Informe sobre el trabajo de Compilación de cuarto año de Ciencias de la Computación

Implementación en python3 de un compilador del lenguaje de programación COOL

Alejandro Díaz Roque

2021-01-25

Índice

1. Introducción

El presente informe trata sobre la implementación en `python` de un compilador del lenguaje `COOL` (Class Room Object Oriented Language), un software que se encarga de tomar ficheros escritos en `COOL`, y trasladarlos a código máquina ejecutable por un sistema operativo. El código máquina en este caso es `MIPS32`. Se especificarán las componentes en que se encuentra dividido dicho software, los detalles relevantes en la implementación de cada una, y además se dará una breve explicación de cómo instalar y ejecutar el programa.

2. División en Componentes

El compilador se encuentra dividido en 4 componentes principales:

- **Lexer**: Que se encarga del análisis lexicográfico. Aquí se definen los tokens del lenguaje COOL.
- **Parser**: Donde se define la gramática especificada en el estándar de COOL, y donde además se construye el árbol de derivación correspondiente a la cadena asociada a un fichero para dicha gramática.
- **Semantic**: Donde se definen las clases que están dentro del fichero `.cl`, se crea la jerarquía entre ellas, y además se hace chequeo de tipos para las operaciones que se ejecuten dentro del `.cl`.
- **Generation**: Donde se traslada un `.cl` a un fichero `.mips` equivalente. Para ello, se define una representación intermedia, denominada `CIL` (Class Intermediate Language), se traduce el contenido del archivo `.cl` a esa representación, y lo resultante se traslada a un archivo `.mips`, ejecutable por una arquitectura `MIPS32`.

2.1. Análisis lexicográfico

Dentro de la carpeta /src/compiler/components/lexer, se encuentra el fichero lexer_analyzer.py. En este se define todo lo necesario para construir la lista de tokens de COOL. Se usa el módulo ply.lex de python como framework, para hacer esta tarea de modo ágil.

Luego de ser construidp el lexer, al archivo de entrada se le aplica la operación de *tokenización*, que consiste en dividir el texto escrito en COOL en diferentes tokens, según la definición del lexer.

```
    # Este es el código para tokenizar un .cl
    def tokenizer(stream_input):
        global readjust_col
        readjust_col = 0
        lexer.input(stream_input)
        token_list = []
        lexer.lineno= 1
        #lexer.lexpos= 1
        real_col = {}
        for tok in lexer:
            real_col.update({ str(tok): find_column(stream_input, tok)
                })
            token_list.append(tok)


        return errors, token_list, real_col
```

Un detalle a resaltar es cómo se guarda la columna real de cada token. Por un lado se usa el diccionario real_col, que tiene como llave la representación en string de cada token y como valor el método find_column, que retorna la posición del token en la entrada. Y por el otro, se usa el sistema de seguimiento ply.lex, que pone en el atributo lexpos de cada token su número de ocurrencia, y la variable global readjust_col, para reajustar el valor de lexpos cuando hay un salto de línea. Esto es para reportar un error con la posición conveniente.

2.2. Análisis sintáctico

Tomando como punto de partida lo hecho previamente en el análisis lexicográfico, usando el módulo ply.yacc, se define la gramática de COOL y generar, y a partir de ella el árbol de derivación correspondiente para el archivo .cl de entrada. De igual manera, en este paso se obtiene el *Abstract Syntax Tree* o AST, que es el que sirve de entrada a las componentes **Semantic** y **Generation**. Para la definición del AST, usamos las clases que están definidas en el módulo /src/components/semantic/AST_definitions.py.

2.3. Análisis semántico

En esta fase, nos encargamos de tres aspectos esenciales, dado un AST de COOL:

- Chequear que todos los tipos que se usan en el AST existen.
- Chequear que las definiciones de dichos tipos, y de sus *features* tienen sentido bajo los estándares de COOL.
- Chequear que la relación de jerarquía es correcta.
- Chequear que las operaciones definidas correctamente, tengan un uso consistente.

Para ello, se recorre el AST usando el patrón *visitor*, que consiste en realizar sucesivas visitas al AST recopilando información en cada una que se usa en la próxima.

```
    class NodeVisitor:
        def __init__(self, programContext):
            self.programContext= programContext

        def visit(self, node: Node, **args):
            if isinstance(self, TypeCheckerVisitor):
                if issubclass(type(node), NodeBinaryOperation):
                    return self.visit_NodeBinaryOperation(node, **args)

            visitor_method_name = 'visit_' + node.clsname
            visitor = getattr(self, visitor_method_name, self.
                not_implemented)
            return visitor(node, **args) # Return the new context
                result from the visit

        def not_implemented(self, node: Node, **args):
            raise Exception('Not implemented visit_{} method'.format(
                node.clsname))
```

Cada recorrido del AST, es una clase que hereda de NodeVisitor. Para esta componente, se usan 4 clases, o sea, 4 recorridos, en el siguiente orden:

1. **TypeCollectorVisitor**

2. **TypeBuilderVisitor**
3. **TypeInheritanceVisitor**
4. **TypeCheckerVisitor**

2.3.1. Estructuras adicionales para los recorridos

2.3.1.1. ProgramContext En cada recorrido, se va actualizando un objeto de tipo globalContext y de nombre programContext, que se encarga de ir actualizando toda la información necesaria para

que el contexto del `AST`, tenga sentido. De esta forma, cada vez que `programContext` cambia, lo hace a través de alguno de los métodos definido en su clase asociada, y en cada método chequea, antes de la actualización, si la operación es válida. O sea, que no haya tipos redefinidos, que una operación aritmética tenga sentido, etc.

2.3.1.2. Clase error La clase `error` se usa para definir los errores que se pueden cometer en el proceso de compilación de una entrada en `COOL`, debido a errores de escritura. Se encuentra en el archivo `/src/compiler/utils/errors.py`. Para el caso del `programContext` antes mencionado, se usa un diccionario con los posibles errores en el análisis semántico, llamado `error_selector`.

2.3.1.3. Environment La variable `environment` es un diccionario que sirve para guardar cada nombre de objeto en el `AST` con su tipo correspondiente, para el scope de una expresión.

2.4. Generation Code

Luego de que se chequea que el `AST` es consistente, se vuelve a recorrer el mismo para obtener la representación equivalente en `MIPS32`. Para esto se crean dos clases, usando la misma lógica que en la componente **Semantic**:

1. **CILVisitor**: Para crear la representación en `CIL` del `AST`. Esta representación es un árbol. Los nodos del árbol están definidos en el archivo `/src/compiler/components/generation/CIL_definitions.py`
2. **MipsVisitor**: Para a partir del árbol `CIL`, obtener el archivo `.mips` correspondiente. Para traducir de uno a otro, se usan convenciones, teniendo en cuenta la función de cada registro y cómo se comportan las llamadas al sistema que hace `MIPS32`.

3. Conclusión

La implementación posee detalles corregibles. No obstante, su correctitud y robustez se corresponden con lo exigido en los test de prueba que se evalúan en el *pull request*.