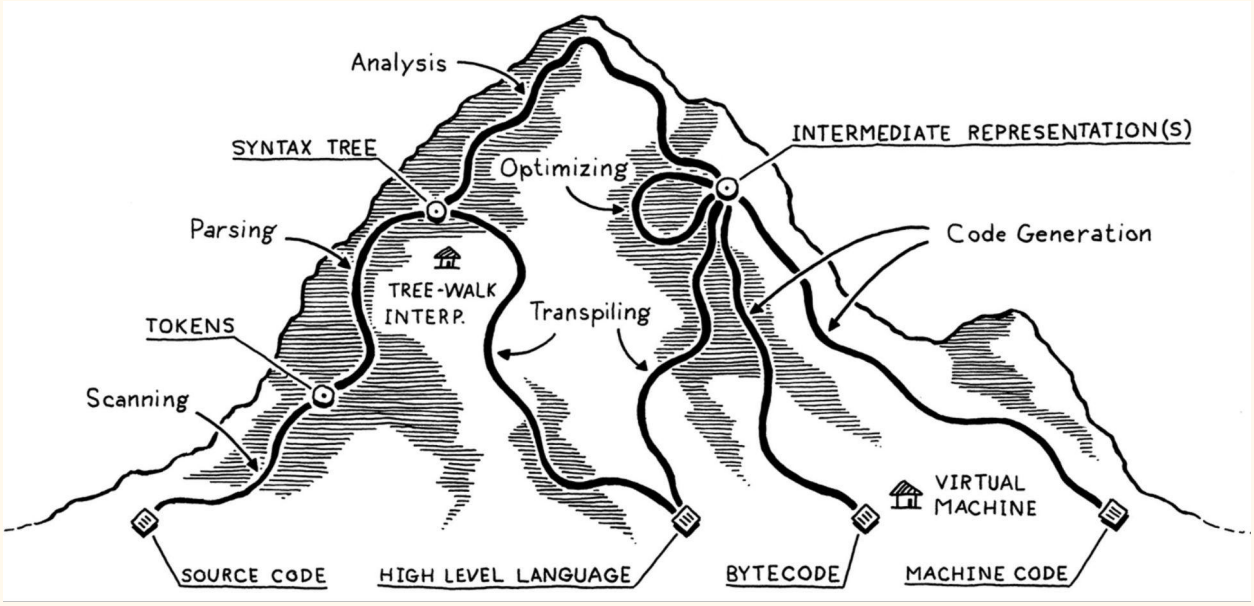
ORGANIZACIÓN DE LENGUAJES Y COMPILADORES 1 (PRIMER SEMESTRE 2020)
Centro Universitario de Occidente

MANUAL TÉCNICO:
IDE LENGUAJE CRL

Fernando Jose Rodriguez Ramirez 202030542



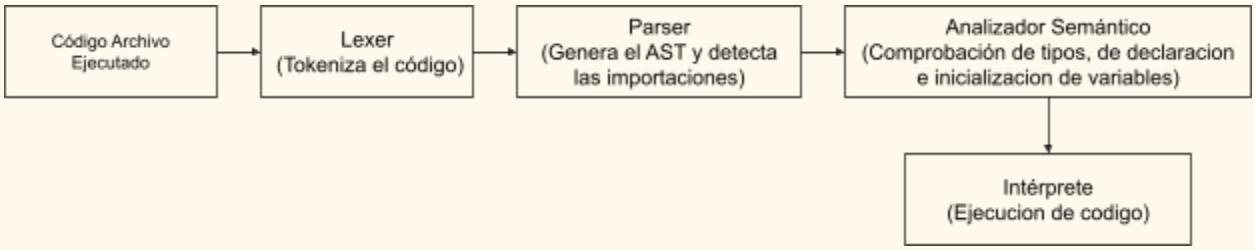
INTRODUCCIÓN

En este manual se explicará el funcionamiento del IDE e Intérprete para el lenguaje CRL (Compi Report Language), así como las técnicas que se usaron para los diferentes analisis e interpretación del lenguaje.

Estructura del Intérprete

El intérprete está compuesto por cuatro fases que analizan un conjunto de archivos con la extensión .crl que indican que el archivo pertenece al lenguaje CRL

Esta estructura es mostrada con más detalle en el siguiente diagrama:



Lexer:

El lexer está escrito al igual que el Parser en Jison, se encarga de obtener los caracteres de la pestaña abierta en el editor una vez se pida que se ejecute el programa y lo pasa por el lexer para obtener sus diferentes partes. Para eso usa las siguientes expresiones regulares:

SALTO = \n|\r digit = [0-9] letter = [a_zA_Z]

identifier = ([a-zA-Z])[a-zA-Z0-9_]* comentario = "!!".*

whitespace = [\n\t] indentacion = \t | " " integer = [1-9]{digit}*|"0"

double = "-"? ({digit}+ "." {digit}+ | {digit}+ ".")

"'''" -> se inicia estado ‘comentario_multilinea’
<comentario_multilinea>””””

<comentario_multilinea><<EOF>>

<comentario_multilinea>\’

<comentario_multilinea>[^’]*

{comentario}

{identifier}".crl" -> 'NOMBRE_ARCHIVO'

{identifier}".".* -> ERROR -> 'NOMBRE_ARCHIVO’

// Palabras Reservadas

"Importar" 'R_IMPORT' "Incerteza" 'R_INCERTEZA'

"Mostrar" 'R_MOSTRAR' "DibujarAST" 'R_D_AST'

"DibujarEXP" 'R_D_EXP' "DibujarTS" 'R_D_TS'

"Retorno" 'R_RETORNO' "Detener" 'R_DETENER'

"Continuar" 'R_CONTINUAR' "Double"

"Para" ‘R_PARA' "Si" 'R_SI'

"Sino" 'R_SINO' "Mientras"

//Palabras reservadas para los tipos de datos

"Double" "Boolean" "String"

"Int" "Char" "Void"

//Simbolos

"," 'COMA' "." 'PUNTO ":" 'DOS_PUNTOS'

";" 'PUNTO_COMA'

"{" 'LLAVE_IZQ' "}" 'LLAVE_DER'

"(" ")" "&&"

"||" "+" "-"

"*" "/" "%"

"^"

"<=" ">=" "=="

"!=" "|&" "<"

">" "="

"~" "!" "true"|"false"

{indentacion}+{comentario}?{SALTO}

{SALTO}

{indentacion}

{double}

{identifier}

{whitespace}

\' -> char_estado

<char_estado> [^\\'\n]

<char_estado> \\n

<char_estado>\\t

<char_estado>\\r

<char_estado> \n

<char_estado> .

<char_estado><<EOF>>

\"

<string_estado>[^\\"\n]+

<string_estado>\\n

<string_estado>\\r

<string_estado>\\t

<string_estado>\"

<string_estado>\n

<string_estado><<EOF>>

.

<<EOF>>

Parser:

El parser la fase encargada de estructurar el programa en un AST para su posterior interpretación, también se encarga de obtener todos los archivos importados desde el archivo que se parsea para poder importarlo posteriormente.

Para la estructuración del programa se tienen la siguiente gramática libre de contexto:

ini

: encabezado instrucciones fin

| encabezado fin

| instrucciones fin

| fin

| SALTO ini

fin

: EOF

| DEDENT_EOF

encabezado

: importaciones incerteza

| importaciones

| incerteza

```
importaciones
    : importaciones importacion
  | importacion
  | importaciones SALTO
importacion : R_IMPORT NOMBRE_ARCHIVO SALTO
incerteza : incerteza SALTO
| incertezad
incertezad : R_INCERTEZA expresion_logica SALTO
instrucciones
    : instrucciones instruccion
    | instruccion
    | instrucciones SALTO


instruccion
    : declaracion_variable SALTO
    | declaracion_funcion
    | asignacion SALTO
    | indentaciones instruccion_funcion          {
    | error SALTO
declaracion_variable
    : tipo_dato lista_variables
    | tipo_dato lista_variables IGUAL expresion_logica
```

```
asignacion : VARIABLE_IDENTIFICADOR IGUAL expresion_logica

tipo_dato
    : R_INT
    | R_DOUBLE
    | R_STRING
    | R_CHAR
    | R_BOOLEAN
declaracion_funcion
    : R_VOID declaracion_funciond SALTO
    | tipo_dato declaracion_funciond SALTO



declaracion_funciond
    : VARIABLE_IDENTIFICADOR PAR_IZQ PAR_DER DOS_PUNTOS
    | VARIABLE_IDENTIFICADOR PAR_IZQ declaracion_parametros PAR_DER DOS_PUNTOS
instrucciones_funcion
    : instrucciones_funcion indentaciones instruccion_funcion
    | instrucciones_funcion indentaciones SALTO
    | indentaciones instruccion_funcion
    | indentaciones SALTO
instruccion_funcion :
```

declaracion_variable SALTO

| asignacion SALTO

| llamada_funcion SALTO

| R_MOSTRAR PAR_IZQ parametros_mostrar PAR_DER SALTO

| R_D_AST PAR_IZQ VARIABLE_IDENTIFICADOR PAR_DER SALTO

|R_D_EXP PAR_IZQ expresion_logica PAR_DER SALTO

| R_D_TS PAR_IZQ PAR_DER SALTO

| R_RETORNO SALTO

| R_RETORNO expresion_logica SALTO

| R_DETENER SALTO

| R_CONTINUAR SALTO

| R_MIENTRAS PAR_IZQ expresion_logica PAR_DER DOS_PUNTOS SALTO

| R_PARA PAR_IZQ R_INT VARIABLE_IDENTIFICADOR IGUAL expresion_logica PUNTO_COMA expresion_logica PUNTO_COMA direccion_para PAR_DER DOS_PUNTOS SALTO

| R_SI PAR_IZQ expresion_logica PAR_DER DOS_PUNTOS SALTO

| R_SINO DOS_PUNTOS SALTO

parametros_mostrar

: parametros_mostrar COMA expresion_logica

| STRING

indentaciones

: indentaciones INDENTACION

| INDENTACION

direccion_para

```
    : SUMA SUMA
    | RESTA RESTA
declaracion_parametros
    : declaracion_parametros COMA declaracion_parametro
    | declaracion_parametro
declaracion_parametro
    : tipo_dato VARIABLE_IDENTIFICADOR
lista_variables
    : lista_variables COMA VARIABLE_IDENTIFICADOR
    | VARIABLE_IDENTIFICADOR
llamada_funcion
    : VARIABLE_IDENTIFICADOR PAR_IZQ PAR_DER
    | VARIABLE_IDENTIFICADOR PAR_IZQ parametros PAR_DER

parametros
    : parametros COMA expresion_logica
    | expresion_logica
expresion_logica
    : expresion_logica SUMA expresion_logica
    | expresion_logica RESTA expresion_logica
    | expresion_logica MULTIPLICACION expresion_logica
    | expresion_logica DIVISION expresion_logica
    | expresion_logica MODULO expresion_logica
```

| expresion_logica POTENCIA expresion_logica

| RESTA expresion_logica

| expresion_logica MAYOR_QUE expresion_logica

| expresion_logica MENOR_QUE expresion_logica

| expresion_logica MAYOR_IGUAL_QUE expresion_logica

| expresion_logica MENOR_IGUAL_QUE expresion_logica

| expresion_logica IGUALDAD expresion_logica

| expresion_logica DIFERENCIA expresion_logica

| expresion_logica INCERTEZA expresion_logica

| expresion_logica AND expresion_logica

| expresion_logica XOR expresion_logica

| expresion_logica OR expresion_logica

| NOT

| PAR_IZQ expresion_logica PAR_DER

| DOUBLE

| VARIABLE_IDENTIFICADOR

| STRING

| CHAR

| llamada_funcion

Es necesario tomar en cuenta que la es imposible parsear un programa en el lenguaje CRL solamente con una gramática libre de contexto esto debido a que la gramática no es de tipo 2 no es libre de contexto, es en su lugar una gramática tipo 1, una gramática sensible al contexto,

debido a que es necesario mantener el control de todas las indentaciones de instrucciones anteriores para saber los scopes de cada instrucción.

Para manejar esto el parser cuenta con un stack que mantiene registro de las indentaciones de las instrucciones que son parseadas, teniendo en cuenta tres casos diferentes:

INDENTACIONACTUAL > INDENTACIONANTERIOR

INDENTACIONACTUAL < INDENTACIONANTERIOR

INDENTACIONACTUAL = INDENTACIONANTERIOR

Por medio de estos tres casos el parser puede decidir en cuál instrucción va anidada cada instrucción que parsea.

El parser también tiene un control para generar un AST con forme el programa es parseado, para esto el AST cuenta con estos tipos de Nodo que representan diferentes partes del programa:

- TipoInstruccion: Son los nodos que representan algún tipo de instrucción

Estos son los nodos de instrucción que no pueden tener otros nodos anidados.

Importacion	Incerteza	DeclaracionVariable
LlamadaFuncion	Asignacion	Continuar
Detener	Retorno	DibujarTabla
DibujarAST	DibujarExpresion	Mostrar

Los siguientes nodos de instrucción si pueden tener otros nodos de instrucción anidados.

DeclaracionFuncion	Si	Sino

En el caso del nodo Sino, este solo deberia ser parte de un nodo Si

- TipoNoTerminal: Los nodos no terminales son nodos que representan la antesala de un grupo de otros nodos.

Parametros

Identificadores

DeclaracionParametro

DeclaracionParametros

CondicionInicialPar

Instrucciones

ParametrosMostrar

- TipoDato: Los nodos de tipo dato son nodos usados para representar como su nombre dice, tipos de datos.

Int	Double	String
Char	Boolean	Void

- TipoExpresionMatematica: Son nodos que representan operaciones matematicas

Suma	Resta	Multiplicacion
Division	Modulo	Potencia
MenosUnitario	Grupo	

- TipoExpresionRelacional: Son nodos que representan operaciones relacionales

MayorQue	MenorQue	MayorIgualQue
MenorIgualQue	Igualdad	Diferencia
Incerteza		

- TipoExpresionLogica: Son nodos que representan operaciones logicas

And	Xor
Or	Not

Cada vez que el árbol se encuentra con un nodo de importación analiza el contenido y busca en el editor algún archivo con el nombre de la importación, este luego es agregado a una lista con los archivos importados.

Una vez el parser analiza el archivo procede a realizar el mismo procedimiento con los archivos en la lista de archivos haciendo el mismo procedimiento de encontrar importaciones y agregarlas a la lista, hasta que ya no haya archivos que analizar.

Analizador Semántico:

Luego de analizar sintácticamente todos los archivos estos se organizan de manera que los archivos con menos o ninguna importación sean los primeros en ser analizados sintácticamente en este orden los archivos posteriormente son analizados semánticamente.

El análisis semántico se encarga de analizar que contextualmente un archivo es correcto para esto recorre el AST generado de cada archivo utilizando un metodo de visitante que visita y "observa" el tipo de nodo en el que se encuentra y decidiendo su siguiente accion a partir de esto, el analizador semantico analiza y genera una tabla de simbolos a partir de ciertas acciones.

La tabla de símbolos tiene la siguiente estructura:

Tabla Padre			
Símbolos	Retornabilidad	Anulabilidad	Tiene Control
Tablas Anidadas			

Cada tabla de símbolos apunta a su tabla padre como también apunta a sus diferentes tablas anidadas, también guarda información acerca de la Retornabilidad, Anulabilidad y Control de la tabla que se explicará más adelante y por supuesto los símbolos de la tabla.

Los símbolos de la tabla son un Map donde la llave es un string que representa una llave a un valor, y el valor es un objeto de tipo atributo.

El objeto atributo puede representar características de alguna variable, función o una instrucción Mostrar. Este objeto guarda el nombre de la variable o función, su tipo de dato y su posición, para las variables también guarda su valor y un indicador de inicialización, para las funciones guarda el nodo de sus instrucciones y una lista de sus parámetros, por otro lado los atributos de una instrucción mostrar guardan una versión formateada de su interpolación que se explicará más adelante.

Reglas de Retornabilidad:

Las reglas de retornabilidad describen como el analizador semántico comprueba que una función que debe retornar un valor tiene un retorno asegurado, esto para evitar un caso donde una función no pueda retornar un valor porque no hay una instrucción *return*.

Para esto el las tablas de símbolos tienen dos parámetros:

1. **Anulabilidad**

   La Anulabilidad es un parámetro que indica si existe la posibilidad de omitir completamente un bloque de código, esto normalmente ocurre cuando una condición no se cumple. La anulabilidad es inherente al tipo de bloque de código.

2. **Retornabilidad**

   La Retornabilidad indica si hay alguna instrucción *return* accesible SIEMPRE dentro del bloque de código. La retornabilidad depende de la existencia de instrucciones return dentro del bloque de código.

Cada tipo de bloque de código tiene normalmente asociado un tipo de scope para poder ser identificado, con ello podemos saber cómo tratar su anulabilidad y retornabilidad.

La anulabilidad está organizada por tipo de bloque del siguiente modo:

1. **Bloque *Si***

   Un bloque Si es anulable, esto debido a que si la condición del Si no se cumple el bloque puede ser omitido completamente.

2. **Bloque *Sino***

   Un bloque Sino es anulable, ya que en caso de que la instrucción Si asociada a este se cumpla nunca se accedera al bloque.

3. **Bloque *Si/Sino***

   Un bloque Si/Sino no es anulable, esto debido a que dada una condición si esta se cumple o no siempre se caerá de algún lado del bloque ya sea del lado del Sí o del Sino.

4. Bloque *Para, Mientras*

Los bloques Para y Mientras son anulables esto debido a que para acceder a estos es necesario cumplir con una condición, condición que si en primera instancia no se cumple el bloque es omitido completamente.

El que un ciclo tenga un control asegurado está dado siguiendo las reglas de retornabilidad pero en lugar de analizar las instrucciones *return* se analizan las instrucciones *Continuar* y *Detener*.

La retornabilidad está da por la siguiente regla:

"Un bloque será retornable siempre y cuando tenga dentro un bloque que sea retornable y no anulable, o que exista una instrucción return declarada en el primer nivel del scope del bloque."

La retornabiliad para los bloques de ciclo también depende de la existencia de instrucciones de control como *Continuar* y *Detener* ya que estos cambian el flujo del ciclo, esto puede causar que instrucciones *return* que eran accesibles en principio, debido a estas instrucciones ya no lo sean, por lo tanto la retornabilidad en el caso de ciclos está dada por la regla:

"Un bloque de ciclo no será retornable si antes de un retorno asegurado existe una sentencia de control siempre accesible"

Formato Mostrar:

Para analizar la interpolación de la instrucción mostrar se hace uso de un analizador léxico que tiene las siguientes expresiones regulares:

"{" -> parametro_state

<parametro_state>"}"

<parametro_state>[^}]+<<EOF>>

<parametro_state>[^}]+

 <<EOF>>

\s

El trabajo de este analizador léxico es el no solo comprobar que las interpolaciones de un mostrar estén bien hechas sino también para separar el string de interpolación en sus partes creando un arreglo de strings y Array para poder saber como posteriormente hacer el reemplazo de los parámetros.

Un ejemplo es la cadena:

“El resultado de la operacion {1} es {0}”

Una vez es analizado queda en un arreglo de la forma:

“El resultado de la operación “

Arreglo[“1”]

“ es “

Arreglo[“0”]

En este formato se pueden identificar las interpolaciones del resto del string más fácilmente para el intérprete, este arreglo una vez realizadas todas las validaciones sobre él es guardado en la tabla de símbolos.

Intérprete:

Por último el intérprete es la fase en el proceso de interpretación que se encarga de la ejecución del programa. Para esto utiliza un “Administrador de Procesos” que se encarga de llamar funciones, incluida la función principal.

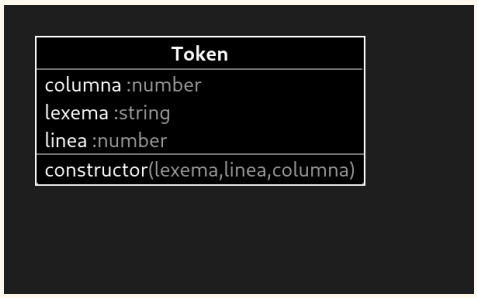
El administrador de procesos para llamar a algún proceso toma como parámetro la llave de la función que se quiere ejecutar, sus parámetros y el archivo en donde se encuentra el proceso.

El administrador crea una copia del estado de la tabla de símbolos de antes de la ejecución del procedimiento para mantener un registro del estado actual de la función en caso que exista recursividad. Posteriormente utiliza la lista de parámetros para sustituir los valores en la tabla por la de los parámetros. Luego gracias al nodo de instrucciones guardado en el atributo de la tabla de símbolos se crea una nueva instancia de intérprete que interprete la función. Una vez la función esté interpretada se guarda cualquier valor que pudiese haber devuelto, restaura el estado de la tabla de símbolos con el registro y posteriormente devuelve el valor retornado si existiese alguno.

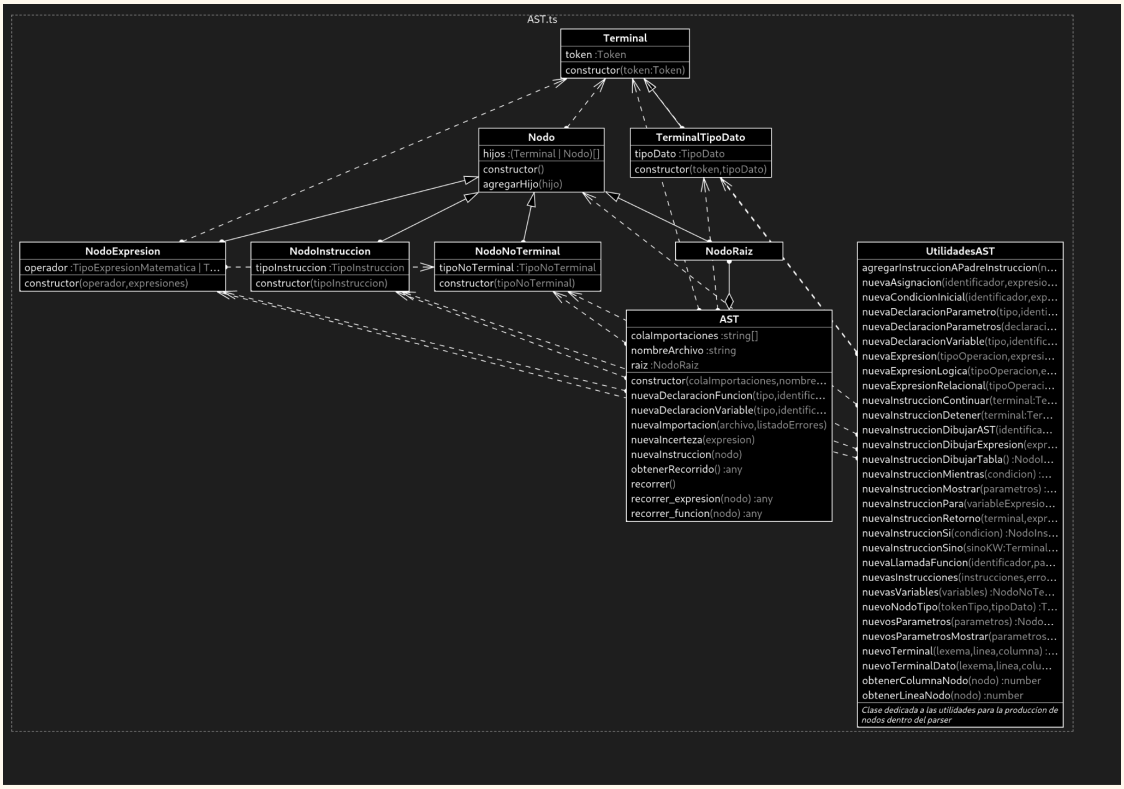
El guardado del registro debe de convertir la tabla de símbolos y sus anidados en un objeto Registro que guarda los valores de sus símbolos, esto debido a que Typescript no permite deepcopy fácilmente de objetos, para esto se obtiene el valor de algún símbolo y posteriormente se guarda en el registro como su primitivo, este proceso se repite para los anidados de la tabla recursivamente.

Diagrama de Clases:

Token



AST



Analizador Semántico

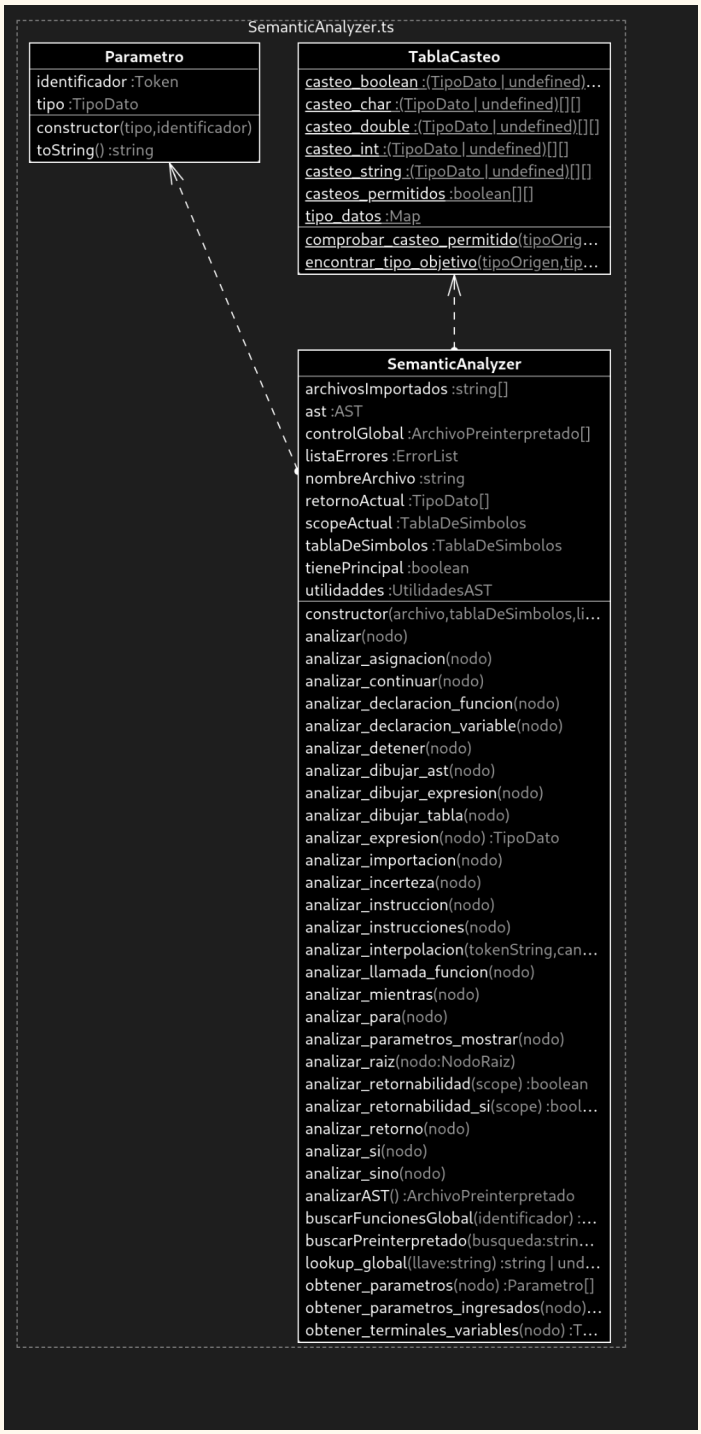
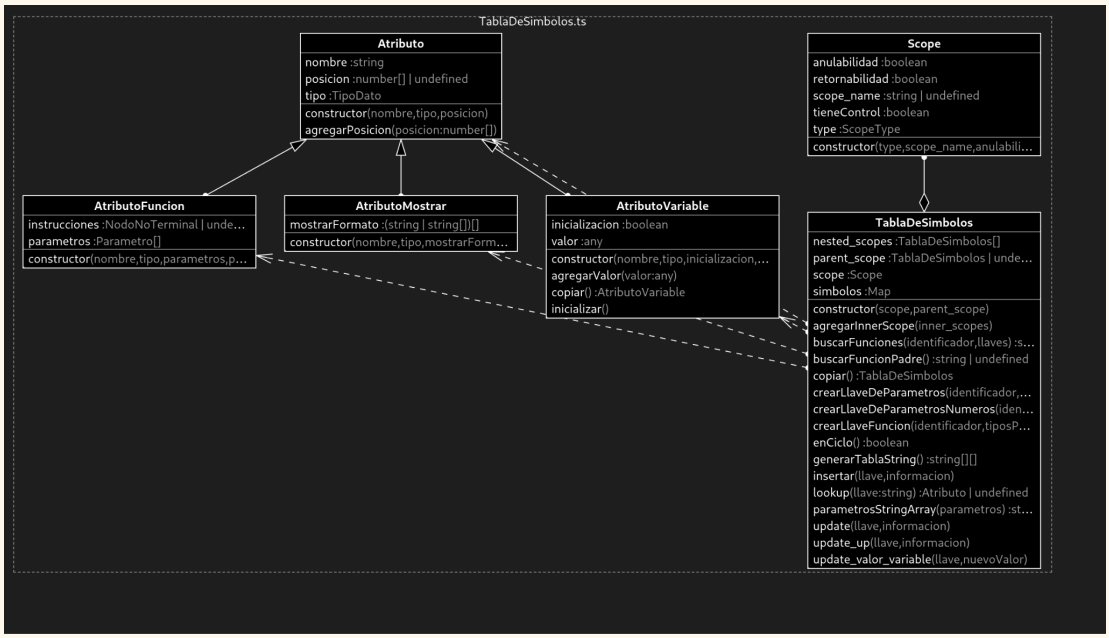
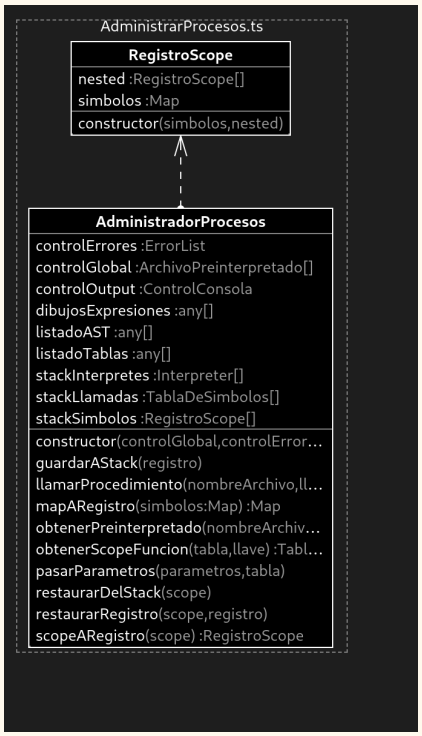


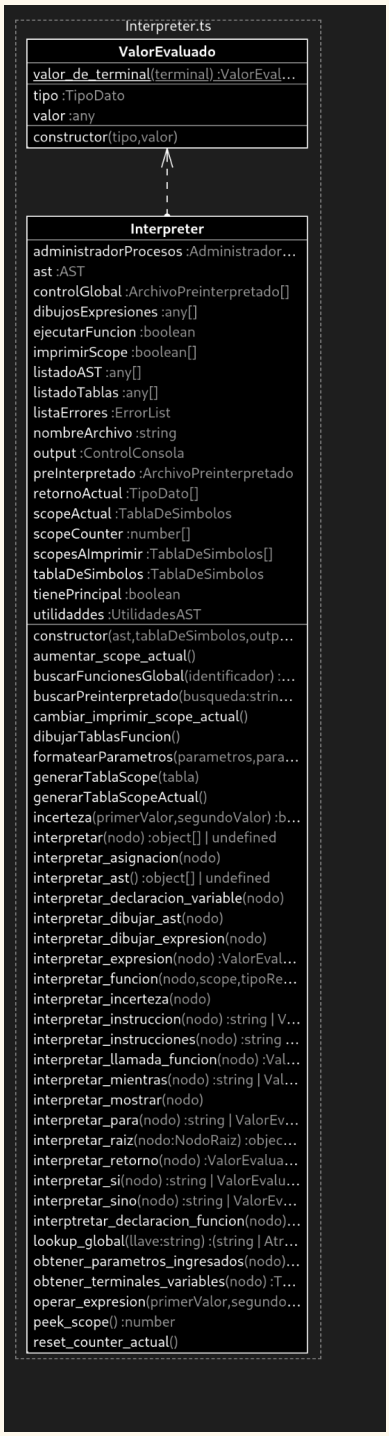
Tabla de Símbolos



Administrador De Procesos



Intérprete


Interpreter.ts
ValorEvaluado
valor_de_terminal(terminal) :ValorEval...
tipo :TipoDato
valor :any
constructor(tipo,valor)
Interpreter
administradorProcesos :Administrador...
ast :AST
controlGlobal :ArchivoPreinterpretado[]
dibujosExpresiones :any[]
ejecutarFuncion :boolean
imprimirScope :boolean[]
listadoAST :any[]
listadoTablas :any[]
listaErrores :ErrorList
nombreArchivo :string
output :ControlConsola
preInterpretado :ArchivoPreinterpretado
retornoActual :TipoDato[]
scopeActual :TablaDeSimbolos
scopeCounter :number[]
scopesAImprimir :TablaDeSimbolos[]
tablaDeSimbolos :TablaDeSimbolos
tienePrincipal :boolean
utilidaddes :UtilidadesAST
constructor(ast,tablaDeSimbolos,outp...
aumentar_scope_actual()
buscarFuncionesGlobal(identificador) :...
buscarPreinterpretado(busqueda:strin...
cambiar_imprimir_scope_actual()
dibujarTablasFuncion()
formatearParametros(parametros,para...
generarTablaScope(tabla)
generarTablaScopeActual()
incerteza(primerValor,segundoValor) :b...
interpretar(nodo) :object[] | undefined
interpretar_asignacion(nodo)
interpretar_ast() :object[] | undefined
interpretar_declaracion_variable(nodo)
interpretar_dibujar_ast(nodo)
interpretar_dibujar_expresion(nodo)
interpretar_expresion(nodo) :ValorEval...
interpretar_funcion(nodo,scope,tipoRe...
interpretar_incerteza(nodo)
interpretar_instruccion(nodo) :string | V...
interpretar_instrucciones(nodo) :string ...
interpretar_llamada_funcion(nodo) :Val...
interpretar_mientras(nodo) :string | Val...
interpretar_mostrar(nodo)
interpretar_para(nodo) :string | ValorEv...
interpretar_raiz(nodo:NodoRaiz) :objec...
interpretar_retorno(nodo) :ValorEvalua...
interpretar_si(nodo) :string | ValorEvalu...
interpretar_sino(nodo) :string | ValorEv...
interptretar_declaracion_funcion(nodo)...
lookup_global(llave:string) :(string | Atr...
obtener_parametros_ingresados(nodo)...
obtener_terminales_variables(nodo) :T...
operar_expresion(primerValor,segundo...
peek_scope() :number
reset_counter_actual()