Informe Etapa 4
Compiladores e Intérpretes

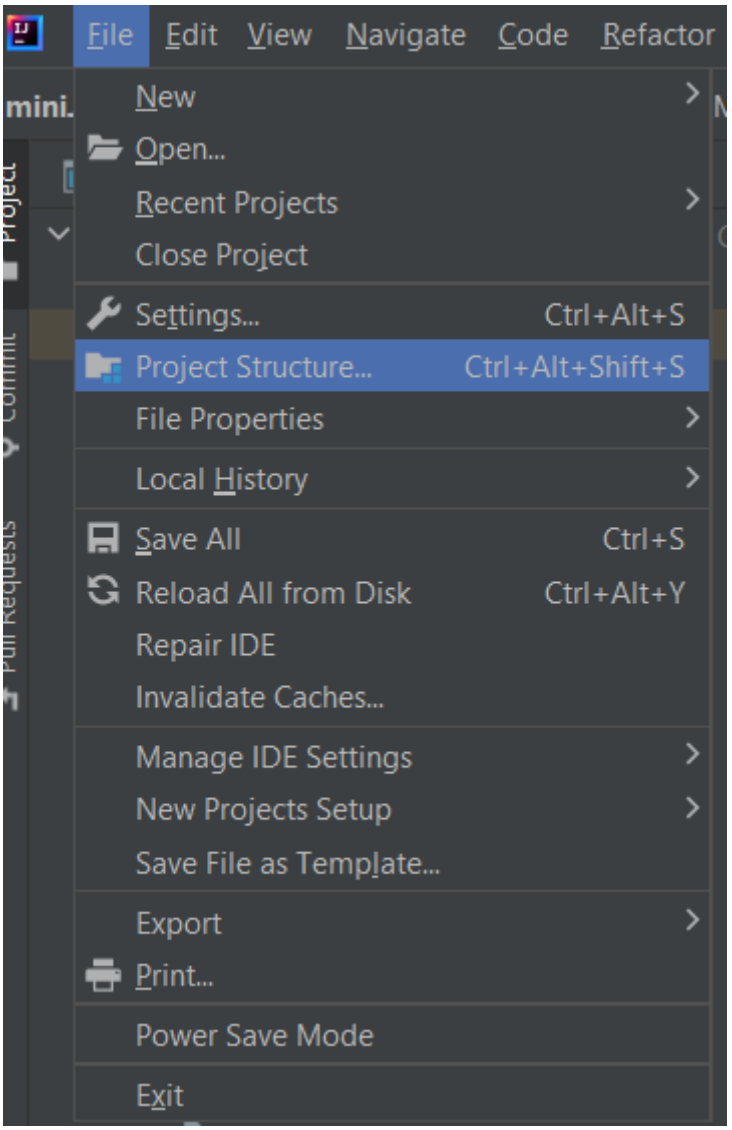
Índice

Modo de uso

Compilación

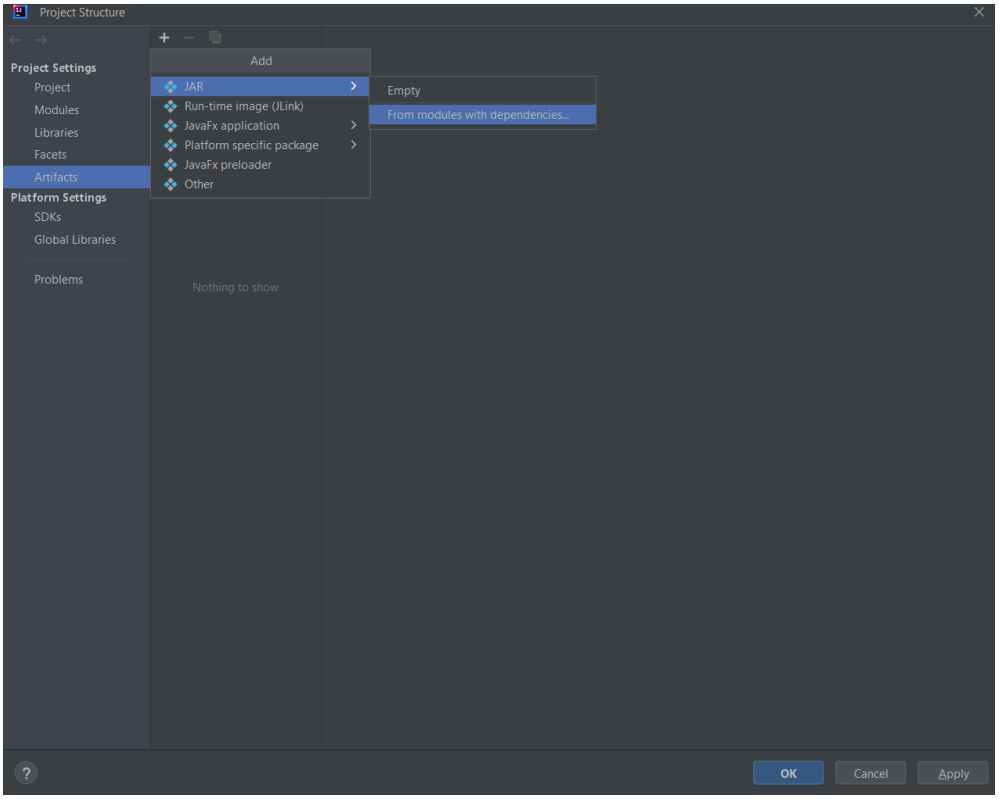
El proyecto fue desarrollado en InteliiJ y compilado con Java 11, para poder compilarlo es necesario usar la versión de Java mencionada y compilar los archivos fuentes en un ejecutable.

Esto se puede hacer desde IntelliJ de la siguiente manera:

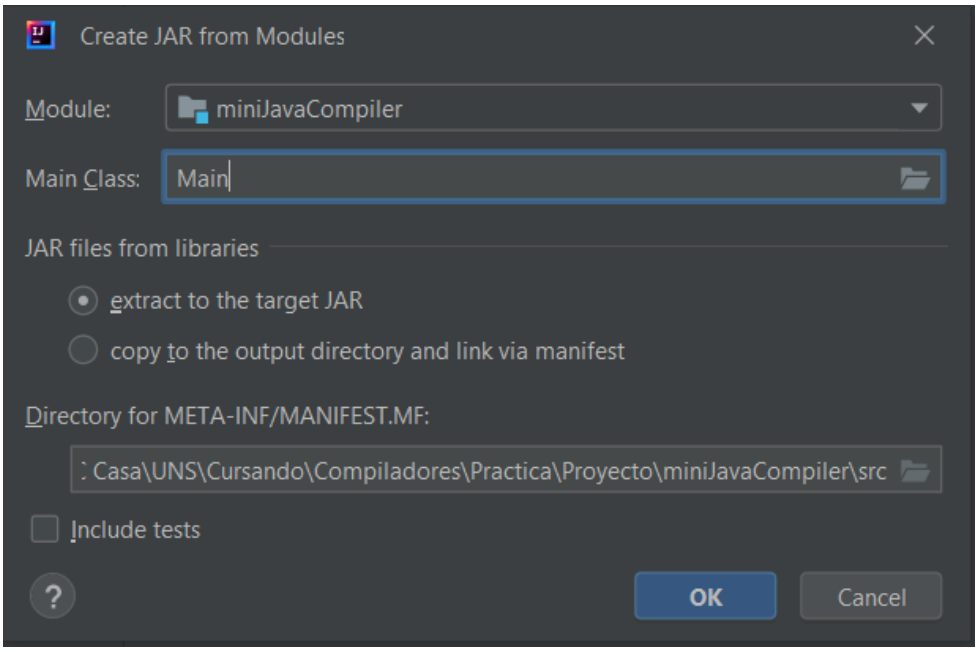


Primero creamos un nuevo proyecto con los archivos fuentes del proyecto, luego

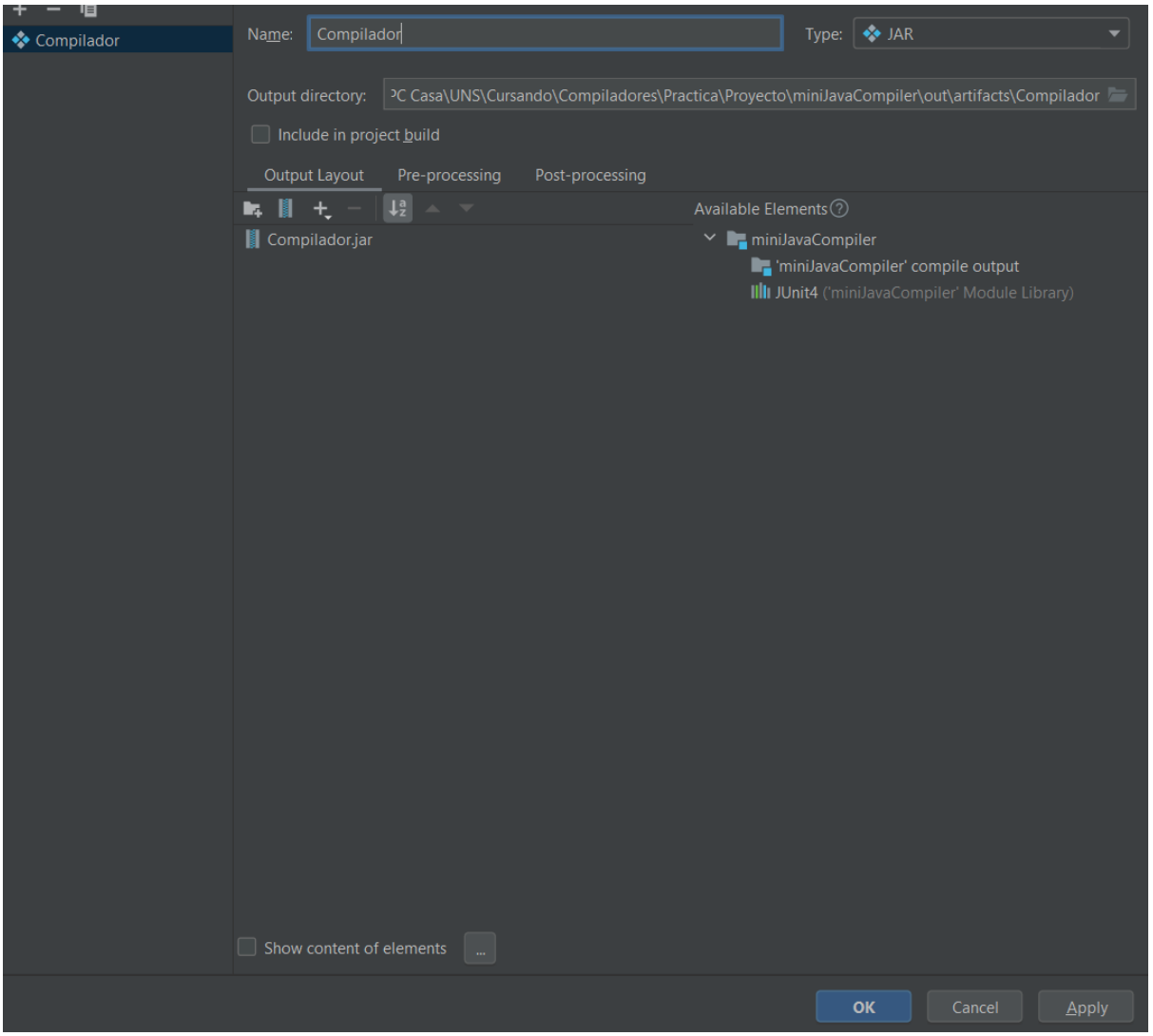
una vez construido, vamos a la estructura del proyecto



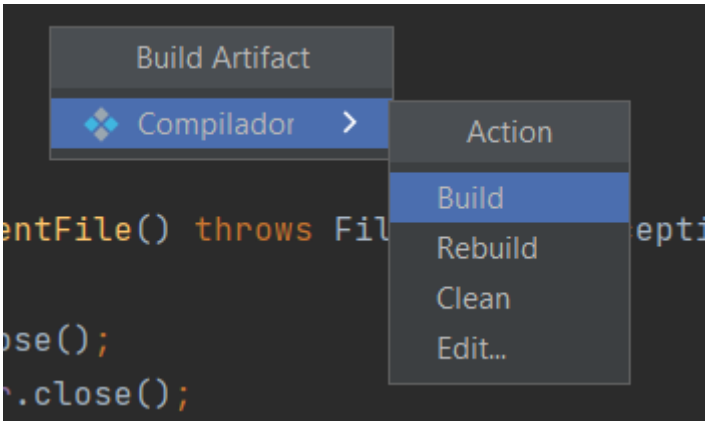
Aquí agregaremos un nuevo artifact que nos ayudará a generar el ejecutable JAR, recordar elegir la opción que crea el JAR basándose en módulos con dependencias



Nos pedira los módulos asociados a este ejecutable y allí seleccionamos el archivo Main del código fuente, luego de damos a OK, chequeamos el nombre y la dirección donde se generar el .jar y aplicamos



Ya preparamos todo para compilar, ahora solo hace falta crearlo, para ello vamos a Build Artifacts y creamos el artifact que acabamos de definir



Uso

Una vez con el ejecutable se puede utilizar el programa desde la terminal de comandos, en el lugar donde se encuentra el ejecutable del programa y los archivos a compilar se debe escribir :

```
java -jar miniJavaCompiler.jar programa.java
```

Siendo miniJavaCompiler.jar el ejecutable y programa.java el archivo a compilar.

Es muy importante recordar que ambos deben estar al mismo nivel para poder acceder de manera correcta, sino también se puede suministrar un camino hacia el archivo a compilar, siempre asumiendo que se parte de la ubicación actual en la que se encuentra la terminal, solo se le puede suministrar un archivo a compilar a la vez, el resto de argumentos serán ignorados.

Errores reconocibles

Sumado a los errores reconocibles del analizador léxico que eran :

- Símbolos que no pertenecen al alfabeto
- Comentarios multilínea sin cerrar
- Cadena de caracteres sin cerrar
- Operadores lógicos no válidos
- Floats que exceder el valor representable
- Floats que están mal formateados

Y los del el analizador sintáctico que son :

- Bloque de ejecución sin abrir o cerrar
- Línea sin terminar correctamente
- Extensión de múltiples clases
- Implementación de múltiples clases
- Anidamiento de parámetros genéricos
- Constructor estatico
- Variables locales con modificadores de visibilidad o estáticos
- Declaración de variables en condiciones de if o while
- Errores relacionados al anidamiento de llamados

Se les agrega los errores relacionados al análisis semántico, estos son , relacionado al chequeo de declaraciones :

- Herencia Circular
- Tipo de atributo, retorno o parámetro no declarado
- Faltante de método main
- Métodos con mismo nombre pero no redefinen
- No implementar la totalidad de una interfaz
- Atributos de tipo *void*
- Mas de un constructor declarado para una misma clase

Y relacionado al chequeo de sentencias :

- Asignación no válida (Tipos no conformes o inválidos)
- Operaciones no válidas (Tipos no conformantes)
- Llamados o accesos a variables/métodos no existentes.
- Colisión de nombres entre variables locales con atributos u otras variables.
- Expresiones no booleanas en estructuras como If o While
- Llamada a un método con tipo de atributos incorrectos

Decisiones de diseño

De etapas posteriores se aumentó la complejidad del analizador sintáctico, a tal que punto que fue difícil implementar un esquema de creación de nodos para el AST , se logró pero con algunos compromisos :

- Si bien las variables clásicas las sigue reconociendo el lenguaje no se les hace chequeo de sentence
- La precedencia al evaluar es de derecha a izquierda, véase 3 + 4 > 3 se evalúa 4 > 3 y luego 3 + (4 > 3) , si se parentiza se puede cambiar la precedencia, esto ocurrió debido a las modificaciones previas realizadas a la gramática , haciendo que se tomará la decisión de que prevalezca este orden de precedencia.

Clases utilizadas

- **FileManager**, establece la interfaz de un manejador de texto
- **ImpFileManager**, implementa la interfaz de un manejador de texto
- **Token**, representa un *token* de un analizador léxico, con id de operación, lexema, etc
- **LexicalAnalyzer**, encargado de realizar el análisis léxico sobre el código
- **LexicalException**, una excepción que puede lanzar LexicalAnalyzer
- **SyntaxAnalyzer**, encargado de realizar el análisis sintáctico sobre el código, le va pidiendo tokens a LexicalAnalyzer
- **SyntaxException**, una excepción que puede lanzar SyntaxAnalyzer que informa de un error sintáctico.
- **SemanticException,** una excepción que puede lanzar cualquier entidad de la tabla de símbolos o el mismo *SyntaxAnalyzer*, informa de un error semántico.
- **Main**, encargada de mostrar los errores o ejecución exitosa, tomar los argumentos de consola , crear el manejador de archivo, analizador léxico y sintáctico.
- **MainStage1,** respectiva clase Main para ejecutar la entrega de la etapa 1.
- **MainStage2,** respectiva clase Main para ejecutar la entrega de la etapa 2.
- **MainStage3,** respectiva clase Main para ejecutar la entrega de la etapa 2.
- **SyntaxAnalyzerStage2**, la clase *SyntaxAnalyzer* perteneciente a la etapa 2.
- **SyntaxAnalyzerStage3**, la clase *SyntaxAnalyzer* perteneciente a la etapa 3.
- **Type**, clase abstracta que modela y define a los tipos del compilador.
- **ReferenceType**, hereda de *Type* y modela los tipos que son por referencia, ergo, tipo clases como *Object* o *String*.
- **PrimitiveType**, hereda de *Type* y modela los tipos primitivos , estos son *int, float, char*, etc.
- **EntityST**, interfaz que modela a grandes rasgos los métodos que usa una entidad que pertenece a la tabla de símbolos.

- **SymbolTable,** modela la tabla de símbolos que se usa durante el análisis semántico, tanto crearla cómo chequear que su contenido sea el correcto.
- **AttributeST,** modela la entidad atributo en la tabla de símbolos, tiene el nombre y el *Type* de un atributo.
- **ClassST,** modela la entidad clase en la tabla de símbolos, tiene nombre, clase que hereda o interfaz a implementar, etc.
- **InterfaceST,** modela la entidad interfaz en la tabla de símbolos, su nombre, si extiende a otra interfaz, que métodos declara, etc.
- **RoutineST,** clase abstracta que define que compone a una rutina o unidad en la tabla de símbolos, nombre, lista de parámetros, como se comparan, etc.
- **ConstructorST,** extiende a *RoutineST* y modela la entidad constructor de la tabla de símbolos.
- **MethodST ,** extiende a *RoutineST* y modela la entidad método en la tabla de símbolos, agrega el tipo de retorno que puede tener un método.
- **ParameterST,** modela la entidad parámetro en la tabla de símbolos , con su identificador y tipo.

Relacionadas al **AST** y el chequeo de sentencias

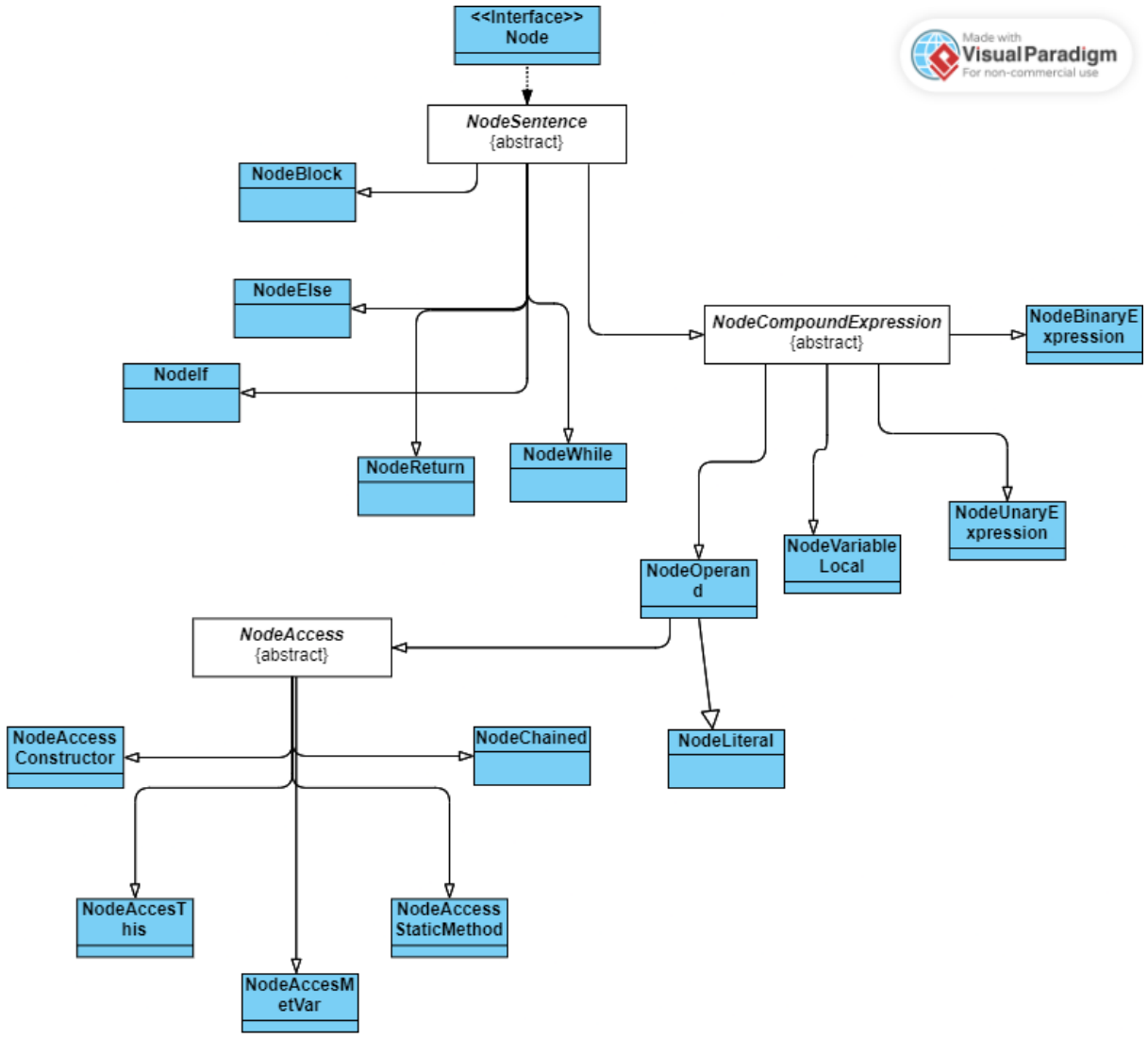
- **Node,** interfaz que define los métodos que deben implementar todos los nodos que componen el AST.
- **NodeAssignment,** representa al nodo asignación con una expresión a su izquierda y otra a su derecha, evaluando si son de tipos admisibles para asignar y estén bien definidos.
- **NodeBinaryExpression,** representa cualquier expresión binaria (sin contar la asignación) que pueda aparecer en el AST, controla que ambas expresiones, izquierda y derecha, estén bien definidos y a su vez el operador también lo esté. Por último, que cumplan con el control de tipos.
- **NodeBlock**, clase que representa la raíz de un bloque de sentencias, almacena las variables locales con su nombre y tipo, además es la encargada de chequear todas sus sentencias
- **NodeCompoundExpression,** clase abstracta que encapsula la idea de las expresiones que puede generar la gramática, brinda métodos útiles para chequear la conformación de tipos y parámetros.
- **NodeElse,** clase que representa la sentencia Else que puede aparecer luego de un If.
- **NodeIf,** clase que representa la sentencia If, con su expresión de condición y su sentencia *then* de ejecución, así de además su posible Else. Chequea que su expresión de condición sea efectivamente booleana y esté bien definida, igual que su sentencia.

- **NodeLiteral,** representa un número, carácter, booleano, etc. Que puede aparecer en el código, tiene un tipo asociado.
- **NodeOperand,** representa a cualquier operador que pueda aparecer en el AST, tanto binario como unario, tiene un tipo que retorna (el resultado de operar con este) y tipos que requiere para operar.
- **NodeReturn,** representa el Return que puede o no contener una expresión, se controla que de tener una expresión este tipo sea conformante con el del método donde aparece.
- **NodeSentence,** clase abstracta que define la noción de una sentencia.
- **NodeUnaryExpression,** clase que define una expresión con un operando o un operando con un operador unario, se encarga de hacer el chequeo de tipos para saber si es válido.
- **NodeVariableLocal**, tiene un identificador y un como el resto de expresiones un tipo de retorno, además tiene una expresión asociada para su inicialización para saber su tipo, se evalua que esta expresión sea correcta.
- **NodeWhile**, representa el While como estructura de control, con su expresión de condición, que se controla que sea de tipo booleano, y su bloque de sentencias bien definidas.
- **NodeAccess,** clase abstracta que define lo que compone a un acceso o llamado, además de guardar una lista con los argumentos y tipos.
- **NodeAccessConstructor,** clase concreta derivada de *NodeAccess* que representa el acceso a un constructor *new <Nombre_Constructor();,* se encarga de chequear que dicho constructor exista y los parámetros este bien definidos.
- **NodeAccessMetVar,** clase concreta derivada de *NodeAccess* que representa el acceso a un método o variable, el atributo *isAtribute,* chequea que si es un acceso a una variable esta puede ser un variable local, parametro u atributo, si es un método que exista y coincida con los argumentos.
- **NodeAccessStaticMethod,** clase concreta derivada de *Node_Access* que representa el acceso estático a un método, chequea que exista el método al que se quiere acceder y este sea estático, además de coincidir en sus argumentos.
- **NodeAccessThis,** clase concreta derivada de *NodeAccess* que representa el acceso *this*, haciendo referencia al objeto actual, delega su chequeo al encadenamiento.
- **NodeChained,** clase concreta derivada de *NodeAccess* que representa cualquier encadenado posible del lenguaje, hace controles para saber si es un atributo, método , parámetro, variable, etc. Además es un factor importante para saber si un encadenado es asignable o no.

A grandes rasgos las clases de tipo **Node** heredan la mayoría directamente de **NodeSentence** y otras de **NodeAcess**, véase cómo se encuentran los paquetes. Los nodos conocen a sus

NodeBlock padre que los creó, entre ellos se los propagan para que todos tengan un nodo padre. Además a veces se pasan el entorno donde existe el bloque, por si es necesario consultar las variables locales u otros métodos o clases.

Diagrama de clase de nodos del AST



Logros intentados

No se han intentado logros en esta etapa ni se a modificado el proyecto para cumplir logros de entregas previas

Ejecución de previas etapas

Existen múltiples clases con el nombre **Main** para poder ejecutar las entregas de las etapas previas, por ejemplo si se ejecuta el método *main* de la clase **MainStage2** entonces se ejecutara la entrega correspondiente a la etapa 2 del proyecto, en este caso era la del analizador sintáctico.

Usando la herramienta de testing provista por la cátedra esto es solamente cambiar el tipo de la constante *init* . Si se tiene en esta herramienta :

```
...
private static final Main init = null;
...
```

Se ejecutaría la batería de *tests* sobre la entrega actual, si se lo cambia por :

```
...
private static final MainStage2 init = null;
...
```

Se ejecutaría la batería de *tests* sobre la entrega de la etapa 2, así pudiendo probar los logros previos o correcciones, **MainStage2** corresponde a la clase **Main** de la etapa 2 y **MainStage1** corresponde a la clase **Main** de la etapa 1.

Otros cambios

Se arregló un error relacionado a admitir constructor que no pertenecían a las clases, por ejemplo se admitía

```
Class B {
     public A(){}
}
```

Ahora ya no se permite.