Zero Flag ($Z = 1$)	SZF
CZF	Clear Zero Flag ($Z = 0$)	CZF
SNF	Set Negative Flag ($N = 1$)	SNF
CNF	Clear Negative Flag ($N = 0$)	CNF
SCF	Set Carry Flag ($C = 1$)	SCF
CCF	Clear Carry Flag ($C = 0$)	CCF
SOF	Set Overflow Flag ($V = 1$)	SOF
COF	Clear Overflow Flag ($V = 0$)	COF

Uso típico: Manipulación manual de flags para algoritmos especializados o testing

7.1.2.10 Resumen del Conjunto de Instrucciones

Categoría	Cantidad	Observaciones
Control de flujo	14	Salto, llamadas, retornos
Aritméticas enteras	48	12 tipos \times 4 tamaños

Categoría	Cantidad	Observaciones
Punto flotante	16	8 tipos x 2 tamaños (float, double)
Lógicas/Bitwise	40	10 tipos x 4 tamaños
Memoria	32	8 tipos x 4 tamaños
Comparación	8	2 tipos x 4 tamaños
Stack	8	2 tipos x 4 tamaños
E/S	5	Memory-mapped I/O
Manipulación flags	8	Control manual de ZNCV
TOTAL	179	Conjunto completo implementado

Nota sobre variantes: El total incluye todas las combinaciones de sufijos. En la práctica, el ISA es extensible y algunos sufijos pueden no estar implementados para ciertas instrucciones (ej: NEG solo implementado para 4 y 8 bytes).

7.2 Sistema de Memoria

7.2.1 Configuración de Memoria

```
MEMORY_LAYOUT = {
    'PROGRAM_START': 0x0000,    # Inicio de código
    'DATA_START': 0x4000,      # Inicio de datos
    'STACK_START': 0x6000,     # Inicio de pila
    'IO_START': 0x8000,        # E/S mapeada
    'MEMORY_SIZE': 25000       # Tamaño total
}
```

7.2.2 Mapa de Memoria

```
0x0000 +-----+
      |  Código Programa  |
      |      (16 KiB)     |
0x4000 +-----+
      |  Área de Datos    |
      |      (8 KiB)     |
0x6000 +-----+
      |      Pila        |
      |      (8 KiB)     |
0x8000 +-----+
      |  E/S Mapeada     |
      | (resto hasta 64KiB) |
0xFFFF +-----+
```

7.2.3 Persistencia de la RAM en archivo de texto (novedad)

La RAM del simulador ahora es persistente entre ejecuciones mediante un archivo de texto humanamente legible. Esto permite reanudar sesiones y depurar estados de memoria con facilidad.

- Ubicación del archivo: junto a `main.py`, con nombre `memory_ram.txt`.
- Ruta anclada de forma absoluta para evitar duplicados por cambios del directorio de trabajo.
- Carga automática al iniciar la aplicación (si el archivo existe) y guardado automático al salir.

Fragmento relevante en `main.py`:

```
from GUI.GUI import SimuladorGUI
from logic.Memory import Memory
```

```

from logic.CPU import CPU
import os

# Usar una ruta explícita para evitar archivos duplicados de RAM
BASE_DIR = os.path.dirname(os.path.abspath(__file__))
RAM_FILE = os.path.join(BASE_DIR, "memory_ram.txt")

mem = Memory(2**16, memory_file=RAM_FILE)
cpu = CPU(mem)
app = SimuladorGUI(cpu)
app.mainloop()

```

7.2.3.1 Formato del archivo de memoria El archivo se escribe en líneas de 8 bytes (64 bits), alineado con el tamaño de palabra de la arquitectura. Ejemplo de línea:

```

0000: 00 00 00 00 00 00 00 00
0008: 00 61 10 00 00 00 00 0A
0010: 00 61 20 00 00 00 00 05

```

- La parte izquierda muestra la dirección base de la fila (en hex). Cada fila avanza de 8 en 8 bytes.
- A la derecha se listan 8 bytes en hexadecimal, separados por espacio.
- Se permiten comentarios con # o ;, que se ignoran al leer.

7.2.3.2 Carga/guardado y API de memoria La clase `logic/Memory.py` expone operaciones para persistencia y mantenimiento:

- `load_from_txt(path: str)`: carga el contenido del archivo a la RAM, ignorando comentarios y direcciones si están presentes.
- `save_to_txt(path: str)`: vuelca toda la RAM al archivo en el formato anterior.
- `clear()`: pone toda la memoria en cero de forma eficiente.

Estas funciones se integran con la GUI: al limpiar la RAM desde el botón correspondiente, se ejecuta `clear()` y se guarda de inmediato en `memory_ram.txt`.

7.2.3.3 Alineamiento de direcciones en el visor En la tabla del visor, las direcciones avanzan como 0x0000, 0x0008, 0x0010, ... Esto se debe a que la arquitectura opera con palabras de 64 bits (8 bytes), por lo que la presentación por filas de 8 bytes facilita la lectura de instrucciones y datos alineados.

7.3 Sistema de E/S

7.3.1 Direcciones de E/S Reservadas

Dirección	Propósito	Acceso
0x8000-0x80FF	Pantalla virtual	R/W
0x8100-0x81FF	Teclado virtual	R
0x8200-0x82FF	Almacenamiento temporal	R/W
0x8300-0x83FF	Debugging/Logging	W

7.3.2 Protocolo de Comunicación

```

# Escribir a dispositivo
SVIO R1, 0x8000    # IO[0x8000] = R1

# Leer de dispositivo
LOADIO R1, 0x8100  # R1 = IO[0x8100]

```



```
# Mostrar en pantalla  
SHOWIO 0x8000      # Mostrar contenido de IO[0x8000]
```

8. Documentación de Experimentación y Resultados

Se presentan los experimentos realizados sobre el analizador lexico desarrollado, con el objetivo de probar y validar el correcto funcionamiento del analizador en la identificación de categorías lexicas y su clasificación en tokens. Para esto en cada escenario se procesa una cadena de caracteres correspondiente a un programa en alto nivel, que debe ser aceptado por el analizador lexico.

8.1 Escenario 1

Para este escenario se toma la cadena de caracteres correspondiente al un código en alto nivel que determina si un entero es par o impar

```
funcion entero4 espar(entero4 a) {  
    a = a%2;  
    booleano resultado = 0;  
    si(a%2 == 0){  
        resultado = 1;  
    }  
    si_no{  
        resultado = 0;  
    }  
    retornar resultado;  
}
```

Retornando como resultado del analisis

```
LexToken(FUNCION, 'funcion', 3, 30)  
LexToken(ENTERO4, 'entero4', 3, 38)  
LexToken(ID, 'espar', 3, 46)  
LexToken(PARIZQ, '(', 3, 51)  
LexToken(ENTERO4, 'entero4', 3, 52)  
LexToken(ID, 'a', 3, 60)  
LexToken(PARDER, ')', 3, 61)  
LexToken(LLAVEIZQ, '{', 3, 63)  
LexToken(ID, 'a', 4, 73)  
LexToken(ASIGNAR, '=', 4, 75)  
LexToken(ID, 'a', 4, 77)  
LexToken(MOD, '%', 4, 78)  
LexToken(ENTERO, 2, 4, 79)  
LexToken(PUNTOCOMA, ';', 4, 80)  
LexToken(BOOLEANO, 'booleano', 5, 90)  
LexToken(ID, 'resultado', 5, 99)  
LexToken(ASIGNAR, '=', 5, 109)  
LexToken(ENTERO, 0, 5, 111)  
LexToken(PUNTOCOMA, ';', 5, 112)  
LexToken(SI, 'si', 6, 122)  
LexToken(PARIZQ, '(', 6, 124)  
LexToken(ID, 'a', 6, 125)  
LexToken(MOD, '%', 6, 126)  
LexToken(ENTERO, 2, 6, 127)  
LexToken(IGUAL, '==', 6, 129)  
LexToken(ENTERO, 0, 6, 132)  
LexToken(PARDER, ')', 6, 133)  
LexToken(LLAVEIZQ, '{', 6, 134)  
LexToken(ID, 'resultado', 7, 148)  
LexToken(ASIGNAR, '=', 7, 158)
```

```

LexToken(ENTERO,1,7,160)
LexToken(PUNTOCOMA,','; ',7,161)
LexToken(LLAVEDER,'}',8,171)
LexToken(SINO,'si_no',9,181)
LexToken(LLAVEIZQ,'{',9,186)
LexToken(ID,'resultado',10,200)
LexToken(ASIGNAR,'=',10,210)
LexToken(ENTERO,0,10,212)
LexToken(PUNTOCOMA,','; ',10,213)
LexToken(LLAVEDER,'}',11,223)
LexToken(RETORNAR,'retornar',12,233)
LexToken(ID,'resultado',12,242)
LexToken(PUNTOCOMA,','; ',12,251)
LexToken(LLAVEDER,'}',13,257)

```

8.2 Escenario 2

Para este escenario se toma la cadena de caracteres correspondiente al un codigo del algoritmo de euclides

```

funcion entero4 euclides(entero4 a, entero4 b) {
    mientras(b != 0) {
        entero4 temp = b;
        b = a % b;
        a = temp;
    }
    retornar a;
}

```

Retornando como resultado del analisis

```

LexToken(FUNCION,'funcion',3,30)
LexToken(ENTERO4,'entero4',3,38)
LexToken(ID,'euclides',3,46)
LexToken(PARIZQ,'(',3,54)
LexToken(ENTERO4,'entero4',3,55)
LexToken(ID,'a',3,63)
LexToken(COMA,',',3,64)
LexToken(ENTERO4,'entero4',3,66)
LexToken(ID,'b',3,74)
LexToken(PARDER,')',3,75)
LexToken(LLAVEIZQ,'{',3,77)
LexToken(MIENTRAS,'mientras',4,87)
LexToken(PARIZQ,'(',4,95)
LexToken(ID,'b',4,96)
LexToken(DIFERENTE,'!=',4,98)
LexToken(ENTERO,0,4,101)
LexToken(PARDER,')',4,102)
LexToken(LLAVEIZQ,'{',4,104)
LexToken(ENTERO4,'entero4',5,118)
LexToken(ID,'temp',5,126)
LexToken(ASIGNAR,'=',5,131)
LexToken(ID,'b',5,133)
LexToken(PUNTOCOMA,','; ',5,134)
LexToken(ID,'b',6,148)
LexToken(ASIGNAR,'=',6,150)
LexToken(ID,'a',6,152)

```

```

LexToken(MOD, '%', 6, 154)
LexToken(ID, 'b', 6, 156)
LexToken(PUNTOCOMA, ';', 6, 157)
LexToken(ID, 'a', 7, 171)
LexToken(ASIGNAR, '=', 7, 173)
LexToken(ID, 'temp', 7, 175)
LexToken(PUNTOCOMA, ';', 7, 179)
LexToken(LLAVEDER, '}', 8, 189)
LexToken(RETORNAR, 'retornar', 9, 199)
LexToken(ID, 'a', 9, 208)
LexToken(PUNTOCOMA, ';', 9, 209)
LexToken(LLAVEDER, '}', 10, 215)

```

8.3 Escenario 3

Para este escenario se toma la cadena de caracteres correspondiente al un código que calcula el determinante de una matriz cuadrada 2x2

```

funcion entero4 determinante(entero4 x1, entero4 x2, entero4 y1, entero4 y2){
    entero8 det = (x1 * y2) - (x2 * y1);
    retornar det;
}

```

Retornando como resultado el análisis

```

LexToken(FUNCION, 'funcion', 3, 30)
LexToken(ENTERO4, 'entero4', 3, 38)
LexToken(ID, 'determinante', 3, 46)
LexToken(PARIZQ, '(', 3, 58)
LexToken(ENTERO4, 'entero4', 3, 59)
LexToken(ID, 'x1', 3, 67)
LexToken(COMA, ',', 3, 69)
LexToken(ENTERO4, 'entero4', 3, 71)
LexToken(ID, 'x2', 3, 79)
LexToken(COMA, ',', 3, 81)
LexToken(ENTERO4, 'entero4', 3, 83)
LexToken(ID, 'y1', 3, 91)
LexToken(COMA, ',', 3, 93)
LexToken(ENTERO4, 'entero4', 3, 95)
LexToken(ID, 'y2', 3, 103)
LexToken(PARDER, ')', 3, 105)
LexToken(LLAVEIZQ, '{', 3, 106)
LexToken(ENTERO8, 'entero8', 4, 116)
LexToken(ID, 'det', 4, 124)
LexToken(ASIGNAR, '=', 4, 128)
LexToken(PARIZQ, '(', 4, 130)
LexToken(ID, 'x1', 4, 131)
LexToken(MULT, '*', 4, 134)
LexToken(ID, 'y2', 4, 136)
LexToken(PARDER, ')', 4, 138)
LexToken(MENOS, '-', 4, 140)
LexToken(PARIZQ, '(', 4, 142)
LexToken(ID, 'x2', 4, 143)
LexToken(MULT, '*', 4, 146)
LexToken(ID, 'y1', 4, 148)
LexToken(PARDER, ')', 4, 150)

```

```

LexToken(PUNTOCOMA, ';', 4, 151)
LexToken(RETORNAR, 'retornar', 5, 161)
LexToken(ID, 'det', 5, 170)
LexToken(PUNTOCOMA, ';', 5, 173)
LexToken(LLAVEDER, '}', 6, 179)

```

8.4 Escenario 4

Para este escenario se toma la cadena de caracteres correspondiente al un código que calcula el determinante de una matriz cuadrada 2x2

```

funcion entero4 abs(entero4 con_signo a){
    si(a >= 0){
        retornar a;
    }
    si_no{
        absoluto = -1*a;
        retornar absoluto;
    }
}

```

Retornando como resultado el análisis

```

LexToken(FUNCION, 'funcion', 3, 30)
LexToken(ENTERO4, 'entero4', 3, 38)
LexToken(ID, 'abs', 3, 46)
LexToken(PARIZQ, '(', 3, 49)
LexToken(ENTERO4, 'entero4', 3, 50)
LexToken(CON_SIGNO, 'con_signo', 3, 58)
LexToken(ID, 'a', 3, 68)
LexToken(PARDER, ')', 3, 69)
LexToken(LLAVEIZQ, '{', 3, 70)
LexToken(SI, 'si', 4, 80)
LexToken(PARIZQ, '(', 4, 82)
LexToken(ID, 'a', 4, 83)
LexToken(MAYORIGUAL, '>=', 4, 85)
LexToken(ENTERO, 0, 4, 88)
LexToken(PARDER, ')', 4, 89)
LexToken(LLAVEIZQ, '{', 4, 90)
LexToken(RETORNAR, 'retornar', 5, 104)
LexToken(ID, 'a', 5, 113)
LexToken(PUNTOCOMA, ';', 5, 114)
LexToken(LLAVEDER, '}', 6, 124)
LexToken(SINO, 'si_no', 7, 134)
LexToken(LLAVEIZQ, '{', 7, 139)
LexToken(ID, 'absoluto', 8, 153)
LexToken(ASIGNAR, '=', 8, 162)
LexToken(MENOS, '-', 8, 164)
LexToken(ENTERO, 1, 8, 165)
LexToken(MULT, '*', 8, 166)
LexToken(ID, 'a', 8, 167)
LexToken(PUNTOCOMA, ';', 8, 168)
LexToken(RETORNAR, 'retornar', 9, 182)
LexToken(ID, 'absoluto', 9, 191)
LexToken(PUNTOCOMA, ';', 9, 199)
LexToken(LLAVEDER, '}', 10, 209)

```

LexToken(LLAVEDER, '}', 11, 223)

8.5 Análisis

Tras el análisis de los resultados obtenidos en los cuatro casos de prueba, se pudo comprobar que el analizador léxico identificó correctamente las categorías léxicas asociadas a cada una de las subcadenas en las cadenas de entrada. En todos los casos, los tokens generados coincidieron con los valores esperados de acuerdo con las reglas definidas en la gramática léxica del lenguaje.

Esto demuestra que las expresiones regulares implementadas en las definiciones de tokens son adecuadas para reconocer las estructuras básicas del lenguaje, como identificadores, palabras reservadas, operadores y delimitadores.

9. Conclusiones

El **Simulador Atlas CPU** representa una herramienta educativa completa que cumple exitosamente con los objetivos establecidos:

9.1 Logros Principales

1. **Funcionalidad Completa:** 47 instrucciones implementadas y validadas
2. **Validación Exhaustiva:** Algoritmos clásicos verificados matemáticamente
3. **Documentación Integral:** Manuales técnicos y educativos completos
4. **Interfaz Intuitiva:** GUI diseñada para facilitar el aprendizaje
5. **Arquitectura Sólida:** Diseño modular y extensible

9.2 Impacto Educativo

- **Experimentación:** Ambiente seguro para pruebas y errores
- **Comprensión:** Visualización directa de conceptos abstractos

2025 - Grupo D - Universidad Nacional de Colombia

Simulador Atlas CPU - Hexacore Technologies Repositorio GitHub: https://github.com/JulianGomezN/maquina_lenguajes