Manual de usuario
CompiScript++

Introducción

Bienvenido al Manual Técnico de CompiScript+, un recurso diseñado para proporcionar una comprensión profunda de la arquitectura, diseño y funcionamiento interno del lenguaje de programación CompiScript+.

Este manual está dirigido a desarrolladores, estudiantes y entusiastas de la programación que deseen explorar en detalle cómo funciona CompiScript+ bajo el capó. A lo largo de este documento, analizaremos los aspectos técnicos del lenguaje, incluyendo su diseño de sintaxis, estructuras de datos internas, algoritmos de análisis léxico y sintáctico, así como la implementación de características clave.

¿Qué es CompiScript+ desde una perspectiva técnica?

CompiScript+ es un lenguaje de programación diseñado como parte del proyecto del curso de Organización de Lenguajes y Compiladores 1. Desde una perspectiva técnica, CompiScript+ se implementa como un conjunto de reglas gramaticales definidas por una gramática formal, que luego son procesadas por un analizador léxico y un analizador sintáctico para generar un árbol de análisis sintáctico. Este árbol se utiliza para comprender la estructura del programa y facilitar la generación de código ejecutable.

Requisitos del Sistema

Servidor (Backend):

Node.js: El servidor está construido en Node.js, por lo que necesitarás tener Node.js instalado en tu sistema.

Dependencias del proyecto:

express: Un framework web para Node.js que se utiliza para crear la API del servidor.

jison: Una herramienta para generar analizadores léxicos y sintácticos a partir de gramáticas.

cors: Middleware para habilitar CORS (Cross-Origin Resource Sharing) en el servidor.

Compilación y ejecución:

typescript: El código del servidor está escrito en TypeScript, por lo que necesitarás compilarlo antes de ejecutarlo.

tsc-watch: Una herramienta para compilar automáticamente el código TypeScript cuando se realicen cambios.

Scripts de npm:

start: Inicia el servidor Node.js.

dev: Inicia el servidor en modo de desarrollo con recarga automática.

test: Ejecuta pruebas (debes definir tus propias pruebas).

Cliente (Frontend):

React: El frontend está construido en React, por lo que necesitarás tener React instalado en tu sistema.

Dependencias del proyecto:

@monaco-editor/react: Un componente para integrar el editor de código Monaco en tu aplicación.

@testing-library/react: Biblioteca para escribir pruebas unitarias en React.

react-dom: Biblioteca para renderizar componentes React en el navegador.

web-vitals: Herramienta para medir el rendimiento de tu aplicación web.

Scripts de npm:

start: Inicia la aplicación React en modo de desarrollo.

build: Crea una versión optimizada para producción de la aplicación.

test: Ejecuta pruebas (debes definir tus propias pruebas).

Arquitectura del Programa

Componentes Principales:

Servidor (Backend):

- API REST: El servidor expone una API REST para recibir solicitudes desde el cliente y procesarlas.

- Analizador Léxico y Sintáctico: Utiliza la gramática definida en el archivo analizador.jison para analizar y compilar el código fuente.

- Manejo de Errores: Implementa mecanismos para manejar errores léxicos, sintácticos y semánticos.

- Seguridad: Implementa medidas de seguridad, como validación de tokens o autenticación de usuarios.

Cliente (Frontend):

- Editor de Código: Utiliza el componente Monaco Editor para permitir a los usuarios escribir y editar código.

- Interfaz de Usuario (UI): Proporciona una interfaz amigable para que los usuarios interactúen con el compilador.

- Comunicación con el Servidor: Realiza solicitudes HTTP al servidor para compilar y ejecutar el código.

- Manejo de Resultados: Muestra los resultados de la compilación y ejecución al usuario.

Flujo de Trabajo:

- El usuario escribe código en el editor de código del cliente.
- El cliente envía el código al servidor a través de la API REST.
- El servidor utiliza el analizador léxico y sintáctico para verificar la validez del código.
- Si no hay errores, el servidor compila el código y devuelve los resultados al cliente.
- El cliente muestra los resultados al usuario (por ejemplo, mensajes de éxito o errores).

Definición del lenguaje

La gramática del lenguaje CompiScript está diseñada para proporcionar una estructura clara y precisa que permita la escritura de programas concisos y expresivos.

Esta gramática define las reglas sintácticas que rigen la construcción de programas en CompiScript, especificando cómo se pueden combinar diferentes elementos del lenguaje para formar expresiones y sentencias válidas.

Gramática

La gramática del lenguaje CompiScript se define utilizando la notación de BackusNaur Form (BNF), que describe las reglas de producción que generan las construcciones sintácticas del lenguaje. Estas reglas especifican la estructura de las sentencias, expresiones y otros componentes del programa.

Por ejemplo, las reglas de producción pueden definir cómo se pueden declarar variables, cómo se pueden realizar operaciones aritméticas o cómo se pueden crear gráficos. Cada regla describe una construcción sintáctica en términos de sus componentes más básicos o en términos de otras construcciones sintácticas.

Terminales y No Terminales La gramática del lenguaje CompiScript utiliza una combinación de símbolos terminales y no terminales. Los símbolos terminales representan tokens específicos del lenguaje, como palabras clave, operadores y símbolos de puntuación. Los símbolos no terminales representan categorías gramaticales más amplias, como expresiones, sentencias y programas completos.

Acciones Semánticas

Además de definir la estructura sintáctica del lenguaje, la gramática de CompiScript incluye acciones semánticas que se ejecutan durante el análisis sintáctico. Estas acciones son bloques de código embebido que realizan operaciones específicas.

Gramática

La gramática utilizada es la siguiente:

<ini> ::= <instrucciones> EOF

<instrucciones> ::= <instrucciones> <instruccion>
| <instruccion>

<instruccion>::= <cout> PYC
| <if_g>
|<declaracion> PYC
| <asignacion> PYC
|<ciclo_while>
|<instruccion_break> PYC
|<instruccion_continue> PYC
|<ciclo_do_while>
|<incremento PYC>
|<for_g>
|<declaracion_metodo>
|<acceso_metodo> PYC
|<instruccion_return> PYC
|<execute> PYC
|<switch_g>
|<declaracion_array> PYC
|<asignacion_array> PYC

<expresion> ::= RES <expresion>

| <expresion> MAS <expresion>

| <expresion> RES <expresion>

| <expresion> MUL <expresion>

| <expresion> DIV <expresion>

| <expresion> MOD <expresion>

| ENTERO

| DOUBLE

| CADENA

| CARACTER

| TRUE

| FALSE

| ENDL

| POW PARIZQ <expresion> COMA <expresion> PARDER

| PARIZQ <expresion> PARDER

| <relacionales>

| <logico>

| ID

| expresion INTERROGACION <expresion> DOSPUNTOS <expresion>

| <acceso_metodo>

| PARIZQ <tipos_datos> PARDER <expresion>

|<acceso_array>

|TOLOWER PARIZQ <expresion> PARDER

|TOUPPER PARIZQ <expresion> PARDER

|ROUND PARIZQ <expresion> PARDER

|ID PUNTO LENGTH PARIZQ PARDER

|TYPEOF PARIZQ <expresion> PARDER

|TOSTRING PARIZQ <expresion> PARDER

|ID PUNTO C_STR PARIZQ PARDER

<relacionales> ::= <expresion> IGUAL <expresion>

| <expresion> DISTINTO <expresion>

| <expresion> MENOR <expresion>

| <expresion> MENORIGUAL <expresion>

| <expresion> MAYOR <expresion>

| <expresion> MAYORIGUAL <expresion>

<logico> ::= <expresion> AND <expresion>

| <expresion> OR <expresion>

| NOT <expresion>

<bloque>::= LLAVEIZQ <instrucciones> LLAVEDER

| LLAVEIZQ LLAVEDER

<cout> ::= COUT OUTPUT <lista_expresiones>

<lista_expresiones>: <lista_expresiones> OUTPUT <expresion>

| <expresion>

<if_g> ::= IF PARIZQ <expresion> PARDER <bloque>

| IF PARIZQ <expresion> PARDER <bloque> ELSE <bloque>

| IF PARIZQ <expresion> PARDER <bloque> ELSE <if_g>

<incremento> ::= ID MAS MAS

| ID RES RES

<tipos_datos> ::= INT

| DOUBLE_ID

| BOOL

| CHAR

| CADENA_ID

| VOID

<lista_ids> ::= <lista_ids> COMA ID

| ID

<declaracion> ::= <tipos_datos> <lista_ids> ASIGNACION <expresion>

| <tipos_datos lista_ids>

<asignacion> ::= ID ASIGNACION <expresion>

<ciclo_while> ::= WHILE PARIZQ <expresion> PARDER <bloque>

<instruccion_break> ::= BREAK

<instruccion_continue> ::= CONTINUE

<instruccion_return> ::= RETURN <expresion>
| RETURN

<ciclo_do_while> ::= DO <bloque> WHILE PARIZQ <expresion> PARDER

<for_g> ::= FOR PARIZQ <declaracion> PYC <expresion> PYC <actalizacion> PARDER <bloque>

<actalizacion> ::= <incremento>
| <asignacion>

<declaracion_metodo> ::= <tipos_datos> ID PARIZQ <lista_parametros> PARDER <bloque>
| <tipos_datos> ID PARIZQ PARDER <bloque>

<lista_parametros> ::= <lista_parametros> COMA <parametro>

| <parametro>

<parametro> ::= <tipos_datos> ID

<lista_parametros_acceso> ::= <lista_parametros_acceso> COMA <expresion>
| <expresion>

<acceso_metodo> ::= ID PARIZQ <lista_parametros_acceso> PARDER
| ID PARIZQ PARDER

<execute> ::= EXECUTE0 <acceso_metodo>

<switch_g> ::= SWITCH PARIZQ <expresion> PARDER <bloque_case>

<bloque_case> ::= LLAVEIZQ <casos> LLAVEDER

<casos> ::= <casos> <caso>
| <caso>

<caso> ::= CASE <expresion> DOSPUNTOS <instrucciones>

| DEFAULT DOSPUNTOS <instrucciones>

<declaracion_array>::= <tipos_datos> ID <lista_corchetes_declaracion> ASIGNACION NEW <tipos_datos lista_corchetes_declaracion>

| <tipos_datos> ID <lista_corchetes_declaracion> ASIGNACION CORIZQ <lista_corchetes_valores_asignacion> CORDER

<lista_corchetes_declaracion>::= <lista_corchetes_declaracion> <valor_arreglo>

| <valor_arreglo>

<valor_arreglo>::= CORIZQ <expresion> CORDER

| CORIZQ CORDER

<lista_corchetes_valores> ::= <lista_corchetes_valores> <valor_arreglo>

| <valor_arreglo>

<lista_corchetes_valores_asignacion> ::= <lista_corchetes_valores_asignacion> COMA <elemento>

| <elemento>

<elemento> ::= <expresion>

| CORIZQ <lista_corchetes_valores_asignacion> CORDER

<acceso_array> ::= ID <lista_corchetes_valores>

<asignacion_array>::= ID <lista_corchetes_valores> ASIGNACION
<expresion>