Manual Técnico de CompiScript+

Introducción

El proyecto CompiScript+ fue desarrollado como parte del curso de Organización de Lenguajes y Compiladores 1 en la Universidad de San Carlos de Guatemala. Su propósito es permitir a los estudiantes de Introducción a la Programación y Computación 1 aprender y experimentar con las generalidades de un lenguaje de programación diseñado específicamente para su formación académica.

Arquitectura del Software

CompiScript+ es un intérprete basado en la arquitectura cliente-servidor. El front-end se desarrolla en HTML y JavaScript, facilitando una interfaz gráfica en el navegador para la entrada de código. El back-end, implementado en Node.js, utiliza Jison para el análisis léxico y sintáctico, gestionando la ejecución del código.

Entorno de Desarrollo

- **Herramientas de Desarrollo:** Node.js, Jison para análisis, Visual Studio Code.
- **Lenguajes de Programación:** JavaScript.
- **Librerías y Frameworks:** Express.js para manejar solicitudes del servidor.

Funcionalidades Implementadas

El intérprete admite la ejecución de un conjunto básico de instrucciones de programación incluyendo:

- Tipos de datos: Enteros, Dobles, Booleanos, Caracteres, Cadenas.
- Operaciones: Aritméticas, Lógicas, de Comparación.

Las operaciones aritméticas se realizan por medio de la clase 'Operador' (Se adjuntan algunas capturas):

```
class Operador{
    constructor(){
    }

    ejecutar(raiz, pila){
        var Resultado1=null;
        var Resultado2=null;
        var Resultado=null;
        switch (raiz.tag) {
            case "EXP":
                if (raiz.childs.length==3) {
                    Resultado1=this.ejecutar(raiz.childs[0], pila);
                    Resultado2=this.ejecutar(raiz.childs[2], pila);
                    var op = raiz.childs[1].value;
                    console.log(op);
                    console.log(Resultado1, "Resultado1");
                    console.log(Resultado2, "Resultado2");
                    switch (op) {
                        case "+":
                        case "-":
                        case ",":
                        case "%":
                        case "*":
                        case "/":
                            return this.aritmetico(Resultado1,Resultado2,raiz.childs[1].fila,raiz.childs[1].
                        case "==":
                        case "!=":
                            return this.igualdad(Resultado1,Resultado2,raiz.childs[1].fila,raiz.childs[1].co
                        case ">":
                        case ">=":
                        case "<":
                        case "<=":
```

```
                        case "!=":
                            return this.igualdad(Resultado1,Resultado2,raiz.childs[1].fila,raiz.childs[1].colu
                        case ">":
                        case ">=":
                        case "<":
                        case "<=":
                            return this.relacional(Resultado1,Resultado2,raiz.childs[1].fila,raiz.childs[1].co
                        case "&&":
                        case "||":
                            return this.logicos(Resultado1,Resultado2,raiz.childs[1].fila,raiz.childs[1].colum
                        default:
                            break;
                    }
                }else if(raiz.childs.length==2){
                   if(raiz.childs[0].value=="!"){
                       Resultado1=this.ejecutar(raiz.childs[1], pila)
                       if(Resultado1.tipo=="bool"){
                           Resultado= new ResultadoOp();
                           Resultado.tipo="bool"
                           Resultado.valor=!Resultado1.valor
                           return Resultado
                       }
                   }else if(raiz.childs[0].value=="-"){

                    Resultado1=this.ejecutar(raiz.childs[1])
                       if(Resultado1.tipo=="int"){
                           Resultado= new ResultadoOp();
                           Resultado.tipo="int"
                           Resultado.valor=-Resultado1.valor
                           console.log(Resultado, "Resultado");
```

```
aritmetico(R1,R2,fila,columna,op){

        let tipo1 = R1.tipo;
        let tipo2 = R2.tipo;
        var res = new ResultadoOp();
        if(tipo1=="error"||tipo2=="error"){
            res.tipo="error";
            return res;
        }
        switch(op){
            case "+":
                switch(tipo1){
                    case "int":
                        switch(tipo2){
                            case "int":
                                res.tipo="int";
                                res.valor=R1.valor+R2.valor;
                                return res;
                            case "double":
                                res.tipo="double";
                                res.valor=R1.valor+R2.valor;
                                return res;
                            case "std::string":
                                res.tipo="std::string";
                                res.valor=R1.valor+R2.valor;
                                return res;
                            case "bool":
                                res.tipo="int";
                                res.valor=R1.valor+R2.valor;
                                return res;
                            case "char":
```

- Estructuras de Control: Condicionales (if, else), Bucles (for, while, do-while).

Los ciclos se realizan por medio de un recorrido in-order para el árbol que se genera a partir del análisis sintactico:

```
}else if(raiz.tag == "DO_WHILE"){

  let newAmbito = new Pila("do_while"+raiz.fila+raiz.columna);
  pila.push(newAmbito);
  op = new Operador()
  res = op.ejecutar(raiz.childs[4], pila)

      do{
        this.interpretar(raiz.childs[1].childs[0], pila)
        if(global.br){
          global.br = false;
          break;
        }else if(global.ret){
          break;
        }
        res = op.ejecutar(raiz.childs[4], pila)
      }while(res.valor)

      pila.pop();
```

Análisis Léxico y Sintáctico

Utiliza Jison para definir la gramática y generar el analizador sintáctico. Los tokens y las reglas gramaticales se especifican en un archivo**.jison**, el cual procesa la entrada para construir un Árbol de Sintaxis Abstracta (AST).

Gestión de Errores

Detecta y reporta errores léxicos, sintácticos y semánticos, proporcionando mensajes claros sobre la naturaleza del error y su ubicación en el código.

Reportes Generados

Genera tres tipos de reportes:

- **Errores:** Listado de errores encontrados durante el análisis.

```
var L_Error = (function(){
    var instancia;


    class Lista{
        constructor(){
            this.principio=null;
            this.fin=null;
        }

        insertar(Error){

            if(this.principio==null){
                this.principio=Error;
                this.fin=Error;
                return;
            }

            this.fin.siguiente=Error;
            Error.anterior=this.fin;
            this.fin=Error;
            console.log(this.fin);
        }

        getErroresHtml(){
            var texto = `<html><head><title>Reporte de Errores</title><style>
            table {
              border-collapse: collapse;
              width: 100%;
              border: 1px solid #ddd;
            }
            th, td {
```

- **Tabla de Símbolos:** Muestra variables y funciones junto con su alcance y tipos.

```
var TS = (function(){
    var instancia;


    class Tabla{
        constructor(){
            this.simbolos=[];
        }

        insertar(simbolo, pila) {
            // Buscar si el símbolo ya existe en la tabla
            let simboloExistente = this.simbolos.find(s => s.nombre === simbolo.nombre);

            // Si el símbolo existe, actualizar su valor
            if (simboloExistente) {
                simboloExistente.valor = simbolo.valor;
                simboloExistente.tipo = simbolo.tipo;  // Opcionalmente actualizar el tipo si es necesario
                console.log(`Valor actualizado para el símbolo ${simbolo.nombre} a ${simbolo.valor}`);
            } else {
                // Si no existe, agregar el nuevo símbolo a la tabla
                this.simbolos.push(simbolo);
                console.log(`Símbolo ${simbolo.nombre} agregado con valor ${simbolo.valor}`);
            }
        }

        getsimbolos(){
            var texto="";

            texto+=`<html><head><title>Reporte de Tabla de Simbolos</title><style>
            table {
                border-collapse: collapse;
                width: 100%;
```

- **AST:** Representación gráfica del árbol de sintaxis abstracta.

Pruebas Realizadas

Se realizaron pruebas unitarias y de integración para validar todas las funcionalidades del intérprete, asegurando que los errores comunes se manejan adecuadamente y que el análisis y ejecución de los programas es correcto.

Conclusión

CompiScript+ es una herramienta efectiva para el aprendizaje de conceptos de programación. Aunque actualmente cubre funcionalidades básicas, presenta una base sólida para extensiones futuras y mejoras.