Segundo Proyecto de Compilación

Intérprete de Cool

Integrantes:

- Luis Ernesto Ibarra Vázquez C-311
- Luis Enrique Dalmau Coopat C-311

1. Resumen:

El documento describe la implementación de un intérprete del lenguaje Cool haciendo hincapié en el sistema de inferencia de tipos. Presenta además el modo de uso del programa final. Cuando se quiera ver un comando en la consola se escribirá en una sola línea precedida por >> en el formato `Consolas`. Los nombres de clases y de paquetes se escribirán en **negritas**, direcciones estilo import de Python o nombres de métodos se escribirán en *cursiva*.

2. Uso:

2.1. Instalación

Instalar las dependencias de requirements.txt

```
>> python3 –m pip install –r requirements.txt
```

De forma opcional puede crear un virtual environment e instalar las dependencias en él.

2.2. Correr el intérprete

Para el fácil manejo y visualización de los resultados del programa se crearon dos módulos que actúan como interfaces:

- Consola: main.py
- Gráfica: main_st.py

2.2.1. Uso de main.py

El uso de main.py tiene la siguiente forma:

```
>> python3 main.py DIRECCIÓN_DEL_PROGRAMA_A_EJECUTAR [-h] [-v] [-i] [-ucp] [-ucl] [-ump] [-uml]
```

- -h: Muestra un mensaje de ayuda para el uso del programa

- -v: Flag que especifica si el procedimiento debe tener una salida verbose
- -i: Flag que especifica si se tiene que generar un archivo con un programa equivalente al entrado pero con los tipos inferidos ya resueltos
- -ucl y -ucp: Flags que especifican si se tiene que modificar el lexer y parser, respectivamente, serializados del lenguaje Cool.
- -uml y -ump: Flags que especifican si se tiene que modificar el lexer y parser, respectivamente, serializados del lenguaje creado para eliminar los comentarios anidados.

Como ejemplo integrado al paquete se encuentra el archivo ./cool_programs/program1.cl, para ejecutarlo sería:

```
>> python3 main.py cool_programs/program1.cl –v –i
```

2.2.2. Uso de main_st.py

El uso de este programa requiere tener instalado el paquete **streamlit**. Una vez instalado correr:

```
>> streamlit run main_st.py
```

Luego abrir en algún navegador la dirección especificada en la consola. Una vez ahí, copie o teclee un programa de Cool en el cuadro de texto y oprima el botón para ejecutar el intérprete y ver su salida.

3. Intérprete:

3.1. Paquetes asociados:

El proyecto presenta los siguientes paquetes:

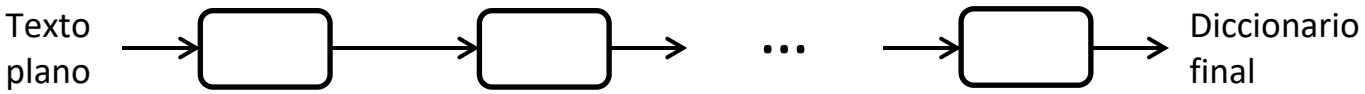
- **cmp**
  - Es el dado en Clase Práctica y contiene las definiciones de los tipos básicos usados en el proceso de parsing (**Grammar**, **Token**, **State**, etc). Además contiene implementaciones básicas para el proceso de análisis semántico (**Scope**, **Context**, **Type**, etc).
- **lib**
  - Resultado del Proyecto I de Compilación el cual provee al intérprete de implementaciones de parsers y lexers para el análisis de los programas.
- **cool**
  - Paquete principal del proyecto en el cual se encuentra la implementación del intérprete. Se divide en varias carpetas teniendo en cuenta los nombres de estas para proveer más información sobre su contenido.
- **test**

- Contiene los test usados para probar el intérprete. Usa **pytest**.
- Para ejecutar los test escriba uno de estos commandos en la consola

```
>> pytest
>> python3 -m pytest
```

3.2. Diseño:

El intérprete está diseñado para construir el resultado final mediante la concatenación de resultados de distintos procesos, esta concatenación se puede ver como un flujo en el que la información entra, se modifica y luego la salida es usada como entrada al siguiente proceso.



Para el modelado de lo anterior se crearon las clases **Pipe** y **Pipeline**. Estas no son más que objetos *callable* que reciben como primer argumento un diccionario y devuelven el mismo diccionario con las modificaciones pertinentes, gracias a esto se logra un flujo desacoplado y fácilmente modificable, aportando gran flexibilidad al programa (Vea *cool.pipeline* y *cool.pipes*).

Para el análisis de los AST se usó el patrón Visitor (Vea *cool.visitors.visitors*).

3.3. Etapas del proceso de interpretado

El flujo del programa se divide principalmente en las siguientes etapas:

1. Análisis Lexicográfico
2. Análisis Sintáctico
3. Análisis Semántico
4. Ejecución

3.3.1. Análisis Lexicográfico y Sintáctico

El objetivo principal de estas etapas es tokenizar el texto plano y generar la derivación de la cadena para posterior análisis. En el proceso se recopilan errores y se hacen operaciones secundarias para satisfacer las especificaciones del lenguaje, como por ejemplo escapar los caracteres especiales en los string de Cool (Vea *cool.pipeline.lexer_syntax_pipeline*).

En este participa el lexer creado para Cool, además de la gramática atributada generada para este. Se usó un parser LALR1 para obtener la derivación de las cadenas. Los parsers y lexers fueron serializados, ya que generarlos cada vez que se corre el programa consume una cantidad de tiempo no deseada (Vea *cool.lexer*, *cool.parser*, *cool.grammar*).

3.3.2. Análisis Semántico

El objetivo de esta etapa es crear el AST partiendo de la derivación de la cadena y comenzar el proceso de inicialización y llenado de las estructuras necesarias para el correcto funcionamiento y detección de errores semánticos en el programa. Entre las estructuras creadas se encuentran **Context**, **Scope** y **Operator**. La clase **Context** es la encargada de guardar toda la información referente a los tipos en un programa de Cool, **Scope** se encarga de guardar información sobre las variables declaradas y por último el **Operator** mantiene información sobre las operaciones que se pueden realizar sobre distintos tipos.

El flujo de esta etapa se puede ver en *cool.pipeline.semantic_pipeline*. En esta se hace fuerte uso del patrón Visitor en los diferentes recorridos del AST. Los recorridos de la primera parte son:

- **TypeCollector**: Se encarga de inicializar la clase **Context** llenándola con los tipos estándar (Object, Int, etc) y los declarados en el programa.
- **TypeBuilder**: Se encarga de completar los tipos previamente recolectados añadiéndoles sus atributos y métodos.
- **TypeChecker**: Verifica que las operaciones entre tipos sean correctas y construye el **Scope** para verificar que todas las variables usadas estén declaradas.

Además una importante función que se realiza en esta fase es la inferencia de tipos mediante el tipo especial AUTO_TYPE.

3.3.2.1. Algoritmo de inferencia

El algoritmo propuesto para la inferencia de los tipos AUTO_TYPE se basa en la idea de ir descartando los tipos que no pueden ser, basándose en las operaciones realizadas sobre los AUTO_TYPE y las operaciones válidas sobre los posibles tipos. Estas operaciones pueden ser asignación, operaciones aritméticas y despachado de métodos. Luego que se tiene toda la información necesaria, se comprueba que no haya incoherencias en los posibles tipos resultantes (Ej: métodos con igual nombre y parámetros en clases no relacionadas) y se sustituye en los nodos del AST, el AUTO_TYPE, por el tipo el más adecuado según su posición (Ej: argumentos el más abstracto, en tipo de retorno el más concreto).

La primera parte del algoritmo consiste en reunir toda la información sobre las operaciones que actúan sobre los AUTO_TYPE, esto se hace de manera automática en el chequeo de tipos mediante la llamada a los métodos *conforms_to*, *get_attribute* y *get_method* de la clase **AutoType** y *operation_defined* de **Operator**. Estos métodos son llamados en el recorrido **TypeChecker** para comprobar que las operaciones a realizar sobre los tipos cumplen las reglas semánticas. El algoritmo se aprovecha de esto y elimina en esta etapa los tipos que no satisfacen las operaciones (Vea *cool.semantic.type.AutoType* y *cool.semantic.operations.Operator.operation_defined*).

La segunda parte consiste en verificar la validez de los posibles tipos. Definimos a un AUTO_TYPE como válido si el grafo de sus posibles tipos (tomando la relación *A padre de B*) es unilateralmente conexo y no vacío. Esta definición se debe a que en caso de no ser unilateralmente conexo, significaría que habría dos sub herencias disjuntas que podrían ocupar el lugar del AUTO_TYPE y en este caso no se podría decidir entre una de estas. Por otra parte en el caso de ser vacío significaría que no hay posibles tipos que satisfagan las operaciones realizadas sobre el AUTO_TYPE. Una vez comprobada la validez del AUTO_TYPE se pone a los nodos el tipo correspondiente a su posición ya sea el más abstracto o concreto. Esta segunda parte se realiza en el recorrido **AutoResolver**. Ejemplos de casos válidos (Izquierda) y no válidos (Derecha):

3.3.3. Ejecución

En esta etapa se ejecuta el método de entrada del programa devolviendo su resultado final o lanzando errores de ejecución en caso de existir. Se puede ver el flujo de ejecución en *cool.pipeline.execution_pipeline*, en este se usa principalmente el recorrido **RunVisitor**.