Documentación

Cómo ejecutar y compilar. El compilador se utiliza desde la carpeta src llamando al archivo `coolc.sh` y a continuación la dirección del archivo de extensión cl que se desea compilar, ejemplo: `./cool.sh /path/file.cl`. Este procedimiento compila su `file.cl` y devuelve un archivo `file.mips` que se ejecuta utilizando el comando `spim -file file.mips`.‘

Requisitos adicionales, dependencias, configuración, etc.

- Instalación de la biblioteca `SLY` de python. Se puede instalar mediante el comando `pip install sly`.
- Instalación del simulador de MIPS `SPIM`. Se puede instalar mediante el comando `sudo apt-get install spim`.

Arquitectura del compilador

Todo el código del compilador se encuentra en la carpeta COOL dentro de src, este se divide en 4 módulos principales:

- coollexer: Se encarga de analizar el código fuente y dividirlo en tokens.
- coolparser: Se encarga de analizar la estructura del código fuente y devolver un árbol de sintaxis abstracta (AST) que recoja la estructura del programa.
- semantic: Se encarga de realizar el chequeo semántico del AST recibido del parser y hacer la verificación de la correctitud de tipos.
- codegen: Luego de realizarse el chequeo semántico a partir del AST se genera el código de bajo nivel mips para ser ejecutado.

Además se encuentran otros módulos auxiliares encargados de los errores en tiempo de compilación y de los diferentes nodos que puede tener el AST.

Fases del compilador

El flujo de las fases de un compilador sigue un proceso secuencial que transforma el código fuente de un programa en un programa ejecutable. Nuestro compilador de COOL sigue el siguiente flujo de fases:

Análisis léxico Esta fase se encarga de analizar el código fuente y dividirlo en unidades léxicas o tokens, como identificadores, palabras clave, operadores y símbolos. Se generan los tokens que representan las unidades léxicas del programa.

Para la lexemización, tokenización y parser se utilizó la biblioteca de python `SLY`. Esta es una biblioteca para escribir analizadores léxicos y gramaticales. Se basa libremente en las herramientas tradicionales de construcción de compiladores lex (tokenizar) y yacc (yet another compiler-compiler). Tomando su clase Lexer como base hemos creado nuestro lexer para el lenguaje COOL agregando todos

los tokens necesarios para el lenguaje, así como literales,palabras claves y algunas funciones necesarias como los ignore para los comentarios.

Además de tokenizar la entrada el análisis léxico puede detectar tokens que no pertenecen al léxico del lenguaje de programación, como caracteres no válidos o secuencias que no tienen significado dentro del lenguaje, el uso incorrecto de operadores o símbolos, que podrían indicar algún problema en la escritura del código, puede identificar si se han escrito incorrectamente palabras clave o identificadores, también detecta errores relacionados con la delimitación de tokens, como la falta de cierre de comillas en cadenas de texto, paréntesis sin emparejar, corchetes o llaves mal balanceados, entre otros.

Análisis sintáctico El analizador sintáctico verifica la estructura del código fuente según las reglas gramaticales del lenguaje de programación COOL. Se construye un AST que representa la estructura jerárquica del programa. Este árbol se utiliza para analizar la corrección sintáctica verificando si las expresiones y declaraciones del programa cumplen con las reglas de la gramática.

Para la obtención del AST se utilizaron las reglas gramaticales definidas en el manual de COOL, incluída la precedencia y asociatividad, junto al algoritmo de análisis sintáctico (parser) LALR(1) implementado en `SLY`. Un analizador LALR (Look-Ahead LR) es una versión simplificada de un analizador LR canónico, para analizar un texto de acuerdo con un conjunto de reglas de producción especificadas por una gramática formal para un lenguaje.

Al igual que con otros tipos de gramáticas LR, un analizador o gramática LALR es bastante eficiente para encontrar el único análisis de abajo hacia arriba correcto en un solo escaneo de izquierda a derecha sobre el flujo de entrada, porque no necesita usar el retroceso. El analizador siempre utiliza una búsqueda anticipada, representando LALR(1) una búsqueda anticipada de un token.

El análisis sintáctico verifica si las construcciones del código cumplen con la gramática del lenguaje de programación. Puede identificar errores como uso incorrecto de operadores, expresiones mal formadas, estructuras de control incompletas, entre otros.

Análisis semántico Durante esta fase, se realiza un análisis más profundo del programa para verificar la coherencia y consistencia semántica. Se comprueba si las variables están correctamente declaradas, si los tipos de datos son compatibles y si se cumplen las reglas semánticas del lenguaje.

En el análisis semántico del lenguaje Cool (Classroom Object-Oriented Language), se realizan diversas tareas para verificar la coherencia y corrección del programa en términos de su significado y contexto.

El análisis semántico está implementado utilizando el patrón visitor en dos momentos, primero a la hora de la declaración de las clases y herencias, donde verifica que no existan errores de herencia, conflictos de nombres, que no se creen

herencias cíclicas, que no se redefinan atributos y que los métodos se redefinan de forma correcta.

En segundo lugar, se realiza el análisis semántico de las expresiones, donde se verifica que los tipos de las expresiones sean correctos, la consistencia de los tipos utilizados en las diferentes expresiones, que los tipos de los parámetros de los métodos sean correctos, que las variables estén declaradas y siempre se utilicen en su ámbito correspondiente, que no se realicen operaciones entre tipos incompatibles.

Esta fase es crucial para garantizar que el programa cumpla con las reglas y restricciones del lenguaje, verificando la corrección de tipos, la coherencia en la herencia, entre otros aspectos fundamentales para el funcionamiento adecuado del programa.

Generación de código En esta última fase, se genera el código objeto o ejecutable final para la plataforma de destino específica. El compilador traduce el código intermedio optimizado a instrucciones de máquina comprensibles por el hardware objetivo, realizando asignaciones de registros, generando instrucciones de ensamblador y resolviendo referencias a memoria.

Durante la generación de código de Cool a MIPS, se deben considerar diversos aspectos específicos de la arquitectura MIPS, como la gestión de la pila, la asignación de registros, las convenciones de llamada a funciones, el manejo de excepciones, entre otros. Además, es crucial asegurarse de que la traducción de las construcciones de Cool a instrucciones de MIPS sea coherente y respete las restricciones y reglas de ambas plataformas.

Todos los detalles acerca de las reglas de gramática utilizada se puede ver en `parser.out`, además de visualizar cada uno de los estados de la ejecución actual.