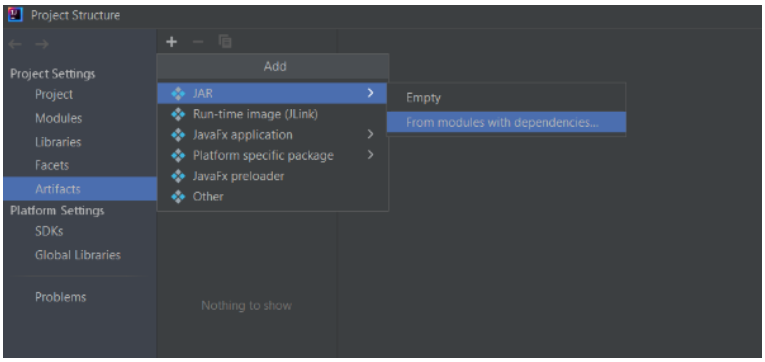
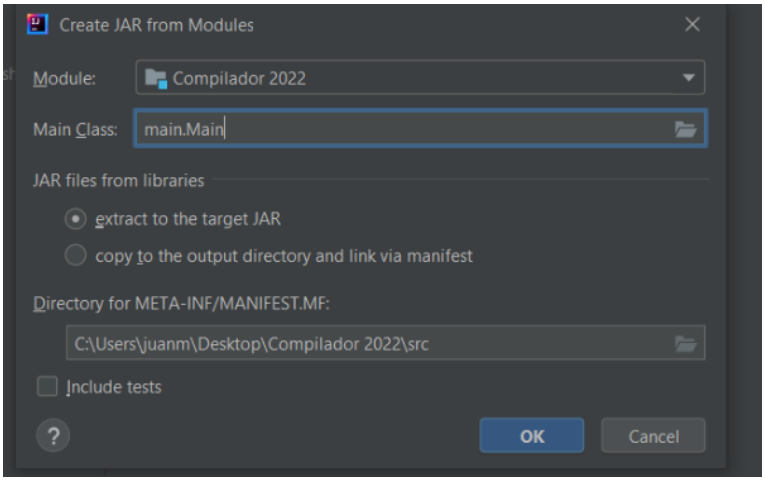
Sofia Crisafulli

Informe Análisis Semántico parte 2

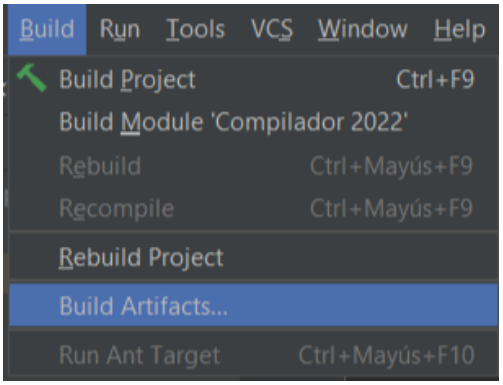
Como compilar el código fuente

Para poder compilar el código fuente en Intellij primero es necesario crear un “artifact”, esto se hace desde el menú de Project Structure en el apartado Artifacts.





Luego, en la barra de herramientas ir a la parte de “build artifacts…”



Luego de esto se creará la carpeta Artifact dentro de la carpeta out. Que contendrá una carpeta con el código compilado en un .jar
Para ejecutar este jar debe abrir cmd y dirigirse a la dirección donde se encuentre el .Jar y luego ejecutar el siguiente comando java -jar [nombre .jar] [nombre del archivo a pasar por parámetro]

Tipos de errores semánticos

Los tipos de errores semánticos que son posibles detectar son los especificados en las reglas de semánticas subidas por la cátedra.

Sofia Crisafulli

Decisiones de diseño

En esta nueva etapa, se tuvo que crear un nuevo módulo como **Nodos** en el que a su vez contiene varios submódulos:

Por cada sentencia y expresión posible, se creó una clase que representa a cada tipo de nodoAST, así por ejemplo para la sentencia if, tenemos la clase NodoIf, para la asignación de la forma acceso = expresion, tenemos la clase NodoAsignacionEpxresion, para un acceso a un constructor tenemos la clase NodoAccesoConstructor, y así siguiendo.

Todo nodoAST cuenta con un método llamado chequear(), el cual es invocado a la hora de hacer el chequeo de sentencias. Dependiendo el tipo de nodo, este método puede variar en su signatura, ya que por ejemplo las sentencias al chequearse no devuelven ningún valor, pero las expresiones al chequearse devuelven el Tipo de la expresión. Inicialmente, cuando vamos a comenzar a chequear las sentencias de una unidad, el primer nodo al cual se le pide chequearse es al NodoBloque, ya que toda unidad posee un atributo que es el nodoBloque que caracteriza el cuerpo de la unidad. Entonces cuando comienza a chequearse este NodoBloque, el mismo al tener una lista con todas sus sentencias (que se insertaron ordenadamente al crearse el AST), solicitará de forma ordenada a cada sentencia que se chequee. Luego cada sentencia lo que hará es ver si está correctamente declarada dependiendo del tipo de sentencia que sea. Así como el nodoBloque delega los chequeos a sus sentencias, pueden haber otras sentencias que además de chequearse, soliciten a sus miembros que se chequeen. Por ejemplo, parte de los controles de la sentencia if (es decir, de NodoIf) consisten en primero solicitar a la expresión de la condición que se chequee para verificar que la misma esté correctamente tipada. Si está correctamente tipada, entonces obtenemos el tipo de la expresión, luego con este tipo, NodoIf verificará que sea de tipo boolean, si no se cumple ésto último, chequear() de NodoIf lanza un error.

Los nodos se decidieron agrupar en 4 categorías diferentes: por un lado tenemos los **nodosSentencia**, cuyo método chequear() es de tipo void. Por otro lado, tenemos **los nodosExpresion**, cuyo metodo chequear() devuelve un Tipo de la expresión. En nodosExpresion también tenemos nodoExpresionCompuesta, nodoExpresionVacia y nodoExpresionAsignacion.

De los NodosEncadenados se dividen en NodoVarEncadenada y NodoLlamadaEncadenada las cuales además del metodo chequear() también tienen un metodo esAsignable() el cual es boolean.

Los NodosAccesos también tienen un metodo esAsignable() en donde chequea si tiene encadenado debe ser una variable o si no tiene encadenado deber ser un acceso a una variable y el metodo chequear() en el que devuelve un Tipo.

Como de NodoSentencia hereda NodoLlamadaOAsignacion para distinguir entre estas, todos los nodos expresiones implementan los métodos booleanos esLlamada() y esAsignacion() de esta forma no se producen ambigüedades.

La relación entre los distintos nodosAST puede verse con mayor detalle en el **diagrama de clases** que viene junto a este documento. A rasgos generales, cada nodo controla lo siguiente:

- **NodoBloque:** que todas sus sentencias sean correctas
- **NodoAsignacionExpresion:** que el acceso del lado izquierdo esté correctamente tipado y sea asignable, que la expresión del lado derecho esté correctamente tipada y que conforme con el acceso.
- **NodoLlamadaOAsignacion:** que el acceso esté correctamente tipado y sea llamable.
- **NodoVarLocal:** que el tipo exista, y que el nombre de la var local declarada no esté repetido en los lugares donde no se puede.
- **NodoReturn:** que si el tipo del método es void, no retorne una expresión, o si no es void y se usa return, que retorne una expresión, y en este último caso que el tipo de la expresión conforme con el tipo del método.
- **NodoIf:** que la expresión de la condición este correctamente tipada y esa booleana, y que la sentencia del cuerpo del if sea correcta. En caso de haber else, que la sentencia del mismo también sea correcta.
- **NodoSentenciaVacia:** es inherentemente correcto.
- **NodoExpresionBinaria**: por cada operador binario (suma, resta, etc.) tenemos una clase que extiende de NodoExpresionBinaria. Cada clase verifica que su expresión del lado izquierdo y su expresión del lado derecho estén correctamente tipadas, y que sus tipos conformen con el operando.
- **NodoExpresionUnaria:** verifica que el operando esté correctamente tipado, y que su tipo sea el que requiere el operador (dependiendo si es +, -, o !).
- **NodoExpresionAsignacion:** chequea que el lado derecha e izquierda sean asignación correctamente tipadas, que el lado izquierdo solo puede ser un acceso^2, es decir que, si tiene un encadenado, el ultimo

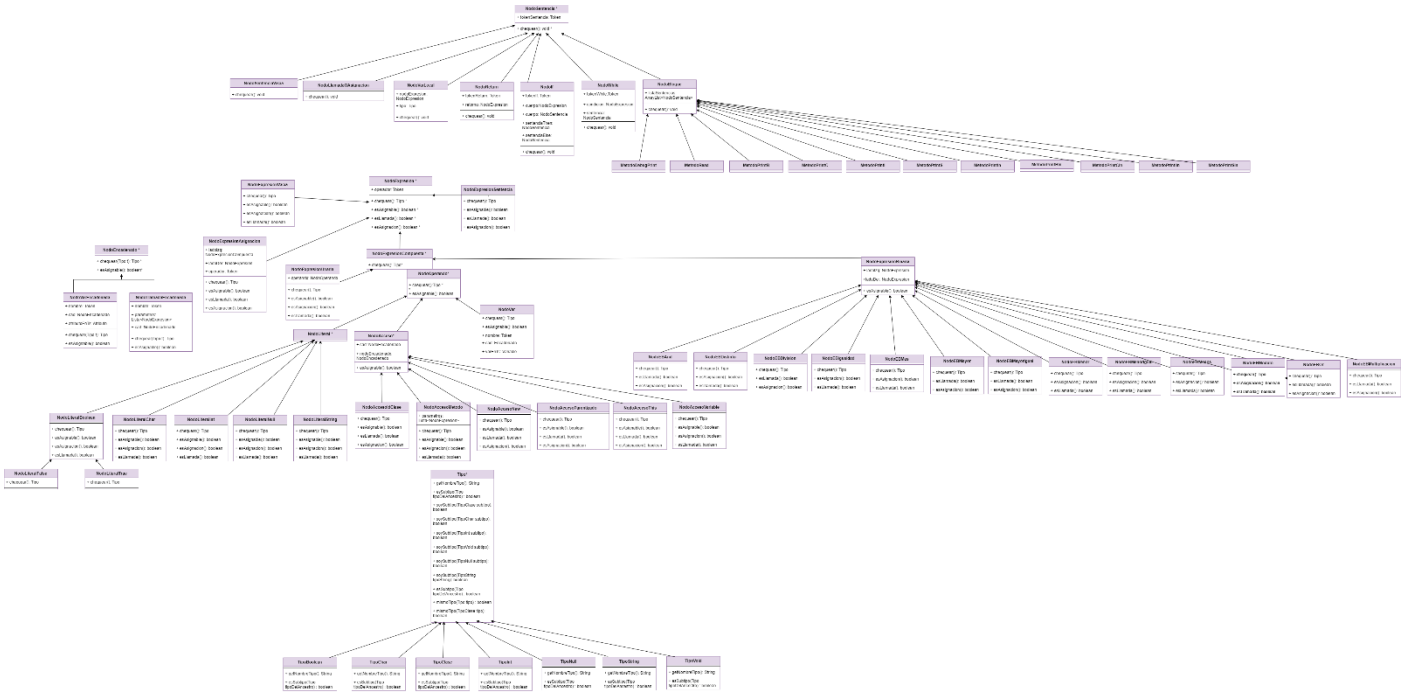
elemento del encadenado debe ser una variable, si no tiene encadenado debe ser un acceso variable y el tipo de la derecha debe ser subtipo del del lado izquierdo.

- **NodosLiterales:** estos son NodoNull, NodoTrue, NodoFalse, NodoEntero, NodoCaracter, NodoString. Simplemente devuelven su tipo, siempre están correctamente tipados.

NodosAcceso

- **NodoAccesoVar:** controla que exista la variable que se está usando en el alcance actual, y de existir, recupera su tipo. Si tiene un encadenado, le pasa el tipo recuperado por parámetro y le pide que se chequee. Si no lo tiene, simplemente devuelve el tipo recuperado.
- **NodoAccesoMetodo:** controla que exista el método que se está intentando invocar, y que los parámetros actuales conformen con los formales. En caso de existir, también controla que si el método a invocar es dinámico, que se esté invocando desde dentro de un método dinámico. En caso de superar estos chequeos, si posee un encadenado, le pasa el tipo del método recuperado por parámetro para que se chequee el encadenado. Sino sólo devuelve el tipo del método.
- **NodoAccesoConstructor:** controla que exista la clase de la cual se está intentando instanciar un nuevo objeto. En caso de existir, controla que los parámetros actuales conformen con los formales. Si se superan estos controles, en caso de tener un encadenado, le pasa por parámetro el tipo que es la clase del constructor. Sino sólo devuelve este tipo.
- **NodoAccesoThis:** solamente controla que no se esté utilizando this en un método estático. Si poseé un encadenado, le pide al mismo que se chequee.
- **NodoAccesoVarEncadenada:** controla que el tipo recibido por parámetro sea un TipoClase, y controla que el atributo al cual se intenta acceder de esa clase exista y que sea público. Si se superan estos controles, en caso de tener un encadenado, le pide al mismo que se chequee pasándole el tipo del atributo recuperado por parámetro.
- **NodoAccesoMetodoEncadenado:** controla que el tipo recibido por parámetro sea un TipoClase, y controla que el método al cual se intenta acceder de esa clase exista. Si se superan estos controles, en caso de tener un encadenado, le pide al mismo que se chequee pasándole el tipo del atributo recuperado por parámetro.
- **Tipo:** cada tipo tiene su propia clase en donde implementan los métodos getNombreTipo() y esSubtipo() con el nombre del ancestro pasado por parámetro

Un diagrama de clases es el siguiente:



Logros

- Imbatibilidad Semántica ||