Manual técnico

Información del sistema:

- OS: `Arch Linux`
- Kernel: `x86_64 Linux 5.16.8-arch1-1`
- CPU: `Intel Core i3-4005U @ 4x 1.7GHz`
- GPU: `Intel Corporation Haswell-ULT Integrated Graphics Controller`
- RAM: `7881MiB`
- IDE: `Visual Studio Code`
- Control de versiones: `git version 2.36.1`

Estructura del proyecto

```
.
├── analyze
│   ├── Analyzer.ts
│   ├── CRLFile.ts
│   ├── DotExpGeneratorVisitor.ts
│   ├── DotGeneratorVisitor.ts
│   ├── ExecuteBlocks.ts
│   ├── ExecuteVisitor.ts
│   ├── ExpressionsVisitor.ts
│   ├── ImportsVisitor.ts
│   ├── SymTableVisitor.ts
│   ├── SymTableVisitorGlobal.ts
│   └── Visitor.ts
├── app
│   ├── editor-manager
│   │   ├── editor-item.ts
│   │   ├── editor-manager.component.css
│   │   ├── editor-manager.component.html
│   │   ├── editor-manager.component.spec.ts
│   │   ├── editor-manager.component.ts
│   │   ├── editor.component.ts
│   │   ├── editor.directive.ts
│   │   ├── result-item.ts
│   │   ├── result.component.ts
│   │   ├── result.directive.ts
│   │   ├── tab-item.ts
│   │   ├── tab.directive.ts
│   │   └── tableResult.component.ts
│   ├── service
│   │   └── graphviz.service.ts
│   ├── tab-header
│   │   ├── tab-header.component.css
│   │   ├── tab-header.component.html
│   │   ├── tab-header.component.spec.ts
│   │   └── tab-header.component.ts
│   ├── text-editor
│   │   ├── text-editor.component.css
│   │   ├── text-editor.component.html
│   │   ├── text-editor.component.spec.ts
│   │   └── text-editor.component.ts
│   ├── app-routing.module.ts
│   ├── app.component.css
│   ├── app.component.html
│   ├── app.component.spec.ts
│   ├── app.component.ts
│   └── app.module.ts
├── assets
│   ├── images
│   │   ├── CompiReportLanguage.png
│   │   └── crlIde.png
```

```
│   └── .gitkeep
├── astMembers
│   ├── Node.ts
│   └── SymbolTable.ts
├── environments
│   ├── environment.prod.ts
│   └── environment.ts
├── errors
│   └── LogError.ts
├── parser
│   ├── ast.ts
│   ├── crl-parser.jison
│   └── parser.js
├── favicon.ico
├── index.html
├── main.ts
├── polyfills.ts
├── styles.css
└── test.ts
```

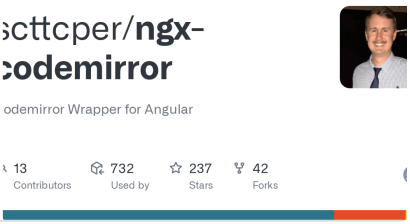
Librerías y herramientas utilizadas

- `ngx-codemirror`

GitHub - scttcper/ngx-codemirror: Codemirror Wrapper for Angular

DEMO: https://ngx-codemirror.vercel.app Latest version available for each version of Angular An Angular component wrapper for CodeMirror that extends ngModel. Based on JedWatson/react-codemirror Used in: tsquery

https://github.com/scttcper/ngx-codemirror



- `Bootstrap 5.1.3`

Introduction

Looking to quickly add Bootstrap to your project? Use jsDelivr, a free open source CDN. Using a package manager or need to download the source files? Head to the downloads page. Copy-paste the

https://getbootstrap.com/docs/5.1/getting-started/introduction/

- `Ts-graphviz`

ts-graphviz

Graphviz library for TypeScript. Export Dot language. TypeScript Support. Supports multiple runtime and module. Node.js, Browser and Deno. UMD, ESM, CommonJS Zero dependencies. The

https://www.npmjs.com/package/ts-graphviz

- `Angular`

Angular

Angular is a platform for building mobile and desktop web applications. Join the community of millions of developers who build compelling user interfaces with Angular.

https://angular.io/

- `Graphviz API`

GraphViz API

QuickChart is an open-source API that generates GraphViz charts. The API takes a graph description in the DOT language and renders it with the latest stable version of GraphViz, returning the output as PNG or SVG. To get started, use the https://quickchart.io/graphviz endpoint. Here's a simple example: https://quickchart.io/graphviz?

https://quickchart.io/documentation/graphviz-api/

CRL

CRL `(Compi Report Lenguaje)` es un lenguaje de programación para que los usuario puedan comprender cómo se generan los sub-árboles de análisis sintáctico de las sentencias de control de un lenguaje, los métodos y funciones de una clase y las diversas tablas de símbolos que se crean en tiempo de compilación en los ámbitos y sub ámbitos de funciones y sentencias de control. Por lo que el lenguaje permitirá generar reportes gráficos de estas estructuras y así obtener una mejor comprensión del funcionamiento del lenguaje.

Definición de tokens

Palabras reservadas

Palabra	Descripción
`Importar`	Para hacer imports
`Incerteza`	Definir incerteza
`clr`	Extensión del archivo
`Double`	Número decimal
`Boolean`	Lódigo
`String`	Cadena
`Int`	Número entero
`Char`	Caracter
`Void`	Tipo de función
`true`	Valor lógico
`false`	Valor lógico
`Principal`	Para la función principal
`Retorno`	Para retornar valores en funciones
`Para`	Ciclo
`Mientras`	Ciclo
`Si`	Condicional
`Sino`	Condicional else
`Detener`	Detener un ciclo
`Continuar`	Ir a la siguiente iteración
`Mostrar`	Función para el formateo de strings
`DibujarAST`	Función para dibujar el árbol sintáctico
`DibujarEXP`	Para dibujar una expresión
`DibujarTS`	Para dibujar la tabla de símbolos

Operadores aritméticos

Operador	Descripción
`+`	Suma
`-`	Resta

Operador	Descripción
`*`	Multiplicación
`/`	División
`%`	Mod
`^`	Exponente(Precedencia por la derecha)

Operadores relacionales

Operador	Descripción
`==`	Comparación
`!=`	Diferente
`<`	Menor que
`>`	Mayor que
`<=`	Menor igual
`>=`	Mayor igual
`~`	Incerteza

Operadores lógicos

Operador	Descripción
`&&`	Operador AND
`||`	Operador OR
`|&`	Operador XOR
`!`	Operador NOT

Otros signos

Signo	Descripción
`.`	Punto
`:`	Dos puntos
`;`	Punto y coma
`,`	Coma

Signo	Descripción
`(`	Paréntesis abrir
`)`	Paréntesis cerrar
`{`	Llaves abrir
`}`	Llaves cerrar
`=`	Igual
`“`	Comillas dobles
`'`	Comillas simples
`!!`	Inicio comentario de una línea
`’’’`	Inicio y fin comentario multilínea

Análisis sintáctico

Este lenguaje cuenta con diferentes declaraciones e instrucciones. Funciones, ciclos, variables, entre otras. A continuación se presenta un resumen de lo que es el archivo de configuración de `Jison`.

Importar archivos

```
import
    : 'Importar' NAME '.' 'crl' new_line_opt
    {
        $$ = new yy.ImportDeclaration(@$, $2);
    }
```

Incerteza

```
incert
    : Incerteza NUMBER new_line_opt
    {$$ = new yy.Incerteza(@$, Number($2))}
    ;
```

Declaración de variables

Para la declaración de variables se toma una lista, aunque para construir el AST se agregan individualmente

```
variable_declarators
    : type variable_list_declarators
    {
        $2.forEach(v => v.type = $1);
        //$$ = new yy.VariableDeclaration(@$, $2, $1);
        $$ = $2;
    }
    ;

variable_list_declarators
    : variable_declarator
    {$$ = [$1]}
    | variable_list_declarators ',' variable_declarator
    {
        $1.push($3);
        $$ = $1;
    }
    ;

variable_declarator
    : variable_id
    {
        $$ = new yy.VariableDeclarator(@$, $1, null);
    }
    | variable_assignment
    {
        $$ = new yy.VariableDeclarator(@$, $1.id, $1.expression);
    }
    ;

variable_id
    : NAME
    {$$ = new yy.Identifier(@$, $1);}
    ;
```

Asignación de variables

```
variable_assignment
    : variable_id '=' expression
    {$$ = new yy.Assignment(@$, $1, $3)}
    ;
```

Declaración de funciones

```
method_declarator
    : type NAME '(' params_list_opt ')' ':' block
    {
        $$ = new yy.functionDeclaration(@$, $2, $4, $1, $7);
    }
    | 'Void' NAME '(' params_list_opt ')' ':' block
    {
        $$ = new yy.functionDeclaration(@$, $2, $4, yy.Type.Void, $7);
    }
    | 'Void' 'Principal' '(' params_list_opt ')' ':' block
    {
        $$ = new yy.functionMain(@$, $7);
    }
    ;
```

Bloques

Los bloques se usan para instrucciones las cuales tiene una tabla de símbolos, tal como: funciones, ciclos y condicionales

```
block
    : NEWLINE INDENT body_block_opt DEDENT
    {$$ = $3}
    ;

body_stmt
    : variable_declarators new_line_opt
    {$$ = $1}
    | variable_assignment new_line_opt
    {$$ = $1}
    | function_call_stmt new_line_opt
    {$$ = $1}
    | if_stmt
    {$$ = $1}
    | for_stmt
    {$$ = $1}
    | while_stmt
    {$$ = $1}
    | drawAST_stmt new_line_opt
    {$$ = $1}
    | drawEXP_stmt new_line_opt
    {$$ = $1}
    | drawTS_stmt new_line_opt
    {$$ = $1}
    | show_stmt new_line_opt
    {$$ = $1}
    | small_stmt
    {$$ = $1}
    ;
```

Instrucción SI

```
if_stmt
    : 'Si' '(' expression ')' ':' block else_opt
    {$$ = new yy.IfStmt(@$, $3, $6, $7)}
    ;
```

Ciclos

```
for_stmt
    : 'Para' '(' init_for ';' expression ';' op_for ')' ':' block
    {
        $$ = new yy.forStmt(@$, $10, $3, $5, $7);
    }
    ;

while_stmt
    : 'Mientras' '(' expression ')' ':' block
    {$$ = new yy.whileStmt(@$, $6, $3);}
    ;
```

Otras instrucciones

Funciones e instrucciones propias del lenguaje como el `Retorno`, `Detener`, `Continuar` y otras funciones que sirven para ver el funcionamiento de un `AST`.

```
show_stmt
    : 'Mostrar' '(' string_literal format_expressions_opt ')'
    {$$ = new yy.Mostrar(@$, $3, $4);}
    ;

drawAST_stmt
    : 'DibujarAST' '(' variable_id ')'
    {$$ = new yy.DibujarAST(@$, $3);}
    ;

drawEXP_stmt
    : 'DibujarEXP' '(' expression ')'
    {$$ = new yy.DibujarEXP(@$,$3);}
    ;

drawTS_stmt
```

```
    : 'DibujarTS' '(' ')'
    {$$ = new yy.DibujarTS(@$);}
    ;

return_stmt
    : 'Retorno' return_expression_opt
    {$$ = new yy.returnStmt(@$, $2)}
    ;

return_expression_opt
    :
    {$$ = null;}
    | expression
    {$$ = $1;}
    ;

break_stmt
    : 'Detener'
    {$$ = new yy.breakStmt(@$);}
    ;
```

Análisis semántico

Para el análisis semántico, con el `AST` previamente armado en el análisis sintáctico, se utilizan el patrón de diseño `Visitor`. Se crean varios `visitors` para las diferentes fases de recorrido del árbol.

El árbol no es recorrido en su totalidad en todas las fases y tampoco se recorre de la misma manera. Existe una función de visitor para todos los nodos.

```
visit(node: Node): void {
    switch (node.constructor.name) {
      case Program.name:
        this.visitProgram(node as Program);
        break;
      case ImportDeclaration.name:
        this.visitImportDeclaration(node as ImportDeclaration);
        break;
      case Incerteza.name:
        this.visitIncerteza(node as Incerteza);
        break;
      case Identifier.name:
        this.visitIdentifier(node as Identifier);
        break;
      case VariableDeclarator.name:
        this.visitVariableDeclarator(node as VariableDeclarator);
```

```
        break;
      case BinaryExpression.name:
        this.visitBinaryExpression(node as BinaryExpression);
        break;
      case LogicalExpression.name:
        this.visitLogicalExpression(node as LogicalExpression);
        break;
      case UnaryExpression.name:
        this.visitUnaryExpression(node as UnaryExpression);
        break;
      case Assignment.name:
        this.visitAssignment(node as Assignment);
        break;
      case CallFunction.name:
        this.visitCallFunction(node as CallFunction);
        break;
      case functionParam.name:
        this.visitfunctionParam(node as functionParam);
        break;
      case returnStmt.name:
        this.visitreturnStmt(node as returnStmt);
        break;
      case continueStmt.name:
        this.visitcontinueStmt(node as continueStmt);
        break;
      case breakStmt.name:
        this.visitbreakStmt(node as breakStmt);
        break;
      case functionDeclaration.name:
        this.visitfunctionDeclaration(node as functionDeclaration);
        break;
      case functionMain.name:
        this.visitfunctionMain(node as functionMain);
        break;
      case IfStmt.name:
        this.visitIfStmt(node as IfStmt);
        break;
      case forStmt.name:
        this.visitforStmt(node as forStmt);
        break;
      case whileStmt.name:
        this.visitwhileStmt(node as whileStmt);
        break;
      case Mostrar.name:
        this.visitMostrar(node as Mostrar);
        break;
      case DibujarAST.name:
        this.visitDibujarAST(node as DibujarAST);
        break;
      case DibujarEXP.name:
        this.visitDibujarEXP(node as DibujarEXP);
        break;
      case DibujarTS.name:
        this.visitDibujarTS(node as DibujarTS);
```

```
        break;
    }
  }
```

El visitor es llamado, dependiendo de la fase, de diferentes formas. En algunas fases puede pasar que visite primero sus hijos y luego a sí mismo. O también puede visitarse a sí mismos antes que sus hijos.

Cada nodo tiene una función `accept(args)` que se encarga de visitar primero los hijos.

```
accpet(visitor: Visitor, ambit: SymTable){}
```

Dependiendo de la fase y del visitor se pueden realizar diferentes acciones. Las fases para son las siguientes

Fases

- `SymTableVisitorGlobal`: Genera el ámbito global. Primero se cargan las funciones y variables. También verifica que no se usen variables que no han sido inicializadas o declaradas.
- `ImportsVisitor`: Busca por los imports, genera el AST y ejecuta los visitors
- `SymTableVisitor`: Genera la tabla de símbolos para las funciones y también verifica que no se usen variables que no han sido inicializadas o declaradas.
- `ExpressionsVisitor`: Valida que las expresiones sean correctas y que tengan los tipos correctos.
- `ExecuteVisitor`: Encargado de ejecutar las instrucciones aproyado de otras clases para generar las imágenes.