Compilación

Proyecto Cool Compiler 2021-2022:

Integrantes:

-Richard García De la Osa C-412. richard.garcia@estudiantes.matcom.uh.cu.

-Andy A. Castañeda Guerra C-412. andy.castaneda@estudiantes.matcom.uh.cu.

Uso del compilador:

Para usar nuestro compilador deberá ejecutar en la consola el siguiente comando:

```
bash coolc.sh $INPUT_FILE
```

El archivo **INPUT_FILE** debe tener extension .cl, correspondiente a un programa del lenguaje Cool.

Este comando generará un fichero con igual nombre pero con extensión .mips, el cual estará listo para ser ejecutado. Para ejecutar el **INPUT_FILE** (el cual debe tener extensión .mips) deberá usar el siguiente comando en la consola:

```
spim -file $INPUT_FILE
```

No hay otros parámetros adicionales para ejecutar nuestro compilador.

Arquitectura del compilador:

El proyecto tiene la siguiente estructura:

Módulos:

-lexer: Este módulo está compuesto por **lexer.py**. En este archivo se encuentran las reglas lexicográficas para tokenizar un programa (texto) escrito el **COOL**. Se utiliza la biblioteca **PLY** de python.

-c_parser: Este módulo está compuesto por **parser.py** y **parsetab.py**. En el archivo **parser.py** están escritas las producciones de la gramática utilizada por nosotros para generar un **AST** de **COOL** estructurado a nuestra conveniencia. Por otra parte **parsetab.py** es un archivo generado por la biblioteca **PLY** al ejecutar el parser en conjunto con el **lexer.py** antes mencionado. Este contiene especificaciones sobre el método de parsing utilizado así como los tokens asociados a sus producciones.

-cool_ast: En este módulo se encuentra el archivo **cool_ast.py** el cual contiene los nodos del **AST** de **COOL** que consideramos necesarios para la implementación de nuestro compilador.

-cil_ast: En este módulo se encuentra el archivo **cil_ast.py** el cual contiene los nodos del **AST** del código intermedio generado (**CIL**) que utilizamos para traducir el código de **COOL** a instrucciones sencillas de forma que sea más fácil transformarlo a código **mips**.

-utils: Compuesto por **errors.py**, este contiene el formato de los errores lexicográficos, sintácticos y semánticos que empleamos durante los recorridos al **AST**. Luego encontramos **mip_utils** donde podemos encontrar 3 clases que son utilizadas durante la conversión de **CIL** a **mips** para acceder de manera más sencilla a los strings de los operadores, registros y tipos de datos en **mips**. El archivo **semantic.py** es uno de los módulos obtenidos durante el desarrollo de las clases prácticas, con algunas modificaciones.

-visitors:

-collector.py: contiene el visitor utilzado solo para registrar los tipos del programa en un objeto context el cual es importado de **semantic.py**. Este visitor solo recorre los nodos que representan declaraciones de clases.

-builder.py: primeramente definimos los tipos básicos de **COOL** así como sus funciones built_in. Además visitamos todos los tipos ya recogidos en el colector para registrar sus métodos y atributos (features). Finalmente nos construimos el grafo de herencia de los tipos para asegurarnos de que no existan ciclos en este.

-checker.py: revisamos que el programa esté semánticamente correcto. Es decir, una vez nuestro programa es sintácticamente válido corresponde asegurarnos que los tipos de las variables y funciones estén en correspondencia con el contexto en el que se usan. También anotamos a cada nodo su tipo estático para posterior uso durante la generación de código intermedio. Cada nodo existe en un ámbito específico en el cual son visibles variables locales y de ámbitos más externos; cada vez que se visita un nodo se crea un ámbito nuevo en el cual se registren sus variables locales para aquellos casos en los que unas variables solapan a otras con el mismo lexema pero en ámbitos padres.

-CooltoCill.py: aca realizamos otro recorrido sobre el **AST** de **COOL** con el cual traducimos las instrucciones de alto nivel a instrucciones que simplifiquen el proceso de generar código mips. Nuestro **CIL** está compuesto por tres secciones: data, donde se registran todas aquellas cadenas de caracteres ya sea especificados por el usuario o por nosotros para lanzar errores en ejecución; type, compuesta por estructuras que agrupan todos los métodos y atributos de cada tipo (features), para luego en la traducción a mips poder organizar de manera más fluida el direccionamiento a estos; por último code, en este se encuentran agrupadas en estructuras el código intermedio generado para cada una de las funciones de los tipos del programa (contructores, built in y definidas por el usuario). Durante el recorrido por los nodos del **AST** de **COOL** se genera a la par un nuevo **AST** esta vez de **CIL** para luego traducir a **mips**.

-CiltoMips.py: se utiliza el **AST** de **CIL** generado por el recorrido anterior para traducir cada nodo a una secuencia de instrucciones en **mips** correspondientes al mismo. Primero se visitan los nodos que se encuentran en la sección de data luego la sección de type y por último la sección de code y de esta forma queda organizado el programa en **mips** añadiendole al final algunas funciones por defecto como 'alocate' para reservar memoria y otras.

Problemas técnicos:

El mayor desafío de implementación que encontramos fue la correcta generación a código intermedio para su posterior traducción a código **mips** de los nodos **switchcase** del **AST** de **COOL**.

Para ello organizamos las ramas del nodo de mayor a menor, donde las mayores son aquellas cuyo tipo se encuentra a mayor profundidad en el árbol de herencia del programa. Por cada una de estas ramas recorremos el subárbol cuya raiz es el tipo actual de la rama, registrando un salto al label asociado a esta rama. Al recorrer cada uno de estos subárboles, si coincide en algún tipo de este, podemos asegurar que este es el menor tipo que lo conforma (dado que recorremos las ramas en orden decreciente en profundida).