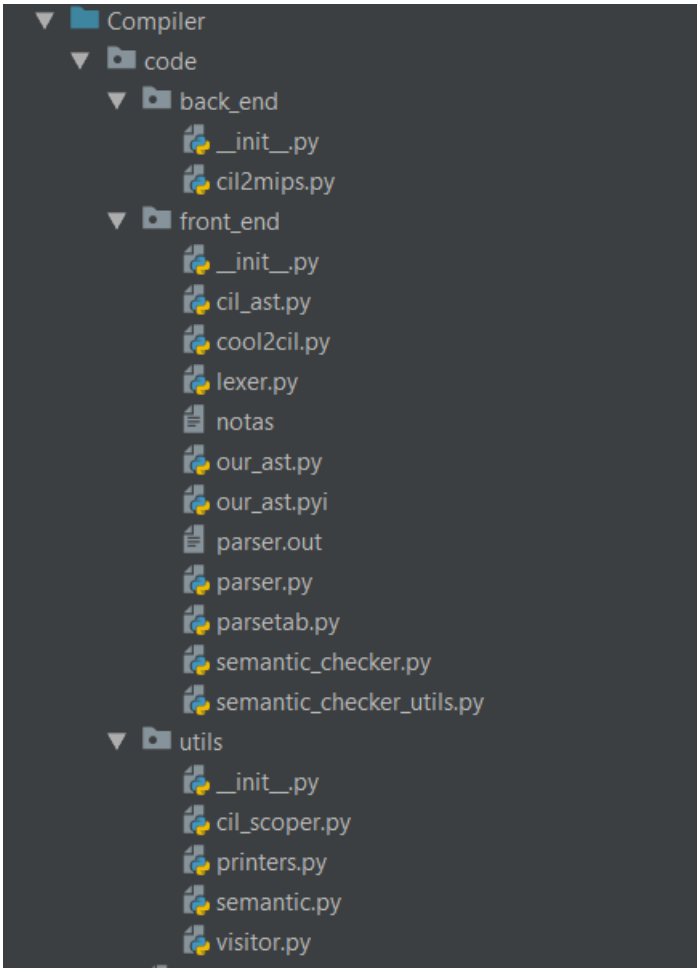
Reporte

Jessica Quesada, David Castillo, Rayniel Ramos | Complementos de Compilación | 01 de julio de 2019

Uso del compilador

Lo mismo que en el readme

Arquitectura del compilador:



Los módulos del proyecto están agrupados en carpetas intentando seguir la estructura general establecida. En la carpeta de back_end se encuentra el modulo encargado de tomar un AST en CIL y exportar, usando el patrón visitor, su código equivalente en MIPS. En la carpeta utils encontramos módulos necesarios que no se encuentran íntimamente relacionados con el proceso de compilación pero que poseen clases o funciones que apoyan este proceso, entre ellos encontramos el modulo cil_scoper.py el cual contiene la clase que ayuda, mediante una estructura de datos que funciona como un mapper, a la simulación del ocultamiento de variables en CIL, o sea, simular la creación de espacios de nombres nuevos en distintos contextos. Encontramos el modulo visitor.py el cual contiene las dos funciones necesarias para facilitar la implementación del patrón visitor por una clase con sobrecarga de métodos (funciones). Encontramos el modulo printers.py el cual contiene las clases necesarias para imprimir el AST de CIL. En la carpeta fornt_end encontramos los módulos encargados de tokenizar, parsear, realizar el análisis semántico y transformación a CIL, también encontramos la estructura de dos AST, el de COOL y el de CIL, asi como la tabla de parsing generada de forma automática por el parser.

Implementación del compilador:

El compilador cuenta con cinco fases fundamentales. La fase en la que se obtienen los tokens de la cadena, análisis léxico, la fase de análisis sintáctico, se obtiene el AST, luego la fase de análisis semántico, en donde se verifica el grafo de herencia, el chequeo de tipos, etc. posteriormente se realiza la generación de código intermedio, para esto, a partir del AST de COOL se obtiene el AST de CIL, transformando cada expresión de COOL y nodos del AST en plantillas de código en CIL, las cuales son el equivalente del código de COOL en CIL, se obtiene el nuevo AST resultante de expandir el código de COOL en estas plantillas, luego está la fase de generación del código de máquina, se realiza el mismo procedimiento anterior, se expande el código de CIL en código de MIPS, y se obtiene el código de MIPS final.

ESTRUCTURA GENERAL DE CLASES Y FUNCIONES:

Análisis léxico:

En el módulo lexer.py en la carpeta front_end es en donde tiene lugar, para esto se usa el lex de PLY, se define la lista de tokens y las palabras reservadas, las expresiones regulares para cada token y las funciones que, una vez halla macheado una expresión regular, deberán llamarse para desambiguar algunos tokens, o modificar el contenido de los mismos, el valor, posteriormente se crea un objeto de tipo lex (lexer) para obtener la lista de tokens a partir de un caso de prueba.

Análisis sintáctico:

En el módulo parser.py de la carpeta front_end se definen las reglas de asociación y prioridades (precedencia) para la gramática, posteriormente se declara la gramática atributada, para ello se crean funciones, las cuales van a recibir el sintetizado, en el docstring de las mismas se escribe la derivación correspondiente, y posteriormente como primer argumento del atributo sintetizado se retorna el valor resultante de haber aplicado la operación correspondiente a la derivación. De esta forma se construye el AST de COOL, la gramática que se utilizo es prácticamente la misma que se encuentra en el manual del lenguaje, es ambigua, pero el parser hace uso de las reglas declaradas. De esta forma se crea un objeto yacc (parser) el cual recibe un objeto lex, para, dada la cadena, obtener la lista de tokens y reducir la gramática.

Análisis semántico:

En el módulo semantic_checker.py de la carpeta front_end se declara un visitor encargado de chequear el uso correcto de los tipos, el uso de métodos y funciones que estén declaradas, el grafo de herencia, etc. y de la semántica en general.

Generación de código CIL:

En el módulo cool2cil.py de la carpeta front_end se declara un visitor que es el encargado de realizar la expansión de código en COOL a su equivalente en CIL y de construir el AST de CIL al mismo tiempo. Este visitor crea los métodos de las clases Object e IO en CIL, y las inscribe en la .code del AST creado, lo mismo sucede con los métodos de String, así como el método encargado de comparar String para comprobar si son iguales, el cual se usa en el operador de igualdad.

Generación de código MIPS:

En el módulo cil2mips.py de la carpeta back_end se encuentra declarado el visitor encargado de realizar un procedimiento similar al que se realizó de COOL para CIL, pero esta vez de CIL para MIPS.

Principales problemas encontrados y soluciones:

Encontrar una expresión regular para los string y los comentarios en el lexer, fue complicado y hubo que guiarse por las expresiones regulares de C++ que son similares.

La idea de conocer el tipo estático de una expresión de COOL en CIL, necesario para llamar a los métodos de los tipos built-in. Al tratarse los built-in String, Int, Bool, como tipos por valor, números y arrays se hizo necesario colocar if en CIL y desambiguar los casos en los que se llamaban a los métodos de estos tipos, de la misma forma hubo que desambiguar el caso de la comparación de dos String, entre otros, esto se resolvió añadiendo un atributo a los nodos del AST de COOL en la fase de chequeo semántico que tendría el tipo estático de esa expresión.

El trabajo con el tipo Void se solucionó utilizando la propiedad _begin, propiedad privada de todos los tipos (con excepción de String, Int y Bool) la cual en el caso de ser 0 significa que es Void, esto se usa en el caso de los dispatch y en el case, para evitar llamados a métodos de un tipo Void, solo se agregan las líneas en CIL que verifican que ese atributo es 0 antes de realizar ningún trabajo con el tipo.

El uso de un atributo privado para todos los tipos que guarde el nombre del tipo en un array, utilizado para el type_name.

El uso de un atributo privado que guarda una tabla virtual, utilizado para garantizar el trabajo con la herencia, en esta tabla virtual se guardan las direcciones de memoria de los métodos del tipo que la posee, es utilizada para el trabajo de los casos en los que desee asignar un tipo hijo a una variable de un tipo que es ancestro.

El uso de un atributo privado len, el cual guarda la longitud del tipo (un tipo no es más que un array donde cada atributo tiene cuatro bytes, la referencia al mismo), utilizado para implementar el método copy de object.

En el caso de los case, para darle solución, se encontró un algoritmo, sencillo y eficiente. Básicamente se ordenan los branch del case por profundidad de sus tipos en el grafo de herencia, de aquí que se pueda utilizar la propiedad begin igual a 0 para el Void y reutilizar esta propiedad, puesto que ahora, se hará DFS desde cada tipo de los branch y utilizando el tiempo de descubrimiento y de finalización, begin y end (notar que aunque begin es una propiedad de cada tipo y utilizada en tiempo de ejecución, end solo se usa en tiempo de compilación y por tanto no es necesario guardarlo, además begin comienza en 1, nunca en menos), se verifica que el begin de la expresión este entre en begin y el end de cada uno de los tipos de los branch ordenados, el tipo que cumpla con esto es aquel ancestro más cercano al tipo de la expresión del case, notar que se conoce en tiempo de compilación el begin y el end de los tipos de las branch, por lo que no es necesario obtener sus atributos, se sustituyen los números directamente en el codigo, por lo que solo se debe extraer el atributo begin del tipo de la expresión del case y verificar con if consecutivos si se cumple la condición, una vez encontrado el if se ejecuta el código de esa rama, de no ser así al final de esta plantilla de CIL se añade una instrucción para lanzar la excepción. Para evitar que la expresión sea Void, pues sencillamente, al conseguir el begin de la misma se pregunta si es 0.

Como se conoce en tiempo de compilación el nombre del atributo se sustituye el offset en ese mismo instante, no es necesario realizar este mecanismo en tiempo de ejecución.