PROYECTO COMPILADOR TINYRU
Manual Técnico

Autores:
-Cairo Lucia Virginia
-Sanchez Lanza Agustina

Dra. Ana Carolina Olivera
Marcelo Adrián Zufía
Compiladores
10 de Junio de 2024

ÍNDICE

1. INTRODUCCIÓN

En el siguiente informe detallamos el desarrollo del compilador TinyRu, este trabajo fue desarrollado a lo largo de un semestre, para la materia Compiladores. Es un proyecto diseñado para ilustrar y aplicar los principios fundamentales de la compilación. A lo largo del proyecto, hemos implementado las etapas esenciales que componen un compilador robusto y funcional: el análisis léxico, el análisis sintáctico predictivo recursivo, y el análisis semántico, junto con la construcción de la tabla de símbolos (TS) y el árbol de sintaxis abstracta (AST) y por ultimo la etapa final de generacion de codigo.

En la etapa inicial, el analizador léxico se encargó de descomponer el código fuente en tokens, elementos básicos que representan las unidades léxicas del lenguaje.

Posteriormente, el analizador sintáctico predictivo recursivo utilizó estos tokens para construir una estructura jerárquica que valida la correcta construcción gramatical del código fuente, garantizando que se adhiera a las reglas de la gramática definida para TinyRu, propuestas por la cátedra.

El análisis semántico, complementado por la tabla de símbolos y el árbol de sintaxis abstracta, verificó la coherencia y validez de las expresiones y declaraciones en el código, asegurando que los tipos de datos y las referencias a variables y funciones fueran correctos y consistentes dentro del contexto del programa.

En esta etapa final del proyecto, nos hemos centrado en la generación de código máquina, un componente crucial que traduce el código de alto nivel a un conjunto de instrucciones ejecutables por la máquina. Para esta tarea, hemos utilizado las instrucciones del conjunto de instrucciones MIPS (Microprocessor without Interlocked Pipeline Stages.

La generación de código máquina implica tomar las representaciones intermedias, como el AST, y convertirlas en instrucciones MIPS eficientes y correctas. Esta fase no solo transforma el código en una forma que pueda ser ejecutada por el hardware, sino que también optimiza y mejora la eficiencia del programa resultante. La implementación de esta etapa nos permitió consolidar y aplicar los conocimientos adquiridos en todas las fases previas, resultando en un compilador que no solo analiza y valida el código fuente, sino que también produce un código ejecutable y optimizado para la arquitectura MIPS.

A lo largo del siguiente documento, documentamos cada una de estas etapas, explicando las soluciones encontradas para cada una de las etapas, en especial se hará énfasis en la generación de código máquina, demostrando el avance y la culminación de nuestro proyecto TinyRu

2. TECNOLOGÍAS UTILIZADAS

Durante el desarrollo del compilador, para todas las etapas, se utilizó IntelliJ IDEA como el entorno de desarrollo principal. El proyecto se configuró como un proyecto Maven para gestionar las dependencias y la construcción del software. Se empleó la última versión de Java 21 como el lenguaje de programación principal. El código fuente del proyecto se ubicó en GitHub, utilizando Git como el sistema de control de versiones para gestionar las ramas y colaborar en el desarrollo del proyecto. Gracias a estas herramientas se logró trabajar en paralelo.

3. ETAPAS DEL COMPILADOR

A continuación, se presenta el desarrollo del compilador TinyRu, detallando cada una de sus etapas fundamentales. Comenzaremos con el análisis léxico, seguido por el análisis sintáctico predictivo recursivo, y el análisis semántico, incluyendo la construcción de la tabla de símbolos (TS) y el árbol de sintaxis abstracta (AST). Finalmente, se abordará la generación de código máquina, utilizando instrucciones MIPS, que constituye la etapa culminante de nuestro proyecto.

3.1. Analizador Léxico

El analizador léxico es la primera fase del compilador TinyRu y tiene como objetivo transformar la secuencia de caracteres del código fuente en una secuencia de tokens. Cada token representa una unidad léxica significativa, como palabras clave, identificadores, operadores, literales y símbolos especiales. Esta etapa es crucial para simplificar el análisis sintáctico posterior al segmentar el texto en componentes manejables y eliminando elementos irrelevantes como espacios en blanco y comentarios.

El diseño del mismo se realizo de forma escalonada, primero definiendo un alfabeto de entrada, luego la tabla de tokens, lo siguiente fue pensar la arquitectura del analizador, las posibles clases a programar, entre otras cosas.

3.1.1. Alfabeto de entrada

Uno de los pasos esenciales antes de comenzar el desarrollo del compilador fue definir el alfabeto de entrada, que consiste en todos los posibles símbolos o caracteres que pueden aparecer en el código fuente.Este alfabeto se seleccionó cuidadosamente para abarcar todos los símbolos y caracteres relevantes que pueden aparecer en el código fuente, teniendo en cuenta el lenguaje de programación reconocido en el manual Tiny Ru.

En la siguiente tabla ‘Tabla 1’, se presentará una descripción detallada del alfabeto considerado, incluyendo los diferentes tipos de caracteres y símbolos especiales. Este análisis del alfabeto es crucial para comprender cómo se realiza la identificación y clasificación de los tokens durante el proceso de análisis léxico.

ALFABETO RECONOCIDO		
[0…9]	@	*
/	+	-
%	&	;
:	.	,
|	¿	?
=	<	>
\	!	“
‘	{	}
[	]	(

(	)	_
[A…Z]	[a…z]	ñ
$	“ ” (representa el espacio)	

Tabla 1: elaboración propia del alfabeto reconocido por el analizador léxico

3.1.2. Tabla de Tokens

A continuación, la tabla 2 presenta una extensión detallada que muestra los tokens reconocidos por el analizador léxico junto con sus respectivas expresiones regulares. Cada token está asociado con una expresión regular que describe su formato y cómo se identifica en el código fuente.

La tabla proporciona una referencia útil para comprender cómo se clasifican y reconocen los diferentes elementos del lenguaje de programación durante el análisis léxico.

Tokens	Expresión Regular
id	(a…z) + ((a...z)+(A…Z)+(0…9)+_)*
struct_name	(A…Z) + ((a...z)+(A…Z)+(0…9)+_)*((a…z)+(A…Z))
keyword_struct	struct
keyword_impl	impl
keyword_else	else
keyword_false	false
keyword_if	if
keyword_ret	ret
keyword_while	while
keyword_true	true
keyword_nil	nil
keyword_new	new
keyword_fn	fn
keyword_st	st
keyword_pri	pri
keyword_self	self
int	[0..9]*
str	“Σ*”

char	‘Σ’ + ‘\Σ’
par_open	(
par_close	)
cor_open	[
cor_close	]
braces_open	{
braces_close	}
semicolon	;
comma	,
colon	:
period	.
class_Int	Int
class_Bool	Bool
class_String	Str
class_Char	Char
class_Array	Array
class_IO	IO
class_Object	Object
type_void	void
op_assig	=
op_less	<
op_greater	>
op_less_equal	<=
op_greater_equal	>=
op_equal	==
op_not_equal	!=
op_sum	+
op_rest	-

op_mult	*
op_div	/
op_mod	%
op_incr	++
op_decr	--
op_and	&&
op_or	||
op_not	!
ret_func	->

Tabla 2: tokens reconocidos por el analizador léxico.

3.1.3. Errores léxicos detectables

Durante este proceso, es común encontrar errores léxicos, que son aquellos que violan las reglas de la gramática del lenguaje de programación. En esta sección, se exponen los errores léxicos que hemos identificado y manejado en nuestro compilador. Estos casos abarcan una amplia variedad de situaciones en las que el código fuente puede contener errores léxicos, desde tokens no reconocidos hasta caracteres inesperados o malformados. Varios de estos errores fueron extraídos del manual Tiny Ru, mientras que otros se basaron en el lenguaje Java.

Las excepciones léxicas tendrán como salida el número de línea donde se produjo el error, número de columna y una breve descripción:

ERROR: LEXICO

| NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION:|

Estos son los que tomamos en cuenta:

- Carácter mal formado. Cuando se encuentra un caracter sin comilla de cierre.
  Ejemplo: `a
  | *DESCRIPCION: "Se esperaba un fin de carácter (') después de a"* |
- Carácter mal formado. Cuando se encuentra un caracter con mas de un elemento.
  Ejemplo: `aa’
  | *DESCRIPCION: "Se esperaba un fin de carácter (') después de a"* |
- Caracter inválido. Cuando se encuentra un caracter nulo.
  Ejemplo: `\0`
  | *DESCRIPCION: "Caracter invalido.Los caracteres no permiten caracter null"*
- Carácter invalido. Cuando se encuentra un caracter con codigo ascii 26 (considerado como EOF en nuestro compilador).
  | *DESCRIPCION:* "Caracter invalido. Los caracteres no permiten EOF" |

- String mal formado. Cuando se encuentra un caracter sin comillas de cierre. Ejemplo: “hola
  | *DESCRIPCION: “*String mal formado. Se esperaba un fin de string*”* |

- String invalido. Cuando se encuentra un string con mas de 1024 caracteres. Ejemplo: “nnnnnnn……” (1025 ‘n’ o más)
  | *DESCRIPCION:* "String supera la cantidad maxima de caracteres en cadena"

- String invalido. Cuando se encuentra un caracter null en un string. Ejemplo: “hola \0 chau”
  | *DESCRIPCION:* "Caracter invalido. Los Str no permiten caracter null" |

- String invalido. Cuando se encuentra un caracter con codigo ascii 26 (considerado como EOF en nuestro compilador) dentro del string.
  | *DESCRIPCION:* "Caracter invalido. Los string no permiten EOF" |

- Operador AND mal formado. Cuando se encuentra un operador AND incompleto. Ejemplo: condicion1 & condicion2
  | *DESCRIPCION:* "Se ha encontrado un solo '&' en la línea. Se esperaba un operador and (&&)" |

- Operador OR mal formado. Cuando se encuentra un operador OR incompleto. Ejemplo: condicion1 | condicion2
  | *DESCRIPCION:* "Se ha encontrado un solo '|' en la línea. Se esperaba un operador or (||)" |

- Identificador de Clase mal formado. Cuando se encuentra un identificador de clase que no finaliza con letra.
  Ejemplo: Clase1 , Clase_
  | *DESCRIPCION: "*"Los nombres de clase deben finalizar con letra. El identificador es invalido " |

- Identificador de Clase invalido. Cuando se encuentra un identificador de clase que contiene un caracter invalido.
  Ejemplo: Cl@se
  | *DESCRIPCION:* "Caracter invalido `@`" |

- Carácter invalido. Cuando se encuentra un caracter griego, hangul o chino. Ejemplo: ‘龍 ‘, ‘사랑 ‘, ‘α‘
  | *DESCRIPCION: "*"Caracter Invalido” |

Estos son los principales errores léxicos que hemos identificado y manejado en nuestro compilador. La política ante la detección de un error es abortar el análisis léxico. No se implementan técnicas de recuperación ante errores.

3.1.4. Casos de Pruebas Léxicos

En esta sección se detallan las pruebas realizadas para evaluar el funcionamiento del analizador léxico. Cada prueba se describe con su respectivo escenario, nombre del test y la salida esperada. Dichas pruebas abarcan una variedad de situaciones para garantizar la robustez y precisión del analizador en diferentes contextos y casos de uso. A continuación, se presenta un resumen de las pruebas realizadas:

Escenario	Nombre Test	Salida esperada
Identificadores	test_id_ok Ver que se reconocen los identificadores correctamente (con numeros, letras y guiones bajos)	CORRECTO: ANALISIS LEXICO
Identificadores	test_id_invalid Ver que se da un mensaje de error adecuado cuando el identificador posee caracteres inválidos	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 8 | COLUMNA 5 | Caracter Invalido '@'|
Identificadores	test_id_keys Ver que se reconocen los identificadores que son palabras claves	CORRECTO: ANALISIS LEXICO
Identificadores de Clase	test_idClass_ok Ver que se reconocen los identificadores de clase correctamente (con numeros, letras y guiones bajos)	CORRECTO: ANALISIS LEXICO
Identificadores de Clase	test_idClass_keys Ver que se reconocen los identificadores de clase que clases predefinidas	CORRECTO: ANALISIS LEXICO
Identificadores de Clase	test_idClass_invalid Ver que se da un mensaje de error adecuado cuando el identificador de clase posee caracteres inválidos	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 8 | COLUMNA 5 | Caracter Invalido '@'|
Identificadores de Clase	test_idClass_num Ver que se da un mensaje de error adecuado cuando el identificador de clase termina con un numero	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 8 | COLUMNA 0 | Los nombres de clase deben finalizar con letra.
Identificadores de Clase	test_idClass_guion Ver que se da un mensaje de error adecuado cuando el identificador de clase termina con un guion	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 8 | COLUMNA 0 | Los nombres de clase deben finalizar con letra. Prueba_ es invalido |
Enteros	test_int_ok Ver que se reconocen los enteros correctamente	CORRECTO: ANALISIS LEXICO
String	test_str_ok Ver que se reconocen los strings correctamente	CORRECTO: ANALISIS LEXICO
String	test_str_null Ver que se da un mensaje de error adecuado cuando el string contiene un null (\0)	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 8 | COLUMNA 9 | Caracter

		invalido. Los Str no permiten caracter null|
String	test_str_limit Ver que se da un mensaje de error adecuado cuando el string contiene mas de 1024 caracteres	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 8 | COLUMNA 1030 | String supera la cantidad maxima de caracteres en cadenas|
String	test_str_unclosed Ver que se da un mensaje de error adecuado cuando el string no contiene " de cierre	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 8 | COLUMNA 10 | String mal formado. Se esperaba un fin de string |
String	test_str_chino Ver que se da un mensaje de error adecuado cuando el string contiene un caracter chino	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 8 | COLUMNA 5 | Caracter Invalido '你'|
String	test_str_griego Ver que se da un mensaje de error adecuado cuando el string contiene un caracter griego	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 8 | COLUMNA 5 | Caracter Invalido 'Ω'|
String	test_str_hangul Ver que se da un mensaje de error adecuado cuando el string contiene un caracter hangul	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 8 | COLUMNA 5 | Caracter Invalido '안'|
String	test_str_eof Ver que se da un mensaje de error adecuado cuando el string contiene un eof	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 8 | COLUMNA 5 | Caracter invalido. Los Str no permiten EOF|
Caracteres	test_char_ok Ver que se reconocen los caracteres correctamente	CORRECTO: ANALISIS LEXICO
Caracteres	test_char_special Ver que se reconocen los caracteres especiales correctamente (\n, \t, \r, \v)	CORRECTO: ANALISIS LEXICO
Caracteres	test_char_null Ver que se da un mensaje de error adecuado cuando el caracter contiene un null (\0)	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 8 | COLUMNA 0 | Caracter invalido. Los caracteres no permiten caracter null|
Caracteres	test_char_valid Ver que se reconocen los	CORRECTO: ANALISIS LEXICO

	caracteres correctamente: comillas simple (\'), comillas dobles (\") backslash (\\)	
Caracteres	test_char_invalid Ver que se da un mensaje de error adecuado cuando el caracter contiene mas de un elemento	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 8 | COLUMNA 1 | Caracter mal formado. Se esperaba fin de caracter (') despues de a|
Caracteres	test_char_fin Ver que se da un mensaje de error adecuado cuando el caracter esta sin comilla de fin '	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 7 | COLUMNA 1 | Caracter mal formado. Se esperaba fin de caracter (') despues de a|
Caracteres	test_char_spacio Ver que se reconocen los caracteres con esapacio correctamente ('\ ' y ' ')	CORRECTO: ANALISIS LEXICO
Caracteres	test_char_eof Ver que se da un mensaje de error adecuado cuando el caracter tiene eof	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 8 | COLUMNA 1 | Caracter invalido. Los caracteres no permiten EOF|
Operadores	test_op_equal Ver que se diferencian los operadores = y ==	CORRECTO: ANALISIS LEXICO
Operadores	test_op_sum Ver que se diferencian los operadores + y ++	CORRECTO: ANALISIS LEXICO
Operadores	test_op_rest Ver que se diferencian los operadores - y --	CORRECTO: ANALISIS LEXICO
Operadores	test_op_less Ver que se diferencian los operadores < y <=	CORRECTO: ANALISIS LEXICO
Operadores	test_op_greater Ver que se diferencian los operadores > y >=	CORRECTO: ANALISIS LEXICO
Operadores	test_op_not Ver que se diferencian los operadores ! y !=	CORRECTO: ANALISIS LEXICO
Operadores	test_op_and Ver que se da un mensaje de error adecuado cuando el operador AND esta mal formado	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 7 | COLUMNA 9 | Se ha encontrado un solo '&' en la línea. Se

		esperaba un operador and (&&)|
Operadores	test_op_or Ver que se da un mensaje de error adecuado cuando el operador OR esta mal formado	ERROR: LEXICO | NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: | | LINEA 7 | COLUMNA 9 | Se ha encontrado un solo '|' en la línea. Se esperaba un operador or (||)|
Todo	fibonacci Ver que se reconocen todos los tokens en un programa completo y real	CORRECTO: ANALISIS LEXICO
Archivo	empty Ver que se muestre un mensaje cuando el archivo esta vacio	ERROR: El archivo fuente '" + archivoFuente + "' está vacío.
Archivo	.text Ver que se muestre un mensaje cuando el archivo no es extension ru	ERROR: El archivo fuente debe tener la extensión '.ru'.
Archivo	EOF Ver que el analizador lexico solo forme tokens hasta que encuentre un eof	CORRECTO: ANALISIS LEXICO

3.1.5. Consideraciones del Analizador Léxico

Durante el desarrollo del analizador léxico, se tuvieron en cuenta diversas consideraciones, que garantizan el correcto funcionamiento del mismo

- Detección de fin de archivo (EOF): para detectar el final de un archivo, el analizador léxico reconoce el carácter ASCII 26 con indicador de fin de archivo.
- Almacenamiento de Tokens String: El analizador léxico guarda los tokens de tipo string tal como se leen literalmente del código fuente. Por ejemplo, el string "\a" se almacenaría como "\a" sin ninguna transformación adicional. No así con los caracteres, ya que si se encuentra '\a', se almacena 'a'.
- Espacio en los Char: Solo se considera el carácter correspondiente al espacio (' ') en los tokens de tipo char. Por lo tanto, el carácter de tabulación (\t) no se tiene en cuenta a menos que se encuentre literalmente escrito como '\t', lo mismo con los caracteres especiales que se traducen como espacios.

3.2. Analizador Sintáctico

La etapa sintáctica de un compilador verifica la estructura gramatical del código fuente, asegurándose de que cumpla con las reglas del lenguaje de programación. Convierte los tokens del analizador léxico en una estructura sintáctica, detecta errores y construye una representación del programa para las siguientes etapas del compilador. Este informe se enfocará en el análisis sintáctico descendente, detallando la eliminación de lambdas, recursividad y factorización, y el diseño implementado para esta fase.

3.2.1. Tratamiento de Gramática

Se utilizó la gramática propuesta por la cátedra, definida en el manual de TinyRU. La misma se encuentra en un primer momento en formato EBNF, luego de varios procesos (algoritmos) que serán mencionados y explicados en orden, a lo largo de la sección 2.2, se obtuvo la gramática equivalente LL(1). Se realizaron estas modificaciones en la gramática inicial para que cumpla las condiciones necesarias para utilizar un analizador sintáctico descendente predictivo recursivo.

Gramática EBFN del Manual TinyRu:

⟨*program*⟩ ::= ⟨*Lista-Definiciones*⟩ ⟨*Start*⟩
⟨*Start*⟩ ::= **start** ⟨*Bloque-Método*⟩
⟨*Lista-Definiciones*⟩ ::= (⟨Struct⟩ | ⟨Impl⟩)*
⟨*Struct*⟩ ::= **struct** idStruct ⟨*Herencia*⟩? **{** ⟨*Atributo*⟩* **}**
⟨*Impl*⟩ ::= **impl** idStruct **{** ⟨*Miembro*⟩+ **}**
⟨*Herencia*⟩ ::= "**:**" ⟨*Tipo*⟩
⟨*Miembro*⟩ ::= ⟨*Metodo*⟩ | ⟨*Constructor* ⟩
⟨*Constructor* ⟩ ::= "**.**" ⟨*Argumentos-Formales*⟩ ⟨*Bloque-Método*⟩
⟨*Atributo*⟩ ::= ⟨*Visibilidad*⟩? ⟨*Tipo*⟩ ⟨*Lista-Declaracion-Variables*⟩ **;**
⟨*Método*⟩ ::= ⟨*Forma-Método*⟩? **fn** idMetAt ⟨*Argumentos-Formales*⟩ **->**
⟨*Tipo-Método*⟩ ⟨*Bloque-Método*⟩
⟨*Visibilidad*⟩ ::= **pri**
⟨*Forma-Método*⟩ ::= **st**
⟨*Bloque-Método*⟩ ::= **{** ⟨*Decl-Var-Locales*⟩* ⟨*Sentencias*⟩* **}**
⟨*Decl-Var-Locales*⟩ ::= ⟨*Tipo*⟩ ⟨*Lista-Declaracion-Variables*⟩ **;**
⟨*Lista-Declaracion-Variables*⟩::= idMetAt
| idMetAt **,** ⟨*Lista-Declaracion-Variables*⟩
⟨*Argumentos-Formales*⟩::= **(** ⟨*Lista-Argumentos-Formales*⟩? **)**
⟨*Lista-Argumentos-Formales*⟩ ::= ⟨*Argumento-Formal*⟩ **,**
⟨*Lista-Argumentos-Formales*⟩ | ⟨*Argumento-Formal*⟩
⟨*Argumento-Formal*⟩ ::= ⟨*Tipo*⟩ idMetAt
⟨*Tipo-Método*⟩ ::= ⟨*Tipo*⟩ | **void**
⟨Tipo⟩ ::= ⟨*Tipo-Primitivo*⟩ | ⟨*Tipo-Referencia*⟩ | ⟨*Tipo-Arreglo*⟩
⟨*Tipo-Primitivo*⟩ ::= **Str** | **Bool** | **Int** | **Char**
⟨*Tipo-Referencia*⟩ ::= idStruct
⟨*Tipo-Arreglo*⟩ ::= **Array** ⟨*Tipo-Primitivo*⟩
⟨*Sentencia*⟩ ::= **;**
| ⟨*Asignación*⟩**;**
| ⟨*Sentencia-Simple*⟩**;**
| **if (**⟨*Expresión*⟩**)** ⟨*Sentencia*⟩
| **if (**⟨*Expresión*⟩**)** ⟨*Sentencia*⟩ **else** ⟨*Sentencia*⟩
| **while (** ⟨*Expresión*⟩ **)** ⟨*Sentencia*⟩
| ⟨Bloque⟩
| **ret** ⟨Expresion⟩? **;**
⟨*Bloque*⟩ ::= **{** ⟨*Sentencia*⟩ **}**
⟨*Asignacion*⟩ ::= ⟨*AccesoVar-Simple*⟩ **=** ⟨*Expresion*⟩
| ⟨*AccesoSelf-Simple*⟩ **=** ⟨*Expresion*⟩
⟨*AccesoVar-Simple*⟩ ::= id ⟨*Encadenado-Simple*⟩* | id **[** ⟨*Expresion*⟩ **]**
⟨*AccesoSelf-Simple*⟩ ::= "**self**" ⟨*Encadenado-Simple*⟩*
⟨*Encadenado-Simple*⟩ ::= "**.**" id
⟨*Sentencia-Simple*⟩ ::= **(**⟨*Expresion*⟩**)**
⟨*Expresion*⟩ ::= ⟨*ExpOr* ⟩
⟨*ExpOr* ⟩ ::= ⟨*ExpOr* ⟩ **||** ⟨*ExpAnd*⟩ | ⟨*ExpAnd*⟩
⟨*ExpAnd*⟩ ::= ⟨*ExpAnd*⟩ **&&** ⟨*ExpIgual*⟩ | ⟨*ExpIgual*⟩
⟨*ExpIgual*⟩ ::= ⟨*ExpIgual*⟩ ⟨*OpIgual*⟩ ⟨*ExpCompuesta*⟩ |
⟨*ExpCompuesta*⟩ ⟨*ExpCompuesta*⟩ ::= ⟨*ExpAd*⟩ ⟨*OpCompuesto*⟩

⟨*ExpAd*⟩ | ⟨*ExpAd*⟩
⟨*ExpAd*⟩ ::= ⟨*ExpAd*⟩ ⟨*OpAd*⟩ ⟨*ExpMul*⟩ | ⟨*ExpMul*⟩
⟨*ExpMul*⟩ ::= ⟨*ExpMul*⟩ ⟨*OpMul*⟩ ⟨*ExpUn*⟩ | ⟨*ExpUn*⟩
⟨*ExpUn*⟩ ::= ⟨*OpUnario*⟩ ⟨*ExpUn*⟩ | ⟨*Operando*⟩
⟨OpIgual⟩ ::= **==** | **!=**
⟨*OpCompuesto*⟩ ::= **<** | **>** | **<=** | **>=**
⟨*OpAd*⟩ ::= **+** | **-**
⟨*OpUnario*⟩ ::= **+** | **-** | **!** | **++** | **--**
⟨*OpMul*⟩ ::= ***** | */* | **%**
⟨*Operando*⟩ ::= ⟨*Literal*⟩ | ⟨*Primario*⟩ ⟨*Encadenado*⟩?
⟨*Literal*⟩ ::= **nil** | **true** | **false** | **intLiteral** | **StrLiteral** | **charLiteral**
⟨*Primario*⟩ ::= ⟨*ExpresionParentizada*⟩
| ⟨*AccesoSelf* ⟩
| ⟨*AccesoVar* ⟩
| ⟨*Llamada-Método*⟩
| ⟨*Llamada-Metodo-Estatico*⟩
| ⟨*Llamada-Constructor* ⟩
⟨*ExpresionParentizada*⟩ ::= **(** ⟨*Expresion*⟩ **)** ⟨*Encadenado*⟩?
⟨*AccesoSelf* ⟩ ::= **self** ⟨*Encadenado*⟩?
⟨AccesoVar ⟩ ::= **id** ⟨*Encadenado*⟩?
| **id [** ⟨*Expresión*⟩ **]**⟨*Encadenado*⟩?
⟨*Llamada-Método*⟩ ::= **id** ⟨*Argumentos-Actuales*⟩ ⟨*Encadenado*⟩ ?
⟨*Llamada-Método-Estático*⟩ ::= **idStruct .** ⟨*Llamada-Método*⟩ ⟨*Encadenado*⟩?
⟨*Llamada-Constructor* ⟩ ::= **new idStruct** ⟨*Argumentos-Actuales*⟩
⟨*Encadenado*⟩? | **new** ⟨*Tipo-Primitivo*⟩ **[** ⟨Expresion⟩ **]**
⟨*Argumentos-Actuales*⟩ ::= **(** ⟨Lista-Expresiones⟩? **)**
⟨*Lista-Expresiones*⟩ ::= ⟨Expresión⟩ | ⟨Expresión⟩ **,** ⟨*Lista-Expresiones*⟩
⟨*Encadenado*⟩ ::= "**.**" ⟨*Llamada-Método-Encadenado*⟩
| "**.**" ⟨*Acceso-Variable-Encadenado*⟩
⟨*Llamada-Método-Encadenado*⟩::= **id** ⟨*Argumentos-Actuales*⟩
⟨*Encadenado*⟩? ⟨*Acceso-Variable-Encadenado*⟩ ::= **id** ⟨*Encadenado-opt*⟩
| **id [** ⟨*Expresion*⟩ **]** ⟨*Encadenado*⟩?

3.2.1.1 Transformación EBNF a BNF

Pasar la gramática de Forma Extendida de Backus-Naur (EBNF) a Forma Normal de Backus-Naur (BNF) implica transformar la gramática de una notación más expresiva en una forma más estándar y estructurada que sea más fácil de procesar por el analizador sintáctico.

Para realizar la conversión de EBNF a BNF se eliminaron símbolos no terminales con "?" (0 o 1 ocurrencia), y con "+" (1 o más ocurrencias) y con "*" (0 o más ocurrencias). Estos se reemplazaron con nuevos no terminales como se muestra a continuación:

Aclaramos que en rojo se muestra la regla de produccion en formato EBFN y en verde como quedo modificada en formato BFN:

⟨Lista-Definiciones⟩ ::= (⟨Struct⟩ | ⟨Impl⟩) ∗

⟨Lista-Definiciones⟩ ::= ⟨Definiciones⟩
⟨Definiciones⟩ ::= ⟨Struct⟩ ⟨Definiciones⟩ | ⟨Impl⟩ ⟨Definiciones⟩ | ⟨Struct⟩ | ⟨Impl⟩ | λ

⟨Struct⟩ ::= struct idStruct ⟨Herencia⟩ **?** { ⟨Atributo⟩ ∗ }
⟨Struct⟩ ::= struct idStruct ⟨Herencia-opt⟩ { ⟨Atributos⟩ }
⟨Atributos⟩ ::= ⟨Atributo⟩ | ⟨Atributos⟩ ⟨Atributo⟩ | λ
⟨Herencia-opt⟩ ::= ⟨Herencia⟩ | λ

⟨Impl⟩ ::= impl idStruct { ⟨Miembro⟩ **+** }
⟨Impl⟩ ::= impl idStruct { ⟨Miembros⟩ }
⟨Miembros⟩ ::= ⟨Miembro⟩ | ⟨Miembro⟩ ⟨Miembros⟩

⟨Atributo⟩ ::= ⟨Visibilidad⟩ **?** ⟨Tipo⟩ ⟨Lista-Declaracion-Variables⟩ ;
⟨Atributo⟩ ::= ⟨Visibilidad-opt⟩ ⟨Tipo⟩ ⟨Lista-Declaracion-Variables⟩ ;
⟨Visibilidad-opt⟩ ::= ⟨Visibilidad⟩ | λ

⟨Método⟩ ::= ⟨Forma-Método⟩ **?** fn idMetAt ⟨Argumentos-Formales⟩ ->
⟨Tipo-Método⟩ ⟨Bloque-Método⟩
⟨Método⟩ ::= ⟨Forma-Método-opt⟩ fn idMetAt ⟨Argumentos-Formales⟩ ->
⟨Tipo-Método⟩ ⟨Bloque-Método⟩
⟨Forma-Método-opt⟩ ::= ⟨Forma-Método⟩ | λ

⟨Bloque-Método⟩ ::= { ⟨Decl-Var-Locales⟩ ∗ ⟨Sentencia⟩ ∗ }
⟨Bloque-Método⟩ ::= { ⟨Declaraciones⟩ ⟨Sentencias⟩ }
⟨Declaraciones⟩ ::= ⟨Decl-Var-Locales⟩ | ⟨Decl-Var-Locales⟩ ⟨Declaraciones⟩ |
λ ⟨Sentencias⟩ ::= ⟨Sentencia⟩ | ⟨Sentencia⟩ ⟨Sentencias⟩ | λ

⟨Argumentos-Formales⟩::= (⟨Lista-Argumentos-Formales⟩ **?**)
⟨Argumentos-Formales⟩::= (⟨Lista-Argumentos-Formales-opt⟩)
⟨Lista-Argumentos-Formales-opt⟩::= ⟨Lista-Argumentos-Formales⟩ | λ

⟨Sentencia⟩ ::= ;
| ⟨Asignación⟩;
| ⟨Sentencia-Simple⟩;
| if (⟨Expresión⟩) ⟨Sentencia⟩
| if (⟨Expresión⟩) ⟨Sentencia⟩ else ⟨Sentencia⟩
| while (⟨Expresión⟩) ⟨Sentencia⟩
| ⟨Bloque⟩
| ret ⟨Expresion⟩ **?**;
⟨Sentencia⟩ ::= ;
| ⟨Asignación⟩;
| ⟨Sentencia-Simple⟩;
| if (⟨Expresión⟩) ⟨Sentencia⟩
| if (⟨Expresión⟩) ⟨Sentencia⟩ else ⟨Sentencia⟩
| while (⟨Expresión⟩) ⟨Sentencia⟩
| ⟨Bloque⟩
| ret ⟨Expresion-opt⟩ ;
⟨Expresion-opt⟩::= ⟨Expresion⟩ | λ

⟨Bloque⟩ ::= { ⟨Sentencia⟩ ∗ }
⟨Bloque⟩ ::= { ⟨Sentencias⟩ }

⟨AccesoVar-Simple⟩ ::= id ⟨Encadenado-Simple⟩ ∗ | id [⟨Expresion⟩]
⟨AccesoVar-Simple⟩ ::= id ⟨EncadenadosSimples⟩ | id [⟨Expresion⟩]
⟨EncadenadosSimples⟩ ::= ⟨Encadenado-Simple⟩
| ⟨Encadenado-Simple⟩ ⟨EncadenadosSimples⟩ | λ

⟨AccesoSelf-Simple⟩ ::= self ⟨Encadenado-Simple⟩ ∗
⟨AccesoSelf-Simple⟩ ::= self ⟨*EncadenadosSimples*⟩

⟨Operando⟩ ::= ⟨Literal⟩ | ⟨Primario⟩ ⟨Encadenado⟩ **?**
⟨Operando⟩ ::= ⟨Literal⟩ | ⟨Primario⟩ ⟨Encadenado-opt⟩
⟨Encadenado-opt⟩::= ⟨Encadenado⟩ | λ

⟨ExpresionParentizada⟩ ::= (⟨Expresion⟩) ⟨Encadenado⟩ **?**
⟨ExpresionParentizada⟩ ::= (⟨Expresion⟩) ⟨Encadenado-opt⟩

⟨AccesoSelf ⟩ ::= self ⟨Encadenado⟩ **?**
⟨AccesoSelf ⟩ ::= self ⟨Encadenado-opt⟩

⟨AccesoVar ⟩ ::= id ⟨Encadenado⟩ **?** | id [⟨Expresión⟩]⟨Encadenado⟩ **?**
⟨AccesoVar ⟩ ::= id ⟨Encadenado-opt⟩ | id [⟨Expresión⟩
]⟨Encadenado-ot⟩

⟨Llamada-Método⟩ ::= id ⟨Argumentos-Actuales⟩ ⟨Encadenado⟩ **?**
⟨Llamada-Método⟩ ::= id ⟨Argumentos-Actuales⟩ ⟨Encadenado-opt⟩

⟨Llamada-Método-Estático⟩ ::= idStruct . ⟨Llamada-Método⟩ ⟨Encadenado⟩ **?**
⟨Llamada-Método-Estático⟩ ::= idStruct . ⟨Llamada-Método⟩
⟨Encadenado-opt⟩

⟨Llamada-Constructor ⟩ ::= new idStruct ⟨Argumentos-Actuales⟩ ⟨Encadenado⟩
? | new ⟨Tipo-Primitivo⟩ [⟨Expresion⟩]
⟨Llamada-Constructor ⟩ ::= new idStruct ⟨Argumentos-Actuales⟩
⟨Encadenado-opt⟩ | new ⟨Tipo-Primitivo⟩ [⟨Expresion⟩]

⟨Argumentos-Actuales⟩ ::= (⟨Lista-Expresiones⟩ **?**)
⟨Argumentos-Actuales⟩ ::= ⟨Lista-Expresiones-opt⟩
⟨Lista-Expresiones-opt⟩::= ⟨Lista-Expresiones⟩ | λ

⟨Llamada-Método-Encadenado⟩ ::= id ⟨Argumentos-Actuales⟩ ⟨Encadenado⟩ **?**
⟨Llamada-Método-Encadenado⟩ ::= id ⟨Argumentos-Actuales⟩
⟨Encadenado-opt⟩

⟨Acceso-Variable-Encadenado⟩ ::= id ⟨Encadenado⟩ **?** | id [⟨Expresion⟩] ⟨Encadenado⟩ **?**
⟨Acceso-Variable-Encadenado⟩ ::= id ⟨Encadenado-opt⟩ | id [⟨Expresion⟩]
⟨Encadenado-opt⟩

3.2.1.2 Eliminación de símbolos improductivos y no accesibles:

Para eliminar la recursividad, primero se deben eliminar los símbolos improductivos, que son aquellos que no pueden generar ninguna cadena de terminales, y tambien los símbolos no accesibles, que son aquellos que no se pueden alcanzar desde el símbolo inicial mediante ninguna secuencia de producciones.

Para el caso de nuestra gramatica, todos los símbolos son productivos lo que significa que todos tienen la capacidad de derivar en alguna cadena de terminales. Tampoco se identificaron símbolos inaccesibles, todos los símbolos presentes en la gramática tienen una ruta para ser alcanzados desde el símbolo inicial ⟨program⟩. Por lo tanto, no se requiere eliminar simbolos.

3.2.1.3 Eliminación de producciones lambda:

Una vez que se ha limpiado la gramática de símbolos improductivos y no accesibles, se procede a eliminar las producciones lambda, es decir, aquellas que permiten que un no terminal derive en la cadena vacía (λ).

Esto se realiza eliminando las producciones de la forma A → λ, luego: para cada regla de la forma R → uAv se agrega una nueva regla de producción R → uv, para cada regla de producción de la forma R → uAvAw agrega: R → uAvw, R → uvAw y R → uvw y, finalmente, si existe alguna regla de producción de la forma R → A, se agrega la producción R → λ.

Al realizar este proceso obtuvimos lo siguiente. Siguiendo la representacion usada anteriormente, en rojo se muestra lo que se elimina (reglas con lambda λ) y en verde lo que se modifica o agrega.

⟨*program*⟩ ::= ⟨*Lista-Definiciones*⟩ ⟨*Start*⟩ | ⟨*Start*⟩
⟨*Lista-Definiciones*⟩ ::= ⟨*Definiciones*⟩ **| λ**

⟨*Struct*⟩ ::= “**struct**” **idStruct** ⟨*Herencia-opt*⟩ “**{**“ ⟨*Atributos*⟩ ”**}**”
| “**struct**” **idStruct** “**{**“ ⟨*Atributos*⟩ ”**}**”
| “**struct**” **idStruct** ⟨*Herencia-opt*⟩ “**{**“ ”**}**”
| “**struct**” **idStruct** “**{**“ ”**}**”
⟨*Atributos*⟩ ::= ⟨*Atributo*⟩ | ⟨*Atributo*⟩ ⟨*Atributos*⟩ **| λ**
⟨*Herencia-opt*⟩ ::= ⟨*Herencia*⟩ **| λ**

⟨*Atributo*⟩ ::= ⟨*Visibilidad-opt*⟩ ⟨*Tipo*⟩ ⟨*Lista-Declaracion-Variables*⟩ “**;**”
| ⟨*Tipo*⟩ ⟨*Lista-Declaracion-Variables*⟩ “**;**”
⟨*Visibilidad-opt*⟩ ::= ⟨*Visibilidad*⟩ **| λ**

⟨*Método*⟩ ::= ⟨*Forma-Método-opt*⟩ “**fn**” **idMetAt** ⟨*Argumentos-Formales*⟩ “**->**” ⟨*Tipo-Método*⟩

⟨*Bloque-Método*⟩ | "**fn**" **idMetAt** ⟨*Argumentos-Formales*⟩ "**->**" ⟨*Tipo-Método*⟩ ⟨*Bloque-Método*⟩
⟨*Forma-Método-opt*⟩ ::= ⟨*Forma-Método*⟩ | **λ**

⟨*Bloque-Método*⟩ ::= "**{**" ⟨*Declaraciones*⟩ ⟨*Sentencias*⟩ "**}**"
| "**{**" ⟨*Sentencias*⟩ "**}**"
| "**{**" ⟨*Declaraciones*⟩ "**}**"
| "**{**" "**}**"
⟨*Declaraciones*⟩ ::= ⟨*Decl-Var-Locales*⟩ | ⟨*Decl-Var-Locales*⟩ ⟨*Declaraciones*⟩ |
λ ⟨*Sentencias*⟩ ::= ⟨*Sentencia*⟩ | ⟨*Sentencia*⟩ ⟨*Sentencias*⟩ | **λ**

⟨*Argumentos-Formales*⟩::= "**(**" ⟨*Lista-Argumentos-Formales-opt*⟩ "**)**"
| "(" ")"
⟨*Lista-Argumentos-Formales-opt*⟩::= ⟨*Lista-Argumentos-Formales*⟩ | **λ**

⟨*Sentencia*⟩ ::= "**;**"
| ⟨*Asignación*⟩"**;**"
| ⟨*Sentencia-Simple*⟩"**;**"
| "**if**" "**(**"⟨*Expresión*⟩"**)**" ⟨*Sentencia*⟩
| "**if**" "**(**"⟨*Expresión*⟩"**)**" ⟨*Sentencia*⟩ "**else**" ⟨*Sentencia*⟩
| "**while**" "**(**" ⟨*Expresión*⟩ "**)**" ⟨*Sentencia*⟩
| ⟨Bloque⟩
| "**ret**" ⟨Expresion-opt⟩ "**;**"
| "**ret**" ";"
⟨*Expresion-opt*⟩::= ⟨*Expresion*⟩ | **λ**

⟨*AccesoVar-Simple*⟩ ::= **id** ⟨*EncadenadosSimples*⟩ | **id** "**[**" ⟨*Expresion*⟩ "**]**" | **id**

⟨*EncadenadosSimples*⟩ ::= ⟨*Encadenado-Simple*⟩
| ⟨*Encadenado-Simple*⟩ ⟨*EncadenadosSimples*⟩ | **λ**
⟨*AccesoSelf-Simple*⟩ ::= "**self**" ⟨*EncadenadosSimples*⟩ | "**self**"

⟨*Operando*⟩ ::= ⟨*Literal*⟩ | ⟨*Primario*⟩ ⟨*Encadenado-opt*⟩ | ⟨*Primario*⟩
⟨*Encadenado-opt*⟩::= ⟨*Encadenado*⟩ | **λ**
⟨*ExpresionParentizada*⟩ ::= "**(**" ⟨*Expresion*⟩ "**)**" ⟨*Encadenado-opt*⟩

⟨*AccesoSelf* ⟩ ::= "**self**" ⟨*Encadenado-opt*⟩ | "**self**"
⟨AccesoVar⟩ ::= **id** ⟨*Encadenado-opt*⟩
| **id**
| **id** "**[**" ⟨*Expresión*⟩ "**]**"⟨*Encadenado-opt*⟩
| **id** "**[**" ⟨*Expresión*⟩ "**]**"
⟨*Llamada-Método*⟩ ::= **id** ⟨*Argumentos-Actuales*⟩ ⟨*Encadenado-opt*⟩
| **id** ⟨*Argumentos-Actuales*⟩
⟨*Llamada-Método-Estático*⟩ ::= **idStruct** "**.**" ⟨*Llamada-Método*⟩ ⟨*Encadenado-opt*⟩

| **idStruct** "." ⟨*Llamada-Método*⟩

⟨*Llamada-Constructor* ⟩ ::= "**new**" idStruct⟨*Argumentos-Actuales*⟩ ⟨*Encadenado-opt*⟩

| "**new**" **idStruct** ⟨*Argumentos-Actuales*⟩

| "**new**" ⟨*Tipo-Primitivo*⟩ "**[**" ⟨E*xpresion*⟩ "**]**"

⟨*Argumentos-Actuales*⟩ ::= "**(**" ⟨Lista-Expresiones-opt⟩ "**)**" | "**(**" "**)**"

⟨*Lista-Expresiones-opt*⟩ ::= ⟨*Lista-Expresiones*⟩ **| λ**

3.2.1.4 Eliminación de producciones simples:

Ahora, elimina las producciones simples o unarias, que son aquellas en las que un no terminal deriva directamente en otro no terminal sin consumir ningún terminal en el proceso.

Esto se realiza siguiendo un algoritmo que, en resumen, para cada símbolo no terminal B en la gramática tal que exista una producción de la forma A → B: Para cada producción de la forma B → α, se agrega la producción de la forma A → α y se elimina la producción de la forma B → α.

Siguiendo la representacion usada, en rojo se muestran las reglas de producciones simples que se eliminan y en verde lo que se modifica o agrega en la gramatica.

Por la regla: ⟨*Lista-Definiciones*⟩ ::= ⟨*Definiciones*⟩ , modificamos:

- ⟨program⟩ ::= ~~⟨Lista-Definiciones⟩~~ ⟨*Definiciones*⟩ ⟨Start⟩ | ⟨Start⟩

Por: ⟨*Herencia-opt*⟩ ::= ⟨*Herencia*⟩ , modificamos:

- ⟨*Struct*⟩ ::= "**struct**" **idStruct** ~~⟨*Herencia-opt*⟩~~ ⟨*Herencia*⟩ "**{**" ⟨*Atributos*⟩ "**}**"
  | "**struct**" **idStruct** "**{**" ⟨*Atributos*⟩ "**}**"
  | "**struct**" **idStruct** ~~⟨*Herencia-opt*⟩~~ ⟨*Herencia*⟩ "**{**" "**}**"
  | "**struct**" **idStruct** "**{**" "**}**"

Por: ⟨*Visibilidad-opt*⟩ ::= ⟨*Visibilidad*⟩ , modificamos:

- ⟨*Atributo*⟩ ::= ⟨~~*Visibilidad-opt*~~⟩ ⟨*Visibilidad*⟩ ⟨*Tipo*⟩ ⟨*Lista-Declaracion-Variables*⟩ "**;**"

Por: ⟨*Forma-Método-opt*⟩ ::= ⟨*Forma-Método*⟩, modificamos:

- ⟨*Método*⟩ ::= ~~⟨*Forma-Método-opt*⟩~~ ⟨*Forma-Método-opt*⟩ "**fn**" **idMetAt** …

Por: ⟨*Lista-Argumentos-Formales-opt*⟩::= ⟨*Lista-Argumentos-Formales*⟩, modificamos:

- ⟨*Argumentos-Formales*⟩::= "**(**" ~~⟨*Lista-Argumentos-Formales-opt*⟩~~ ⟨*Lista-Argumentos-Formales*⟩ "**)**" | "**(**" "**)**"

Por: ⟨*Expresion-opt*⟩::= ⟨*Expresion*⟩, modificamos:

- ⟨*Sentencia*⟩ ::= "**;**"
  | ⟨*Asignación*⟩"**;**"
  | ⟨*Sentencia-Simple*⟩"**;**"

| “**if**” “**(**“⟨*Expresión*⟩”**)**” ⟨*Sentencia*⟩
| “**if**” “**(**“⟨*Expresión*⟩”**)**” ⟨*Sentencia*⟩ “**else**” ⟨*Sentencia*⟩
| “**while**” “**(**“ ⟨*Expresión*⟩ “**)**” ⟨*Sentencia*⟩
| ⟨Bloque⟩
| “**ret**” ~~⟨Expresion-opt⟩~~ ⟨Expresion⟩ “**;**”
| “**ret**” “**;**”

Por: ⟨*Encadenado-opt*⟩::= ⟨*Encadenado*⟩ , modificamos:

- ⟨*Operando*⟩ ::= ⟨*Literal*⟩ | ⟨*Primario*⟩ ~~⟨*Encadenado-opt*⟩~~ ⟨*Encadenado*⟩ | ⟨*Primario*⟩

- ⟨*ExpresionParentizada*⟩ ::= “**(**“ ⟨*Expresion*⟩ “**)**” ~~⟨*Encadenado-opt*⟩~~ ⟨*Encadenado*⟩ | “**(**“ ⟨*Expresion*⟩ “**)**”

- ⟨*AccesoSelf* ⟩ ::= “**self**” ~~⟨*Encadenado-opt*⟩~~ ⟨*Encadenado*⟩ | “**self**”

- ⟨AccesoVar⟩ ::= **id** ~~⟨*Encadenado-opt*⟩~~ ⟨*Encadenado*⟩
  | **id**
  | **id** “**[**“ ⟨*Expresión*⟩ “**]**“ ~~⟨*Encadenado-opt*⟩~~ ⟨*Encadenado*⟩
  | **id** “**[**“ ⟨*Expresión*⟩ “**]**“

- ⟨*Llamada-Método*⟩ ::= **id** ⟨*Argumentos-Actuales*⟩ ~~⟨*Encadenado-opt*⟩~~ ⟨*Encadenado*⟩ | **id** ⟨*Argumentos-Actuales*⟩

- ⟨*Llamada-Método-Estático*⟩ ::= **idStruct** “**.**” ⟨*Llamada-Método*⟩ ~~⟨*Encadenado-opt*⟩~~ ⟨*Encadenado*⟩

- ⟨*Llamada-Constructor* ⟩ ::= “**new**” **idStruct** ⟨*Argumentos-Actuales*⟩ ~~⟨*Encadenado-opt*⟩~~ ⟨*Encadenado*⟩

- ⟨*Llamada-Método-Encadenado*⟩::= **id** ⟨*Argumentos-Actuales*⟩ ~~⟨*Encadenado-opt*⟩~~ ⟨*Encadenado*⟩ | **id** ⟨*Argumentos-Actuales*⟩

- ⟨*Acceso-Variable-Encadenado*⟩ ::= **id** ~~⟨*Encadenado-opt*⟩~~ ⟨*Encadenado*⟩ | **id**
  | **id** “**[**“ ⟨*Expresion*⟩ “**]**” ~~⟨*Encadenado-opt*⟩~~ ⟨*Encadenado*⟩
  | **id** “**[**“ ⟨*Expresion*⟩ “**]**”

Por: ⟨*Lista-Expresiones-opt*⟩ ::= ⟨*Lista-Expresiones*⟩, modificamos:

- ⟨*Argumentos-Actuales*⟩ ::= “**(**“ ~~⟨Lista-Expresiones-opt⟩~~ ⟨Lista-Expresiones⟩ “**)**” | “**(**“ “**)**”

3.2.1.5 Eliminación de recursividad por izquierda:

Una vez que se completan los pasos anteriores, que son para limpiar la gramática, se

aborda la eliminación de la recursividad por la izquierda. Esto se realiza eliminando la recursidad por la izquierda inmediata y no inmediata. A continuacion, en rojo se muestran las reglas de producciones con recursividad a izquierda y en verde su correspondiente modificación.

⟨*Expresion* ⟩ ::= ⟨*Expresion* ⟩ "**||**" ⟨*ExpAnd*⟩ | ⟨*ExpAnd*⟩
⟨*Expresion*⟩ ::= ⟨*ExpAnd*⟩ ⟨*Expresion*1⟩
⟨*Expresion1* ⟩ ::= "**||**" ⟨*ExpAnd*⟩ ⟨*Expresion*1⟩ | λ

⟨*ExpAnd*⟩ ::= ⟨*ExpAnd*⟩ "**&&**" ⟨*ExpIgual*⟩ | ⟨*ExpIgual*⟩
⟨*ExpAnd*⟩ ::= ⟨*ExpIgual*⟩ ⟨*ExpAnd1*⟩
⟨*ExpAnd1*⟩ ::= "**&&**" ⟨*ExpIgual*⟩ ⟨*ExpAnd1*⟩ | λ

⟨*ExpIgual*⟩ ::= ⟨*ExpIgual*⟩ ⟨*OpIgual*⟩ ⟨*ExpCompuesta*⟩ |
⟨*ExpCompuesta*⟩ ⟨*ExpIgual*⟩ ::= ⟨*ExpCompuesta*⟩ ⟨*ExpIgual1*⟩
⟨*ExpIgual*1⟩ ::= ⟨*OpIgual*⟩ ⟨*ExpCompuesta*⟩ ⟨*ExpIgual1*⟩ | λ

⟨*ExpAd*⟩ ::= ⟨*ExpAd*⟩ ⟨*OpAd*⟩ ⟨*ExpMul*⟩ | ⟨*ExpMul*⟩
⟨*ExpAd*⟩ ::= ⟨*ExpMul*⟩ ⟨*ExpAd1*⟩
⟨*ExpAd1*⟩ ::= ⟨*OpAd*⟩ ⟨*ExpMul*⟩ ⟨*ExpAd1*⟩ | λ

⟨*ExpMul*⟩ ::= ⟨*ExpMul*⟩ ⟨*OpMul*⟩ ⟨*ExpUn*⟩ | ⟨*ExpUn*⟩
⟨*ExpMul*⟩ ::= ⟨*ExpUn*⟩ ⟨*ExpMul1*⟩
⟨*ExpMul1*⟩ ::= ⟨OpMul⟩ ⟨*ExpUn*⟩ ⟨*ExpMul1*⟩ | λ

⟨*Lista-Expresiones*⟩ ::= ⟨Expresión⟩ | ⟨Expresión⟩ "**,**" ⟨*Lista-Expresiones*⟩ ⟨*Lista-Expresiones*⟩
::= ⟨*ExpAnd*⟩ ⟨*Expresion*1⟩ | ⟨*ExpAnd*⟩⟨*Expresion*1⟩ "**,**" ⟨*Lista-Expresiones*⟩

3.2.1.6 Factorización:

Finalmente realizamos la factorización por la izquierda que es una transformación gramatical útil para producir la gramática adecuada para el análisis descendente predictivo. Cuando la elección entre dos producciones cuyo lado derecho es idéntico no está claro podemos reescribir dichas producciones para diferir la decisión hasta haber visto la suficiente entrada como para poder realizar la elección correcta. A continuacion, en rojo se muestran las reglas de producciones que consideramos que se deben factorizar y en verde su correspondiente factorización.

⟨Definiciones⟩ ::= ⟨Struct⟩ ⟨Definiciones1⟩
| ⟨Impl⟩ ⟨Definiciones⟩
| ⟨Struct⟩
| ⟨Impl⟩
⟨Definiciones⟩ ::= ⟨Struct⟩ ⟨Definiciones 1⟩ | ⟨Impl⟩ ⟨Definiciones1⟩
⟨Definiciones1⟩ ::= ⟨Definiciones⟩ | λ

⟨*Struct*⟩ ::= "**struct**" **idStruct** ⟨*Herencia*⟩ "**{**" ⟨*Atributos*⟩ "**}**"
| "**struct**" **idStruct** "**{**" ⟨*Atributos*⟩ "**}**"
| "**struct**" **idStruct** ⟨*Herencia*⟩ "**{**" "**}**"

| “**struct**” **idStruct** “**{**“ ”**}**”
⟨*Struct*⟩ ::= “**struct**” **idStruct** ⟨*Struct1*⟩
⟨*Struct1*⟩ ::= ⟨*Herencia*⟩ “**{**“ ⟨*Struct2*⟩
| “**{**“ ⟨*Struct2*⟩
⟨*Struct2*⟩ ::= ⟨*Atributos*⟩ ”**}**” | ”**}**”

⟨*Atributos*⟩ ::= ⟨*Atributo*⟩ | ⟨*Atributo*⟩ ⟨*Atributos*⟩
⟨*Atributos*⟩ ::= ⟨*Atributo*⟩ ⟨*Atributos1*⟩
⟨*Atributos1*⟩ ::= λ | ⟨*Atributos*⟩

⟨*Miembros*⟩ ::= ⟨*Miembro*⟩ | ⟨*Miembro*⟩ ⟨*Miembros* ⟩
⟨*Miembros*⟩ ::= ⟨*Miembro*⟩ ⟨*Miembros1*⟩
⟨*Miembros1*⟩ ::= λ | ⟨*Miembros*⟩

⟨*Bloque-Método*⟩ ::= “**{**“ ⟨*Declaraciones*⟩ ⟨*Sentencias*⟩ “**}**”
| “**{**“⟨*Sentencias*⟩ “**}**”
| “**{**“ ⟨*Declaraciones*⟩ “**}**
| “**{**“ “**}**”
⟨*Bloque-Método*⟩ ::= “**{**“ ⟨*Bloque-Método1*⟩
⟨*Bloque-Método1*⟩ ::= ⟨*Declaraciones*⟩ ⟨*Bloque-Método2*⟩
| ⟨*Sentencias*⟩ “**}**”
| “**}**”
⟨*Bloque-Método2*⟩ ::= ⟨*Sentencias*⟩ “**}**” | “**}**”

⟨*Declaraciones*⟩ ::= ⟨*Decl-Var-Locales*⟩ **|** ⟨*Decl-Var-Locales*⟩
⟨*Declaraciones*⟩ ⟨*Declaraciones*⟩ ::= ⟨*Decl-Var-Locales*⟩ ⟨*Declaraciones1*⟩
⟨*Declaraciones*⟩ ::= λ **|** ⟨*Declaraciones*⟩

⟨*Sentencias*⟩ ::= ⟨*Sentencia*⟩ **|** ⟨*Sentencia*⟩ ⟨*Sentencias*⟩
⟨*Sentencias*⟩ ::= ⟨*Sentencia*⟩ ⟨*Sentencias1*⟩
⟨*Sentencias1*⟩ ::= λ **|** ⟨*Sentencias*⟩

⟨*Lista-Declaracion-Variables*⟩::= **idMetAt**
| **idMetAt** “**,**” ⟨*Lista-Declaracion-Variables*⟩
⟨*Lista-Declaracion-Variables*⟩::= **idMetAt**
⟨*Lista-Declaracion-Variables1*⟩ ⟨*Lista-Declaracion-Variables1*⟩::= λ **|** “**,**”
⟨*Lista-Declaracion-Variables*⟩

⟨*Argumentos-Formales*⟩::= “**(**“ ⟨*Lista-Argumentos-Formales*⟩ “**)**”
| “**(**“ “**)**”
⟨*Argumentos-Formales*⟩::= “**(**“ ⟨*Argumentos-Formales1*⟩
⟨*Argumentos-Formales1*⟩::= ⟨*Lista-Argumentos-Formales*⟩ “**)**” **|** “**)**”

⟨*Lista-Argumentos-Formales*⟩ ::= ⟨*Argumento-Formal*⟩ “**,**”
⟨*Lista-Argumentos-Formales*⟩ | ⟨*Argumento-Formal*⟩
⟨*Lista-Argumentos-Formales*⟩ ::= ⟨*Argumento-Formal*⟩
⟨*Lista-Argumentos-Formales1*⟩ ⟨*Lista-Argumentos-Formales1*⟩ ::= “**,**”
⟨*Lista-Argumentos-Formales*⟩ | λ

⟨*Sentencia*⟩ ::= “**;**”
| ⟨*Asignación*⟩”**;**”
| ⟨*Sentencia-Simple*⟩”**;**”
| “**if**” “**(**“⟨*Expresión*⟩”**)**” ⟨*Sentencia*⟩
| “**if**” “**(**“⟨*Expresión*⟩”**)**” ⟨*Sentencia*⟩ “**else**” ⟨*Sentencia*⟩
| “**while**” “**(**“ ⟨*Expresión*⟩ “**)**” ⟨*Sentencia*⟩
| ⟨Bloque⟩
| “**ret**” ⟨Expresion⟩ “**;**”
| “**ret**” “**;**”
⟨*Sentencia*⟩ ::= “**;**”
| ⟨*Asignación*⟩”**;**”
| ⟨*Sentencia-Simple*⟩”**;**”
| “**if**” “**(**“⟨*Expresión*⟩”**)**” ⟨*Sentencia*⟩ ⟨*Sentencia1*⟩
| “**while**” “**(**“ ⟨*Expresión*⟩ “**)**” ⟨*Sentencia*⟩
| ⟨Bloque⟩
| “**ret**” ⟨*Sentencia2*⟩
⟨*Sentencia1*⟩ ::= λ | “**else**” ⟨*Sentencia*⟩
⟨*Sentencia2*⟩ ::= ⟨Expresion⟩ “**;**” | “**;**”

⟨*Bloque*⟩ ::= “**{**“ ⟨*Sentencias*⟩ “**}**” | “**{**“ “**}**”
⟨*Bloque*⟩ ::= “**{**“ ⟨*Bloque1*⟩
⟨*Bloque1*⟩ ::= ⟨*Sentencias*⟩ “**}**” | “**}**”

⟨*AccesoVar-Simple*⟩ ::= **id** ⟨*EncadenadosSimples*⟩
| **id**
| **id** “**[**“ ⟨*Expresion*⟩ “**]**”
⟨*AccesoVar-Simple*⟩ ::= **id** ⟨*AccesoVar-Simple1*⟩
⟨*AccesoVar-Simple1*⟩ ::= ⟨*EncadenadosSimples*⟩
| λ
| “**[**“ ⟨*Expresion*⟩ “**]**”

⟨*EncadenadosSimples*⟩ ::= ⟨*Encadenado-Simple*⟩
| ⟨*Encadenado-Simple*⟩ ⟨*EncadenadosSimples*⟩
⟨*EncadenadosSimples*⟩ ::= ⟨*Encadenado-Simple*⟩
⟨*EncadenadosSimples1*⟩ ⟨*EncadenadosSimples1*⟩ ::= λ |
⟨*EncadenadosSimples*⟩

⟨*AccesoSelf-Simple*⟩ ::= “**self**” ⟨*EncadenadosSimples*⟩ | “**self**”

⟨*AccesoSelf-Simple*⟩ ::= "**self**" ⟨*AccesoSelf-Simple1*⟩
⟨*AccesoSelf-Simple1*⟩ ::= ⟨*EncadenadosSimples*⟩ | λ

⟨*ExpCompuesta*⟩ ::= ⟨*ExpAd*⟩ ⟨*OpCompuesto*⟩ ⟨*ExpAd*⟩ | ⟨*ExpAd*⟩
⟨*ExpCompuesta*⟩ ::= ⟨*ExpAd*⟩ ⟨*ExpCompuesta1*⟩
⟨*ExpCompuesta1*⟩ ::= ⟨*OpCompuesto*⟩ ⟨*ExpAd*⟩ | λ

⟨*Operando*⟩ ::= ⟨*Literal*⟩ | ⟨*Primario*⟩ ⟨*Encadenado*⟩ | ⟨*Primario*⟩
⟨*Operando*⟩ ::= ⟨*Literal*⟩ | ⟨*Primario*⟩ ⟨*Operando1*⟩
⟨*Operando1*⟩ ::= ⟨*Encadenado*⟩ | λ

⟨*ExpresionParentizada*⟩ ::= "**(**" ⟨*Expresion*⟩ "**)**" ⟨*Encadenado*⟩
| "**(**" ⟨*Expresion*⟩ "**)**"
⟨*ExpresionParentizada*⟩ ::= "**(**" ⟨*Expresion*⟩ "**)**"
⟨*ExpresionParentizada1*⟩ | "**(**" ⟨*Expresion*⟩ "**)**"
⟨*ExpresionParentizada1*⟩ ::= ⟨*Encadenado*⟩ | λ

⟨*AccesoSelf* ⟩ ::= "**self**" ⟨*Encadenado*⟩ | "**self**"
⟨*AccesoSelf* ⟩ ::= "**self**" ⟨*AccesoSelf*1⟩
⟨*AccesoSelf*1⟩ ::= ⟨*Encadenado*⟩ | λ

⟨AccesoVar⟩ ::= **id** ⟨*Encadenado*⟩
| **id**
| **id** "**[**" ⟨*Expresión*⟩ "**]**"⟨*Encadenado*⟩
| **id** "**[**" ⟨*Expresión*⟩ "**]**"
⟨AccesoVar⟩ ::= **id** ⟨AccesoVar1⟩
⟨AccesoVar1⟩ ::= ⟨*Encadenado*⟩
| λ
| "**[**" ⟨*Expresión*⟩ "**]**"⟨AccesoVar2⟩
⟨AccesoVar2⟩ ::= ⟨*Encadenado*⟩ | λ

⟨*Llamada-Método*⟩ ::= **id** ⟨*Argumentos-Actuales*⟩ ⟨*Encadenado*⟩
| **id** ⟨*Argumentos-Actuales*⟩
⟨*Llamada-Método*⟩ ::= **id** ⟨*Argumentos-Actuales*⟩ ⟨*Llamada-Método1*⟩
⟨*Llamada-Método1*⟩ ::= ⟨*Encadenado*⟩ | λ

⟨*Llamada-Método-Estático*⟩ ::= **idStruct** "**.**" ⟨*Llamada-Método*⟩
⟨*Encadenado*⟩ | **idStruct** "**.**" ⟨*Llamada-Método*⟩

⟨*Llamada-Método-Estático*⟩ ::= **idStruct** "**.**" ⟨*Llamada-Método*⟩

⟨*Llamada-Método-Estático1*⟩ ⟨*Llamada-Método-Estático1*⟩ ::= ⟨*Encadenado*⟩ | λ

⟨*Llamada-Constructor* ⟩ ::= "**new**" **idStruct** ⟨*Argumentos-Actuales*⟩
⟨*Encadenado*⟩ | "**new**" **idStruct** ⟨*Argumentos-Actuales*⟩ | "**new**" ⟨*Tipo-Primitivo*⟩
"**[**" ⟨E*xpresion*⟩ "**]**"

⟨*Llamada-Constructor* ⟩ ::= "**new**" ⟨*Llamada-Constructor*1⟩
⟨*Llamada-Constructor*1⟩ ::= **idStruct** ⟨*Argumentos-Actuales*⟩
⟨*Llamada-Constructor2*⟩ | ⟨*Tipo-Primitivo*⟩ "**[**" ⟨E*xpresion*⟩ "**]**"
⟨*Llamada-Constructor2*⟩ ::= ⟨*Encadenado*⟩ | λ

⟨*Argumentos-Actuales*⟩ ::= "**(**" ⟨Lista-Expresiones⟩ "**)**" | "**(**" "**)**"
⟨*Argumentos-Actuales*⟩ ::= "**(**" ⟨*Argumentos-Actuales*⟩
⟨*Argumentos-Actuales1*⟩ ::= ⟨Lista-Expresiones⟩ "**)**" | "**)**"

⟨*Encadenado*⟩ ::= "**.**" ⟨*Llamada-Método-Encadenado*⟩
| "**.**" ⟨*Acceso-Variable-Encadenado*⟩
⟨*Encadenado*⟩ ::= "**.**" ⟨*Encadenado1*⟩
⟨*Encadenado*⟩ ::= ⟨*Llamada-Método-Encadenado*⟩
| ⟨*Acceso-Variable-Encadenado*⟩

⟨*Llamada-Método-Encadenado*⟩::= **id** ⟨*Argumentos-Actuales*⟩
⟨*Encadenado*⟩ | **id** ⟨*Argumentos-Actuales*⟩
⟨*Llamada-Método-Encadenado*⟩::= **id** ⟨*Argumentos-Actuales*⟩
⟨*Llamada-Método-Encadenado1*⟩
⟨*Llamada-Método-Encadenado1*⟩ ::= ⟨*Encadenado*⟩ | λ

⟨*Acceso-Variable-Encadenado*⟩ ::= **id** ⟨*Encadenado*⟩
| **id**
| **id** "**[**" ⟨*Expresion*⟩ "**]**" ⟨*Encadenado*⟩
| **id** "**[**" ⟨*Expresion*⟩ "**]**"
⟨*Acceso-Variable-Encadenado*⟩ ::=**id**
⟨*Acceso-Variable-Encadenado1*⟩ ⟨*Acceso-Variable-Encadenado1*⟩
::= ⟨*Encadenado*⟩ | λ | "**[**" ⟨*Expresion*⟩ "**]**"
⟨*Acceso-Variable-Encadenado2*⟩
⟨*Acceso-Variable-Encadenado2*⟩ ::= ⟨*Encadenado*⟩ | λ

⟨*Lista-Expresiones*⟩ ::= ⟨*ExpAnd*⟩ ⟨*Expresion1*⟩ | ⟨*ExpAnd*⟩⟨*Expresion1*⟩ "**,**" ⟨*Lista-Expresiones*⟩

⟨*Lista-Expresiones*⟩ ::= ⟨*ExpAnd*⟩ ⟨*Expresion1*⟩ ⟨*Lista-Expresiones1*⟩

⟨*Lista-Expresiones1*⟩ ::= λ | "**,**" ⟨*Lista-Expresiones*⟩

3.2.1.7 Gramática Final:

Nuestra gramatica sin recursión a izquierda y factorizada queda como sigue:

⟨*S*⟩ ::= ⟨*program*⟩$
⟨*program*⟩ ::= ⟨*Definiciones*⟩ ⟨*Start*⟩ | ⟨*Start*⟩
⟨*Start*⟩ ::= "**start**" ⟨*Bloque-Método*⟩
⟨Definiciones⟩ ::= ⟨Struct⟩ ⟨Definiciones 1⟩ | ⟨Impl⟩ ⟨Definiciones1⟩
⟨Definiciones1⟩ ::= ⟨Definiciones⟩ | λ
⟨*Struct*⟩ ::= "**struct**" **idStruct** ⟨*Struct1*⟩
⟨*Struct1*⟩ ::= ⟨*Herencia*⟩ "**{**" ⟨*Struct2*⟩
| "**{**" ⟨*Struct2*⟩
⟨*Struct2*⟩ ::= ⟨*Atributos*⟩ "**}**" | "**}**"
⟨*Atributos*⟩ ::= ⟨*Atributo*⟩ ⟨*Atributos1*⟩
⟨*Atributos1*⟩ ::= ⟨*Atributos*⟩ | λ
⟨*Impl*⟩ ::= "**impl**" **idStruct** "**{**" ⟨*Miembros*⟩ "**}**"
⟨*Miembros*⟩ ::= ⟨*Miembro*⟩ ⟨*Miembros1*⟩
⟨*Miembros1*⟩ ::= ⟨*Miembros*⟩ | λ
⟨*Herencia*⟩ ::= "**:**" ⟨*Tipo*⟩
⟨*Miembro*⟩ ::= ⟨*Metodo*⟩ | ⟨*Constructor*⟩
⟨*Constructor* ⟩ ::= "**.**" ⟨*Argumentos-Formales*⟩ ⟨*Bloque-Método*⟩
⟨*Atributo*⟩ ::= ⟨*Visibilidad*⟩ ⟨*Tipo*⟩ ⟨*Lista-Declaracion-Variables*⟩ "**;**"
| ⟨*Tipo*⟩ ⟨*Lista-Declaracion-Variables*⟩ "**;**"
⟨*Método*⟩ ::= ⟨*Forma-Método*⟩ "**fn**" **idMetAt** ⟨*Argumentos-Formales*⟩ "**->**"
⟨*Tipo-Método*⟩ ⟨*Bloque-Método*⟩
| "**fn**" **idMetAt** ⟨*Argumentos-Formales*⟩ "**->**" ⟨*Tipo-Método*⟩
⟨*Bloque-Método*⟩ ⟨*Visibilidad*⟩ ::= "**pri**"
⟨*Forma-Método*⟩ ::= "**st**"
⟨*Bloque-Método*⟩ ::= "**{**" ⟨*Bloque-Método1*⟩
⟨*Bloque-Método1*⟩ ::= ⟨*Declaraciones*⟩ ⟨*Bloque-Método2*⟩
| ⟨*Sentencias*⟩ "**}**"
| "**}**"
⟨*Bloque-Método2*⟩ ::= ⟨*Sentencias*⟩ "**}**" | "**}**"
⟨*Declaraciones*⟩ ::= ⟨*Decl-Var-Locales*⟩ ⟨*Declaraciones1*⟩
⟨*Declaraciones1*⟩ ::= ⟨*Declaraciones*⟩ | λ
⟨*Sentencias*⟩ ::= ⟨*Sentencia*⟩ ⟨*Sentencias1*⟩
⟨*Sentencias1*⟩ ::= ⟨*Sentencias*⟩ | λ
⟨*Decl-Var-Locales*⟩ ::= ⟨*Tipo*⟩ ⟨*Lista-Declaracion-Variables*⟩ "**;**"
⟨*Lista-Declaracion-Variables*⟩::= **idMetAt**
⟨*Lista-Declaracion-Variables1*⟩ ⟨*Lista-Declaracion-Variables1*⟩::= "**,**"
⟨*Lista-Declaracion-Variables*⟩ | λ
⟨*Argumentos-Formales*⟩::= "**(**" ⟨*Argumentos-Formales1*⟩

⟨*Argumentos-Formales1*⟩::= ⟨*Lista-Argumentos-Formales*⟩ “**)**” | “**)**”
⟨*Lista-Argumentos-Formales*⟩ ::= ⟨*Argumento-Formal*⟩
⟨*Lista-Argumentos-Formales1*⟩ ⟨*Lista-Argumentos-Formales1*⟩ ::= “**,**”
⟨*Lista-Argumentos-Formales*⟩ | λ ⟨*Argumento-Formal*⟩ ::= ⟨*Tipo*⟩ **idMetAt**
⟨*Tipo-Método*⟩ ::= ⟨*Tipo*⟩ | “**void**”
⟨Tipo⟩ ::= ⟨*Tipo-Primitivo*⟩ | ⟨*Tipo-Referencia*⟩ | ⟨*Tipo-Arreglo*⟩
⟨*Tipo-Primitivo*⟩ ::= “**Str**” | “**Bool**” | “**Int**” | “**Char**”
⟨*Tipo-Referencia*⟩ ::= **idStruct**
⟨*Tipo-Arreglo*⟩ ::= “**Array**” ⟨*Tipo-Primitivo*⟩
⟨*Sentencia*⟩ ::= “**;**”
| ⟨*Asignación*⟩”**;**”
| ⟨*Sentencia-Simple*⟩”**;**”
| “**if**” “**(**“⟨*Expresión*⟩”**)**” ⟨*Sentencia*⟩ ⟨*Sentencia1*⟩
| “**while**” “**(**“ ⟨*Expresión*⟩ “**)**” ⟨*Sentencia*⟩
| ⟨Bloque⟩
| “**ret**” ⟨*Sentencia2*⟩
⟨*Sentencia1*⟩ ::= “**else**” ⟨*Sentencia*⟩ | λ
⟨*Sentencia2*⟩ ::= ⟨Expresion⟩ “**;**” | “**;**”
⟨*Bloque*⟩ ::= “**{**“ ⟨*Bloque1*⟩
⟨*Bloque1*⟩ ::= ⟨*Sentencias*⟩ “**}**” | “**}**”
⟨*Asignacion*⟩ ::= ⟨*AccesoVar-Simple*⟩ “**=**” ⟨*Expresion*⟩
| ⟨*AccesoSelf-Simple*⟩ “**=**” ⟨*Expresion*⟩
⟨*AccesoVar-Simple*⟩ ::= **id** ⟨*AccesoVar-Simple1*⟩
⟨*AccesoVar-Simple1*⟩ ::= ⟨*EncadenadosSimples*⟩ | “**[**“ ⟨*Expresion*⟩ “**]**” | λ
⟨*EncadenadosSimples*⟩ ::= ⟨*Encadenado-Simple*⟩
⟨*EncadenadosSimples1*⟩ ⟨*EncadenadosSimples1*⟩ ::=
⟨*EncadenadosSimples*⟩ | λ
⟨*AccesoSelf-Simple*⟩ ::= “**self**” ⟨*AccesoSelf-Simple1*⟩
⟨*AccesoSelf-Simple1*⟩ ::= ⟨*EncadenadosSimples*⟩ | λ
⟨*Encadenado-Simple*⟩ ::= “**.**” **id**
⟨*Sentencia-Simple*⟩ ::= “**(**“⟨*Expresion*⟩”**)**”
⟨*Expresion*⟩ ::= ⟨*ExpAnd*⟩ ⟨*Expresion*1⟩
⟨*Expresion1* ⟩ ::= “**||**” ⟨*ExpAnd*⟩ ⟨*Expresion*1⟩ | λ
⟨*ExpAnd*⟩ ::= ⟨*ExpIgual*⟩ ⟨*ExpAnd1*⟩
⟨*ExpAnd1*⟩ ::= “**&&**” ⟨*ExpIgual*⟩ ⟨*ExpAnd1*⟩ | λ
⟨*ExpIgual*⟩ ::= ⟨*ExpCompuesta*⟩ ⟨*ExpIgual1*⟩
⟨*ExpIgual*1⟩ ::= ⟨*OpIgual*⟩ ⟨*ExpCompuesta*⟩ ⟨*ExpIgual1*⟩ | λ
⟨*ExpCompuesta*⟩ ::= ⟨*ExpAd*⟩ ⟨*ExpCompuesta1*⟩
⟨*ExpCompuesta1*⟩ ::= ⟨*OpCompuesto*⟩ ⟨*ExpAd*⟩ | λ
⟨*ExpAd*⟩ ::= ⟨*ExpMul*⟩ ⟨*ExpAd1*⟩
⟨*ExpAd1*⟩ ::= ⟨*OpAd*⟩ ⟨*ExpMul*⟩ ⟨*ExpAd1*⟩ | λ
⟨*ExpMul*⟩ ::= ⟨*ExpUn*⟩ ⟨*ExpMul1*⟩
⟨*ExpMul1*⟩ ::= ⟨OpMul⟩ ⟨*ExpUn*⟩ ⟨*ExpMul1*⟩ | λ
⟨*ExpUn*⟩ ::= ⟨*OpUnario*⟩ ⟨*ExpUn*⟩ | ⟨*Operando*⟩
⟨OpIgual⟩ ::= “**==**” | “**!=**”
⟨*OpCompuesto*⟩ ::= “**<**” | “**>**” | “**<=**” | “**>=**”
⟨*OpAd*⟩ ::= “**+**” | “**-**”
⟨*OpUnario*⟩ ::= “**+**” | “**-**” | “**!**” | “**++**” | “**--**”

⟨*OpMul*⟩ ::= “*****” | “**/**” | “**%**”
⟨*Operando*⟩ ::= ⟨*Literal*⟩ | ⟨*Primario*⟩ ⟨*Operando1*⟩
⟨*Operando1*⟩ ::= ⟨*Encadenado*⟩ | λ
⟨*Literal*⟩ ::= “**nil**” | “**true**” | “**false**” | **intLiteral** | **StrLiteral** | **charLiteral**
⟨*Primario*⟩ ::= ⟨*ExpresionParentizada*⟩
| ⟨*AccesoSelf* ⟩
| ⟨*AccesoVar* ⟩
| ⟨*Llamada-Método*⟩
| ⟨*Llamada-Metodo-Estatico*⟩
| ⟨*Llamada-Constructor* ⟩
⟨*ExpresionParentizada*⟩ ::= “**(**“⟨*Expresion*⟩“**)**”
⟨*ExpresionParentizada1*⟩
⟨*ExpresionParentizada1*⟩ ::= ⟨*Encadenado*⟩ | λ
⟨*AccesoSelf* ⟩ ::= “**self**” ⟨*AccesoSelf*1⟩
⟨*AccesoSelf*1⟩ ::= ⟨*Encadenado*⟩ | λ
⟨AccesoVar⟩ ::= **id** ⟨AccesoVar1⟩
⟨AccesoVar1⟩ ::= ⟨*Encadenado*⟩ | “**[**“ ⟨*Expresión*⟩ “**]**“⟨AccesoVar2⟩ | λ
⟨AccesoVar2⟩ ::= ⟨*Encadenado*⟩ | λ
⟨*Llamada-Método*⟩ ::= **id** ⟨*Argumentos-Actuales*⟩ ⟨*Llamada-Método1*⟩
⟨*Llamada-Método1*⟩ ::= ⟨*Encadenado*⟩ | λ
⟨*Llamada-Método-Estático*⟩ ::= **idStruct** “**.**” ⟨*Llamada-Método*⟩
⟨*Llamada-Método-Estático1*⟩ ⟨*Llamada-Método-Estático1*⟩ ::= ⟨*Encadenado*⟩ | λ
⟨*Llamada-Constructor* ⟩ ::= “**new**” ⟨*Llamada-Constructor*1⟩
⟨*Llamada-Constructor*1⟩ ::= **idStruct** ⟨*Argumentos-Actuales*⟩
⟨*Llamada-Constructor2*⟩ | ⟨*Tipo-Primitivo*⟩ “**[**“ ⟨*Expresion*⟩ “**]**”
⟨*Llamada-Constructor2*⟩ ::= ⟨*Encadenado*⟩ | λ
⟨*Argumentos-Actuales*⟩ ::= “**(**“ ⟨*Argumentos-Actuales1*⟩
⟨*Argumentos-Actuales1*⟩ ::= ⟨Lista-Expresiones⟩ ”**)**” | ”**)**”
⟨*Lista-Expresiones*⟩ ::= ⟨*ExpAnd*⟩⟨*Expresion*1⟩ ⟨*Lista-Expresiones1*⟩
⟨*Lista-Expresiones1*⟩ ::= λ | “**,**” ⟨*Lista-Expresiones*⟩
⟨*Encadenado*⟩ ::= “**.**” ⟨*Encadenado1*⟩
⟨*Encadenado1*⟩ ::= ⟨*Llamada-Método-Encadenado*⟩
| ⟨*Acceso-Variable-Encadenado*⟩
⟨*Llamada-Método-Encadenado*⟩::= **id** ⟨*Argumentos-Actuales*⟩
⟨*Llamada-Método-Encadenado1*⟩
⟨*Llamada-Método-Encadenado1*⟩ ::= ⟨*Encadenado*⟩ | λ
⟨*Acceso-Variable-Encadenado*⟩::= **id**
⟨*Acceso-Variable-Encadenado1*⟩
⟨*Acceso-Variable-Encadenado1*⟩ ::= ⟨*Encadenado*⟩ | “**[**“
⟨*Expresion*⟩“**]**” ⟨*Acceso-Variable-Encadenado2*⟩ | λ
⟨*Acceso-Variable-Encadenado2*⟩ ::= ⟨*Encadenado*⟩ | λ

3.2.1.8 Primeros y Siguientes:

Calculamos los conjuntos de primeros y siguientes de la gramática, ya que son herramientas utilizadas en el análisis sintáctico.

Primeros (FIRST): Los primeros de un símbolo no terminal en una gramática representan el conjunto de todos los terminales que pueden comenzar una cadena derivada de ese símbolo no terminal. A continuacion, se muestra el conjunto primeros para cada no terminal de nuestra gramatica:

- P(⟨*S*⟩) = P(⟨*program*⟩) = {struct, impl, start}
- P(⟨*program*⟩) = P(⟨*Definiciones*⟩) U P(⟨*Start*⟩) = {struct, impl, start}
- P(⟨*Start*⟩) = {start}
- P(⟨Definiciones⟩) = P(⟨Struct⟩) U P(⟨Impl⟩) = {struct, impl}
- P(⟨Definiciones1⟩) = P(⟨Definiciones⟩) U λ = {struct, impl, λ}
- P(⟨*Struct*⟩) = {struct}
- P(⟨*Struct*1⟩) = P(⟨*Herencia*⟩) U "{" = { :, { }
- P(⟨*Struct2*⟩) = P(⟨*Atributos*⟩) U "}" = {pri, Str, Bool, Int, Char, IdStruc, Array, } } - P(⟨*Atributos*⟩) = P(⟨*Atributo*⟩) = {pri, Str, Bool, Int, Char, IdStruc, Array} - P(⟨*Atributos1*⟩) = P(⟨*Atributos*⟩) U λ = {pri, Str, Bool, Int, Char, IdStruc, Array, λ } - P(⟨*Impl*⟩) = {impl}
- P(⟨*Miembros*⟩) = P(⟨*Miembro*⟩)= {st, fn, .}
- P(⟨*Miembros1*⟩)= P(⟨*Miembro*⟩) U λ = {st, fn, ., λ}
- P(⟨*Herencia*⟩) = { : }
- P(⟨*Miembro*⟩) = P(⟨*Metodo*⟩) U P(⟨*Constructor*⟩)= {st, fn, .}
- P(⟨*Constructor* ⟩) = { . }
- P(⟨*Atributo*⟩) = P(⟨*Visibilidad*⟩) U P(⟨*Tipo*⟩)= {pri, Str, Bool, Int, Char, IdStruc, Array} - P(⟨*Método*⟩)= P(⟨*Forma-Metodo*⟩) U "fn" = {st, fn}
- P(⟨*Visibilidad*⟩)= { pri }
- P(⟨*Forma-Método*⟩)= { st }
- P(⟨*Bloque-Método*⟩)= { { }
- P(⟨*Bloque-Método1*⟩) = P(⟨*Declaraciones*⟩) U P(⟨*Sentencias*⟩) U } = {Str ,Bool, Int, Char, IdStruc, Array, ; , if, while, ret, id, self, (, {, } }
- P(⟨*Bloque-Método2*⟩) = P(⟨*Sentencias*⟩) U } = { ; , if, while, ret, id, self, (, {, } }
- P(⟨*Declaraciones*⟩) = P(⟨*Decl-Var-Locales*⟩) = {Str ,Bool, Int, Char, IdStruc, Array}
- P(⟨*Declaraciones1*⟩) =P(⟨*Declaraciones*⟩) U λ ={Str ,Bool, Int, Char, IdStruc, Array, λ }
- P(⟨*Sentencias*⟩)= P(⟨*Sentencia*⟩) = { ; , if, while, ret, id, self, (, { }
- P(⟨*Sentencias1*⟩) = P(⟨*Sentencias*⟩) + λ = { ; , if, while, ret, id, self, (, {, λ }
- P(⟨*Decl-Var-Locales*⟩) = P(⟨*Tipo*⟩) = {Str ,Bool, Int, Char, IdStruc, Array}
- P(⟨*Lista-Declaracion-Variables*⟩) = {idMetAt}
- P(⟨*Lista-Declaracion-Variables1*⟩)= { , , λ }
- P(⟨*Argumentos-Formales*⟩) = { (}
- P(⟨*Argumentos-Formales1*⟩) = P(⟨*Lista-Argumentos-Formales*⟩) U) = {Str ,Bool, Int, Char, IdStruc, Array,)}
- P(⟨*Lista-Argumentos-Formales*⟩) = ⟨*Argumento-Formal*⟩ = {Str ,Bool, Int, Char, IdStruc, Array}
- P(⟨*Lista-Argumentos-Formales1*⟩) = { , , λ}
- P(⟨*Argumento-Formal*⟩) = P(⟨*Tipo*⟩)= {Str ,Bool, Int, Char, IdStruc, Array} - P(⟨*Tipo-Método*⟩) = P(⟨*Tipo*⟩) U void = {Str ,Bool, Int, Char, IdStruc, Array, void}
- P(⟨*Tipo*⟩) = P(⟨*Tipo-Primitivo*⟩) U P(⟨*Tipo-Referencia*⟩) U P(⟨*Tipo-Arreglo*⟩) = {Str ,Bool, Int, Char, IdStruc, Array}

- P(⟨*Tipo-Primitivo*⟩)= {Str ,Bool, Int, Char }
- P(⟨*Tipo-Referencia*⟩) = {IdStruc}
- P(⟨*Tipo-Arreglo*⟩) = {Array}
- P(⟨*Sentencia*⟩) = { ; , if, while, ret } U P(⟨*Asignación*⟩) U P(*Sentencia-Simple*⟩) U P(⟨*Bloque*⟩)= { ; , if, while, ret, id, self, (, { }
- P(⟨*Sentencia1*⟩) = {else , λ}
- P(⟨*Sentencia2*⟩) = { ; } U P(⟨Expresion⟩) = {+, -, !, ++, --, nil, true, false, intLiteral, StrLiteral, charLiteral, (, self, id, idStruct, new, ; }
- P(⟨*Bloque*⟩) = { { }
- P(⟨*Bloque1*⟩) = { } } U P(⟨*Sentencias*⟩)= { ; , if, while, ret, id, self, (, {, }} - P(⟨*Asignacion*⟩) = P(⟨*AccesoVar-Simple*⟩) U P(⟨*AccesoSelf-Simple*⟩)= { id, self } - P(⟨*AccesoVar-Simple*⟩) = { id }
- P(⟨*AccesoVar-Simple1*⟩) = P(⟨*EncadenadosSimples*⟩) U {λ , [} = { ., λ, [}
- P(⟨*EncadenadosSimples*⟩) = P(⟨*Encadenado-Simple*⟩) = { . }
- P(⟨*EncadenadosSimples1*⟩) = P(⟨*EncadenadosSimples*⟩) U {λ}= { ., λ }
- P(⟨*AccesoSelf-Simple*⟩) = {self}
- P(⟨AccesoSelf-Simple1⟩) = P(⟨EncadenadosSimples⟩) U {λ} = { ., λ }
- P(⟨*Encadenado-Simple*⟩) = { . }
- P(⟨*Sentencia-Simple*⟩)= { (}
- P(⟨*Expresion*⟩) = P(⟨*ExpAnd*⟩) = {+, -, !, ++, --, nil, true, false, intLiteral, StrLiteral, charLiteral, (, self, id, idStruct, new }
- P(⟨*Expresion1*⟩)= {||, λ}
- P(⟨*ExpAnd*⟩) = P(⟨*ExpIgual*⟩) = {+, -, !, ++, --, nil, true, false, intLiteral, StrLiteral, charLiteral, (, self, id, idStruct, new }
- P(⟨*ExpAnd1*⟩) = { &&, λ }
- P(⟨*ExpIgual*⟩) = P(⟨*ExpCompuesta*⟩) = {+, -, !, ++, --, nil, true, false, intLiteral, StrLiteral, charLiteral, (, self, id, idStruct, new }
- P(⟨*ExpIgual*1⟩) = P(⟨*OpIgual*⟩) U λ = {==, !=, λ}
- P(⟨*ExpCompuesta*⟩) = P(⟨*ExpAd*⟩) = {+, -, !, ++, --, nil, true, false, intLiteral, StrLiteral, charLiteral, (, self, id, idStruct, new }
- P(⟨*ExpCompuesta1*⟩) = P(⟨*OpCompuesto*⟩) U λ = {<, >, <=, >=, λ }
- P(⟨*ExpAd*⟩) = P(⟨*ExpMul*⟩) = {+, -, !, ++, --, nil, true, false, intLiteral, StrLiteral, charLiteral, (, self, id, idStruct, new }
- P(⟨*ExpAd1*⟩) = P(⟨*OpAd*⟩) U λ = { + ,- , λ }
- P(⟨*ExpMul*⟩) = P(⟨*ExpUn*⟩) = {+, -, !, ++, --, nil, true, false, intLiteral, StrLiteral, charLiteral, (, self, id, idStruct, new }
- P(⟨*ExpMul1*⟩) = P(⟨OpMul⟩) U λ = {*, /, %, λ}
- P(⟨*ExpUn*⟩) = P(⟨*OpUnario*⟩) U P(⟨*Operando*⟩) = {+, -, !, ++, --, nil, true, false, intLiteral, StrLiteral, charLiteral, (, self, id, idStruct, new }
- P(⟨OpIgual⟩) = {==, !=}
- P(⟨*OpCompuesto*⟩) = {<, >, <=, >= }
- P(⟨*OpAd*⟩) = { +, -}
- P(⟨*OpUnario*⟩) = {+, -, !, ++, --}
- P(⟨*OpMul*⟩) = {*, /, %}
- P(⟨*Operando*⟩) = P(⟨*Literal*⟩) U P(⟨*Primario*⟩) = { nil, true, false, intLiteral, StrLiteral, charLiteral, (, self, id, idStruct, new }
- P(⟨*Operando1*⟩) = P(⟨*Encadenado*⟩) + λ = { ., λ }
- P(⟨⟨*Literal*⟩) = { nil, true, false, intLiteral, StrLiteral, charLiteral }

- P(⟨*Primario*⟩) = P(⟨*ExpresionParentizada*⟩) U P(⟨*AccesoSelf*⟩) U P(⟨*AccesoVar*⟩) + P(⟨*Llamada-Método*⟩) U P(⟨*Llamada-Metodo-Estatico*⟩) U P(⟨*Llamada-Constructor*⟩) = { (, self, id, idStruct, new }
- P(⟨*ExpresionParentizada*⟩) = { (}
- P(⟨*ExpresionParentizada1*⟩) = P(⟨*Encadenado*⟩) U λ = { ., λ }
- P(⟨*AccesoSelf* ⟩) = { self }
- P(⟨*AccesoSelf*1⟩) = P(⟨*Encadenado*⟩) U λ = { ., λ }
- P(⟨AccesoVar⟩) = { id }
- P(⟨AccesoVar1⟩) = P(⟨*Encadenado*⟩) U "[" U λ = { ., [, λ }
- P(⟨AccesoVar2⟩) = P(⟨*Encadenado*⟩) U λ = { ., λ }
- P(⟨*Llamada-Método*⟩) = { id }
- P(⟨*Llamada-Método1*⟩) = P(⟨*Encadenado*⟩) U λ = { ., λ }
- P(*Llamada-Método-Estático*) = { idStruct }
- P(*Llamada-Método-Estático1*) = P(⟨*Encadenado*⟩) U λ = { ., λ }
- P(*Llamada-Constructor*) = { new }
- P(*Llamada-Constructor1*) = "idStruct" U P(⟨*Tipo-Primitivo*⟩) = {Str ,Bool , Int, Char, idStruct}
- P(*Llamada-Constructor2*) = P(⟨*Encadenado*⟩) U λ = { ., λ }
- P(⟨*Argumentos-Actuales*⟩) = { (}
- P(⟨*Argumentos-Actuales1*⟩) = P(⟨Lista-Expresiones⟩) U ")" = {+, -, !, ++, --, nil, true, false, intLiteral, StrLiteral, charLiteral, (,), self, id, idStruct, new }
- P(⟨*Lista-Expresiones*⟩) = P(⟨Expresión⟩) U P(⟨*ExpAnd*⟩) = {+, -, !, ++, --, nil, true, false, intLiteral, StrLiteral, charLiteral, (, self, id, idStruct, new }
- P(⟨*Encadenado*⟩) = { . }
- P(⟨*Encadenado1*⟩) = P(⟨*Llamada-Método-Encadenado*⟩) U P(⟨*Acceso-Variable-Encadenado*⟩) = { id }
- P(⟨*Llamada-Método-Encadenado*⟩) = { id }
- P(⟨*Llamada-Método-Encadenado1*⟩) = P(⟨*Encadenado*⟩) U λ = { ., λ }
- P(⟨*Acceso-Variable-Encadenado*⟩) = { id }
- P(⟨*Acceso-Variable-Encadenado1*⟩) = P(⟨*Encadenado*⟩) U "[" + λ = { ., [, λ }
- P(⟨*Acceso-Variable-Encadenado2*⟩) = P(⟨*Encadenado*⟩) U λ = { ., λ }

Siguientes (FOLLOW): Los siguientes de un símbolo no terminal en una gramática representan el conjunto de todos los terminales que pueden seguir al símbolo no terminal en alguna derivación de la cadena. A continuación, se muestra el conjunto siguientes para cada no terminal de nuestra gramatica:

- S(⟨*program*⟩) = {$}
- S(⟨*Start*⟩)= S(⟨*program*⟩) = {$}
- S(⟨Definiciones⟩) = P(⟨*Start*⟩) U S(⟨Definiciones1⟩) = { start }
- S(⟨Definiciones1⟩) = S⟨Definiciones⟩) = { start }
- S(⟨*Struct*⟩)= P⟨Definiciones1⟩) U S(⟨Definiciones⟩)= {start, struct, impl, λ}
- S(⟨*Struct1*⟩)= S(⟨*Struct*⟩)= {start, struct, impl, λ}
- S(⟨*Struct2*⟩)=S(⟨*Struct1*⟩)= {start, struct, impl, λ}
- S(⟨*Atributos*⟩)= } U S(⟨*Atributos1*⟩)= { } }
- S(⟨*Atributos1*⟩) = S(⟨*Atributos*⟩)= { } }
- S(⟨*Impl*⟩) = P⟨Definiciones1⟩) U S(⟨Definiciones⟩)= {start, struct, impl, λ}
- S(⟨*Miembros*⟩)= "}" U S(⟨*Miembros1*⟩)= { } }

- S(⟨*Miembros1*⟩)= S(⟨*Miembros*⟩)= { } }
- S(⟨*Herencia*⟩)= { { }
- S(⟨*Miembro*⟩)= P(⟨*Miembros1*⟩) U S(⟨*Miembros*⟩) = {st, fn, ., λ, } }
- S(⟨*Constructor*⟩)=S(⟨*Miembro*⟩)= {st, fn, ., λ, } }
- S(⟨*Atributo*⟩)= P(⟨*Atributos1*⟩) U S(⟨*Atributos*⟩)= {pri, Str, Bool, Int, Char, IdStruc, Array, λ,} }
- S(⟨*Metodo*⟩)=S(⟨*Miembro*⟩)= {st, fn, ., λ, } }
- S(⟨*Visibilidad*⟩)= P(⟨*Tipo*⟩)= {Str ,Bool, Int, Char, IdStruc, Array}
- S(⟨*Forma-Método*⟩)= { fn }
- S(⟨*Bloque-Método*⟩)=S(⟨*Start*⟩) U S(⟨*Constructor*⟩) u S(⟨*Metodo*⟩)={$, st, fn, ., λ, } }
- S(⟨*Bloque-Método1*⟩)=S(⟨*Bloque-Método*⟩)= {$, st, fn, ., λ, } }
- S(⟨*Bloque-Método2*⟩)=S(⟨*Bloque-Método1*⟩)= {$, st, fn, ., λ, } }
- S(⟨*Declaraciones*⟩) = { ; , if, while, ret, id, self, (, {, } }
- S(⟨*Declaraciones1*⟩) = { ; , if, while, ret, id, self, (, {, } }
- S(⟨*Sentencias1*⟩) = S(⟨*Sentencias*⟩)= { } }
- S(⟨*Sentencias*⟩) = { } }
- S(⟨*Decl-Var-Locales*⟩)= P(⟨*Declaraciones1*⟩) U S(⟨*Declaraciones*⟩) = {Str ,Bool, Int, Char, IdStruc, Array, λ , ; , if, while, ret, id, self, (, {, } }
- S(⟨*Lista-Declaracion-Variables*⟩) = { ; }
- S(⟨*Lista-Declaracion-Variables1*⟩) = S(⟨*Lista-Declaracion-Variables*⟩) = { ; }
- S(⟨*Argumentos-Formales*⟩)= P(⟨*Bloque-Metodo*⟩) U "->" = { { , ->}
- S(⟨*Argumentos-Formales1*⟩)= S(⟨*Argumentos-Formales*⟩)= { { , -> }
- S(⟨*Lista-Argumentos-Formales1*⟩) = {) }
- S(⟨*Lista-Argumentos-Formales*⟩) = {) }
- S(⟨*Argumento-Formal*⟩)= P(⟨*Lista-Argumentos-Formales1*⟩) U S(⟨*Lista-Argumentos-Formales*⟩)= { , , λ ,)}
- S(⟨*Tipo-Método*⟩)= P(⟨*Bloque-Método*⟩)= { { }
- S(⟨*Tipo*⟩)=S(⟨*Herencia*⟩) U P(⟨*Lista-Declaracion-Variables*⟩) U "idMetAt" U S(⟨*Tipo-Método*⟩)= { {, idMetAt }
- S(⟨*Tipo-Primitivo*⟩) = S(⟨*Tipo*⟩) U S(⟨*Tipo-Arreglo*⟩) U "[" = { {, idMetAt, [} - S(⟨*Tipo-Referencia*⟩) = S(⟨*Tipo*⟩)={ {, idMetAt }
- S(⟨*Tipo-Arreglo*⟩) = S(⟨*Tipo*⟩)={ {, idMetAt }
- S(⟨*Sentencia*⟩)= P(⟨*Sentencias1*⟩) U S(⟨*Sentencia1*⟩)={ ; , if, while, ret, id, self, (, {, λ }
- S(⟨*Sentencia1*⟩)= S(⟨*Sentencia*⟩)={ ; , if, while, ret, id, self, (, {, λ } - S(⟨*Sentencia2*⟩) = S(⟨*Sentencia*⟩) = { ; , if, while, ret, id, self, (, {, λ } - S(⟨*Bloque*⟩) = S(⟨*Sentencia*⟩) = { ; , if, while, ret, id, self, (, {, λ }
- S(⟨*Bloque1*⟩) = S(⟨*Bloque*⟩) = { ; , if, while, ret, id, self, (, {, λ }
- S(⟨*Asignacion*⟩)= { ; }
- S(⟨*AccesoVar-Simple*⟩) = { = }
- S(⟨*AccesoVar-Simple1*⟩) = S(⟨*AccesoVar-Simple*⟩) = { = }
- S(⟨*EncadenadosSimples*⟩)=S(⟨*AccesoVar-Simple1*⟩) U S⟨*AccesoSelf-Simple1*⟩= { = }
- S(⟨*EncadenadosSimples1*⟩)=S(⟨*EncadenadosSimples*⟩)={ = }
- S(⟨*AccesoSelf-Simple*⟩)={ = }
- S(⟨*AccesoSelf-Simple1*⟩) = S(⟨*AccesoSelf-Simple*⟩) = { = }
- S(⟨*Encadenado-Simple*⟩) = P(⟨*EncadenadosSimples1*⟩) U S(⟨*EncadenadosSimples*⟩) = { ., λ ,=}
- S(⟨*Sentencia-Simple*⟩) = { ; }

- S(⟨*Expresion*⟩)= {) ;]} U S(⟨*Asignacion*⟩) U S(⟨*Lista-Expresiones*⟩) ={), ;,]}
- S(⟨*Expresion1* ⟩)= S(⟨*Expresion*⟩) U {,}= {) , ; ,] , ,}
- S(⟨*ExpAnd*⟩)= P(⟨*Expresion*1⟩) U S(⟨*Expresion*⟩) U S(⟨*Expresion1*⟩) U S(⟨*Lista-Expresiones*)= {||,), ;,] , ,}
- S(⟨*ExpAnd1*⟩)= S(⟨*ExpAnd*⟩) = {||,), ;,] , ,}
- S(⟨*ExpIgual*⟩)=P(⟨*ExpAnd1*⟩) U S(⟨*ExpAnd*⟩) U S(⟨*ExpAnd1*⟩) = {) , ; ,] , ,, ||, &&}
- S(⟨*ExpIgual*1⟩)=S(⟨*ExpIgual*⟩) = {) , ; ,] , ,,||,&&,λ}
- S(⟨*ExpCompuesta*⟩)= P(⟨*ExpIgual1*⟩) U S(⟨*ExpIgual*⟩) U S(⟨*ExpIgual*1⟩)= {==, !=, λ,) , ; ,] , ,, ||, &&}
- S(⟨*ExpCompuesta1*⟩)=S(⟨*ExpCompuesta*⟩)= {==, !=, λ,) , ; ,] , ,, ||, &&} - S(⟨*ExpAd*⟩)= P(⟨*ExpCompuesta1*⟩) U S(⟨*ExpCompuesta1*⟩) U S(⟨*ExpCompuesta*⟩)= {==, !=, λ,) , ; ,] , ,, ||, &&, <, >, <=, >=, λ }
- S(⟨*ExpAd1*⟩) = S(⟨*ExpAd*⟩) = {) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ } - S(⟨*ExpMul1*⟩)=S(⟨*ExpMul*⟩)={+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ } - S(⟨*ExpMul*⟩)= P(⟨*ExpAd1*⟩) U S(⟨*ExpAd1*⟩) U S(⟨*ExpAd*⟩)={+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S(⟨*ExpUn*⟩)=P(⟨*ExpMul1*⟩)= {*, /, %, λ} + S(⟨*ExpMul*⟩)+ S(⟨*ExpMul1*⟩)={*, /, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S(⟨*Operando*⟩) = S(⟨*ExpUn*⟩) ={*, /, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S(⟨*Operando1*⟩) =S(⟨*Operando*⟩)={*, /, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S(⟨*Literal*⟩) =S(⟨*Operando*⟩) = {*, /, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ } - S(⟨*Primario*⟩) = P(⟨*Operando1*⟩) U S(⟨*Operando*⟩)={*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S(⟨*ExpresionParentizada*⟩) = S(⟨*Primario*⟩) = {*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S(⟨*ExpresionParentizada1*⟩) = S(⟨*ExpresionParentizada*⟩) = {*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S(⟨*AccesoSelf* ⟩) = S(⟨*Primario*⟩) = {*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S(⟨*AccesoSelf*1⟩) = S(⟨*AccesoSelf*⟩) = {*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S(⟨AccesoVar⟩) = S(⟨*Primario*⟩) = {*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S(⟨AccesoVar1⟩) = S(⟨AccesoVar⟩) = {*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S(⟨AccesoVar2⟩) = S(⟨AccesoVar1⟩) = {*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S(⟨Llamada-Método⟩) = S(⟨*Llamada-Método-Estático*⟩) U S(⟨*Primario*⟩) = {*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=,λ}
- S(⟨Llamada-Método1⟩) = S(⟨Llamada-Método⟩) = {*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=,λ}
- S(⟨*Llamada-Método-Estático*⟩)=S(⟨*Primario*⟩) = {*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=,λ}
- S⟨*Llamada-Método-Estático1*⟩ = {*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=,λ} - S⟨*Llamada-Constructor* ⟩ = S(⟨*Primario*⟩) = {*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S⟨*Llamada-Constructor*1⟩ = S⟨*Llamada-Constructor* ⟩ = {*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S⟨*Llamada-Constructor2*⟩ = S⟨*Llamada-Constructor*1⟩ = {*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S⟨*Argumentos-Actuales*⟩ = P(⟨*Llamada-Método1*⟩) U S(⟨*Llamada-Método*⟩) U P(*Llamada-Constructor2*) U S⟨*Llamada-Constructor*1⟩ U S⟨*Llamada-Método-Encadenado*⟩= {*, . ,/, %,+,-,) , ; ,], ,,||,&&,==,!=,<, >, <=, >=,λ}

- S⟨*Argumentos-Actuales1*⟩ = S⟨*Argumentos-Actuales*⟩ = {*, . ,/, %,+,-,) , ; ,],
,,||,&&,==,!=,<, >, <=, >=,λ}
- S(⟨*Lista-Expresiones*) = {) }
- S(⟨*Lista-Expresiones1*) = S(⟨*Lista-Expresiones*) ={) }
- S⟨*Encadenado*⟩ = S(⟨Operando1⟩) U S(⟨ExpresionParentizada1⟩) U S(AccesoSelf1) U S(AccesoVar1) U S(AccesoVar2) U S(Llamada-Método1) U S(Llamada-Método-Estático1) U S(Llamada-Constructor2) U S(Llamada-Método-Encadenado1) U S(Acceso-Variable-Encadenado1) U S(Acceso-Variable-Encadenado2) = {*, /, %,+,-,) , ;, . ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S⟨*Encadenado1*⟩ = S⟨*Encadenado*⟩ = {*, /, %,+,-,) , ;, . ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S⟨*Llamada-Método-Encadenado*⟩ = S⟨*Encadenado1*⟩ = {*, /, %,+,-,) , ;, . ,],
,,||,&&,==,!=,<, >, <=, >=, λ }
- S⟨*Llamada-Método-Encadenado1*⟩ = S⟨*Llamada-Método-Encadenado*⟩ = {*, /, %,+,-,) ,
;, . ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S⟨*Acceso-Variable-Encadenado*⟩ = S⟨*Encadenado1*⟩ = {*, /, %,+,-,) , ;, . ,],
,,||,&&,==,!=,<, >, <=, >=, λ }
- S⟨*Acceso-Variable-Encadenado1*⟩ = S⟨*Acceso-Variable-Encadenado*⟩ = {*, /, %,+,-,) , ;,
. ,], ,,||,&&,==,!=,<, >, <=, >=, λ }
- S⟨*Acceso-Variable-Encadenado2*⟩ = S⟨*Acceso-Variable-Encadenado1*⟩ = {*, /, %,+,-,) , ;,
. ,], ,,||,&&,==,!=,<, >, <=, >=, λ }

3.2.2. Implementación del ASD Predictivo Recursivo

Para programar el analizador sintáctico descendente predictivo recursivo se sigue la teoría de simulación de la derivación a izquierda mediante una cadena de llamadas recursivas. Cada llamada simula un paso en la derivación donde cada método será un no terminal de la gramática que vamos expandiendo en la derivación. Como la gramática es LL(1) las llamadas serán unívocamente determinadas.

Se desarrolla un método por cada no terminal de la gramática. En el cuerpo del método se simula el comportamiento de las reglas de producción para ese no terminal codificando las partes derechas de las mismas. Si se encontra un no terminal se llama al método que implementa dicho no terminal. Para un terminal se 'machea' el token actual.

Siguiendo esto, para la implementación se realiza lo siguiente:

- *Definición de Métodos para los No Terminales:* Cada no terminal en la gramática se mapeará a un método en el código del analizador. Estos métodos serán responsables de avanzar en el análisis sintáctico de acuerdo con las reglas de producción asociadas a cada no terminal.

- *Predicción de la Producción a Seguir:* Se utiliza la predicción para determinar qué producción seguir en cada paso del análisis. Esto implica verificar los conjuntos de primeros y/o siguientes de los no terminales.

- *Manejo de Recursión y λ-producciones:* Al encontrar recursión o producciones lambda (λ), se ajusta la lógica para manejar correctamente estos casos. Si un no terminal puede derivar en λ, simplemente omitie la llamada al método correspondiente y se continua con el análisis.

Ejemplo de Implementación:

Considerando la gramática simplificada

A ::= Bx | yC
B ::= zD | v | λ

Aplicando el enfoque ASD Predictivo Recursivo, se implementan métodos para A y B:

```
function A():
    if (token_actual ∈ Primeros(B)):
        B()
        Match("x")
    ellse if (token_actual == "y"):
        Match("y")
        C()
    else:
        return Error

function B():
    if (token_actual ∈ Primeros(zD)):
        Match("z")
        D()
    else if (token_actual == "v"):
        Match("v")
    else if (token_actual ∈ Siguientes(B)):
        // no hacemos nada para λ-producción
    else:
        return Error

function Match(nombre):
    if (token_actual.nombre == nombre):
        token_actual = Analizador_Léxico.next_token()
    else:
        return Error
```

En este ejemplo, cada método representa un no terminal de la gramática, y las llamadas recursivas se realizan según las reglas de producción. La función `Match` se encarga de emparejar el token actual con el terminal esperado

3.2.3. Errores sintácticos detectables

Durante el proceso de compilación, después de pasar del análisis léxico sin errores, entra en juego el análisis sintáctico, donde se verifica la estructura gramatical del código fuente. Durante este proceso, es posible encontrar errores sintácticos, que son aquellos que violan las reglas de la gramática del lenguaje de programación.

En esta sección, se exponen los errores sintácticos que hemos identificado y manejado en nuestro compilador. Estos casos abarcan diversas situaciones en las que el código fuente puede contener errores sintácticos, desde estructuras gramaticales incorrectas hasta tokens mal ubicados.

Las excepciones sintácticas tendrán como salida el número de línea donde se produjo el error, número de columna y una breve descripción:

ERROR: SINTÁCTICO

| NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION:|

Estos son los que tomamos en cuenta:

- Expresiones mal formadas.
  Ejemplo: num = num1 + * num2
  | DESCRIPCION: "Se esperaba: literal. Se encontró '*' " |

- Falta de terminación de apertura de parentesis, corchetes o llave. Ejemplo: if condición)
  | DESCRIPCION: "Se esperaba '(' " |

- Falta de terminación de cierre de parentesis, corchetes, llave o punto y coma. Ejemplo: if (condición
  | DESCRIPCION: "Se esperaba ')' " |

- Declaración de variables inválida.
  Ejemplo: int x = 1;
  | DESCRIPCION: "Falta un valor de inicialización para la variable x" |

- Declaración de funciones inválida.
  Ejemplo: fn function(Int argumento){}
  | DESCRIPCION: "Se esperaba: ->. Se encontró: { " |

- Argumentos de función mal formados.
  Ejemplo: fn function(argumento) -> Int{}
  | DESCRIPCION: "Se esperaba: identificador de Tipo o ')' " |

- Declaración de clase incompleta.
  Ejemplo: struct { Int a;}
  | DESCRIPCION: "Se esperaba: struct_name." |

Esta lista cubre una amplia variedad de posibles errores sintácticos que el compilador puede encontrar durante el análisis del código fuente.

La política ante la detección de un error es abortar el análisis sintáctico. No se implementan técnicas de recuperación ante errores.

3.2.4. Casos de Pruebas

En esta sección se detallan las pruebas realizadas para evaluar el funcionamiento del analizador léxico. Cada prueba se describe con su respectivo escenario, nombre del test y la salida esperada. Dichas pruebas abarcan una variedad de situaciones para garantizar la robustez y precisión del analizador en diferentes contextos y casos de uso.

Escenario	Nombre Test	Salida esperada
program	test_error_sinTokens.Ver que si ingresas un archivo vacio sin tokens, el copilador lo detecte como error	ERROR: SINTÁCTICO. Se esperaba: start, struc, impl
Start	test_start_vacio_ok. Ver que se acepte un start vacio	CORRECTO: ANALISIS SINTACTICO
Start	test_start_tipos_ok. Ver que se acepten todos los tipos(Referencia, Primitivo, Arreglo) dentro del start	CORRECTO: ANALISIS SINTACTICO
Start	test_start_puntoycoma. Ver que se acepte un start con pto y coma	CORRECTO: ANALISIS SINTACTICO
Start	test_error_lineas_debajo_start. Verificar que se maneje correctamente el error cuando hay codigo debajo de start	ERROR: SINTÁCTICO. Se esperaba: EOF
Start	test_error_multiples_start. Verificar que se maneje correctamente el error cuando definen mas de un start.	ERROR: SINTÁCTICO. Se esperaba: EOF
Start	test_error_sin_start. Verificar que se maneje correctamente el error al omitir la funcion start.	ERROR: SINTÁCTICO. Se esperaba: start
Start	test_error_ret_start. Ver que dentro de start, luego de colocar ret, espere ';'	ERROR: SINTÁCTICO. Se esperaba: ';'
Start	test_error_pri_start. Verificar que se maneje correctamente el error al querer definir como privada una variable en start	ERROR: SINTÁCTICO.
Start	test_start_asignación_ok. Ver que se puedan definir asignaciones dentro start	CORRECTO: ANALISIS SINTACTICO
Start	test_start_bloqueVacio_ok.Ver que dentro de las llaves del start, se pueda definir otro bloque vacio	CORRECTO: ANALISIS SINTACTICO
Struct	test_struct_ok. Verificar que se reconozca y acepte la funcion struct	CORRECTO: ANALISIS SINTACTICO
Struct	test_struct_atributos_ok. Verificar que se reconozcan y acepte la definicion de atributos de distintos tipos en struct	CORRECTO: ANALISIS SINTACTICO
Struct	test_error_struct_struct. Verificar que se maneje correctamente el error al definir un struct dentro de otro	ERROR: SINTÁCTICO. Se esperaba: Identificador de Tipo, pri o '}'. Se encontró: struct.
Struct	test_error_func_struct. Verificar que se maneje correctamente el error al definir un funciones dentro de struct	ERROR: SINTÁCTICO. Se esperaba: Identificador de Tipo, pri o '}'. Se encontró: fn.
Struct	test_error_struct_sinNombre. Verificar que se maneje correctamente el error al definir un struct sin nombre	ERROR: SINTÁCTICO. Se esperaba: struct_name. Se encontró: {

Declaración	test_error_definir_en_declaracion. Verificar que haya error cuando se quiere definir una variable en la misma linea que se declara	ERROR: SINTÁCTICO. Se esperaba: ',' o ';'. Se encontró =
Parentesis	test_error_falta_). Verificar que se maneje correctamente el error al omitir ')'	ERROR: SINTÁCTICO. Se esperaba: ')'
Parentesis	test_error_falta_(. Verificar que se maneje correctamente el error al omitir '('	ERROR: SINTÁCTICO. Se esperaba: '('
Corchete	test_error_falta_]. Verificar que se maneje correctamente el error al omitir ']'	ERROR: SINTÁCTICO. Se esperaba: ']'
Corchete	test_error_falta_[. Verificar que se maneje correctamente el error al omitir '['	ERROR: SINTÁCTICO. Se esperaba: '['
Llave	test_error_falta_}. Verificar que se maneje correctamente el error al omitir '}'	ERROR: SINTÁCTICO. Se esperaba: '}'
Llave	test_error_falta_{. Verificar que se maneje correctamente el error al omitir '{'	ERROR: SINTÁCTICO. Se esperaba: '{'
Constructor	test_constructor. Verificar que dentro de impl se pueda colocar un constructor(.())	CORRECTO: ANALISIS SINTACTICO
Sentencia	test_sentencia_while. Verificar == dentro de una sentencia	ERROR: SINTACTICO. Se esperaba ==
Expresion	test_error_expresion_while. Verificar que dentro de la sentencia while, la expresion este escrita entre ()	ERROR: SINTACTICO. Se esperaba (
Op. Incremento / Comparación	test_incr_comp. Verificar que detecte bien el op de incremento y el de comparación	CORRECTO: ANALISIS SINTACTICO
Bool	test_bool.Verificar que se puedan definiri variables bool.	CORRECTO: ANALISIS SINTACTICO
Herencia	test_herencia_tipo_ok . Ver que dentro del struct se pueda aplicar herencia colocando los ":" y a continuación un tipo, en el test se prueban los 3 tipos	CORRECTO: ANALISIS SINTACTICO
Herencia	test_herencia_notok . Ver que de un msj de error cuando elimino los : de la herencia	ERROR: SINTACTICO Se esperaba: ':' o '{'. Se encontró: (Caracter encontrado)|
fibonacci	test_fibonacci . Ver que se acepte el programa fibonacci	CORRECTO: ANALISIS SINTACTICO

3.3. Analizador Semántico - Declaraciones

El chequeo de declaraciones es una etapa crucial en la compilación de lenguajes de programación, enfocándose en verificar que las declaraciones de variables y funciones sean correctas y consistentes en el código. Este proceso va más allá de la estructura gramatical del código fuente y se preocupa por su coherencia lógica.

Durante el chequeo de declaraciones, el analizador semántico examina el código para detectar posibles errores relacionados con la declaración de variables y funciones. Esto

incluye verificar que todas las variables sean declaradas antes de su uso, que las funciones sean llamadas con los parámetros correctos y que no haya duplicación en las declaraciones.

En esta sección, desarrollaremos todos los pasos realizados para construir el chequeo de declaraciones en nuestro compilador TinyRu, explicando desde el diseño implementado, gramática de atributos, programación y pruebas del mismo. Abordaremos cómo se garantizó la integridad y corrección en las declaraciones, asegurando que el código fuente sea semánticamente válido.

3.3.1. Tratamiento de Gramática

En el proceso de desarrollo de un analizador semántico, es fundamental contar con una gramática bien definida que guíe la interpretación y validación del código fuente. En este informe, se describe el proceso de transformación de una gramática de la etapa 2, es decir BNF, sin recursividad izquierda y factorizada, en una gramática de atributos. Este paso es crucial para las tareas de analizador semántico desde la capacidad de realizar análisis contextual hasta detectar errores semánticos en el código.

Las gramáticas de atributos son gramáticas libres de contexto aumentadas con atributos (datos) en los símbolos (terminales o no,de cada regla de producción) y reglas (o acciones) semánticas (funciones de atributos) asociadas a las reglas de producción. Está extiende la gramática convencional BNF mediante la incorporación de atributos asociados a los símbolos no terminales.La misma se ha construido mediante el EDT, con acciones semánticas para realizar análisis contextual y asignar información semántica a la estructura del programa.

3.3.1.1 Esquema de Traducción (EDT)

Los Árboles de Derivación Extendidos (EDTs) son una representación de la estructura jerárquica de un programa después de que ha sido analizado sintácticamente. A diferencia de los árboles de análisis sintáctico completos, los EDTs contienen fragmentos de programa incrustados dentro de los cuerpos de las reglas de producción. Estos fragmentos, son conocidos como acciones semánticas y pueden aparecer en cualquier posición dentro del cuerpo de una producción. Las acciones semánticas le asignan significado semántico a la estructura del programa y se ejecutan durante el análisis sintáctico para realizar tareas semánticas como la verificación de tipos, la construcción de tablas de símbolos y la generación de código intermedio. Mediante la combinación de gramáticas libres de contexto con acciones semánticas, los EDTs proporcionan una representación compacta y manejable de la estructura y el significado de un programa, lo que facilita su posterior procesamiento en las etapas de análisis semántico y generación de código.

A continuación, las reglas a las que aplicamos las acciones:

⟨*Start*⟩ ::= “**start**” `{EntradaStart e = new EntradaStart();}` // Aca no verificamos si ya esta start porque en el analizador sintactico ya lo hacemos ⟨*Bloque-Método*⟩

⟨*Struct*⟩ ::= “**struct**” **idStruct** `{if noEstaTs(idStruct.lex) EntradaStruct e = new EntradaStruct(idStruct.lex); else EntradaStruct e = TS.buscarStruct('idStruct'); TS.struct_actual = e}` ⟨*Struct1*⟩

⟨*Struct1*⟩ ::= ⟨*Herencia*⟩ `{String nombre_ancestro = herencia(); TS.getClaseActual().setHerencia(nombre_ancestro); TS.insertStruct(TS.getClaseActual());}` "**{**" ⟨*Struct2*⟩
| "**{**" `{TS.getClaseActual().setHerencia("Object"); TS.insertStruct(TS.getClaseActual());}` ⟨*Struct2*⟩

⟨*Impl*⟩ ::= "**impl**" **idStruct** `{if noEstaTs(idStruct.lex) EntradaStruct e = new EntradaStruct(idStruct.lex); else EntradaStruct e = TS.buscarStruct('idStruct'); TS.struct_actual = e}` "**{**" ⟨*Miembros*⟩ "**}**"

⟨*Herencia*⟩ ::= "**:**" ⟨*Tipo*⟩ `{ret currentToken.getLexema();}`

⟨*Constructor* ⟩ ::= "**.**" `{EntradaMetodo e = new EntradaMetodo(constructor) TS.metodo_actual = e}` ⟨*Argumentos-Formales*⟩ ⟨*Bloque-Método*⟩

⟨*Atributo*⟩ ::= ⟨*Visibilidad*⟩ `{String tipo = tokenActual.lex;}` ⟨*Tipo*⟩ `{EntradaAtributo e = new EntradaAtributo(tokenActual.lex, tipo, false, pos); TS.getClaseActual().insertAtributo(e)}` ⟨*Lista-Declaracion-Variables*⟩ "**;**"
| `{String tipo = tokenActual.lex;}` ⟨*Tipo*⟩ `{EntradaAtributo e = new EntradaAtributo(tokenActual.lex, tipo, true, pos); TS.getClaseActual().insertAtributo(e)}` ⟨*Lista-Declaracion-Variables*⟩ "**;**"

⟨*Método*⟩ ::= ⟨*Forma-Método*⟩ "**fn**" **idMetAt** `{EntradaMetodo e = new EntradaMetodo(tokenActual.lex,true,pos; TS.getClaseActual().insertMetodo(e); TS.metodo_actual = e}` ⟨*Argumentos-Formales*⟩ "**->**" `{String tipo_ret = tokenActual.lex; TS.getClaseActual().getMetodoActual().setRet(tipo_ret);}` ⟨*Tipo-Método*⟩ ⟨*Bloque-Método*⟩
| "**fn**" **idMetAt** `{EntradaMetodo e = new EntradaMetodo(tokenActual.lex,false,pos) TS.metodo_actual = e}` ⟨*Argumentos-Formales*⟩ "**->**" ⟨*Tipo-Método*⟩ `{String tipo_ret = tipoMetodo(); TS.getMetodoActual().setRet(tipo_ret);}` ⟨*Bloque-Método*⟩

⟨*Argumento-Formal*⟩ ::= ⟨*Tipo*⟩ `{String tipo = tipo();}` **idMetAt** `EntradaParametro e = new EntradaParametro(id.lex, tipo, pos); else error; TS.getClaseActual().getMetodoActual().insertParametro(e);}`

3.3.2. Implementación de Tabla de Símbolos

La tabla de símbolos es una estructura de datos fundamental en el análisis semántico de un compilador. Su función principal es almacenar información sobre los identificadores (variables, funciones, etc.) presentes en el programa, como sus nombres, tipos, valores, entre otros.

Entonces, la idea detrás de la tabla de símbolos es contar con un almacenamiento global al EDT donde se vaya guardando información de las entidades a medida que se va

reconociendo sintácticamente el código fuente. Considerando además que la TS pueda pasarla al módulo encargado de los controles semánticos (segunda pasada).

En este informe, se describe el diseño e implementación de la tabla de símbolos en el contexto del analizador semántico desarrollado.

3.3.2.1 Diseño de tabla de símbolos

Para representar eficazmente la tabla de símbolos en nuestra implementación, hemos optado por utilizar una estructura de tabla hash con tablas internas. Esta elección se debe a que una tabla hash proporciona un acceso rápido a los elementos mediante una función que asigna claves a índices en la tabla.

En nuestra tabla de símbolos, cada entrada está destinada a gestionar diferentes identificadores de nuestra gramática, ya que cada uno requiere almacenar información específica. Cada entidad en el código tendrá su propia entrada dentro de la tabla de símbolos, que contendrá todos los detalles asociados.

Para nuestro lenguaje orientado a objetos TinyRu, la estructura de la tabla de símbolos muestra los ámbitos de declaración. Esto significa que la tabla contiene una sección para todas las posibles clases(struct), y cada clase tendrá sus propias tablas para almacenar todas las entidades declaradas dentro de ella, como métodos, variables de clase, atributos y constructores. Del mismo modo, los métodos y constructores tendrán sus propias tablas para sus parámetros y variables locales.

La construcción y actualización de la tabla de símbolos ocurre durante el análisis sintáctico, donde cada declaración de una entidad se traduce en una actualización correspondiente en la tabla. Esto se diseñó con las acciones semánticas en el EDT construido anteriormente.

3.3.2.2 Representación de la tabla de símbolos

Representamos la tabla de símbolos generada a través de un archivo JSON con la siguiente estructura:

1. Nombre del Archivo (nombre): Indica el nombre del archivo o programa analizado:, "TS-nombreArchivo.ru"

2. Estructuras (structs): Esta es una lista que contiene información sobre todas las estructuras definidas en el programa. Cada estructura tiene los siguientes campos:

- Nombre (nombre): El nombre de la estructura.
- Hereda de (heredaDe): Indica de qué estructura hereda la actual. Si no hereda de ninguna, se asume que hereda de un objeto base (Object).
- Atributos (atributos): Una lista de los atributos definidos en la estructura, cada uno con los siguientes campos:
    - Nombre (nombre): El nombre del atributo.
    - Tipo (tipo): El tipo de dato del atributo.
    - Public (public): Indica si el atributo es público (true) o privado (false).
    - Posición (posicion): El orden en el que aparece el atributo en la estructura.
- Métodos (metodos): Una lista de los métodos definidos en la estructura, cada uno con los siguientes campos:
    - Nombre (nombre): El nombre del método.
    - Static (static): Indica si el método es estático (true) o no (false).

- Retorno (retorno): El tipo de dato que retorna el método.
- Posición (posicion): El orden en el que aparece el método en la estructura.
- paramF: Una lista de los parámetros del método, cada uno con los siguientes campos:
  - Nombre (nombre): El nombre del parámetro.
  - Tipo (tipo): El tipo de dato del parámetro.
  - Posición (posicion): La posición del parámetro en la lista de parámetros del método.
- variables: Una lista de las variables declaradas del método, cada una con los siguientes campos:
  - Nombre (nombre): El nombre de la variable.
  - Tipo (tipo): El tipo de dato de la variable.
  - Posición (posicion): El orden en el que aparece declarada la variable en el metodo.

- Constructor (constructor): Contiene información sobre el constructor de la estructura. cada uno con los siguientes campos:
  - Nombre (nombre): constructor.
  - paramF: Una lista de los parámetros del constructor, cada uno con los siguientes campos:
    - Nombre (nombre): El nombre del parámetro.
    - Tipo (tipo): El tipo de dato del parámetro.
    - Posición (posicion): La posición del parámetro en la lista de parámetros.
  - variables: Una lista de las variables declaradas en el constructor, cada una con los siguientes campos:
    - Nombre (nombre): El nombre de la variable.
    - Tipo (tipo): El tipo de dato de la variable.
    - Posición (posicion): El orden en el que aparece declarada la variable en el constructor.

3. Start: Esta sección representa la función `start` del programa, que es el punto de inicio. Contiene los siguientes campos:

- Nombre (nombre): start
- Retorno (retorno): void
- Posición (posición): 0
- Atributos (atributos): Una lista de las variables locales que se declaran dentro de ella. Cada uno con los siguientes campos:
  - Nombre (nombre): El nombre de la variable
  - Tipo (tipo): El tipo de dato del atributo.
  - Posición (posición): La posición en la lista de atributos de la función `start`.

Se aclara que en la TS también se agregan las clases predefinidas (IO, Object, Int, Char, Str, Bool, Array) y sus métodos a pesar de no estar explícitas en el programa a analizar.
En resumen, este JSON proporciona una representación estructurada de las clases, métodos, parámetros y variables locales definidas en el programa, lo que permite un análisis semántico y una comprensión más profunda de su estructura y comportamiento.

Con la tabla de símbolos construida, se procede a realizar las tareas propias del análisis semántico las cuales son:

- recolectar, entender y controlar todas las entidades declaradas (*Chequeo de Declaraciones*)
- recolectar, entender y controlar todas las sentencias asociadas a las entidades recolectadas (*Chequeo de Sentencias*)

En esta etapa realizamos el Chequeo de Declaraciones.

3.3.3. Chequeo de Declaraciones.

Como TinyRu es un lenguaje con referencias hacia adelante es necesario realizar el chequeo de declaraciones una vez finalizado el análisis sintáctico, cuando se recopiló toda la información de las entidades del programa. Es decir en dos pasadas:

En la primera: mientras se realiza el análisis sintáctico, se construye la tabla de símbolos como se definió anteriormente.

En la segunda: se realiza el chequeo de declaraciones que consta de dos partes importantes: el chequeo de entidades y la consolidación de la tabla de símbolos, ambos conceptos serán explicados más adelante.

3.3.3.1 Chequeo de Entidades

En esta etapa verificamos que:

1. No haya nombres repetidos en el mismo contexto
2. Todo nombre usado en una declaración haya sido declarado en el contexto adecuado.
3. No haya herencia circular.
4. Todo método redefinido tenga exactamente la misma signatura que el ancestro.

En particular, vemos que para cada clase:

- No pueda tener dos métodos con el mismo nombre,
- No pueda tener dos atributos con el mismo nombre,
- Se define un constructor
- No pueda tener más de un constructor,
- Se defina un “impl”
- Se defina un “struct”
- No pueda heredar de clases predefinidas como Array, Int, etc.
- No pueda heredar de clases inexistentes (que no han sido declaradas).
- Pueda tener un atributo y un método con el mismo nombre
- Pueda tener una variable local y un método con el mismo nombre
- No pueda tener métodos con variables locales declaradas con el mismo nombre que algún parámetro del método.

Para cada método:

- Sea sobreescrito solo si toda su signatura coincide con la de su ancestro
- Dos parámetros no tengan el mismo nombre
- Dos variables no tengan el mismo nombre

- Di pueda haber un parámetro y una variable con el mismo nombre
- La declaración de sus variables sea correcta y con tipos existentes
- La declaración de sus parámetros sea correcta y con tipos existentes

Para cada atributo:

- Un atributo NO puede tener el mismo nombre que un atributo ancestro.
- Su declaración sea correcta y con tipos existentes

3.3.3.2 Consolidación de la Tabla de Símbolos

La consolidación consiste en actualizar todas las tablas con las entidades que son heredadas de otras clases. Es decir:

- Agregar métodos heredados (consolidar la tabla de métodos)
- Agregar atributos heredados (consolidar la tabla de atributos)

Teniendo las siguientes consideraciones:

- La tabla de métodos de cada clase debe considerar los métodos sobre-escritos.
- Una clase deberá tener acceso a la última versión de los métodos que tiene y hereda.
- Controlar si un método está correctamente redefinido
- Los atributos privados a pesar de no ser visibles si son heredados.

3.3.4. Errores semánticos detectables

En esta sección, se exponen los errores semánticos que hemos identificado y manejado en nuestro compilador. Estos casos abarcan diversas situaciones en las que el código fuente puede contener errores semánticos.

Las excepciones semánticas tendrán como salida el número de línea donde se produjo el error, número de columna y una breve descripción:

ERROR: ERROR: SEMANTICO - DECLARACIONES
| NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: |

Estos son los principales errores semánticos que hemos tomado en cuenta:

- Structs con el mismo nombre dentro del mismo codigo.
  Ejemplo:
  ```
  struct Clase{
      pri Array Str a;
      Str b;
  }
  struct Clase{
      int num;
  }
  ```
  | DESCRIPCION: Ya existe un struct con el nombre "Clase"|

- Mas de un impl para el mismo struct.

Ejemplo:

```
impl Clase {
    .(Int vector){
        this.vector = vector; }
    .()
}
```

| DESCRIPCION: Ya existe un impl para el struct "Clase"|

- Un struct sin constructor.
  Ejemplo:

```
impl Clase{
 fn met ()-> Int{}
}
```

| DESCRIPCION: No se definio un constructor para la clase "Clase"|

- Mas de un constructor para el mismo struct.
  Ejemplo:

```
impl Clase{
 fn met ()-> Int{}
}
impl Clase{}
```

| DESCRIPCION: Ya existe un método constructor en la clase "Clase"|

- Una clase solo con impl, sin struct.
  | DESCRIPCION: Definición de estructura incompleta. No se declaró un struct para la clase "A"|

- Una clase solo con struct, sin impl.
  | DESCRIPCION: Definición de estructura incompleta. No se declaro un impl para la clase "A"|

- Atributos con el mismo nombre dentro de un Struct.
  Ejemplo:

```
struct Hola:C{
    pri Array Str a;
    Str a;
}
```

| DESCRIPCION: “Ya existe un atributo con el nombre "a" en la clase "Hola"”|

- Métodos con el mismo nombre dentro de un Struct.
  Ejemplo:

```
impl Clase{
 fn met ()-> Int{}
 st fn met ()-> Array Int{}
}
```

| DESCRIPCION: Ya existe un metodo con el nombre "met" en la clase "Clase"|

- Redefinir parámetros en un método .
  Ejemplo:
  ```
  fn f(Str a) -> void{Char a;}
  ```
  | DESCRIPCION: No se puede redefinir el parámetro "a" en el método "f"|

- Declarar variables con el mismo nombre dentro de start
  Ejemplo:
  ```
  start{
   Int a;
   Str a;
  }
  ```
  | DESCRIPCION: Ya existe una variable con el nombre "a" en la clase "start"|

- Declarar variables con el mismo nombre dentro de un método
  Ejemplo:
  ```
  fn m1()->void{Char b1,e,c,e;}
  ```
  | DESCRIPCION: Ya existe una variable con el nombre "e" en el método "m1"|

- Heredar de clases predefinidas.
  Ejemplo:
  ```
  struct A: Int{}
  ```
  | DESCRIPCION: No esta permitido heredar de la clase predefinida Int |

- Heredar de clases no definidas.
  | DESCRIPCION: Herencia de struct inexistente. El struct "A" hereda del struct inexistente: "B"|

- Herencia circular
  Ejemplo:
  ```
  struct A: B{}
  struct B: C{}
  struct C: A{}
  ```
  | DESCRIPCION: Herencia Cíclica en la clase C |

- Redefinir atributo de un metodo heredado .
  Ejemplo:
  ```
  struct A:B{
  Int a;
  }
  struct B{
  Str b,c,a;
  }
  ```
  | DESCRIPCION: No se puede redefinir el atributo "a" en la clase "A". Ya que es un atributo heredado de "B"|

- Redefinir incorrectamente un metodo heredado (distinta firma) .
  Ejemplo:

```
struct A:B{
Int a;
}
struct B{
Str b,c,a;
}
```

| DESCRIPCION: No se puede redefinir el atributo "a" en la clase "A". Ya que es un atributo heredado de "B"|

3.3.5. Casos de Pruebas Semanticos

En esta sección se detallan las pruebas realizadas para evaluar el funcionamiento del analizador semántico. Cada prueba se describe con su respectivo escenario, nombre del test y la salida esperada. Dichas pruebas abarcan una variedad de situaciones para garantizar la robustez y precisión del analizador en diferentes contextos y casos de uso.

A continuación, se presenta un resumen de las pruebas realizadas:

Escenario	Nombre Test	Salida esperada
Struct	test_error_2 Verificar que haya error cuando se quiere definir dos structs con el mismo nombre	ERROR: SEMANTICO - DECLARACIONES Ya existe un struct con el nombre "A"
Struct	test_error_3 Verificar que haya error cuando se quiere definir dos impl para el mismo struct	ERROR: SEMANTICO - DECLARACIONES Ya existe un impl para el struct "A"
Struct	test_error_4 Verificar que haya error cuando se quiere definir un struct con mas de un constructor	ERROR: SEMANTICO - DECLARACIONES Ya existe un metodo constructor en la clase "A
Struct	test_error_5 Verificar que haya error cuando se quiere definir un struct sin un constructor	ERROR: SEMANTICO - DECLARACIONES No se definio un constructor para la clase "A"
Struct	test_error_6 Verificar que haya error cuando se quiere definir dos structs con el mismo nombre	ERROR: SEMANTICO - DECLARACIONES Ya existe un struct con el nombre "A"
Struct	test_error_7 Verificar que haya error cuando se quiere definir una clase solo	ERROR: SEMANTICO - DECLARACIONES Definicion de estructura incompleta. No se declaro un struct para la clase "A"

Escenario	Nombre Test	Salida esperada
	con impl, sin struct	
Struct	test_error_8 Verificar que haya error cuando se quiere definir una clase solo con struct, sin impl	ERROR: SEMANTICO - DECLARACIONES Definicion de estructura incompleta. No se declaro un impl para la clase "A"
Struct	test_ok_2 Verificar que se permita definir un atributo y un metodo con el mismo nombre	CORRECTO: SEMANTICO - DECLARACIONES
Atributo	test_error_1 Verificar que haya error cuando se quiere definir dos atributos con el mismo nombre en el mismo contexto	ERROR: SEMANTICO - DECLARACIONES Ya existe un atributo con el nombre "h" en la clase "A"
Atributo	test_ok_8 Verificar que si defino un atributo privado en una superclase, tambien lo herede la subclase (Json)	CORRECTO: SEMANTICO - DECLARACIONES
Metodo	test_error_12 Verificar que haya error cuando se quiere definir dos metodos con el mismo nombre en el mismo contexto	ERROR: SEMANTICO - DECLARACIONES Ya existe un metodo con el nombre "f" en la clase "A"
Metodo	test_error_13 Verificar que haya error cuando se quiere redefinir parametros en un metodo	ERROR: SEMANTICO - DECLARACIONES No se puede redefinir el parametro "a" en el metodo "f"
Metodo	test_ok_3 Verificar que en el caso de que una superclase y una subclase definan el mismo nombre de metodo pero con diferentes atributos	CORRECTO: SEMANTICO - DECLARACIONES
Variable	test_error_14 Verificar que haya error cuando se quiere declarar variables con el mismo nombre dentro de start	ERROR: SEMANTICO - DECLARACIONES Ya existe una variable con el nombre "a" en la clase "start"
Variable	test_error_15 Verificar que haya error	ERROR: SEMANTICO - DECLARACIONES

Escenario	Nombre Test	Salida esperada
	cuando se quiere declarar variables con el mismo nombre dentro de un metodo	Ya existe una variable con el nombre "e" en el metodo "m1"
Variable	test_error_16 Verificar que haya error cuando se quiere declarar variables con el mismo nombre dentro de un metodo	ERROR: SEMANTICO - DECLARACIONES No se puede redefinir el atributo "a" en la clase "A". Ya que es un atributo heredado de "B"
Herencia	test_ok_1 Verificar que se consolide bien la TS, cuando se redefine un metodo heredado con la misma firma	CORRECTO: SEMANTICO - DECLARACIONES
Herencia	test_error_9 Verificar que haya error cuando se quiere definir un struct que herede de una clase predefinida	ERROR: SEMANTICO - DECLARACIONES No esta permitido heredar de la clase predefinida Char
Herencia	test_error_10 Verificar que haya error cuando se quiere definir un struct que herede de una clase inexistente	ERROR: SEMANTICO - DECLARACIONES Herencia de struct inexistente. El struct "A" hereda del struct inexistente: "C" .Primero defina "C" para poder utilizarlo.
Herencia	test_error_11 Verificar que haya error cuando hay herencia circular	ERROR: SEMANTICO - DECLARACIONES Herencia Cíclica en la clase C
Herencia	test_error_17 Verificar que haya error cuando se quiere redefinir un metodo heredado con distinta firma (distinto tipo de ret)	ERROR: SEMANTICO - DECLARACIONES No se puede redefinir el metodo heredado "f0" porque no retorna el mismo tipo que su metodo heredado
Herencia	test_error_18 Verificar que haya error cuando se quiere redefinir un metodo heredado con distinta firma (distinta cantidad de parametros)	ERROR: SEMANTICO - DECLARACIONES No se puede redefinir el metodo heredado "f0" porque no tiene la misma cantidad de parametros que su metodo heredado
Herencia	test_error_19 Verificar que haya error cuando se quiere redefinir un metodo heredado con distinta firma (distinto tipo paramF)	ERROR: SEMANTICO - DECLARACIONES No se puede redefinir el metodo heredado "f0" porque el parámetro formal en la posicion 0 tiene distinto tipo

Escenario	Nombre Test	Salida esperada
Herencia	test_error_20 Verificar que haya error cuando se quiere redefinir un método heredado con distinta firma (distinto nombre paramF)	ERROR: SEMANTICO - DECLARACIONES No se puede redefinir el método heredado "f0" porque el parámetro formal en la posición 0 tiene distinto nombre.
Herencia	test_ok_6 Verificar que los constructores de superclase y subclase puedan tener parametros diferentes	CORRECTO: SEMANTICO - DECLARACIONES
Herencia	test_ok_4 Verifica que cuando un struct no hereda de nada, hereda de Object	CORRECTO: SEMANTICO - DECLARACIONES
Declaración	test_error_21 Verificar que haya error cuando se quiere declarar con un tipo que no se ha definido (variable)	ERROR: SEMANTICO - DECLARACIONES Tipo no definido: el tipo "C" de la variable "a" no está definido
Declaración	test_error_22 Verificar que haya error cuando se quiere declarar con un tipo que no se ha definido (en atributos)	ERROR: SEMANTICO - DECLARACIONES Tipo no definido: "C" no está definido
Declaración	test_error_23 Verificar que haya error cuando se quiere declarar con un tipo que no se ha definido (en parametros)	ERROR: SEMANTICO - DECLARACIONES Tipo de parámetro no definido: "C" no está definido
Declaración	test_error_24 Verificar que haya error cuando se quiere declarar con un tipo que no se ha definido (en ret de método)	ERROR: SEMANTICO - DECLARACIONES Tipo de retorno no definido: "C" no está definido
Declaración	test_error_25 Verificar que haya error cuando se quiere declarar con un tipo que no se ha definido (en paramF de constructor)	ERROR: SEMANTICO - DECLARACIONES Tipo de parámetro no definido: "C" no está definido
Declaración	test_error_26 Verificar que haya error cuando se quiere declarar con un tipo que no se ha definido (en variable de constructor)	ERROR: SEMANTICO - DECLARACIONES Tipo de variable no definido: "C" no está definido

Escenario	Nombre Test	Salida esperada
Declaración	test_ok_9 Verificar que se pueda definir un obj de una clase definida	CORRECTO: SEMANTICO - DECLARACIONES
Tipo	test_ok_5 Verificar que se pueda definir un metodo de tipo void y el json este correcto	CORRECTO: SEMANTICO - DECLARACIONES
Tipo	test_error_29 Verificar que no se pueda definir dos parametros iguales para el mismo metodo	ERROR: SEMANTICO - DECLARACIONES Ya existe un parámetro con el nombre "b" en el metodo "f0"
Tipo	test_ok_7 Verificar que si defino un atributo de tipo Array, se escriba correctamente en el json	CORRECTO: SEMANTICO - DECLARACIONES
Tipo	test_ok_8 Verificar que si defino un atributo privado en una superclase, también lo herede la subclase (Json)	CORRECTO: SEMANTICO - DECLARACIONES
Tipo	test_error_30 Verificar que no se pueda definir un tipo de variable de un tipo que no fue definido previamente	ERROR: SEMANTICO - DECLARACIONES Tipo no definido: "Construccion" no esta definido

El conjunto completo de archivos de prueba en formato .ru se encuentra adjunto en la carpeta llamada test.

3.3.6. Consideraciones

Durante el desarrollo del analizador semántico, se tuvieron en cuenta diversas consideraciones que garantizan el correcto funcionamiento del mismo:

- Puede haber un atributo y un método de clase con los mismos nombres
- Puede haber una variable y un método de clase con los mismos nombres
- Una variable local declarada en un método puede tener el mismo nombre que un atributo del struct
- Un parámetros de un método puede tener el mismo nombre que un atributo del struct
- Un metodo y su variable local pueden tener el mismo nombre. Ejemplo: fn a() -> void{ Int a;}
- Un metodo y su parámetro pueden tener el mismo nombre. Ejemplo:fn a(Str a)->void{ }

3.4. Analizador Semántico - Sentencias

A continuacion, se describen los pasos para construir la etapa de chequeo de sentencias del analizador semántico del compilador TinyRu. Esto incluye la verificación de reglas semánticas como tipos de datos, asignación correcta de variables y compatibilidad de tipos en operaciones.

En esta etapa, se detallan procesos como la construcción del árbol AST (Árbol de Sintaxis Abstracta) y el uso de la tabla de símbolos para realizar verificaciones en expresiones, asegurando así que el código sea semánticamente correcto.

3.4.1. Tratamiento de Gramática

En el proceso de chequeo de sentencias del analizador semántico, es fundamental contar con una gramática bien definida que guíe la interpretación y validación del código fuente. En este informe, se describe el proceso de transformación de una gramática de la etapa 3, es decir gramática de atributos, con las respectivas acciones semánticas para el chequeo de declaraciones y construcción de TS, a una gramática correcta para la etapa 4.

Lo que se realizó, fue el agregado de las acciones semánticas que nos permitieran construir los AST necesarios para nuestro compilador, respetando el manual del mismo.

Las gramáticas de atributos son gramáticas libres de contexto aumentadas con atributos (datos) en los símbolos (terminales o no,de cada regla de producción) y reglas (o acciones) semánticas (funciones de atributos) asociadas a las reglas de producción. Está extiende la gramática convencional BNF mediante la incorporación de atributos asociados a los símbolos no terminales.La misma se ha construido mediante el EDT, con acciones semánticas para realizar análisis contextual y asignar información semántica a la estructura del programa.

3.4.1.1 Esquema de Traducción (EDT)

Los Árboles de Derivación Extendidos (EDTs) son una representación de la estructura jerárquica de un programa después de que ha sido analizado sintácticamente. A diferencia de los árboles de análisis sintáctico completos, los EDTs contienen fragmentos de programa incrustados dentro de los cuerpos de las reglas de producción. Estos fragmentos, son conocidos como acciones semánticas y pueden aparecer en cualquier posición dentro del cuerpo de una producción. Las acciones semánticas le asignan significado semántico a la

estructura del programa y se ejecutan durante el análisis sintáctico para realizar tareas semánticas como la verificación de tipos, la construcción de tablas de símbolos y la generación de código intermedio. Mediante la combinación de gramáticas libres de contexto con acciones semánticas, los EDTs proporcionan una representación compacta y manejable de la estructura y el significado de un programa, lo que facilita su posterior procesamiento en las etapas de análisis semántico y generación de código.

A continuación, las reglas a las que aplicamos las acciones:

⟨*Impl*⟩ ::= ”**impl**“ **idStruct** `{NodoStruct nodo = new NodoStruct(idStruct.lex); AST.insertStruct(nodo), AST.setCurrentStruct(nodo)}` “**{**“ ⟨*Miembros*⟩ “**}**”

⟨*Constructor* ⟩ ::= “**.**” ⟨*Argumentos-Formales*⟩ `{NodoMetodo nodo = new`

```
NodoMetodo("constructor");
AST.structActual.insertMet(nodo),
AST.setCurrentMetodo(nodo)} ⟨Bloque-Método⟩
"
```

⟨*Método*⟩ ::= ⟨*Forma-Método*⟩ "**fn**" idMetAt {NodoMetodo nodo = new NodoMetodo(idMetAt); AST.structActual.insertMet(nodo), AST.setCurrentMetodo(nodo)} ⟨*Argumentos-Formales*⟩ "**->**" ⟨*Tipo-Método*⟩ ⟨*Bloque-Método*⟩ | "**fn**" idMetAt {NodoMetodo nodo = new NodoMetodo(idMetAt); AST.structActual.insertMet(nodo), AST.setCurrentMetodo(nodo)} ⟨*Argumentos-Formales*⟩ "**->**" ⟨*Tipo-Método*⟩ ⟨*Bloque-Método*⟩

⟨*Sentencias*⟩ ::= ⟨*Sentencia*⟩ {NodoLiteral nodo = sentencia(); AST.metodoActual.insertSentencia(nodo)} ⟨*Sentencias1*⟩

⟨*Sentencia*⟩ ::= "**;**" {ret NodoLiteral nodo = new NodoLiteral(";")} | ⟨*Asignación*⟩"**;**" {ret NodoAsignacion nodo = asignacion()} | ⟨*Sentencia-Simple*⟩"**;**" {ret NodoExpresion nodo = sentenciaS()} | "**if**" "**(**"⟨*Expresión*⟩"**)**" {NodoLiteral exp = expresion(); NodoIf nodoIf = new NodoIf(exp);}⟨*Sentencia*⟩ {NodoLiteral s = sentencia(); nodoIf.insertSentencia(s);} ⟨*Sentencia1*⟩ {NodoLiteral sElse = sentencia1(); nodoIf.insertSentenciaElse(s); ret nodoIf}
| "**while**" "**(**" ⟨*Expresión*⟩ "**)**" {NodoLiteral exp = expresion(); NodoWhile nodoWhile = new NodoWhile(exp);}⟨*Sentencia*⟩ {NodoLiteral s = sentencia(); nodoWhile.insertSentencia(s); ret nodoWhile}
| ⟨Bloque⟩ {NodoBloque nodo = bloque(); ret nodo;}
| "**ret**" ⟨*Sentencia2*⟩ {NodoLiteral s = sentencia2(); NodoExpresion nodo = new NodoExpresion("Retorno",s);}

⟨*Sentencia1*⟩ ::= "**else**" ⟨*Sentencia*⟩ {ret sentencia();} | λ

⟨*Sentencia2*⟩ ::= ⟨Expresion⟩ "**;**" {ret expresion();} | "**;**"

⟨*Asignacion*⟩ ::= ⟨*AccesoVar-Simple*⟩{NodoLiteral nodoI = accesoVarSimple()} "**=**" ⟨*Expresion*⟩ {NodoLiteral nodoD = expresion(); ret NodoAsignacion nodo = new NodoAsignacion(nodoI, nodoD)}
| ⟨*AccesoSelf-Simple*⟩ {NodoLiteral nodoI =

accesoSelfSimple() } "=" ⟨*Expresion*⟩ {NodoLiteral nodoD = expresion(); ret NodoAsignacion nodo = new NodoAsignacion(nodoI, nodoD)}

⟨*EncadenadosSimples*⟩ ::= ⟨*Encadenado-Simple*⟩ {NodoLiteral nodoI = encadenadoSimple()} ⟨*EncadenadosSimples1*⟩ {NodoLiteral nodoD = encadenadosSimple1(); ret NodoAcceso nodo = new NodoAcceso(nodoI, nodoD)}

⟨*Expresion*⟩ ::= ⟨*ExpAnd*⟩ {NodoLiteral nodoI = expAnd()} ⟨*Expresion1*⟩ {NodoLiteral nodoD = expresion1(); ret new NodoExpBin(nodoI,||, nodoD)}

⟨*Expresion1* ⟩ ::= "||" ⟨*ExpAnd*⟩ {NodoLiteral nodoI = expAnd()} ⟨*Expresion1*⟩ {NodoLiteral nodoD = expresion1(); ret new NodoExpBin(nodoI,||, nodoD)} | λ

⟨*ExpAnd*⟩ ::= ⟨*ExpIgual*⟩ {NodoLiteral nodoI = expIgual()} ⟨*ExpAnd1*⟩ {NodoLiteral nodoD = expAnd1(); ret new NodoExpBin(nodoI,&&, nodoD)}

⟨*ExpAnd1*⟩ ::= "&&" ⟨*ExpIgual*⟩ {NodoLiteral nodoI = expIgual()} ⟨*ExpAnd1*⟩ {NodoLiteral nodoD = expAnd1(); ret new NodoExpBin(nodoI,&&, nodoD)} | λ

⟨*ExpIgual*⟩ ::= ⟨*ExpCompuesta*⟩ {NodoLiteral nodoI = expCompuesta()} ⟨*ExpIgual1*⟩ {NodoLiteral nodoD = expIgua1(); ret new NodoExpBin(nodoI,==, nodoD)}

⟨*ExpIgual*1⟩ ::= ⟨*OpIgual*⟩ ⟨*ExpCompuesta*⟩ {NodoLiteral nodoI = expCompuesta()} ⟨*ExpIgual1*⟩ {NodoLiteral nodoD = expIgua1(); ret new NodoExpBin(nodoI,==, nodoD)} | λ

⟨*ExpCompuesta*⟩ ::= ⟨*ExpAd*⟩ {NodoLiteral nodoI = expAd()} ⟨*ExpCompuesta1*⟩ {NodoLiteral nodoD = expCompuesta1(); ret new NodoExpBin(nodoI,op, nodoD)}

⟨*ExpAd*⟩ ::= ⟨*ExpMul*⟩ {NodoLiteral nodoI = expMul()} ⟨*ExpAd1*⟩ {NodoLiteral nodoD = expAd1(); ret new NodoExpBin(nodoI,op, nodoD)}

⟨*ExpAd1*⟩ ::= ⟨*OpAd*⟩ ⟨*ExpMul*⟩ {NodoLiteral nodoI = expMul()} ⟨*ExpAd1*⟩ {NodoLiteral nodoD = expAd1(); ret new NodoExpBin(nodoI,op,nodoD)} λ

⟨*ExpUn*⟩ ::= ⟨*OpUnario*⟩ ⟨*ExpUn*⟩ {ret new NodoExpun(expUn(),opUnario())} | ⟨*Operando*⟩ {ret operando()}

⟨*ExpresionParentizada*⟩ ::= "**(**" ⟨*Expresion*⟩ "**)**" ⟨*ExpresionParentizada1*⟩ {ret new NodoAcceso(expresion(),expresionParentizada())}

⟨*AccesoSelf* ⟩ ::= "**self**" ⟨*AccesoSelf*1⟩ {ret new NodoAcceso(self,accesoSelf1())}

⟨*Llamada-Método*⟩ ::= id ⟨*Argumentos-Actuales*⟩ {nodoI = new NodoLlamadaMetodo(id,argumentosActuales())}
⟨*Llamada-Método1*⟩ {if(llamada metodo1() == lambda) { ret nodoI} else {ret new NodoAcceso(nodoI,llamadaMetood1())}

⟨*Llamada-Método-Estático*⟩ ::= idStruct "**.**" ⟨*Llamada-Método*⟩ ⟨*Llamada-Método-Estático1*⟩ {if(llamadametodoEstatico1() == lambda) { ret llamadaMetodo()} else {ret new NodoAcceso(nodoI,llamadaMetood1())}

3.4.2. Implementación de AST

El Árbol de Sintaxis Abstracta (AST) es una estructura de datos que representa la estructura jerárquica de un programa de manera abstracta. En resumen, es una representación intermedia del código fuente que facilita su análisis y manipulación durante la etapa de compilación.

El AST se construye durante el análisis sintáctico del compilador, a partir del Árbol de Derivación de Sintaxis (EDT) generado en la fase de análisis léxico y sintáctico. Mientras que el EDT refleja la estructura exacta y completa de las reglas gramaticales del lenguaje, el AST simplifica esta estructura para capturar la lógica y la semántica del programa de una manera más manejable.

3.4.2.1 Diseño de AST

En nuestro caso, el AST lo construimos a partir del EDT de la etapa 3, pero con algunas modificaciones y adiciones para incorporar acciones semánticas relacionadas con el chequeo de tipos. Esto significa que, además de simplemente capturar la estructura del programa, el AST también almacena información sobre los tipos de datos de las expresiones y las variables.

Nuestro AST esta conformado por nodos, que nos ayudan a realizar el chequeo de tipos.Cada nodo representa una construcción específica del lenguaje, como una asignación, acceso a una variable, llamada a un método, bucle while, condición if, entre otros. Luego a partir de un método diseñado en nuestro código y definido como checkTypes(ts), una vez construida la tabla de símbolos y la estructura AST, vamos chequeando nodo a nodo su correcto tipo, teniendo en cuenta herencia de tipos, clases objeto, clases predefinidas, tipo de nodos hijos, etc.

Para diseñar e implementar el AST realizamos algunos pasos, que se detallan a continuación:

- Identificar los tipos de nodos que va a tener el AST:
  - Nodo: es la subestructura principal, de el heredan todos los demas nodos.
  - NodoAcceso: este nodo pretende identificar todos los accesos que se realicen en código fuente. Desde acceso a atributos, hasta encadenados de acceso a métodos, etc.
    - Ej: a.llamadaM1().llamadaM2()
  - NodoAccesoArray: este nodo pretende identificar por separado, los accesos a una array, dado que se escriben distintos a los accesos comunes.
    - Ej: a[4]
  - NodoAsignación: este nodo pretende identificar el correcto tipo de todas las asignaciones del programa, por esto analiza el lado izquierdo y el lado derecho del '=' y busca que ambos lados sean del mismo tipo o cumplan con las reglas de objeto y herencia especificadas en tinyRu, a continuación coloco algunos ejemplos:

■ Ej: la superclase de los objetos de clase es Object, por ende toda variable definida como Object puede recibir cualquier tipo de objeto de clase, de Struct que estén definidos en el codigo fuente.

- Se permite la herencia, por lo tanto si D hereda de C y C hereda de A, yo puedo crear una variable de tipo A y asignarle una instancia de objeto del tipo C o D.
- Todos las instancia de objeto o las declaraciones de variables de tipo objeto, pueden recibir 'nil'

  - NodoBloque: este nodo identifica todos los bloques de sentencias que se ubiquen dentro de if, while, entre otros.
  - NodoIf/NodoElse: este nodo está conformado por 2 nodos, cuando en el código fuente existe if()else(), es decir hay sentencias else, entonces al nodoIf le insertamos las sentencias del nodoElse, esto lo realizamos para imprimir correctamente el json. La función final es identificar condicionales if o ifelse.
  - NodoWhile: este nodo pretende identificar estructuras de bucle while y chequear los tipos de las sentencias que se encuentran dentro.
  - NodoExpBin: este nodo identifica toda expresión binaria y sus tipos del lado izquierdo y derecho del operador(aritmético o lógico) en cuestión. En caso de que se presente una expresión que no se puede resolver por diferencia de tipos, emite una excepción semántica.
  - NodoExpUn: este nodo representa a toda expresión unaria (ej: a++).
  - NodoExpresion:este nodo identifica a la expresión completa, no tiene lado izquierdo y lado derecho si no que evalúa la expresión como un todo y le coloca tipo.
  - NodoLiteral: identifica las partes más pequeñas de una expresión. Identifica los id y busca depende de su tipo si se encuentra en la ts o no, si es una variable, un idstruct, el nombre de un método o si ha sido declarado en algún lado. También es el encargado de colocarle tipo a los literales, para cuando sean evaluados en asignaciones, expresiones, etc.
  - NodoLlamadaMetodo: se encarga de analizar todas las llamadas a métodos del código, busca si los métodos existen en el struct definido, si han sido heredados, analiza el caso particular de crear un objeto llamando a un constructor, análisis de parámetros, entre otras cosas.

- NodoMetodo: se utiliza para identificar los métodos y guardar todo lo q pertenece a un método, para luego imprimir el json.
- NodoStruct: idem método. Es simplemente para respetar la consigna de jerarquia.

- Luego comenzamos con la construcción del AST, en la misma ejecución del análisis sintáctico y la ts.
- Por ultimo con la ts y el ast, se realiza el chequeo de sentencias(mediante metodo checkTypes(ts))

El método checkTypes(ts), se sobreescribe en los diferentes nodos que heredan todos de la clase Nodo. Esta sobreescritura se realiza porque depende qué nodo estamos recorriendo, son las validaciones que tendremos que realizar. Este método es básicamente una consolidación del árbol AST, donde según se incumplan las reglas se emiten excepciones semánticas, donde se indica errores de tipos.

3.4.2.2 Representación del AST

Al igual que la etapa 3, se representó el AST a través de un archivo JSON con la siguiente estructura:

1. Nombre del Archivo (nombre): Indica el nombre del archivo o programa analizado:, "AST-nombreArchivo.ru"

2. Estructuras (structs): Esta es una lista que contiene información sobre todas las estructuras definidas en el programa. Cada estructura tiene los siguientes campos:

- Nombre (name): El nombre del struct.
- Métodos (métodos): Una lista de los métodos definidos en la estructura, cada uno con los siguientes campos:
  - nombreMetodo: Nombre del método.
  - sentencias: Lista de Nodo Sentencias dentro del bloque.

Entonces todos los metodos tienen sus Nodos Sentencia. Estas pueden ser de diferentes tipos y tiene sus respectivos campos, tales como:

- Nodo Asignacion: Representa una asignacion nodoI = nodoD.
  - tipo: Tipo de la asignación.
  - NodoIzq: Nodo del lado izquierdo de la asignación.
  - NodoDer: Nodo del lado derecho de la asignación.

- NodoIf: Representa una sentencia condicional if.
  - NodoExpresion: Nodo de la expresión condicional.
  - sentencias: Lista de sentencias a ejecutar si la condición es verdadera. ○ NodoElse: (opcional) contiene sentencias a ejecutar si la condición es falsa.

- NodoWhile: Representa un bucle while.
  - NodoExpresion: Nodo de la expresión condicional.
  - sentencias: Lista de sentencias a ejecutar si la condición es verdadera.

- NodoRetorno: Representa una sentencia de retorno.

- tipo: Tipo de la expresión de retorno.
- NodoExpresion: Nodo de la expresión que se retorna.

- Sentencia Simple: Representa una sentencia simple.
  - tipo: Tipo de la expresión.
  - NodoExpresion: Nodo de la expresión.

- Bloque: Representa un bloque de sentencias entre llaves {}.
  - NodoSentencias: Lista de nodos sentencias dentro del bloque.

Como se observa los nodos anteriores pueden contener dentro otros nodos. Entonces tenemos el Nodo Expresión. Las expresiones pueden ser de diferentes tipos, como:

- NodoLiteral: Representa un valor literal.
  - nombre: Nombre del literal.
  - tipo: Tipo del literal.
  - valor: Valor del literal.

- NodoLlamada Metodo: Representa una llamada a método.
  - tipo: Tipo de retorno del método.
  - metodo: Nombre del método llamado.
  - argumentos: Lista de argumentos pasados al método.

- NodoAcceso: Representa un acceso a un atributo desde objeto de tipo struct ○ nodoIzq: Nodo del objeto al que se accede.
  - nodoDer: Nodo del atributo al que se accede.

- Acceso Array: Representa un acceso a un elemento de un array.
  - array: Nodo del array.
  - indice: Nodo del índice del elemento a acceder.

- Expresion Parentizada: Representa expresión entre paréntesis.
  - expresion: Nodo de la expresión interna.

En resumen, este JSON proporciona una representación estructurada de las clases, métodos y sus sentencias definidas en el programa, lo que permite un análisis semántico y una comprensión más profunda de su estructura y comportamiento.

En la etapa anterior con la tabla de símbolos construida, realizamos la tarea de: recolectar, entender y controlar todas las entidades declaradas (*Chequeo de Declaraciones*) Ahora, con el AST formado, continuamos a: recolectar, entender y controlar todas las sentencias asociadas a las entidades recolectadas (*Chequeo de Sentencias*). Entonces, en esta etapa realizamos el Chequeo de Sentencias.

3.4.3. Chequeo de Sentencias.

Como TinyRu es un lenguaje con referencias hacia adelante es necesario realizar el chequeo de sentencias una vez finalizado el análisis sintáctico, cuando se recopiló toda la información de las entidades del programa. Es decir en dos pasadas:

- En la primera: mientras se realiza el análisis sintáctico, se construye la tabla de símbolos y el arbol abstracto AST.
- En la segunda: se realiza el chequeo de declaraciones y sentencias .

En esta etapa trabajamos sobre el cuerpo de las unidades (métodos/constructores) y verificamos que:

1. Resolución de nombres
2. Chequeo de tipos
3. Herencia de tipos

Para cada método:

- Si un método es llamado de otro método, chequeamos que exista dentro del struct donde estamos o herede de la clase donde se definió el método inicialmente.
- Chequeamos que los tipos de parámetros que enviamos coincida con lo que espera el método y definimos en la ts.
- Chequeamos que los métodos retornen lo definido en su firma o en el caso de que retornen objeto, que sean del mismo tipo o tipos heredados.

Todos los chequeos se realizan a partir de la ejecución de la función checkstypes(ts) que recibe como parámetro la tabla de símbolos y en base al nodo que se está trabajando hace las validaciones correspondientes. La misma será explicada caso a caso, en la sección posterior.

3.4.4. Errores semánticos detectables

En esta sección, se exponen los errores semánticos que hemos identificado y manejado en nuestro compilador. Estos casos abarcan diversas situaciones en las que el código fuente puede contener errores semánticos.

Las excepciones semánticas tendrán como salida el número de línea donde se produjo el error, número de columna y una breve descripción:

ERROR: ERROR: SEMANTICO - SENTENCIAS
| NUMERO DE LINEA: | NUMERO DE COLUMNA: | DESCRIPCION: |

Estos son los principales errores semánticos que hemos tomado en cuenta:

nodos de Accesos (de la forma nodoI.nodoD o nodoI[nodoD])

- Cuando se quiere hacer un acceso de array en una id que no es de tipo array.
  Ejemplo:
  ```
  struct A{}
  impl A{ .(){
  Int a;
  a[2] = 1;}}
  ```
  | DESCRIPCION: "a" no es un array por lo que no se puede acceder a un indice |

- Cuando se quiere hacer un acceso de array en una id que no esta declarado.
  Ejemplo:

```
struct A{}
impl A{ .(){
a[2] = 1;}}
```
| DESCRIPCION: "a" no esta declarado en el struct 'A' ni en el metodo constructor |

- Cuando se quiere hacer un acceso de array en un indice que no es int.
  Ejemplo:
```
struct A{Str s}
impl A{ .(){
Array Int a;
a[s] = 1;}}
```
  | DESCRIPCION: La posicion para acceder debe ser de tipo Int

- Cuando se quiere hacer usar un identificador no declarado como indice.
  Ejemplo:
```
struct A{}
impl A{ .(){
Array Int a;
a[s] = 1;}}
```
  | DESCRIPCION: El id "s" no esta declarado en el struct 'A' ni en el metodo 'constructor' |

- Cuando se quiere hacer un acceso a un id que no es de tipo struct.
  Ejemplo:
```
impl B{.(Int a){a.b = 1;}}
```
  | DESCRIPCION: “a” no es un struct por lo que no se puede usar para realizar un acceso|

- Cuando se quiere hacer un acceso a un id que no esta declarado.
  Ejemplo:
```
struct A{}
impl A{ .(){
a.s = 1;}}
struct B{Str s;}
impl B{ .(){}}
```
  | DESCRIPCION: "a" no esta declarado en el struct 'A' ni en el metodo constructor |

- Cuando se quiere hacer un acceso a un atributo que no existe en el struct. Ejemplo:
```
struct A{}
struct B{A a;}
impl B{.(){a.b = 1;}}
```
  | DESCRIPCION: El atributo “b” no existe en el struct “a”. Por lo tanto no se puede acceder a el |

- Cuando se quiere hacer un acceso a un metodo que no existe en el struct. Ejemplo:
```
struct A{}
struct B{A a;}
impl B{.(){(a.b());}}
```
  | DESCRIPCION: El metodo “b” no existe en el struct “a”. Por lo tanto no se puede acceder a el |

Para los nodos de Asignacion (de la forma nodoI = nodoD)

- Cuando se quiere hacer una asignacion con tipos incompatibles
  Ejemplo:
  ```
  Str string;
  string = nil;
  ```
  | DESCRIPCION: Incompatibilidad de tipos. No se puede asignar 'nil' a una variable de tipo 'Str'.|

- Cuando se quiere hacer una asignacion con tipos incompatibles
  Ejemplo:
  ```
  Array Int array1;
  Array Str array2;
  array1 = array2;
  ```
  | DESCRIPCION: Incompatibilidad de tipos. No se puede asignar un objeto de tipo Array Str a la variable 'array1' definida de tipo Array Int |

- Cuando se quiere hacer usar un identificador no declarado en una asignacion
  Ejemplo:
  ```
  struct B{}
  impl B{.(){a = 1;}}
  ```
  | DESCRIPCION: El identificador con el nombre "a" no esta declarado en el metodo "constructor" |

- Cuando se quiere hacer una asignacion con tipos incompatibles
  Ejemplo:
  ```
  impl A {
  fn m() -> void {
  D d;
  d = self;}
  ```
  | DESCRIPCION: Incompatibilidad de tipos. No se puede asignar un objeto de tipo A a la variable definida de tipo D |

Para los nodos de Expresion Binaria (de la forma nodoI operador nodoD

- Cuando se quiere hacer una operacion aritmetica con tipos que no sean Int
  Ejemplo:
  ```
  Str c;
  Int a;
  b = a - c;
  ```
  | DESCRIPCION:Incompatibilidad de tipos. No se puede realizar una operacion "-" entre un Int y un Object. Ambos deben ser enteros |

- Cuando se quiere hacer una operacion logica con tipos distintos
  Ejemplo:
  ```
  Bool a; Bool b;
  Object c;
  b = a == c;
  ```
  | DESCRIPCION:Incompatibilidad de tipos. No se puede realizar una operacion "==" entre un Bool y un Object|

Para los nodos de If y While (de la forma if/while(expresion){sentencias})

- Cuando se quiere hacer una que la expresion no sea bool
  Ejemplo:

```
if(1+2){}
```
| DESCRIPCION:La condicion del if debe ser de tipo Bool |

Para los nodos de Llamada a metodo (de la forma metodo(argumentos))

- Cuando se quiere usar un constructor o metodo no definido
  | DESCRIPCION: No se puede crear una instancia de 'R' porque no existe el struct. Primero debe crear el struct 'R'. |

- Cuando se quiere definir el tamaño de un array con algo que no es de tipo Int
  | DESCRIPCION: Incompatibilidad de tipos. No se puede definir el tamaño de un array con algo cuyo tipo no es 'Int' |

- Cuando se llama a un constructor o metodo con la cantidad de parametros incorrecta
  | DESCRIPCION: Cantidad de argumentos incorrectos en la llamada a metodo. Para llamarlo debe pasar la cantidad de parametros corresponientes indicados en su firma. Se esperaban “x” parametros y llegaron “y”. |

- Cuando cuando se llama a un constructor o metodo con un parametro cuyo tipo que no se encuentra en el arbol de herencia del tip que se espera como parametro. | DESCRIPCION: Incompatibilidad de tipos. No se puede llamar a un metodo, pasando un parametro de tipo “x” cuando la firma del metodo expecifica que espera un tipo 'y' debido a que no se encuentra en su arbol de herencia.|

- Cuando se llama a un constructor o metodo con el tipo de argumento incorrecto. | DESCRIPCION: El tipo del parametro 'x' no coincide con el tipo del parametro en la firma del metodo|

Para todos los nodos se chequea que las variables utilizadas esten declaradas, esto se hace en el Nodo Literal, identificando la variable y buscandola en la tabla de simbolos para comprobar su existencia en ella y para poder obtener su tipo correspondiente.

3.4.5. Casos de Pruebas Semanticos

En esta sección se detallan las pruebas realizadas para evaluar el funcionamiento del analizador semántico. Cada prueba se describe con su respectivo escenario, nombre del test y la salida esperada. Dichas pruebas abarcan una variedad de situaciones para garantizar la robustez y precisión del analizador en diferentes contextos y casos de uso.

A continuación, se presenta un resumen de las pruebas realizadas:

Escenario	Nombre Test	Salida esperada
Acceso	test_error_2: Verificar que haya error cuando se quiere hacer un acceso de array en una id que no es de tipo array	ERROR: SEMANTICO - SENTENCIAS "a" no es de tipo Array por lo que no se puede usar para un acceso
Acceso	test_error_3: Verificar que haya error cuando se quiere hacer un acceso de array en una id que no esta declarado	ERROR: SEMANTICO - SENTENCIAS El id "a" no esta declarado en el struct 'A' ni en el metodo 'constructor'
Acceso	test_error_4: Verificar que haya error cuando se quiere hacer un acceso de array en un indice	ERROR: SEMANTICO - SENTENCIAS La posicion para acceder debe ser de tipo Int

	que no es int	
Acceso	test_error_5: Verificar que haya error cuando se quiere hacer un acceso de array en un indice que no esta declarado	ERROR: SEMANTICO - SENTENCIAS El id "s" no esta declarado en el struct 'A' ni en el metodo 'constructor'
Acceso	test_error_6: Verificar que haya error cuando se quiere hacer un acceso en una id que no es de tipo struct	ERROR: SEMANTICO - SENTENCIAS "a" no es de tipo Array ni Struct por lo que no se puede usar usar para un acceso
Acceso	test_error_7: Verificar que haya error cuando se quiere hacer un acceso en una id que no esta declarado	ERROR: SEMANTICO - SENTENCIAS El id "a" no esta declarado en el struct 'A' ni en el metodo 'constructor'
Acceso	test_error_8: Verificar que haya error cuando se quiere hacer un acceso a un atributo que no existe en el struct	ERROR: SEMANTICO - SENTENCIAS Acceso incorrecto. No se puede acceder al atributo "s" ya que no existe en el struct "B"
Acceso	test_error_9: Verificar que haya error cuando se quiere hacer un acceso a un metodo que no existe en el struct	ERROR: SEMANTICO - SENTENCIAS No se puede llamar a un metodo que no existe. Debe definir el metodo 's' en el struct 'B' o heredarlo
Acceso	test_error_10: Verificar que haya error cuando se quiere hacer un acceso encadenado a un atributo que no existe en el struct	ERROR: SEMANTICO - SENTENCIAS Acceso incorrecto. No se puede acceder al atributo "y" ya que no existe en el struct "C"
Acceso	test_error_11: Verificar que haya error cuando se quiere hacer un acceso encadenado a un atributo que no existe en el struct	ERROR: SEMANTICO - SENTENCIAS El id "s" no esta declarado como atributo del struct 'B'.
Acceso	test_ok_1: Verificar que se forme bien el json cuando hay un acceso encadenado correcto	CORRECTO: SEMANTICO - SENTENCIAS
Acceso	test_ok_2: Verificar que se forme bien el json cuando hay un acceso encadenado correcto	CORRECTO: SEMANTICO - SENTENCIAS
Asignacion	test_ok_3: Verificar que se forme bien el json cuando se hace una asignacion algo del tipo Object = Struct	CORRECTO: SEMANTICO - SENTENCIAS
Asignacion	test_error_12: Verificar que haya error cuando se quiere hacer una asignacion con Arrays con tipos distintos	ERROR: SEMANTICO - SENTENCIAS Incompatibilidad de tipos. No se puede asignar un objeto de tipo Array Str a la variable 'array1' definida de tipo Array Int

Asignacion	test_error_13: Verificar que haya error cuando se quiere hacer una asignacion con tipos distintos	ERROR: SEMANTICO - SENTENCIAS Incompatibilidad de tipos. No se puede asignar un objeto de tipo Str a la variable definida de tipo Int
Asignacion	test_error_14: Verificar que haya error cuando se quiere hacer una asignacion con tipos no heredados	ERROR: SEMANTICO - SENTENCIAS Incompatibilidad de tipos. No se puede asignar un objeto de tipo A a la variable definida de tipo D.
Asignacion	test_error_15: Verificar que haya error cuando se quiere hacer una asignacion de un nil incorrecta	ERROR: SEMANTICO - SENTENCIAS Incompatibilidad de tipos. No se puede asignar 'nil' a una variable de tipo 'Str'.
Expresion Binaria	test_error_16: Verificar que haya error cuando se quiere hacer una operacion binaria logica con tipos distintos	ERROR: SEMANTICO - SENTENCIAS Incompatibilidad de tipos. No se puede realizar una operacion "==" entre un Bool y un Object
Expresion Binaria	test_error_17: Verificar que haya error cuando se quiere hacer una operacion aritmetica con tipos que no sean Int	ERROR: SEMANTICO - SENTENCIAS Incompatibilidad de tipos. No se puede realizar una operacion "-" entre un Int y un Object. Ambos deben ser enteros
While	test_error_18: Verificar que haya error cuando se quiere hacer que la expresion no sea bool	ERROR: SEMANTICO - SENTENCIAS La condicion del while debe ser de tipo Bool
If	test_error_19: Verificar que haya error cuando se quiere hacer que la expresion no sea bool	ERROR: SEMANTICO - SENTENCIAS La condicion del if debe ser de tipo Bool
Llamada Metodo	test_error_20: Verificar que haya error cuando se quiere usar un constructor no definido	ERROR: SEMANTICO - SENTENCIAS No se puede crear una instancia de 'R' porque no existe el struct. Primero debe crear el struct 'R'.
Llamada Metodo	test_error_21: Verificar que haya error cuando se quiere defiir el tamaño de un array con algo que no es de tipo Int	ERROR: SEMANTICO - SENTENCIAS Incompatibilidad de tipos. No se puede definir el tamaño de un array como 'x'.El tipo debe ser 'Int'
Llamada Metodo	test_error_22: Verificar que haya error cuando se llama a un constructor con la cantidad de parametros incorrecta	ERROR: SEMANTICO - SENTENCIAS Cantidad de argumentos incorrectos en la llamada a metodo 'constructor'. Para llamarlo debe pasar la cantidad de parametros corresponientes indicados en su firma.Se esperaban 0 parametros y llegaron 1.
Llamada Metodo	test_error_23: Verificar que haya error cuando se llama a un constructor con un parametro que no se encuentra en su arbol de herencia	ERROR: SEMANTICO - SENTENCIAS Incompatibilidad de tipos. No se puede llamar a un metodo, pasando un parametro de tipo Object cuando la firma del metodo expecifica que espera un tipo 'D' debido a que no se

		encuentra en su arbol de herencia.
Llamada Metodo	test_error_24: Verificar que haya error cuando se llama a un constructor con un parametro de un tipo incorrecto	ERROR: SEMANTICO - SENTENCIAS Llamada a constructor del struct 'R' incorrecta. El constructor espera un 'Int' y recibe el parametro 'g' de tipo 'D'
Llamada Metodo	test_error_25: Verificar que haya error cuando se llama a un metodo que no existe	ERROR: SEMANTICO - SENTENCIAS No se puede llamar a un metodo que no existe. Debe definir el metodo 'metodo' en el struct 'start' o heredarlo.
Llamada Metodo	test_error_26: Verificar que haya error cuando se llama a un metodo con la cantidad de argumentos incorrecta	ERROR: SEMANTICO - SENTENCIAS Cantidad de argumentos incorrectos en el metodo 'metodo'. Para llamarlo debe pasar la cantidad de parametros corresponientes indicados en su firma
Llamada Metodo	test_error_27: Verificar que haya error cuando se llama a un metodo con el tipo de argumento que no se encuentra en su arbol de herencia.	ERROR: SEMANTICO - SENTENCIAS Incompatibilidad de tipos. No se puede llamar a un metodo, pasando un parametro de tipo Object cuando la firma del metodo expecifica que espera un tipo 'Int' debido a que no se encuentra en su arbol de herencia.
Llamada Metodo	test_error_28: Verificar que haya error cuando se llama a un metodo con el tipo de argumento de tipo incorrecto.	ERROR: SEMANTICO - SENTENCIAS El tipo del parametro 'f' no coincide con el tipo del parametro en la firma del metodo 'metodo'
Llamada Metodo	test_error_29: Verificar que haya error cuando hace una llamada estatica a un metodo que no es st.	ERROR: SEMANTICO - SENTENCIAS No se puede realizar la llamada al metodo ya que este no es st. Debe definir el metodo 'm' en el struct 'A' como estatico.
Self	test_error_30: Verificar que haya error cuando se usa self en start	ERROR: SEMANTICO - SENTENCIAS No se puede realizar un self en start
Prueba General	test_ok_4: Verificar que se forme bien el json	CORRECTO: SEMANTICO - SENTENCIAS
Return	test_error_31: verificar error cuando la firma del metodo retorna void, pero el metodo retorna otra cosa	ERROR: SEMANTICO - SENTENCIAS El retorno del metodo '...' no puede ser '...' porque esta definido como void
Return	test_error_32: verificar que el retorno de la firma de un metodo, coincida con el tipo de salida esperado	ERROR: SEMANTICO - SENTENCIAS El retorno del metodo '...' no puede ser '...' porque esta definido como "..."
Return	test_ok_5: verifica que cuando el metodo devuelva el tipo declarado en su firma, pase sin problemas.	CORRECTO: SEMANTICO - SENTENCIAS

El conjunto completo de archivos de prueba en formato .ru se encuentra adjunto en la carpeta llamada test.

3.5. Generación de Codigo

Para la etapa final del compilador, el código fuente en lenguaje de alto nivel (TinyRu) se traduce a un código intermedio (MIPS) a través de un proceso de generación de código. A continuación se detalla cómo se produce la generación de código en el compilador, explicando las estructuras generadas mediante MIPS y su funcionamiento. Se describe principalmente la estructura de los registros de activación, el acceso a parámetros y variables, la creación y uso de objetos, y la invocación de métodos.

3.5.1. Registro de Activación

Un registro de activación es crucial para manejar las llamadas a métodos en el lenguaje de programación. Este registro almacena información esencial, como la dirección de retorno, las direcciones de memoria dinámicas y estáticas, los parámetros de los metodos, y las variables locales. En MIPS, el registro de activación se organiza de la siguiente manera:

- Dirección de retorno: Almacena la dirección a la que se debe retornar después de la ejecución de la función.
- Frame Pointer (FP): Apunta al inicio del registro de activación desde donde se llama.
- Espacio para parámetros: Los valores pasados a la función.
- Espacio para variables locales: Almacena las variables definidas dentro de la función.

Esta estructura permite que cada función tenga su propio contexto de ejecución, facilitando la recursividad y las llamadas anidadas.

3.5.2. Manejo de Metodos

3.5.2.1 Llamada a metodo

Antes de una llamada a un metodo, los parámetros se empujan en la pila, y se guardan la dirección de retorno y el Frame Pointer. Al inicio del metodo, se reserva espacio para las variables locales y se guarda el estado actual del programa:

```
addi $sp, $sp, -4
sw $a0, 0($sp)  # Empujar el primer parámetro en la pila
addi $sp, $sp, -8
sw $ra, 4($sp)  # Guardar la dirección de retorno
sw $fp, 0($sp)  # Guardar el Frame Pointer actual
```

Al final del metodo, se restaura el estado anterior:

```
move $sp, $fp  # Restaurar el puntero de pila
lw $fp, 0($sp) # Restaurar el Frame Pointer
lw $ra, 4($sp) # Restaurar la dirección de retorno
addi $sp, $sp, 8
jr $ra  # Retornar al llamador
```

3.5.2.2 Acceso de Parámetros y Variables

Para acceder a los parámetros y variables locales de una función en MIPS, se utilizan offsets desde el Frame Pointer (FP). Por ejemplo:

```
lw $t0, 8($fp)  # Cargar el primer parámetro en $t0
```

Esta convención permite un acceso eficiente y estructurado a los datos dentro de un metodo.

3.5.2.3 Retorno de Valores

El valor de retorno de una función se coloca en el registro $v0 antes de que la función termine. Luego, la función utiliza la instrucción `jr $ra` para retornar el valor al metodo llamador:

```
move $v0, $t2  # Colocar el valor de retorno en $v0
jr $ra         # Retornar al llamador
```

3.5.3. Manejo de Structs

3.5.3.1 Creación de Instancias de Structs

La creación de un struct en MIPS implica reservar espacio en el heap e inicializar sus atributos. Por ejemplo, para la clase `Fibonacci`, esto se hace utilizando la syscall para reservar memoria:

```
li $v0, 9       # Llamada a la syscall de sbrk
li $a0, 16          # Tamaño del objeto (segun la cantidad de
atributos)
syscall
move $s0, $v0   # Guardar la dirección base del objeto
```

Luego atributos del objeto se inicializan con los valores por defecto, con desplazamientos apropiados desde la base del objeto. Por ejemplo, si el primero atributo (que se accede segun su posicion) es de tipo entero se inicializa con 0:

```
sw $zero, 4($s0)  # Inicializar el primer (0(pos) * 4) atributo
```

3.5.3.2 Acceso a Métodos de Instancias

Los métodos de un struct se acceden mediante la Virtual Table (VT). La VT es una estructura que contiene punteros a las funciones miembro del objeto. Para llamar a un método, se carga la dirección de la VT y luego se carga la dirección del método específico:

```
lw $t1, 0($s0)  # Cargar la dirección de la VT
lw $t2, 4($t1)  # Cargar la dirección del método sucesion_fib
jalr $t2        # Llamar al método
```

3.5.3.3 Acceso a Atributos de Instancias

Para acceder a los atributos de un objeto instanciado, primero se necesita cargar la dirección base del objeto (por ejemplo $s1) y luego usar esta, junto con un desplazamiento para acceder al atributo. Este desplazamiento se calcula con la posicion del mismo multiplicando por 4:

```
lw $t0, 4($s1)  # Accede al primer atributo del objeto en $s1
```

3.5.4. Casos de Pruebas

En esta sección se detallan las pruebas realizadas para evaluar la generacion de codigo. Cada prueba se describe con su respectivo escenario, nombre del test y la salida esperada.

A continuación, se presenta un resumen de las pruebas realizadas:

Escenario	Nombre Test	Salida esperada
Generacion de Codigo	test_1: Verificar que se genere correctamente el archivo .asm	CORRECTO GENERACION DE CODIGO
Generacion de Codigo	test_2: Verificar que se genere correctamente el archivo .asm	CORRECTO GENERACION DE CODIGO
Generacion de Codigo	EXEC_BASICO01.ru: Verificar que se genere correctamente el archivo .asm para el codigo de entrada .ru propuesto por la catedra	CORRECTO GENERACION DE CODIGO
Generacion de Codigo	EXEC_CALLBASICO01.ru: Verificar que se genere correctamente el archivo .asm para el codigo de entrada .ru propuesto por la catedra	CORRECTO GENERACION DE CODIGO
Generacion de Codigo	EXEC_VARINSTBASICO01.ru: Verificar que se genere correctamente el archivo .asm para el codigo de entrada .ru propuesto por la catedra	CORRECTO GENERACION DE CODIGO
Generacion de Codigo	fibonacci: Verificar que se genere correctamente el archivo .asm	CORRECTO GENERACION DE CODIGO

4. ARQUITECTURA DEL COMPILADOR

A continuación se presenta una descripción general de las clases creadas y las relaciones entre ellas, a través de un diagrama UML.

4.1. Clases para el Analizador Lexico

El analizador léxico desarrollado consta de cuatro clases principales, cada una desempeñando un papel crucial en el proceso de análisis léxico del compilador. A

continuación, se presenta una descripción general de estas clases y las relaciones entre ellas.

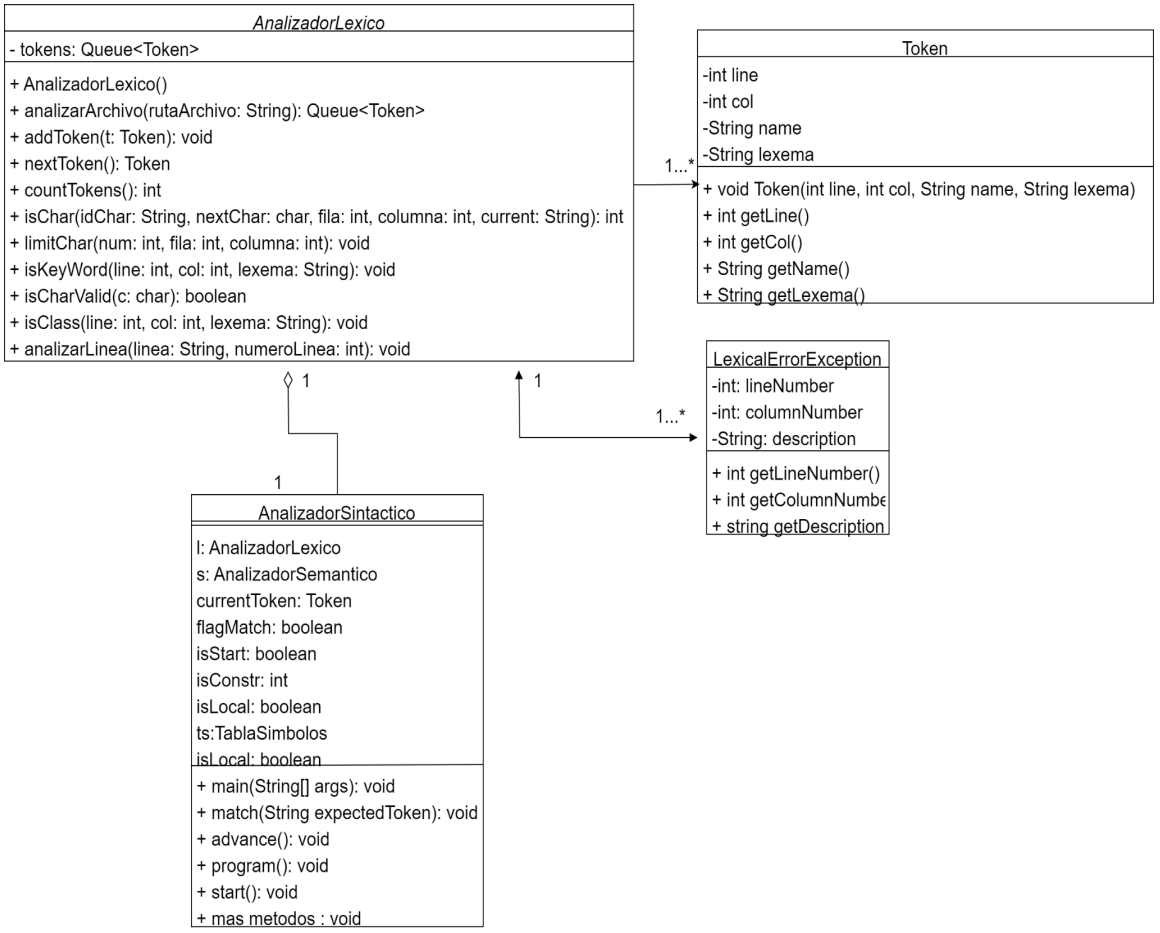

Figura 1: elaboración propia del UML del Analizador Léxico

4.1.1. Clase Token

La clase Token representa un token generado durante el análisis léxico. Cada token contiene información relevante, como el número de línea y columna donde se encuentra en el código fuente, el nombre del token y su lexema asociado. Esta clase proporciona 4 atributos:

- line: número de fila donde comienza el lexema
- col: número de columna donde comienza el lexema
- name: nombre del token
- lexema: lexema

y los siguientes métodos para acceder a los atributos: getLine(), getCol(), getName() y getLexema().

4.1.2. Clase AnalizadorLexico

La clase AnalizadorLexico es responsable de realizar el análisis léxico del código fuente. Utiliza un conjunto de reglas predefinidas en el manual para reconocer y generar tokens a partir de la secuencia de caracteres del archivo fuente. Además, gestiona cualquier error léxico que pueda surgir durante el análisis. Esta clase fue desarrollada en base a un manual para un lenguaje de programación reducido, propuesto por la cátedra. Cuenta con un

constructor que tiene un solo atributo: lista de tokens.Por otro lado, crea una estructura Queue , para guardar los tokens que serán consumidos por el executor. Contiene 10 métodos que serán explicados a continuación, sin embargo el más importante es analizarLinea.

- analizarArchivo(String rutaArchivo): se encarga de leer un archivo de texto línea por línea, analizando cada una de ellas y generando tokens correspondientes. Después de analizar todo el archivo, devuelve una cola (Queue) que contiene todos los tokens encontrados.
- addToken(Token t): se encarga de agregar token a la queue tokens.
- nextToken(): devuelve y elimina un token de la queue.
- countTokens():contar los tokens que hay en la cola.
- isChar(String idChar, char nextChar,int fila ,int columna, String current): se encarga de verificar que el caracter enviado, sea un token de tipo char.
- limitChar(int num, int fila, int columna): verifica si un string no excede la cantidad de caracteres posibles.
- isKeyWord(int line, int col, String lexema): verifica si un id es palabra reservada.
- isCharValid(char c): identifica que es el caracter enviado para poder formar los id
- isClass(int line, int col, String lexema): identifica si la clase formada es una ya predefinida.
- analizarLinea(String linea, int numeroLinea): es el método más importante, recibe la línea a analizar y el número de línea, lee caracter por caracter gracias a un for y dentro está integrado por un switch case, que es el encargado de analizar caso por caso, para poder clasificar el carácter de acuerdo al alfabeto definido, que más adelante se expondrá junto con el manual del lenguaje propuesto, en caso que no sea un token válido, el método es encargado de crear la excepción correspondiente.

4.2. Clases para el Analizador Sintáctico

El analizador sintáctico desarrollado consta de cinco clases principales, cada una al igual que el analizador léxico, representa un papel fundamental. A continuación se presenta una descripción general de la clases mencionadas y las relaciones entre ellas, a través de un diagrama UML

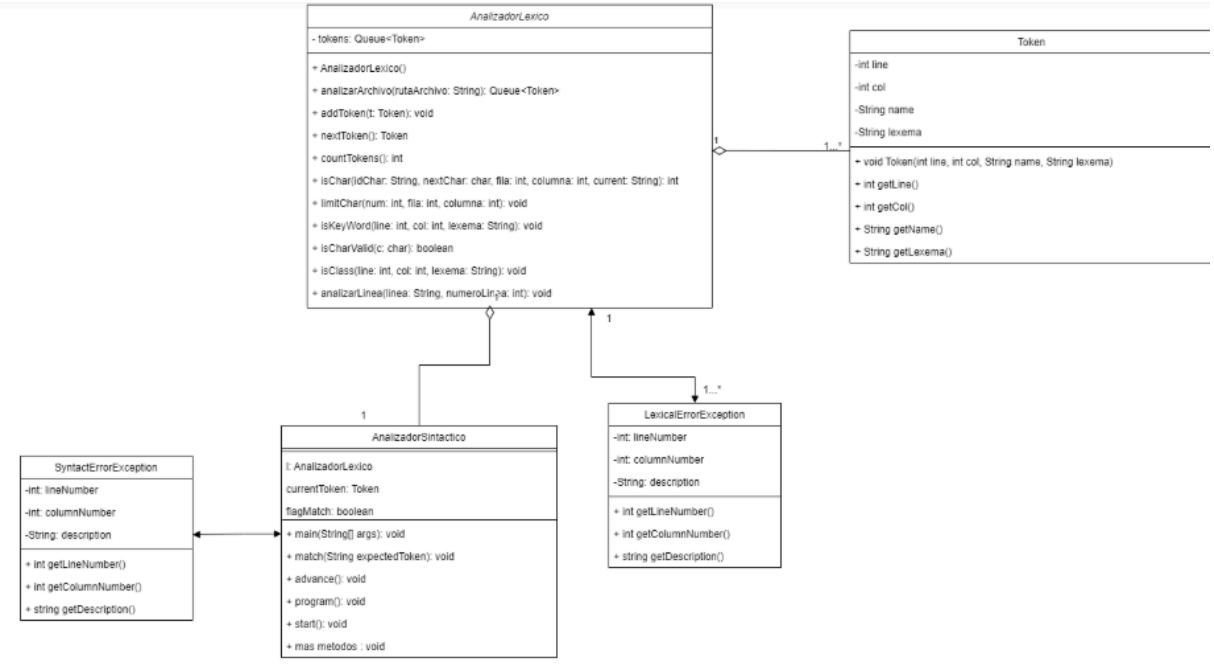

Figura 2: elaboración propia del UML del Analizador Sintactico

4.2.1. Clase AnalizadorSintactico

La clase Analizador Sintáctico es la clase principal que inicia el análisis léxico del compilador. En su método main(), inicializa el analizador léxico y comienza el análisis sintáctico desde el símbolo inicial, es decir analiza un archivo fuente especificado y escribe cualquier error léxico/sintáctico que pueda ocurrir durante el proceso. Esta clase, también es la encargada de llamar al analizador Semántico cuando termina de hacer el análisis sintáctico, además contiene programadas las acciones semánticas de manera tal que cuando se encuentre declaraciones de variables, métodos o clases, se agreguen a la tabla de símbolos en una primera pasada. Por último se encarga de llamar a la función encargada de generar el json, que será explicada mas adelante

- match(): verifica si el token actual coincide con el token esperado. Si no coincide, genera una excepción de error sintáctico.
- advance(): Avanza al siguiente token en la secuencia de tokens del analizador léxico.
- Métodos para cada regla de la gramática: Cada método en la clase corresponde a una regla de la gramática del lenguaje que se está analizando. Estos métodos se encargan de realizar el análisis sintáctico/semántico para cada parte del programa, verificando si la secuencia de tokens se ajusta a la estructura esperada según la gramática del lenguaje (program, start, struc, impl, etc). Además guardan los id correspondientes a variables, métodos y clases y también información relevante sobre estas estructuras.

4.3. Clases para el Analizador Semántico - Declaraciones

El analizador semántico desarrollado consta de ocho clases principales y 6 clases propias para generar la tabla de símbolos, cada una al igual que el analizador léxico y el analizador sintáctico, representa un papel fundamental. A continuación se presenta una descripción general de las clases mencionadas y las relaciones entre ellas, a través de un diagrama UML.

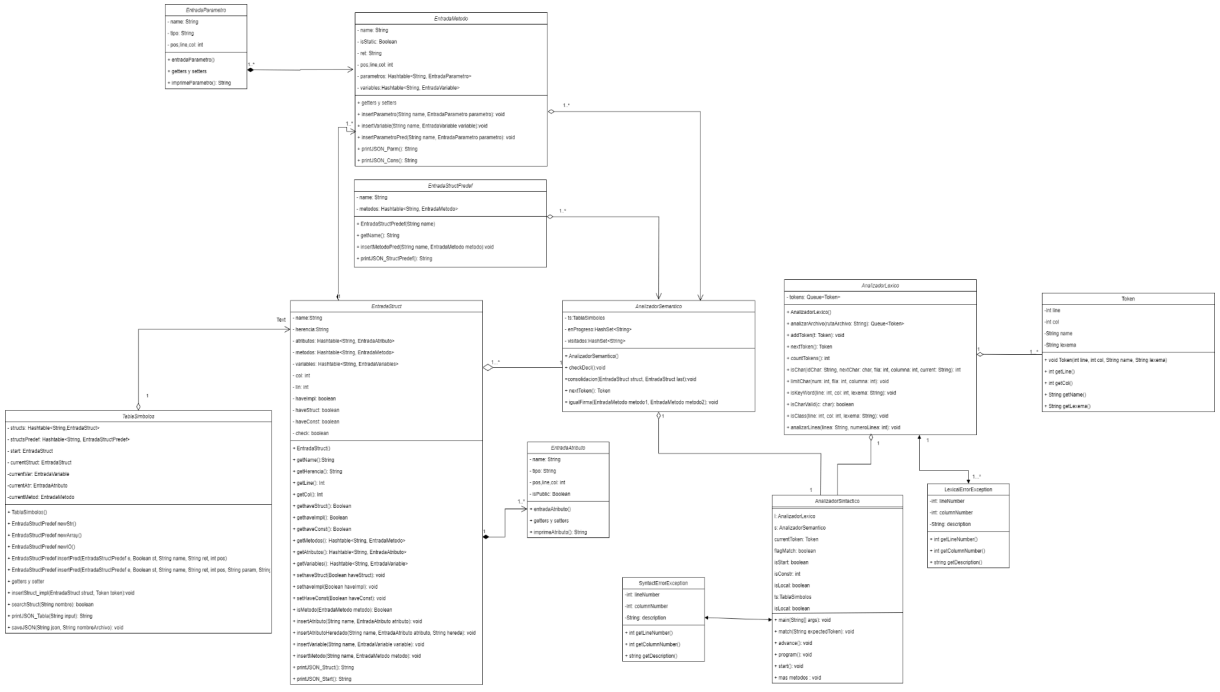

Figura 3: elaboración propia del UML del Analizador Semantico

4.3.1. Clase AnalizadorSemántico

La clase AnalizadorSemantico es una parte esencial de nuestro compilador, que se encarga de realizar el análisis semántico del programa fuente escrito en tinyRu. Esta clase, trabaja en conjunto con una tabla de símbolos, representada por la clase TablaSimbolos, que contiene información sobre las estructuras de datos, variables, métodos, etc., definidas en el programa.

La clase AnalizadorSemantico contiene métodos específicos para realizar diferentes aspectos del análisis semántico, como la verificación de declaraciones, la consolidación de atributos, la resolución de nombres, etc. También maneja la detección de errores semánticos y lanza excepciones específicas (SemantErrorException) cuando encuentra un problema en el código que viola las reglas semánticas del lenguaje. A continuación se mencionan los métodos que integran la clase.

- checkDecl(): este método se encarga de realizar el chequeo de declaraciones. Itera sobre todas las entradas de la tabla de símbolos y verifica que cada estructura tenga una declaración completa, incluyendo la definición de la estructura (struct), la implementación (impl) y un constructor.
  Acciones importantes:
  - Verifica que cada estructura tenga una declaración de estructura (struct).
  - Verifica que cada estructura tenga una implementación (impl).
  - Verifica que cada estructura tenga un constructor definido.
  - Llama al método consolidacion para consolidar la tabla de símbolos.

- consolidacion(EntradaStruct struct, EntradaStruct last): se encarga de consolidar los atributos y métodos heredados
  Acciones importantes:
  - Verifica la presencia de herencia circular.
  - Verifica la existencia de la estructura heredada.
  - Inserta los atributos heredados.
  - Inserta los métodos heredados.
  - Verifica que los métodos heredados redefinidos tengan la misma firma.
  -
- igualFirma(EntradaMetodo metodo1, EntradaMetodo metodo2): este método verifica si dos métodos tienen la misma firma(signatura)
  Acciones importantes:
  -Verifica que ambos métodos sean estáticos si se heredan.
  -Verifica que ambos métodos tengan el mismo tipo de retorno.
  -Verifica que los métodos tengan la misma cantidad de parámetros.
  -Verifica que los parámetros tengan el mismo tipo y nombre en la misma posición.

4.3.2. Clases para la Tabla de Simbolos

4.3.2.1 Clase TablaSimbolos

La clase TablaSimbolos es como un diccionario gigante donde guardamos todas las cosas importantes sobre las estructuras y métodos de nuestro programa. Por un lado, tenemos un lugar para nuestras estructuras definidas por el usuario y, por otro lado, otro lugar para las estructuras predefinidas del lenguaje, como Object, IO.

- Métodos para crear clases predefinidas:
  -newStr(), newArray(), newIO(): Estos métodos crean y devuelven instancias de EntradaStructPredef que representan las clases predefinidas Str, Array e IO respectivamente. Cada uno de estos métodos agrega métodos predefinidos a estas clases como length(), concat(), out_str(), out_int(), etc. Para las demas clases predefinidas (Int,Char,Object,Bool), no se requeire un metodo para crearlas, ya que no tienen metodos predefinidos.

- Métodos para insertar estructuras:
  -insertStruct_struct(EntradaStruct struct, Token token): Inserta una estructura definida por el usuario en la tabla structs. Verifica si ya existe una estructura con el mismo nombre y si la estructura tiene una implementación pero no una declaración, o viceversa.
  -insertStruct_impl(EntradaStruct struct, Token token): Inserta una implementación de una estructura en la tabla structs. Realiza verificaciones similares a insertStruct_struct().

- Métodos para obtener información:
  -getCurrentStruct(), getCurrentVar(), getCurrentAtr(), getCurrentMetod(): Métodos que devuelven la estructura, variable, atributo y método actualmente en proceso.
  -getTableStructs(), getStructsPred(), getStruct(String nombre): Métodos para obtener la tabla de estructuras definidas por el usuario, la tabla de estructuras predefinidas y una estructura específica por su nombre.

- Funciones auxiliares:
  -printJSON_Tabla(String input): Genera una representación JSON de toda la tabla de símbolos y la estructura start.
  -saveJSON(String json, String nombreArchivo): Guarda el JSON generado en un archivo con el nombre especificado.

4.3.2.2 Clase EntradaStruct

Se utiliza para representar las estructuras (o clases) en la tabla de símbolos.

- Getters:
  Métodos como getName(), getHerencia(), getLine(), getCol(), gethaveStruct(), gethaveImpl(), gethaveConst(), getMetodos(), getAtributos() y getVariables() son getters que devuelven los valores de los atributos de la estructura.

- Setters:
  Métodos como setHerencia(), sethaveStruct(), sethaveImpl() y setHaveConst() son setters que permiten modificar los valores de ciertos atributos de la estructura.

- Funciones:
  -public boolean isMetodo(EntradaMetodo metodo): este método comprueba si un método dado está presente en la lista de métodos del struct.
  -public void insertAtributo(String name, EntradaAtributo atributo): Inserta un nuevo atributo en la tabla hash de atributos del struct.

-public void insertAtributoHeredado(String name, EntradaAtributo atributo, String hereda): inserta un atributo heredado en la tabla hash de atributos de la estructura.
-public void insertVariable(String name, EntradaVariable variable): Inserta una nueva variable en la tabla hash de variables de la estructura.
-public void insertMetodo(String name, EntradaMetodo metodo): Inserta un nuevo método en la tabla hash de métodos de la estructura.
-public String printJSON_Struct(): Genera un JSON que representa la estructura y sus atributos y métodos.
-public String printJSON_Start(): Genera un JSON que representa las variables de inicio de la estructura

4.3.2.3 Clase EntradaStructPredef

La clase EntradaStructPredef representa los struct predefinidos del lenguaje. Estos, ya están definidos en el sistema y no se pueden modificar. La clase proporciona métodos para insertar métodos predefinidos y generar una representación JSON de estos métodos.

- Getters:
  -public String getName(): Devuelve el nombre de la estructura predefinida.

- Funciones:
  -public void insertMetodo(String name, EntradaMetodo metodo, Token token): Inserta un nuevo método predefinido en la tabla hash de métodos. Si ya existe un método con el mismo nombre, lanza una excepción SemantErrorException.
  -public void insertMetodoPred(String name, EntradaMetodo metodo): Inserta un nuevo método predefinido en la tabla hash de métodos sin realizar ninguna verificación.
  -public String printJSON_StructPredef(): Genera un JSON que representa los métodos predefinidos de la estructura. Si no hay métodos predefinidos, devuelve un JSON vacío. Este método ordena los métodos por su posición y los convierte en cadenas JSON.

4.3.2.4 Clase EntradaMetodo

La clase EntradaMetodo se utiliza para representar los métodos en el contexto del análisis semántico de un programa. Proporciona funcionalidades para definir métodos, gestionar parámetros y variables locales, y generar representaciones JSON de los métodos y sus elementos asociados. Esta clase posee 2 constructores, uno para los métodos y otro para tratar el caso especial del constructor.

- Getters:
  -Métodos para obtener los valores de los atributos nombre, isStatic, ret, pos, line y col, así como las tablas de parámetros y variables.

- Setters:
  -public void setRet(String ret): Establece el tipo de retorno del método.
  -public void setPos(int pos): Establece la posición del método.

- Funciones:
  -public void insertParametro(String name, EntradaParametro parametro): Inserta un parámetro en la tabla de parámetros del método.
  -public void insertVariable(String name, EntradaVariable variable): Inserta una variable local en la tabla de variables del método.
  -public void insertParametroPred(String name, EntradaParametro parametro): Inserta un parámetro predefinido en la tabla de parámetros del método.
  -public String printJSON_Parm(): Genera una representación JSON de los parámetros y variables locales del método.

  -public String printJSON_Const(): Genera una representación JSON de los parámetros del constructor del método.

4.3.2.5 Clase EntradaAtributo

La clase EntradaAtributo representa un atributo dentro de un struct o clase en un contexto semántico.

- getType(): Devuelve el tipo de dato del atributo.
-getPublic(): Devuelve true si el atributo es público, false en caso contrario.
-getPos(): Devuelve la posición del atributo.
-getName(): Devuelve el nombre del atributo.
-getLine(): Devuelve la línea donde se declara el atributo.
-getCol(): Devuelve la columna donde se declara el atributo.
-setPos(int pos): Establece la posición del atributo.
-imprimeAtributo(): Devuelve una representación en formato JSON del atributo.

4.3.2.6 Clase EntradaParametro

La clase EntradaParametro modela un parámetro de un método en un contexto semántico.

-getPos(): Devuelve la posición del parámetro.
-getName(): Devuelve el nombre del parámetro.
-getLine(): Devuelve la línea donde se declara el parámetro.
-getCol(): Devuelve la columna donde se declara el parámetro.
-getType(): Devuelve el tipo de dato del parámetro.
-imprimeParametro(): Devuelve una representación en formato JSON del parámetro.

4.3.2.7 Clase EntradaVariable

La clase EntradaVariable modela una variable dentro del contexto semántico de un programa.

-getName(): Devuelve el nombre de la variable.
-getType(): Devuelve el tipo de la variable.
-getPos(): Devuelve la posición de la variable.
-getLine(): Devuelve la línea donde se declara la variable.

-getCol(): Devuelve la columna donde se declara la variable.
-imprimeVar(): Genera una representación en formato JSON de la variable para su uso fuera de métodos.
-imprimeVarMet(): Genera una representación en formato JSON de la variable para su uso dentro de métodos.

4.4. Clases para el Analizador Semántico - Sentencias

El analizador semántico desarrollado consta de ocho clases principales y 6 clases propias para generar la tabla de símbolos, cada una al igual que el analizador léxico y el analizador sintáctico, representa un papel fundamental. A continuación se presenta una descripción general de las clases mencionadas y las relaciones entre ellas, a través de un diagrama UML.

IMAGEN

Figura 4: elaboración propia del UML del Analizador Semantico

4.4.1. Clases para el AST

4.4.1.1 Clase AST

La clase AST se encarga de manejar y manipular estructuras de datos relacionadas con un árbol de sintaxis abstracta. A continuación, se explica cada uno de sus métodos: public NodoStruct getCurrentStruct(): Retorna la estructura actual (currentStruct).

Getters y Setters:

-getProfundidad(): Retorna la pila de profundidad (profundidad).
-getStructs(): Retorna el mapa de estructuras (structs).
-setCurrentStruct(): Establece la estructura actual.
-setCurrentMetodo(): Establece el método actual.

Métodos Funcionales:

-insertStruct(String name, NodoStruct nodo): Inserta una nueva estructura en el mapa structs con el nombre dado y el objeto NodoStruct.
- printJSON_Arbol(String input): Genera una representación en formato JSON del árbol de sintaxis.
-saveJSON(String json, String nombreArchivo): Guarda el JSON generado en un archivo.

4.4.1.2 Clase Nodo

Es una representación del nodo en un árbol de sintaxis abstracta (AST). Cada nodo puede contener información sobre su posición en el código fuente (línea y columna), su tipo, su valor, su nombre y una lista de sentencias (NodoLiteral) asociadas. A continuación, se explica cada método:

Getters y setters:

-getCol(): Retorna la columna del nodo.
-getLine(): Retorna la línea del nodo.

-getNodeType(): Retorna el tipo del nodo.
-getName(): Retorna el nombre del nodo.
-getValue(): Retorna el valor del nodo.
-getParent(): Retorna el nombre del nodo padre. -setNodeType(String nodeType): Establece el tipo del nodo.
-setParent(String parent): Establece el nombre del nodo padre.

Métodos Funcionales:
-insertSentencia(NodoLiteral sentencia): Añade una sentencia (NodoLiteral) a la lista de sentencias del nodo.
-checkTypes(TablaSimbolos ts): Un método para verificar los tipos en el nodo utilizando una tabla de símbolos (TablaSimbolos).

La implementación específica de este método puede variar según el tipo de nodo y el análisis de tipos requerido. En este código, el método devuelve true siempre y cuando no ejecute ninguna excepción, indicando que la verificación de tipos siempre es exitosa. En una implementación completa, este método debería contener la lógica para comprobar y establecer los tipos correspondientes.

4.4.1.3 Clase NodoAcceso

La clase NodoAcceso representa un nodo del árbol de sintaxis abstracta que maneja el acceso a un atributo o método de un objeto. Este nodo contiene dos variables: el nodo izquierdo (nodoI), que representa el objeto o instancia de un struct, y el nodo derecho (nodoD), que representa el atributo o método al que se accede en el nodo izquierdo. A continuación, se explican sus métodos principales:

-printSentencia(): genera una representación en formato JSON del nodo NodoAcceso.

-checkTypes(ts): verifica la validez de los tipos en el acceso a los nodos. La verificación incluye varios pasos:

1) Configuración del Nodo Izquierdo (nodoI): Si nodoI tiene un nodo padre, se establece el nodo padre. Se llama al método checkTypes en nodoI para verificar sus tipos.
2) Configuración del Nodo Derecho (nodoD): Se establece el tipo del nodo izquierdo (nodoI) como el nodo padre de nodoD. Verificación del Tipo de Nodo Izquierdo:
3) Verificaciones varias de tipo:
    - Si nodoI es de tipo struct y ese struct existe en la tabla de símbolos (ts), se realizan verificaciones adicionales.
    - Si nodoD es un NodoLiteral, se verifica que sea un atributo válido del struct correspondiente.
    - Si el atributo no existe, se lanza una excepción semántica.
    - Se obtiene el tipo del atributo y se establece en nodoD.
    - Si nodoD no es un NodoLiteral, se llama a checkTypes en nodoD para verificar sus tipos.
4) Verificación Adicional para Tipos Específicos:
    - Si nodoI es de tipo IO o Str, se realizan verificaciones adicionales para asegurar que el acceso sea válido.
    - Si el acceso es incorrecto, se lanza una excepción semántica.
5) Asignación del Tipo al NodoAcceso

4.4.1.4 Clase NodoAccesoArray

Se usa para representar el acceso a un elemento de un array en el árbol de sintaxis abstracta (AST). Este nodo es útil para manejar las expresiones que involucran la indexación de arrays. A continuación explicamos los metodos

-printSentencia(): este método genera una representación en formato JSON del nodo NodoAccesoArray para propósitos de depuración. La salida JSON incluye:

-checkTypes(): este método verifica la validez de los tipos en la expresión de acceso al array y establece los tipos correspondientes en el nodo. La verificación incluye varios pasos:

1) Verificación del Array: Llama a checkTypes en el nodo del array para verificar sus tipos. Comprobación de Tipo Array:
2) Divide el tipo del nodo del array para comprobar si es un tipo array. Si el tipo del nodo no es array, se lanza una excepción semántica.
3) Verificación del Índice: Llama a checkTypes en la expresión del índice para verificar sus tipos. Comprueba que el tipo de la expresión del índice sea entero.Si el tipo de la expresión no es entero, se lanza una excepción semántica.
4) Asignación del Tipo: Establece el tipo del NodoAccesoArray al tipo de los elementos del array. Establece el nodo padre adecuado

4.4.1.5 Clase NodoAsignacion

La clase NodoAsignacion representa una asignación en el árbol de sintaxis abstracta (AST). Este nodo maneja las asignaciones en el código fuente, donde un valor o expresión (nodo derecho) se asigna a una variable o campo (nodo izquierdo). A continuación se especifican los metodos:

-printSentencia():este método genera una representación en formato JSON del nodo NodoAsignacion para propósitos de depuración. La salida JSON incluye:

-checkTypes(): este método verifica la validez de los tipos en la asignación y asegura que los tipos de los nodos izquierdo y derecho sean compatibles. La verificación incluye varios pasos:

1) Chequeo del Nodo Izquierdo (nodoI): Se llama al método checkTypes en el nodo izquierdo para verificar su tipo.
2) Chequeo del Nodo Derecho (nodoD):Se llama al método checkTypes en el nodo derecho para verificar su tipo.
3) Verificación de Compatibilidad de Tipos: Se obtienen los tipos del nodo izquierdo (typeNI) y del nodo derecho (typeND).
    - Si el nodo derecho es de tipo nil, se verifica que el nodo izquierdo no sea de tipo Int, Str, Bool, o Char, ya que nil solo puede asignarse a objetos o clases definidas.
    - Si los tipos de los nodos izquierdo y derecho no coinciden:
    - Si el nodo izquierdo es de tipo Object, el nodo derecho no debe ser de tipos primitivos (Int, Str, Bool, Char) o arrays.
    - Si el nodo izquierdo no es de tipo Object, se verifica la herencia entre los tipos. Si no hay una relación de herencia adecuada, se lanza una excepción semántica.
4) Asignación del Tipo: Se establece el tipo del nodo NodoAsignacion al tipo del nodo izquierdo.

4.4.1.6 Clase NodoBloque

La clase NodoBloque y representa un bloque de código en un árbol de sintaxis abstracta (AST), estos bloques se encuentran dentro de otras sentencias. Esta clase maneja una lista

de sentencias (NodoLiteral) que están contenidas dentro del bloque. A continuación se explica cada parte de la clase:

Getters

getSentencias(): Devuelve la lista de sentencias del bloque.

Metodos Funcionales

-insertSentencia(NodoLiteral sentencia): agrega una sentencia a la lista de sentencias del bloque.

-printSentencia(String space): genera una representación en formato JSON del nodo NodoBloque para propósitos de depuración. La salida JSON incluye:

-checkTypes(TablaSimbolos ts):este método verifica los tipos de todas las sentencias contenidas en el bloque. La verificación incluye los siguientes pasos:

1) Chequeo de Sentencias: Recorre la lista de sentencias y llama al método checkTypes de cada sentencia para asegurarse de que los tipos son válidos según la tabla de símbolos (TablaSimbolos).
2) Establecimiento del Tipo: Después de verificar todas las sentencias, establece el tipo del nodo bloque a null, indicando que el bloque en sí no tiene un tipo específico.
3) Retorno: Devuelve true para indicar que la verificación de tipos se completó con éxito 2

4.4.1.7 Clase NodoExpresion

NodoExpresion representa una expresión dentro de un árbol de sintaxis abstracta (AST). La clase incluye atributos, un constructor y métodos para manejar y verificar tipos de expresiones que serán explicados a continuación:

Metodos Funcionles.

checkTypes(ts):verifica los tipos dentro del nodo NodoExpresion. Si el nodo representa una sentencia Retorno, realiza varias comprobaciones:

- Si el retorno es void, asegura que la firma del método también sea void. Si no, lanza una excepción.
- Si hay una expresión de retorno, verifica que el tipo de la expresión coincida con el tipo de retorno esperado del método. Si no coincide, realiza comprobaciones adicionales, especialmente para tipos de objetos y herencias, y lanza excepciones en caso de incompatibilidades.
- Para otros tipos de expresiones, simplemente verifica el tipo de la expresión anidada y lo establece en el nodo actual.

4.4.1.8 Clase NodoExpBin

Esta clase maneja operaciones binarias, como aritméticas y lógicas, entre dos nodos.A continuación se explican los métodos de esta clase:

- printSentencia():genera una representación en formato de cadena de la expresión binaria para propósitos de impresión o depuración.Estructura la salida mostrando el tipo del nodo, el operador, y las representaciones de los nodos izquierdo y derecho.
- checkTypes(ts): verifica los tipos de los nodos izquierdo y derecho, asegurando que sean compatibles con el operador binario. -Para operadores lógicos (||, &&, ==, !=), verifica que ambos nodos sean del mismo tipo y establece el tipo del nodo a Bool.

- Para operadores aritméticos y de comparación (<, >, <=, >=, +, -, *, /), asegura que ambos nodos sean enteros (Int). Establece el tipo del nodo a Bool para comparaciones y a Int para operaciones aritméticas.
- Lanza una excepción SemantErrorException si los tipos de los nodos no son compatibles con el operador, proporcionando detalles sobre la incompatibilidad.

4.4.1.9 Clase NodoExpUn

Representa una expresión unaria en un árbol de sintaxis abstracta (AST). Esta clase maneja operaciones unarias, como la negación lógica (!) y los operadores de incremento/decremento (++, --).

- printSentencia(): genera una representación en formato de cadena de la expresión unaria para propósitos de impresión o depuración.La salida muestra el tipo del nodo, el operador, y la representación en cadena de la expresión anidada
- checkTypes():verifica el tipo de la expresión unaria para asegurar que el operador sea compatible con el tipo de la expresión.Primero, llama a checkTypes en la expresión anidada para asegurar que su tipo sea determinado y válido.Luego, maneja dos casos principales:
  - Operador ! (Negación lógica):asegura que el tipo de la expresión sea Bool. Si no lo es, lanza una excepción.Establece el tipo del nodo resultante a Bool.
  - Operadores ++ y -- (Incremento y Decremento):asegura que el tipo de la expresión sea Int. Si no lo es, lanza una excepción. Establece el tipo del nodo resultante a Int.

4.4.1.10 Clase NodoIf

La clase maneja la expresión condicional y las sentencias que se ejecutan si la condición es verdadera, así como las sentencias opcionales de un bloque else, pero para esto último se creo un nodo Else, q se inserta en el nodoIf.

Getters y Setters:

- setNodoElse():establece el bloque else para el if.
- getSentencias():devuelve la lista de sentencias dentro del bloque if.

Métodos Funcionales:

- insertSentencia():añade una sentencia a la lista de sentencias del bloque if.
- printSentencia():genera una representación en formato JSON de la estructura if para propósitos de impresión o depuración.La salida muestra el nodo if, la expresión condicional, y las sentencias anidadas. También incluye el bloque else si está presente.
- checkTypes(): verifica que todos los tipos dentro del nodo if sean válidos. Llama a checkTypes en la expresión condicional y asegura que sea de tipo Bool. Luego, verifica los tipos de cada sentencia en el bloque if y el bloque else. Si la expresión condicional no es Bool, lanza una excepción SemantErrorException detallando la incompatibilidad. Finalmente, establece el tipo del nodo a null y retorna true si todos los chequeos fueron exitosos. 2

4.4.1.11 Clase NodoElse

Este nodo maneja las sentencias que se ejecutan cuando la condición de un nodo if asociado es falsa.

- insertSentencia():permite añadir una nueva sentencia al bloque else. Esto facilita la construcción dinámica de bloques else con múltiples sentencias.
- printSentencia():este método devuelve una cadena en formato JSON que representa el nodo else. La salida incluye las sentencias del bloque else. Esto es útil para depuración y visualización del árbol de sintaxis abstracta.
- checkTypes():este método verifica que todos los tipos dentro del nodo else sean válidos. Recorre cada sentencia en la lista de sentencias del bloque else y llama a checkTypes en cada una para verificar su validez. Establece el tipo del nodo a null y retorna true si todos los chequeos fueron exitos

4.4.1.12 Clase NodoLiteral

Los nodos literales pueden incluir tipos de datos básicos como enteros, cadenas, caracteres, booleanos, así como identificadores y otros elementos específicos del lenguaje. La clase maneja la validación de tipos y proporciona métodos para representar el nodo en formato JSON.A continuación se explica cada metodo:

- printSentencia(String space):Genera una representación en formato JSON del nodo literal. Incluye el nombre, tipo y valor del nodo.
- checkTypes(TablaSimbolos ts):verifica la validez del tipo del nodo literal en el contexto de una tabla de símbolos (ts). Si el valor del nodo es "st" y su nombre está en la tabla de estructuras, es válido. Para literales predefinidos como enteros, cadenas, caracteres, booleanos, punto y coma, self e IO, la verificación se omite porque ya tienen tipos definidos.

En el caso de la estructura start, el método verifica si el identificador está declarado como variable de start o como atributo del struct del que se accede. Para otros structs, se comprueba si el identificador está declarado como variable del método actual, parámetro del método o atributo del struct, lanzando una excepción si no está declarado en ninguna de estas ubicaciones. Además, el método asegura que no se utilice self en la estructura start.

4.4.1.13 Clase NodoLlamadaMetodo

Representa una llamada a un método en el árbol sintáctico del lenguaje y realiza verificaciones detalladas para asegurar que la llamada sea semánticamente correcta, lanzando excepciones en caso de encontrar errores. Posee los siguientes metodos:

- insertArgumento(NodoLiteral argumento):agrega un argumento a la lista de argumentos.
- printSentencia(String space):genera una representación JSON de la llamada al método, incluyendo su tipo, nombre y argumentos.
- checkTypes(ts): el método checkTypes en la clase NodoLlamadaMetodo valida la corrección de tipos en llamadas a métodos, considerando diferentes casos como constructores, métodos normales y accesos especiales. Verifica la existencia del método y la coincidencia en la cantidad y tipos de argumentos según la firma del método.

Además, maneja casos especiales como llamadas a métodos estáticos y en estructuras predefinidas. Si alguna verificación falla, se lanza una excepción con un mensaje detallado del error.

4.4.1.14 Clase NodoWhile

Representa la estructura de un bucle while en un árbol de análisis sintáctico abstracto (AST) y proporciona funcionalidades para imprimir la estructura del bucle en formato JSON y

verificar la corrección de tipos en su expresión condicional y en las sentencias dentro del bucle. Sus métodos son:

- printSentencia(): genera una representación en JSON del bucle, incluyendo la expresión condicional y las sentencias dentro del bucle.
- insertSentencia(): inserta las sentencias en la pila sentencias.
- checkTypes(ts): realiza la validación de tipos, asegurándose de que la expresión condicional sea de tipo booleano y luego verifica los tipos de las sentencias dentro del bucle. Si alguna verificación de tipo falla, se lanza una excepción con un mensaje detallado del error.

4.4.1.15 Clase NodoMetodo

Corresponde a la definición de un método en el código fuente. Almacena el nombre del método y una lista de las sentencias que componen su cuerpo. Sus metodos son:

- insertSentencia(NodoLiteral sentencia): Inserta una sentencia en la lista de sentencias del método.
- printNodoMet(): Genera una representación en formato JSON del cuerpo del método, incluyendo las sentencias que lo componen. Este método es utilizado para imprimir la estructura del método cuando se genera la salida del análisis sintáctico abstracto

4.4.1.16 Clase NodoStruct

Representa un nodo del árbol abstracto de sintaxis (AST) para la definición de estructuras en un lenguaje de programación. Sus metodos son:

- insertMetodo(String name, NodoMetodo nodo): Inserta un método en el mapa de métodos de la estructura.
- insertSentencia(NodoLiteral sentencia): Inserta una sentencia en la lista de sentencias del bloque start de la estructura.
- printNodoStruct(): Genera una representación en formato JSON de los métodos definidos dentro de la estructura, incluyendo las sentencias que componen cada método. Este método es utilizado para imprimir la estructura de los métodos cuando se genera la salida del análisis sintáctico abstracto.
- printNodoStart(): Genera una representación en formato JSON de las sentencias dentro del bloque start de la estructura. Este método es utilizado para imprimir la estructura del bloque start cuando se genera la salida del análisis sintáctico abstracto.

4.5. Clase para la Generacion de Código

Para parte de generacion de codigo intermedio en MIPS se creo una sola clase `CodeGenerator` en Java que se utiliza para generar código en lenguaje ensamblador MIPS. Esta funciona a partir de la Tabla de Símbolos (clase TablaSimbolos) y el Árbol de Sintaxis Abstracta (clase AST). A continuación, se explica cada una de las secciones y métodos del código en detalle:

4.5.1. Clase CodeGenerator

La clase CodeGenerator genera código MIPS a partir de una tabla de símbolos y el árbol de sintaxis abstracta, creando las secciones de datos y texto, y asegurando la correcta representación de estructuras, métodos y variables en el código resultante.

Metodos:

- generateCode: Genera el código ensamblador completo recorriendo la tabla de símbolos y el AST. Llama a `generateData`, `generateText` y `generatePred` para generar las secciones correspondientes del código. Imprime y retorna el código generado.
- generateData: Genera la sección de datos del código ensamblador. Recorre las estructuras predefinidas (`EntradaStructPredef`) y las estructuras definidas por el usuario (`EntradaStruct`) en la tabla de símbolos.Para cada estructura, genera su tabla de métodos virtuales (`vtable`).
- generateText: Genera la sección de texto del código ensamblador. Recorre los nodos del AST para generar el código MIPS correspondiente a cada uno. Genera el código para el método `main` y los constructores y métodos de cada estructura.
- generatePred: Genera el código para los métodos predefinidos de entrada/salida en MIPS. Incluye métodos para imprimir enteros, cadenas, booleanos y caracteres.
- generateNodeCode`** (método de `NodoLiteral` y otros nodos): Genera el código MIPS para un nodo específico del AST. La implementación específica de este método dependerá de cada clase de nodo en el AST.

4.6. Clases Exception para Errores

Las clases LexicalErrorException, SyntacErrorException y SematicErrorException son una excepción personalizada que se lanza cuando se produce un error lexico, sintactico o semantico respectivamente, durante el análisis del código fuente. Estas excepciones contiene información detallada sobre el error, incluyendo el número de línea y columna donde ocurrió y una descripción del error.

- lineNumber: número de fila donde se produce el error
- columNumber: número de columna donde se produce el error
- description: descripción del error

4.7. UML

Importante: Anteriormente se dejan ilustraciones de los UML del proyecto, sin embargo dentro de la carpeta de entrega, se dejará la imagen del UML definitivo en tamaño real, para poder visualizar mejor las clases mencionadas.

5. COMPILACIÓN Y USO DEL CÓDIGO FUENTE

5.1 Requisitos previos

Para llevar a cabo el proceso de compilación y ejecución descrito en las instrucciones que siguen, es importante asegurarse de cumplit los siguientes requisitos previos:

IntelliJ IDEA instalado: Asegúrarse de tener IntelliJ IDEA instalado en su computadora. Puede descargarse e instalarse desde el sitio web oficial de JetBrains.

Proyecto Java: Tener el proyecto Java configurado en IntelliJ IDEA. Este se encuentra adjunto en la entrega.

Antes de ejecutar el programa, asegurarse de tener instalado Java Runtime Environment (JRE). El programa `tinyRU.jar` requiere que Java esté instalado en su sistema para ejecutarse.

5.2 Compilación

A continuación se muestra un instructivo para compilar y crear el archivo '.jar':

1. Abrir IntelliJ IDEA: Iniciar IntelliJ IDEA y abrir el proyecto java "tinyRU".
2. Estructurar el Proyecto: Ir al menú "File" y seleccionar "Project Structure".

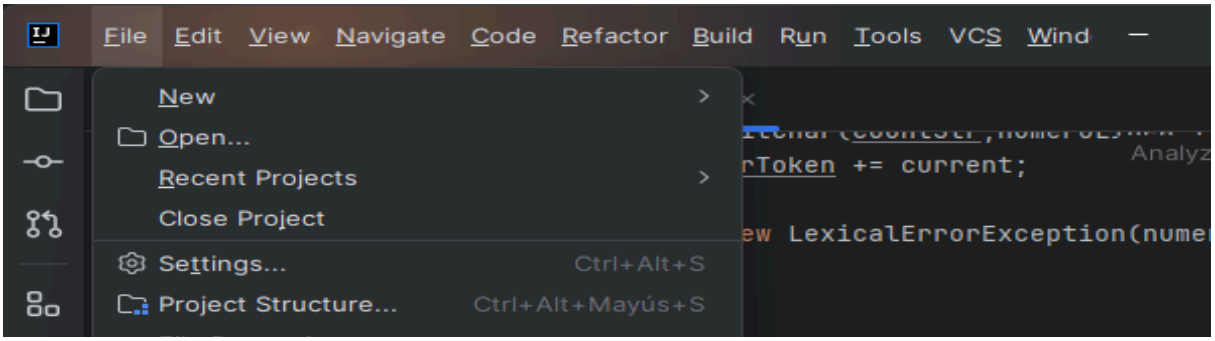

Figura 6: captura de instrucción 2

3. Agregar artefacto: En la ventana de "Project Structure", click en "Artifacts" en el panel izquierdo. Luego, hacer clic en el botón "+" y seleccionar la opción JAR y la segunda opcion.

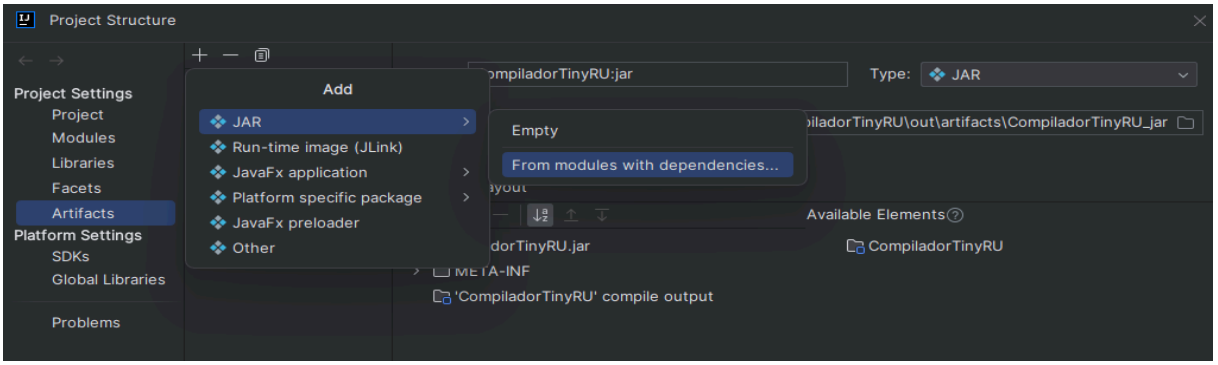

Figura 7: captura de instrucción 3

4. Seleccionar Archivo del Proyecto: Seleccionar el archivo principal del proyecto a incluir en el artefacto (este es, el que contiene el metodo main principal. En nuestro caso la clase Executor). Luego, clic en "OK".

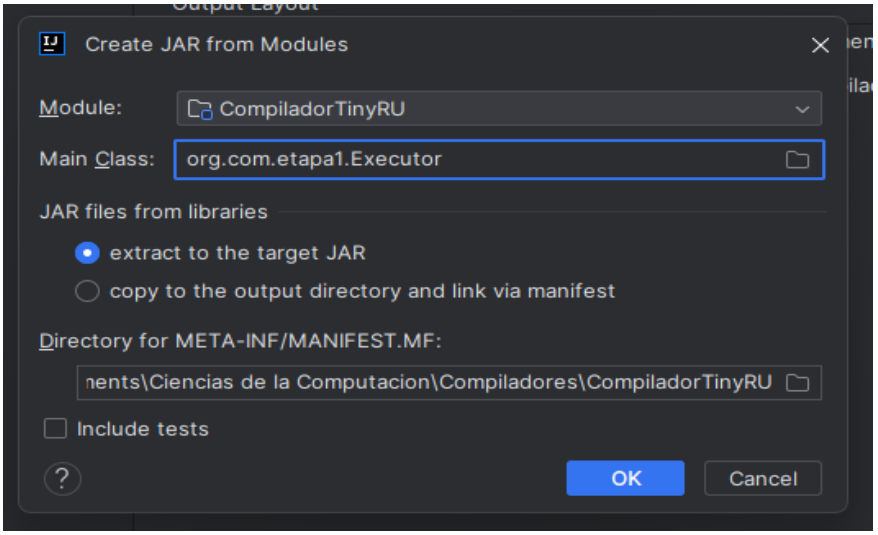

Figura 8: captura de instrucción 4

5. Clic en "Apply" y luego en "OK" para cerrar la ventana de "Project Structure".
6. Compilar el Artefacto: Ir al menú "Build" y seleccionar "Build Artifact".

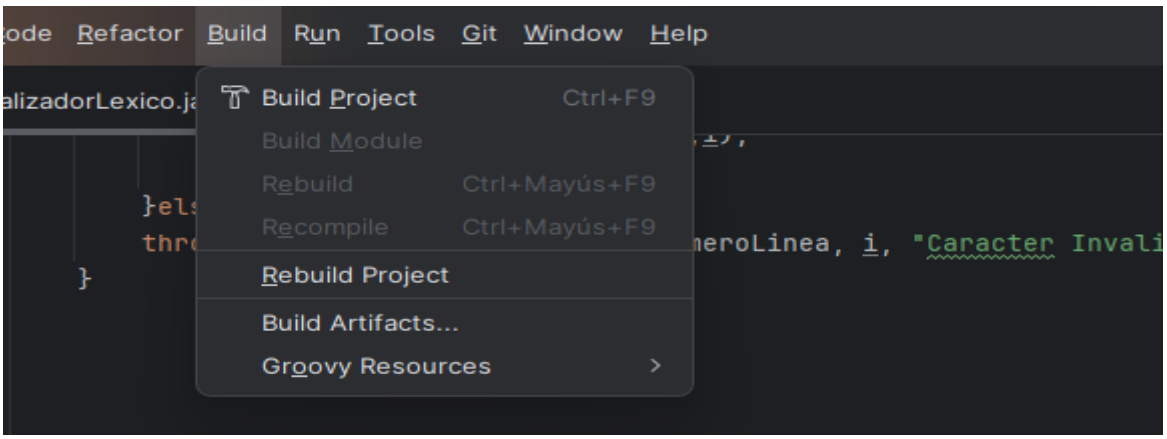

Figura 5:captura de instrucción 5

7. Selecciona la opción Buil para construir el artefacto que acabamos de configurar.

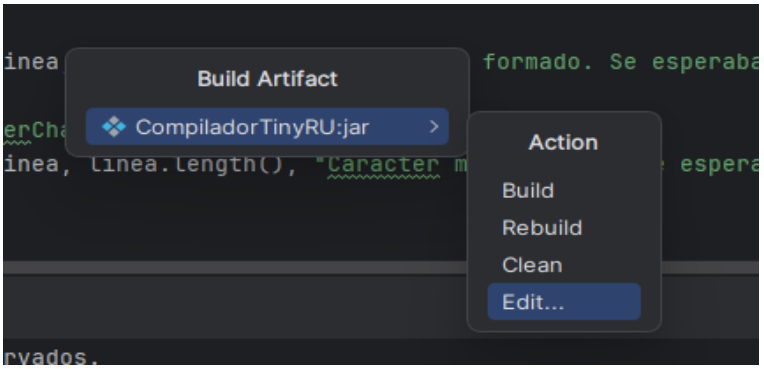

Figura 6: captura de instrucción 7

8. Verificar la Creación del Archivo: Abrir la carpeta donde se encuentra el proyecto y navegar dentro de la carpeta “out”, abrir la carpeta “artifacts” y alli se encuentra la carpeta que contiene el archivo ‘.jar’

5.3 Instrucciones de Ejecución

Ahora detallamos las instrucciones para ejecutar el programa `etapa3.jar`. La sintaxis de invocación respeta la siguiente convención:

java -jar tinyRU.jar <ARCHIVO_FUENTE>

Para ejecutar el programa `tinyRU.jar` junto con su código fuente, siga estos pasos:

1. Descargue el archivo `.jar`. Asegúrese de haber descargado el archivo `tinyRU.jar` en su sistema desde la ubicación proporcionada.
2. Abra una terminal o línea de comandos en su sistema operativo.
3. Navegue al directorio que contiene el archivo `.jar`. Use el comando `cd` para cambiar al directorio que contiene el archivo `tinyRU.jar`.
4. Ejecute el programa: Utilice el siguiente comando para ejecutar el programa y obtener la salida por la terminal:
   > java -jar tinyRU.jar ruta/al/archivo.ru
   Reemplace `ruta/al/archivo.ru` con la ruta al archivo `.ru` que desea analizar.

5.4 Ejemplo de Ejecución

Supongamos que desea compilar el archivo `prueba.ru` que se encuentra en el mismo directorio que el archivo `.jar`:

> java -jar tinyRU.jar prueba.ru

Este comando iniciará el programa y realizará el análisis semántico, sintáctico y léxico del archivo `prueba.ru` y finalmente generara un archivo .asm con el codigo MIPS.

6. CONCLUSIÓN

El desarrollo de este compilador fue un proceso complejo y desafiante que nos permitio explorar en profundidad cada una de las etapas fundamentales de la compilación. Desde la construcción del analizador léxico hasta la implementación de la generación de código, abordamos múltiples aspectos teóricos y prácticos que son cruciales en el diseño de un compilador funcional y eficiente.

El uso de muchas tecnologías nos permitió aprender sobre ellas para luego implementar las etapas del compilador de manera modular y escalable. Cada etapa, desde el análisis léxico hasta la generación de código, fue diseñada y desarrollada personalmente,

En conclusión, el desarrollo de este compilador no solo fueuna tarea técnica significativa, sino también que nos permitio comprender profundamente los principios y prácticas de la compilación y los lenguajes de programación en si, y creemos que este aprendizaje va a ser muy util para nuestra vida profesional como Licenciadas en Ciencias de la Computación.