

A large, light gray Fibonacci spiral is centered on the page, starting from a small square in the upper right and expanding outwards. It passes through the authors' names and the title.

*Documentación Técnica*

# WRITINGMACHINE

Jorge Luis Guillén Campos  
2022178359

Henry Daniel Nuñez Perez  
2022089224

Kun Kin Zheng Liang  
2022205015

# ÍNDICE

<b>1. Diseño del Compilador</b>	<b>3</b>
1.1. Arquitectura . . . . .	3
1.1.1. Front-End . . . . .	3
1.1.2. Back-End . . . . .	4
<b>2. Decisiones Técnicas</b>	<b>6</b>
2.1. Front-End . . . . .	6
2.1.1. Asignación de Registros . . . . .	6
2.1.2. Optimización del Código Intermedio . . . . .	7
2.1.3. Generación de Ensamblador y Objeto . . . . .	7
2.1.4. Manejo de Errores . . . . .	7
<b>3. Optimizaciones</b>	<b>8</b>
<b>4. Pruebas</b>	<b>9</b>
<b>5. Problemas conocidos</b>	<b>11</b>
<b>6. Problemas encontrados</b>	<b>12</b>
6.1. Errores en el reconocimiento de expresiones debido a la ambigüedad gramatical . . . . .	12
6.2. Problemas de recursividad en la creación del árbol y análisis semántico al manejar expresiones anidadas . . . . .	13
6.3. Problemas para definir operaciones numéricas fuera de parámetros . . . . .	14
6.4. Conexión entre el IDE y el código del analizador que solo recibía archivos .txt . . . . .	15
<b>7. Conclusiones</b>	<b>16</b>
<b>8. Recomendaciones</b>	<b>17</b>
<b>Referencias</b>	<b>18</b>

## DISEÑO DEL COMPILADOR

### I.I ARQUITECTURA

El compilador está conformado por un diseño modular que emplea dos fases generales: front-end y back-end. La arquitectura planteada permite una mayor escalabilidad y mantenibilidad del compilador del lenguaje creado con Python, de esta forma se facilita la implementación de cada uno de los componentes indispensables para cada fase y se simplifica la implementación de optimizaciones para cada nivel presente en el proceso de compilación del lenguaje desarrollado.

#### I.I.I. Front-End

- **Análisis Léxico:**

Implementado por medio de PLY (Python Lex-Yacc) que reconoce tokens definidos mediante expresiones regulares para el lenguaje en cuestión. El reconocimiento incluye el seguimiento de línea y columna para mensajes de error.

```
1  # Definicion de tokens.
2  tokens = (
3      'DEF', 'ID', 'NUMBER', 'BOOLEAN', 'LBRACKET',
4      'RBRACKET', 'LPAREN', 'RPAREN', 'SEMI', 'COMMA',
5      #...otros tokens
6  )
7
8  # Expresiones regulares para tokens.
9  t_DEF = r'Def'
10 t_LPAREN = r'\('
11 t_RPAREN = r'\)'
12 #...otras expresiones regulares.
```

Código Fuente 1: Ejemplo de definición de tokens y expresiones regulares

Asimismo, el *lexer* incluye el manejo de comentarios de línea usando `'//'` y la verificación obligatoria de un comentario inicial. Los errores son gestionados a través de un *tracking* de número de línea y columna con mensajes de error específicos. Los tokens incluyen las palabras clave como *proc*, *call*, etc., así como los identificadores, números y operadores.

- **Análisis Sintáctico:**

Emplea un parser ascendente LR(1) por medio de PLY, se basa en una gramática formal bien definida por medio de reglas de producción. Incluye un sistema de precedencia de operadores y la construcción de AST mediante nodos especializados, el AST se construye a partir de los tokens. Se implementó una

gramática que involucra estructuras de control, definición de procedimientos, expresiones y operaciones.

```

1      # Regla para el control While
2      def p_while_statement(p):
3          '''while_statement : WHILE LBRACKET statement RBRACKET
4              LBRACKET program RBRACKET WHENEND'''
          p[0] = WhileStatement(condition=p[3], body=p[6]) # La
              condicion se toma de p[3] y el cuerpo de p[6]

```

Código Fuente 2: Ejemplo de estructura de control

```

1      # Regla para los procedimientos
2      def p_procedure_statement(p):
3          '''procedure_statement : PROC ID LPAREN parameter_list
4              RPAREN LBRACKET program RBRACKET SEMI END'''
          p[0] = ProcedureStatement(procedure_name=p[2], arguments=p
              [4], body=p[7])

```

Código Fuente 3: Ejemplo de estructura de control

```

1      # Regla para expresiones y operaciones
2      def p_expression_binop(p):
3          '''expression : expression PLUS expression
4              | expression MINUS expression
5              | expression MULT_OP expression
6              | expression DIV_OP expression'''
7          p[0] = BinaryOperation(p[1], p[2], p[3])

```

Código Fuente 4: Ejemplo de estructura de control

Por otro lado, el AST se construye empleando clases específicas para cada tipo de nodo base o específico, por ejemplo *expression*, *statement*, *program*, *callstatement*, etc.

- **Análisis Semántico:**

Este es implementado a través del patrón de diseño Visitor. El análisis involucra la verificación de tipos, manejo de ámbito de variables y la validación de las reglas semánticas establecidas para el lenguaje.

Para el desarrollo del análisis semántico se requirió una tabla de símbolos para la verificación de las variables con sus tipos y ámbitos, así como la validación de declaraciones. Para la verificación de tipos se revisan los tipos en operaciones, argumentos en llamadas y condiciones booleanas.

## 1.1.2. Back-End

- **Generador de Código Intermedio:** Durante esta etapa, se recorre el AST utilizando un patrón Visitor, donde cada nodo genera instrucciones específicas:
  - **Operaciones aritméticas:** Los nodos de suma, resta, multiplicación y división generan instrucciones como 'add', 'sub', 'mul' y 'sdiv', respectivamente.

- **\*\*Estructuras de control:\*\*** Los nodos de bucles y condicionales generan bloques básicos con instrucciones de salto condicional ('br') o incondicional.
- **\*\*Llamadas a funciones:\*\*** Los nodos de llamadas generan instrucciones 'call' que invocan procedimientos definidos en el IR.

Además, se asignan valores a registros virtuales para mantener la consistencia del IR y garantizar que las operaciones respeten las dependencias de datos. Cada instrucción es insertada en el contexto de un bloque básico dentro de una función, que representa el programa compilado.

```
1 def visit_BinaryOperation(self, node):
2     left = self.visit(node.left)
3     right = self.visit(node.right)
4
5     if node.operator == '+':
6         return self.builder.add(left, right, name="addtmp")
7     elif node.operator == '-':
8         return self.builder.sub(left, right, name="subtmp")
```

Código Fuente 5: Ejemplo de generación de código intermedio para operaciones

Finalmente, el módulo LLVM generado se verifica mediante 'binding.parse\_assembly()' para asegurar que cumple con las reglas del formato IR. Este código intermedio sirve como entrada para la fase de optimización.

- **Optimizador:**

La optimización se realiza utilizando el 'ModulePassManager' de 'llvmlite'. El proceso incluye la aplicación de varias pasadas de optimización, entre estas:

- **Fusión de constantes:** Combina valores constantes para reducir el número de instrucciones.
- **Eliminación de código muerto:** Remueve instrucciones y bloques que no afectan el resultado final del programa.
- **Combinación de instrucciones:** Simplifica y reorganiza instrucciones para optimizar el rendimiento.
- **Simplificación del grafo de control:** Reduce la complejidad del flujo de control eliminando saltos innecesarios.

Después de aplicar esas optimizaciones, se verifica nuevamente la validez del IR optimizado para garantizar que los cambios realizados no afecten la corrección del programa.

- **Generador de Código Ensamblador y Objeto:** El IR optimizado se convierte en código ensamblador mediante las capacidades del 'TargetMachine' de 'llvmlite'. Este proceso requiere de:

1. Configuración de un *target triple* compatible con la arquitectura actual ('x86\_64-pc-windows-msvc').

2. Verificación del módulo optimizado para asegurar consistencia antes de la generación de ensamblador.
3. Emisión del código ensamblador utilizando el ‘emit\_assembly()’ de ‘TargetMachine’.

Posteriormente, el archivo ensamblador generado se guarda en disco. Finalmente, se ensambla en un archivo objeto utilizando ‘Clang’. Este archivo puede vincularse a otros módulos o ejecutarse directamente según los requerimientos del sistema.

```

1      # Configuración de la máquina objetivo
2      target_machine = target.create_target_machine(opt=2)
3      assembly_code = target_machine.emit_assembly(llvm_mod)
4      # Guardado del código ensamblador
5      self.save_assembly_to_file(assembly_code, "output.s")
6      # Ensamblado con Clang
7      object_file = self.assemble_to_object_with_clang("output.s", "
      output.o")

```

Código Fuente 6: Fragmento de generación de código ensamblador

Este enfoque asegura que el compilador produzca código eficiente y compatible con la arquitectura destino.

2

## DECISIONES TÉCNICAS

### 2.1 FRONT-END

La confección de esta sección del compilador está compuesta por diferentes etapas de revisión, las cuales el lenguaje fuente debe atravesar y superar exitosamente con la finalidad de ser procesado por el backend.

El análisis léxico se realizó con PLY debido a su facilidad de integración con Python, así como la robustez para el procesamiento de tokens. Para establecer una estructura correcta del programa se posee una implementación especial para la verificación de un comentario inicial, que, de no estar produciría un error. Los errores se identifican por medio del número de línea y columna a través de la terminal propia de la interfaz del lenguaje.

En el desarrollo del backend del compilador se tomaron diversas decisiones técnicas para asegurar un buen balance entre rendimiento y la modularidad. Las decisiones más importantes son:

#### 2.1.1. Asignación de Registros

La asignación de registros se gestiona de manera interna por LLVM. LLVM utiliza un enfoque dividido en dos etapas: la asignación de registros virtuales y la asignación de registros físicos.

*Registros Virtuales:* LLVM emplea registros virtuales ilimitados, lo que permite simplificar las operaciones y no preocuparse inicialmente por las limitaciones físicas de la arquitectura de destino. Durante la fase de

generación de código máquina, el módulo `TargetMachine` transforma los registros virtuales en registros físicos específicos de la arquitectura objetivo (`x86_64-pc-windows-msvc`). Este proceso utiliza un algoritmo de asignación similar `graph coloring`, en donde se construye un grafo donde los nodos representan los registros virtuales y las aristas indican interferencias.

*Registros Físicos:* Se emplea un algoritmo de coloreado de grafos para asignar cada registro virtual a un registro físico disponible. Esto incluye la resolución de conflictos y, si es necesario, realizar *spilling*.

*Optimizaciones Locales:* LLVM realiza optimizaciones locales, como fusionar registros si no hay interferencia y reordenamiento de instrucciones para minimizar accesos a memoria.

### 2.1.2. Optimización del Código Intermedio

Se utilizó un gestor de optimizaciones (`ModulePassManager`) para aplicar transformaciones clave al código intermedio antes de generar el código ensamblador. Las optimizaciones principales son:

*Eliminación de Código Muerto:* Asegura que las instrucciones y bloques que no afectan el resultado final sean eliminados, lo que redujo el tamaño del código ensamblador y finalmente representa una mejora en el tiempo de ejecución.

*Simplificación de CFG:* Se redujo la complejidad del flujo de control, eliminando saltos redundantes y simplificando bloques básicos.

*Combinación de Instrucciones:* LLVM transformó instrucciones múltiples y redundantes en operaciones más eficientes, lo que redujo la latencia y el consumo de recursos.

### 2.1.3. Generación de Ensamblador y Objeto

La generación del código ensamblador y objeto implicó la selección de `TargetMachine`. Esta decisión buscó un equilibrio entre tiempo de compilación y calidad del código generado. Asimismo:

*Configuración del Triple Objetivo:* Se utilizó el triple `x86_64-pc-windows-msvc`, lo que asegura compatibilidad con sistemas Windows modernos.

*Limpieza del Ensamblador:* Antes de ensamblar, se eliminó cualquier directiva incompatible, lo que garantizó que el ensamblador generado pudiera ser procesado correctamente por Clang.

*Clang para Ensamblar:* La elección de Clang como ensamblador permitió aprovechar su soporte nativo para LLVM para simplificar la integración.

### 2.1.4. Manejo de Errores

Se implementaron mecanismos extensivos para detectar errores en cada etapa del proceso de compilación. Entre estos la validación del IR inicial y el optimizado, pues se verificó que el código intermedio generado

y optimizado fuera válido antes de pasar a etapas posteriores. Asimismo, los errores se capturaron mediante excepciones, las cuales brindan mensajes claros y específicos para facilitar la depuración.

## OPTIMIZACIONES

La generación de código en el proyecto empleó diferentes optimizaciones proporcionadas por LLVM para mejorar la eficiencia y calidad del código generado. Dichas optimizaciones se aplicaron tanto al nivel del código intermedio (IR) como durante la generación del código ensamblador final. Las optimizaciones más significativas fueron:

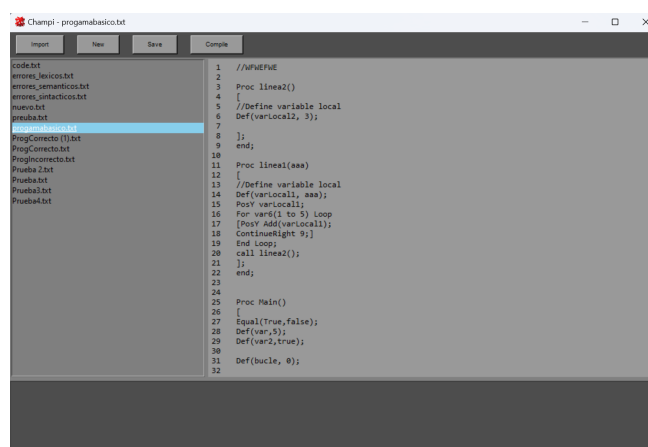
- **Fusión de Constantes:** Se identificaron constantes redundantes definidas varias veces en el IR y se combinan en una única, lo que reduce el tamaño del código y los accesos innecesarios a memoria.
- **Eliminación de Código Muerto:** Se eliminan instrucciones o bloques de código que no afectan el resultado final del programa.
- **Combinación de Instrucciones:** Ciertas instrucciones se simplifican y se combinan en operaciones equivalentes más simples. Por ejemplo, expresiones como  $x + 0$  o  $x \times 1$  son sustituidas por  $x$ .
- **Simplificación del Grafo de Control:** Se optimizó el flujo de control eliminando bloques básicos innecesarios y simplificando saltos condicionales. En este caso algunos bloques generados por condicionales y bucles fueron combinados o eliminados.
- **Optimización de Código Ensamblador:** Se aplicaron optimizaciones específicas para la arquitectura x86\_64. Estas incluyeron la selección eficiente de instrucciones y el reordenamiento para maximizar el paralelismo.
- **Reducción de Accesos a Memoria:** Los accesos redundantes a memoria fueron minimizados mediante análisis avanzado. Esto redujo el costo de almacenamiento en memoria.
- **Optimización de Registros:** Los registros virtuales fueron combinados y reutilizados eficientemente, lo cual redujo la necesidad de *spilling* y mejoró el rendimiento general del código.

Estas optimizaciones resultaron en un código eficiente y alineado con los estándares de la arquitectura utilizada.



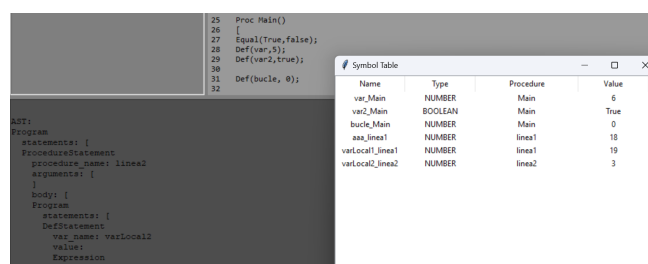
## PRUEBAS

La prueba realizada consistió en la ejecución de un programa básico funcional del lenguaje empleando Champivaca. Inicialmente se ejecuta el código, que mostrará de forma inmediata la tabla de símbolos, el AST, errores (de haber), código IR optimizado, código ensamblador e inclusive código objeto de forma interna. La secuencia de los procesos sigue una línea similar a la anteriormente descrita, con la excepción de que la tabla de símbolos se obtiene posterior al análisis sintáctico. A continuación, se observa una serie de imágenes que ilustran la mayoría de estos procesos y sus resultados.



```
1 //WUeFuE
2
3 Proc linea2()
4 [
5   //Define variable local
6   Def(varLocal2, 3);
7
8 ];
9
10 end;
11
12 Proc linea1(aaa)
13 [
14   //Define variable local
15   Def(varLocal1, aaa);
16   Post varLocal1;
17   For var1(1 to 5) Loop
18     [Post Add(varLocal1);
19     ContinueRight #];
20   End Loop;
21   call linea2();
22 ];
23 end;
24
25 Proc Main()
26 [
27   Equal(True,false);
28   Def(var,5);
29   Def(var2,true);
30   Def(bucle, 0);
31
32 ]
```

Figura 1: Ejemplo de programa básico funcional de Champivaca.



Name	Type	Procedure	Value
var_Main	NUMBER	Main	6
var2_Main	BOOLEAN	Main	True
bucle_Main	NUMBER	Main	0
aaa_linea1	NUMBER	linea1	18
varLocal1_linea1	NUMBER	linea1	19
varLocal2_linea2	NUMBER	linea2	3

```
AST:
Program
  statements: [
    ProcedureStatement
      procedure_name: linea2
      arguments: [
        ]
      body: [
        Program
          statements: [
            DefStatement
              var_name: varLocal2
              value:
                Expression
                  values:
```

Figura 2: Ejemplo de la tabla de símbolos y el AST generado para un programa básico funcional de Champivaca.

```
define void @linea2() {  
entry:  
    ret void  
}  
  
define void @lineal() {  
entry:  
    ret void  
}  
  
define void @Main() {  
entry:  
    %var = alloca i32, align 4  
    br label %while_condition  
  
while_condition:
```

Figura 3: Ejemplo de código IR optimizado para un programa básico funcional de Champivaca.

```
El assembly es:  
Código ensamblador generado exitosamente:  
    .text  
    .file    "<string>"  
    .globl   linea2  
    .p2align 4, 0x90  
    .type    linea2,@function  
linea2:  
    .cfi_startproc  
    retq  
.Lfunc_end0:  
    .size    linea2, .Lfunc_end0-linea2  
    .cfi_endproc  
  
    .globl   lineal  
    .p2align 4, 0x90
```

Figura 4: Ejemplo de código ensamblador generado a partir de la IR optimizada de un programa básico funcional de Champivaca.

## PROBLEMAS CONOCIDOS

**Control de errores léxicos:** Los errores dentro de los tres tipos de análisis son controlados mediante listas definidas como globales. Para poder realizar la ejecución continua de diferentes programas, cada vez que se ejecuta la función `Compile`, se reestablece su valor. Sin embargo, la lista relacionada a los errores léxicos no se reestablece de manera correcta. Esto provoca que los errores léxicos de programas anteriores se sigan visualizando en ejecuciones posteriores.

**Terminal amigable al usuario:** La terminal del entorno de desarrollo diseñado imprime todos los resultados de las ejecuciones de los tres análisis. Dependiendo del tamaño del programa, esta salida puede ser más extensa o no. En casos donde la salida es extensa, el usuario debe navegar manualmente por la terminal. Esto debido a que no se logró implementar en la interfaz gráfica una barra de navegación rápida. Impidiendo una característica importante en la amigabilidad del sistema.

**Expresiones faltantes:** Actualmente, en la implementación del análisis sintáctico y semántico del sistema, no se han logrado integrar de manera adecuada expresiones clave como las funciones y el bloque `main`, elementos esenciales para la ejecución y definición de la lógica del programa. Esto implica que no se está considerando correctamente la estructura jerárquica del código ni su flujo de control principal, lo que afecta la interpretación y validación de los programas.

Además, otro desafío presente es la falta de diferenciación entre variables locales y globales dentro del análisis. En la etapa actual, todas las variables son tratadas por igual, lo cual puede causar conflictos de contexto y errores en la gestión del alcance de las variables, afectando la semántica correcta del programa.

Ambos aspectos —la falta de reconocimiento de las funciones y el bloque `main`, así como la diferenciación de variables— limitan la precisión del análisis y generan dificultades a la hora de detectar errores o validar el comportamiento de los programas.

**Limitación semántica:** En la implementación actual, las reglas gramaticales no presentan ambigüedad en la sintaxis; sin embargo, algunos problemas fueron resueltos mediante generalizaciones que han tenido consecuencias en el análisis semántico. Esta generalización ha permitido que ciertas funciones y operadores, que normalmente deberían restringirse a recibir expresiones específicas, acepten de manera inapropiada expresiones de control, como bucles `for`, dentro de sus parámetros.

Este comportamiento es problemático porque compromete la semántica del sistema, permitiendo que estructuras de control que deberían estar limitadas a contextos específicos se utilicen dentro de expresiones donde no tienen sentido lógico o operativo. Por ejemplo, operadores simples podrían aceptar bloques de código como parámetros, lo que no es deseable ni correcto en términos de la semántica del lenguaje.

## 6

**PROBLEMAS ENCONTRADOS**

## 6.1 ERRORES EN EL RECONOCIMIENTO DE EXPRESIONES DEBIDO A LA AMBIGÜEDAD GRAMATICAL

En el desarrollo inicial del sistema, se identificaron errores repetidos en el reconocimiento de expresiones, lo cual afectaba la interpretación y validación de las mismas. Este problema surgió debido a la ambigüedad gramatical en las reglas diseñadas para el lenguaje. En específico, las expresiones que involucraban palabras reservadas del lenguaje, como operadores y estructuras de control, presentaban confusiones durante el análisis sintáctico haciendo que no fueran reconocidas a lo largo del análisis.

Este tipo de ambigüedad ocurre cuando una misma cadena de tokens puede ser interpretada de muchas maneras por la gramática, generando problemas en la descomposición de la estructura del código y produciendo resultados inesperados.

Inicialmente, se intentó resolver la ambigüedad mediante una generalización de todas las expresiones. La idea era tratar todos los tipos de expresiones bajo una misma categoría gramatical, en este caso statement. El objetivo de esta solución era simplificar el análisis sintáctico, permitiendo que todas las expresiones fueran tratadas de manera uniforme sin importar su contexto.

Sin embargo, generó nuevas complicaciones. Al generalizar las expresiones, se redujo la capacidad de la gramática para diferenciar entre distintos tipos de estructuras de control, operadores y otras palabras reservadas. Y el sistema no podía determinar el contexto correcto en el que se usaban ciertas expresiones. Como resultado, la generalización de las expresiones no resolvió el problema y complicó aún más el análisis semántico [1].

Para solucionar el problema de verdad, se crearon expresiones específicas para cada palabra reservada del lenguaje. En lugar de tratar todas las expresiones de manera uniforme, se definieron reglas gramaticales especí-

ficas para cada operador, estructura de control y palabra reservada. Esto permitió que la gramática identificara de manera precisa el uso de cada elemento dentro del código, eliminando la ambigüedad [4].

Para evitar este tipo de errores en el futuro se recomienda una descomposición gramatical temprana. En lugar de generalizar expresiones desde el inicio, es preferible establecer reglas claras para cada tipo de expresión que se espera encontrar en el lenguaje. Esto facilita el manejo de casos específicos sin generar confusión.

En conclusión, se puede reafirmar que el problema de la ambigüedad gramatical es común en el diseño de lenguajes de programación. Y que la solución más eficiente fue la creación de reglas específicas para cada palabra reservada y operador.

## 6.2 PROBLEMAS DE RECURSIVIDAD EN LA CREACIÓN DEL ÁRBOL Y ANÁLISIS SEMÁNTICO AL MANEJAR EXPRESIONES ANIDADAS

Otro de los problemas más complejos y que llevó más tiempo de resolver fue que surgió durante el desarrollo la recursividad descontrolada al intentar construir el árbol sintáctico y realizar el análisis semántico de expresiones que contenían otras expresiones dentro de sus parametros. Esta situación se dio durante los casos de prueba y era esperable ya que nunca se pensó en una solución a esto.

El sistema se enfrentaba a un problema donde, al analizar expresiones dentro de otras expresiones, la recursividad no estaba bien controlada y tiraba un error de no poder crear nodos, lo que causaba un comportamiento no deseado en el análisis del código. Esto resultaba en bucles infinitos o en un análisis incompleto, lo cual afectaba directamente la capacidad de evaluar correctamente las expresiones semánticas y de generar el árbol sintáctico correspondiente [2].

La primera solución que se intentó fue verificar los tipos de entrada en cada operador. Esta solución buscaba resolver el problema al asegurar que cada operador recibiera entradas válidas y coherentes antes de realizar el análisis de la expresión. Pero no funcionó como se esperaba.

Aunque esta verificación de tipos ayudó a identificar algunos errores, no resolvió el problema de fondo ya que igual no se realizaba ninguna operación. La recursividad seguía apareciendo porque el análisis sintáctico y semántico continuaba anidando llamadas recursivas sin un control efectivo.

La solución efectiva para este problema fue la implementación de un método general de visita para los nodos del árbol sintáctico. Este patrón, inspirado en el Visitor Pattern común aprendido en la tarea de investigación del intérprete, consiste en definir un método que permite recorrer el árbol sintáctico de manera flexible y controlada. Dependiendo del tipo de nodo que se encuentre durante el recorrido, se ejecuta una visita específica que define cómo procesar ese nodo y sus hijos.

Por estas situaciones, es recomendable implementar un patrón de visita al inicio del desarrollo de un compilador o analizador de lenguajes cuando se prevea el uso de expresiones anidadas. Esto ayuda a manejar la recursividad y garantiza un flujo de análisis claro y eficiente.

En conclusión, el problema de la recursividad se puede abordar de manera sencilla con la implementación de un método general de visita, basado en el Visitor Pattern. Este método permite manejar las expresiones anidadas de manera controlada, evitando bucles recursivos innecesarios y mejorando tanto la eficiencia como la precisión del análisis.

### 6.3 PROBLEMAS PARA DEFINIR OPERACIONES NUMÉRICAS FUERA DE PARÁMETROS

Otro problema encontrado fue que el sistema empezó a fallar al intentar definir operaciones numéricas fuera del contexto de los parámetros, lo que afectaba la capacidad de realizar cálculos aritméticos en otras partes del código. Este problema se manifestó específicamente en bucles y estructuras de control, donde las expresiones aritméticas, como las condiciones de parada en bucles while, no se podían procesar correctamente.

Aunque el sistema permitía definir operaciones numéricas dentro de parámetros, cuando se intentaba usar expresiones aritméticas en otros contextos, estas no eran reconocidas. Lo que generaba errores de semántica y ejecución. Esto limitaba el uso de operaciones básicas como las definidas en el cuadro de sintaxis mínimas.

El primer intento de solución fue en corregir la gramática para permitir que las expresiones aritméticas pudieran definirse fuera de los parámetros. Se intentó ampliar las reglas gramaticales existentes para cubrir estos nuevos contextos de uso, como condiciones en bucles o expresiones dentro de bloques condicionales.

Sin embargo, esta solución inicial no funcionó debido a los problemas de ambigüedad gramatical previamente mencionados en otro problema encontrado. La gramática seguía generando conflictos al interpretar expresiones aritméticas fuera de los parámetros.

La solución final fue la corrección simultánea tanto de la gramática como del manejo de la ambigüedad. Esto debido a que la ambigüedad en las expresiones afectaba tanto a la interpretación de las operaciones aritméticas como a otras áreas del código. Entonces se abordaron ambos problemas en paralelo.

Derivado de lo aprendido en este problema, se puede recomendar que cuando se enfrenten problemas de ambigüedad en la gramática, se aborden los problemas de manera conjunta, en lugar de intentar corregir cada aspecto por separado.

Además, como conclusión, el problema de no poder definir operaciones numéricas fuera de los parámetros evidenció la interdependencia entre las reglas gramaticales y la necesidad de manejar adecuadamente la ambigüedad en el lenguaje.

#### 6.4 CONEXIÓN ENTRE EL IDE Y EL CÓDIGO DEL ANALIZADOR QUE SOLO RECIBÍA ARCHIVOS .TXT

Por último, otro de los problemas encontrados durante el desarrollo del sistema fue la incapacidad del código del analizador para recibir y procesar entradas directamente desde el entorno de desarrollo integrado (IDE). El analizador está diseñado para leer archivos en formato .txt, lo que impedía una integración fluida con el IDE de compilación creado para el proyecto.

Esto implicaba que cualquier código que se quisiera analizar o ejecutar debía ser manualmente guardado en un archivo .txt antes de ser procesado por el analizador. Esto limitaba la funcionalidad del sistema, ya que no existían métodos para conectar ambas clases.

El primer intento de solucionar este problema fue diseñar una lista de instrucciones que pudieran ser interpretadas directamente por el analizador. Sin embargo, este enfoque demostró ser demasiado complejo, ya que requería la construcción de un sistema adicional para interpretar y procesar las listas de instrucciones.

La solución que resultó más efectiva fue mantener el uso de archivos .txt como interfaz de comunicación entre el IDE y el analizador, pero ayudar a ser más fluido el proceso. Se desarrolló una funcionalidad en el IDE que permitía generar automáticamente un archivo .txt a partir del código escrito en el entorno de compilación [3].

Gracias a lo aprendido en este caso, se recomienda que en casos donde la infraestructura existente tiene limitaciones técnicas (como el uso de archivos de texto para la comunicación entre sistemas), se busque automatizar el proceso en lugar de rediseñar completamente el sistema.

Además, se mostró que, la simplificación de soluciones es la manera más efectiva de solución. Aunque se exploraron soluciones más complejas, como el uso de listas de instrucciones, el enfoque más viable fue automatizar el uso de los archivos .txt, permitiendo que el flujo entre el IDE y el analizador se realizara sin intervención manual.

## 7

## CONCLUSIONES

La modularidad del sistema fue muy importante en cada etapa de desarrollo del compilador, desde el análisis léxico hasta el semántico. Al dividir el proyecto en módulos claramente definidos (léxico, sintáctico, visitor, analyzer e ide), se facilitó la comprensión del código, y también la localización y resolución de errores. La utilización del patrón Visitor para el análisis semántico y la generación del AST (Abstract Syntax Tree) permitió manejar eficientemente las expresiones y mejorar la reutilización del código. Esta estrategia modular hizo que el sistema fuera más escalable y permitiera la adición de nuevas características en el futuro con menor riesgo de generar conflictos, ya que, como se observa en el código fuente, cada clase es muy robusta por su naturaleza de solución.

Una de las ventajas más significativas que tiene el ide es la simplicidad y automatización del proceso de creación y lectura de archivos .txt entre el IDE y el compilador. Aunque inicialmente surgieron problemas en la conexión entre el IDE y el código del analizador, la solución de generar y procesar estos archivos automáticamente simplificó el flujo de trabajo. Haciendo más entendible y eficaz el manejo de código, ya que los 3 módulos fueron probados bajo archivos de este tipo. Esta automatización no solo facilitó la interacción entre el compilador y el IDE, sino que también mejora la experiencia del usuario.

A lo largo del desarrollo del analizador, implementar un sistema sólido de manejo de errores fue esencial para identificar y corregir problemas. El sistema de manejo de errores no solo identifica errores léxicos, sintácticos y semánticos, sino que también informa al usuario de manera clara y precisa, lo que ayuda a corregir el código más fácilmente, incluyendo detalles del error. Esto refleja la importancia de diseñar compiladores y lenguajes con un enfoque robusto en la detección y reporte de errores, ya que la claridad en los mensajes de



error es importante para mejorar la experiencia del usuario.

Por último, las ambigüedades gramaticales presentaron un desafío importante, especialmente al intentar manejar expresiones complejas y operadores ya que generaban errores de recursividad y reconocimiento. Sin embargo, al crear expresiones específicas para cada palabra reservada y corregir la gramática para eliminar estas ambigüedades, el sistema logró procesar correctamente las distintas construcciones del lenguaje y construir el árbol de manera efectiva. Esta experiencia demostró que el tratamiento adecuado de la gramática es esencial para asegurar que el compilador pueda funcionar correctamente.

## 8

**RECOMENDACIONES**

Primero, aunque ya se ha implementado un sistema de manejo de errores sólido, es recomendable seguir mejorando este aspecto del compilador ya que el producto actual no es el final. Agregar categorías adicionales de errores y ofrecer sugerencias automáticas para corregir errores comunes puede ser una mejora muy importante para mejorar la experiencia de usuario. Además, se podrían incluir advertencias (warnings) para señalar posibles problemas no críticos en el código, lo que mejoraría aún más la experiencia del usuario y ayudaría a evitar errores futuros antes de la ejecución del programa.

Dado que el compilador ya incluye un sistema para la generación de un AST y el análisis semántico, el siguiente paso podría ser optimizar estos procesos para mejorar el rendimiento, lo cual se va a hacer en el siguiente Sprint. La optimización de la generación de código, ya sea en términos de eficiencia computacional o en la reducción de complejidad en las operaciones del compilador, proporciona un beneficio, especialmente en casos donde el tamaño del código fuente es grande o contenga expresiones complejas. También se recomienda explorar la optimización del árbol de sintaxis abstracta para reducir su tamaño o mejorar la velocidad de ejecución.

Además, a medida que el compilador y el lenguaje se desarrollen, se puede considerar la adición de más características avanzadas no implementadas en este Sprint, como la posibilidad de crear y manejar funciones de orden superior, la integración de manejo de memoria dinámica, o la implementación de closures. También sería recomendable incluir soporte para más estructuras de control y tipos de datos avanzados, lo que haría que el lenguaje sea más versátil y útil.

Por último, en vistas más a futuro del funcionamiento, es recomendable integrar pruebas automatizadas que verifiquen el funcionamiento correcto del sistema en múltiples escenarios antes de su integración real. Cada vez que se agregue una nueva característica o se modifique alguna parte del compilador, estas pruebas permitirán verificar que no se introducen nuevos errores. Esto permitiría al desarrollador una mayor fluidez en la escritura y actualización de código.

## REFERENCIAS

- [1] G. HERNÁNDEZ, *Gramáticas Ambiguas*, [https://web2.uaeh.edu.mx/docencia/P\\_Presentaciones/huejutla/sistemas/2017/Gramaticas\\_Ambiguas.pdf](https://web2.uaeh.edu.mx/docencia/P_Presentaciones/huejutla/sistemas/2017/Gramaticas_Ambiguas.pdf), 2017.
- [2] GINNI, *What is Left Recursion and how it is eliminated?* <https://www.tutorialspoint.com/what-is-left-recursion-and-how-it-is-eliminated>, 2021.
- [3] W.J. T. y ADAM CRYMBLE, *Trabajar con archivos de texto en Python*, <https://programminghistorian.org/es/lecciones/trabajar-con-archivos-de-texto>, 2022.
- [4] E. A. F. MARTÍNEZ, *Análisis Sintáctico II*, [https://docencia.eafranco.com/materiales/compiladores/17\\_Analisis\\_sintactico\\_II.pdf](https://docencia.eafranco.com/materiales/compiladores/17_Analisis_sintactico_II.pdf).