Compilador TinyRust+

Manual Técnico

Autores:

Fernández, Juan Manuel

Volman, Mariel

Dra. Ana Carolina Olivera

Compiladores

1er semestre del 2023

CONTENIDO

1. INTRODUCCIÓN

En el presente manual se detallan las etapas llevadas a cabo para el desarrollo de un compilador para TinyRust+.

TinyRust+ es un lenguaje de programación reducido basado en Rust, que cuenta con algunas limitaciones para facilitar el desarrollo del compilador.

La primera etapa consiste en el análisis léxico, en ella se detallan el alfabeto de entrada y los tokens reconocidos por el compilador. La segunda etapa consiste en el análisis sintáctico, donde se detalla la gramática que genera el lenguaje. La siguiente etapa consiste en el análisis semántico, la cual se divide en el chequeo de declaraciones con la creación y consolidación de la tabla de símbolos (TDS) y el chequeo de sentencias con el armado del árbol de análisis sintáctico abstracto (AST). Finalmente, la última etapa consiste en la generación de código intermedio.

En las siguientes secciones se describe cada una de las etapas mencionadas anteriormente en mayor profundidad.

2. ANÁLISIS LÉXICO

El análisis léxico es la primera etapa para el desarrollo de un compilador.

El analizador léxico se encarga de leer caracter a caracter el código fuente que se desea compilar, agrupando los caracteres significativos en lexemas y produciendo una secuencia de tokens como salida. Además, elimina aquellos caracteres innecesarios como espacios en blanco, comentarios, tabulaciones o saltos de línea, entre otros. También se ocupa de reportar los errores léxicos.

En las siguientes secciones, se describe el proceso de desarrollo del analizador, la implementación, los errores que es capaz de detectar y los casos de prueba utilizados para testear el funcionamiento del mismo.

a. Desarrollo

En esta sección se detalla el proceso de desarrollo del Analizador Léxico. El mismo consiste en definir un alfabeto de entrada con los caracteres permitidos y luego, elaborar una lista con el nombre de los tokens que se deben reconocer y la expresión regular asociada a cada uno.

Los tokens son extraídos a partir del manual de TinyRust+.

i. Alfabeto de entrada

El alfabeto aceptado por el analizador comprende todos los caracteres ASCII imprimibles.[1]

Σ = { ,!, ", #, $, %, &, ', (,), *, +, ,, -, ., /, [0-9], :, ;, <, =, >, ?, @, [A-Z], [, \,], ^, _, `, [a-z], {, }, |, ~}

ii. Tokens reconocidos

En la siguiente tabla, se detallan los tokens reconocidos por el analizador, junto con la expresión regular que los define.

TOKEN	EXPRESIÓN REGULAR
p_class	class
p_if	if
p_else	else
p_while	while
p_true	true
p_false	false
p_new	new
p_fn	fn
p_main	main
p_create	create
p_pub	pub
p_static	static

[1] https://elcodigoascii.com.ar/

TOKEN	EXPRESIÓN REGULAR
p_return	return
p_self	self
p_void	void
p_Array	Array
p_Int	I32
p_Bool	Bool
p_Char	Char
p_String	Str
p_nil	nil
op_par_abre	(
op_par_cierra	)
op_cor_abre	[
op_cor_cierra	]
op_llave_abre	{
op_llave_cierra	}
op_punto_coma	;
op_coma	,

TOKEN	EXPRESIÓN REGULAR
op_dos_puntos	:
op_punto	.
op_flecha	->
op_asignacion	=
op_menor	<
op_mayor	>
op_menor_o_igual	<=
op_mayor_o_igual	>=
op_igual	==
op_distinto	!=
op_suma	+
op_resta	-
op_mult	*
op_div	/
op_mod	%
op_and	&&
op_or	||

TOKEN	EXPRESIÓN REGULAR
op_not	!
id_clase	[A..Z] [a..z | A..Z | 0..9 | _]*
id_objeto	[a..z] [a..z | A..Z | 0..9 | _]*
lit_ent	0 | [1..9][0..9]*
lit_cad	“ Σ* “
lit_car	‘ (\)? Σ ‘

Tabla 1: Tokens reconocios por el analizador

Además, se reconocen los siguientes espacios en blanco, los cuales son ignorados por el analizador:

- espacio (ascii 32)
- nueva línea (ascii 10)
- retorno de carro (ascii 13)
- tabulación horizontal (ascii 9)
- tabulación vertical (ascii 11)
- comentarios multilínea: Todos los caracteres desde /* hasta */ son ignorados.
- comentarios simples: Todos los caracteres de desde // hasta el final de la línea son ignorados.

b. Implementación

En esta sección se detalla la implementación del Analizador Léxico, indicando a continuación las clases que lo componen y su respectivo diagrama UML en la Figura 1.

i. Token

Esta clase es la encargada de representar los tokens del lenguaje. Sus atributos son:

- *fila*: Indica el número de línea donde se encuentra el token.
- *columna*: Indica el número de columna donde comienza el token.
- *tipoToken*: Corresponde al tipo de token.
- *lexema*: Corresponde al lexema asociado al token.

Cuenta con los métodos "*obtener*" y "*establecer*" por cada atributo para obtener y establecer los respectivos valores de cada Token.

ii. Lexico

Esta clase es la que lleva a cabo todo el trabajo del analizador léxico en sí. Se encarga de leer el archivo de entrada y devolver los tokens reconocidos. Tiene como atributos la "*filaActual*" y la "*columnaActual*", donde almacena la información de cuál fila y columna está leyendo y un método principal llamado "*sigToken*" que a medida que lee el archivo fuente deriva los tokens a sus respectivos autómatas reconocedores y luego devuelve el token reconocido.

iii. Automata

Esta es una superclase de la cual derivan los autómatas encargados de reconocer cada uno de los tokens de TinyRust+ descritos en la Tabla 1. Cuenta con un método principal denominado "*reconocerToken*" que es sobreescrito por cada una de las subclases para devolver el token correspondiente a la clase detectada.

De esta clase derivan las siguientes clases:

- **AutomataIdentificador**: se encarga de reconocer identificadores, tanto de objetos como de clases, y palabras reservadas.
- **AutomataLiteral**: se encarga de reconocer literales enteros, caracteres y cadenas.
- **AutomataOperador**: se encarga de reconocer operadores.
- **AutomataComentario**: se encarga de reconocer comentarios simples y multilínea.

iv. ErrorLéxico

Esta clase es la encargada de manejar los errores léxicos.

Cuando el analizador encuentra un error de este tipo, se crea un error léxico que recibe como parámetros la "*fila*" y la "*columna*" donde se encontró el error y un "*mensaje*" que contiene la descripción del error.

El error es mostrado por pantalla de acuerdo con el siguiente formato:

ERROR: LÉXICO
| NÚMERO DE LÍNEA: | NÚMERO DE COLUMNA: | DESCRIPCIÓN: |

Luego se aborta la ejecución del programa.

v. EjecutadorLexico

Esta clase es la que se encarga de ejecutar el analizador léxico.

Contiene el método "*main*", que recibe como argumento la ruta del archivo de entrada e instancia un objeto de la clase **Lexico**, donde se llama a la función "*sigToken*" iterativamente y se va almacenando los tokens reconocidos en una lista. Finalmente, se imprime dicha lista

por pantalla. Opcionalmente, se puede recibir un segundo argumento indicando la ruta de un archivo de salida, donde se guardará la lista de tokens.

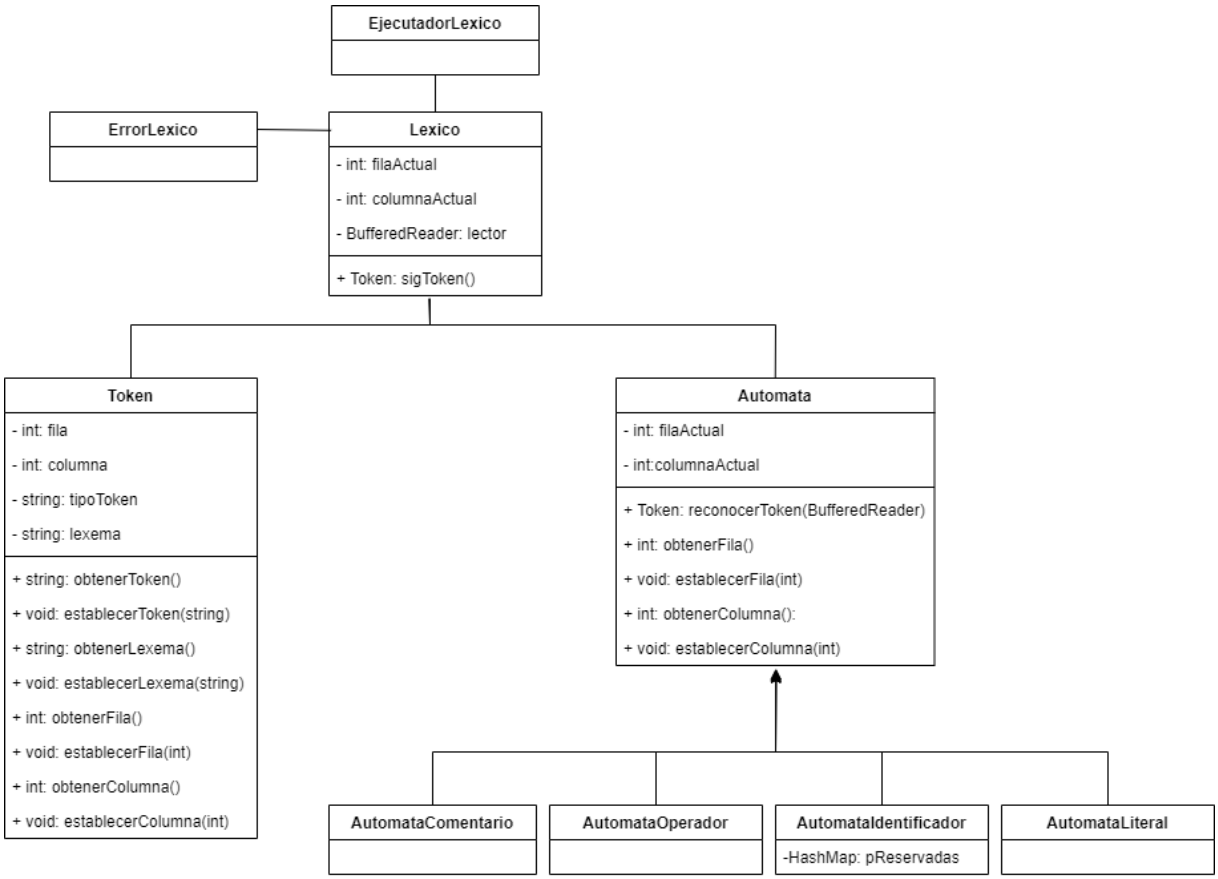

Figura 1: Diagrama de clases Analizador Léxico

c. Errores detectables

A continuación, se detallan los errores reconocidos por el Analizador Léxico:

1. Identificador mal formado (objeto o clase): cuando un identificador contiene caracteres inválidos o no está formado correctamente. Ej: a@x
2. Número inválido: cuando un número no está formado correctamente. Ej: 15a
3. Símbolo inválido: cuando se detecta un símbolo que no corresponde a ningún token. Ej: #
4. Comentario no cerrado: cuando se detecta el inicio de un comentario y no el cierre. Ej: /* esto es un comentario sin cerrar
5. Operador mal formado (and, or): cuando se encuentra un símbolo (& o |) y el siguiente no es el mismo símbolo. Ej: &- o |or
6. Caracter inválido o no cerrado: cuando se comienza a declarar un caracter y no se encuentra el símbolo de cierre. Ej: ‘a

7. String no cerrado: cuando se comienza a declarar un string y no se encuentra el símbolo de cierre. Ej: “hola

d. Casos de prueba

Por cada error explicado en la sección anterior hay un caso de prueba para corroborar el funcionamiento, los mismos se ubican en la carpeta “testLexico”. En cada uno se encuentra un comentario multilínea al comienzo del archivo que indica el tipo y la ubicación del error, sin embargo, el test1 corresponde al error del ítem 1, el test2 al error descripto en el ítem 2, y así sucesivamente. También se cuenta con casos de prueba exitosos que corroboran que los comentarios sean ignorados y los tokens sean reconocidos correctamente, los mismos corresponden a los tests denominados test_cN, donde “N” indica el número de test.

3. ANÁLISIS SINTÁCTICO

La segunda etapa consiste en el análisis sintáctico.

El analizador sintáctico recibe una secuencia de tokens del analizador léxico y verifica que dicha secuencia pueda generarse a partir de la gramática del lenguaje. También se ocupa de reportar los errores sintácticos.

En esta etapa se lleva a cabo la implementación de un analizador sintáctico descendente predictivo recursivo.

En las siguientes secciones, se describe el proceso de desarrollo del analizador, la implementación, los errores que es capaz de detectar y los casos de prueba utilizados para testear el funcionamiento del mismo.

a. Desarrollo

Para crear el analizador sintáctico necesitamos definir una gramática LL(1), sin recursividad, factorizada y no ambigua. Para ello, se utiliza la gramática definida en el manual de TinyRust+ en formato BNFE y se la modifica hasta obtener su gramática equivalente que cumpla con dichas características.

En LL(1), la primera L hace referencia a que los terminales se leen de izquierda a derecha, la segunda L se refiere a que el método de análisis predictivo construye una derivación a izquierda, y el número entre paréntesis indica el número de terminales que se deben consultar para decidir que regla de producción aplicar.

En las secciones siguientes se explican las diferentes conversiones realizadas utilizando la siguiente convención:

Reglas de producción originales

- Reglas de producción reemplazantes

i. Gramática EBNF a BNF

El primer paso es convertir la gramática EBNF en BNF. Para ello se eliminan las extensiones de la gramática EBNF. Los símbolos no terminales donde hay un “?” (0 o 1 ocurrencia) y donde hay un “*” (0 o más ocurrencias) se reemplazan por nuevos no terminales que deriven en dicho no terminal 1 o más veces o en lambda según corresponda.

Start ::= Clase* Método-Main

- Start ::= Clase’ Método-Main
- Clase’ ::= Clase Clase’ | λ

Clase ::= "class" "idClase" Herencia? "{" Miembro* "}"

- Clase ::= "class" "idClase" Herencia’ "{" Miembro’ "}"
- Herencia’ ::= Herencia | λ
- Miembro’ ::= Miembro Miembro’ | λ

Atributo ::= Visibilidad? Tipo ":" Lista-Declaración-Variables ";"

- Atributo ::= Visibilidad’ Tipo ":" Lista-Declaración-Variables ";"
- Visibilidad’ ::= Visibilidad | λ

Método ::= Forma-Método? "fn" "idMétodoVariable" Argumentos-Formales "->" Tipo-Metodo Bloque-Método

- Método ::= Forma-Método’ "fn" "idMétodoVariable" Argumentos-Formales "->" Tipo-Metodo Bloque-Método
- Forma-Método’ ::= Forma-Método | λ

Bloque-Método ::= "{" Decl-Var-Locales* Sentencia* "}"

- Bloque-Método ::= "{" Decl-Var-Locales’ Sentencia’ "}"
- Decl-Var-Locales’ ::= Decl-Var-Locales Decl-Var-Locales’ | λ
- Sentencia’ ::= Sentencia Sentencia’ | λ

Argumentos-Formales ::= "(" Lista-Argumentos-Formales? ")"

- Argumentos-Formales ::= "(" Lista-Argumentos-Formales’ ")"
- Lista-Argumentos-Formales’ ::= Lista-Argumentos-Formales | λ

Sentencia ::= "return" Expresion? ";"

- Sentencia ::= "return" Expresion’ ";"
- Expresion’ ::= Expresion | λ

Bloque ::= "{" Sentencia* "}"

- Bloque ::= "{" Sentencia’ "}"
- Sentencia’ ::= Sentencia Sentencia’ | λ

Asignacion-Variable-Simple ::= "id" Encadenado-Simple* | "id" "[" Expresion "]"

- Asignacion-Variable-Simple ::= "id" Encadenado-Simple' | "id" "[" Expresion "]"
- Encadenado-Simple' ::= Encadenado-Simple Encadenado-Simple' | λ

Asignacion-Self-Simple ::= "self" Encadenado-Simple*

- Asignacion-Self-Simple ::= "self" Encadenado-Simple'

Operando ::= Literal | Primario Encadenado?

- Operando ::= Literal | Primario Encadenado'
- Encadenado' ::= Encadenado | λ

ExpresionParentizada ::= "(" Expresion ")" Encadenado?

- ExpresionParentizada ::= "(" Expresion ")" Encadenado'

AccesoSelf ::= "self" Encadenado?

- AccesoSelf ::= "self" Encadenado'

AccesoVar ::= "id" Encadenado? | "id" "[" Expresion "]"

- AccesoVar ::= "id" Encadenado' | "id" "[" Expresion "]"

Llamada-Método ::= "id" Argumentos-Actuales Encadenado?

- Llamada-Método ::= "id" Argumentos-Actuales Encadenado'

Llamada-Método-Estático ::= "idClase" "." Llamada-Metodo Encadenado?

- Llamada-Método-Estático ::= "idClase" "." Llamada-Metodo Encadenado'

Llamada-Constructor ::= "new" "idClase" Argumentos-Actuales Encadenado? | "new" Tipo-Primitivo "[" Expresion "]"

- Llamada-Constructor ::= "new" "idClase" Argumentos-Actuales Encadenado' | "new" Tipo-Primitivo "[" Expresion "]"

Argumentos-Actuales ::= "(" Lista-Expresiones? ")"

- Argumentos-Actuales ::= "(" Lista-Expresiones' ")"
- Lista-Expresiones' ::= Lista-Expresiones | λ

Llamada-Método-Encadenado ::= "id" Argumentos-Actuales Encadenado?

- Llamada-Método-Encadenado ::= "id" Argumentos-Actuales Encadenado'

Acceso-Variable-Encadenado ::= "id" Encadenado? | id "[" Expresion "]"

- Acceso-Variable-Encadenado ::= "id" Encadenado' | id "[" Expresion "]"

ii. Eliminación de recursividad

El siguiente paso es eliminar la recursividad a izquierda. Para ello, se reemplazan las reglas de producción de la forma **E -> E A | B** por **E -> B E'** y **E' -> A E´ | λ,**

pasando la recursividad hacia la derecha.

ExpOr ::= ExpOr "||" ExpAnd | ExpAnd

- ExpOr ::= ExpAnd ExpOr'
- ExpOr' ::= "||" ExpAnd ExpOr' | λ

ExpAnd ::= ExpAnd "&&" ExpIgual | ExpIgual

- ExpAnd ::= ExpIgual ExpAnd'
- ExpAnd' ::= "&&" ExpIgual ExpAnd' | λ

ExpIgual ::= ExpIgual OpIgual ExpCompuesta | ExpCompuesta

- ExpIgual ::= ExpCompuesta ExpIgual'
- ExpIgual' ::= OpIgual ExpCompuesta ExpIgual' | λ

ExpAdd ::= ExpAdd OpAdd ExpMul | ExpMul

- ExpAdd ::= ExpMul ExAdd'
- ExpAdd' ::= OpAdd ExpMul ExpAdd' | λ

ExpMul ::= ExpMul OpMul ExpUn | ExpUn

- ExpMul ::= ExpUn ExpMul'
- ExpMul' ::= OpMul ExpUn ExpMul' | λ

iii. Factorización

Una vez eliminada la recursividad se procede a factorizar, es decir, reemplazar todas aquellas reglas de producción que derivan en más de una regla con el mismo prefijo, por otras reglas que no cumplan esa condición.

Lista-Declaración-Variables ::= "idMétodoVariable" | "idMétodoVariable" "," Lista-Declaración-Variables

- Lista-Declaración-Variables ::= "idMétodoVariable" Lista-Declaración-Variables'
- Lista-Declaración-Variables' ::= "," Lista-Declaración-Variables | λ

Lista-Argumentos-Formales ::= Argumento-Formal "," Lista-Argumentos-Formales | Argumento-Formal

- Lista-Argumentos-Formales ::= Argumento-Formal Lista-Argumentos-Formales2
- Lista-Argumentos-Formales2 ::= "," Lista-Argumentos-Formales | λ

Sentencia ::= "if" "(" Expresión ")" Sentencia | "if" "(" Expresión ")" Sentencia "else" Sentencia

- Sentencia ::= "if" "(" Expresión ")" Sentencia Sentencia2
- Sentencia2 ::= "else" Sentencia | λ

Asignacion-Variable-Simple ::= "id" Encadenado-Simple' | "id" "[" Expresion "]"

- Asignacion-Variable-Simple ::= "id" Asignacion-Variable-Simple'
- Asignacion-Variable-Simple' ::= Encadenado-Simple' | "[" Expresion "]"

ExpCompuesta ::= ExpAdd OpCompuesto ExpAdd | ExpAdd

- ExpCompuesta ::= ExpAdd ExpCompuesta'
- ExpCompuesta' ::= OpCompuesto ExpAdd | λ

Primario ::= AccesoVar | Llamada-Método
AccesoVar ::= "id" Encadenado' | "id" "[" Expresion "]"
Llamada-Método ::= "id" Argumentos-Actuales Encadenado'

- Primario ::= "id" Primario'
- Primario' ::= AccesoVar' | Llamada-Método'
- AccesoVar' ::= Encadenado' | "[" Expresion "]"
- Llamada-Método' ::= Argumentos-Actuales Encadenado'

Llamada-Constructor ::= "new" "idClase" Argumentos-Actuales Encadenado' | "new" Tipo-Primitivo "[" Expresion "]"

- Llamada-Constructor ::= "new" Llamada-Constructor'
- Llamada-Constructor' ::= "idClase" Argumentos-Actuales Encadenado' | Tipo-Primitivo "[" Expresion "]"

Lista-Expresiones ::= Expresión | Expresión "," Lista-Expresiones

- Lista-Expresiones ::= Expresión Lista-Expresiones2
- Lista-Expresiones2 ::= "," Lista-Expresiones | λ

Encadenado ::= "." Llamada-Metodo-Encadenado | "." Acceso-Variable-Encadenado
Llamada-Método-Encadenado ::= "id" Argumentos-Actuales Encadenado'
Acceso-Variable-Encadenado ::= "id" Encadenado' | "id" "[" Expresion "]"

- Encadenado ::= "." "id" Encadenado2
- Encadenado2 ::= Llamada-Método-Encadenado | Acceso-Variable-Encadenado
- Llamada-Método-Encadenado ::= Argumentos-Actuales Encadenado'
- Acceso-Variable-Encadenado ::= Encadenado' | "[" Expresion "]"

iv. Eliminación de reglas improductivas

Finalmente, se revisa la gramática y se reducen algunas reglas con el objetivo de simplificarla.

Clase ::= "class" "idClase" Herencia' "{" Miembro' "}"
Herencia' ::= Herencia | λ
Herencia ::= ":" "idClase"

- Clase ::= "class" "idClase" RestoClase
- RestoClase ::= ":" "idClase" "{" Miembro' "}" | "{" Miembro' "}"

Atributo ::= Visibilidad' Tipo ":" Lista-Declaración-Variables ";"
Visibilidad' ::= Visibilidad | λ
Visibilidad ::= "pub"

- Atributo ::= pub Tipo ":" Lista-Declaración-Variables ";" | Tipo ":" Lista-Declaración-Variables ";"

Método ::= Forma-Método' "fn" "idMétodoVariable" Argumentos-Formales "->" Tipo-Metodo Bloque-Método
Forma-Método' ::= Forma-Método | λ
Forma-Método ::= "static"

- Método ::= "static" "fn" "idMétodoVariable" Argumentos-Formales "->" Tipo-Metodo Bloque-Método | "fn" "idMétodoVariable" Argumentos-Formales "->" Tipo-Metodo Bloque-Método

Argumentos-Formales ::= "(" Lista-Argumentos-Formales' ")"
Lista-Argumentos-Formales' ::= Lista-Argumentos-Formales | λ

- Argumentos-Formales ::= "(" Lista-Argumentos-Formales'
- Lista-Argumentos-Formales' ::= Lista-Argumentos-Formales ")" | ")"

Sentencia ::= "return" Expresion' ";"
Expresion' ::= Expresion | λ

- Sentencia ::= "return" Expresion'
- Expresion' ::= Expresion ";" | ";"

Encadenado-Simple' ::= Encadenado-Simple Encadenado-Simple' | λ
Encadenado-Simple ::= "." "id"

- Encadenado-Simple' ::= "." "id" Encadenado-Simple' | λ

Argumentos-Actuales ::= "(" Lista-Expresiones' ")"
Lista-Expresiones' ::= Lista-Expresiones | λ

- Argumentos-Actuales ::= "(" Lista-Expresiones'
- Lista-Expresiones' ::= Lista-Expresiones ")" | ")"

Encadenado' ::= Encadenado | λ
Encadenado ::= "." "id" Encadenado2

- Encadenado' ::= "." "id" Encadenado2 | λ

Encadenado2 ::= Llamada-Método-Encadenado | Acceso-Variable-Encadenado
Llamada-Método-Encadenado ::= Argumentos-Actuales Encadenado'
Acceso-Variable-Encadenado ::= Encadenado' | "[" Expresion "]"

- Encadenado2 ::= Argumentos-Actuales Encadenado'| Encadenado' | "[" Expresion "]"

v. Gramática obtenida

A continuación, se describe la gramática final del lenguaje.

G = (Vn, Vt, S, P)

Vn = {Start, Clase', Método-Main, Clase, Resto-Clase, Miembro', Miembro, Atributo, Constructor, Método, Bloque-Método, Decl-Var-Locales', Sentencia', Decl-Var-Locales, Lista-Declaración-Variables, Lista-Declaración-Variables', Argumentos-Formales, Lista-Argumentos-Formales', Lista-Argumentos-Formales, Lista-Argumentos-Formales2, Argumento-Formal, Tipo-Método, Tipo, Tipo-Primitivo, Tipo-Referencia, Tipo-Array, Sentencia, Sentencia2, Expresion', Bloque, Asignación, Asignación-Variable-Simple, Asignación-Variable-Simple', Encadenado-Simple', Asignacion-Self-Simple, Sentencia-Simple, Expresión, ExpOr, ExpOr', ExpAnd, ExpAnd', ExpIgual, ExpIgual', ExpCompuesta, ExpCompuesta', ExpAdd, ExpAdd', ExpMul, ExpMul', ExpUn, OpIgual, OpCompuesto, OpAdd, OpUnario, OpMul, Operando, Encadenado', Literal, Primario, Primario', ExpresionParentizada, AccesoSelf, AccesoVar', Llamada-Método', Llamada-Método-Estático, Llamada-Constructor, Llamada-Constructor', Argumentos-Actuales, Lista-Expresiones', Lista-Expresiones, Lista-Expresiones2, Encadenado2}

Vt = {class, if, else, while, true, false, new, fn, create, pub, static, return, self, void, Array, I32, Bool, Char, Str, nil, (,) , [,] , { , } , ; , , , : , . , -> , = , < , > , <= , >= , == , != , + , - , * , / , % , && , || , ! , id_clase, id_objeto, lit_ent, lit_cad, lit_car}

S = {Start}

P = {Start ::= Clase' Método-Main

Clase' ::= Clase Clase' | λ

Método-Main ::= "fn" "main" "(" ")" Bloque-Método

Clase ::= "class" "idClase" Resto-Clase

Resto-Clase ::= ":" "idClase" "{" Miembro' "}" | "{" Miembro' "}"

Miembro' ::= Miembro Miembro' | λ

Miembro ::= Atributo | Constructor | Método

Atributo ::= "pub" Tipo ":" Lista-Declaración-Variables ";" | Tipo ":" Lista-Declaración-Variables ";"

Constructor ::= "create" Argumentos-Formales Bloque-Método

Método ::= "static" "fn" "idMétodoVariable" Argumentos-Formales "->" Tipo-Metodo Bloque-Método | "fn" "idMétodoVariable" Argumentos-Formales "->" Tipo-Metodo Bloque-Método

Bloque-Método ::= "{" Decl-Var-Locales' Sentencia' "}"

Decl-Var-Locales' ::= Decl-Var-Locales Decl-Var-Locales' | λ

Sentencia' ::= Sentencia Sentencia' | λ

Decl-Var-Locales ::= Tipo ":" Lista-Declaración-Variables ";"

Lista-Declaración-Variables ::= "idMétodoVariable" Lista-Declaración-Variables'

Lista-Declaración-Variables' ::= "," Lista-Declaracion-Variables | λ

Argumentos-Formales ::= "(" Lista-Argumentos-Formales'

Lista-Argumentos-Formales' ::= Lista-Argumentos-Formales ")" | ")"

Lista-Argumentos-Formales ::= Argumento-Formal Lista-Argumentos-Formales2

Lista-Argumentos-Formales2 ::= "," Lista-Argumentos-Formales | λ

Argumento-Formal ::= Tipo ":" "idMétodoVariable"

Tipo-Método ::= Tipo | "void"

Tipo ::= Tipo-Primitivo | Tipo-Referencia | Tipo-Array

Tipo-Primitivo ::= "Bool" | "I32" | "Str" | "Char"

Tipo-Referencia ::= "idClase"

Tipo-Array ::= "Array" Tipo-Primitivo

Sentencia ::= ";"

| Asignacion";"

| Sentencia-Simple ";"

| "if" "(" Expresion ")" Sentencia Sentencia2

| "while" "(" Expresion ")" Sentencia

| Bloque

| "return" Expresion'

Sentencia2 ::= "else" Sentencia | λ

Expresion' ::= Expresion ";" | ";"

Bloque ::= "{" Sentencia' "}"

Asignacion ::= Asignacion-Variable-Simple "=" Expresión | Asignación-Self-Simple "=" Expresion

Asignacion-Variable-Simple ::= "id" Asignacion-Variable-Simple'

Asignacion-Variable-Simple' ::= Encadenado-Simple' | "[" Expresion "]"

Encadenado-Simple' ::= "." "id" Encadenado-Simple' | λ

Asignacion-Self-Simple ::= "self" Encadenado-Simple'

Sentencia-Simple ::= "(" Expresion ")"

Expresion ::= ExpOr

ExpOr ::= ExpAnd ExpOr'

ExpOr' ::= "||" ExpAnd ExpOr' | λ

ExpAnd ::= ExpIgual ExpAnd'

ExpAnd' ::= "&&" ExpIgual ExpAnd' | λ

ExpIgual ::= ExpCompuesta ExpIgual'

ExpIgual' ::= OpIgual ExpCompuesta ExpIgual' | λ

ExpCompuesta ::= ExpAdd ExpCompuesta'

ExpCompuesta' ::= OpCompuesto ExpAdd | λ

ExpAdd ::= ExpMul ExAdd'

ExpAdd' ::= OpAdd ExpMul ExpAdd' | λ

ExpMul ::= ExpUn ExpMul'

ExpMul' ::= OpMul ExpUn ExpMul' | λ

ExpUn ::= OpUnario ExpUn | Operando

OpIgual ::= "==" | "!="

OpCompuesto ::= "<" | ">" | "<=" | ">="

OpAdd ::= "+" | "-"

OpUnario ::= "+" | "-" | "!"

OpMul ::= "*" | "/" | "%"

Operando ::= Literal | Primario Encadenado'

Encadenado' ::= "." "id" Encadenado2 | λ

Literal ::= "nil" | "true" | "false" | "intLiteral" | "StrLiteral" | "charLiteral"

Primario ::= ExpresionParentizada | AccesoSelf | "id" Primario' | Llamada-MetodoEstatico | Llamada-Constructor

Primario' ::= AccesoVar' | Llamada-Método'

ExpresionParentizada ::= "(" Expresion ")" Encadenado'

AccesoSelf ::= "self" Encadenado'

AccesoVar' ::= Encadenado' | "[" Expresion "]"

Llamada-Método' ::= Argumentos-Actuales Encadenado'

Llamada-Método-Estático ::= "idClase" "." Llamada-Metodo Encadenado'

Llamada-Constructor ::= "new" Llamada-Constructor'

Llamada-Constructor' ::= "idClase" Argumentos-Actuales Encadenado' | Tipo-Primitivo "[" Expresion "]"

Argumentos-Actuales ::= "(" Lista-Expresiones'

Lista-Expresiones' ::= Lista-Expresiones ")" | ")"

Lista-Expresiones ::= Expresión Lista-Expresiones2

Lista-Expresiones2 ::= "," Lista-Expresiones | λ

Encadenado2 ::= Argumentos-Actuales Encadenado' | Encadenado' | "[" Expresion "]" }

Cabe destacar que la gramática obtenida no es estrictamente LL(1), ya que posee ambigüedad, es decir, que para la misma cadena existe más de una derivación posible distinta. Este es el caso de la sentencia "if".

A continuación, se muestran dos derivaciones posibles para la siguiente cadena:

"if" "(" Expresion ")" "if" "(" Expresion ")" Sentencia "else" Sentencia

Para ello, se parte desde el No Terminal "Sentencia" y se distingue en color rojo el No Terminal que se está derivando.

- Derivación 1:

Sentencia -> "if" "(" Expresion ")" Sentencia Sentencia2 -> "if" "(" Expresion ")" "if" "(" Expresion ")" Sentencia Sentencia2 Sentencia2 -> "if" "(" Expresion ")" "if" "(" Expresion ")" Sentencia "else" Sentencia Sentencia2 -> **"if" "(" Expresion ")" "if" "(" Expresion ")" Sentencia "else" Sentencia "λ"**

En este caso el primer No Terminal "Sentencia2" deriva en "else Sentencia", mientras que el segundo deriva en λ.

- Derivación 2:

Sentencia -> "if" "(" Expresion ")" Sentencia Sentencia2 -> "if" "(" Expresion ")" "if" "(" Expresion ")" Sentencia Sentencia2 Sentencia2 -> "if" "(" Expresion ")" "if" "(" Expresion ")" Sentencia "λ" Sentencia2 -> **"if" "(" Expresion ")" "if" "(" Expresion ")" Sentencia "λ" "else" Sentencia**

En este caso el primer No Terminal "Sentencia2" deriva en λ, mientras que el segundo deriva en "else Sentencia".

Como se puede observar, existen dos derivaciones distintas para la misma cadena, sin embargo, el analizador sí es LL(1), ya que a la hora de derivar se opta por asociar el "else" al "if" más cercano para no tener este inconveniente.

b. Implementación

Una vez obtenida la gramática, se procede a implementar el analizador sintáctico. Para ello, se crea un método por cada no terminal de la gramática y dentro de cada uno se implementa la regla de producción correspondiente. Si la regla incluye no terminales, se realiza la llamada a dichos métodos, y si la regla incluye terminales, se intenta "machear" el token leído con el terminal que indica la gramática.

A continuación, se detallan las clases que conforman al Analizador Sintáctico y su respectivo diagrama UML en la Figura 2.

i. Sintactico

Esta clase se encarga de la implementación del Analizador Sintáctico Descendente Predictivo Recursivo. Cuenta con un método por cada no terminal de la gramática, incluyendo el no terminal inicial "*Start*" y un método "*éxito*" que corrobora la correcta finalización del analizador.

ii. AuxiliarSintactico

Esta clase es la encargada de interactuar con el Analizador Léxico. Lleva el registro del "*tokenActual*" que se está leyendo, e implementa las funciones "*verifico*", que se encarga de mirar si el token que viene corresponde con uno de los terminales indicados por la gramática (sin consumirlo), y "*matcheo*", que compara el terminal que índica la gramática con el token actual y si coinciden solicita el siguiente token, caso contrario lanza un error.

iii. ErrorSintactico

Esta clase es la encargada de manejar los errores sintácticos.

Cuando el analizador encuentra un error de este tipo, se crea un error sintáctico que recibe como parámetros la "*fila*" y la "*columna*" donde se encontró el error y un "*mensaje*" que contiene la descripción del error.

El error es mostrado por pantalla de acuerdo con el siguiente formato:

ERROR: SINTÁCTICO

| NÚMERO DE LÍNEA: | NÚMERO DE COLUMNA: | DESCRIPCIÓN: |

La descripción contiene un mensaje que dice: "Se esperaba: … , se encontró …" que indica qué token se recibió y cuál debería haber llegado.

Luego se aborta la ejecución del programa.

iv. EjecutadorSintactico

Contiene el método "*main*", que recibe como argumento la ruta del archivo de entrada e instancia un objeto de la clase **Sintactico**, donde se instancia un objeto de la clase **Lexico**, para llevar a cabo la etapa anterior y luego se inicia la ejecución de esta etapa llamando al método "*start*", correspondiente al inicio de la gramática. Una vez finalizada la ejecución, si no hubo ningún error, se imprime por pantalla un mensaje que indica que el análisis ha terminado correctamente.

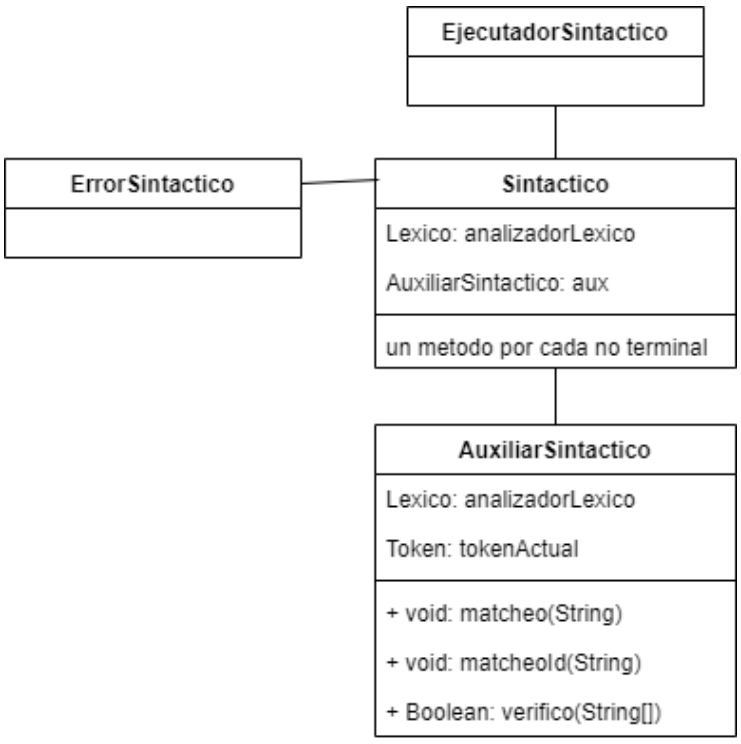

Figura 2: Diagrama de Clases Analizador Sintáctico

c. Errores detectables

Los errores reconocidos por el analizador sintáctico son todos aquellos que no cumplen con la gramática del lenguaje, es decir, cuando se espera que venga determinado token de acuerdo con la regla de producción y viene otro.

Se distinguen por su particularidad los siguientes errores:

1. No puede venir nada después del método *"main"*
2. Falta el método *"main"*
3. No puede haber otro método o variable llamada *"main"*, ya que es una palabra reservada del lenguaje y se usa exclusivamente donde se encuentra definida en la gramática.

Además, se pueden detectar errores léxicos de la etapa anterior.

d. Casos de prueba

Para corroborar el funcionamiento del analizador, se generaron casos de prueba con diferentes errores, los mismos se ubican en la carpeta "testSintactico". En cada archivo se encuentra al comienzo un comentario multilínea que indica cuál es el error que debe aparecer y su ubicación. Además, se cuenta con un caso de prueba exitoso, denominado "test_c0", que corresponde al código de ejemplo que se brinda al final del manual de TinyRust+. (Los tests correctos de la etapa anterior también pueden ser utilizados para probar esta etapa).

4. ANÁLISIS SEMÁNTICO: DECLARACIONES

En esta etapa se lleva a cabo la implementación de una porción del analizador semántico que extiende las funcionalidades del analizador sintáctico desarrollado en la etapa anterior, con el fin de verificar si en el código recibido todas las entidades han sido correctamente declaradas. Para ello, se extiende la gramática, especificando un esquema de traducción que permita crear las estructuras vinculadas a una tabla de símbolos, la cual almacenará la información correspondiente a todas las entidades declaradas.

Luego, se procede a programar el analizador semántico implementando las acciones añadidas al esquema de traducción y las clases necesarias para la construcción de la tabla de símbolos.

Además, se detectan los errores que debe reconocer el Analizador Semántico y se desarrollan diferentes casos de prueba para testear el funcionamiento del mismo.

a. Desarrollo

Para implementar el chequeo de declaraciones del analizador semántico necesitamos definir un esquema de traducción. Para ello, se utiliza la gramática de TinyRust+ obtenida en

la etapa anterior (factorizada y sin recursión a izquierda) y se le agregan las acciones semánticas correspondientes.

i. Esquema de Traducción

A continuación, se describe el esquema de traducción de TinyRust+.

Start ::= Clase' Método-Main
Clase' ::= Clase Clase' | λ
Método-Main ::= {Calse nuevaClase = new Clase("Fantasma"); tds.establecerClaseActual(nuevaClase)} "fn" "main" "(" ")" {Metodo nuevoMetodo = new Metodo("main"); tds.establecerMetodoActual(nuevoMetodo)} Bloque-Método {tds.obtenerClaseActual().insertarMetodo(nuevoMetodo); tds.insertarClase(nuevaClase)}
Clase ::= "class" "idClase" {Calse nuevaClase = new Clase(idClase.lexema); tds.establecerClaseActual(nuevaClase)} Resto-Clase {tds.insertarClase(nuevaClase)}
Resto-Clase ::= ":" "idClase" {tds.obtenerClaseActual().establecerHerencia(idClase.lexema)} "{" Miembro' "}" | {tds.obtenerClaseActual().establecerHerencia("Object")} "{" Miembro' "}"
Miembro' ::= Miembro Miembro' | λ
Miembro ::= Atributo | Constructor | Método
Atributo ::= "pub" {visibilidad=True}Tipo {tipoAtributo=Tipo()} ":" Lista-Declaración-Atributos ";" | {visibilidad=False} Tipo {tipoAtributo=Tipo()} ":" Lista-Declaración-Atributos ";"
Lista-Declaración-Atributos ::= "idAtributo" {Atributo nuevoAtributo = new Atributo(idAtributo.lexema,tipoAtributo,visibilidad); tds.obtenerClaseActual().insertarAtributo(nuevoAtributo)} Lista-Declaración-Atributos'
Lista-Declaración-Atributos ::= "," Lista-Declaración-Atributos | λ
Constructor ::= "create" {Constructor nuevoConstructor = new Constructor(); tds.establecerMetodoActual(nuevoConstructor)} Argumentos-Formales {tds.obtenerClaseActual().establecerConstructor(nuevoConstructor)} Bloque-Método
Método ::= "static" {formaMétodo=True} "fn" "idMétodo" {Metodo nuevoMetodo = new Metodo(idMétodo.lexema,formaMetodo); tds.establecerMetodoActual(nuevoMetodo)} Argumentos-Formales "->" Tipo-Método {tipoMétodo=Tipo-Método(); nuevoMetodo.establecerTipoRetorno(tipoMétodo); tds.obtenerClaseActual().insertarMetodo(nuevoMetodo)} Bloque-Método | {formaMétodo=False} "fn" "idMétodo" {Metodo nuevoMetodo = new Metodo(idMétodo.lexema,formaMetodo); tds.establecerMetodoActual(nuevoMetodo)} Argumentos-Formales "->" Tipo-Método {tipoMétodo=Tipo-Método(); nuevoMetodo.establecerTipoRetorno(tipoMétodo); tds.obtenerClaseActual().insertarMetodo(nuevoMetodo)} Bloque-Método
Bloque-Método ::= "{" Decl-Var-Locales' Sentencia' "}"
Decl-Var-Locales' ::= Decl-Var-Locales Decl-Var-Locales' | λ
Sentencia' ::= Sentencia Sentencia' | λ
Decl-Var-Locales ::= Tipo {tipoVariable=Tipo()} ":" Lista-Declaración-Variables ";"
Lista-Declaración-Variables ::= "idVariable" Variable nuevaVariable = new Variable(idVariable.lexema), tipoVariable); tds.obtenerMetodoActual().insertarVariable(nuevaVariable); Lista-Declaración-Variables'
Lista-Declaración-Variables' ::= "," Lista-Declaracion-Variables | λ
Argumentos-Formales ::= "(" Lista-Argumentos-Formales'
Lista-Argumentos-Formales' ::= Lista-Argumentos-Formales ")" | ")"
Lista-Argumentos-Formales ::= Argumento-Formal {Parametro nuevoParametro = Argumento-Formal(); tds.obtenerMetodoActual().insertarParametro(nuevoParametro)} Lista-Argumentos-Formales2
Lista-Argumentos-Formales2 ::= "," Lista-Argumentos-Formales | λ

Argumento-Formal ::= Tipo {tipoParámetro=Tipo()} ":" "idParámetro" {Parametro p = new Parametro(idParámetro.lexema,tipoParámetro)}
Tipo-Método ::= Tipo {return Tipo()} | "void" {return new TipoVoid("void")}
Tipo ::= Tipo-Primitivo {return TipoPrimitivo()} | Tipo-Referencia {return TipoReferencia()} | Tipo-Array {return TipoArray()}
Tipo-Primitivo ::= "Bool" {return new TipoPrimitivo("Bool")} | "I32" {return new TipoPrimitivo("I32")} | "Str" {return new TipoPrimitivo("Str")} | "Char" {return new TipoPrimitivo("Char")}
Tipo-Referencia ::= "idClase" {return new TipoReferencia(idClase.lexema)}
Tipo-Array ::= "Array" Tipo-Primitivo {return new TipoArray(tipoPrimitivo.tipo)}
Sentencia ::= ";"
| Asignacion";"
| Sentencia-Simple ";"
| "if" "(" Expresion ")" Sentencia Sentencia2
| "while" "(" Expresion ")" Sentencia
| Bloque
| "return" Expresion'
Sentencia2 ::= "else" Sentencia | λ
Expresion' ::= Expresion ";" | ";"
Bloque ::= "{" Sentencia' "}"
Asignacion ::= Asignacion-Variable-Simple "=" Expresión | Asignación-Self-Simple "=" Expresion
Asignacion-Variable-Simple ::= "id" Asignacion-Variable-Simple'
Asignacion-Variable-Simple' ::= Encadenado-Simple' | "[" Expresion "]"
Encadenado-Simple' ::= "." "id" Encadenado-Simple' | λ
Asignacion-Self-Simple ::= "self" Encadenado-Simple'
Sentencia-Simple ::= "(" Expresion ")"
Expresion ::= ExpOr
ExpOr ::= ExpAnd ExpOr'
ExpOr' ::= "||" ExpAnd ExpOr' | λ
ExpAnd ::= ExpIgual ExpAnd'
ExpAnd' ::= "&&" ExpIgual ExpAnd' | λ
ExpIgual ::= ExpCompuesta ExpIgual'
ExpIgual' ::= OpIgual ExpCompuesta ExpIgual' | λ
ExpCompuesta ::= ExpAdd ExpCompuesta'
ExpCompuesta' ::= OpCompuesto ExpAdd | λ
ExpAdd ::= ExpMul ExAdd'
ExpAdd' ::= OpAdd ExpMul ExpAdd' | λ
ExpMul ::= ExpUn ExpMul'
ExpMul' ::= OpMul ExpUn ExpMul' | λ
ExpUn ::= OpUnario ExpUn | Operando
OpIgual ::= "==" | "!="
OpCompuesto ::= "<" | ">" | "<=" | ">="
OpAdd ::= "+" | "-"
OpUnario ::= "+" | "-" | "!"
OpMul ::= "*" | "/" | "%"
Operando ::= Literal | Primario Encadenado'
Encadenado' ::= "." "id" Encadenado2 | λ
Literal ::= "nil" | "true" | "false" | "intLiteral" | "StrLiteral" | "charLiteral"
Primario ::= ExpresionParentizada | AccesoSelf | "id" Primario' | Llamada-MetodoEstatico | Llamada-Constructor
Primario' ::= AccesoVar' | Llamada-Método'
ExpresionParentizada ::= "(" Expresion ")" Encadenado'

AccesoSelf ::= "self" Encadenado'
AccesoVar' ::= Encadenado' | "[" Expresion "]"
Llamada-Método' ::= Argumentos-Actuales Encadenado'
Llamada-Metodo ::= "id" Argumentos-Actuales Encadenado'
Llamada-Método-Estático ::= "idClase" "." Llamada-Metodo Encadenado'
Llamada-Constructor ::= "new" Llamada-Constructor'
Llamada-Constructor' ::= "idClase" Argumentos-Actuales Encadenado' | Tipo-Primitivo "[" Expresion "]"
Argumentos-Actuales ::= "(" Lista-Expresiones'
Lista-Expresiones' ::= Lista-Expresiones ")" | ")"
Lista-Expresiones ::= Expresión Lista-Expresiones2
Lista-Expresiones2 ::= "," Lista-Expresiones | λ
Encadenado2 ::= Argumentos-Actuales Encadenado' | Encadenado' | "[" Expresion "]"

b. Implementación

Una vez obtenido el esquema de traducción, se procede a implementar el chequeo de declaraciones del analizador semántico. A continuación, se detallan las principales clases que lo conforman y su respectivo diagrama UML en la Figura 3.

i. Semantico

Esta clase se encarga de la implementación general del Analizador Semántico, en ella se crea y se consolida la tabla de símbolos.

ii. TablaDeSimbolos

Esta clase es la encargada de la implementación de la tabla de símbolos, donde se almacena toda la información de las entidades declaradas en el código fuente. Por cada entidad existe una clase que modela la información que se guarda en cada una de ellas.

La tabla de símbolos lleva el registro de la *"claseActual"* y el *"métodoActual"* que se está leyendo, para guardar la información en el lugar adecuado y contiene un HashMap *"clases"* con todas las clases declaradas.

iii. Clase

Esta clase almacena la información de las clases.

Sus atributos principales son:

- *nombre*: Corresponde al nombre de la clase.
- *heredaDe*: Corresponde al nombre de la clase padre.
- *atributos*: Es un HashMap que contiene los atributos de la clase.
- *métodos*: Es un HashMap que contiene los métodos de la clase.
- *constructor*: Corresponde al constructor de la clase.

iv. Función

Esta clase modela las funciones del lenguaje, que pueden ser constructores o métodos. Contiene un HashMap *"parámetros"* donde se insertan los parámetros de la función

y otro *"variables"* donde se insertan las variables locales a dicha función y un *"nombre"* que distingue a cada función indicando el nombre del método o constructor.

v. Constructor

Esta clase hereda de Función y se usa para representar a los constructores.

vi. Método

Esta clase hereda de Función y se usa para representar a los métodos.

Además de los atributos de una función, contiene un atributo *"tipoRetorno"*, que indica el tipo de retorno del método, otro atributo *"esEstatico"*, que indica la forma del método y un atributo *"posición"*, que almacena la información de la posición del método en la clase.

vii. Variable

Esta clase modela las variables del lenguaje, que pueden ser variables locales a una función, atributos de una clase o parámetros de una función. Contiene como atributos un *"nombre",* que distingue a cada variable, un *"tipo"*, que indica el tipo de la variable, y la *"posición"*, que indica la posición de dicha variable con respecto a las otras.

viii. Parámetro

Esta clase hereda de Variable y se usa para representar a los parámetros de una función.

ix. Atributo

Esta clase hereda de Variable y se usa para representar a los atributos de una clase. Contiene un atributo denominado *"esPublico"*, que indica la visibilidad del atributo declarado.

x. Tipo

Esta clase representa los tipos de datos aceptados en TinyRust+. Contiene un atributo *"tipo"* que indica el nombre de cada uno. De ella derivan las siguientes clases de tipos:

- **TipoPrimitivo**: su atributo tipo puede ser "Str", "Char", "I32" o "Bool".
- **TipoReferencia**: su atributo tipo es un "idClase".
- **TipoArreglo**: su atributo tipo es un tipoPrimitivo.
- **TipoVoid**: su atributo tipo es "void".

xi. ErrorSemantico

Esta clase es la encargada de manejar los errores semánticos.

Cuando el analizador encuentra un error de este tipo, se crea un error semántico que recibe como parámetros la "*fila*" y la "*columna*" donde se encontró el error y un "*mensaje*" que contiene la descripción del error.

El error es mostrado por pantalla de acuerdo con el siguiente formato:

ERROR: SEMÁNTICO - DECLARACIONES
| NÚMERO DE LÍNEA: | NÚMERO DE COLUMNA: | DESCRIPCIÓN: |

Luego se aborta la ejecución del programa.

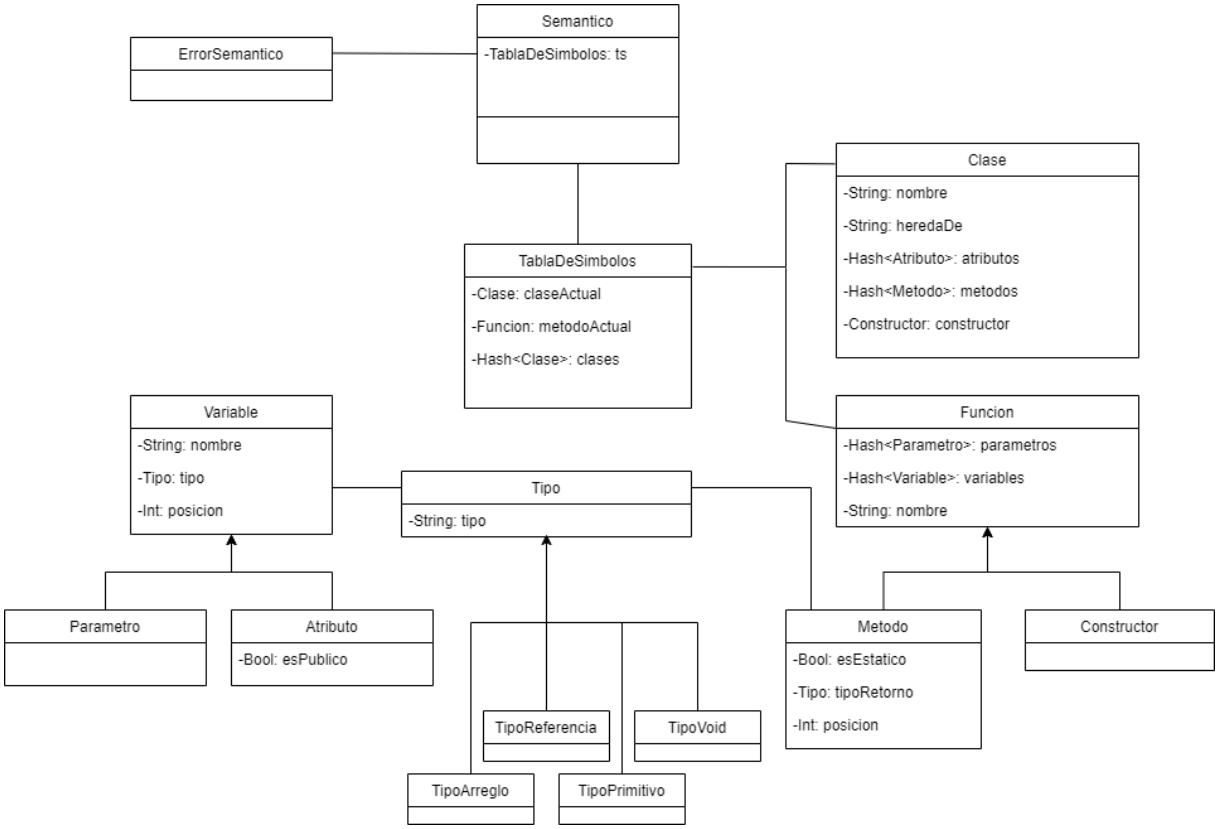

Figura 3: Diagrama de Clases Analizador Semántico - Declaraciones

c. Errores detectables

A continuación, se detallan los errores reconocidos por el Analizador Semántico en el chequeo de declaraciones:

1. No puede haber dos clases con el mismo nombre
2. No puede haber dos métodos con el mismo nombre dentro de una misma clase
3. No puede haber dos atributos con el mismo nombre dentro de una misma clase
4. No puede haber herencia circular
5. No puede haber dos variables con el mismo nombre dentro de una función
6. No puede haber dos parámetros con el mismo nombre dentro de una función
7. No se puede heredar de una clase sin declarar ni se puede utilizar una clase como tipo referencia sin declarar
8. No se puede redefinir un atributo
9. No se puede cambiar la firma de un método heredado (cantidad de parámetros y su tipo, tipo de retorno y forma) y solo se pueden redefinir métodos no estáticos

10. No puede haber más de un constructor por clase

Los errores número 4, 7, 8 y 9 son chequeados durante la consolidación de la tabla de símbolos, mientras que los demás se detectan al momento de construir la tabla.

Además, se pueden reportar errores tanto léxicos como sintácticos de las etapas anteriores.

d. Casos de prueba

Para corroborar el funcionamiento del analizador, se generaron casos de prueba con diferentes errores, los mismos se ubican en la carpeta “testSemantico”. En cada archivo se encuentra al comienzo un comentario multilínea que indica cuál es el error que debe aparecer y su ubicación.

5. ANÁLISIS SEMÁNTICO: SENTENCIAS

En esta etapa se lleva a cabo la implementación del analizador semántico completo, extendiendo las funcionalidades desarrolladas en la etapa anterior, con el fin de realizar el chequeo de sentencias, durante el cual se realiza el chequeo de tipos y la resolución de nombres de las entidades del código fuente. Para ello, se extiende la gramática, especificando un esquema de traducción que permita crear las estructuras de los árboles sintácticos abstractos, correspondientes a los bloques de código del programa fuente.

Luego, se procede a programar el analizador semántico implementando las acciones añadidas y las clases necesarias para la construcción del árbol sintáctico abstracto.

Además, se detectan los errores que debe reconocer el Analizador Semántico y se desarrollan diferentes casos de prueba para testear el funcionamiento del mismo.

a. Desarrollo

Para implementar el chequeo de sentencias del analizador semántico necesitamos definir un esquema de traducción. Para ello, se utiliza la gramática de TinyRust+ obtenida en la etapa 2 (factorizada y sin recursión a izquierda), a la cual se le agregaron las acciones semánticas correspondientes al chequeo de declaraciones en la etapa 3, y se le agregan nuevas acciones semánticas correspondientes a esta etapa.

i. Esquema de Traducción

A continuación, se describe el esquema de traducción de TinyRust+. Las acciones en azul corresponden al chequeo de declaraciones y las acciones en verde al chequeo de sentencias. Por simplicidad, se omite el traspaso de los nodos en el esquema de traducción, entendiendo que son pasados por párametro o retornados a las funciones que los necesitan.

Start ::= Clase’ Método-Main
Clase’ ::= {NodoClase ASTClase = AST.agregarHijo()} Clase Clase’ | λ

Método-Main ::= {Calse nuevaClase = new Clase(“Fantasma”); tds.establecerClaseActual(nuevaClase)} "fn" "main" "(" ")" {Metodo nuevoMetodo = new Metodo(“main”); tds.establecerMetodoActual(nuevoMetodo)} {NodoClase ASTClaseM = AST.agregarHijo(), NodoMetodo ASTMetodoM = ASTClaseM.agregarMetodo(), NodoBloque ASTBloqueM = ASTMetodoM.agregarBloque()} Bloque-Método
{tds.obtenerClaseActual().insertarMetodo(nuevoMetodo); tds.insertarClase(nuevaClase)}
Clase ::= "class" "idClase" {Calse nuevaClase = new Clase(idClase.lexema); tds.establecerClaseActual(nuevaClase)} Resto-Clase {tds.insertarClase(nuevaClase)}
Resto-Clase ::= ":" "idClase" {tds.obtenerClaseActual().establecerHerencia(idClase.lexema)} "{" Miembro’ "}" | {tds.obtenerClaseActual().establecerHerencia(“Object”)} "{" Miembro’ "}"
Miembro’ ::= Miembro Miembro’ | λ
Miembro ::= Atributo | {NodoMetodo ASTMetodo = ASTClase.agregarMetodo()} Constructor | {NodoMetodo ASTMetodo = ASTClase.agregarMetodo()} Método
Atributo ::= "pub" {visibilidad=True}Tipo {tipoAtributo=Tipo()} ":" Lista-Declaración-Atributos ";" | {visibilidad=False} Tipo {tipoAtributo=Tipo()} ":" Lista-Declaración-Atributos ";"
Lista-Declaración-Atributos ::= "idAtributo" {Atributo nuevoAtributo = new Atributo(idAtributo.lexema,tipoAtributo,visibilidad); tds.obtenerClaseActual().insertarAtributo(nuevoAtributo)} Lista-Declaración-Atributos’
Lista-Declaración-Atributos ::= "," Lista-Declaración-Atributos | λ
Constructor ::= "create" {Constructor nuevoConstructor = new Constructor(); tds.establecerMetodoActual(nuevoConstructor)} Argumentos-Formales {tds.obtenerClaseActual().establecerConstructor(nuevoConstructor)} {NodoBloque ASTBloque = ASTMetodo.agregarBloque()} Bloque-Método
Método ::= "static" {formaMétodo=True} "fn" "idMétodo" {Metodo nuevoMetodo = new Metodo(idMétodo.lexema,formaMetodo); tds.establecerMetodoActual(nuevoMetodo)} Argumentos-Formales "->" Tipo-Método {tipoMétodo=Tipo-Método(); nuevoMetodo.establecerTipoRetorno(tipoMétodo); tds.obtenerClaseActual().insertarMetodo(nuevoMetodo)} {NodoBloque ASTBloque = ASTMetodo.agregarBloque()} Bloque-Método | {formaMétodo=False} "fn" "idMétodo" {Metodo nuevoMetodo = new Metodo(idMétodo.lexema,formaMetodo); tds.establecerMetodoActual(nuevoMetodo)} Argumentos-Formales "->" Tipo-Método {tipoMétodo=Tipo-Método(); nuevoMetodo.establecerTipoRetorno(tipoMétodo); tds.obtenerClaseActual().insertarMetodo(nuevoMetodo)} {NodoBloque ASTBloque = ASTMetodo.agregarBloque()} Bloque-Método
Bloque-Método ::= "{" Decl-Var-Locales’ Sentencia’ "}"
Decl-Var-Locales’ ::= Decl-Var-Locales Decl-Var-Locales’ | λ
Sentencia’ ::= Sentencia Sentencia’ | λ
Decl-Var-Locales ::= Tipo {tipoVariable=Tipo()} ":" Lista-Declaración-Variables ";"
Lista-Declaración-Variables ::= "idVariable" Variable nuevaVariable = new Variable(idVariable.lexema), tipoVariable); tds.obtenerMetodoActual().insertarVariable(nuevaVariable); Lista-Declaración-Variables’
Lista-Declaración-Variables’ ::= "," Lista-Declaracion-Variables | λ
Argumentos-Formales ::= "(" Lista-Argumentos-Formales’
Lista-Argumentos-Formales’ ::= Lista-Argumentos-Formales ")" | ")"
Lista-Argumentos-Formales ::= Argumento-Formal {Parametro nuevoParametro = Argumento-Formal(); tds.obtenerMetodoActual().insertarParametro(nuevoParametro)} Lista-Argumentos-Formales2
Lista-Argumentos-Formales2 ::= "," Lista-Argumentos-Formales | λ
Argumento-Formal ::= Tipo {tipoParámetro=Tipo()} ":" "idParámetro" {Parametro p = new Parametro(idParámetro.lexema,tipoParámetro)}
Tipo-Método ::= Tipo {return Tipo()} | "void" {return new TipoVoid(“void”)}

Tipo ::= Tipo-Primitivo {return TipoPrimitivo()} | Tipo-Referencia {return TipoReferencia()} | Tipo-Array {return TipoArray()}
Tipo-Primitivo ::= "Bool" {return new TipoPrimitivo(“Bool”)} | "I32" {return new TipoPrimitivo(“I32”)} | "Str" {return new TipoPrimitivo(“Str”)} | "Char" {return new TipoPrimitivo(“Char”)}
Tipo-Referencia ::= "idClase" {return new TipoReferencia(idClase.lexema)}
Tipo-Array ::= "Array" Tipo-Primitivo {return new TipoArray(tipoPrimitivo.tipo)}
Sentencia ::= ";"
| {NodoAsignacion ASTAsignacion = ASTBloque.agregarAsignacion()} Asignacion";"
| {NodoExpresion ASTExpresion = ASTBloque.agregarExpresion()} Sentencia-Simple ";"
| {NodoIf ASTIf = ASTBloque.agregarIf()} "if" "(" Expresion ")" {NodoBloque ASTSentenciaThen = ASTIf.agregarSentenciaThen()} Sentencia {NodoBloque ASTSentenciaElse = ASTIf.agregarSentenciaElse()} Sentencia2
| {NodoWhile ASTWhile = ASTBloque.agregarWhile()} "while" "(" Expresion ")" {NodoExpresion condicion = ASTWhile.agregarCondicion(); NodoBloque ASTSentenciaW = ASTWhile.agregarBloqueW()} Sentencia
| Bloque
| {NodoReturn ASTReturn = ASTBloque.agregarReturn()} "return" Expresion’ {NodoExpresion expresionRetorno = ASTReturn.agregarExpresion()}
Sentencia2 ::= "else" Sentencia | λ
Expresion’ ::= Expresion ";" | ";" {return NodoExpresion}
Bloque ::= "{" Sentencia’ "}"
Asignacion ::= {NodoVariable ladoIzq = new NodoVariable(), ASTAsignacion.establecerLadoIzq(ladoIzq)} Asignacion-Variable-Simple "=" Expresión {NodoExpresion ladoDer = Expresion(),ASTAsignacion.establecerLadoDer(ladoDer)} | {NodoVariable ladoIzq = new NodoVariable(), ASTAsignacion.establecerLadoIzq(ladoIzq)} Asignación-Self-Simple "=" Expresion {NodoExpresion ladoDer = Expresion(), ASTAsignacion.establecerLadoDer(ladoDer)}
Asignacion-Variable-Simple ::= "id" Asignacion-Variable-Simple’
Asignacion-Variable-Simple’ ::= Encadenado-Simple’ | "[" Expresion "]" {NodoArreglo arreglo = new NodoArreglo(), arreglo.establecerEnc(Expresion), var.establecerEnc(arreglo)}
Encadenado-Simple’ ::= "." "id" {NodoVariable varEnc = new NodoVariable(“id.lexema”), var.establecerEnc(varEnc)} Encadenado-Simple’ | λ
Asignacion-Self-Simple ::= "self" Encadenado-Simple’
Sentencia-Simple ::= "(" Expresion ")" {return NodoExpresion}
Expresion ::= ExpOr {return ExpOr}
ExpOr ::= ExpAnd ExpOr’ {if (ExpOr’ == null) : return ExpAnd, else : return new NodoExpBinaria()}
ExpOr’ ::= "||" ExpAnd ExpOr’ {if (ExpOr’ == null) : return ExpAnd, else : return new NodoExpBinaria()} | λ {return null}
ExpAnd ::= ExpIgual ExpAnd’ {if (ExpAnd’ == null) : return ExpIgual, else : return new NodoExpBinaria()}
ExpAnd’ ::= "&&" ExpIgual ExpAnd’ {if (ExpAnd’ == null) : return ExpIgual, else : return new NodoExpBinaria()} | λ {return null}
ExpIgual ::= ExpCompuesta ExpIgual’ {if (ExpIgual’ == null) : return ExpCompuesta, else : return new NodoExpBinaria()}
ExpIgual’ ::= OpIgual ExpCompuesta ExpIgual’ {if (ExpIgual’ == null) : return ExpCompuesta, else : return new NodoExpBinaria()} | λ {return null}
ExpCompuesta ::= ExpAdd ExpCompuesta’ {if (ExpCompuesta’ == null) : return ExpAdd, else : return new NodoExpBinaria()}
ExpCompuesta’ ::= OpCompuesto ExpAdd {return ExpAdd} | λ {return null}
ExpAdd ::= ExpMul ExAdd’ {if (ExpAdd’ == null) : return ExpMul, else : return new NodoExpBinaria()}
ExpAdd’ ::= OpAdd ExpMul ExpAdd’ {if (ExpAdd’ == null) : return ExpMul, else : return new NodoExpBinaria()} | λ {return null}

ExpMul ::= ExpUn ExpMul' {if (ExpMul' == null) : return ExpUn, else : return new NodoExpBinaria()}
ExpMul' ::= OpMul ExpUn ExpMul' {if (ExpMul' == null) : return ExpUn, else : return new NodoExpBinaria()} | λ {return null}
ExpUn ::= OpUnario ExpUn {return new NodoExpUnaria} | Operando {return Operando}
OpIgual ::= "==" | "!="
OpCompuesto ::= "<" | ">" | "<=" | ">="
OpAdd ::= "+" | "-"
OpUnario ::= "+" | "-" | "!"
OpMul ::= "*" | "/" | "%"
Operando ::= Literal {return Literal} | Primario Encadenado' {return Primario}
Encadenado' ::= "." "id" Encadenado2 | λ
Literal ::= "nil" | "true" | "false" | "intLiteral" | "StrLiteral" | "charLiteral" {return new NodoLiteral(tokenActual.lexema)}
Primario ::= ExpresionParentizada | AccesoSelf | "id" Primario' | Llamada-MetodoEstatico | Llamada-Constructor {return NodoExpresion (en todos los casos)}
Primario' ::= {NodoVariable var = new NodoVariable()} AccesoVar' | {NodoLlamadaMetodo llamadaM = new NodoLlamadaMetodo()} Llamada-Método'
ExpresionParentizada ::= "(" Expresion ")" {NodoExpresion exp = new NodoExpresion()} Encadenado'
AccesoSelf ::= "self" {NodoVariable var = new NodoVariable()} Encadenado'
AccesoVar' ::= Encadenado' | "[" Expresion "]" {NodoArreglo arreglo = new NodoArreglo(), arreglo.establecerEnc(Expresion), var.establecerEnc(arreglo)}
Llamada-Método' ::= Argumentos-Actuales Encadenado'
Llamada-Metodo ::= "id" {NodoLlamadaMetodo llamadaM = new NodoLlamadaMetodo("id.lexema")} Argumentos-Actuales Encadenado'
Llamada-Método-Estático ::= "idClase" "." Llamada-Metodo Encadenado'
Llamada-Constructor ::= "new" Llamada-Constructor'
Llamada-Constructor' ::= "idClase" {NodoLlamadaMetodo llamadaC = new NodoLlamadaMetodo("id.lexema")} Argumentos-Actuales Encadenado' | Tipo-Primitivo "[" Expresion "]"
Argumentos-Actuales ::= "(" Lista-Expresiones'
Lista-Expresiones' ::= Lista-Expresiones ")" | ")"
Lista-Expresiones ::= Expresión Lista-Expresiones2
Lista-Expresiones2 ::= "," Lista-Expresiones | λ
Encadenado2 ::= {NodoLlamadaMetodo llamadaM = new NodoLlamadaMetodo()} Argumentos-Actuales Encadenado' | {NodoVariable var = new NodoVariable()} Encadenado' | {NodoVariable var = new NodoVariable()} "[" Expresion "]" {NodoArreglo arreglo = new NodoArreglo(), arreglo.establecerEnc(Expresion), var.establecerEnc(arreglo)}

b. Implementación

Una vez obtenido el esquema de traducción, se procede a implementar el chequeo de sentencias del analizador semántico. A continuación, se detallan las principales clases que lo conforman y su respectivo diagrama UML en la Figura 4.

i. Semantico

Esta clase, como vimos anteriormente, se encarga de la implementación general del Analizador Semántico. En ella se creó y se consolidó la tabla de símbolos y ahora se crea y se chequea el AST.

ii. Nodo

Esta clase es la encargada de la implementación del AST, representa los nodos del árbol donde se almacena toda la información de las sentencias del código fuente.

Por cada sentencia existe una clase que representa un tipo de nodo distinto y modela la información que se guarda en cada uno de ellos.

iii. NodoAST

El NodoAST es la raíz del árbol, el cual contiene una lista de todas las clases declaradas.

iv. NodoClase

El NodoClase contiene como atributos el *"nombre"* de la clase y un listado con todos sus métodos.

v. NodoMetodo

El NodoMetodo contiene como atributos el *"nombre"* del método y su *"bloque"*.

vi. NodoBloque

El NodoBloque contiene un conjunto de sentencias. Las mismas pueden ser condicionales (if/while), el retorno del método o expresiones.

- **NodoIf**: contiene una *"condición"*, una *"sentenciaThen"* y una *"sentenciaElse"*
- **NodoWhile**: contiene una *"condición"* y un *"bloqueWhile"*
- **NodoReturn**: contiene una expresión de *"retorno"*
- **NodoExpresion**: puede ser una expresión binaria, unaria, una asignación, una llamada a método, un acceso a variable o un literal

vii. ErrorSemantico

Esta clase, como vimos anteriormente, es la encargada de manejar los errores semánticos.

Cuando el analizador encuentra un error de este tipo, se crea un error semántico que recibe como parámetros la "*fila*" y la "*columna*" donde se encontró el error y un "*mensaje*" que contiene la descripción del error.

Si el error semántico es de sentencias se recibe además un parámetro *"sent"* = true, para distinguirlo de los errores semánticos de declaraciones.

El error es mostrado por pantalla de acuerdo con el siguiente formato:

ERROR: SEMÁNTICO - SENTENCIAS
| NÚMERO DE LÍNEA: | NÚMERO DE COLUMNA: | DESCRIPCIÓN: |

Luego se aborta la ejecución del programa.

viii. EjecutadorSemantico

Esta clase es la que se encarga de ejecutar el analizador semántico completo.

Contiene el método "*main*", que recibe como argumento la ruta del archivo de entrada e instancia un objeto de la clase **Semantico**, donde se inicializan la tabla de símbolos y el AST y se instancia un objeto de la clase **Sintactico**, para llevar a cabo las etapas anteriores.

Una vez realizado el análisis sintáctico con la creación de la tabla de símbolos y el armado del AST, se termina el chequeo de declaraciones consolidando la tabla de símbolos y se realiza el chequeo de sentencias sobre el AST. Finalmente, se crean dos archivos de salida en formato json, uno correspondiente a la estructura de la tabla de símbolos y el otro correspondiente a la estructura del AST.

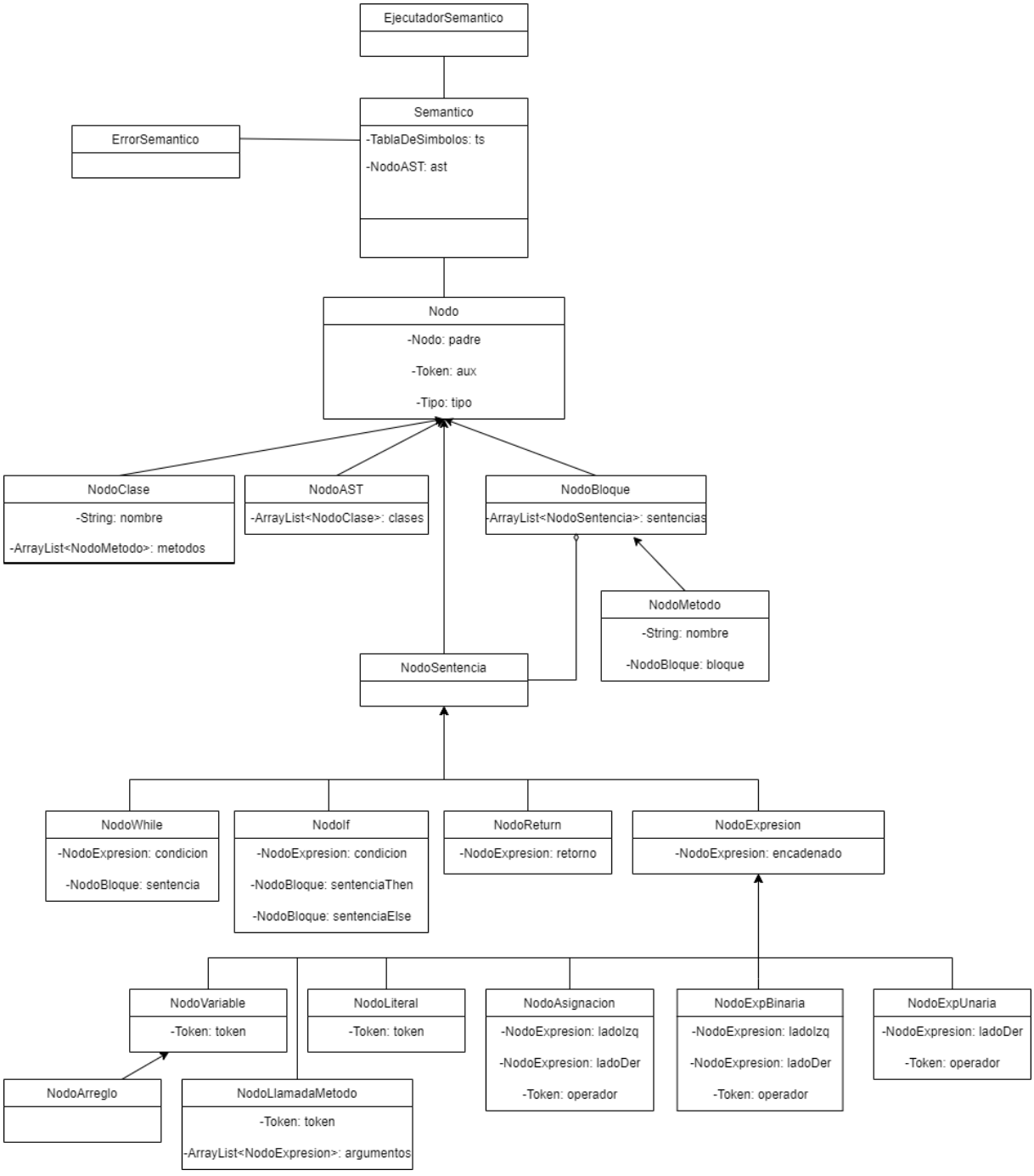

Figura 4: Diagrama de Clases Analizador Semántico - Sentencias

c. Errores detectables

A continuación, se detallan los errores reconocidos por el Analizador Semántico en el chequeo de sentencias:

1. Asignación: el lado derecho debe ser del mismo tipo o un subtipo del lado izquierdo
2. If: La condición debe ser de tipo booleana

3. While: La condición debe ser de tipo booleana
4. Return: el tipo de retorno del método debe coincidir con el declarado, los constructores y el método “main” no poseen sentencia de retorno, los demás métodos deben tener una sentencia de retorno al final del método
5. Expresiones binarias: el lado izquierdo debe ser del mismo tipo que el lado derecho y los tipos de operandos deben ser compatibles con el operador
    a. Los operadores +, -, *, / y % solo admiten operandos de tipo I32 y el resultado es una expresión de tipo I32
    b. Los operadores <, <=, >, >= solo admiten operandos de tipo I32 y el resultado es una expresión de tipo Bool
    c. Los operadores && y || solo admiten operandos de tipo Bool y el resultado es una expresión de tipo Bool
    d. Los operadores == y != admiten cualquier tipo de operando y el resultado es una expresión de tipo Bool
6. Expresiones unarias: el tipo de operando debe ser compatible con el operador
    a. Los operadores + y – solo admiten operandos de tipo I32 y el resultado es una expresión de tipo I32
    b. El operador ! solo admite operandos de tipo Bool y el resultado es una expresión de tipo Bool
7. Llamadas a métodos: los argumentos deben coincidir en cantidad y tipo con los parámetros del método llamado (el cual debe estar declarado en el alcance utilizado). Además, los métodos estáticos no pueden acceder a métodos dinámicos.
8. Llamada a constructor: la clase para la cual se quiere crear un objeto nuevo debe existir
9. Variables: las variables utilizadas deben estar definidas en el alcance que se las quiere acceder (ya sean variables locales, parámetros, atributos o la variable especial “self”). Los atributos no pueden ser accedidos desde métodos estáticos.
10. Arreglos: la expresión de acceso al arreglo debe ser de tipo I32

Todos los errores son chequeados durante una segunda pasada, luego de haber construido el AST y consolidado la tabla de símbolos.

d. Casos de prueba

Para corroborar el funcionamiento del analizador, se generaron casos de prueba con diferentes errores, los mismos se ubican en la carpeta “testSemantico2”. En cada archivo se

encuentra al comienzo un comentario multilínea que indica cuál es el error que debe aparecer y su ubicación. El nombre del archivo indica la sentencia a la cual está relacionada el error.

Además, se cuenta con casos de prueba exitosos que sirven para corroborar el funcionamiento tanto del chequeo de sentencias como el chequeo de declaraciones, los mismos corresponden a los tests denominados test_cN, donde “N” indica el número de test.

6. REPRESENTACIONES INTERMEDIAS

Como se vio anteriormente, la salida de la tercera etapa consiste en generar dos archivos en formato json, uno con la estructura de la tabla de símbolos y otro con la estructura del árbol de análisis sintáctico. Estas son dos representaciones intermedias del código fuente que servirán posteriormente para la etapa de generación de código.

A continuación, se describe la estructura de cada archivo y se da un ejemplo de salida de la TDS y del AST en base al código presentado en la siguiente figura:

```
1  /*CORRECTO*/
2  class Persona {
3      Str : nombre;
4      I32 : edad;
5      create(Str: nombre) {
6          self.nombre = nombre;
7      }
8      fn obtener_edad(Persona: p) -> I32 {
9          return p.edad;
10     }
11
12 }
13 fn main () {
14     Persona : p;
15     p = new Persona("pepe");
16 }
```

Figura 5: "test_ejemplo.rs"

a. Tabla de Símbolos

La primera línea indica el nombre del archivo fuente. Luego, se muestran las clases, indicando sus nombres, de quién heredan, sus métodos, sus atributos y su constructor. Por cada método se muestra su nombre, si es estático o no, su tipo de retorno, su posición y sus parámetros formales. Por cada atributo se muestra su nombre, su tipo, si es público o no y su posición. Y por cada constructor se muestran sus parámetros formales. Dentro de cada parámetro formal se indica su nombre, su tipo y su posición.

La siguiente figura muestra un ejemplo de una parte de la salida en formato json de la TDS para el código fuente “test_ejemplo.rs” de la Figura 5 (el cual se encuentra en la carpeta testSematico2 del proyecto).

```
{
  "nombre": "test_ejemplo.rs",
  "clases": [
    {
      "nombre": "Persona",
      "heredaDe": "Object",
      "metodos": [
        {
          "nombre": "obtener_edad",
          "static": "false",
          "retorno": "I32",
          "posicion": "0",
          "paramF": [
            {
              "nombre": "p",
              "tipo": "Persona",
              "posicion": "0"
            }
          ]
        }
      ],
      "atributos": [
        {
          "nombre": "nombre",
          "tipo": "Str",
          "public": "false",
          "posicion": "0"
        },
        {
          "nombre": "edad",
          "tipo": "I32",
          "public": "false",
          "posicion": "1"
        }
      ],
      "constructor": {
        "paramF": [
          {
            "nombre": "nombre",
            "tipo": "Str",
            "posicion": "0"
          }
        ]
      }
```


Figura 6: "test_ejemplo.ts.json"

b. *Árbol Sintáctico Abstracto*

La primera línea indica el nombre del archivo fuente. Luego, se muestran las clases, indicando sus nombres y los métodos que contienen. Por cada método se muestra su nombre y su bloque, y dentro de cada bloque, se muestra el conjunto de sentencias que contiene. Las sentencias indican su tipo (Nodo al que pertenecen) y sus respectivos hijos en orden. Por ejemplo, para una expresión binaria, primero se muestra el operador, y luego se indican los hijos, mostrando primero el izquierdo y luego el derecho. Cuando se llega a un operando (literal, variable o llamada a método) se muestra su valor y su encadenado, en caso de que contenga.

La siguiente figura muestra un ejemplo del AST en formato json para el mismo código fuente “test_ejemplo.rs” de la Figura 5 utilizado para mostrar la estructura de la TDS.

```
{
  "nombre": "test_ejemplo.rs",
  "clases": [
    {
      "nombre": "Persona",
      "metodos": [
        {
          "nombre": "constructor",
          "Bloque": {
            "Sentencias": [
              {
                "Nodo": "NodoAsignacion",
                "operador": "=",
                "hijos": [
                  {
                    "Nodo": "NodoVariable",
                    "valor": "self",
                    "encadenado": {
                      "Nodo": "NodoVariable",
                      "valor": "nombre"
                    }
                  },
                  {
                    "Nodo": "NodoVariable",
                    "valor": "nombre"
                  }
                ]
              }
            ]
          }
        },
        {
          "nombre": "obtener_edad",
          "Bloque": {
            "Sentencias": [
              {
                "Nodo": "NodoRetorno",
                "hijos": [
                  {
                    "Nodo": "NodoVariable",
                    "valor": "p",
                    "encadenado": {
                      "Nodo": "NodoVariable",
                      "valor": "edad"
                    }
                  }
                ]
              }
            ]
          }
        }
      ]
    },
    {
      "nombre": "Fantasma",
      "metodos": [
        {
          "nombre": "main",
          "Bloque": {
            "Sentencias": [
              {
                "Nodo": "NodoAsignacion",
                "operador": "=",
                "hijos": [
                  {
                    "Nodo": "NodoVariable",
                    "valor": "p"
                  },
                  {
                    "Nodo": "NodoLlamadaMetodo",
                    "valor": "Persona"
                  }
                ]
              }
            ]
          }
        }
      ]
    }
  ]
}
```


Figura 7: "test_ejemplo.ast.json"

7. GENERACIÓN DE CÓDIGO

La última etapa para el desarrollo del compilador es la generación de código intermedio. En esta etapa, se extiende el desarrollo de las etapas anteriores y, a partir del árbol sintáctico abstracto desarrollado en la etapa anterior, se especifica la generación de código intermedio para un programa TinyRust+. El código generado se basa en la arquitectura de MIPS32.

a. Desarrollo

Para poder generar código correctamente el compilador debe decidir que estructura tendrán los datos y generar código que manipule dichos datos de forma correcta. Para ello, se debe establecer algunos criterios, como el diseño del registro de activación de un método, de un CIR (Class Instant Record) y las VT (Virtual Method Table).

Un registro de activación contiene toda la información que se requiere para manejar la ejecución de un método activado. En la siguiente tabla, se muestra el diseño del registro de activación de un método para este compilador:

Enlace dinámico (RA del llamador)
Parámetros actuales
Variables locales
Puntero al objeto (self)
Puntero de retorno (código del llamador)

Tabla 2: RA de un método

Un CIR es una tabla que se crea por cada objeto que contiene todas las variables de instancia más una referencia a la VT de la clase. A continuación, se muestra el diseño de un CIR para este compilador:

Variables de instancia 1
…
Variables de instancia n
VT

Tabla 3: CIR de una clase

Finalmente, una VT es una tabla que contiene todos los métodos de una clase. A continuación, se muestra el diseño de una VT:

Método 1
…
Método n

Tabla 4: VT de una clase

b. Implementación

Para la generación de código intermedio, se implementa una clase que inicia con la generación de código y se añade a cada Nodo del AST una función denominada *"genCodigo"* que va construyendo un archivo .asm con el código intermedio generado.

i. GeneradorCodigo

Esta clase es la que se encarga de iniciar con la generación de código. Para ello, instancia un objeto de la clase **Semantico**, encargado de realizar las etapas anteriores, y luego obtiene el AST y llama a la función *"genCodigo"*, generando un archivo de salida .asm con el código intermedio.

ii. Ejecutador

Esta clase es la que se encarga de ejecutar el compilador completo.

Contiene el método "*main*", que recibe como argumento la ruta del archivo de entrada e instancia un objeto de la clase **GeneradorCodigo**, donde se comienza con la ejecución de las diferentes etapas como se explicó previamente.

c. Errores detectables

A continuación, se detallan los errores reconocidos en tiempo de ejecución, que se incluyen en la generación de código:

1. División por cero (operador "/")
2. División por cero en módulo (operador "%")

d. Casos de prueba

Esta etapa cuenta con diversos casos de prueba exitosos que corroboran que el código generado sea correcto. Los mismos se ubican en la carpeta "test" dentro del directorio "GeneracionCodigo" y se denominan test_cN, donde "N" indica el número de test. En cada uno se encuentra un comentario que describe lo que se está testeando. Además, se cuenta con un test por cada error explicado en la sección anterior que finaliza su ejecución mostrando por pantalla un mensaje con el error detectado.

8. COMPILACIÓN Y REQUERIMIENTOS MÍNIMOS

El proyecto está implementado en el lenguaje de programación "Java". Para compilar el código es necesario contar con la versión del JDK 17 o superior, lo mismo es requerido para ejecutar el código compilado, no se requiere un sistema operativo o arquitectura específica.

El procedimiento para compilar y utilizar el código consta de lo siguiente:

1. Ejecutar "javac NombreEtapa/*.java" con el nombre de la etapa que se desea compilar, desde una terminal en el directorio raíz del repositorio, esto producirá como salida los archivos .class compilados
2. Ejecutar el comando "jar cfvm NombreEtapa.jar MANIFEST.MF *" en el mismo directorio donde nos encontrábamos posicionados, esto introduce todos los archivos, incluyendo el Manifest.txt, en el archivo .jar

La sintaxis de invocación para la correcta ejecución del código es la siguiente:

java -jar NombreEtapa.jar <ARCHIVO_FUENTE>

donde <ARCHIVO_FUENTE> es el archivo a analizar de TinyRust+. Si el archivo no contiene errores la ejecución terminará cuando se llegue al fin del archivo y se mostrará por pantalla un mensaje exitoso. Caso contrario, se abortará la ejecución y se especificará el error detectado.