Reporte del Proyecto de Complementos de Compilación

Uso del Compilador

Para ejecutar el compilador se deberá correr en la terminal el comando:

```
$ python coolc.py <archivo.cl> <archivo.mips>
```

Adicionalmente se le pueden adicionar al compilador diferentes argumentos opcionales. Estos argumentos son los siguientes:

- `--cil`: Este argumento indica al compilador que se desea generar un archivo con el código `CIL` generado en las etapas intermedias del proceso de compilación.
- `-v` o `--verbose`: Estos argumentos opcionales indican al compilador que se desea ejecutar el mismo en modo verboso. El modo verboso agregará a la salida estándar del compilador información adicional. La estructura del ast de `COOL` generado y datos sobre los tipos definidos en el código son parte de esta información adicional.
- `-h` o `--help`: Estos argumentos mostrarán una ayuda mínima referente al uso del comando, así como de los demás argumentos opcionales del mismo.

Arquitectura del Compilador

El compilador cuenta con un módulo principal llamado `cmp` en el cual se encuentra todo el código para usar el mismo. A su vez el módulo principal está dividido en dos submódulos `cool_lang` y `cil`.

COOL

El submódulo `cool_lang` contiene las herramientas para la realización de las fases de análisis lexicográfico, análisis sintáctico y análisis semántico del código escrito en `COOL`. Como resultado de la utilización de las herramientas de este módulo se puede obtener del código un AST válido.

Las principales clases para realizar estos procesos son:

- `COOL_LEXER`: Esta clase es la encargada de la tokenización del código escrito en el archivo de entrada y notificar los posibles errores léxicos en los mismos.

- `COOL_PARSER`: Esta clase permite a partir del código tokenizado construir el AST de `COOL` y notificar los posibles errores sintácticos.
- `COOL_CHECKER`: Esta clase es la encargada de realizar los diferentes recorridos al AST de para verificar que este correcto y de notificar de los errores encontrados. En adición a la verificación de errores sintácticos, se producirá también una clase `Context` con todos los datos necesarios acerca de los tipos definidos en el código.

CIL

El submódulo `cil` contiene las herramientas necesarias para obtener un código de `MIPS` a partir de un AST correctamente verificado. Como pasos intermedios se genera un AST de `CIL`, siendo posible obtener también un código en este lenguaje.

Para esta tarea las principales clases usadas son:

- `COOL_TO_CIL_VISITOR`: Esta clase es la encargada de transformar el AST de `COOL` en un AST de `CIL`. Ya en esta fase se asume que los errores en la entrada no existen y que el AST de `COOL` es uno correcto.
- `CIL_TO_MIPS`: Esta clase permite generar de un AST de `CIL` el código `MIPS` equivalente al mismo.

Problemas técnicos

Como se puede observar por lo anteriormente descrito el proceso de compilación ocurrió en cinco fases, tres de ellas enfocadas en la validación y corrección de errores en la entrada y dos de ellas en la generación de código `MIPS`.

A continuación se enumeran algunas de las decisiones tomadas durante las distintas fases del proceso.

Las fases de análisis léxico y sintáctico se completaron de manera normal, haciendo uso de las definiciones de tokens dadas por el Manual de COOL. Sin embargo, la gramática de `COOL` definida en el Manual es ambigua, y por tanto, requirió que su transformación a una gramática que no lo fuera. Para estas tareas se utilizó el módulo de Python `ply`. Esta herramienta se utilizó para definir las expresiones regulares para los tokens y para la definición de la gramática LR equivalente a la de `COOL`.

De igual forma durante la fase de análisis semántico se utilizaron tres recorridos para realizar la validación del AST construido.

El primero de los recorridos fue el encargado de recolectar todos los tipos definidos. Al finalizar la recolección se verificó que no existiera herencia cíclica en la jerarquía

de tipos. Una vez que se verificó esto se tomó la decisión de ordenar los nodos del AST que representan a las definiciones de clases de forma que en los siguientes recorridos si se visita una definición de un tipo su tipo padre ya habrá sido visitado. Esto se puede lograr ya que la jerarquía de CIL está unificada en la clase `Object`.

El segundo de los recorridos fue el encargado de construir los tipos recolectados en el primero. En este segundo recorrido fue donde se agregó la información referentes a los tipos básicos que trae el lenguaje y que el programador puede usar sin haberlos definido. Esta información, si bien carecía de estructuras en el AST, garantiza la correcta evaluación futura de las reglas de tipos.

El tercer recorrido fue el encargado propiamente de realizar el análisis semántico del AST. Esto fue posible de hacer gracias a la información obtenida en los recorridos anteriores. Este análisis se realizó siguiendo las reglas definidas en el Manual.

Una vez realizado el análisis semántico se pasa a la generación de código. Como primer paso para esto se definió el lenguaje intermedio `CIL` siendo usado como base para el mismo algunas de las ideas dadas en el curso anterior. Si bien muchas definiciones son compartidas si se agregaron muchos nuevos nodos para expresar diferentes acciones y otros fueron transformados. Es objetivo de `CIL` alcanzar un equilibrio entre la abstracción que brinda `COOL` y una cierta cercanía en estructura a `MIPS` que permitiera luego una mejor y más sencilla transformación hacia `MIPS`.

Del AST obtenido en las fases anteriores se realizó una transformación hacia un AST de `CIL` que expresara el mismo comportamiento que el construido en CIL.

En esta fase se decidió también tratar a todos los tipos como objetos por referencia. Siendo entonces los objetos por valor de `Int` y `Bool` ahora objetos con una propiedad `value`. El tipo `String` pasó también a cumplir con esta estructura. Esto facilitaba el uso de las funciones heredadas de `Object`. Si bien pareciera que se corre el riesgo de que estos objetos pierda el comportamiento que tenían en `COOL` al ser por valor, esto no ocurre ya que las operaciones que los modifican solo son generadas para `CIL` y por tanto no accesibles desde `COOL`.

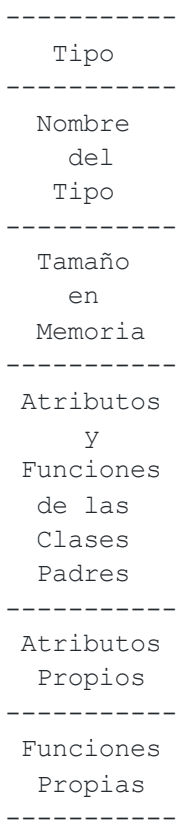
Es también en esta fase donde se da por fin una implementación real a los tipos y funciones básicas definidas por el lenguaje. Estas hacen uso de nodos especiales del AST que solo son instanciadas para su uso desde `COOL` a través de las funciones básicas. Ejemplo son los métodos para la entrada y salida de datos brindada por la clase `IO`. Con estas estructuras definidas en el AST de `CIL`, durante la traducción a `MIPS`, ellas serán generadas automáticamente.

Un caso notable en esta transformación es el de las expresiones del tipo `case`. La naturaleza de esta expresión implica que sea posible conocer cuando el tipo dinámico de una expresión es subtipo de otro. Debido a la complejidad de realizar esta operación tanto en `CIL` como en `MIPS` se decidió realizar una transformación al

AST de las expresiones `case` durante la traducción. Para esto se agregaron nuevos nodos de forma que la evaluación del `case` cambiará. La idea para esto fue agregar todos los subtipos de todas las clases usadas en el `case` al mismo y que tuvieran como cuerpo la misma expresión que tiene el padre más cercano definido en el `case` original. Luego estos fueron reordenados desde el más específico hasta el más general. Esta reordenación permitió que solo hiciera falta verificar que el tipo dinámico de la expresión case fuera el mismo que el del caso analizado, y que fuera el primero que cumpliera esto el caso correcto a aplicar.

En la última fase de generación se hizo importante definir lo que sería la estructura de la instancia un tipo en memoria. Según las principales necesidades de los nodos más abstractos de `CIL`, se trató de que la información necesaria para el correcto funcionamiento de los mismos estuviera presente en cada instancia de un tipo cualquiera. Esto para eliminar la necesidad de crear cualquier estructura de un tipo y evitar realizar operaciones complejas en `MIPS`. También se buscó lograr una forma en que se pudiera cumplir con el comportamiento polimórfico de los tipos. Así como que cada instan

Según estas necesidades la representación de las instancias de los tipos en memoria puede verse de la siguiente forma:



En esta estructura los tipos tendrán siempre alcanzable y a posiciones fijas los siguientes datos:

- Un entero que identifica únicamente a las instancias de un mismo tipo. Dos instancias serán del mismo tipo si sus enteros de tipo son iguales.
- Un entero que será la referencia al inicio de un string, previamente almacenado en memoria, que contiene el nombre del tipo.
- Un entero que es el valor el tamaño de la instancia en memoria.
- Luego enteros que referencian a los atributos y funciones del tipo instanciado. Estos, para permitir el polimorfismo, se ordenarán siguiendo el orden en que fueron declarados a lo largo de la jerarquía del tipo instanciado, desde el más general hasta el más específico. Por ejemplo si tenemos los tipos `A`, `B` y `C`. Primero se encontrarán los atributos de `A`, luego las funciones de `A`, los atributos de `B`, las funciones de `B`, los atributos de `C` y por último las funciones de `C`. Cada grupo de atributos o funciones estarán ordenados según el orden en que fueron declarados en ese tipo. Si alguna función de un tipo más general es sobre escrita por algún tipo más específico está seguirá apareciendo en el conjunto del tipo más general y no en la del específico, pero con la referencia cambiada hacia la dirección de la nueva implementación. Esto permite que si un `C` es tratado como un `A` llamadas a funciones de `A` ejecutarán las funciones sobre escritas de `C`.