UNCUYO
UNIVERSIDAD
NACIONAL DE CUYO

FACULTAD DE
INGENIERÍA

Universidad Nacional de Cuyo

Facultad de Ingeniería

tinyRust+

Compilador

Manual del Desarrollador

Autor: Morales, Juan Martín

Docente a cargo: Dra. Ana Carolina Olivera

Contenido

1 Introducción

El presente manual brinda una guía exhaustiva sobre el desarrollo del compilador para el lenguaje `tinyRust+`. Este documento está organizado por secciones, donde cada una representa una etapa del proceso de compilación o características del compilador.

Las secciones principales de este manual incluyen: características del lenguaje, análisis léxico, análisis sintáctico, análisis semántico y generación de código. En cada sección del manual se discutirán los aspectos teóricos y técnicos de la etapa correspondiente. Además, contiene información de cómo construir el proyecto (mediante el código fuente) y como compilarlo y ejecutarlo.

El objetivo principal de este manual es detallar cada decisión de diseño, que herramientas se utilizaron y que conceptos teóricos acompañaron la construcción del compilador. Al finalizar la lectura del documento se espera que el lector tenga un conocimiento mínimo del proceso de desarrollo del compilador para `tinyRust+`, el cual le permita comprender el proyecto completo y realizar las modificaciones pertinentes.

2 Características de tinyRust+ y puesta a prueba

El lenguaje `tinyRust+` es un lenguaje de programación reducido del popular `Rust`, además tiene modificaciones en la estructura para facilitar el desarrollo. El alcance del proyecto es suficiente para desarrollar habilidades académicas, aunque también incluye características propias de los lenguajes modernos.

Es un lenguaje orientado a objetos con un tipado fuerte y estático, en el cual las variables tienen un tipo de dato asociado que es estrictamente aplicado en tiempo de compilación. Este sistema de tipos garantiza que no ocurran errores de tipo en tiempos de ejecución.

Al ser orientado a objetos el código de `tinyRust+`:

- Se organiza en clases que pueden heredar atributos y comportamientos de otras clases.
- Es posible sobrescribir los métodos no estáticos heredados, pero no los atributos.
- También permite polimorfismo en los objetos instanciados de las clases.
- Permite declarar la visibilidad de un atributo para cumplir con el principio de ocultamiento.

Cada código a compilar debe tener un punto de entrada, que en este caso es un método *main*, que deberá estar declarado fuera del *scope* clases.

A medida que se avance en las distintas etapas se detallarán en mayor medida las características de `tinyRust+`.

3 Esquema general del compilador

El compilador de `tinyRust+` se ha diseñado mediante un enfoque orientado a objetos, y para implementar tal diseño se utilizó el lenguaje de programación *Java*. Las clases utilizadas se han encapsulado en varios paquetes, según las funciones específicas que se desempeñen. A continuación, se listan los paquetes principales (y subpaquetes) utilizados en el diseño del compilador, junto con una breve descripción de su funcionamiento:

- *Frontend*: Este paquete, y con ayuda de los paquetes *AST* y *SymbolTable,* encapsula todas de las funcionalidades únicas y exclusivas de las etapas de análisis del código fuente.
    - *Lexer*: Contiene las clases útiles para la etapa del análisis léxico ("*tokenización"*) y el reporte de errores de este.
    - *Parser*: Contiene las clases involucradas en la etapa del análisis sintáctico y semántico. Y reporta errores sintáctico y semánticos en las declaraciones.
- *AST:* Este paquete contiene las clases que ayudarán a construir la representación intermedio del árbol sintáctico abstracto (AST). Contiene funcionalidades para realizar los chequeos de semántica sentencial. Depende del paquete *Backend* para la generación de código final del compilador.
- *SymbolTable:* Encapsula las clases útiles a la hora de crear la tabla de símbolos del compilador. Otra función es realizar chequeos semánticos en las declaraciones del código a compilar.
- *DataType:* Contiene lo fundamental para representar los tipos de datos de `tinyRust+`.
- *Utils:* Contiene pequeñas utilidades para facilitar ciertas tareas relacionadas con traducción (*json*, *mips32).*
- *Backend:* Junto a las dos representaciones intermedias (AST y tabla de símbolos), contiene la clase *CodeGenerator,* que será la que principalmente traduzca estas representaciones a instrucciones de MIPS32.

A continuación, un diagrama de paquete UML del proyecto.

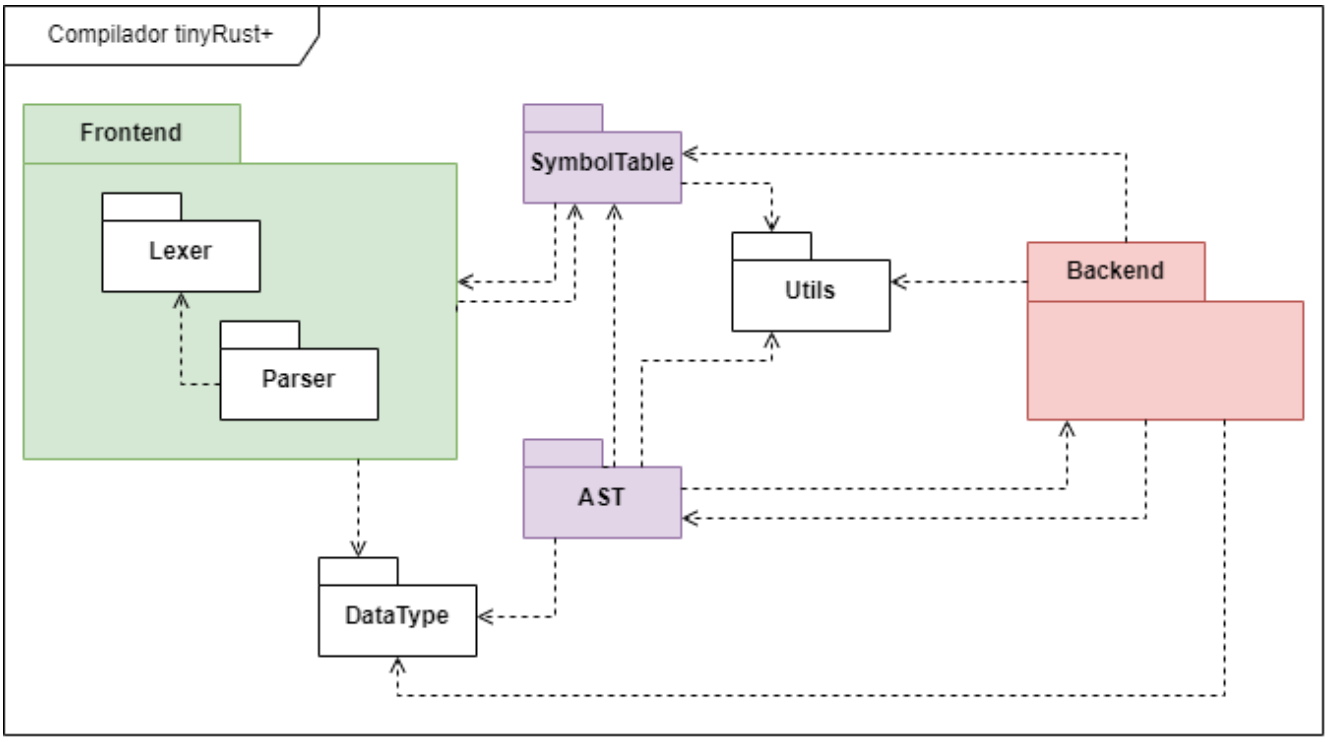


Figura 1: Diagrama de paquete UML de tinyRust+

4 Análisis léxico

El análisis léxico, también conocido como escaneo, es el paso inicial para poder analizar y compilar programas escritos en este lenguaje, y su función es tomar el código fuente del programa, eliminar comentarios y espacios en blanco, convertirlo en una serie de tokens que se pueden procesar y analizar más adelante en el proceso de compilación y también identificar errores léxicos.

A continuación se describirá el proceso de construcción del analizador léxico y su implementación en como se identificarán, se separarán y procesarán los lexemas del código fuente. Se abordarán los siguientes aspectos: la elección del alfabeto de entrada, la definición de las expresiones regulares, las estructuras de datos utilizadas y la estrategia implementada.

4.1 Alfabeto de entrada

El alfabeto de entrada es el conjunto de símbolos que el analizador léxico reconocerá y procesará en un programa fuente. El alfabeto de entrada debe incluir todos los símbolos que aparecen en el programa fuente. Si se omiten algunos símbolos, el analizador léxico no podrá reconocerlos y se producirán errores. Para representarlo se utilizó la clase Alphabet, la cual, mediante un HashSet, se almacenan los símbolos de forma eficiente.

El conjunto que representa el alfabeto de entrada es el siguiente:

$$\Sigma = \{a, \ldots, n, o, \ldots, z, A, \ldots, N, O, \ldots, Z, 0, \ldots, 9, \_, \text{"}, \text{'}, \backslash, |, \&, \%, +, -, *, \hat{}, /, ¿, ?, ¡, !, @, \#, (,), \{, \}, [,], -, ,, ., ;, espacio, \backslash n, \backslash r, \backslash t, :, <, >, =, \$, EOF\}.$$

Por cuestiones de simplicidad se optó por utilizar un alfabeto simple. Observar que el alfabeto no incluye el símbolo 'ñ'. También cabe destacar que '\n' es el carácter nueva línea, '\r' es el retorno de carro, '\t' es la tabulación horizontal y *espacio* es un espacio en blanco.

4.2 Tokens reconocidos y expresiones regulares

Los tokens son las piezas lógicas del programa fuente asociados a un lexema y se identifican mediante patrones específicos del texto. A continuación, se presentan los tokens reconocidos por el analizador léxico, junto con sus expresión regulares asociadas:

Token ID. Subcadena que empieza con una letra o el símbolo '_' y sigue con una secuencia de letras, símbolo '_' y/o dígitos. Se reconoce mediante la expresión regular:

*([a..z] + [A..Z] + _)•([a..z] + [A,,Z] + _ + [0..9])**

Token NUM. Secuencias de dígitos que representan números enteros. Se reconocen mediante la expresión regular: *[0..9]•[0..9])**

Token CHAR. Subcadena de un un solo carácter entre comillas simples. Se reconoce mediante la expresión regular: *[a..z] + [A..Z] +_ + [0..9] + " + \ + | + & + % + + + - + * + / + ¿ + ? + ¡ + ! + @ + # + (+) + { + } + [+] + - + , + . + ; + : + < + > + = + $ + ' + espacio*

Token STR. Subcadena de un un solo carácter entre comillas simples. Se reconoce mediante la expresión regular: *([a..z] + [A..Z] + _ + [0..9] + " + \ + | + & + % + '+' + - + * + / + ¿ + ¡ + ! + @ + # + (+) + { + } + [+] + - + ',' + . + ; + : + < + > + = + $ + espacio + \t)**

Token AS. Operador *asignación*. Se reconoce mediante la expresión regular:

=

Token EQ. Operador *igual*. Se reconoce mediante la expresión regular:

==

Token AND. Operador *and*. Se reconoce mediante la expresión regular:

&&

Token OR. Operador *or*. Se reconoce mediante la expresión regular:

||

Token L: operador *menor*. Se reconoce mediante la expresión regular:

<

Token LE. Operador *menor o igual*. Se reconoce mediante la expresión regular:

<=

Token G. Operador *mayor*. Se reconoce mediante la expresión regular:

>

Token GE. Operador *mayor o igual*. Se reconoce mediante la expresión regular:

>=

Token SUBS. Operador *resta.* Se reconocer mediante la expresión regular:

-

Token RETT. Operador *tipo de retorno*. Se reconoce mediante la expresión regular:

->

Token NEQ. Operador *desigualdad*. Se reconoce mediante la expresión regular:

!=

Token EXCL. Operador *negación*. Se reconoce mediante la expresión regular:

!

Token COMMA. Coma. Se reconoce mediante la expresión regular:

,

Token SEMMI. Punto y coma. Se reconoce mediante la expresión regular:

;

Token DOT. Descriptor de acceso a propiedad. Se reconoce mediante la expresión regular:

.

Token LBRACE. Llave abierta. Se reconoce mediante la expresión regular:

{

Token RBRACE. Llave cerrado. Se reconoce mediante la expresión regular:

}

Token LBRACKET. Corchete abierto. Se reconoce mediante la expresión regular:

[

Token RBRACKET. Corchete cerrado. Se reconoce mediante la expresión regular:

]

Token LPAREN. Paréntesis abierto. Se reconoce mediante la expresión regular:

(

Token RPAREN. Paréntesis cerrado. Se reconoce mediante la expresión regular:

)

Token INH. Indicador de *herencia*. Se reconoce mediante la expresión regular:

:

Token ADD. Operador *suma*. Se reconoce mediante la expresión regular:

+

Token MULT. Operador *multiplicación*. Se reconoce mediante la expresión regular:

*

Token REM. Operador *resto*. Se reconoce mediante la expresión regular:

%

4.3 Diseño de clases Lexer

Se han identificado las clases relevantes para el analizador léxico y se han definido sus comportamientos y relaciones. A continuación, se presenta el diseño de clases junto a sus descripciones.

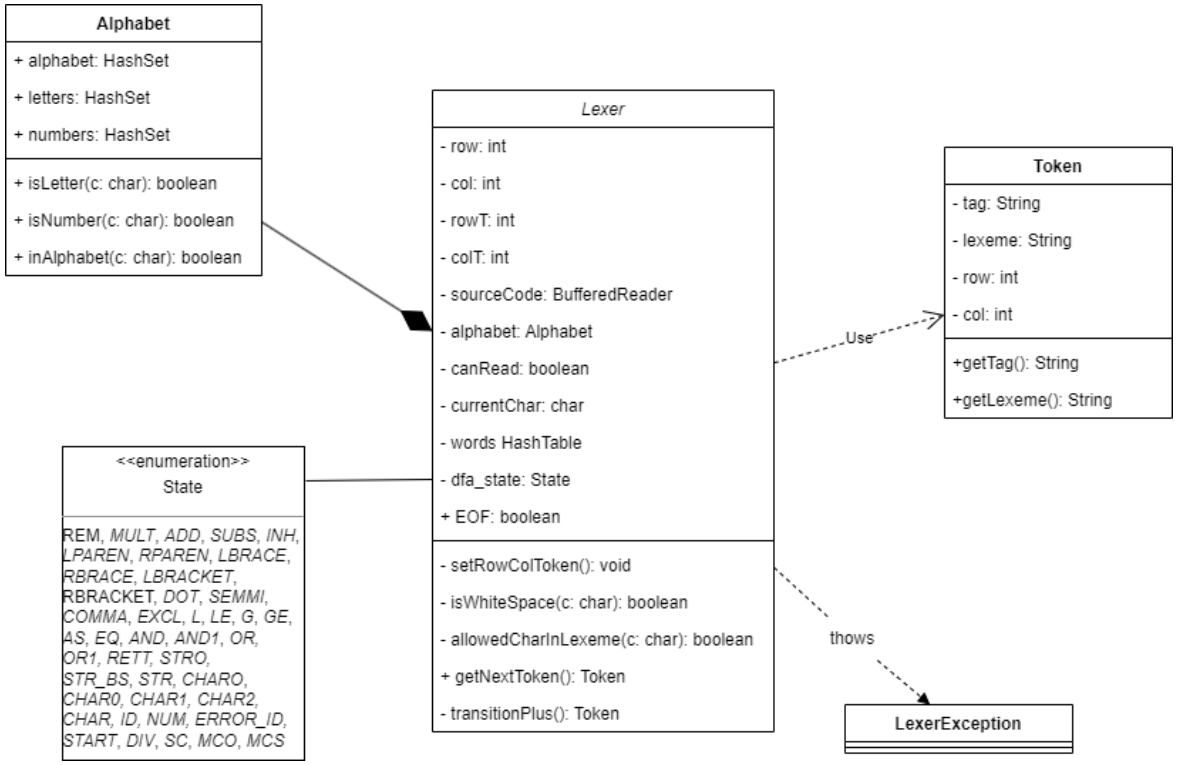


Figura 2: Diagrama UML del paquete Lexer

4.3.1 Clase Lexer

La clase Lexer representa al componente principal del analizador léxico único, perteneciente al compilador.

- Atributos:
  - *row* (tipo de dato: int): la fila donde se posiciona el lector de caracteres.
  - *col* (tipo de dato: int): la columna donde se posiciona el lector de caracteres.
  - *rowT* (tipo de dato: int): la fila donde se posiciona el posible token siguiente.

- *colT* (tipo de dato: int): la columna donde se posiciona el posible token siguiente.
- *sourceCode* (tipo de dato: BufferedReader): el código fuente a escanear.
- *alphabet* (tipo de dato: Alphabet): el objeto alfabeto.
- *canRead* (tipo de dato: boolean): variable que permite determinar si puede leer el siguiente carácter o no.
- *currentChar* (tipo de dato: char): carácter que está siendo leído actualmente.

- *words* (tipo de dato: HashTable<String, String>): tabla con las palabras claves reservadas.
- *dfa_state* (tipo de dato: State): estado actual del autómata finito usado para reconocer lexemas.
- *EOF* (tipo de dato: int): variable que indica si se ha llegado al fin del archivo.

- Métodos:
  - *setRowColToken()*: método privado que establece la ubicación de un posible token.
  - *isWhiteSpace(c: char)*: método privado que determina si el carácter pasado por parámetro es un espacio en blanco.
  - *allowedCharInLexeme(c: char)*: método privado que determina si el carácter pasado por parámetro está permitido en el lexema actual.
  - *getNextToken()*: método público que retorna el próximo token en el programa fuente.
  - *transitionPlus()*: método privadado que representa la transición de un autómata, con características adicionales para el manejo de errores y el retorno de los tokens.

4.3.2 Clase Alphabet

La clase Alphabet representa al conjunto de todos los símbolos del alfabeto de entrada del lenguaje. Además, posee métodos para verificar la pertenencia al conjunto o subconjuntos.

- Atributos:
  - *alphabet (tipo de dato: HashSet<Char>)*: conjunto de todos los símbolos del alfabeto de entrada,
  - *letters (tipo de dato: HashSet<Char>)*: subconjunto de alphabet con todos las letras,
  - *numbers (tipo de dato: HashSet<Char>)*: subconjunto de alphabet con todos los números.

- Métodos:
    - *isLetter(c: char)*: determina si un carácter pasado por argumento es una letra perteneciente al alfabeto o no.
    - *isLetter(c: char)*: determina si un carácter pasado por argumento es una número perteneciente al alfabeto o no.

    - *inAlphabet(c: char)*: determina si un carácter pasado por argumento pertenece al alfabeto o no.

4.3.3 Clase Token

Clase representativa de los *Tokens*. Además de su etiqueta y lexema asociada, contiene la fila y columna de su ubicación en el código fuente.

- Atributos:
    - *tag (tipo de dato: String)*: etiqueta que representa al token en sí,
    - *lexeme (tipo de dato: String)*: lexema asociado al token,
    - *row (tipo de dato: int)*: fila del token respecto al programa fuente,
    - *col (tipo de dato: int)*: columna del token respecto al programa fuente.
- Métodos:
    - *getTag()*: obtiene el valor del atributo tag,
    - *getLexeme()*: obtiene el valor del atributo lexeme.

A continuación, se presenta el diagrama de clases en UML que representa el analizado léxico en nuestro compilador. De esta forma es posible visualizar las relaciones entre clases.

4.4 Errores léxicos

Son los errores detectados por el compilador durante la fase de análisis léxico. A continuación, se listan los posibles errores léxicos que pueden ocurrir en `tinyRust+`.

- El símbolo no pertenece al alfabeto de entrada.
- Identificador no válido.
- Salto de línea ilegal dentro de literal *char*.
- Literal carácter invalido.
- Literal carácter invalido. Hay más de un carácter.
- Salto de línea ilegal dentro de literal *string*.
- Carácter ilegal.
- Comentario multilínea sin cerrar.
- Literal *char* sin cerrar.

- Literal *string* sin cerrar.
- Literal carácter inválido
- Operador inválido
    - Operador inválido. Se esperaba && (AND).
    - Operador inválido. Se esperaba || (OR).

5 Análisis sintáctico

En esta etapa del compilador leeremos la secuencia de *tokens* generados en el programa fuente y verificaremos que se cumplas todas las reglas sintácticas que se detallarán en esta sección.

5.1 Gramática inicial (EBNF)

Para definir la sintaxis de `tinyRust+` haremos uso de las gramáticas libres de contexo. Comenzaremos definiendo la gramática inxiar en la forma *Backus-Naur* extendida.

```
Start ::= Clase* Main
Main ::= "fn main ()" BloqueMetodo
Clase ::= "class" "idClase" Herencia? "{" Miembro* "}"
Herencia ::= ":" "idClase"
Miembro ::= Atributo | Constructor | Metodo
Atributo ::= Visibilidad? Tipo ":" ListaDeclaracionVariables ";"
Constructor ::= "create" ArgumentosFormales Bloque
Metodo ::= FormaMetodo? "fn" "idMetodoVariable" "->" TipoMetodo
ArgumentosFormales BloqueMetodo
ArgumentosFormales ::= "(" ListaArgumentosFormales? ")"
ListaArgumentosFormales ::= ArgumentoFormal "," ListaArgumentosFormales |
ArgumentoFormal
ArgumentoFormal ::= Tipo ":" "idMetodoVariable"
FormaMetodo ::= "static"
Visibilidad ::= "pub"
TipoMetodo ::= Tipo | void
Tipo ::= TipoPrimitivo | TipoReferencia | TipoArray
TipoPrimitivo ::= "Bool" | "I32" | "Str" | "Char"
TipoReferencia ::= "idClase"
TipoArray ::= "Array" TipoPrimitivo
ListaDeclaracionVariables::= "idMetodoVariable" | "idMetodoVariable" ","
ListaDeclaracionVariables
BloqueMetodo ::= "{" DeclVarLocales* Sentencia* "}"
DeclVarLocales ::= "var" Tipo ListaDeclaracionVariables
Sentencia ::= ";"
    | Asignacion";"
    | SentenciaSimple ";"
    | "if" "(" Expresion ")" Sentencia
    | "if" "(" Expresion ")" Sentencia "else" Sentencia
    | "while" "(" Expresion ")" Sentencia
    | Bloque
    | "return" Expresion?";"
```

```
Bloque ::= "{" Sentencia* "}"
Asignacion ::= AsignacionVariableSimple "="" Expresion |
AsignacionSelfSimple "="" Expresion
AsignacionVariableSimple  ::= "id" EncadenadoSimple* | "id" "[" Expresion
"]"
AsignacionSelfSimple ::= "self" EncadenadoSimple*
EncadenadoSimple ::= "." "id"
SentenciaSimple ::= "(" Expresion ")"
Expresion ::= ExpOr
ExpOr ::= ExpOr "||" ExpAnd | ExpAnd
ExpAnd ::= ExpAnd "&&" ExpIgual | ExpIgual
ExpIgual ::= ExpIgual OpIgual ExpCompuesta | ExpCompuesta
ExpCompuesta ::= ExpAdd OpCompuesto ExpAdd | ExpAdd
ExpAdd ::= ExpAdd OpAdd 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" | "stringLiteral" |
"charLiteral"
Primario ::= ExpresionParentizada | AccesoSelf | AccesoVar  | LlamadaMetodo
| LlamadaMetodoEstatico | LlamadaConstructor
ExpresionParentizada ::= "(" Expresion ")" Encadenado?
AccesoSelf ::= "self" Encadenado?
AccesoVar ::= "id" Encadenado?
LlamadaMetodo ::= "id" ArgsActuales  Encadenado?
LlamadaMetodoEstático ::= "idClase" "." LlamadaMetodo Encadenado?
LlamadaConstructor ::= "new" "idClase" ArgumentosActuales Encadenado? |
"new" TipoPrimitivo "[" Expresion "]"
ArgumentosActuales ::= "(" ListaExpresiones? ")"
ListaExpresiones ::= Expresion | Expresion "," ListaExpresiones
Encadenado ::= "." LlamadaMetodoEncadenado | "." AccesoVariableEncadenado
LlamadaMetodoEncadenado ::= "id" ArgumentosActuales Encadenado?
AccesoVariableEncadenado ::= "id" Encadenado? | id "[" Expresion "]"
```

5.2 Análisis descendente predictivo

Un analizador sintáctico descendente es un tipo de analizador sintáctico que trabaja siguiendo una estrategia de análisis de arriba hacia abajo. Es decir, comienza con un símbolo inicial y busca una derivación que produzca la cadena de entrada.

En cada paso el problema clave es determinar la regla de producción que debe aplicarsepara un no terminal de la gramática. Una vez elegida la regla de producción, se relacionarán lostokens devueltos por analizador léxico con el cuerpo de la producción (lado derecho).

Para resolver este problema implementaremos un analizador descendente predictivo. Basado en la cadena de entrada que falta procesar, se hará una predicción (sin *backtracking*) de qué regla de producción usar. Para eso se realizaron varias transformaciones para obtener una gramática más adecuada para el análisis sintáctico.

1. Transformación de EBNF a BNF
2. Eliminación de recursividad a la izquierda.
3. Factorización a la izquierda.

5.3 Conversión de EBNF a BNF.

Aplicando este procedimiento sobre la gramática inicial:

- Para cada no terminal opcional (con ?) se realiza lo siguiente:

$$A ::= a\ B?\ C$$

$$A' ::= a\ N\ C\ N ::= B \mid \lambda$$

- Para cada no terminal repetitivo (con *) se realiza lo siguiente:

$$A ::= a\ B^*\ C$$

$$A' ::= a\ N\ C\ N ::= B\ N \mid \lambda$$

Nos queda la siguiente gramática:

```
Start ::= ClaseR Main
ClaseR ::= Clase ClaseR | λ
Main ::= "fn" "main" "(" ")" BloqueMetodo
Clase ::= "class" "idClase" HerenciaO "{" MiembroR "}"
HerenciaO ::= Herencia | λ
MiembroR ::= Miembro MiembroR | λ
Herencia ::= ":" "idClase"
Miembro ::= Atributo | Constructor | Metodo
Atributo ::= VisibilidadO Tipo ":" ListaDeclaracionVariables ";"
VisibilidadO ::= Visibilidad | λ
Constructor ::= "create" ArgumentosFormales Bloque
Metodo ::= FormaMetodoO "fn" "idMetodoVariable" "->" TipoMetodo ArgumentosFormales
BloqueMetodo
FormaMetodoO ::= FormaMetodo | λ
ArgumentosFormales ::= "(" ListaArgumentosFormalesO ")"
ListaArgumentosFormalesO ::= ListaArgumentosFormales | λ
ListaArgumentosFormales ::= ArgumentoFormal "," ListaArgumentosFormales |
ArgumentoFormal
ArgumentoFormal ::= Tipo ":" "idMetodoVariable"
FormaMetodo ::= "static"
Visibilidad ::= "pub"
TipoMetodo ::= Tipo | void
Tipo ::= TipoPrimitivo | TipoReferencia | TipoArray
TipoPrimitivo ::= "Bool" | "I32" | "Str" | "Char"
TipoReferencia ::= "idClase"
TipoArray ::= "Array" TipoPrimitivo
```

```
ListaDeclaracionVariables::= "idMetodoVariable" | "idMetodoVariable" ","
ListaDeclaracionVariables
BloqueMetodo ::= "{" DeclVarLocalesR SentenciaR "}"
DeclVarLocalesR ::= DeclVarLocales DeclVarLocalesR | λ
SentenciaR ::= Sentencia SenteciaR | λ
DeclVarLocales ::= "var" Tipo ListaDeclaracionVariables
Sentencia ::= ";"
    | Asignacion";"
    | SentenciaSimple ";"
    | "if" "(" Expresion ")" Sentencia
    | "if" "(" Expresion ")" Sentencia "else" Sentencia
    | "while" "(" Expresion ")" Sentencia
    | Bloque
    | "return" ExpresionO ";"
ExpresionO ::= Expresion | λ
Bloque ::= "{" SentenciaR "}"
Asignacion ::= AsignacionVariableSimple "="" Expresion | AsignacionSelfSimple "=""
Expresion
AsignacionVariableSimple  ::= "id" EncadenadoSimpleR | "id" "[" Expresion "]"
AsignacionSelfSimple ::= "self" EncadenadoSimpleR
EncadenadoSimple ::= "." "id"
SentenciaSimple ::= "(" Expresion ")"
Expresion ::= ExpOr
ExpOr ::= ExpOr "||" ExpAnd | ExpAnd
ExpAnd ::= ExpAnd "&&" ExpIgual | ExpIgual
ExpIgual ::= ExpIgual OpIgual ExpCompuesta | ExpCompuesta
ExpCompuesta ::= ExpAdd OpCompuesto ExpAdd | ExpAdd
ExpAdd ::= ExpAdd OpAdd ExpMul | ExpMul
ExpMul ::= ExpMul OpMul ExpUn | ExpUn
ExpUn ::= OpUnario ExpUn | Operando
OpIgual ::= "==" | "!="
OpCompuesto ::= "<" | ">" | "<=" | ">="
OpAd ::= "+" | "-"
OpUnario ::= "+" | "-" | "!"
OpMul ::= "*" | "/" | "%"
Operando ::= Literal | Primario EncadenadoO
EncadenadoO ::= Encadenado | λ
Literal ::= "nil" | "true" | "false" | "intLiteral" | "stringLiteral" |
"charLiteral"
Primario ::= ExpresionParentizada | AccesoSelf | AccesoVar  | LlamadaMetodo  |
LlamadaMetodoEstatico | LlamadaConstructor
ExpresionParentizada ::= "(" Expresion ")" EncadenadoO
AccesoSelf ::= "self" EncadenadoO
AccesoVar ::= "id" EncadenadoO
LlamadaMetodo ::= "id" ArgsActuales  EncadenadoO
LlamadaMetodoEstático ::= "idClase" "." LlamadaMetodo EncadenadoO
LlamadaConstructor ::= "new" "idClase" ArgumentosActuales EncadenadoO | "new"
TipoPrimitivo "[" Expresion "]"
ArgumentosActuales ::= "(" ListaExpresionesO ")"
ListaExpresionesO ::= ListaExpresiones | λ
ListaExpresiones ::= Expresion | Expresion "," ListaExpresiones
Encadenado ::= "." LlamadaMetodoEncadenado | "." AccesoVariableEncadenado
LlamadaMetodoEncadenado ::= "id" ArgumentosActuales EncadenadoO
AccesoVariableEncadenado ::= "id" EncadenadoO | id "[" Expresion "]"
```

5.4 Eliminación de recursividad a la izquierda yfactorización a la izquierda.

La recursividad a la izquierda ocurre cuando un símbolo no terminal puede derivar directamente en sí mismo, lo que puede provocar que el analizador sintáctico entre en un bucle infinito al intentar analizar la entrada. La eliminación de la recursividad a la izquierda es esencial para garantizar que el analizador sintáctico funcione correctamente.

La factorización a la izquierda ocurre cuando dos o más producciones tienen un prefijo común. Al factorizar la gramática, se elimina el prefijo común y se divide la producción en dos o más producciones distintas, lo que permite al analizador sintáctico predecir de manera más eficiente cuál será la producción a aplicar.

Para aplicar esto se utilizaron los algoritmos presentes en las siguientes figuras:

Algoritmo 3 Eliminación de λ en el lado derecho de las reglas de producción

Require: $G = (V_{n_1}, V_{t_1}, S_1, P_1)$

while $\exists A \to \lambda$ donde A no es el símbolo no terminal inicial de la gramática **do**
    removemos $A \to \lambda$
    for cada regla $R \to uAv$ **do**
        Agregamos $R \to uv$
    end for
    for cada regla $R \to uAvAw$ **do**
        Agregaremos $R \to uAvw$, $R \to uvAw$ y $R \to uvw$
    end for
    if $\exists R \to A$ **then**
        Agregaremos $R \to \lambda$ siempre que no este $R \to \lambda$ o la hayamos removido antes.
    end if
end while
return G

Figura 3: Algoritmo de eliminación de producciones lambda en el lado derecho de las reglas de producción

Algoritmo 6 Eliminación de la Recursividad Inmediata por la izquierda

Require: $G_1 = (V_{n_1}, V_{t_1}, S_1, P_1)$ con producciones con recursividad inmediata por la izquierda en un terminal A. No debe contener ningún $\alpha_i = \lambda$

Ensure: $G_1 = (V_{n_2}, V_{t_2}, S_1, P_2)$ sin recursividad inmediata por la izquierda en una producción en las reglas del no terminal A.

1: Agrupamos las producciones $A \rightarrow A\alpha_1|A\alpha_2|\cdots A\alpha_m|\beta_1|\beta_2|\cdots|\beta_n$ en donde ninguna β_i empieza con una A.
2: Eliminamos las producciones $A \rightarrow A\alpha_1|A\alpha_2|\cdots A\alpha_m|\beta_1|\beta_2|\cdots|\beta_n$.
3: Agregamos producciones de la forma $A \rightarrow \beta_1 A'|\beta_2 A'|\cdots|\beta_n A'$
4: Agregamos $A' \rightarrow \alpha_1 A'|\alpha_2 A'|\cdots|\alpha_m A'|\lambda$
5: **return** G_1

Figura 4: Algoritmo de eliminación de la Recursividad Inmediata por la izquierda

Algoritmo 7 Eliminación de la Recursividad por la izquierda

Require: $G_1 = (V_{n_1}, V_{t_1}, S_1, P_1)$ sin ciclos ni producciones λ.

Ensure: $G_2 = (V_{n_2}, V_{t_2}, S_1, P_2)$ sin recursividad por la izquierda.

1: Ordenar los no terminales de cierta forma $A_1, A_2, \cdots, A_n$
2: **for each** i de 1 a n **do**
3: **for each** j de 1 a $i-1$ **do**
4: sustituir cada producción de la forma $A_i \rightarrow A_j\gamma$ por las producciones $A_i \rightarrow \delta_1\gamma|\delta_2\gamma|\cdots|\delta_k\gamma$ en donde $A_j \rightarrow \delta_1|\delta_2|\cdots|\delta_k$ sean todas producciones A_j actuales.
5: **end for**
6: eliminar la recursividad inmediata por la izquierda entre las producciones A_i
7: **end for**
8: **return** G_2

Figura 5: Algoritmo de eliminación de la Recursividad por la izquierda

5.5 Gramática final

Aplicando los algoritmos anteriores de forma adecuada (sin factorizar en algunos casos) nuestra gramática resultante es la siguiente:

```
Start ::= ClaseR Main | Main
ClaseR ::= Clase ClaseR | Clase
Main ::= "fn" "main" "(" ")" BloqueMetodo
Clase ::= "class" "idClase" Clase_1
Clase_1 ::= Herencia "{" Clase_2 | "{" Clase_2
Clase_2 ::= MiembroR "}" | "}"
MiembroR ::= Miembro MiembroR | Miembro
Herencia ::= ":" "idClase"
Miembro ::= Atributo | Constructor | Metodo
Atributo ::= Visibilidad Tipo ":" ListaDeclVar ";"    | Tipo ":" ListaDeclVar ";"
Constructor ::= "create" ArgsFormales BloqueMetodo
Metodo ::= FormaMetodo "fn" "id" ArgsFormales "->" TipoMetodo BloqueMetodo
        | "fn" "id" ArgsFormales "->" TipoMetodo  BloqueMetodo
ArgsFormales ::= "(" ArgsFormales_1
ArgsFormales_1 ::= ListaArgsFormales ")" | ")"
ListaArgsFormales ::= ArgFormal | ArgFormal "," ListaArgsFormales
ArgFormal ::= Tipo ":" "id"
FormaMetodo ::= "static"
Visibilidad ::= "pub"
TipoMetodo ::= Tipo | "void"
Tipo ::= TipoPrimitivo | TipoReferencia | TipoArray
```

```
TipoPrimitivo ::= "Bool" | "I32" | "Str" | "Char"
TipoReferencia ::= "idClase"
TipoArray ::= "Array" TipoPrimitivo
ListaDeclVar ::= "id"
        | "id" "," ListaDeclVar2
BloqueMetodo ::= "{" BloqueMetodo_1
BloqueMetodo_1 ::= DeclVarLocalesR BloqueMetodo_2
        | SentenciaR "}" | "}"
BloqueMetodo_2 ::= SentenciaR "}" | "}"
DeclVarLocalesR ::= DeclVarLocales DeclVarLocalesR | DeclVarLocales
SentenciaR ::= Sentencia SentenciaR2 | Sentencia
DeclVarLocales ::= Tipo ":" ListaDeclVar ";"
Sentencia ::= ";"
        | Asignacion ";"
        | SentSimple ";"
        | "while" "(" Expresion ")" Sentencia
        | Bloque
        | If
        | "return" Return
If ::= "if" "(" Expresion ")" Sentencia Else
        | "if" "(" Expresion ")" Sentencia
Else ::= "else" Sentencia
Return ::= Expresion ";" | ";"
Bloque ::= "{" BloqueMetodo_1
Bloque_1 ::= SentenciaR "}" | "}"
Asignacion ::= AsignVarSimple "=" Expresion | AsignSelfSimple "=" Expresion
AsignVarSimple ::= "id" AsignVarSimple_1
AsignVarSimple_1 ::= EncadenadoSimpleR | "[" Expresion "]"
AsignSelfSimple ::= "self" EncadenadoSimpleR
EncadenadoSimpleR ::= EncadenadoSimple EncadenadoSimpleR2 | EncadenadoSimple
EncadenadoSimple ::= "." "id"
SentSimple ::= "(" Expresion ")"
Expresion ::= ExpAnd ExpresionRD | ExpAnd
ExpresionRD ::= "||" ExpAnd ExpresionRD2 | "||" ExpAnd
ExpAnd ::= ExpIgual ExpAndRD | ExpIgual
ExpAndRD ::= "&&" ExpIgual ExpAndRD2 | "&&" ExpIgual
ExpIgual ::= ExpCompuesta ExpIgualRD | ExpCompuesta
ExpIgualRD ::= OpIgual ExpCompuesta ExpIgualRD2 | OpIgual ExpCompuesta
ExpCompuesta ::= ExpAdd1 OpCompuesto ExpAdd2 | ExpAdd
ExpAdd ::= ExpMul ExpAddRD | ExpMul
ExpAddRD ::= OpAdd ExpMul ExpAddRD2 | OpAdd ExpMul
ExpMul ::= ExpUn ExpMulRD  | ExpUn
ExpMulRD ::= OpMul ExpUn ExpMulRD2 | OpMul ExpUn
ExpUn ::= OpUnario ExpUn2  | Operando
OpIgual ::= "==" | "!="
OpCompuesto ::= "<" | ">" | "<=" | ">="
OpAdd ::= "+"  | "-"
OpUnario ::= "+" | "-" | "!"
OpMul ::= "*"  | "/" | "%"
Operando ::= Literal | Primario Encadenado | Primario
Literal ::= "nil" | "true" | "false" | "intLiteral" | "stringLiteral" |
"charLiteral"
Primario ::= ExprPar | AccesoSelf | VarOMet | LlamadaMetEst | LLamadaConst
ExprPar ::= "(" Expresion ")" Encadenado | "(" Expresion ")"
AccesoSelf ::= "self" Encadenado | "self"
VarOMet ::= "id" VarOMet_1 | "id"
```

```
VarOMet_1 ::= Encadenado  | ArgsActuales | ArgsActuales Encadenado  | "["
Expresion "]"
LlamadaMet ::= "id" ArgsActuales Encadenado      | "id" ArgsActuales
LlamadaMetEst ::= "idClase" "." LlamadaMet Encadenado      | "idClase" "."
LlamadaMet
LLamadaConst ::= "new" LLamadaConst_1
LLamadaConst_1 ::= "idClase" ArgsActuales Encadenado | "idClase" ArgsActuales |
TipoPrimitivo "[" Expresion "]"
ArgsActuales ::= "(" ArgsActuales_1
ArgsActuales_1 ::= ListaExpresiones ")"
ListaExpresiones ::= Expresion | Expresion "," ListaExpresiones2
Encadenado ::= "." Encadenado_1
Encadenado_1 ::= "id" VarOMet_1  | "id"
```

Cabe destacar que la gramática final no es LL(1), ya que hay reglas que no han sido factorizadas a izquierda y en el caso de las sentencias condicionales, existe ambigüedad que luego es resuelta en el AST.

5.6 Conjuntos Primero y Siguiente

El conjunto de primeros y siguientes de una gramática es esencial para la construcción de un analizador sintáctico descendente predictivo recursivo. El conjunto de primeros de un símbolo no terminal es el conjunto de símbolos terminales que pueden aparecer como primer símbolo de una cadena derivada del símbolo no terminal. Por otro lado, el conjunto de siguientesde un símbolo no terminal es el conjunto de símbolos terminales que pueden aparecer inmediatamente después del símbolo no terminal en una cadena derivada.

El conjunto de primeros y siguientes es importante para el analizador sintáctico descendente predictivo recursivo porque permite predecir qué producción aplicar en función delsímbolo de entrada actual y del siguiente símbolo esperado.

Para calcular los conjuntos primeros y siguientes de toda la gramática, utilizamos los siguientes algoritmos:

1. Si X es un terminal, entonces PRIMERO$(X) = \{X\}$.
2. Si X es un no terminal y $X \to Y_1 Y_2 \cdots Y_k$ es una producción para cierta $k \geq 1$, entonces se coloca a en PRIMERO(X) si para cierta i, a está en PRIMERO(Y_i), y ϵ está en todas las funciones PRIMERO(Y_1), ..., PRIMERO(Y_{i-1}); es decir, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. Si ϵ está en PRIMERO(Y_j) para todas las $j = 1, 2, \ldots, k$, entonces se agrega ϵ a PRIMERO(X). Por ejemplo, todo lo que hay en PRIMERO(Y_1) se encuentra sin duda en PRIMERO(X). Si Y_1 no deriva a ϵ, entonces no agregamos nada más a PRIMERO(X), pero si $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, entonces agregamos PRIMERO(Y_2), y así sucesivamente.
3. Si $X \to \epsilon$ es una producción, entonces se agrega ϵ a PRIMERO(X).

Figura 6: Algoritmo para calcular conjuntos Primero de la gramática

1. Colocar \$ en SIGUIENTE(S), en donde S es el símbolo inicial y \$ es el delimitador derecho de la entrada.

2. Si hay una producción $A \rightarrow \alpha B \beta$, entonces todo lo que hay en PRIMERO(β) excepto ϵ está en SIGUIENTE(B).

3. Si hay una producción $A \rightarrow \alpha B$, o una producción $A \rightarrow \alpha B \beta$, en donde PRIMERO($\beta$) contiene a ϵ, entonces todo lo que hay en SIGUIENTE(A) está en SIGUIENTE(B).

Figura 7: Algoritmo para calcular conjuntos Siguientes de la gramática.

Los conjuntos Primeros de la gramática final son:

```
Primeros(Start) = "class", "fn"
Primeros(ClaseR) = "class"
Primeros(Main) = "fn"
Primeros(Clase) = "class"
Primeros(Clase_1) = ":", "{"
Primeros(Clase_2) = "pub", "Bool", "I32", "Str", "Char", "idClase", "Array",
"create", "static", "fn", "}"
Primeros(MiembroR) = "pub", "Bool", "I32", "Str", "Char", "idClase", "Array",
"create", "static", "fn"
Primeros(Herencia) = ":"
Primeros(Miembro) = "pub", "Bool", "I32", "Str", "Char", "idClase", "Array",
"create", "static", "fn"
Primeros(Atributo) = "pub", "Bool", "I32", "Str", "Char", "idClase", "Array"
Primeros(Constructor) = "create"
Primeros(Metodo) = "static", "fn"
Primeros(ArgsFormales) = "("
Primeros(ArgsFormales_1) = "Bool", "I32", "Str", "Char", "idClase", "Array", ")"
Primeros(ListaArgsFormales) = "Bool", "I32", "Str", "Char", "idClase", "Array"
Primeros(ArgFormal) = "Bool", "I32", "Str", "Char", "idClase", "Array"
Primeros(FormaMetodo) = "static"
Primeros(Visibilidad) = "pub"
Primeros(TipoMetodo) = "Bool", "I32", "Str", "Char", "idClase", "Array", "void"
Primeros(Tipo) = "Bool", "I32", "Str", "Char", "idClase", "Array"
Primeros(TipoPrimitivo) = "Bool", "I32", "Str", "Char"
Primeros(TipoReferencia) = "idClase"
Primeros(TipoArray) = "Array"
Primeros(ListaDeclVar) = "idMetodoVariable"
Primeros(BloqueMetodo) = "{"
Primeros(BloqueMetodo_1) = "Bool", "I32", "Str", "Char", "idClase", "Array", "if",
";", "id", "(", "while", "{", "return", "}", "self"
Primeros(BloqueMetodo_2) = "if", ";", "id", "(", "while", "{", "return", "}",
"self"
Primeros(DeclVarLocalesR) = "Bool", "I32", "Str", "Char", "idClase", "Array"
Primeros(SentenciaR) = "if", ";", "id", "(", "while", "{", "return", "self"
Primeros(DeclVarLocales) = "Bool", "I32", "Str", "Char", "idClase", "Array"
Primeros(Sentencia) = "if", ";", "id", "(", "while", "{", "return"
Primeros(Bloque) = "{"
Primeros(Bloque_1) = "if", ";", "id", "(", "while", "{", "return", "}"
Primeros(Asignacion) = "id", "self"
Primeros(AsignVarSimple) = "id"
Primeros(AsignVarSimple_1) = ".", "["
Primeros(AsignSelfSimple) = "self"
```

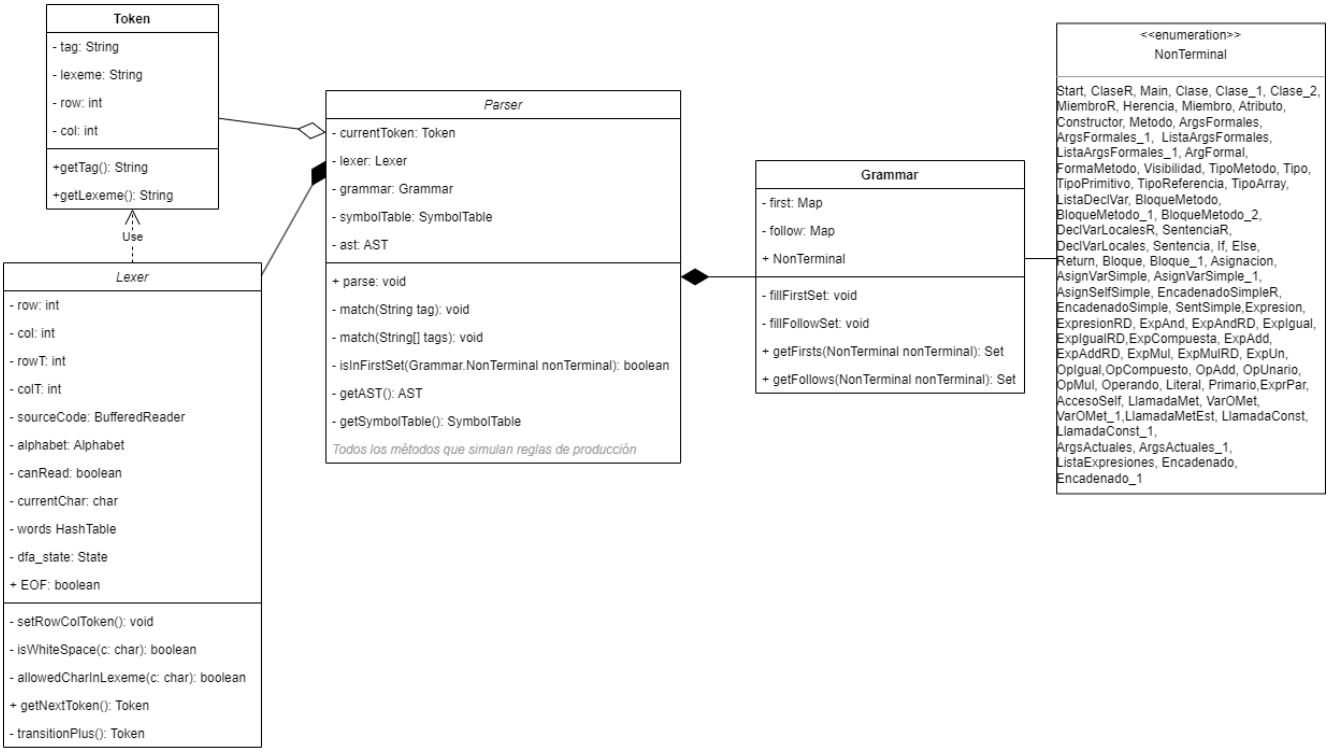
```
Primeros(EncadenadoSimpleR) = "."
Primeros(EncadenadoSimple) = "."
Primeros(SentSimple) = "("
Primeros(Expresion) = "+", "-", "!", "nil", "true", "false", "intLiteral",
"stringLiteral", "charLiteral", "(", "self", "id", "idClase", "new"
Primeros(ExpresionRD) =  "||"
Primeros(ExpAnd) = "+", "-", "!", "nil", "true", "false", "intLiteral",
"stringLiteral", "charLiteral", "(", "self", "id", "idClase", "new"
Primeros(ExpAndRD) = "&&"
Primeros(ExpIgual) = "+", "-", "!", "nil", "true", "false", "intLiteral",
"stringLiteral", "charLiteral", "(", "self", "id", "idClase", "new"
Primeros(ExpIgualRD) = "==", "!="
Primeros(ExpCompuesta) = "+", "-", "!", "nil", "true", "false", "intLiteral",
"stringLiteral", "charLiteral", "(", "self", "id", "idClase", "new"
Primeros(ExpAdd) = "+", "-", "!", "nil", "true", "false", "intLiteral",
"stringLiteral", "charLiteral", "(", "self", "id", "idClase", "new"
Primeros(ExpAddRD) = "+", "-"
Primeros(ExpMul) = "+", "-", "!", "nil", "true", "false", "intLiteral",
"stringLiteral", "charLiteral", "(", "self", "id", "idClase", "new"
Primeros(ExpMulRD) = "*", "/", "%"
Primeros(ExpUn) = "+", "-", "!", "nil", "true", "false", "intLiteral",
"stringLiteral", "charLiteral", "(", "self", "id", "idClase", "new"
Primeros(OpIgual) = "==", "!="
Primeros(OpCompuesto) = "<", ">", "<=", ">="
Primeros(OpAdd) = "+", "-"
Primeros(OpUnario) = "+", "-", "!"
Primeros(OpMul) = "*", "/", "%"
Primeros(Operando) = "nil", "true", "false", "intLiteral", "stringLiteral",
"charLiteral", "(", "self", "id", "idClase", "new"
Primeros(Literal) = "nil", "true", "false", "intLiteral", "stringLiteral",
"charLiteral"
Primeros(Primario) = "(", "self", "id", "idClase", "new"
Primeros(ExprPar) = "("
Primeros(AccesoSelf) = "self"
Primeros(VarOMet) = "id"
Primeros(VarOMet_1) = ".", "(", "["
Primeros(LlamadaMet) = "id"
Primeros(LlamadaMetEst) = "idClase"
Primeros(LLamadaConst) = "new"
Primeros(LLamadaConst_1) = "idClase", "Bool", "I32", "Str", "Char"
Primeros(ArgsActuales) = "("
Primeros(ArgsActuales_1) = ")", "+", "-", "!", "nil", "true", "false",
"intLiteral", "stringLiteral", "charLiteral", "(", "self", "id", "idClase", "new"
Primeros(ListaExpresiones) = "+", "-", "!", "nil", "true", "false", "intLiteral",
"stringLiteral", "charLiteral", "(", "self", "id", "idClase", "new"
Primeros(Encadenado) = "."
Primeros(Encadenado_1) = "id"
```

Debido a que no hay producciones *lambda* en el conjunto de primeros, no fué de utilidad disponer de los conjunto Siguiente, y por lo tanto no se incluye.

5.7 Diseño

En el diseño de esta etapa se incluyo un paquete *Parser* que contiene dos clases principales: *Parser* y *Grammar*, y una clase para manejar errores sintácticos llamada *ParserException*. Además del modelado de las características y funcionalidades del analizador sintáctico, se tienen en cuenta comportamientos que son útiles en la siguiente etapa: análisis semántico.

A continuación, se presentan el diagrama de las clases construidas, junto a sus descripciones.



5.8 Clase Parser

La clase Parser representa al analizador sintáctico único, perteneciente al compilador.

- Atributos:
    - currentToken (tipo de dato: Token). Token actual.
    - lexer (tipo de dato: Lexer). Objeto único y final Lexer (analizador sintáctico).
    - grammar (tipo de dato: Grammar). Objeto Grammar. Contiene no

terminales y los primeros de cada no terminal.

- symbolTable (tipo de dato: SymbolTable). Tabla de símbolos, útil para etapas posteriores.
- ast (tipo de dato: AST). Árbol de sintáxis abstracto, útil para etapas posteriores.

- Métodos:
  - match(tag: String): método que verifica si el token actual “matchea” con el terminal pasadopor argumento.
  - match(tags: String[]): sobrecarga del método anterior que verifica si el token actual“matchea” con alguno de los terminales pasados por argumento,
  - isInFistSet(nonTerminal: NonTerminal): Determina si el token actual se encuentra en el conjunto de primeros del no terminal pasado por argumento,
  - Luego por cada regla de la grámatica final existe un método que simulará la regla de asociada. Si no se cumple una regla sintáctica, el método correspondiente lanzará una excepción con el error sintáctico.

5.9 Clase Grammar

Esta clase representa los no terminales de la gramática junto a sus conjuntosde primeros y siguientes.

- Atributos:
  - first (tipo de dato: Map<NonTerminal, Set>): conjuntos de Primeros de todos los noterminales de la gramática,
  - follow (tipo de dato: Map<NonTerminal, Set>): conjuntos de Siguientes de algunos noterminales de la gramática,
  - Enum NonTerminal: enumerador para representar todos los no terminales de lagramática.
- Métodos:
  - fillFirstSet (): Método de llenado de los conjuntos de Primeros, fillFollowSet (): Método de llenado de los conjuntos de Siguientes.
  - getFirsts(nonTerminal: NonTerminal): getter (público) del conjunto de primeros del noterminal pasado por parámetro,

- getFollows(nonTerminal: NonTerminal): getter (público) del conjunto de siguientes del noterminal pasado por parámetro.

5.10 Errores sintácticos

Durante esta etapa se irán comprobando las reglas sintácticas a medida que se leen *tokens* siguiendo un enfoque descendente. Si no se cumplen dichas reglas (definidas en la gramática final) se lanzará una excepción con una descripción que puede ser:

- Construcción gramatical incorrecta donde se esperaba algún token de un conjunto de posibles tokens, y el token actual no *matchea* con ninguno de estos.
- Después de haber analizado una estructura del método *Main*, se lee cualquier token.
- Se esperaba un método *Main*.
- Se esperaba una Clase y se encontró otra cosa. Y lo mismo para cuando se espera un Atributo, Constructor, Método, Bloque, Tipo, identificador de clase, identificador de variable, tipo de retorno, Tipo Primitivo, Tipo de Referencia, Tipo de Arreglo, estructura condicional, bucle *while*, sentencia simple, retorno del método, asignación, Expresión, Operando, algún tipo de operador (igualdad, aritmético, orden), literal carácter, literal numérico, literal cadena o lista de expresio

6 Análisis semántico

El analizador semántico comprobará la consistencia semántica del programa fuente con la definición de `tinyRust+`, para así garantizar que los nombres de las variables, métodos y clases se estén utilizando correctamente. Si hay inconsistencias o errores, el análisis semántico puede señalarlos para que se corrijan antes de que se ejecute el programa.

Las tareas del analizador semántico se pueden dividir en dos:

- Chequeo de declaraciones. Se recolecta, entiende y controlan todas las entidades declaradas.
- Chequeo de sentencias. Se recolecta, entiende y controlan todas las sentencias asociadas a las entidades recolectadas.

6.1 Chequeo de declaraciones

En nuestra implementación actual de analizador semántico sobre declaraciones se verificará solamente que:

- No haya declaraciones de clases con el mismo identificador.
- Las clases no heredan de clases base inexistentes.
- No se redefinan atributos heredados.
- Dentro de un bloque no se definan atributos o métodos con identificadores repetidos.
- Dentro la lista de argumentos de un método, los identificadores de parámetros sean distintos.
- No hayan definidas herencias circulares.

Ya que solo nos enfocaremos en la tarea de recolectar, entender y controlar todas las entidades declaradas (chequeo de declaraciones).

Para la implementación del analizador semántico se utilizó un *intercalado* con el analizador sintáctico. Los controles semánticos se implementan dentro del analizador sintáctico.

Estas son las reglas semánticas que se han tenido en cuenta en las declaraciones de `tinyRust+`:

- Clases.
    - Todos los nombres de clases son visibles globalmente (no hay clases privadas).
    - No puede haber dos clases con el mismo nombre en el mismo código fuente.
    - Dentro de una clase no se pueden definir métodos con el mismo nombre ni atributos con el mismo nombre. Pero un método y un atributo pueden tener el mismo nombre.
- Atributos.

    - Todos los atributos tienen un alcance privado, salvo aquellos que tengan la palabra reservada *pub*. Si no tienen el modificador de acceso *pub*, solo podrán ser accedidos desde la clase actual.
- Métodos.
    - Todos los métodos tienen alcance global.
    - Aquellos métodos que tengan antes de la palabra *fn* la palabra clave *static* serán métodos de clase.
    - Puede haber 0 o más parámetros formales. Los identificadores utilizados en la lista de parámetros deben ser distintos.
    - Dentro del bloque del método las variables declaradas deben tener nombres distintos. Esto incluye a los nombres de los parámetros formales del método.
- Constructor.
    - Es el único método que puede tener argumentos distintos en una subclase y una superclase.
- Herencia.
    - En el caso de que una superclase y una subclase definan el mismo nombre de método, entonces la definición dada en la subclase tiene prioridad.
    - La redefinición de un método heredado solo es posible siempre que el número de argumentos, los tipos de parámetros formales y el tipo de retorno sean exactamente iguales en ambas definiciones.
    - Un método de clase (*static*) no puede redefinirse.
    - Es ilegal redefinir los nombres de los atributos.
    - Una subclase no puede acceder a atributos de una superclase a menos que sean públicos.
    - Si una definición de clase no especifica de que clase hereda, entonces hereda de *Object* de forma predeterminada.
    - Una clase puede heredar solo de una sola clase (no se permite herencia múltiple).
    - La relación superclase-subclase en las clases define un grafo. Este grafo no puede contener ciclos. Por ejemplo, si A hereda de B, entonces B no debe heredar de A.
    - Una clase hereda de otra, siempre y cuando esta otra clase se declare en algún lugar del código fuente.

- Tipos.
    - En `tinyRust+`, toda clase es un tipo. Antes de compilar se declaran, y añaden a la tabla de símbolos. Las clases base son: *Object*, *I32*, *Str*, *Char*, *Bool* y *IO*.
    - Los tipos de todas las variables deben estar declarados en el código o ser clases base.
    - Polimorfismo. Si un método o variable espera un valor de tipo A, entonces cualquier valor del tipo B se puede utilizar en su lugar, siempre que A sea un antepasado de B en la jerarquía de clases.
    - El sistema de tipos garantiza en tiempo de compilación que la ejecución de un programa no resulte en un error de tipo en tiempo de ejecución.

6.2 Gramática de atributos

Para el analizador sintáctico utilizamos gramáticas libres de contexto, para ser más específicos utilizamos un subconjunto de ellas que incluye a las gramáticas LL(1). Para el analizador semántico no es suficiente con estás gramáticas. Es necesario definir unas gramáticas más ricas.

Las gramáticas de atributos son un tipo de gramática formal que se utiliza para definir y calcular atributos de los elementos de un programa, y que son útiles para diseñar *acciones semánticas* dentro del analizado sintáctico. Estos atributos pueden ser valores, tipos o cualquier otra información que se necesite para el análisis semántico de un programa.

Cada producción en la gramática de atributos se puede asociar con una función que calcula los valores de los atributos de los símbolos no terminales de la producción, utilizando los valores de los atributos de los símbolos terminales que aparecen en la producción.

Los tipos de atributos pueden ser de tres tipos:

- Sintetizados: su valor se calcula en base al valor de los atributos de los nodos hijos, sus atributos y las constantes.
- Heredados: un atributo heredado es aquel definido a partir de los atributos propios del nodo, los de sus hermanos o su padre en el árbol de análisis. El orden en que se calculan los atributos heredados es importante.
- Intrínsecos: un atributos es intrínseco cuando su valor se asume dado de antemano. Un ejemplo claro son los atributos asociados a los tokens. Pueden verse también como atributos sintetizados, los cuales se sintetizan externamente.

La gramática que utilizaremos para diseñar las acciones semánticas contendrá atributos sintetizados y heredados. En el caso de los heredados deben depender solamente de los atributos a su izquierda en la producción, o los atributos heredados del padre. A una gramática de este estilo se le llama *L-atribuidas*, y son las que usan analizadores sintácticos descendentes recursivos.

6.3 Tabla de símbolos

La tabla de símbolos es una estructura de datos fundamental en cualquier compilador. Este mantiene registro de todos las entidades (variables, métodos, clases, etc.) definidas en el programa fuente y sus propiedades, como el tipo de datos, ámbito, posición, lexema asociado entre otros. Cada entidad estará asociada a una entrada en la tabla de símbolos, y dependiendo del tipo de esta, está tendrá sus características propias y un comportamiento diferente lo que indicará como se utilizan de forma correcta y como compilarlas de manera correcta.

En un programa, una entidad puede ser declarada en varios ámbitos o alcances diferentes. Por lo tanto, las tablas de símbolos deben ser capaces de soportar múltiples declaraciones de la misma entidad. Esto implica que la tabla de símbolos debe ser capaz de manejar adecuadamente las resolución de identificadores y la asociación correcta de cada declaración de la entidad con su respectivo ámbito.

La tabla de símbolos contará con una o más estructuras centrales para el análisis semántico que mantiene la información de todas las entidades declaradas. Cada entidad tendrá una entrada dentro de la tabla, y una entrada contendrá toda la información al tipo de entidad que representa. Esta deberá tener como función buscar y obtener la entrada de una entidad en base al lexema del token identificador.

La tabla de símbolos se irá construyendo a medida que se realiza el análisis sintáctico y leemos un entidad. A medida que sucede este se harán ciertos chequeos semánticos y se almacenarán estas entidades. Al finalizar el análisis sintáctico, la tabla de símbolo se deberá revisar para las entidades que representan las clases y sus herencias. Se realiza una consolidación, agregando entidades a las clases que heredan mientras se verifica que se viole ninguna de las reglas semánticas ya mencionadas.

6.4 Árbol de sintáxis abstracta. AST

Una vez finalizadas las fases del análisis léxico y sintáctico, el programa fuente no es nuevamente consultado. El analizador semántico necesita, para realizar la segunda pasada sobre el código, una representación de este lo suficientemente poderosa para capturar la estructura gramatical del flujo de tokens. Para chequear declaraciones utilizamos la tabla de símbolos, para el chequeo de sentencias utilizamos junto a la tabla de símbolos el árbol sintáctico abstracto. Estos describen la estructura gramatical del flujo de tokens de un programa.

En estos árboles nos abstraemos de algunos componentes sintácticos que no son útiles para el semántico o que quedan implícitamente modelados como paréntesis, puntos y comas, comas, etc. Cada nodo interior representa una sentencia, como puede ser una operación binaria, un llamado a un método o una asignación entre otras, y los hijos de estos nodos representan los argumentos correspondientes.

Se deben identificar lo tipos de nodos que va a tener el AST. Cada tipo de nodo tendrá su propio significado, sus propios posibles nodos hijos e información vinculada al nodo para el análisis semántico. Más adelante se mostrará el diseño de cada tipo de nodo utilizado para crear el AST de un programa de `tinyRust+`.

6.5 Chequeo de sentencias

Una vez completado el chequeo de declaraciones, y consolidada la tabla de símbolos, se procederá a realizar el chequeo de sentencias.

El chequeo de sentencias trabaja sobre el cuerpo de los métodos/constructores, verificando que no tenga errores semánticos y recopilando información de qué entidades utiliza y cómo lo hace. Se enfoca en el chequeo de tipos y la resolución de nombres en sentencias y expresiones.

La resolución de nombres se refiere a que cuando se utiliza un identificador en una expresión o sentencia, se controle que sea el nombre de la entidad adecuada. Un nombre en cierto contexto se refiere a cierta entidad, y la resolución de nombres nos permite identificar la entidad adecuada junto a la información en la tabla de símbolos.

6.5.1 Chequeo de tipos

En el lenguaje `tinyRust+` haremos chequeos de tipo estático (en tiempo de compilación) para comprobar que no haya errores de tipo. Es necesario que los tipos de los elementos estén declarados, o sean inferibles estáticamente, como es en el caso de las expresiones. Que sean inferibles significa que a cada sentencia que ocurre en el programa se le puede asignar un tipo. Esto permite evitar que ocurran algunos errores en tiempo de ejecución, aunque no con tanta precisión como lo haría un lenguaje con chequeo de tipo dinámico.

Las reglas que rigen las operaciones permitidas en tipos forman el sistema de tipos del lenguaje. Este tiene cuatro componentes principales:

- Un conjunto de tipos base o tipos primitivos. En `tinyRust+` son las clases I32, Str, Char, Bool, Object, IO.
- Reglas para construir nuevos tipos a partir de los tipos existentes. En `tinyRust+` son las de construcción de clases, propias de un lenguaje orientado a objetos.
- Un método para determinar si dos tipos son equivalentes o compatibles.

- Y reglas para inferir el tipo de cada expresión en un código fuente.

En el chequeo de tipos es donde controlamos si los operadores utilizados en una operación son de tipos válidos. Como segunda pasada, se recorrerá el AST en un simple recorrido en postorden, y por cada sentencia, representada mediante un subárbol del AST del programa, se aplicarán reglas vinculadas a la sobreescritura, polimorfismo, equivalencia/compatibilidad. Más adelante se detallará como se realizan los chequeos en cada nodo particular del AST.

Estas son las reglas semántica tenidas en cuenta a la hora de chequear los tipos de las sentencias:

- **Expresiones aritméticas**. La sintaxis de una expresión aritmética es <*exp1r*> <*op*> <*expr2*>, dónde <op> puede ser “+”, “-“, “*”, “/” o “%”. Primero se chequea el tipo de <*expr1*> y luego el de <*expr2*>. El tipo de ambos debe ser de tipo *I32* para que no haya errores. Lo mismo para las expresiones aritméticas unarias <*op*> <*expr*>*,* <*expr*> debe ser de tipo I32
- **Expresiones de comparación**. . La sintaxis es la misma que antes, pero <*op*> representa a los operadores de comparación
  - Igual (“==”) y desigual (“!=”).El tipo de ambos debe ser el mismo para realizar la comparación. En tipos no básicos, la igualdad (y desigualdad) simplemente verifica la igualdad de la referencia.
  - Mayor (“>”), mayor o igual (“>=”), menor (“<”) y menor o igual (“<=”). El tipo de ambos debe ser *I32*, ya que es el único que tipo que tiene forma de comparar valores mediante el orden natural de los números enteros.
- **Expresiones booleanas** *and* (“&&”), *or* (“||”) y *not* (“!”). El tipo de las expresiones involucradas deberá ser *Bool*.
- **Llamadas a métodos.**
  - Normales. Suelen tener la siguiente forma: *<id>(<expr1>, ..., <expN>).*
    - El método <*id*> deberá estar declarado dentro de la clase actual (de dónde se realiza el llamado).
    - La cantidad de expresiones *N* debe coincidir con la cantidad de parámetros formales que tenga el método <*id*>.
    - El tipo de la expresión número *i,* debe coincidir con el i-ésimo parámetro formal del método <*id*>.
    - El tipo asociado a este llamado será el del tipo de retorno.

- Llamadas a métodos estáticos. Suelen tener la siguiente forma: *<idClase>.<id>(<expr1>, ..., <expN>).*
  - Las reglas son idénticas al llamado a método anterior, con la diferencia de que el método *<id>* deberá tener el modificador *static* y deberá estar declarado dentro de la clase *<idClase>*.
- Llamadas a métodos encadenados*.* Suelen tener la siguiente forma: *<exprA>.<id>(<expr1>, ..., <expN>).*
  - Al igual que en las llamadas a métodos no encadenadas, se verificará la firma del método *<id>* (nombre y parámetros). Y además el tipo asociado a este llamado será el tipo del retorno.
  - La expresión *<exprA>* debe tener asignado el tipo asociado a una clase declarada que contenga el método cuyo identificador es *<id>*.

- **Acceso a variable**. El acceso a una variable puede ser:
  - Normal. Formato *<idVar>*
    - Dentro del método actual se busca la declaración de la variable *<idVar>*. Si no se encuentra, y el método actual no es el *main*. Se buscará que esté declarada en la clase actual (resolución de nombres). El tipo asociado será el tipo de la declaración.
  - Encadenado. Formato: *<expr>.<idVar>*
    - Se verificará que la expresión *<expr>* tenga un tipo asociado a una clase declarada y que contenga el atributo *<idVar>* con el modificador *pub*.
    - El tipo de la sentencia corresponderá al tipo de *<idVar>*.
  - Desde la referencia a la clase actual con *self*. Formato: *self . <idVar>*
    - Se verificará que no se utilice la palabra clave *self* en el ámbito del método *main*.
    - Se comprobará que el atributo *<idVar>* haya sido declarado en la clase actual.
    - El tipo de la sentencia corresponderá al tipo de *<idVar>*.
  - Los accesos a variables de tipo *Array*, comprueban las mismas reglas anteriores, pero adicionando el chequeo de que la expresión del índice de acceso sea de tipo *I32*, y el tipo de la sentencia corresponderá al tipo del arreglo (tipo primitivo).
- **Asignación.** *<ladoIzquierdo> = <expr>*
  - El tipo de el lado izquierdo de la asignación debe *matchear* con el tipo de la expresión *<exp>* del lado derecho.

- **New** (Constructor)
  - Una expresión *new* tiene la forma: *new <tipo> ([<parametrosActuales>]).*
  - Para el caso de los arreglos la expresión *new* queda de la siguiente forma: *new <tipo> [<expresión>]*. El lado izquierdo de la asignación deberá ser del tipo correspondiente a la declaración previa (*Array <tipoPrimitivo>*).
- **Bucle while**
  - Tiene la siguiente forma: *while <condición> <sentencia>.* La condición debe tener asociada un tipo *Bool*.
- **Condicionales**
  - Tiene la siguiente forma: *if <condición> <sentencia> [else <sentencia>].* La condición debe tener asociada un tipo *Bool*.
- **Retorno de método**
  - Solo se utiliza si el tipo de retorno es distinto de *void*.
  - El tipo de la expresión debe *matchear* con el tipo de retorno del método declarado.
  - Se debe verificar que dentro del bloque de un método con tipo de retorno distinto de *void*, contenga una sentencia *return*, para cualquier flujo posible en la ejecución (considerando bucles *while* y condicionales *if-[else]*).

6.6 Esquema de traducción tinyRust+

Las acciones semánticas que utilizaremos en la gramática de atributos para `tinyRust+`, serán porciones de pseudocódigo java. En estas acciones se incluirán las funciones asociadas tanto a atributos sintetizados y heredados, como a el almacenamiento, búsqueda y obtención de entidades y sus características. Se utilizan las acciones realizadas para construir la tabla de símbolos y el AST.

A partir de la gramática final utilizada para el análisis sintáctico, se creo el esquema de traducción utilizado por `tinyRust+` presentado a continuación.

```
Start ::= ClaseR Main {ast=new AST(Main.nodo)}
        | Main {ast=new AST(Main.nodo)}

ClaseR ::= Clase ClaseR {ast.addClass(Clase.nodo);}
        | Clase {ast.addClass(Clase.nodo);}

Main ::= "fn" "main" "(" ")" BloqueMetodo {ST.currentMethod=ST.main;
             main = MethodNode(BloqueMetodo.nodo);}

Clase ::= "class" "idClase" Clase_1 {ClassEntry c = new
             ClassEntry("idClase".lex,true);
```

```
                if (ST.classTable.addClass(c)=!= null) error;
                ST.currentClass=c; Clase1.nodo = new
                ClassNode("idClase");}

Clase_1 ::= Herencia "{" Clase_2 {Clase_2.nodo = Clase_1.nodo;}
        | "{" Clase_2 {ClassEntry c = ST.classTable.get("Object");
                ST.currentClass.setInheritance(c):}

Clase_2 ::= MiembroR "}" {MiembroR.nodo = Clase_2.nodo;}
        | "}" {}

MiembroR ::= Miembro MiembroR2 {Miembro.nodo=MiembroR.nodo;
MiembroR2.nodo=MiembroR.nodo;}
        | Miembro {Miembro.nodo=MiembroR.nodo}

Herencia ::= ":" "idClase" {ClassEntry c =
                ST.classTable.get("idClase".lex);
                ST.currentClass.setInheritance(c);}

Miembro ::= Atributo {}
        | Constructor {Miembro.nodo.addMethod(Constructor.nodo)}
        | Metodo {Miembro.nodo.addMethod(Metodo.nodo)}

Atributo ::= Visibilidad Tipo ":" ListaDeclVar ";"
                {ListaDeclVar.pub = true; ListaDeclVar.tipo = Tipo.tipo;
ListaDeclVar.atributo = true;}
        | Tipo ":" ListaDeclVar ";"
                {ListaDeclVar.pub = false; ListaDeclVar.tipo =
                Tipo.tipo; ListaDeclVar.atributo = true;}

Constructor ::= "create" ArgsFormales BloqueMetodo
                {ConstructorEntry co = new
                ConstructorEntry(ST.currentClass.id);
                ST.currentMethod=co;
                constr=new MethodNode(ST.currentClass.id,
                BloqueMetodo.nodo); Constructor.nodo = constr;}

Metodo ::= FormaMetodo "fn" "id" ArgsFormales "->" TipoMetodo
                BloqueMetodo
                {MethodEntry m = new MethodEntry("id".lex,
                TipoMetodo.tipo, true);
                if (ST.currentClass.addMethod(m)!=null) error;
                ST.currentMethod=m;
                m.setReturnType(TipoMetodo.tipo);
                m=new MethodNode("id", BloqueMetodo.nodo);
                Metodo.nodo=m;}
        | "fn" "id" ArgsFormales "->" TipoMetodo  BloqueMetodo
                {MethodEntry m = new MethodEntry("id".lex,
                TipoMetodo.tipo, false);
                ST.currentClass.addMethod(m); ST.currentMethod=m;
                m.setReturnType(TipoMetodo.tipo);
                m=new MethodNode("id", BloqueMetodo.nodo);
                Metodo.nodo=m;}

ArgsFormales ::= "(" ArgsFormales_1 {}
```

```
ArgsFormales_1 ::= ListaArgsFormales ")" {}
        | ")" {}

ListaArgsFormales ::= ArgFormal {}
        | ArgFormal "," ListaArgsFormales {}

ArgFormal ::= Tipo ":" "id"
              {if (ST.currentMethod.addParameter(new
              ParameterEntry("id".lex, Tipo.tipo)))!=null error;}

FormaMetodo ::= "static" {}

Visibilidad ::= "pub" {}

TipoMetodo ::= Tipo {TipoMetodo.tipo = Tipo.tipo}
        | "void" {TipoMetodo.tipo = "void".lex}

Tipo ::= TipoPrimitivo {Tipo.tipo = TipoPrimitivo.tipo}
        | TipoReferencia {Tipo.tipo = TipoReferencia.tipo}
        | TipoArray {Tipo.tipo = TipoArray.tipo}

TipoPrimitivo ::= "Bool" {TipoPrimitivo.tipo = new
              PrimitiveType("Bool");}
        | "I32" {TipoPrimitivo.tipo = new PrimitiveType("I32");}
        | "Str" {TipoPrimitivo.tipo = new PrimitiveType("Str");}
        | "Char" {TipoPrimitivo.tipo = new PrimitiveType("Char");}

TipoReferencia ::= "idClase" {TipoReferencia.type = new
              ReferenceType("idClase".lex)};

TipoArray ::= "Array" TipoPrimitivo {TipoArray.tipo = new
              ArrayType(TipoPrimitivo.tipo)}

ListaDeclVar ::= "id" {if (ListaDeclVar.attr) {
              if (ST.currentClass.addVariable(new
              AttributeEntry("id".lex, ListaDeclVar.tipo))!=null)
              error;}
              else {
              if (ST.currentMethod.addVariable(new VarEntry("id".lex,
              ListaDeclVar.tipo))!=null) error;}}
        | "id" "," ListaDeclVar2 {if (ListaDeclVar.attr) {
              if (ST.currentClass.addVariable(new
              AttributeEntry("id".lex, ListaDeclVar.tipo))!=null)
              error;}
              else {
              if (ST.currentMethod.addVariable(new VarEntry("id".lex,
              ListaDeclVar.tipo))!=null) error;}
              ListaDeclVar2.tipo=ListaDeclVar.tipo;
              ListaDeclVar2.pub=ListaDeclVar.pub; ListaDeclVar2.attr}

BloqueMetodo ::= "{" BloqueMetodo_1 {BloqueMetodo.nodo =
              BloqueMetodo_1.nodo;}
```

```
BloqueMetodo_1 ::= DeclVarLocalesR BloqueMetodo_2 {b=new BlockNode();
BloqueMetodo_2.nodo = b; BloqueMetodo_1.nodo = b;}
        | SentenciaR "}" {b=new BlockNode(); SentenciaR.nodo=b;
BloqueMetodo_1.nodo=b;}
        | "}" {b=new BlockNode(); BloqueMetodo_1.nodo;}
BloqueMetodo_2 ::= SentenciaR "}"
              {SentenciaR.nodo=BloqueMetodo_2.nodo;}
        | "}" {}

DeclVarLocalesR ::= DeclVarLocales DeclVarLocalesR {}
        | DeclVarLocales {}

SentenciaR ::= Sentencia SentenciaR2
              {SentenciaR.addSentence(Sentencia.nodo);
              SentenciaR2.nodo=SentenciaR.nodo;}
        | Sentencia {SentenciaR.addSentence(Sentencia.nodo);}

DeclVarLocales ::= Tipo ":" ListaDeclVar ";"
              {ListaDeclVar.tipo=Tipo.tipo;
              ListaDeclVar.atributo=false; ListaDeclVar.pub=false}

Sentencia ::= ";" {}
        | Asignacion ";" {Sentencia.nodo=Asignacion.nodo;}
        | SentSimple ";" {Sentencia.nodo=SentSimple.nodo;}
        | "while" "(" Expresion ")" Sentencia {wh=new
              WhileNode(Expresion.nodo, Sentencia.nodo);
              Sentencia.nodo=wh;}
        | Bloque {Sentencia.nodo = Bloque.nodo;}
        | If {Sentencia.nodo=If.nodo;}
        | "return" Return {Sentencia.nodo=Return.nodo;}

If ::= "if" "(" Expresion ")" Sentencia Else {ifelse=new
              ifElseNode(Expresion.nodo, Sentencia.nodo, Else.nodo);
              If.nodo = ifelse;}
        | "if" "(" Expresion ")" Sentencia {if=new
              ifElseNode(Expresion.nodo, Sentencia.nodo); If.nodo =
              ifelse;}

Else ::= "else" Sentencia {Else.nodo=Sentencia.nodo;}

Return ::= Expresion ";" {r=new ReturnNode(Expresion.nodo);
              Return.nodo=r;}
        | ";" {r=new ReturnNode(); Return.nodo=r;}

Bloque ::= "{" Bloque_1 {Bloque.nodo=Bloque_1.nodo;}

Bloque_1 ::= SentenciaR "}" {b=new BlockNode(); SentenciaR.nodo=b;
              Bloque_1.nodo=b;}
        | "}" {b=new BlockNode(); Bloque_1.nodo=b;}

Asignacion ::= AsignVarSimple "=" Expresion {a=new
              AssignNode(AsignVarSimple.nodo, Expresion.nodo);
              Asignacion.nodo=a;}
        | AsignSelfSimple "=" Expresion {a=new
              AssignNode(AsignSelfSimple.nodo, Expresion.nodo);
              Asignacion.nodo=a;}
```

```
AsignVarSimple ::= "id" AsignVarSimple_1 {var=VarNode("id");
              AsignVarSimple.nodo=var;}

AsignVarSimple_1 ::= EncadenadoSimpleR
              {AsignVarSimple_1.nodo.encadenado =
              EncadenadoSimpleR.nodo}
        | "[" Expresion "]"
              {AsignVarSimple_1.nodo.index=Expresion.nodo;}

AsignSelfSimple ::= "self" EncadenadoSimpleR {s=VarNode("self");
              s.chain=EncadenadoSimpleR.nodo; AsignSelfSimple.nodo=}

EncadenadoSimpleR ::= EncadenadoSimple EncadenadoSimpleR2
              {EncadenadoSimple.nodo.encadenado =
              EncadenadoSimpleR2.nodo;
              EncadenadoSimpleR.nodo=EncadenadoSimple.nodo;}
        | EncadenadoSimple
              {EncadenadoSimpleR.nodo=EncadenadoSimple.nodo;}

EncadenadoSimple ::= "." "id" {EncadenadoSimple.nodo=new
              VarNode("id");}

SentSimple ::= "(" Expresion ")"  {SentSimple.nodo=Expresion.nodo;}

Expresion ::= ExpAnd ExpresionRD {ls=ExpAnd.nodo;
              orExp=ExpresionRD.nodo; orExp.leftSide=ls;
              Expresion.nodo=orExp;}
        | ExpAnd {ls=ExpAnd.nodo; Expresion.nodo=ls;}

ExpresionRD ::= "||" ExpAnd ExpresionRD2 {orExp=new BinExpNode("||");
              rs=ExpresionRD2.nodo; rs.leftSide=ExpAnd.nodoM;
              orExp.rightSide=rs; ExpresionRD.nodo=orExp;}
        | "||" ExpAnd {orExp=new BinExpNode("||");
              orExp.rightSide=ExpAnd.nodo; ExpresionRD.nodo=orExp;}

ExpAnd ::= ExpIgual ExpAndRD {ls=ExpIgual.nodo; andExp=ExpAndRD.nodo;
              andExp.leftSide=ls; ExpAnd.nodo=andExp;}
        | ExpIgual {ls=ExpIgual.nodo; ExpAnd.nodo=ls;}

ExpAndRD ::= "&&" ExpIgual ExpAndRD2 {andExp=new BinExpNode("&&");
              rs=ExpAndRD2.nodo; rs.leftSide=ExpIgual.nodo;
              andExp.rightSide=rs; ExpAndRD.nodo=andExp;}
        | "&&" ExpIgual {andExp=new BinExpNode("&&");
              andExp.rightSide=ExpAnd.nodo; ExpresionRD.nodo=andExp;}

ExpIgual ::= ExpCompuesta ExpIgualRD {ls=ExpCompuesta.nodo;
              eqExp=ExpIgualRD.nodo; eqExp.leftSide=ls;
              ExpIgual.nodo=eqExp;}
        | ExpCompuesta {ls=ExpCompuesta.nodo; ExpIgual.nodo=ls;}

ExpIgualRD ::= OpIgual ExpCompuesta ExpIgualRD2 {eqExp=new
              BinExpNode(OpIgual.token); rs=ExpIgualRD2.nodo;
              rs.leftSide=ExpCompuesta.nodo; eqExp.rightSide=rs;
ExpIgualRD.nodo=eqExp;}
        | OpIgual ExpCompuesta {eqExp=new BinExpNode(OpIgual.token);
              eqExp.rightSide=ExpCompuesta.nodo;
ExpIgualRD.nodo=eqExp;}
```

```
ExpCompuesta ::= ExpAdd1 OpCompuesto ExpAdd2 {compExp=new
                BinExpNode(OpCompuesto.token);
                compExp.leftSide=ExpAdd1.nodo;
                compExp.rightSide=ExpAdd2.nodo;
                ExpCompuesta.nodo=compExp;}
        | ExpAdd {ExpCompuesta.nodo=ExpAdd.nodo;}

ExpAdd ::= ExpMul ExpAddRD {ls=ExpMul.nodo; addExp=ExpAddRD.nodo;
                addExp.leftSide=ls; ExpAdd.nodo=addExp;}
        | ExpMul {ExpAdd.nodo=ExpMul.nodo;}

ExpAddRD ::= OpAdd ExpMul ExpAddRD2 {addExp=new
                BinExpNode(OpAdd.token); rs=ExpAddRD2.nodo;
                rs.leftSide=ExpMul.nodo; addExp.rightSide=rs;
                ExpAddRD.nodo=addExp;}
        | OpAdd ExpMul {addExp=newBineExpNode(OpAdd.token);
                addExp.rightSide=ExpMul.nodo; ExpAddRD.nodo=addExp;}

ExpMul ::= ExpUn ExpMulRD {ls=ExpUn.nodo; multExp=ExpMulRD.nodo;
                multExp.leftSide=ls; ExpMul.nodo=multExp;}
        | ExpUn {ExpMult.nodo=ExpUn.nodo;}

ExpMulRD ::= OpMul ExpUn ExpMulRD2 {multExp=new
                BinExpNode(OpMul.token); rs=ExpMulRD2.nodo;
                rs.leftSide=ExpUn.nodo; multExp.rightSide=rs;
                ExpMulRD.nodo=multExp;}
        | OpMul ExpUn {multExp=new BinExpNode(OpMul.token);
                multExp.rightSide=ExpUn.nodo; ExpMulRD.nodo=multExp;}

ExpUn ::= OpUnario ExpUn2 {unExp=new UnExpNode(OpUnario.token);
                unExp.rightSide=ExpUn2.nodo; ExpUn.nodo=unExp;}
        | Operando {ExpUn.nodo=Operando.nodo}

OpIgual ::= "==" {OpIgual.token="==";}
        | "!=" {OpIgual.token="=!";}

OpCompuesto ::= "<" {OpCompuesto.token="<";}
        | ">" {OpCompuesto.token=">";}
        | "<=" {OpCompuesto.token="<=";}
        | ">=" {OpCompuesto.token=">=";}

OpAdd ::= "+" {OpAdd.token="+";}
        | "-" {OpAdd.token="-";}

OpUnario ::= "+" {OpUnario.token="+"}
        | "-" {OpUnario.token=""}
        | "!" {OpUnario.token="!"}

OpMul ::= "*" {OpMul.token="*"}
        | "/" {OpMul.token="/"}
        | "%" {OpMul.token="%"}

Operando ::= Literal {Operando.nodo=Literal.nodo}
        | Primario Encadenado {Primario.nodo.chain=Encadenado.nodo;
Operando.nodo=Primario.nodo;}
        | Primario {Operando.nodo=Primario.nodo;}
```

```
Literal ::= "nil" {t new PrimitiveType("nil"); l=new
              LiteralNode("nil", t); Literal.nodo=l;}
          | "true" {t = new PrimitiveType("Bool"); l=new
              LiteralNode("Bool", t); Literal.nodo=l;}
          | "false" {t = new PrimitiveType("Bool"); l=new
              LiteralNode("Bool", t); Literal.nodo=l;}
          | "intLiteral" {t = new PrimitiveType("I32"); l=new
              LiteralNode("I32", t); Literal.nodo=l;}
          | "stringLiteral" {t = new PrimitiveType("Str"); l=new
              LiteralNode("Str", t); Literal.nodo=l;}
          | "charLiteral" {t = new PrimitiveType("Char"); l=new
              LiteralNode("niCharl", t); Literal.nodo=l;}

Primario ::= ExprPar {Primario.nodo=ExprPar.nodo;}
          | AccesoSelf {Primario.nodo=AccesoSelf.nodo;}
          | VarOMet {Primario.nodo=VarOMet.nodo;}
          | LlamadaMetEst {Primario.nodo=LlamadaMetEst.nodo;}
          | LLamadaConst {Primario.nodo=LLamadaConst.nodo;}

ExprPar ::= "(" Expresion ")" Encadenado {Expresion.nodo.encadenado =
              Encadenado.nodo; ExprPar.nodo=Expresion.nodo}
          | "(" Expresion ")" {ExpPar.nodo=Expresion.nodo;}

AccesoSelf ::= "self" Encadenado {s=new VarNode("self");
              s.encadenado=Encadenado.nodo; AccesoSelf.nodo=s;}
          | "self" {s=new VarNode("self"); AccesoSelf.nodo=s;}

VarOMet ::= "id" VarOMet_1 {VarOMet_1.token="id";
              VarOMet.nodo=VarOMet_1.nodo;}
          | "id" {VarOMet.nodo=new VarNode("id");}

VarOMet_1 ::= Encadenado {v=new VarNode(VarOMet_1.token);
              v.encadenado=Encadenado.nodo; VarOMet_1.nodo=v;}
          | ArgsActuales {c=new CallNode(VarOMet_1.token);
              ArgsActuales.nodo=c; VarOMet_1.nodo=c;}            #
          | ArgsActuales Encadenado {c=new CallNode(VarOMet_1.token);
              ArgsActuales.nodo=c; c.encadenado=Encadenado.nodo;
              VarOMet_1.nodo=c; }
          | "[" Expresion "]" {arr=new ArrayNode(VarOMet_1.token);
              arr.indice=Expresion.nodo; VarOMet_1.nodo = arr;}

LlamadaMet ::= "id" ArgsActuales Encadenado
              {m=new CallNode("id"); ArgsActuales.nodo=m;
              m.encadenado=Encadenado.nodo; LlamadaMet.nodo=m;}
          | "id" ArgsActuales
              {m=new CallNode("id"); ArgsActuales.nodo=m;
              LlamadaMet.nodo=m;}


LlamadaMetEst ::= "idClase" "." LlamadaMet Encadenado
              {sc=new CallNode(); sc.claseMetEst="idClase";
              LlamadaMet.nodo=sc; sc.encadenado=Encadenado.nodo;
              LlamadaMetEst.nodo=sc;}
          | "idClase" "." LlamadaMet
              {sc=new CallNode(); sc.claseMetEst="idClase";
              LlamadaMet.nodo=sc; LlamadaMetEst.nodo=sc; }
```

```
LLamadaConst ::= "new" LLamadaConst_1
              {LlamadaConst.nodo=LLamadaConst_1.nodo;}
LLamadaConst_1 ::= "idClase" ArgsActuales Encadenado {c=new
              CallNode("idClase"); c.isConstructor=trueM;
              ArgsActuales.nodo=c; c.encadenado=Encadenado.nodo;
              LLamadaConst_1.nodo=c;}
         | "idClase" ArgsActuales {c=new CallNode("idClase");
              c.isConstructor=trueM; ArgsActuales.nodo=c;
              LLamadaConst_1.nodo=c;}
         | TipoPrimitivo "[" Expresion "]" {arr=new ArrayNode();
              arr.tipo= TipoPrimitivo.tipo; arr.indice=Expresion.nodo;
              LlamadaConst_1.nodo=arr;}

ArgsActuales ::= "(" ArgsActuales_1
              {ArgsActuales_1.nodo=ArgsActuales.nodo;}

ArgsActuales_1 ::= ListaExpresiones ")"
              {ListaExpresiones.nodo=ArgsActuales_1.nodo;}

ListaExpresiones ::= Expresion
              {ListaExpresiones.nodo.agregarParam(Expresion.nodo);}
         | Expresion "," ListaExpresiones2
              {ListaExpresiones.nodo.agregarParam(Expresion.nodo);
              ListaExpresiones2.nodo=ListaExpresiones.nodo;}

Encadenado ::= "." Encadenado_1 {Encadenado.nodo=Encadenado_1.nodo;}

Encadenado_1 ::= "id" VarOMet_1 {VarOMet_1.token="id";
              Encadenado_1.nodo=VarOMet_1.nodo;}
         | "id" {Encadenado_1.nodo = new VarNode("id");}
```

6.7 Diseño

A continuación se presentará el diseño de los paquetes utilizados para la tabla de símbolos, el árbol de sintaxis abstracta (AST), las representaciones de los tipos de datos y el analizador sintáctico. Estos elementos son fundamentales para el análisis semántico y se mostrará las funcionalidades de cada uno de ellos.

6.7.1 Paquete DataType

En este paquete se contienen las clases que representaran las características de tipado asociado a variables, expresiones y métodos.

Classes

Class	Description
ArrayType	Esta clase representa al tipo Array.
PrimitiveType	Esta clase representa el tipo primitivo.
ReferenceType	Esta clase representa el tipo referencia.
Type	Esta clase representa un tipo, tanto del retorno de una función, de un atributo/variable local o de un argumento.

Figura 8: Clases del paquete DataType junto a sus descripciones.

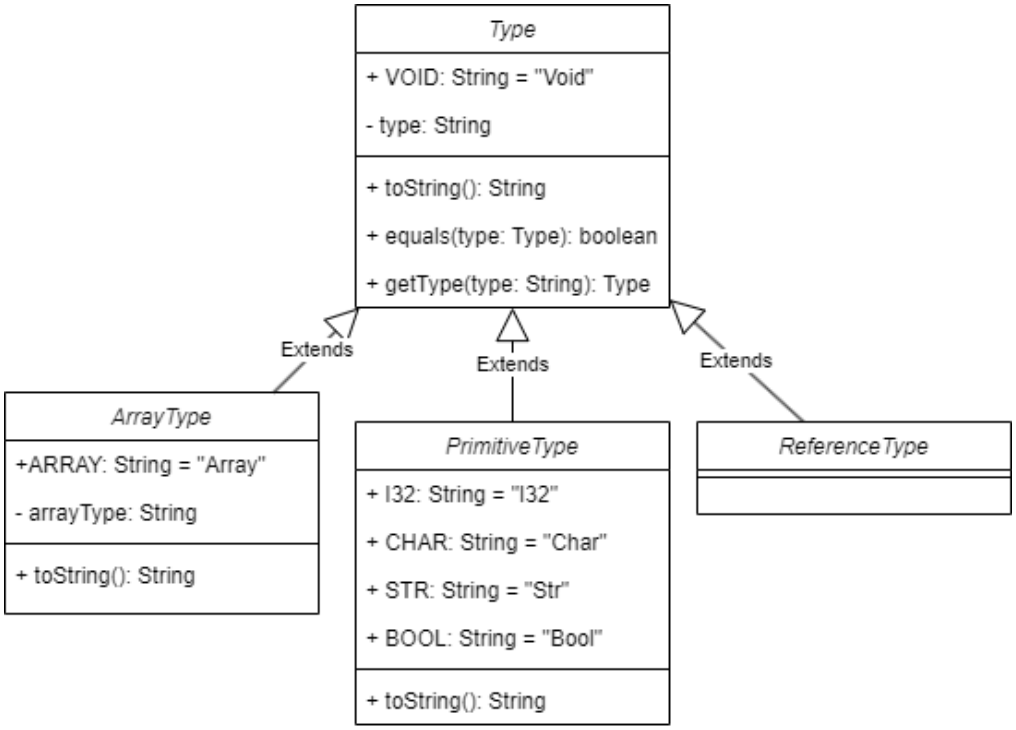


Figura 9: Diagrama UML del paquete DataType.

6.7.1.1 Clase Type

Esta clase representa un tipo, tanto del retorno de una función, de un atributo/variable local o de un argumento.

Contiene constantes declaradas con *String*'s que representan cada identificador de los tipos primitivos, *void* y *Array*.

- Atributos:
  - *type: String*. Nombre del tipo.

- Métodos:
    - *equals(type: Type) -> boolean*: Compara las características de tipo con la de otra instancia de Type.
    - *toString() -> String*: retorna un String con la información formateada del tipo.
    - *(static) createType(type String) -> Type:* crea una nueva instancia del tipo indicado por el parámetro.

6.7.1.2 Clase ArrayType

Esta clase representa al tipo Array. Contiene información sobre el tipo de Array y métodos útiles en las tareas del compilador.

- Hereda de: *Type*.
- Atributos:
    - *arrayType: String*. Tipo del Array (primitivo).
- Métodos:
    - getArrayType(): getter de *arrayType*.
    - toString(): retorna un String con la información del tipo *Array* formateada.

6.7.1.3 Clase PrimitiveType

Esta clase representa a los tipos primitivos. Las valores del nombre del tipo está discretizados en: *I32, Str, Char, Bool*

- Hereda de: *Type*.
- Atributos (heredados).
- Métodos (heredados).

6.7.1.4 Clase ReferenceType

Esta clase representa el tipo referencia. El nombre del tipo será un identificador de clase declarado.

- Hereda de: *Type*.
- Atributos (heredados).
- Métodos (heredados).

6.7.2 Paquete SymbolTable

Contiene todas las clases que darán estructura a la tabla de símbolos y sus entradas.

Classes

Class	Description
AttributeEntry	Esta clase representa los atributos declarados junto a información relevante para el análisis y métodos de utilidad.
ClassEntry	Representa a las clases declarados junto a información que da la identidad de la propia clase.
ConstructorEntry	Esta clase representa las características esenciales de un constructor de clase.
MainEntry	Clase que representa al método main.
Method	Clase que abstrae las características y comportamientos esenciales de todos los tipos de métodos: constructores, main, métodos
MethodEntry	Representa a los métodos de instancia y clase (static) declarados dentro de una clase.
ParameterEntry	Esta clase representa un parámetro (argumento) que tendrá asociado un método.
SymbolTable	Esta clase representa la tabla de símbolos utilizada por el compilador para almacenar la información relevante obtenida de las clases y el método main.
VarEntry	Esta clase representa una entrada de una variable local o atributo, en la tabla de símbolos.

Figura 10: Clases del paquete SymbolTable junto a sus descripciones.

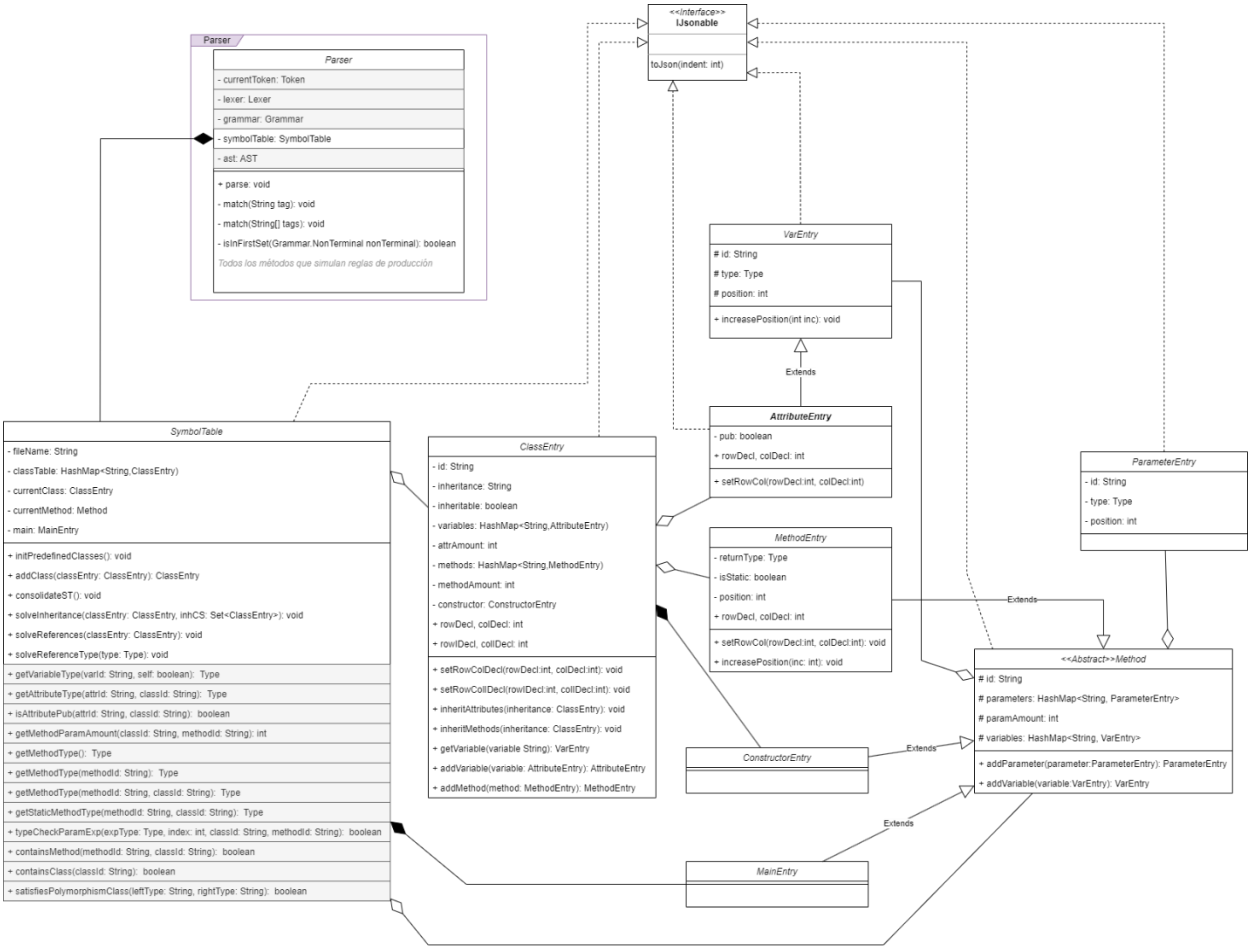


Figura 11: Diagrama UML del paquete SymbolTable.

6.7.2.1 Clase SymbolTable

Esta clase representa la tabla de símbolos utilizada por el compilador para almacenar la información relevante obtenida de las clases y el método main.También tiene una referencia a la clase actual, como tambíen al método actual Además contiene métodos necesarios para la manipulación de la tabla.

- Implementa la interfaz *IJsonable*.
- Atributos:
  - *filename: String*. Nombre del archivo con el código fuente.
  - *classTable HashMap<String, ClassEntry>*. HashMap con las entradas de clases declaradas.
  - *currentClass ClassEntry*. Referencia a la entrada de la clase actual (cuando se está leyendo un *token* dentro de esta clase).
  - *currentMethod Method*. Referencia a la entrada del método o constructor actual.
  - *main: MainEntry*. Referencia a la entrada del método *main*.
- Métodos.

- (Privado) *initPredefinedClasses() -> void*: método privado que inicializa la tabla de símbolos con las clases predefinidas: *Object*, *IO*, *Str*, *I32*, *Bool* y *Char*, junto a sus métodos definidos en la guía de `tinyRust+`. Es llamado en el constructor.
- *addClass(classEntry: ClassEntry) -> void*: método privado que agrega, si no existem una clase en la tabla de clases.
- *consolidateST() -> void*: punto de entrada para consolidar la tabla de símbolos una vez analizado sintácticamente todo el código.
- (Privado) *solveReferences(classEntry: ClassEntry) -> void:* Resuelve las referencias de tipo de atributos y de variables locales (en los métodos de una clase determinada).
- (Privado) *solveReferenceType(type: Type, token: Token, entity: String)*: Determina si se puede tipar una variable/parámetro.
- *solveInheritance() -> void*: método que a medida que resuelve el árbol de herencias de forma recursiva, chequea si existen herencias cíclicas o clases no declaradas.
- *getVariableType(varId: String, self: boolean) -> Type*: Obtiene el tipo de una variable de acuerdo al alcance más cercano.
- *getAttributeType(attrId: String, classId: String) -> Type*: Obtiene el tipo de un atributo de una clase particular.
- *isAttributePub(attrId: String, classId: String) -> boolean*: Determina si un atributo de una clase determinada es de acceso global.
- *getMethodParamAmount(classId*: *String, methodId: String) -> int*: Obtiene la cantidad de parámetros de un método de una clase.
- *getMethodType() -> Type:* Busca y devuelve el tipo de retorno del método actual, que debe estar declarado en la clase actual.
- *getMethodType(methodId: String) -> Type:* Busca y devuelve el tipo de retorno del método especificado, que debe estar declarado en la clase actual.
- *getMethodType(methodId: String, classId: String) -> Type:* Busca y devuelve el tipo de retorno del método especificado con su identificador *methodId* dentro de la clase *classId*.
- *getStaticMethodType(methodId: String, classId: String) -> Type*: Obtiene el tipo de retorno de un método que debe ser estático, y debe estar declarado dentro de la clase especificada.
- *typeCheckParamExp(expType*: *Type, index: int, classId: String, methodId: String) -> boolean*: Verifica que el tipo de un parámetro, de un método con

identificador *methodId* declarado en la clase *classId*, coincida con el tipo de una expresión *expType*.

- *containsMethod(methodId*: *String, classId: String) -> boolean*: Verifica que un método haya sido declarado en la clase especificada.
- *containsClass(classId: String) -> boolean*: Verifica si la clase con identificador *classId* ha sido declarada.
- *satisfiesPolymorphism(leftType*: *String, rightType: String) -> boolean:* Verifica que la clase asociada a *leftType* sea equivalente a la clase *rightType*, o que sea una superclase de la clase *leftType*.
- *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información del objeto en sí, en formato *json*.
- *Getters* y *setters* de los atributos.

6.7.2.2 Clase ClassEntry

Representa a las clases declarados junto a información que da la identidad de la propia clase. Además de otra información y métodos útiles para los chequeos semánticos.

- Atributos:
  - *id*: *String*. Identificador de clase
  - *inheritance: String*. Identificador de clase heredada.
  - *inheritable: boolean*. Indica si es heredable.
  - *variables: HashMap<String, AttributeEntry>.* Mapa con los atributos declarados.
  - *attrAmount: int*. Cantidad de atributos declarados.
  - *methods: HashMap<String, MethodEntry>.* Mapa con los métodos declarados.
  - *methodAmount: int*. Cantidad de métodos declarados.
  - *constructor: ConstructorEntry*. Referencia al constructor de la clase.
  - *rowDecl, colDecl: int*. Localidad del token asociado al identificador de la clase.
  - *rowIDecl, colIlDecl: int*. Localidad del token asociado al identificador de la clase heredada.
- Métodos:
  - *inheritAttributes(inheritance: ClassEntry) -> void*: Hereda los métodos de la clase especificada. Útil en la consolidación de la tabla de símbolos.
  - *inheritMethods(inheritance: ClassEntry) -> void*: Hereda los atributos de la clase especificada. Útil en la consolidación de la tabla de símbolos.
  - *addVariable(variable: AttributeEntry, auto: boolean) -> void*: Agrega, si no existe, un atributo declarado al mapa de atributos. El parámetro auto determina si establecer la posición del atributo sumandole 1 a la cantidad

de atributos declarados antes de ser agregado, o no.

- *addMethod(variable: MethodEntry, auto: boolean) -> void*: Agrega, si no existe, un método declarado al mapa de métodos. El parámetro auto determina si establecer la posición del méotod sumándole 1 a la cantidad de atributos declarados antes de ser agregado, o no.
- *toJson(indents: int) -> void:* método implementado mediante la interfaz *IJsonable*. Retorna un string con la información del objeto en sí, en formato *json.*
- *Getters* y *setters* de los atributos.

6.7.2.3 Clase AttributeEntry

Esta clase representa los atributos declarados junto a información relevante para el análisis y métodos de utilidad.

- Hereda de: *VarEntry*
- Atributos:
  - *pub: boolean*. Indica si el atributo es público.
- Métodos:
  - *toJson(indents: int):* método implementado mediante la interfaz *IJsonable*. Retorna un string con la información del objeto en sí, en formato *json.*
  - *Getters* y *setters* de los atributos.

6.7.2.4 Clase: VarEntry

Esta clase representa a una variable declarada junto a información y comportamiento relevante de esta, útil para la construcción de la tabla de símbolos.

- Atributos:
  - *id: String*. Identificador de la variable declarada.
  - *type: Type*. Objeto que representa el tipo de la variable.
  - *token: Token*. Token asociado al identificador de la variable.
  - *position: int*. Indica la posición de la variable.
- Métodos:
  - *increasePosition(int inc) -> void*: incrementa el valor de *position* en *inc* unidades.
  - *toJson(indents: int) -> void:* método implementado mediante la interfaz *IJsonable*. Retorna un string con la información del objeto en sí, en formato *json.*
  - *Getters* y *setters* de los atributos.

6.7.2.5 Clase Method

Esta clase representa a una variable declarada junto a información y comportamiento relevante de esta, útil para la construcción de la tabla de símbolos.

- Abstracta.
- Atributos:
  - *id: String*. Identificador del método declarada.
  - *parameters: HashMap<String, ParameterEntry>*. HashMap con los parámetros (argumentos) del método.
  - *paramAmount: int*. Cantidad de parámetros.
  - *variables: HashMap<String, VarEntry>*. HashMap con las variables locales del método.
  - *token: Token*. Token asociado al identificador del método.
- Métodos:
  - *containsParameter(parameterId: String) -> boolean*. Indica si el método contiene un parámetro con cierto identificador.
  - *addParameter(parameter: ParameterEntry) -> ParameterEntry*: Agrega, si no existe, un parámetro al Map de parámetros, asignando su posición, e incrementando la variable paramAmount.
  - *addVariable(variable: VarEntry) -> VarEntry*: Agrega, si no existe, una variable al mapa de variables locales *variables*.
  - *toJson(indents: int) -> void:* método implementado mediante la interfaz *IJsonable*. Retorna un string con la información del objeto en sí, en formato *json.*
  - *Getters* y *setters* de los atributos.

6.7.2.6 Clase MethodEntry

Esta clase representa a una variable declarada junto a información y comportamiento relevante de esta, útil para la construcción de la tabla de símbolos.

- Hereda de: *Method*.
- Atributos:
  - returnType: Type. Eipo de retorno.
  - isStatic: boolean. Establece si el método es estático.
  - position: int. Posición del método en la clase.
- Métodos:
  - *increasePosition(inc: int) -> void*: incrementa la posición del método en lo

que indique el parámetro *inc*.

- *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información del objeto en sí, en formato json.
- *Getters* y *setters* de los atributos.

6.7.2.7 Clase ConstructorEntry

Esta clase representa a una variable declarada junto a información y comportamiento relevante de esta, útil para la construcción de la tabla de símbolos.

- Hereda de: Method.
- Atributos (todos heredados).
- Métodos:
  - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información del objeto en sí, en formato json.

6.7.2.8 Clase MainEntry

Clase que representa al método main. Hereda de la clase Method todas las características necesarias para esta.

- Hereda de: *Method*.
- Atributos (todos heredados).

- Métodos:
  - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información del objeto en sí, en formato json.

6.7.2.9 Clase ParameterEntry

Esta clase representa un parámetro (argumento) que tendrá asociado un método. Incluye información relevante al mismo, junto a métodos de utilidad.

- Atributos:
  - *id: String*. Identificador del parámetro.
  - *type: Type*. Objeto que representa el tipo del parámetro.
  - *position: int*. Posición del parámetro dentro de la lista de parámetros.

- *token*: *Token*. Token asociado al identificador del parámetro.

- Métodos:
  - *indexCorrespondence(index: int) -> boolean*: Verifica que la posición pasada por parámetro coincida con la del parámetro que representa la instancia actual.
  - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información del objeto en sí, en formato json.
  - *Getters* y *setters* de los atributos.
  -

6.7.3 Paquete AST

Contiene todas las clases que darán forma al árbol sintáctico abstracto. Junto a la interfaz utilizada para hacer los chequeos de declaraciones.

Class	Description
AccessNode	Esta clase abstracta representa un Nodo de acceso.
ArrayNode	Esta clase representa un Nodo de Arreglo.
AssignNode	Esta clase representa un Nodo de Asignación de un valor a una variable.
AST	La clase ASTNode representa la raíz del árbol de sintaxis abstracta(AST).
BinExpNode	Esta clase representa un Nodo de una Expresión Binaria.
BlockNode	La clase BlockNode representa un bloque de un método, constructor, estructura de control, etc.
CallNode	Esta clase representa un Nodo de un llamado a método.
ClassNode	La clase ClassNode representa un Nodo de una clase.
ExpNode	Clase abstracta que representa a los Nodos Expresiones del AST.
IfElseNode	Esta clase representa un Nodo del AST para una sentencia de control if-else.
ISentenceCheck	Interfaz que contiene el método encargado de chequear los tipos de las sentencias, y propagar estos tipos.
LiteralNode	Clase que representa a un NodoLiteral: literal entero, literal cadena, literal caracter, literal nil, literal bool.
MethodNode	La clase MethodNode representa un Nodo de un método: método de instancia, de clase (static), constructor, método main.
Node	Clase abstracta principal que representa a todos los nodos de un AST.
ReturnNode	Clase que representa a una sentencia return.
SentenceNode	Clase abstracta que representa una sentencia en un método del programa.
UnExpNode	Esta clase representa un Nodo de una Expresión Unaria.
VarNode	Esta clase representa un Nodo de un Acceso a una variable.
WhileNode	Esta clase representa un Nodo del AST para la sentencia de control iterativa while.

Figura 12: Clases del paquete AST junto a sus descripciones.

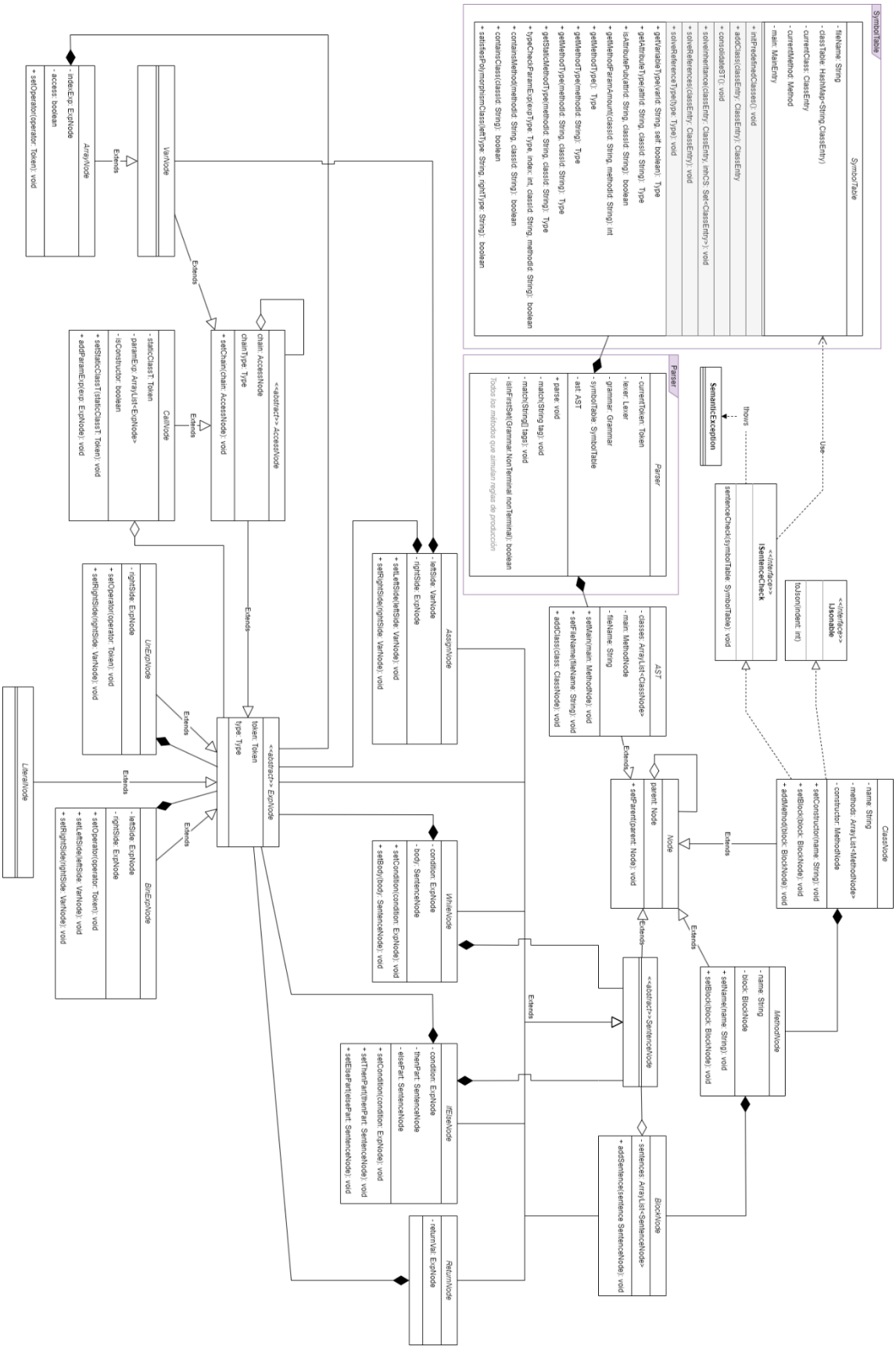


Figura 13: Diagrama UML del paquete AST

6.7.3.1 Interfaz ISentenceCheck

Interfaz que contiene el método encargado de chequear los tipos de las sentencias.

- Método:
  - *sentenceCheck(symbolTable*: *SymbolTable) -> void*. Utilizando la tabla de símbolos, se chequea la validez semántica de la sentencia, se propaga el tipo y también se resuelven nombres.

6.7.3.2 Clase Node

Clase abstracta principal que representa a todos los nodos de un AST. Cada subclase de esta, al agregar nodos hijos establecerá una relación de parentesco consigo misma.

- Abstracta.
- Implementa: *ISentenceCheck*
- Atributos:
  - *parent: Node*. Nodo padre.
- Métodos:
  - *setParent(parent*: *Node) -> void.* Setter de la referencia al nodo padre.

6.7.3.3 Class AST

La clase AST representa la raíz del AST. Contiene información sobre sobre las clases y método *main* del código fuente.

- Hereda de: *Node*.
- Atributos:
  - *fileName: String*. Nombre del archivo.
  - *classes*: *ArrayList<ClassNode>.* Lista de los nodos que representan las clases declaradas en el código fuente.
  - *main*: *MethodNode*. Nodo que representa el método *main*.
- Métodos:
  - *addClass(classNode*: *ClassNode) -> void.* Agrega un nodo método a la lista de métodos.
  - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información relevante del nodo, en formato json.
  - *Getters* y *setters* de los atributos.

6.7.3.4 Clase ClassNode

La clase ClassNode representa un Nodo de una clase. Contiene métodos con sus comportamiento (sentencias), y el método constructor.

- Hereda de: *Node*.
- Atributos:
  - *name: String*. Identificador de la clase.
  - *methods*: *ArrayList<MethodNode>.* Lista de los nodos que representan métodos.
  - *constructor*: *MethodNode*. Nodo constructor de la clase.
- Métodos:
  - *addMethod(method*: *MethodNode) -> void.* Agrega un nodo método a la lista de métodos.
  - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información relevante del nodo, en formato json.
  - *Getters* y *setters* de los atributos.

6.7.3.5 Clase MethodNode

La clase MethodNode representa un Nodo de un método: método de instancia, de clase (static), constructor, método main.

- Hereda de: *Node*.
- Atributos:
  - *name: String*. Identificador del método.
  - *block*: *BlockNode*. Nodo que representa el bloque del método.
- Métodos:
  - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información relevante del nodo, en formato json.
  - *Getters* y *setters* de los atributos.

6.7.3.6 Clase SentenceNode

- Hereda de: *Node*.
- Abstracta.

- Atributos:
    - *hasReturnStmt: boolean*. Determina si la sentencia, o dentro de esta hay un *return*.
- Métodos:
    - *Getter* y *setter* del atributo.

6.7.3.7 Clase BlockNode

La clase *BlockNode* representa un bloque de un método, constructor, estructura de control, etc. Contiene una lista ordenada de las sentencias dentro de este.

- Hereda de: *SentenceNode*.
- Atributos:
    - *sentences: ArrayList<SentenceNode>.* Lista de nodos sentencia.
- Métodos:
    - *addSentence(sentence*: *SentenceNode) -> void.* Agrega un nodo sentencia a la lista de sentencias.
    - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información relevante del nodo, en formato json.
    - *Getter* y *setter* del atributo.

6.7.3.8 Clase ExpNode

Clase abstracta que representa a los Nodos Expresiones del AST. Contiene información sobre el token asociado, y el tipo de la expresión.

- Hereda de: *SentenceNode*.
- Abstracta.
- Atributos:
    - *token: Token*. Token asociado a la expresión
    - *type*: *Type*. Tipo de la expresión.
- Métodos:
    - *Getters* y *setters* de los atributos.

6.7.3.9 Clase BinExpNode

Esta clase representa un Nodo de una Expresión Binaria. Contiene información sobre las expresiones del lado izquierdo y derecho. Y el token representa el operador binario asociado.

- Hereda de: *ExpNode*.
- Atributos:
  - *leftSide: ExpNode*. Lado izquierdo de la expresión binaria.
  - *rightSide*: *ExpNode*. Lado derecho de la expresión binaria.
- Métodos:
  - *setOperator(operator: Token) -> void.* Asocia el token de un operador binario al atributo *token* heredado de *ExpNode.*
  - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información relevante del nodo, en formato json.
  - *Getters* y *setters* de los atributos.

6.7.3.10 Clase UnExpNode

Esta clase representa un Nodo de una Expresión Unaria. Contiene información sobre la expresión del lado derecho y el token del operador asociado.

- Hereda de: *ExpNode*.
- Atributos:
  - *rightSide*: *ExpNode*. Lado derecho de la expresión unaria.
- Métodos:
  - *setOperator(operator: Token) -> void.* Asocia el token de un operador unario al atributo *token* heredado de *ExpNode.*
  - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información relevante del nodo, en formato json.
  - *Getter* y *setter* del atributo.

6.7.3.11 Clase AccessNode

Esta clase abstracta representa un Nodo de acceso. Un Nodo de acceso puede ser un llamado a una función o el acceso a una variable. Este tipo de nodos son encadenables con otros Nodos de esta clase

- Hereda de: *ExpNode*.
- Abstracta.
- Atributos:
  - *chain: AccessNode*. Nodo encadenado a la instancia actual.
  - *chainType*: *Type*. Tipo del nodo en particular. El atributo heredado *type* se usará para tipar la expresión encadenada.

- Métodos:
    - *Getters* y *setters* de los atributos.

6.7.3.12 Clase CallNode

Esta clase representa un Nodo de un llamado a método. Los llamados que incluye son a: constructores, métodos estáticos y no estáticos.

- Hereda de: *AccessNode*.
- Atributos:
    - *staticClassT: Token*. Token de la clase que contiene un método estático. *Null* si no es un llamado estático.
    - *paramExp*: *ArrayList<ExpNode>*. Lista de expresiones (argumentos).
    - *isConstructor*: *boolean*. Determina si es un llamado a un constructor.
- Métodos:
    - *addParamExp(exp: ExpNode) -> void*. Agrega una expresión a la lista ordenada de expresiones que utilizará el llamado al método.
    - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información relevante del nodo, en formato json.
    - *Getters* y *setters* de los atributos.

6.7.3.13 Clase VarNode

Esta clase representa un Nodo de un Acceso a una variable.. Los acceso que incluye son a: atributos self, atributos, variables.

- Hereda de: *AccessNode*.
- Atributos (todos heredados).
- Métodos:
    - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información relevante del nodo, en formato json.

6.7.3.14 Clase ArrayNode

Esta clase representa un Nodo de Arreglo. Contiene información sobre el índice utilizado para acceder o construir un *Array*.

- Hereda de: *VarNode*.
- Atributos:
    - *indexExp: ExpNode*. Expresión del índicel del arreglo

- *access*: *boolean*. Lista de expresiones (argumentos).

- Métodos:
  - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información relevante del nodo, en formato json.
  - *Getters* y *setters* de los atributos.

6.7.3.15 Clase LiteralNode

Clase que representa a un NodoLiteral: literal entero, literal cadena, literal caracter, literal nil, literal bool.

- Hereda de: *ExpNode*.
- Atributos (todos heredados).
- Métodos:
  - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información relevante del nodo, en formato json.

6.7.3.16 Clase WhileNode

Esta clase representa un Nodo del AST para la sentencia de control iterativa *while*. Contiene información sobre la expresión de condición (debe ser de tipo *Bool*) e información sobre la sentencia que se ejecutará iterativamente mientras se cumpla la condición establecida.

- Hereda de: *SentenceNode*.
- Atributos:
  - *condition: ExpNode*. Expresión (tipo *Bool*) que deberá cumplirse para entrar en el cuerpo del *while* (*body*).
  - *body*: *SentenceNode*. Cuerpo del *while*. Este se ejecutará iterativas veces mientras se cumpla la condición *condition*.
- Métodos:
  - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información relevante del nodo, en formato json.
  - *Getters* y *setters* de los atributos.

6.7.3.17 Clase IfElseNode

Esta clase representa un Nodo del AST para una sentencia de control if-else .Contiene información sobre la expresión de condición (debe ser de tipo Bool), información sobre la

sentencia que se ejecutará si se cumple la condición, y un atributo opcional con la sentencia que se ejecutará si no se cumple la condición.

- Hereda de: *SentenceNode.*
- Atributos:
  - *condition: ExpNode*. Expresión (tipo *Bool*) que deberá cumplirse para entrar en el cuerpo del *while* (*body*).
  - *thenPart*: *SentenceNode*. Sentencia “entonces”. Si se cumple la condición *condition* se entrará en esta nodo sentencia.
  - *elsePart*: *SentenceNode*. Sentencia “else” (opcional). Si no se cumple la condición *condition* se entrará en esta nodo sentencia.
- Métodos:
  - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información relevante del nodo, en formato json.
  - *Getters* y *setters* de los atributos.

6.7.3.18 Clase AssignNode

Esta clase representa un Nodo de Asignación de un valor a una variable. En el lado izquierdo (leftSide) hay un Nodo Variable, y en el lado derecho (rightSide) hay un Nodo Expresión.

- Hereda de: *SentenceNode.*
- Atributos:
  - *leftSide: VarNode*. Lado izquierdo de la asignación.
  - *rightSide*: *ExpNode*. Lado derecho de la asignación. El tipo de este deberá coincidir con el lado izquierdo.
- Métodos:
  - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información relevante del nodo, en formato json.
  - *Getters* y *setters* de los atributos.

6.7.3.19 Clase ReturnNode

Clase que representa a una sentencia return.

- Hereda de: *SentenceNode.*
- Atributos:
  - *returnVal: ExpNode*. Nodo expresión del retorno. Deberá coincidir con el

tipo de retorno del método asociado.

- Métodos:
  - *toJson(indents: int) -> void*: método implementado mediante la interfaz IJsonable. Retorna un string con la información relevante del nodo, en formato json.
  - *Getters* y *setters* de los atributos.
  -

7 Generación de código

En `tinyRust+` no existe una etapa de generación de código intermedio. Una vez finalizadas las tareas del *Frontend* (análisis), se procederá a la etapa de *Backend* del compilador, concretamente a la generación de código.

Esta es la última etapa de el compilador de `tinyRust+` y es la encargada de traducir las representaciones intermedias generadas en la etapa anterior, a un lenguaje de bajo nivel. En este caso el lenguaje elegido es el del set de instrucciones de MIPS32. Con MIPS32 simularemos un *Stack Machine* con 32 registros

7.1 Diseño

En un paquete aparte dentro del paquete *Backend* hay una interfaz llamada *VisitorCodeGen* y una clase llamada *CodeGenerator*. Se ha seguido el patrón de diseño *Visitor* donde la clase *CodeGenerator* implementa la interfaz *VisitorCodeGen*. Luego cada nodo del *AST* contiene un método llamado *codeGen*, que toma como argumento cualquier instancia que haya implementado *VisitorCodeGen* y ejecuta su método correspondiente. De esta forma se separan los objetos (nodos) pertenecientes a la representación intermedia (AST), del código que generará el código de `tinyRust+`, valga la redundancia.

Primero se explicarán las estructuras que utilizarán los datos (manipularan), tales como *virtual table*, *class instance record* y registros de activación.

7.1.1 Registros de activación

Una invocación de un método es una *activación* del método. El tiempo de vida de la activación conlleva ejecutar todas las instrucciones del método, incluyendo otros llamados a métodos.

Para llevar a cabo la ejecución de métodos es necesaria una estructura que almacenaremos en el *stack* de memoria. Esta estructura es el *registro de activación* y contiene toda la información necesaria para la ejecución de un método activado.

Desde direcciones de memoria superiores hacia direcciones de memoria inferiores, se presenta un detallado listado con la información almacenada del registro de activación:

1. Dirección de retorno ($ra) del método llamador. Esta información resulta útil en los momento de retorno o finalización del bloque del método, permitiendo regresar a la siguiente línea del punto de origen del llamado. En el método *main* el valor de esta es cero.
2. *Frame Pointer* ($fp) del método llamador. De gran utilidad al concluir (finalizar) el método actual. Brinda acceso a la localidad del registro de activación del método llamador. En el método *main* el valor de este es cero.
3. Referencia *self*. Esta referencia hace alusón a la instancia a la cual pertenece el método en cuestión, propocionando un vínculo directo con las misma. En un método estático o *main* el valor de este es cero.
4. Parámetros. Se trata de una lista de valores de los parámetros del método. El parámetro `i` se ubica en una dirección de memoria más baja que el parámetro `i-1`, estableciendo un orden descendente.
5. Variables locales. Aquí encontramos una lista de variables locales, declaradas dentro del método. De manera similar a los parámetros, la variable `i` se ubica en una dirección de memoria más baja que la variable `i-1`.
6. Extras. Son direcciones temporales utilizadas para almacenar información, como por ejemplo, al asignar variables. Una vez que se tiene la referencia de la variable a asignar (lado izquierdo) se almacena en la pila (*push*), específicamente se guardará en una dirección “extra”, luego cuando se tiene el valor o la referencia del lado derecho, desapilo (*pop*) la referencia izquierda, y procedo a realizar la asignación.

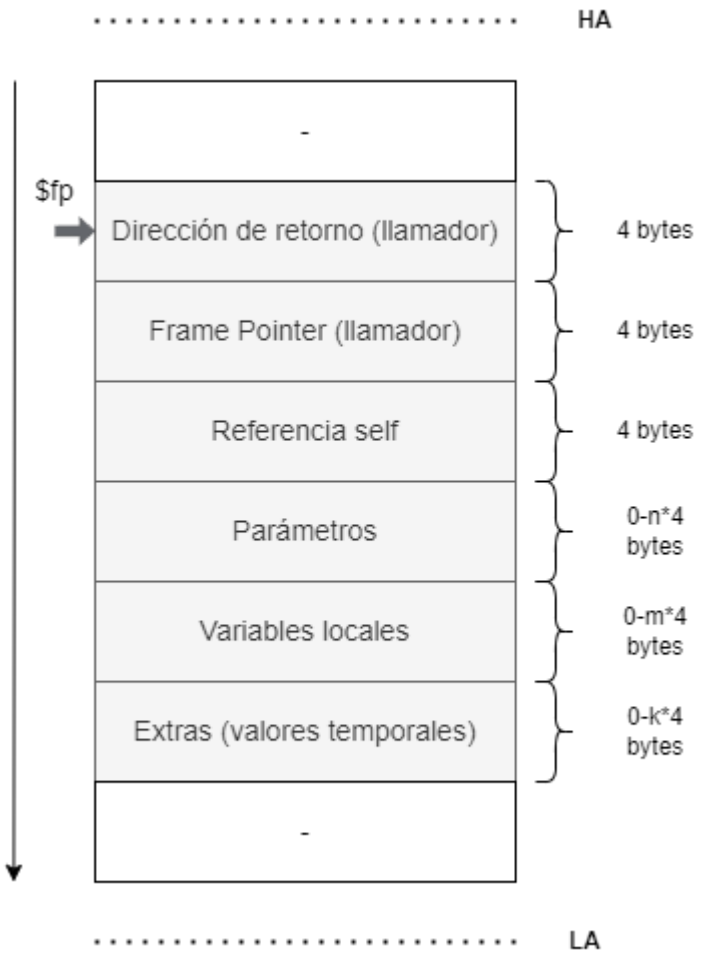


Figura 14: Registro de activación de un método

Como se puede observar en la figura 14, las únicas locaciones que siempre van a estar en un registro de activación son: dirección de retorno del llamador, el frame pointer del llamador, y la referencia self.

7.1.2 Convenciones al usar registros

En este apartado se describirán las convenciones empleadas sobre el conjunto de registros de MIPS32, a la hora de generar código.

El *frame pointer*, representado por el registro `$fp`, apunta a la primera dirección del registro de activación (dirección de retorno del llamador). Al momento de generar código, para todo cuerpo de un método se accede a todas estas direcciones, excepto a las “extras”, utilizando como referencia el *frame pointer*.

El *stack pointer* `$sp` es utilizado para apilar y desapilar los registros de activación, y para

alocar direcciones extra (valores temporales). Se apila información, en el registro de activación, al momento de realizar un llamado a un método. Se desapila el registro de activación una vez finaliza el método (*return* o fin de bloque de método).

Los registros `$a0`, `$t0` y `$t1` son utilizados como acumuladores (principalmente `$a0`). El registro `$s0` se usa convenientemente para almacenar la dirección *self* y que persista durante el ciclo de vida del registro de activación.

Al ejecutar funciones que retornan un valor de tipo (distinto de *void*), resultó conveniente almacenar el valor de retorno en el registro `$v0`. Esto se tiene en cuenta al momento de las asignaciones, si el lado derecho es una expresión que realiza una llamada a una función que retorna un valor, y ese valor no se utiliza para computar una operación de tipo aritmética, booleana o de comparación, entonces se asigna la información que hay en el registro `$v0`. En caso contrario, se asigna el valor/dirección que contiene `$a0`.

7.1.3 Class Instance Record y Virtual Tables

Una tabla virtual (o virtual table) de una clase, es una referencia a una porción de memoria estática `(.data)` que contiene cero o más palabras (*words*) que referencian a un label de un método de la clase en cuestión.

`tinyRust+` genera una tabla virtual para cada clase declarada, la cual contiene todos los métodos definidos en dicha clase, excepto el constructor. Al constructor se accede de forma directa usando el label con el siguiente formato: `[Nombre de la clase]_[Nombre de la clase]`.

Para instanciar objetos es necesario utilizar una estructura dinámica a la cual sea fácil acceder en tiempo de ejecución. Para eso se realizan llamadas al sistema para realizar una alocación en el *heap* de la memoria, del tamaño que necesite el objeto. La llamada al sistema retorna una dirección del segmento de memoria donde comienza el *class instance record* (CIR), y al espacio reservado se accede desde la dirección retornada hacia valores superiores (tantos como el espacio solicitado).

En la dirección inicial del CIR (la retornada) se almacena la dirección de la tabla virtual asociada a la clase correspondiente. Para consultar y modificar los atributos accedemos al CIR (teniéndolo en el registro $t1), y si el atributo a modificar tiene posición `i`, la referencia a este se hace con un *offset* positivo de `4+4*i($t1)`.

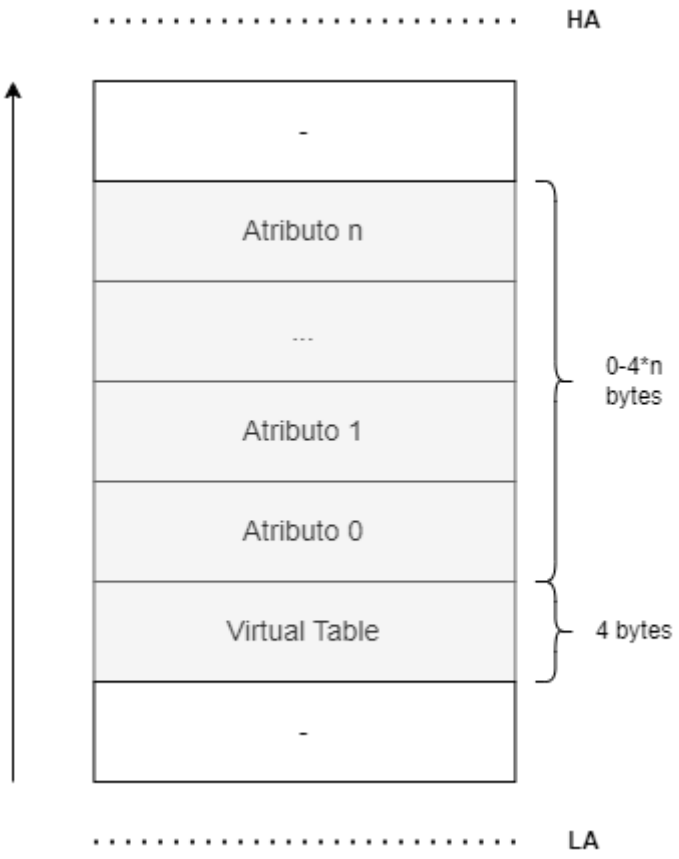


Figura 15: Estructura del Class Instance Record

7.1.4 Diseño de clases

Las clases utilizadas para esta etapa se encuentran empaquetadas en *Backend.* Se encuentra las siguientes clases (e interfaz).

7.1.4.1 Interfaz VisitorCodeGen

Esta interfaz contiene todos los métodos sobrecargados que generarán código para MIPS32 según el Nodo del AST que contega en el argumento. Utilizada para utilizar el patrón de diseño Visitante.

- Métodos:

  Cada uno de estos métodos generá una porción de código distinta según el nodo del AST que reciban por argumento.

  - *visit(accessNode: AccessNode) -> void.* Generador de código para todo el código `tinyRust+`.
  - *visit(arrayNode: ArrayNode) -> void.* Generador de código para construcciones de *array*s y accesos indexados.

- *visit(assignNode: AssignNode) -> void.* Generador de código para las asignaciones `<var> = <exp>`.
- *visit(ast: AST) -> void.*
- *visit(binExpNode: BinExpNode) -> void.* Generador de código para expresiones binarias.
- *visit(blockNode: BlockNode) -> void.* Generador de código para todas las sentencias de un bloque.
- *visit(callNode: CallNode) -> void.* Generador de código para llamado a métodos (encadenados o no). Incluye constructores y métodos estáticos.
- *visit(classNode: ClassNode) -> void.* Generador de código para el cuerpo de una clase.
- *visit(ifElseNode: IfElseNode) -> void.* Generador de código para las estructuras condicionales.
- *visit(literalNode: LiteralNode) -> void.* Generador de código para literales.
- *visit(methodNode: MethodNode) -> void.* Generador de código para el cuerpo de un método.
- *visit(returnNode: ReturnNode) -> void.* Generador de código para sentencias de retorno.
- *visit(unExpNode: UnExpNode) -> void.* Generador de código para expresiones binarias.
- *(varNode: VarNode) -> void.* Generador de código para acceso a variables (encadenadas o no) de cualquier tipo. Incluye lógica para acceder al valor o a la referencia, y también resuelve referencias self.
- *visit(whileNode: WhileNode) -> void.* Generador de código para las sentencias *while*.

7.1.4.2 Clase CodeGenerator

Esta clase es la que efectivamente implementará los métodos de la interfaz *VisitorCodeGen*. Además contiene atributos y métodos importantes para el proceso de generar código.

- Implementa: *VisitorCodeGen*
- Atributos:
  - *data: StringBuilder*. Porción de código que se incluirá en `.data`.
  - *vtables: StringBuilder*. Porción de código que incluirá las *vtables* dentro de `.data`.
  - *text: StringBuilder*. Porción de código con las instrucciones traducidas que estarán contenidas en `.text`.

- *branchAmount: int*. Numerador de las etiquetas generadas en instrucciones de salto (branch).
- *symbolTable: SymbolTable*. Referencia a la tabla de símbolos generada.
- *currentClass: ClassEntry*. Referencia a la clase actual (scope).
- *currentMethod: Method*. Referencia al método actual (scope).
- *chainedTo: ClassEntry*. Referencia a la entrada de la clase encadenada al nodo actual.
- *f_getVal: boolean*. Flag útil para la generación de código de los nodos *VarNode*. Indica si hay que obtener la referencia a una variable o el valor de esta.
- *f_self: boolean*. Flag útil para la generación de código de los nodos *VarNode*. Indica si la referencia a una variable hay que localizarla en los atributos de la clase actual o en el scope del método.
- *f_main boolean*. Flag útil para reconocer si se está posicionado en el *scope* del método *main*.

- Métodos:
  Implementados por la interfaz *VisitorCodeGen*.
  - *void predCodeGen()*. Este método genera el código de las clases predefinidas de `tinyRust+` (IO, Str, Array).
  - *String getCode()*. Obtiene el código traducido a `asm` a través de los atributos *data*, *vtables* y *text*. Retorna un *String* con este código.
  -

7.1.4.3 Visitor en AST

Ya está definido el código para traducir las representaciones intermedias, como es el AST. La superclase asbtracta principal del paquete *AST* es *Node*, para finalizar el patrón *Visitor* se agregó el método público *codeGen()* a *Node* de la siguiente forma:

```
public void codeGen(VisitorCodeGen visitor) {
        visitor.visit(this);
}
```

De esta forma cada nodo del AST se vincula con el generador de código *CodeGenerator* el cual implementa *VisitorCodeGen* y puede genera su código correspondiente

7.1.4.4 Diagrama general del compilador tinyRust+

Con el propósito de ofrecer una visión integral y concisa del diseño completo, se presenta

el siguiente diagrama UML del compilador, que representa los componentes clave de cada paquete.

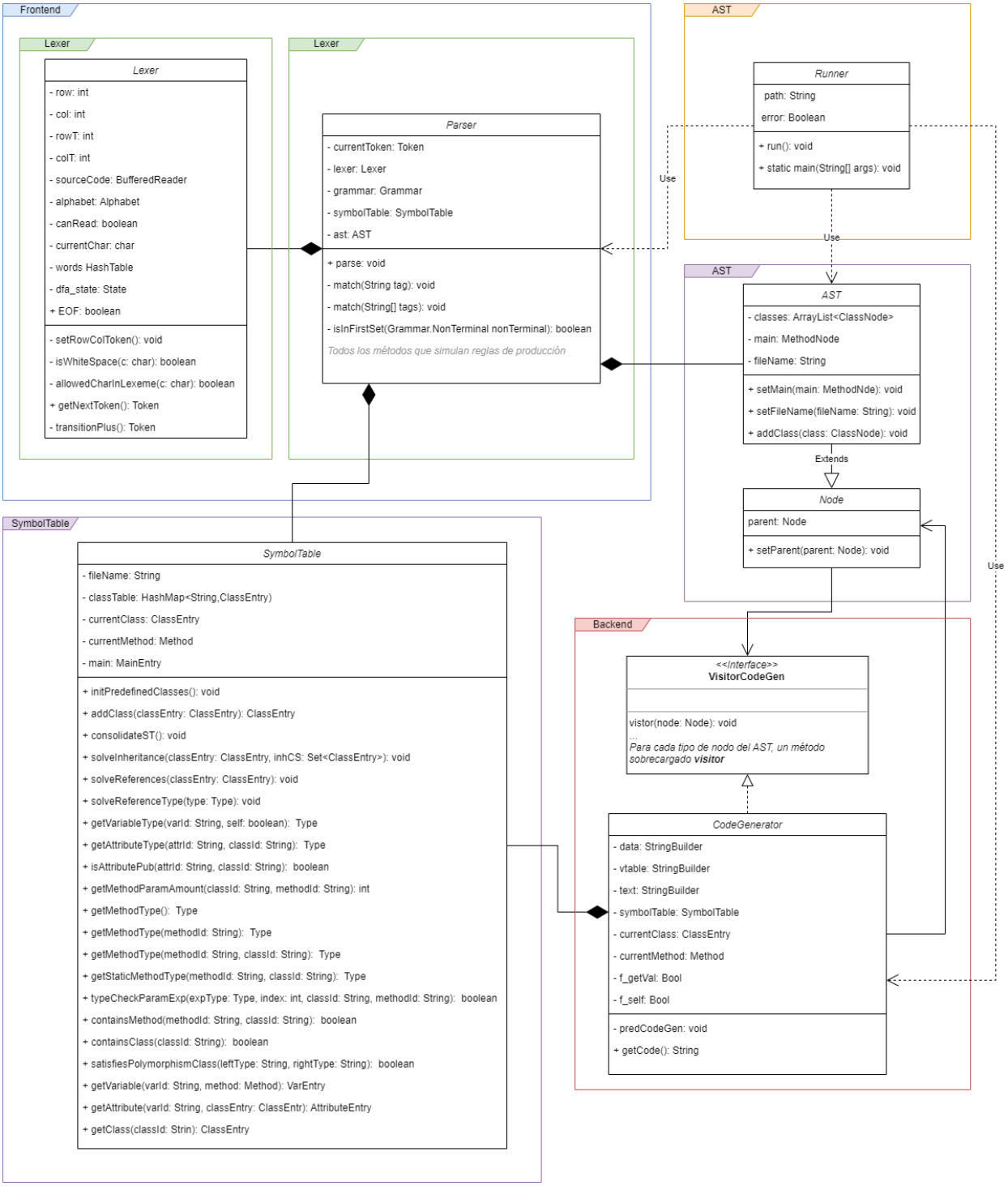


Figura 16: Diagrama UML de la generación de código

7.1.4.5 Clase Runner

Finalmente, como punto de partida del compilador, haremos uso de la clase *Runner* para iniciar su ejecución. La clase *Runner* desempeña un papel fundamental al llevar a cabo el análisis

completo y la generación correspondiente de código. Es a través de esta clase que se coordina y controla el flujo de trabajo del compilador, permitiendo que todos los procesos necesarios se lleven a cabo de manera ordenada y eficiente.

- Atributos:
  - path: String. Dirección del archivo a escanear y compilar.
  - error: boolean. Atributo que determina si han ocurrido errores en tiempo de compilación.
- Métodos:
  - *run() -> void*. Lee el archivo fuente, e instancia al *Parser*. Luego pone en marcha el análisis sintáctico. Una vez finalizado el análisis, se extrae la tabla de símbolos y el *AST*, y se procede a generar el código para MIPS32 mediante la tabla de símbolos y el árbolo sintáctico abstracto.
  - *main(args: String[]) -> void*. Punto de entrada del compilador. Obtiene la dirección del archivo `tinyRust+` y pone “en marcha” el compilador.

8 Ejemplo de compilación

A continuación, se presentará un ejemplo de traducción de código en el lenguaje `tinyRust+` al lenguaje ensamblador MIPS32. A través de este ejemplo, podremos observar el proceso de traducción y comprender cómo se transforma el código fuente en `tinyRust+` en instrucciones específicas del conjunto de instrucciones MIPS32. Este ejemplo nos permitirá apreciar la correspondencia entre ambos lenguajes y comprender cómo se lleva a cabo la traducción para obtener un programa ejecutable en el entorno MIPS32.

8.1 Código fuente Fibonacci

```
class Fibonacci {
 pub I32: suma;
 I32: i,j;
 fn sucesion_fib(I32: n)-> void{
     I32: idx;
     i=0; j=1; suma=0; idx = 0;
     while (idx <= n){
          (out_idx(idx));
          (out_val(i));
          suma = i + j;
          i = j;
          j = suma;
          idx = idx + 1;
     }
 }
 create(){
     i=0;
     j=0;
     suma=0;
 }
 fn out_idx(I32: num) -> void{
     (IO.out_str("f_"));
     (IO.out_i32(num));
     (IO.out_str("="));
 }
```

```
    fn out_val(I32: s) -> void{
        (IO.out_i32(s));
        (IO.out_str("\n"));
    }
}
fn main(){
    Fibonacci: fib;
    I32: n;
    fib = new Fibonacci();
    n = IO.in_i32();
    (fib.sucesion_fib(n));
}
```

`tinyRust+` compila archivos con la extensión `.tr`. Para seguir este ejemplo de prueba, u otro, seguir las siguientes instrucciones, sino saltear a la sección 8.2 para ver el código MIPS32 generado.

1. Abrir un intérprete de comandos.
2. Entrar en directorio donde se encuentre el ejecutable **tinyRust+.jar.**
3. Obtener la dirección del archivo a compilar: /path/to/source/**code.tr**.
4. Ejecutar el siguiente comando:
   `java -jar tinyRust+.jar /path/to/source/code.tr`
5. En el mismo directorio se crea un archivo `.asm` con el código traducido a el del conjunto de instrucciones de MIPS32.
6. Puede ser ejecutado en un simulador de MIPS32 como puede ser MARS o QtSpim.

8.2 Código MIPS32 Fibonacci

Luego de compilar el código fuente, se habrá creado un archivo `.asm` con el código MIPS32 generado como se muestra a continuación.

```
        .data
true: .asciiz "true"
false: .asciiz "false"
str_f_: .asciiz "f_"
str_61: .asciiz "="
str_92n: .asciiz "\n"
```

```
# vtables_section
        .data
Fibonacci_vtable:
        .word Fibonacci_sucesion_fib
        .word Fibonacci_out_idx
        .word Fibonacci_out_val
        .text
        .globl main
         j main


IO_out_str:
        sw $ra, 0($fp)
        lw $a0, -12($fp)
        li $v0, 4
        syscall
        addiu $sp, $sp, 12
        lw $fp, 0($sp)
        addiu $sp, $sp, 4
        jr $ra
IO_out_i32:
        sw $ra, 0($fp)
        lw $a0, -12($fp)
        li $v0, 1
        syscall
        addiu $sp, $sp, 12
        lw $fp, 0($sp)
        addiu $sp, $sp, 4
        jr $ra
IO_out_bool:
        sw $ra, 0($fp)
        lw $a0, -12($fp)
        li $t0, 1
        beq $a0, $t0, IO_out_bool_true_case
        la $a0, false
        j IO_out_bool_end
IO_out_bool_true_case:
        la $a0, true
IO_out_bool_end:
```

```
        li $v0, 4
        syscall
        addiu $sp, $sp, 12
        lw $fp, 0($sp)
        addiu $sp, $sp, 4
        jr $ra
IO_out_char:
        sw $ra, 0($fp)
        lw $a0, -12($fp)
        li $v0, 11
        syscall
        addiu $sp, $sp, 12
        lw $fp, 0($sp)
        addiu $sp, $sp, 4
        jr $ra
IO_in_i32:
        sw $ra, 0($fp)
        li $v0, 5
        syscall
        addiu $sp, $sp, 8
        lw $fp, 0($sp)
        addiu $sp, $sp, 4
        jr $ra
IO_in_bool:
        sw $ra, 0($fp)
        li $v0, 5
        syscall
        addiu $sp, $sp, 8
        lw $fp, 0($sp)
        addiu $sp, $sp, 4
        jr $ra
Fibonacci_Fibonacci:
        sw $ra, 0($fp)
        lw $s0, -8($fp)
        la $a0, Fibonacci_vtable
        sw $a0, 0($s0)
        addiu $sp, $sp, 0
        move $a0, $fp
```

```
        addiu $a0, $a0, -8
        lw $s0, ($a0)
        addiu $a0, $s0, 8
        sw $a0, 0($sp)
        addiu $sp, $sp, -4
        li $a0, 0
        lw $t1, 4($sp)
        addiu $sp, $sp, 4
        sw $a0, ($t1)
        move $a0, $fp
        addiu $a0, $a0, -8
        lw $s0, ($a0)
        addiu $a0, $s0, 12
        sw $a0, 0($sp)
        addiu $sp, $sp, -4
        li $a0, 0
        lw $t1, 4($sp)
        addiu $sp, $sp, 4
        sw $a0, ($t1)
        move $a0, $fp
        addiu $a0, $a0, -8
        lw $s0, ($a0)
        addiu $a0, $s0, 4
        sw $a0, 0($sp)
        addiu $sp, $sp, -4
        li $a0, 0
        lw $t1, 4($sp)
        addiu $sp, $sp, 4
        sw $a0, ($t1)
        lw $a0, -8($fp) #
        addiu $sp, $sp, 8
        lw $fp, 0($sp)
        addiu $sp, $sp, 4
        lw $ra, 0($sp)
        jr $ra
Fibonacci_sucesion_fib:
        sw $ra, 0($fp)
        lw $s0, -8($fp)
```

```
addiu $sp, $sp, -4
move $a0, $fp
addiu $a0, $a0, -8
lw $s0, ($a0)
addiu $a0, $s0, 8
sw $a0, 0($sp)
addiu $sp, $sp, -4
li $a0, 0
lw $t1, 4($sp)
addiu $sp, $sp, 4
sw $a0, ($t1)
move $a0, $fp
addiu $a0, $a0, -8
lw $s0, ($a0)
addiu $a0, $s0, 12
sw $a0, 0($sp)
addiu $sp, $sp, -4
li $a0, 1
lw $t1, 4($sp)
addiu $sp, $sp, 4
sw $a0, ($t1)
move $a0, $fp
addiu $a0, $a0, -8
lw $s0, ($a0)
addiu $a0, $s0, 4
sw $a0, 0($sp)
addiu $sp, $sp, -4
li $a0, 0
lw $t1, 4($sp)
addiu $sp, $sp, 4
sw $a0, ($t1)
move $a0, $fp
addiu $a0, $a0, -16
sw $a0, 0($sp)
addiu $sp, $sp, -4
li $a0, 0
lw $t1, 4($sp)
addiu $sp, $sp, 4
```

```
        sw $a0, ($t1)
while_0:
        move $a0, $fp
        addiu $a0, $a0, -16
        lw $a0, ($a0)
        sw $a0, ($sp)
        addiu $sp, $sp, -4
        move $a0, $fp
        addiu $a0, $a0, -12
        lw $a0, ($a0)
        lw $t0, 4($sp)
        addiu $sp, $sp, 4
        sle $a0, $t0, $a0
        beq $a0, $zero, end_while_0
        la $t0, Fibonacci_vtable
        lw $t1, 4($t0)
        addiu $sp, $sp, -4
        sw $fp, 0($sp)
        addiu $sp, $sp, -4
        lw $t0, -8($fp)
        sw $t0, 0($sp)
        addiu $sp, $sp, -4
        move $a0, $fp
        addiu $a0, $a0, -16
        lw $a0, ($a0)
        sw $a0, 0($sp)
        addiu $sp, $sp, -4
        addiu $fp, $sp, 16
        jalr $t1
        la $t0, Fibonacci_vtable
        lw $t1, 8($t0)
        addiu $sp, $sp, -4
        sw $fp, 0($sp)
        addiu $sp, $sp, -4
        lw $t0, -8($fp)
        sw $t0, 0($sp)
        addiu $sp, $sp, -4
        move $a0, $fp
```

```
addiu $a0, $a0, -8
lw $s0, ($a0)
addiu $a0, $s0, 8
lw $a0, ($a0)
sw $a0, 0($sp)
addiu $sp, $sp, -4
addiu $fp, $sp, 16
jalr $t1
move $a0, $fp
addiu $a0, $a0, -8
lw $s0, ($a0)
addiu $a0, $s0, 4
sw $a0, 0($sp)
addiu $sp, $sp, -4
move $a0, $fp
addiu $a0, $a0, -8
lw $s0, ($a0)
addiu $a0, $s0, 8
lw $a0, ($a0)
sw $a0, ($sp)
addiu $sp, $sp, -4
move $a0, $fp
addiu $a0, $a0, -8
lw $s0, ($a0)
addiu $a0, $s0, 12
lw $a0, ($a0)
lw $t0, 4($sp)
addiu $sp, $sp, 4
add $a0, $t0, $a0
lw $t1, 4($sp)
addiu $sp, $sp, 4
sw $a0, ($t1)
move $a0, $fp
addiu $a0, $a0, -8
lw $s0, ($a0)
addiu $a0, $s0, 8
sw $a0, 0($sp)
addiu $sp, $sp, -4
```

```
move $a0, $fp
addiu $a0, $a0, -8
lw $s0, ($a0)
addiu $a0, $s0, 12
lw $a0, ($a0)
lw $t1, 4($sp)
addiu $sp, $sp, 4
sw $a0, ($t1)
move $a0, $fp
addiu $a0, $a0, -8
lw $s0, ($a0)
addiu $a0, $s0, 12
sw $a0, 0($sp)
addiu $sp, $sp, -4
move $a0, $fp
addiu $a0, $a0, -8
lw $s0, ($a0)
addiu $a0, $s0, 4
lw $a0, ($a0)
lw $t1, 4($sp)
addiu $sp, $sp, 4
sw $a0, ($t1)
move $a0, $fp
addiu $a0, $a0, -16
sw $a0, 0($sp)
addiu $sp, $sp, -4
move $a0, $fp
addiu $a0, $a0, -16
lw $a0, ($a0)
sw $a0, ($sp)
addiu $sp, $sp, -4
li $a0, 1
lw $t0, 4($sp)
addiu $sp, $sp, 4
add $a0, $t0, $a0
lw $t1, 4($sp)
addiu $sp, $sp, 4
sw $a0, ($t1)
```

```
        j while_0
end_while_0:
        addiu $sp, $sp, 16
        lw $fp, 0($sp)
        addiu $sp, $sp, 4
        lw $ra, 0($sp)
        jr $ra
Fibonacci_out_idx:
        sw $ra, 0($fp)
        lw $s0, -8($fp)
        addiu $sp, $sp, 0
        addiu $sp, $sp, -4
        sw $fp, 0($sp)
        addiu $sp, $sp, -4
        lw $t0, -8($fp)
        li $t0, 0
        sw $t0, 0($sp)
        addiu $sp, $sp, -4
        la $a0, str_f_
        sw $a0, 0($sp)
        addiu $sp, $sp, -4
        addiu $fp, $sp, 16
        jal IO_out_str
        addiu $sp, $sp, -4
        sw $fp, 0($sp)
        addiu $sp, $sp, -4
        lw $t0, -8($fp)
        li $t0, 0
        sw $t0, 0($sp)
        addiu $sp, $sp, -4
        move $a0, $fp
        addiu $a0, $a0, -12
        lw $a0, ($a0)
        sw $a0, 0($sp)
        addiu $sp, $sp, -4
        addiu $fp, $sp, 16
        jal IO_out_i32
        addiu $sp, $sp, -4
```

```
        sw $fp, 0($sp)
        addiu $sp, $sp, -4
        lw $t0, -8($fp)
        li $t0, 0
        sw $t0, 0($sp)
        addiu $sp, $sp, -4
        la $a0, str_61
        sw $a0, 0($sp)
        addiu $sp, $sp, -4
        addiu $fp, $sp, 16
        jal IO_out_str
        addiu $sp, $sp, 12
        lw $fp, 0($sp)
        addiu $sp, $sp, 4
        lw $ra, 0($sp)
        jr $ra
Fibonacci_out_val:
        sw $ra, 0($fp)
        lw $s0, -8($fp)
        addiu $sp, $sp, 0
        addiu $sp, $sp, -4
        sw $fp, 0($sp)
        addiu $sp, $sp, -4
        lw $t0, -8($fp)
        li $t0, 0
        sw $t0, 0($sp)
        addiu $sp, $sp, -4
        move $a0, $fp
        addiu $a0, $a0, -12
        lw $a0, ($a0)
        sw $a0, 0($sp)
        addiu $sp, $sp, -4
        addiu $fp, $sp, 16
        jal IO_out_i32
        addiu $sp, $sp, -4
        sw $fp, 0($sp)
        addiu $sp, $sp, -4
        lw $t0, -8($fp)
```

```
        li $t0, 0
        sw $t0, 0($sp)
        addiu $sp, $sp, -4
        la $a0, str_92n
        sw $a0, 0($sp)
        addiu $sp, $sp, -4
        addiu $fp, $sp, 16
        jal IO_out_str
        addiu $sp, $sp, 12
        lw $fp, 0($sp)
        addiu $sp, $sp, 4
        lw $ra, 0($sp)
        jr $ra
main:
        move $fp, $sp
        addiu $sp, $sp, -12
        addiu $sp, $sp, -8
        li $a0, 16
        li $v0, 9
        syscall
        sw $v0, 8($sp)
        move $a0, $fp
        addiu $a0, $a0, -12
        sw $a0, 0($sp)
        addiu $sp, $sp, -4
        addiu $sp, $sp, -4
        sw $fp, 0($sp)
        addiu $sp, $sp, -4
        lw $t0, -12($fp)
        sw $t0, 0($sp)
        addiu $sp, $sp, -4
        addiu $fp, $sp, 12
        jal Fibonacci_Fibonacci
        lw $t1, 4($sp)
        addiu $sp, $sp, 4
        sw $v0, ($t1)
        move $a0, $fp
        addiu $a0, $a0, -16
```

```
        sw $a0, 0($sp)
        addiu $sp, $sp, -4
        addiu $sp, $sp, -4
        sw $fp, 0($sp)
        addiu $sp, $sp, -4
        lw $t0, -12($fp)
        li $t0, 0
        sw $t0, 0($sp)
        addiu $sp, $sp, -4
        addiu $fp, $sp, 12
        jal IO_in_i32
        lw $t1, 4($sp)
        addiu $sp, $sp, 4
        sw $v0, ($t1)
        la $t0, Fibonacci_vtable
        lw $t1, 0($t0)
        addiu $sp, $sp, -4
        sw $fp, 0($sp)
        addiu $sp, $sp, -4
        lw $t0, -12($fp)
        sw $t0, 0($sp)
        addiu $sp, $sp, -4
        move $a0, $fp
        addiu $a0, $a0, -16
        lw $a0, ($a0)
        sw $a0, 0($sp)
        addiu $sp, $sp, -4
        addiu $fp, $sp, 16
        jalr $t1
        addiu $sp, $sp, 16
        addiu $sp, $sp, 4
        li $v0, 10
        syscall
}
```

El generador de código agrega líneas de comentarios para facilitar la lectura del código MIPS, tales como *“# ASIGNACION*”, *“# LADO IZQUIERDO*”, *“# LLAMADO*”, etc.

Referencias

1. Aho, A. V. (2013). Compilers: Principles, Techniques, and Tools 2nd By Alfred V. Aho (International Economy Edition) (2nd ed.). Pearson.

2. Fischer, C. N., Cytron, R. K., & LeBlanc, R. J. (2009). Crafting a compiler. Addison-Wesley.