Manual Técnico – Analizador de Código

Descripción General

Este servicio es una API REST desarrollada en ASP.NET Core, que expone un endpoint POST /analizar. Al recibir código fuente como texto, ejecuta análisis léxico, sintáctico y semántico utilizando ANTLR4. Si no hay errores, genera un archivo .s con código ensamblador ARM64.

Este modulo forma parte de un compilador escrito en C# utilizando ANTLR4 y genera codigo en ensamblador para arquitectura ARM64. El archivo contiene dos clases principales:

- CodigoARM64: Representa una unidad de codigo ensamblador generada por una expresion o instruccion.

- VisitorARM64: Es el visitor principal que recorre el AST generado por ANTLR y produce el codigo ensamblador ARM64.

Dependencias y Referencias

- Microsoft.AspNetCore.Builder, Microsoft.AspNetCore.Http: Middleware y manejo HTTP.

- Microsoft.Extensions.Hosting: Configuración y hosting de la app.

- Antlr4.Runtime: Herramientas de análisis léxico y sintáctico.

- Proyecto_1: Espacio de nombres principal.

- Proyecto_1.AnalizadorLexico: Gramática generada por ANTLR.

- Proyecto_1.Excepciones: Listeners de errores personalizados.

- Visitor: Clase personalizada para análisis semántico.

- VisitorARM64: Clase para generación de código ARM64.

Configuración Inicial del Servidor

Se inicializa la aplicación web.

Se configura CORS para permitir solicitudes desde cualquier origen (útil para React frontend).

Middleware y Rutas

Habilita CORS antes de definir las rutas.

Define una ruta POST para analizar el código fuente.

Lógica de Análisis

Paso a paso del procesamiento:

1. Lectura del cuerpo del request.

2. Análisis léxico con listeners personalizados.

3. Análisis sintáctico con listeners personalizados.

4. Generación del árbol de análisis (AST).

5. Análisis semántico y recopilación de errores/tabla de símbolos.

6. Validación de errores y generación del archivo .s si todo está correcto.

Respuesta de la API

Si el análisis es exitoso:

```
{
  "resultado": [ "...mensajes y errores..." ],
  "ast": "(inicio (...))",
  "simbolos": [ "...tabla de símbolos..." ]
}
```

En caso de error:

```
{
  "error": "Mensaje de excepción"
}
```

Archivos de Salida

Directorio de salida: ./output/

Archivo generado: programa.s

Consideraciones

Este backend se integra con un frontend en React.

El archivo generado puede compilarse con aarch64-linux-gnu-gcc.

Estructura de CodigoARM64

Esta clase se utiliza para encapsular los resultados intermedios de cada nodo del AST. Contiene:

- Texto: Codigo ensamblador generado.

- RegistroResultado: Registro donde queda el resultado de la operacion.

- Tipo: Tipo de dato (int, float64, string, bool, rune, slice<T>).

- MensajeLiteral: Representacion en texto (solo para strings, bools y runes).

- EtiquetaMensaje: Etiqueta en .data usada para imprimir mensajes.

- ResultadoFloat: Valor numerico para evaluaciones constantes.

- EtiquetaLongitud, RegistroResultadoExtra: Apoyan el manejo de slices y strings.

Estructura de VisitorARM64

Hereda de AnalizadorLexicoBaseVisitor<object> y redefine los nodos para generar ensamblador. Maneja tres secciones:

- textSection: Instrucciones de ejecucion.

- dataSection: Literales y estructuras de datos.

- funcSection: Declaracion de funciones.

Utiliza StringBuilder y contadores internos para generar etiquetas, manejar registros temporales y bucles.

Funciones Clave

1. VisitPrint: Genera el codigo para imprimir variables de cualquier tipo. Maneja tipos primitivos y slice<>, imprimiendo cada elemento con formato.

2. VisitDeclaracionVar y VisitAsignacionVar: Permiten declarar y asignar variables. Se reserva espacio en .data y se generan instrucciones mov, str, ldr segun el tipo.

3. VisitAppend: Agrega elementos a un slice. Reserva nueva memoria, copia los valores anteriores, añade los nuevos y actualiza punteros.

4. VisitInstruccion_if, VisitInstruccion_switch, VisitFor*: Implementan estructuras de control. Se generan etiquetas y saltos condicionales (b, cmp, bne, beq, blt, etc.).

5. VisitSumaYresta, VisitMultiplicacionYdivision: Calcula expresiones aritmeticas. Convierte tipos si es necesario y usa instrucciones como add, sub, mul, udiv, fmul, fadd.

6. VisitStrconvAtoi, VisitStrconvParseFloat: Permiten conversiones de string a numeros, usando funciones auxiliares como atoi y parse_float.

7. GetCodigo: Concatena todas las secciones (.data, .text, funciones) para producir el archivo final programa.s.

Consideraciones Especiales

- Todos los registros temporales se manejan desde x9 a x30 para enteros y d9 a d30 para flotantes.

- Si se accede a un slice con un indice invalido, se invoca una rutina acceso_fuera_rango.

- Se define una rutina print_* por cada tipo soportado (int, float, bool, string, rune).