Instituto Tecnológico de Estudios Superiores de Monterrey, Campus Monterrey

Materia:

Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 502)

Nombre de tarea: Documentación final - módulo compiladores

Semestre: 8vo

Alumno:

Rafael Lorenzo Hernandez Zambrano A00836774

1) Scanner y Parser

Selección para la herramienta automática del Parser y Scanner

ANTLR
La herramienta automática ANTLR se justifica por su integración simplificada de análisis léxico y sintáctico en un único archivo, su capacidad para manejar ambigüedades gramaticales (como la recursión izquierda mediante reescritura) y un proceso de depuración más accesible que los métodos *bottom-up* como CUP. Su algoritmo LL(*) moderno compensa las desventajas de los métodos descendentes. En cuanto a la documentación, ANTLR es densa, mientras que JFlex + CUP puede sentirse desactualizada.

Definición final de la gramática

1. Expresiones regulares (RE)

Identificador (ID):

```
[a-zA-Z][a-zA-Z0-9_]*
```

Un identificador debe comenzar con una letra y puede ser seguido por letras, números o guiones bajos.

Constante entera (cte_int):

```
[0-9]+
```

Una constante entera consiste en uno o más dígitos consecutivos.

Constante flotante (cte_float):

```
[0-9]+\.[0-9]+
```

Una constante flotante se compone de una parte entera inicial, seguida de un punto decimal y la parte fraccional.

Constante de cadena (cte_string):

```
"[^"]*"
```

Una constante de cadena consiste en cualquier secuencia (incluyendo vacía) encerrada entre comillas dobles. Los caracteres internos no pueden ser comillas dobles.

2. Lista de Tokens

2.1 Palabras Reservadas

```
program
main
end
var
int
float
void
if
else
while
do
print
return
```

2.2 Identificadores y Constantes

```
ID
cte_int
cte_float
cte_string
```

2.3 Operadores

```
=     asignación
+     suma
-     resta
*     multiplicación
/     división
>     mayor que
<     menor que
!=    diferente
==    igual
```

2.4 Símbolos

```
;     punto y coma
,     coma
:     dos puntos
(     paréntesis izquierda
)     paréntesis derecha
{     llave izquierda
}     llave derecha
```

3. Reglas Gramaticales (CFG)

3.1 Estructura General del Programa

```
PROGRAMA → program ID ; SECCION_VARS? SECCION_FUNC* main BLOQUE end
```

3.2 Declaración de Variables

```
SECCION_VARS → var DECL_VAR+

DECL_VAR → LISTA_ID : TIPO ;

LISTA_ID → ID
         | ID , LISTA_ID
```

3.3 Tipos

```
TIPO → int | float | void
```

3.4 Definición de Funciones

```
SECCION_FUNC → DEF_FUNC

DEF_FUNC → TIPO ID ( PARAMS? ) SECCION_VARS? BLOQUE? ;
```

3.4.1 Parámetros formales

```
PARAMS → PARAM
       | PARAM , PARAMS

PARAM → ID : TIPO
```

3.5 Bloques y Sentencias

```
BLOQUE → { LISTA_SENT? }

LISTA_SENT → SENTENCIA LISTA_SENT
            | ε
```

3.5.1 Sentencias válidas

```
SENTENCIA → ASIGNACION
          | IF
          | WHILE
          | PRINT
          | LLAMADA_FUNC_STMT
          | RETURN
```

4. Sentencias

4.1 Asignación

```
ASIGNACION → ID = EXPR ;
```

4.2 Sentencia print

```
PRINT → print ( LISTA_PRINT? ) ;

LISTA_PRINT → EXPR
            | EXPR , LISTA_PRINT
```

4.3 Llamada a función como sentencia

```
LLAMADA_FUNC_STMT → LLAMADA_FUNC ;
```

4.4 Sentencia while

```
WHILE → while ( EXPR ) do BLOQUE ;
```

4.5 Sentencia if–else

```
IF → if ( EXPR ) BLOQUE ELSE_PARTE? ;

ELSE_PARTE → else BLOQUE
           | ε
```

4.6 Sentencia return

```
RETURN → return ( EXPR ) ;
```

5. Expresiones

5.1 Expresión Relacional

```
EXPR → EXPR_ARIT OP_REL EXPR_ARIT
     | EXPR_ARIT

OP_REL → >
       | <
       | !=
       | ==
```

5.2 Expresión Aritmética

```
EXPR_ARIT → EXPR_ARIT OP_SUMA TERM
          | TERM
```

```
OP_SUMA → +
        | -
```

5.3 Términos

```
TERM → TERM OP_MULT FACTOR
     | FACTOR

OP_MULT → *
         | /
```

5.4 Factores

```
FACTOR → ( EXPR )
        | + FACTOR
        | - FACTOR
        | CTE
        | LLAMADA_FUNC
        | ID
```

5.5 Constantes

```
CTE → cte_int
    | cte_float
    | cte_string
```

6. Llamada a Función

6.1 Invocación

```
LLAMADA_FUNC → ID ( ARGUMENTOS? )
```

6.2 Lista de argumentos

```
ARGUMENTOS → EXPR
           | EXPR , ARGUMENTOS
```

Consideraciones de ANTLR sobre la gramática:

ANTLR utiliza un enfoque LL(*) de análisis sintáctico descendente. Esto significa que el parser toma decisiones a partir del lookahead sin retrocesos extensivos. Debido a ello, la gramática debe escribirse de forma compatible con su estrategia de reconocimiento.

En primer lugar, ANTLR **no soporta la recursión izquierda directa**. Las producciones típicas de expresiones de la forma `E -> E + T` provocan ciclos infinitos. Para solventarlo,

es necesario **reformular la gramática jerárquicamente**, definiendo niveles de precedencia: expresiones relacionales, sumas y restas, multiplicaciones, operadores unarios y átomos.

Además, ANTLR trata ciertos literales, como `true` y `false`, como **valores terminales equivalentes a otros tipos literales**, en lugar de modelarlos como identificadores o números especiales. Esto simplifica la interpretación semántica y permite que formen parte natural de la regla `atom`.

Otro aspecto relevante es que ANTLR **prioriza las coincidencias léxicas más largas y la primera regla definida**. Por ello, las palabras reservadas deben declararse antes que los identificadores, y el orden de las reglas léxicas afecta al comportamiento del lexer.

Finalmente, ANTLR permite descartar espacios en blanco y comentarios mediante reglas con `-> skip`. Esto evita que la gramática sintáctica se contamine con producciones accesorias, manteniéndola limpia y cercana a la semántica del lenguaje.

Estas características hacen que la especificación gramatical para ANTLR difiera de la presentada en gramáticas formales clásicas (BNF/EBNF) o generadores LR.

2) Semántica

Posteriormente, del proceso del análisis léxico y sintáctico, el siguiente paso fundamental en el proceso de compilación es el análisis semántico.

Esta fase se encarga de verificar que el código, aunque sintácticamente correcto, tenga sentido lógico y cumpla con las reglas del lenguaje que no pueden ser expresadas únicamente con la gramática libre de contexto (CFG).

El análisis semántico verifica:

- Coherencia de tipos en expresiones y asignaciones.
- Correcta declaración y uso de variables y funciones.
- Correspondencia entre firmas de funciones y sus llamadas.
- Reglas de alcance (scope) y contextos.

2.1 Directorio de Funciones (FuncDir)

Descripción General

Es la estructura de control de mayor nivel dentro del compilador.

Representa un registro de todas las funciones declaradas en el programa, incluyendo dos ámbitos especiales:

- **global** → tratado como una pseudo-función de tipo void.
- **main** → función real de tipo void con su propio ámbito.

Almacena para cada función:

- Tipo de retorno (int, float, bool, void)
- Lista ordenada de parámetros
- Tabla de variables locales

Motivación

El Parser permite funciones con:

- Tipo de retorno
- Parámetros
- Variables locales

Por lo tanto se requiere una estructura centralizada para:

- Verificar duplicidad de funciones.
- Registrar la firma (signature).
- Mantener el scope asociado a cada función.
- Validar llamadas vs definición formal.

Implementación Real

FuncDir se implementa como un contenedor de funciones:

- funcs: Dict[String → FuncInfo]
- current: String → nombre de la función actualmente analizada

Cada FuncInfo incluye:

- return_type

- params: List[(nombre, tipo)]
- var_table: Dict[nombre → VarInfo]

Esto permite búsquedas promedio en **O(1)**.

Aspectos importantes a tener en cuenta

- El ámbito inicial es **global**, ya creado al construir el FuncDir.
- MAIN se registra como una función más.
- Al terminar una función, el compilador regresa a current = "global".

Operaciones Semánticas Principales

Operación	Descripción
add_function(nombre, tipoRetorno)	Registra una nueva función. Si ya existe, lanza error. Cambia el ámbito actual.
lookup_function(nombre)	Consulta una función al momento de las llamadas. Si no existe → error.
add_param(nombre, tipo)	Agrega parámetros formales a la firma y a su VarTable local.

Si se declara:

float area(x: float, y: float)

Entonces:

- area.return_type = float
- area.params = [(x,float), (y,float)]
- area.var_table = {x, y, ...}

2.2 Tabla de Variables (VarTable)

Descripción

Cada función posee su propia tabla de variables.

Esta estructura almacena:

- Los identificadores declarados
- Sus tipos asociados
- Su dirección virtual (fase posterior)

La tabla está dentro del FuncInfo asociado al ámbito actual.

Justificación

Las reglas del lenguaje establecen:

- Las variables se declaran por nombre y tipo.
- No puede haber duplicación dentro del mismo scope.
- Variables pueden consultarse desde global.

La Tabla de Variables permite:

- Validar re-declaraciones inconsistentes.
- Resolver tipos de expresiones.
- Soportar la generación de cuadruplos con direcciones virtuales.

Implementación Real

Se usa un diccionario:

func.var_table: Dict[nombre → VarInfo]

Si no se encuentra en el scope actual, se busca en global.

Importante

No existe “shadowing”:

si una variable existe globalmente y localmente, se toma la local.

Operaciones

Operación	Descripción
add_var(nombre, tipo)	Inserta variable nueva. Si ya existe → error semántico.
lookup(nombre)	Resuelve primero en el ámbito actual, luego en global. Si no existe → error.

2.3 Cubo Semántico (Semantic Cube)

Descripción

Estructura que define las reglas combinatorias entre operadores y tipos.

Permite que el compilador **decida el tipo resultante** o detecte **incompatibilidades**.

El cubo almacena reglas para:

- Operadores aritméticos
- Operadores relacionales
- Igualdad y desigualdad
- Asignaciones

Se representa como un objeto inmutable consultable en O(1).

Motivación

Las operaciones entre tipos no pueden resolverse desde la sintaxis (CFG).

Ejemplo:

int + float → float

pero

int = float → error

Centralizar las reglas evita duplicación en el parser o el listener.

Operaciones Principales

Operación	Uso
check_op(op, t1, t2)	Devuelve el tipo resultante o ERROR
check_assign(target, expr)	Detecta asignaciones válidas/inválidas

Reglas Implementadas

- **Jerarquía numérica:**
    - int + float → float
    - float + int → float
- Multiplicación y suma siguen la regla de promoción.
- División siempre produce **float** si hay al menos un operando float.
- Resto (%) solo válido entre enteros.
- Operaciones relacionales (>, <, ==, !=) producen **bool**.
- Asignación:
    - float = int permitida (promoción)
    - int = float inválida (pérdida de precisión)
    - bool = bool válida

2.4 Errores Semánticos

Todos los errores semánticos se encapsulan mediante:

class SemanticError(Exception)

Esto permite abortar compilación de forma controlada con un mensaje claro.

2.5 Flujo de Trabajo dentro del Listener

Durante el análisis sintáctico (ANTLR o CUP), los **puntos neurálgicos** del parser generan llamadas a estas estructuras cuando ocurren eventos semánticos:

- **Declaración de función → FuncDir.add_function()**
- **Declaración de parámetro → FuncDir.add_param()**
- **Declaración de variable → VarTableHelper.add_var()**
- **Uso de variable → lookup()**
- **Operación binaria → cube.check_op()**
- **Asignación → cube.check_assign()**
- **Fin de función → FuncDir.set_global()**

Esto asegura que las fases posteriores (generación de cuadruplos y memoria virtual) reciban información consistente.

3) Código de Expresiones y estatutos lineales

3.1 Estructuras de datos utilizadas dentro del compilador Patito

A continuación se detallarán las estructuras (filas y pilas) utilizadas dentro de esta sección del proyecto Patito.

Pilas (PatitoSemanticListener.py): operand_stack guarda direcciones de memoria (no valores) y se sincroniza con type_stack; operator_stack casi no se usa; jump_stack almacena índices de cuádruplos para backpatch en IF/WHILE; while_start_stack recuerda el inicio del ciclo para el salto hacia atrás; expr_value cachea resultados de expr para evitar pops erróneos.

Temporales y memoria: TempManager entrega (name, addr) pero el listener solo usa addr; se resetea el contador al entrar a cada bloque de función. VirtualMemory asigna segmentos glob/loc/tmp/const; constantes se deduplican con get_or_add_constant.

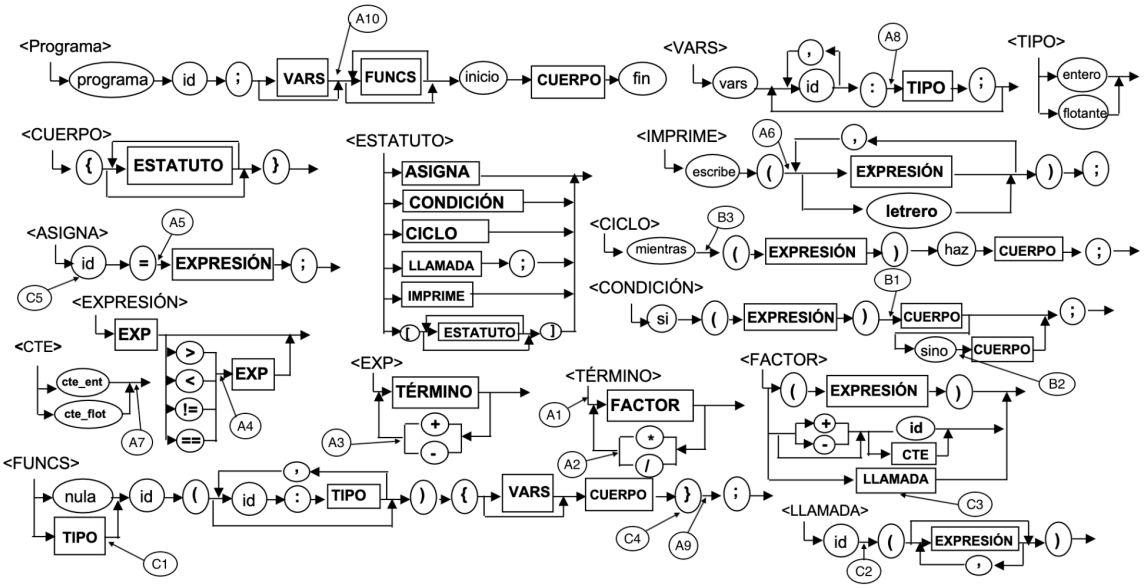
Directorio de funciones (FuncDir.py): cada función guarda ret, vars, params, start, return_addr, has_return; al declarar función no-void se reserva una variable global para el valor de retorno y se reutiliza en llamadas. current_scope() decide glob/loc para asignaciones de memoria.

Tabla de variables (VarTableHelper.py): add_var pide dirección según el scope y registra; lookup resuelve primero local luego global (lanza SemanticError si no existe).

Cubo semántico (semantics.py): admite + - * / % < > == != con coerción implícita float = int; check_assign permite bool = bool y float = int, el resto debe coincidir.

3.2 Puntos neurálgicos y señalamientos sobre el diagrama

Ubicación visual de los puntos neurálgicos dentro del diagrama de sintaxis del lenguaje Patito.



Código	Descripción o función del punto neurálgico
A1	Coloca valores, identificadores y temporales en la pila de operandos junto con su tipo correspondiente, permitiendo que las reglas semánticas posteriores siempre tengan acceso al par (valor–tipo).
A2	Toma dos operandos y su operador (*, /) y produce un temporal como resultado, generando el cuádruplo que representa la operación aritmética.
A3	Funciona igual que A2, pero para operadores de suma y resta (+, –). Garantiza que la operación se evalúe y se almacene en un temporal.

A4	Genera cuádruplos de comparación (<, >, !=), creando un temporal booleano que sirve como base para decisiones de control de flujo.
A5	Válida tipos entre variable y expresión, y genera el cuádruplo de asignación que actualiza el valor de la variable en memoria.
A6	Produce un cuádruplo por cada expresión a imprimir, manejando cadenas, constantes y temporales de manera uniforme.
A7	Registra literales enteros, flotantes o cadenas en la pila de operandos, permitiendo su uso inmediato en operaciones o funciones.
A8	Añade variables al directorio de variables de la función actual (variables locales), asociándose con su tipo y espacio de memoria correspondiente.
A9	Restablece el ámbito a nivel global después de terminar una función, limpiando el contexto semántico local.
A10	Registra variables definidas en la sección global, haciéndolas accesibles a funciones y al bloque principal.
B1	Tras evaluar la condición del if, genera el cuádruplo GOTOF que controla el flujo según el resultado booleano de la expresión.
B2	Genera el salto incondicional GOTO para evitar ejecutar el bloque else tras completar el bloque if, y hace backpatch del GOTOF hacia el inicio del else.
B3	Implementa ciclos while guarda el inicio del ciclo, generando GOTOF para la salida y GOTO para volver al inicio, completando los saltos mediante backpatch.

Nombre del punto	Comportamiento dentro de la semántica	Cuádruplos generados
C1	Cuando el parser identifica la firma de la función y su bloque { CUERPO }, el compilador registra nombre, tipo de retorno, parámetros y variables locales. Al entrar al bloque, se marca la dirección de inicio (start).	Ninguno aún (solo setup de Function Directory).
C2	Al detectar id comienza la activación de una función: se genera el Activation Record. Luego se recorren las expresiones dentro de los parámetros reales para generar PARAM.	ERA fPARAM arg → paramGOSUB f, start
C3	Si la función tiene tipo, se genera un temporal para almacenar el valor retornado y se copia el contenido del return register. Esto permite usar la función como parte de expresiones más grandes.	= retAddr , temp(y el temporal se empuja a la pila de operandos)
C4	Al cerrar el bloque de la función, se completa la definición del Activation Record y se genera el cuádruplo que marca el fin del contexto local.	ENDFUNC
C5	Se detecta un retorno implícito tipo Pascal: asignar al nombre de la función equivale a asignar el valor de retorno. Se valida el tipo y se genera el RET real.	RET valor

4) Código de Estatutos y uso de Direcciones Virtuales

4.1 Implementación de la Memoria Virtual

La implementación se basa en una distribución precisa de direcciones para facilitar la gestión de tipos y ámbitos.

Distribución y Segmentación de Direcciones Virtuales

- **Segmento Base:** Cada segmento tiene un tamaño fijo de 1000 direcciones (`SEGMENT_SIZE = 1000`). La asignación lineal utiliza `MemorySegment.alloc()`, que lanza un `MemoryError` si se agota el espacio.
- **Segmentos Globales (Ámbito del Programa):**
  - `glob_int`: 1000 – 1999
  - `glob_float`: 2000 – 2999
  - `glob_bool`: 3000 – 3999
- **Segmentos Locales (Por Activación/Función):**
  - `loc_int`: 4000 – 4999
  - `loc_float`: 5000 – 5999
  - `loc_bool`: 6000 – 6999
- **Segmentos de Constantes (Tipo de Dato):**
  - `const_int`: 7000 – 7999
  - `const_float`: 8000 – 8999
  - `const_bool`: 9000 – 9999
  - `const_string`: 10000 – 10999
- **Segmentos Temporales (Por Activación/Función):**
  - `temp_int`: 11000 – 11999
  - `temp_float`: 12000 – 12999
  - `temp_bool`: 13000 – 13999

Mecanismos de Solicitud de Direcciones

Tipo de Elemento	Proceso de Asignación
Variables (Vars)	`VarTableHelper.add_var` invoca `VirtualMemory.alloc_var(scope, vtype)` usando el ámbito actual (`FuncDir.current_scope()`, sea global o local). La dirección y metadata se registran en `FuncDir.add_variable (name, type, address)`.

Constantes	`PatitoSemanticListener.get_or_add_constant` verifica si la constante existe (`constant_lookup`) y, si es nueva, llama a `alloc_const(vtype)`. Al salir de un átomo (`exitAtom`), la dirección se coloca en la `operand_stack`.
Temporales	`TempManager.new_temp(ty)` solicita la dirección a través de `alloc_temp(ty)` y retorna un par `(tN, addr)`. El *listener* utiliza esta dirección para generar los cuádruplos.

Uso de Direcciones en Cuádruplos

El formato de los cuádruplos es siempre `(opcode, dir_izq, dir_der, dir_res)`, donde todas las entradas son direcciones virtuales.

- **Operaciones Aritméticas/Relacionales:** Las direcciones resultantes (`dir_res`) son asignadas a variables temporales (`temp_*`) por las funciones como `exitAddOp`, `exitMultOp` y `exitRelExpr`.
- **Asignación:** En `exitAssignStmt`, la dirección resultante (`dir_res`) corresponde a la dirección de la variable de destino (global o local), mientras que la dirección de la expresión (`expr_addr`) es típicamente una temporal o una constante.
- **Saltos:** Las instrucciones de salto condicional (`GOTOF`) y incondicional (`GOTO`) utilizan `dir_izq` para la condición (generalmente una dirección de temporal booleana, `temp_bool`).

Consideraciones Clave del Diseño

1. **Inferencia de Contexto:** Los segmentos son disjuntos, lo que permite que la Máquina Virtual (VM) infiera el tipo de dato y el ámbito de la variable simplemente a partir del rango de su dirección virtual.
2. **Manejo de Booleanos:** Las constantes incluyen su propio segmento booleano (9000–9999), independiente de los segmentos de booleanos para variables locales y globales.
3. **Gestión de Temporales en Ejecución:** Aunque los contadores de temporales se reajustan al inicio de un bloque de función (`enterBlock`) durante la compilación, la memoria virtual es estática por ejecución del compilador. Para simular múltiples llamadas en *runtime*, la VM debe gestionar y/o restablecer los temporales basándose en las activaciones de la función.

5) Código de Funciones y Ejecución de Expresiones

Protocolo de Llamadas a Función y Control de Flujo

Esta sección detalla cómo el compilador traduce las llamadas a funciones y cómo la Máquina Virtual (VM) orquesta la memoria y el puntero de instrucción (IP) durante la ejecución.

5.1 Instrucciones de Cuádruplos (OpCodes)

La gestión de funciones se realiza mediante cinco operaciones fundamentales que coordinan la creación, llenado, activación y destrucción de los registros de activación (*Activation Records*).

5.1.1 ERA (Expansion of Activation Record)

- **Sintaxis:** `ERA, func_id, -, -`
- **Compilación:** Se genera al detectar una llamada a función.
- **Runtime (VM):**
  - Invoca `prepare_activation(func_id)`.
  - Instancia un nuevo objeto `ActivationRecord` pero **no lo coloca en la pila de llamadas** todavía.
  - Este objeto se almacena temporalmente en `pending_activation`, reservando sus ventanas de memoria local (`loc_*`) y temporal (`temp_*`) basadas en el `SEGMENT_LAYOUT`.

5.1.2 PARAM (Parameter Passing)

- **Sintaxis:** `PARAM, source_addr, -, dest_param_addr`
- **Compilación:**
  - Valida cantidad y tipos contra la firma de la función (Semantic Cube).
  - Asigna la dirección real precalculada del parámetro formal.
- **Runtime (VM):**
  - Lee el valor del argumento desde el contexto actual (`load(source_addr)`).
  - Escribe el valor en el contexto pendiente usando `store_pending(dest_param_addr, value)`.
  - **Importante:** Esto permite inicializar las variables locales de la función antes de que esta comience a ejecutarse.

5.1.3 GOSUB (Transfer Control)

- **Sintaxis:** `GOSUB, func_id, -, start_addr`
- **Compilación:** `start_addr` apunta al primer cuádruplo del bloque de la función.
- **Runtime (VM):**
    - Guarda el puntero de instrucción actual para el retorno (`return_ip = ip + 1`).
    - Ejecuta `push_prepared_activation()`: mueve el registro de `pending` al tope del `call_stack`.
    - Actualiza el IP global a `start_addr`.

5.1.4 RET (Return Value)

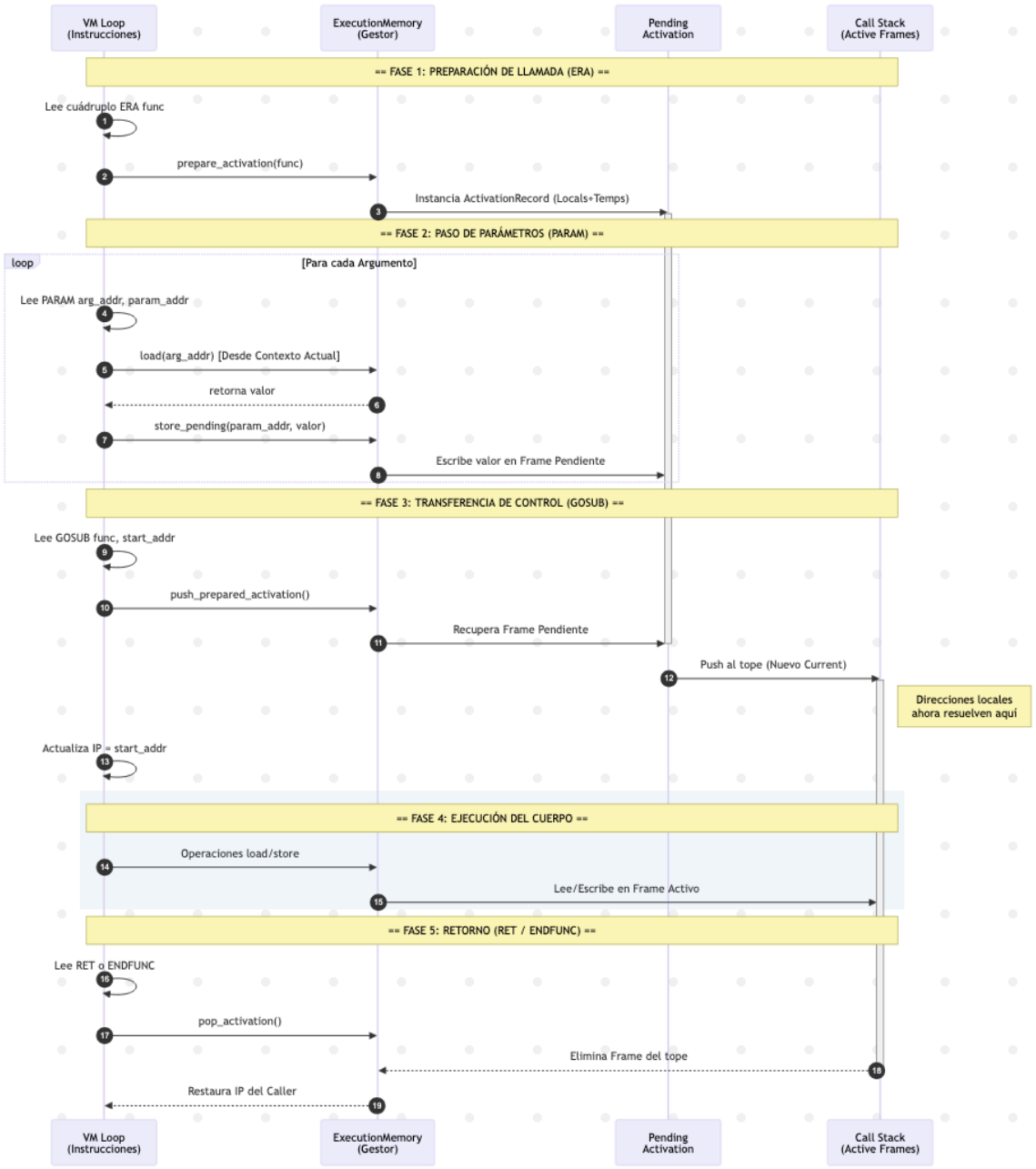
- **Sintaxis:** `RET, -, -, -` (y asignaciones previas de retorno).
- **Mecanismo de Retorno:**
    1. Dentro de la función, se asigna el resultado a una dirección global reservada: `valor -> return_addr_global`.
    2. Se emite `RET`.
    3. En el *caller*, inmediatamente después del `GOSUB`, se genera un cuádruplo para recuperar el valor: `return_addr_global -> temporal`.
- **Runtime (VM):** Ejecuta `pop_activation()`, liberando la memoria del frame actual y restaurando el IP guardado.

5.1.5 ENDFUNC (End of Function)

- **Sintaxis:** `ENDFUNC, -, -, -`
- **Compilación:** Se emite siempre al final de la declaración de una función como red de seguridad.
- **Runtime (VM):** Actúa igual que `RET` (libera memoria y retorna) para asegurar que el stack no se corrompa si el usuario olvidó un `return` en funciones `void`.

5.2 Diagrama de Secuencia: Ciclo de Vida de una Llamada

El siguiente diagrama ilustra la interacción temporal entre el procesamiento de cuádruplos y los estados de la memoria en la VM.



5.3 Estrategia de Direccionamiento y Memoria

5.3.1 Asignación Estática (Compilación)

El compilador utiliza asignadores lineales (*bump allocators*) para otorgar direcciones virtuales únicas durante el análisis semántico.

- **Variables:** `VarTableHelper` delega a `VirtualMemory.alloc_var`.
- **Temporales:** `TempManager` solicita `alloc_temp`. Los contadores se reinician por función para reutilizar offsets relativos, aunque la dirección virtual final es absoluta dentro del rango del tipo.
- **Constantes:** Se gestionan globalmente para evitar duplicados (`get_or_add_constant`).

5.3.2 Resolución Dinámica (Ejecución)

La `ExecutionMemory` utiliza una estrategia de **búsqueda jerárquica** para resolver operaciones `load/store`:

1. **Scope Local/Temporal:** Se busca primero en las ventanas del `current_activation` (tope del stack).
2. **Scope Global:** Si la dirección no está en el rango local, se busca en las ventanas globales.
3. **Scope Constante:** Finalmente, se consulta la memoria de constantes (solo lectura para operaciones estándar).

5.3.3 Inicialización Perezosa (Lazy Initialization)

Para optimizar el uso de recursos, las `MemoryWindow` no pre-rellenan sus mapas de datos.

- Al hacer `load(addr)`, si la dirección es válida pero no existe en el diccionario, se retorna un valor por defecto (`0`, `0.0`, `False`, `""`).
- Esto evita errores de "variable no inicializada" en runtime y reduce el overhead de crear nuevos marcos de pila.

5.4 Consideraciones de Diseño del Sistema

- **Inferencia de Tipos O(1):** Gracias a los rangos de direcciones disjuntos (ej. `1000-1999` vs `2000-2999`), la VM no necesita consultar tablas de metadatos en runtime; la dirección misma indica el segmento y tipo.
- **Seguridad de Memoria:** Al encapsular las variables locales y temporales dentro del objeto `ActivationRecord`, se garantiza su destrucción automática al salir de la función (`pop_activation`), previniendo fugas de memoria lógicas.
- **Eficiencia:** El uso de diccionarios dispersos permite declarar arreglos grandes o tener muchas variables potenciales sin consumir memoria RAM real hasta que se escriben valores en ellas.

6) Implementación completa de la máquina virtual

Arquitectura de Memoria en Tiempo de Ejecución (VM)

6.1. Visión General

El sistema de memoria de la Máquina Virtual opera bajo un esquema de **Direccionamiento Virtual Segmentado**. No se utilizan tablas de símbolos en tiempo de ejecución; en su lugar, la semántica de cada dirección (tipo de dato y alcance) está codificada implícitamente en su rango numérico.

- **Inferencia O(1):** La VM determina el segmento y tipo de dato simplemente verificando el rango de la dirección (ejemplo 11000-11999 corresponde a temp_int).
- **Almacenamiento Disperso:** Se utiliza un enfoque de memoria no contigua para optimizar el uso de recursos, almacenando solo las direcciones que realmente se escriben.

6.2 Estructuras de Datos (Runtime)

La arquitectura se compone de tres clases principales que jerarquizan el acceso a los datos.

6.2.1 MemoryWindow (Nivel Base)

Representa un segmento lógico de memoria (ejemplo *Enteros Locales* o *Flotantes Temporales*).

- **Estructura Interna:** Utiliza un diccionario disperso (_data) para mapear dirección virtual -> valor real.
- **Métodos Clave:**
    - contains(addr): Válida si la dirección pertenece al rango de esta ventana (O(1)).
    - load(addr): Retorna el valor. *Nota:* Implementa **inicialización perezosa** (retorna valores por defecto si no existe, excepto en constantes donde falla si no está precargada).
    - store(addr, value): Escribe el valor si el rango es válido.
    - snapshot(): Utilería para depuración.

6.2.2 ActivationRecord (Nivel Función/Frame)

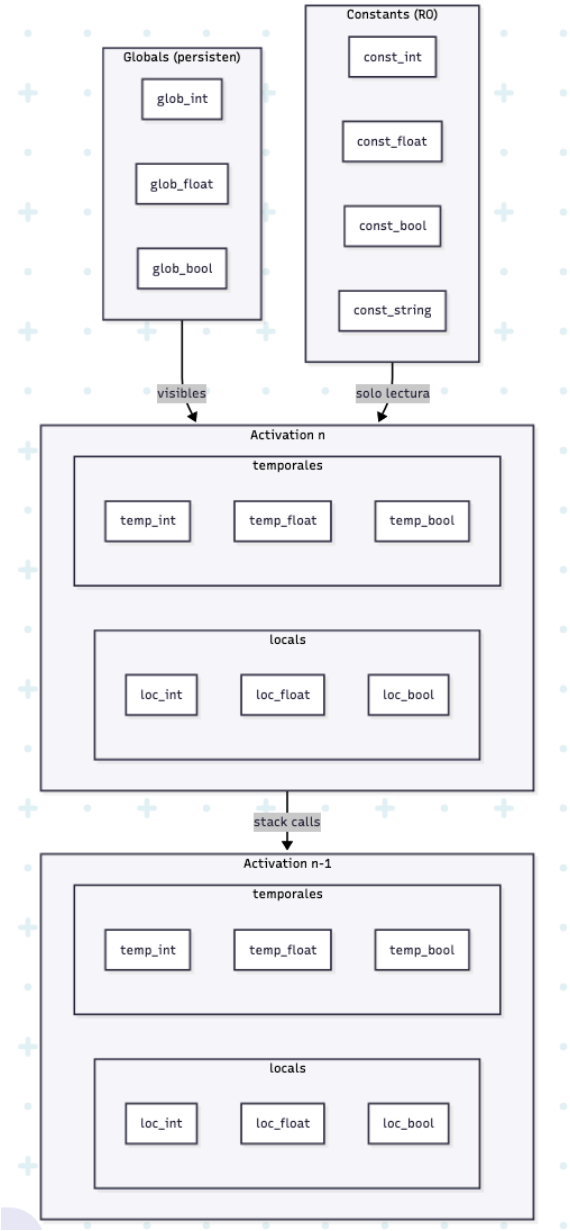
Representa el contexto de ejecución de una función llamada (Stack Frame).

- **Responsabilidad:** Administra el ciclo de vida de las variables locales y temporales.
- **Composición:** Instancia múltiples MemoryWindow (loc_int, temp_float, etc.) basándose en el SEGMENT_LAYOUT definido.
- **Ciclo de Vida:** Se crea al llamar una función y se destruye al retornar, liberando automáticamente toda la memoria local y temporal asociada.

6.2.3 ExecutionMemory (Orquestador)

Es el gestor global de la memoria de la VM. Controla la pila de llamadas y los segmentos persistentes.

- **Estructura:**
  - **Ventanas Persistentes:** Globales (glob_*) y Constantes (const_*).
  - **Call Stack:** Pila de objetos ActivationRecord.
  - **Punteros de Estado:**
    - current_activation: El frame en ejecución (tope del stack).
    - pending_activation: Un frame temporal creado durante la preparación de llamada (ERA) para recibir parámetros antes del cambio de contexto.



6.3 Resolución de Direcciones (Load/Store)

El método _window_for_address(addr) implementa la lógica de búsqueda jerárquica para encontrar qué MemoryWindow debe atender una petición:

1. **Nivel Local/Temporal:** Se busca primero en current_activation. Si el frame activo contiene el rango de la dirección, se opera ahí.
2. **Nivel Global:** Si no es local, se busca en las ventanas persistentes glob_*.
3. **Nivel Constante:** Finalmente, se busca en const_* (generalmente solo lectura).

Excepción: Durante la carga de parámetros (PARAM), la escritura se desvía a pending_activation en lugar de current_activation.

6.4 Ciclo de Vida de una Llamada (Call Flow)

El flujo de operaciones de los cuádruplos mapea directamente a métodos del ExecutionMemory:

Cuádruplo / Acción	**Método VM**	**Descripción**
ERA func	prepare_activation(tag)	Instancia un nuevo ActivationRecord pero lo guarda en pending. Reserva memoria para lo que la función necesitará.
PARAM val, addr	store_pending(addr, val)	Escribe el valor del argumento en la dirección local del pending_activation.
GOSUB func, lbl	push_prepared_activation()	Mueve el frame de pending al tope del call_stack. El contexto cambia.
(Cuerpo)	load/store	Se ejecutan operaciones sobre las ventanas loc_* y temp_* del frame activo.
ENDFUNC / RET	pop_activation()	Se elimina el frame del tope de la pila. Las memorias locales y temporales se destruyen. El IP regresa al *caller*.

6.5 Justificación del Diseño

¿Por qué utilizar este esquema de Memory Windows y Direccionamiento Virtual?

1. **Eficiencia de Espacio (Sparsity):** Al usar diccionarios internos en lugar de arrays fijos, un programa que declara un array de 1000 enteros pero solo usa el índice 0 y 999 solo ocupará 2 espacios de memoria real, no 1000.
2. **Validación Rápida:** Los rangos disjuntos permiten saber si una dirección es válida o está fuera de límites sin iterar estructuras complejas.
3. **Gestión Automática de Memoria:** Al encapsular locales y temporales en el objeto ActivationRecord, el *Garbage Collector* del lenguaje anfitrión (Java/Python) limpia automáticamente la memoria al hacer pop(), evitando fugas de memoria (memory leaks) manuales.

7) Casos de prueba finales

Los siguientes tres casos de prueba fueron utilizados para verificar que todas las funcionalidades estuvieran adecuadamente implementadas dentro del programa.

Caso 1:

```
program Fibonacci;

var
   n : int;
   resultado : int;

int fib(num : int)
var
   a, b : int;
{
   if (num < 2) {
      return(num);
   } else {
      return(fib(num - 1)+ fib(num - 2));
   };
};

void imprimirValor(valor : int)
{
   print(valor);
};

main
{
   n = 20;
   resultado = fib(n);
   imprimirValor(resultado);
}
end
```

Resultado:

```
Análisis sintáctico y semántico completado.

=== CUADRUPLOS GENERADOS ===
```

```
000  (51, None, None, 22)
001  (19, 4000, 7000, 13000)
002  (50, 13000, None, 6)
003  (13, 4000, None, 1002)
004  (73, 4000, None, None)
005  (51, None, None, 19)
006  (15, 4000, 7001, 11000)
007  (70, None, None, fib)
008  (71, 11000, None, 4000)
009  (72, fib, None, 1)
010  (13, 1002, None, 11001)
011  (15, 4000, 7000, 11002)
012  (70, None, None, fib)
013  (71, 11002, None, 4000)
014  (72, fib, None, 1)
015  (13, 1002, None, 11003)
016  (14, 11001, 11003, 11004)
017  (13, 11004, None, 1002)
018  (73, 11004, None, None)
019  (74, None, None, None)
020  (60, 4003, None, None)
021  (74, None, None, None)
022  (13, 7002, None, 1000)
023  (70, None, None, fib)
024  (71, 1000, None, 4000)
025  (72, fib, None, 1)
026  (13, 1002, None, 11005)
027  (13, 11005, None, 1001)
028  (70, None, None, imprimirValor)
029  (71, 1001, None, 4003)
030  (72, imprimirValor, None, 20)

=== EJECUCIÓN EN MÁQUINA VIRTUAL ===
6765
```

Caso 2:

```
program Factorial;

var
    n : int;
    res : int;

int factorial(x : int)
var
    temp : int;
{
    if (x == 0) {
        return(1);
    } else {
        temp = factorial(x - 1);
        return(x * temp);
    };
};

void validarYCalcular(valor : int)
var
    r : int;
{
    if (valor < 0) {
        print("Error: factorial no definido");
    } else {
        r = factorial(valor);
        print("Resultado factorial =");
        print(r);
    };
```

```
};

main
{
    n = 7;
    validarYCalcular(n);
}
end
```

Resultado:

```
Análisis sintáctico y semántico completado.

=== CUADRUPLOS GENERADOS ===
000  (51, None, None, 28)
001  (21, 4000, 7000, 13000)
002  (50, 13000, None, 6)
003  (13, 7001, None, 1002)
004  (73, 7001, None, None)
005  (51, None, None, 15)
006  (15, 4000, 7001, 11000)
007  (70, None, None, factorial)
008  (71, 11000, None, 4000)
009  (72, factorial, None, 1)
010  (13, 1002, None, 11001)
011  (13, 11001, None, 4001)
012  (16, 4000, 4001, 11002)
013  (13, 11002, None, 1002)
014  (73, 11002, None, None)
015  (74, None, None, None)
016  (19, 4002, 7000, 13001)
017  (50, 13001, None, 20)
018  (60, 10000, None, None)
019  (51, None, None, 27)
020  (70, None, None, factorial)
021  (71, 4002, None, 4000)
022  (72, factorial, None, 1)
023  (13, 1002, None, 11003)
024  (13, 11003, None, 4003)
025  (60, 10001, None, None)
026  (60, 4003, None, None)
027  (74, None, None, None)
028  (13, 7002, None, 1000)
029  (70, None, None, validarYCalcular)
030  (71, 1000, None, 4002)
031  (72, validarYCalcular, None, 16)

=== EJECUCIÓN EN MÁQUINA VIRTUAL ===
"Resultado factorial ="
5040
```

Caso 3:

```
program Evaluacion;

var
    numero : int;
    i : int;

bool esPar(x : int)
var
    r : int;
{
    r = x % 2;
    if (r == 0) {
        return(true);
```

```
    } else {
        return(false);
    };
};

int sumaHasta(n : int)
var
    total : int;
{
    total = 0;

    while (n > 0) do {
        total = total + n;
        n = n - 1;
    };

    return(total);
};

main
{
    numero = 5;

    // Estructura condicional
    if (esPar(numero)) {
        print(numero, "es par");
    } else {
        print(numero, "es impar");
    };

    // Ciclo para imprimir cuenta regresiva
    i = numero;
    while (i > 0) do {
        print(i);
        i = i - 1;
    };

    // Sumar desde 1 hasta numero
    print("Suma desde 1 hasta", numero, "=", sumaHasta(numero));
}
end
```

Resultado:

Análisis sintáctico y semántico completado.

```
=== CUADRUPLOS GENERADOS ===
000  (51, None, None, 22)
001  (22, 4000, 7000, 11000)
002  (13, 11000, None, 4001)
003  (21, 4001, 7001, 13000)
004  (50, 13000, None, 8)
005  (13, 9000, None, 3000)
006  (73, 9000, None, None)
007  (51, None, None, 10)
008  (13, 9001, None, 3000)
009  (73, 9001, None, None)
010  (74, None, None, None)
011  (13, 7001, None, 4003)
012  (18, 4002, 7001, 13001)
013  (50, 13001, None, 19)
014  (14, 4003, 4002, 11001)
015  (13, 11001, None, 4003)
016  (15, 4002, 7002, 11002)
017  (13, 11002, None, 4002)
018  (51, None, None, 12)
```

```
019  (13, 4003, None, 1002)
020  (73, 4003, None, None)
021  (74, None, None, None)
022  (13, 7003, None, 1000)
023  (70, None, None, esPar)
024  (71, 1000, None, 4000)
025  (72, esPar, None, 1)
026  (13, 3000, None, 13002)
027  (50, 13002, None, 31)
028  (60, 1000, None, None)
029  (60, 10000, None, None)
030  (51, None, None, 33)
031  (60, 1000, None, None)
032  (60, 10001, None, None)
033  (13, 1000, None, 1001)
034  (18, 1001, 7001, 13003)
035  (50, 13003, None, 40)
036  (60, 1001, None, None)
037  (15, 1001, 7002, 11003)
038  (13, 11003, None, 1001)
039  (51, None, None, 34)
040  (70, None, None, sumaHasta)
041  (71, 1000, None, 4002)
042  (72, sumaHasta, None, 11)
043  (13, 1002, None, 11004)
044  (60, 10002, None, None)
045  (60, 1000, None, None)
046  (60, 10003, None, None)
047  (60, 11004, None, None)

=== EJECUCIÓN EN MÁQUINA VIRTUAL ===
5
"es impar"
5
4
3
2
1
"Suma desde 1 hasta"
5
"="
15
```

Declaración del uso de inteligencia artificial generativa dentro de proyecto de compiladores

En términos generales, las herramientas de inteligencia artificial generativa fueron utilizadas dentro de este trabajo para:

1. Mejorar la claridad, la organización y la corrección gramatical dentro de la documentación
2. Apoyo en la identificación y solución de problemas técnicos (debugging) al implementar las estructuras
3. Preparación de los casos de prueba (test cases) para las respectivas etapas incluyendo los casos de prueba finales (implementación completa con la máquina virtual).

A continuación se detallan las herramientas, prompts y resultados obtenidos al utilizar estas herramientas:

- La documentación fue escrita dentro Google Docs, al contar con Gemini Pro, directamente se integra esta aplicación dentro de este programa. Al seleccionar un texto, este da la opción de poder {formalizarlo, refactorizar, acortarlo o desarrollarlo}. Se usó comúnmente la función refactorizar para mejorar la estética visual de los apartados, y formalizar {con un lenguaje más académico} cuando el texto se veía muy ambiguo o difícil de comprender.
- Siendo el programa Compilador Patito, compuesto a varios archivos .py (además de Patito.g4 con el Scanner y Parser), la herramienta de Codex de OpenAI con el modelo GPT-5.1-Codex-Max (similar a Cursor), al tener archivos interdependientes entre sí, puede identificar y proponer soluciones en base a los códigos de error. Durante varias ocasiones los mensajes de error eran sumamente ambiguos al mencionar partes del código correspondiente a la herramienta automática de ANTLR, dificultando aún más su resolución de una manera eficiente. Prompt: {¿Cuál es la manera más eficiente de poder corregir este error}, respuesta: {el archivo … presenta este error, modificar estas líneas …}
- Finalmente, el modelo GPT-5.1 dado la gramática implementada dentro ANTLR (el archivo Patito.g4), Prompt: {Se proporciona el archivo .g4, dado el archivo con la gramática implementada dentro de la herramienta ANTLR dame un caso de prueba con Fibonacci y Factorial respectivamente}, respuesta: {proporciona los correspondientes archivos .txt con los casos de prueba}.

Repositorio de github del Compilador Patito:
https://github.com/rafhdz/proyecto-compiladores