Hulk-Compiler:

El proyecto consiste en la implementación de un compilador para el lenguaje de programación HULK. Nuestro compilador abarca las etapas iniciales de todo proceso de compilación, el análisis léxico, el análisis sintáctico y el análisis semántico. Para la implementación se utiliza como lenguaje de programación `python: v3.11.4` y la API auxiliar provista en clases prácticas.

Sobre HULK:

La gramática de HULK se encuentra en el archivo `code/grammar/hulk_grammar.py`, para la cual nos hemos apoyado en la API provista en clases. Para una mejor comprensión visual hemos representado sus producciones en el archivo `code/grammar/grammarProposal.md`. La gramática cubre la mayoría de los aspectos abordados en la documentación de HULK.

Arquitectura del compilador:

El compilador sigue una arquitectura de tres etapas:

- **Análisis Léxico:** El lexer es el encargado de fragmentar una cadena inicial de código fuente del lenguaje en unidades lógicas, concidas como tokens, para facilitar un análisis sintáctico posterior.
- **Análisis Sintáctico:** El parser tiene la tarea de chequear la sintaxis de la secuencia de tokens proporcionada por el lexer, asegurando que cumpla con las reglas de la gramática.
- **Chequeo Semántico:** En esta etapa se realizan verificaciones semánticas para garantizar que el código cumpla con los predicados que caracterizan al lenguaje.

Pasemos a explicar detalladamente cada una de las etapas.

Lexer:

Generador de Lexer:

En un primer momento, nos surge la necesidad de tener una representación de las diferentes formas que puede adoptar un símbolo del lenguaje HULK; para ello hacemos uso de las expresiones regulares. Dicha representación se encuentra en `/code/token_table.py`.

Dado que las representaciones de los símbolos del lenguaje son expresiones regulares, y sabemos que el lenguaje de las expresiones regulares es regular, podemos asociar a cada una de estas un autómata finito determinista que la reconoce.

Teniendo los autómatas reconocedores de nuestras expresiones regulares, mapeamos su representación a una instancia de la clase `State`, ubicada en `/code/cmp/automata.py`, permitiéndonos agregarle al estado final una etiqueta que almacena su tipo como token del lenguaje.

Finalmente, pasamos a construir el autómata que reconoce en orden lineal cada token de una cadena específica de HULK.

Análisis léxico:

Dada una cadena del lenguaje pasamos a identificar los tokens que representan sus símbolos. Para ello, hacemos un recorrido sobre el autómata obtenido del generador de lexer moviéndonos por las transiciones con lookahead 1, las que, en caso de no existir, marcan el final del token que se está analizando. A dicho token se le asigna su tipo, en caso de exisitir, respetando cierto orden de precedencia.

Parser:

Para el proceso de parsing de una cadena, hemos optado por criterio bottom-up, para lo cual tenemos una clase abstracta `ShiftReduceParser` encargada de realizar el proceso de parsing.

Generador de Parser:

Como resultado del generador de parser, se obtiene la tabla Action-Goto. Para construir dicha tabla se implementa en la clase `SLR1Parser`, la función `_build_parsing_table`, en la que se parte de la gramática aumentada y del cálculo del conjunto de follows de los símbolos de la gramática, con los que se contruye el autómata de los items LR0, que se utilizará posteriormente para registar las diferentes acciones a tomar por cada par (estado, símbolo), en dependencia de lo que sugieran los items del estado.

La característica diferencial del parser SLR1 es proponer una operación reduce, solamente para los símbolos x, tales que x aparece en la lista de follows de S, siendo S la cabecera de la producción asociada a un item de reducción en un estado determinado.

Análisis sintáctico:

La clase abstracta `ShiftReduceParser` es la encargada de contener la implementación del algoritmo reconocedor general shift-reduce. Para ello se apoya en la tabla Action-Goto obtenida del generador de parser SLR1, simulando un recorrido por el autómata LR0, y detectando errores sintácticos durante dicho recorrido.

Semantic Cheker:

Una vez realizado el análisis sintáctico y no se hayan detectado errores, pasamos a la fase de análisis semántico para garantizar que el código fuente cumpla con las reglas semánticas del lenguaje. En el contexto del AST, el chequeo semántico implica verificar la consistencia de nuestros tipos.

Para realizar este análisis de manera eficiente adoptamos un enfoque bottom-up. Esto implica realizar un recorrido en post-orden sobre el AST, donde calculamos y verificamos los tipos asociados a los nodos hijos antes de verificar la consistencia de los tipos en los nodos padres; para ello, definimos 3 métodos recursivos apoyándonos en el patrón visitor proporcionado en clase.

TypeCollector:

Con este recorrido buscamos recolectar los tipos propios de HULK, incluyendo las definiciones de tipos proporcionadas en el código fuente.

TypeBuilder:

Con este recorrido buscamos definir las propiedades y funciones correspondientes a los tipos registrados en el recorrido anterior. Dado que en HULK está permitida la herencia, aquí también nos aseguramos de que no existan ciclos entre las definiciones de tipos. Además, para garantizar en caso de que un nodo hijo reescriba una función de un nodo padre, se asgura que en el momento de definición de un hijo ya se haya visitado al padre, teniendo constancia de esta forma de los valores heredados para el análisis.

TypeCheker:

Con este recorrido buscamos recorrer todos los nodos que constituyen nuestro AST. Para ello, utilizamos la clase `Context` proporcionada en el archivo `/code/cmp/semantic.py` en la que tenemos registrados los tipos definidos en nuestro lenguaje. A su vez, para poder ubicar cada definición de variable en un contexto específico se utiliza la clase `Scope`, ubicada en la misma ruta antes mencionada, que nos permite ir registrando scopes anidados. A medida que se realiza este recorrido se va chequeando que se haga un uso correcto de tipos a lo largo de las expresiones utilizadas, que no se referencien funciones o atributos inexistentes o fuera de scope y que la invocación de funciones y tipos sea la correcta.