Informe de Complementos de Compilación

Datos Generales

Autores

- Luis Alejandro Lara Rojas
- Carlos Alejandro Arrieta Montes de Oca

Ejecución

Para ejecutar el proyecto se necesita tener instalado `Python` y el conjunto de dependencias listado en requirements.txt.

Para instalar las dependencias desde la dirección `<project-dir>/src`, puede utilizar:

```
make install
```

Una vez estén instaladas las dependencias, puede compilar y ejecutar cualquier archivo de código cool utilizando el comando:

```
make main <code-file>.cl
```

Sobre el proyecto

Para el diseño e implementación de este proyecto se toma como base los contenidos adquiridos durante tercer año de análisis lexicográfico, sintáctico y semántico de programas en lenguaje COOL.

Este proyecto está basado en gran parte en las implementaciones hechas durante tercer año del Lexer, Parser y Semantic Checker.

Estructura del proyecto

Como se puede ver en main.py el proyecto está estructurado en 4 componentes principales:

1. Lexer
2. Parser
3. Semantic Checker
4. Code Generator

Cada una de estas componentes será explicada con mayor detenimiento a continuación.

Lexer

Para esta componente se utilizó el paquete de Python PLY lex.

Basicamente nos auxiliamos de todo el poder que nos brinda PLY y las expresiones regulares, lo que nos causó mayor problema fueron los comentarios y los strings, en cuyos casos se utilizaron expresiones regulares para detectar el comienzo, y se usaron algoritmos creados por nosotros para formar el string o ignorar el comentario hasta su fin, siempre manteniendo actualizado el número de línea para poder brindar mejores mensajes de error y poder pasar los tests

Parser

Para el parser se ha utilizado el paquete de Python PLY yacc.

Usamos la facilidad que nos brinda esta biblioteca, y usando una gramática generamos un ast de Cool

Semantic Checker

Esta componente se ha dividido a su vez en 3 componentes principales: - Type Collector: encargado de recolectar los tipos. - Type Builder: encargado de construir los tipos. - Type Checker: encargado de hacer chequeo de tipos.

Cada una de estas sub-componentes serán explicadas a continuación.

Type Collector

Durante la recolección de tipos se visitan todas las declaraciones de clases, se crean los tipos asociados a ellas y se valida la correctitud de las mismas.

Errores detectados: - Herencia cíclica - Redefinición de clases - Nombres de clase no válidos

Type builder

A los tipos creados en la fase anterior se le añaden todos sus atributos y métodos.

Errores detectados: - Mal uso de herencia - Uso de tipos no definidos - Problemas de nombrado de atributos y métodos - Redefinición de atributos - Redefinición

incorrecta de métodos

Type checker

En esta fase se evalúa la correctitud de todas las expresiones del lenguaje y se decide el tipo estático de cada una de ellas.

Errores detectados: - Incompatibilidad de tipos - Uso de tipos no definidos - Uso de variables, tipos y métodos no definidos - mal usos de `self` y `SELF_TYPE` - mal usos del `case`

Code Generator

Esta componente a su vez está constituida por 2 sub-componentes: - COOL-CIL Converter: encargado de convertir el ast de Cool a un ast de CIL - CIL-MIPS Converter: encargado de convertir ast de CIL ast de MIPS.

COOL-CIL Converter

En esta etapa del proceso de compilación, requirió especial atención la generación de las expresiones *case*. Para ello se requiere ordenar las instrucciones de tal modo que se asegure el emparejamiento del tipo de la expresión principal con el tipo más específico declarado en las ramas del *case*.

Errores detectados: - Dispatch estático o dinámico desde un objeto void - Ejecución de un *case* sin que ocurra algún emparejamiento con alguna rama. - División por cero - Substring fuera de rango

> Aunque estos errores realmente se detectan en ejecución, es en esta fase que se genera el código que permite detectarlos.

CIL-MIPS Converter

Para esta fase se ejecuta un recorrido por los nodos de CIL, creando las instrucciones equivalentes en MIPS. Como apoyo a la hora de generar las instrucciones en MIPS, se utilizó

El código para llamar las funciones y crearlas tiene estrecha relación con el diseño del Registro de Activación.

Las principales ideas durante la implementación del Registro de Activación son: - El resultado siempre se guarda en el acumulador. - Los parámetros son guardados en el Registro de Activación y son introducidos antes de llamar en orden inverso. - Es necesario guardar la dirección de retorno. - Se utiliza el $fp como guía para obtener tanto los parámetros como las variables locales, por lo que conviene guardarlo antes de llamar y recuperarlo al volver. - El cuerpo es quien introduce las variables locales. - El iésimo parámetro está en $fp + 4(i + 1) y la iésima variable local está en $fp -4(i + 1).

En este caso se asume que los registros están sucios, entonces no hay necesidad de guardarlos y recuperarlos antes y después de los llamados, lo cual mantiene simple el código, y podría ser hasta más eficiente.

La representación de los objetos se implementó siguiendo las siguientes ideas: - Los objetos se guardan en memoria de forma contigua. - Cada atributo está ubicado en el mismo compartimento del objeto. - Cuando se llama a un método el objeto es self. - El id de clase es la dirección asociada al string que contiene su nombre. - El tamaño es el entero que representa la cantidad de word que componen el objeto. - Se tiene un Dispatch Ptr, que representa un puntero a una tabla de métodos. - Si X hereda de Y, la representación de X es igual a la de Y, excepto que se adicionan nuevos espacios para el resto de atributos. - Los atributos están de forma consecutiva. - Cada atributo está ubicado en el mismo corrimiento para todas las subclases.

A cada clase se le asigna un puntero a una dirección de memoria que indexa todos sus métodos(Dispatch Ptr), incluyendo los heredados. De esta forma se logra que los métodos y los atributos de una clase A y todas sus subclases, se encuentren en los mismos compartimentos.