Informe Etapa 2
Compiladores e Intérpretes

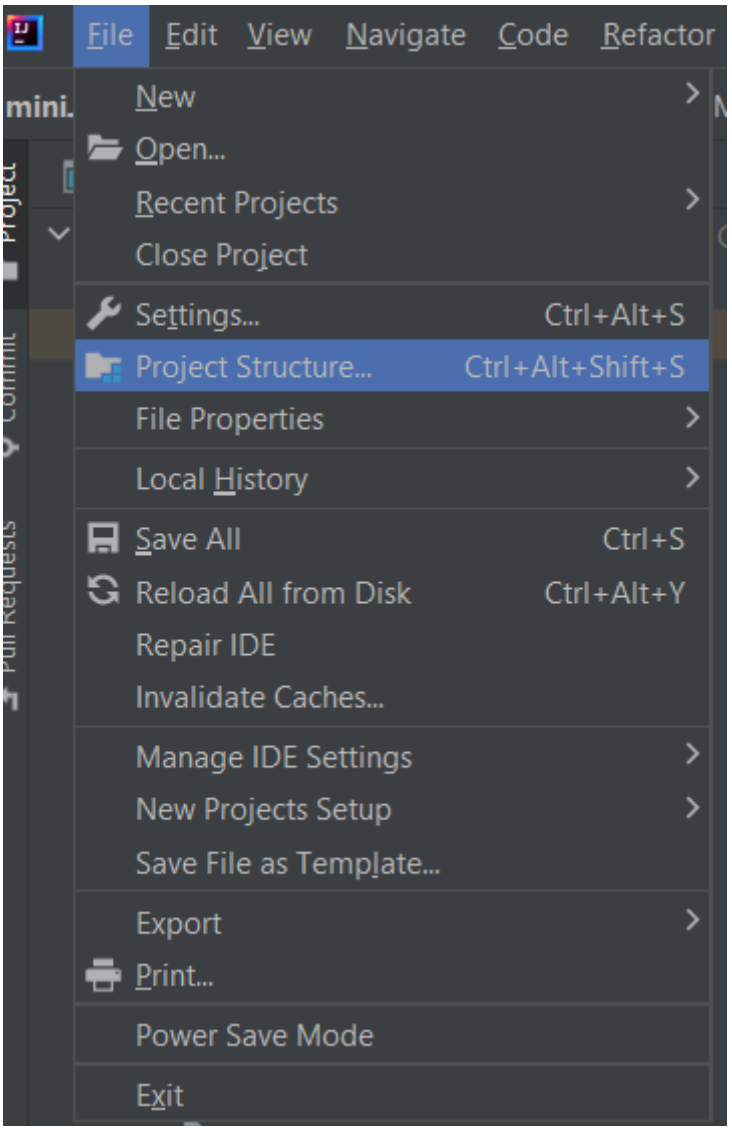
Índice

Modo de uso

Compilación

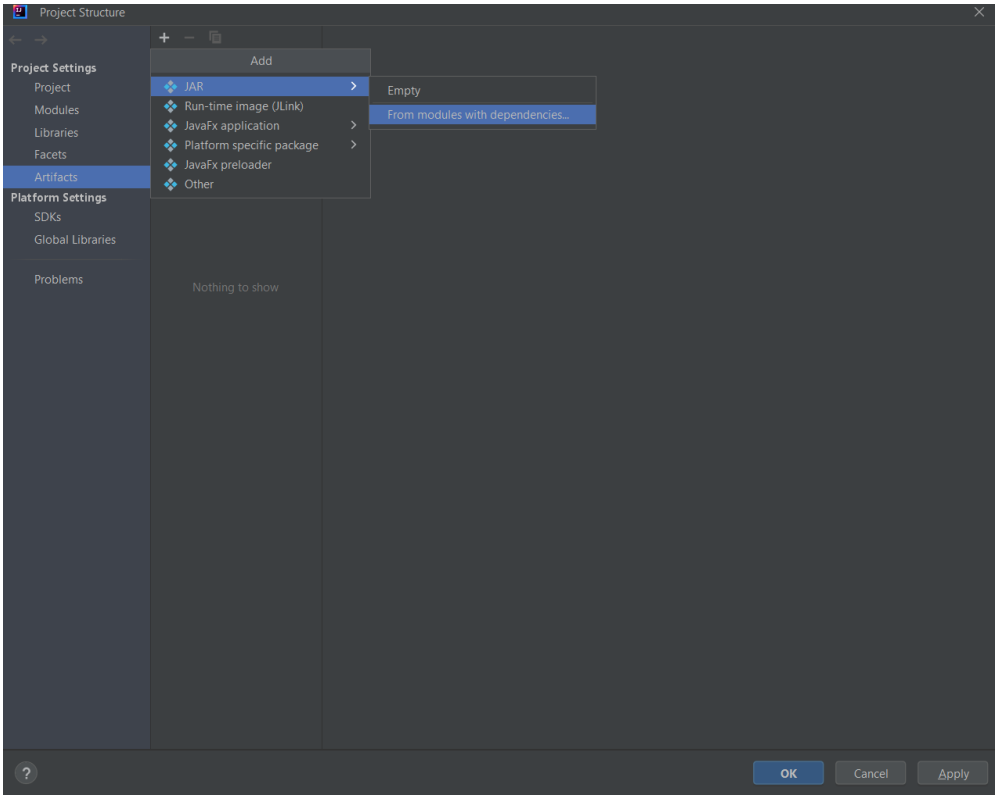
El proyecto fue desarrollado en InteliiJ y compilado con Java 11, para poder compilarlo es necesario usar la versión de Java mencionada y compilar los archivos fuentes en un ejecutable.

Esto se puede hacer desde IntelliJ de la siguiente manera:

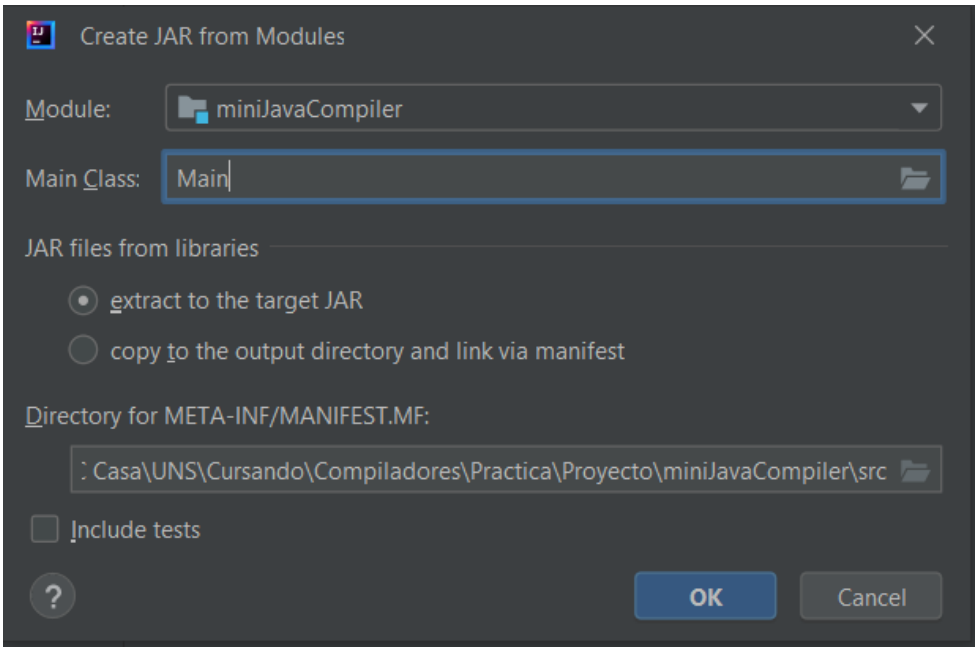


Primero creamos un nuevo proyecto con los archivos fuentes del proyecto, luego

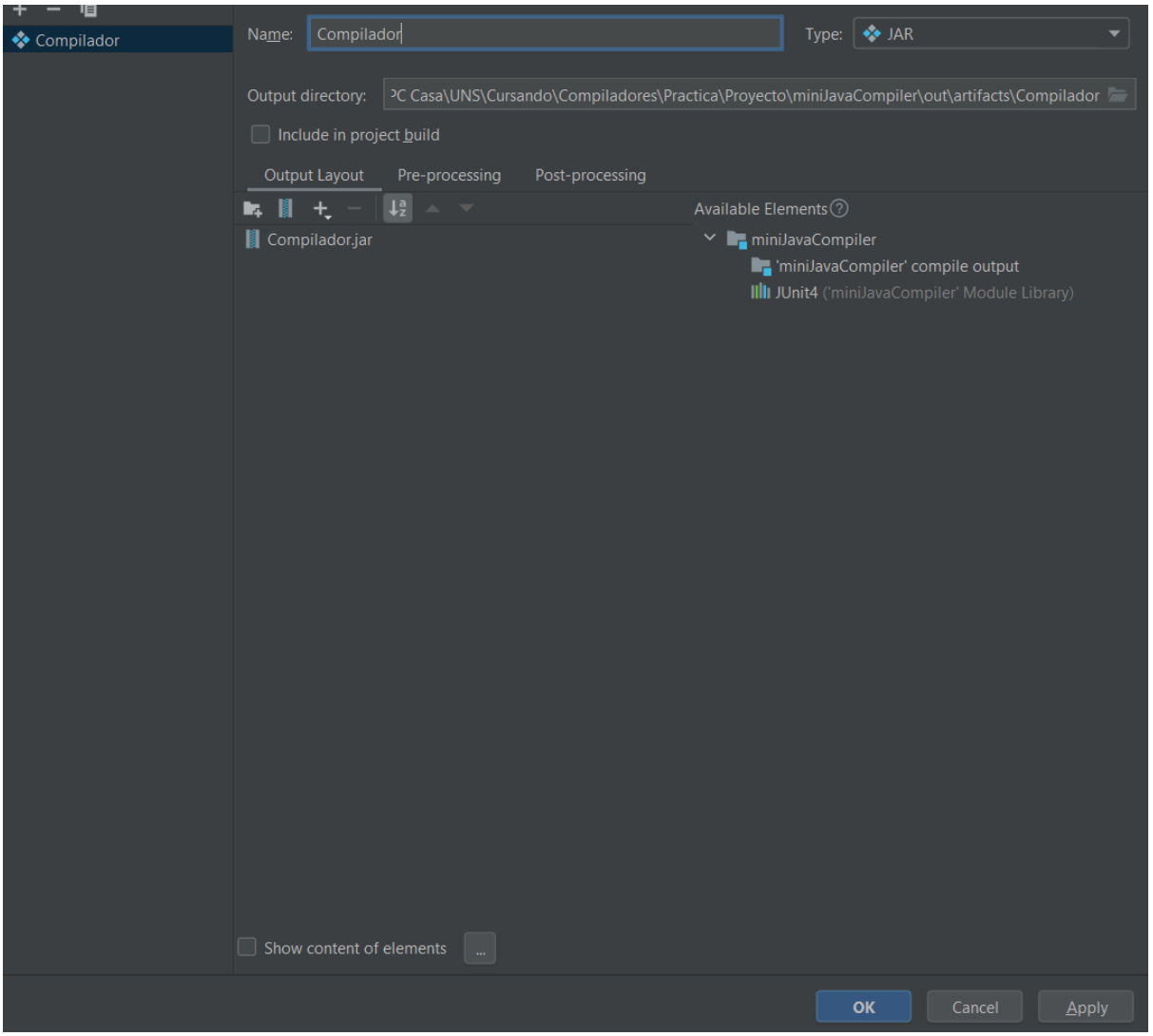
una vez construido, vamos a la estructura del proyecto



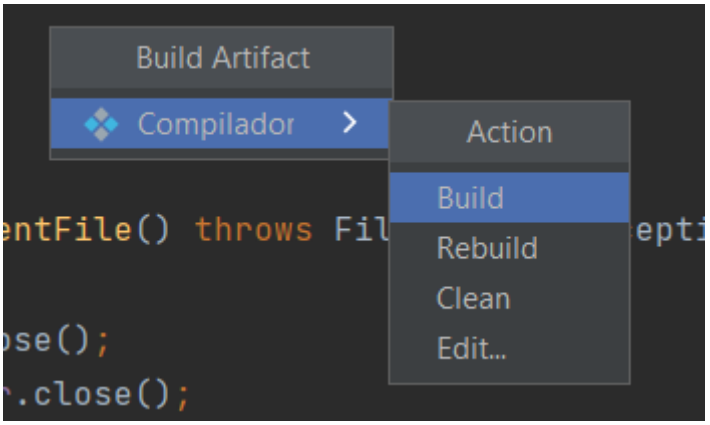
Aquí agregaremos un nuevo artifact que nos ayudará a generar el ejecutable JAR, recordar elegir la opción que crea el JAR basándose en módulos con dependencias



Nos pedira los módulos asociados a este ejecutable y allí seleccionamos el archivo Main del codigo fuente, luego de damos a OK, chequeamos el nombre y la direccion 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.

Transformación de la gramática

Modificación para opcionales

Antes de hacer intentar transformar la gramática libre de contexto que representa la sintaxis del lenguaje en una gramática LL(k) se agrega y modifica la misma para cumplir con algunos logros que se buscaron obtener, las reglas de producción que se modificaron fueron :

```
<Clase concreta> ::= class idClase <HerenciaOpcional>
{<ListaMiembros>}
```
se transformo en
```
<Clase concreta> ::= class idClase <GeneracidadOpcional>
<HerenciaOpcional>{<ListaMiembros>}
```

```
<Atributo> ::= <EstaticoOpcional> <TipoMiembro> idMetVar ;
```
se transformo en
```
<Atributo> ::= <VisibilidadOpcional> <EstaticoOpcional>
<TipoMiembro> idMetVar  <DeclaracionVariabelMultiple>
<InicializacionOpcional> ;
```

```
<Metodo> ::= <EstaticoOpcional> <TipoMiembro> idMetVar
<ArgsFormales> <Bloque>
```
se transformó en
```
<Metodo> ::= <VisibilidadOpcional><EstaticoOpcional>
<TipoMiembro> idMetVar <ArgsFormales> <Bloque>
```

```
<EncabezadoMetodo> ::= <EstaticoOpcional> <TipoMiembro>
idMetVar <ArgsFormales> ;
```
se transformó en
```
<EncabezadoMetodo> ::= <VisibilidadOpcional><EstaticoOpcional>
<TipoMiembro> idMetVar <ArgsFormales> ;
```

```
<Constructor> ::= public idClase <ArgsFormales> <Bloque>
```
se transformó en
```
<Constructor> ::= <VisibilidadOpcional> idClase
<GenericidadOpcional> <ArgsFormales> <Bloque>
```

```
<TipoPrimitivo> ::= boolean | char | int
```
se transformó en
```
<TipoPrimitivo> ::= boolean | char | int | float
```

```
<Literal> ::= null | true | false | intLiteral | charLiteral |
stringLiteral
```
se transformó en
```
<Literal> ::= null | true | false | intLiteral | charLiteral |
stringLiteral | floatLiteral
```

Los que sea agregaron fueron :

```
<DeclaracionClases> ::= , idClase <DeclaracionClases> | ε
<GenericidadOpcional> ::= < idClase <DeclaracionClases> > | ε
<VisibilidadOpcional> ::= public | private | ε
<InicializacionOpcional> ::= <ExpresionCompuesta> | ε
<Sentencia> ::= <VarLocalConTipoPrimitivo>
<VarLocalConTipoPrimitivo> ::= <TipoPrimitivo> idMetVar
<DeclaracionVariableMultiple>  <InicializacionOpcional>
<DeclaracionVariableMultiple>  ::= , idMetVar
<DeclaracionVariableMultiple>  | ε
```

Eliminar recursión a izquierda

En la gramática original no se encuentran muchos casos de recursión a izquierda, el único que se noto fue el que ocurría en la regla :

```
<ExpresionCompuesta>::=<ExpresionCompuesta><OperadorBinario>
<ExpresionBasica>

<ExpresionCompuesta> ::= <ExpresionBasica>
```

Un ejemplo de lo que ocurre es, el siguiente, sea Ec <ExpresionCompuesta> , Ob <OperacionBinaria> y Eb <ExpresionBasica> se da el caso que al derivar :

Ec → Ec Ob Eb → Ec Ob Eb Ob Eb → ... → Eb Ec Ob Eb Ob Eb Ob Eb Ob Eb ...

Vemos que en cualquier derivación tendrá la forma de
`<ExpresionBasica> (<OperadorBinario><Expresión Basica>)*`
, esta última siempre se agrega a la derecha dada que la recursión que ocurre a la izquierda, para solucionar esto se modifica la regla para que la recursividad no se dé del lado izquierdo de la derivación. La modificación de la regla previa agrega un nuevo no terminal y regla para solucionar el problema, ahora en la gramática en vez de la regla que generaba la recursión a izquierda se agrega

```
<ExpresionCompuesta>::=<ExpresionBasica><RExpresionCompuesta>
```

```
<RExpresionCompuesta>::=<OperadorBinario><ExpresionBasica>
<RExpresionCompuesta> | ε
```

Con esta regla la derivación previa se vería :

Ec → Eb REc → Eb Ob Eb Rec → Eb Ob Eb Ob Eb Rec → … → Eb Ob Eb … Ob Eb .

Ya no se encuentra esta recursión del lado izquierdo sino que está del lado derecho.

Factorización a izquierda

Se encontraron muchos casos donde en la gramatica ,luego de eliminar su recursion a izquierda, se podia factorizar a izquierda, algunos de esos casos son :

```
<If> ::= if ( <Expresion> ) <Sentencia>
<If> ::= if ( <Expresion> ) <Sentencia> else <Sentencia>
```

Vemos que tiene el mismo prefijo hasta que se llega al no terminal *else* , entonces los factorizamos y queda como resultado :

```
<If> ::= if (<Expresion>) <Sentencia> <Else>
<Else> ::= else <Sentencia> | ε
```

Este previo fue un caso donde la factorización a izquierda fue directa, pero también se dieron casos donde era profunda factorización , uno de estos fue :

```
<EncadenadoOpcional> ::= <VarEncadenada> | <MetodoEncadenado>
| ε
<VarEncadenada> ::= . idMetVar <EncadenadoOpcional>

<MetodoEncadenado> ::= . idMetVar <ArgsActuales>
<EncadenadoOpcional>
```

Aquí vemos que tanto `<VarEncadenada>` como `<MetodoEncadenado>` comparte prefijo, pero no pertenecen a la misma regla de producción, sin embargo si parten de la misma regla, `<EncadenadoOpcional>`, entonces se subieron las respectivas reglas que compartían prefijo

```
<EncadenadoOpcional> ::= . idMetVar <EncadenadoOpcional> |
.idMetVar <ArgsActuales> <EncadenadoOpcional> | ε
```

Y luego se hizo la factorización directa, resultando que la regla al terminar de factorizar queda como :

```
<EncadenadoOpcional> ::= .idMetVar <ArgsActualesOpcionales>
<EncadenadoOpcional> | ε

<ArgsActualesOpcionales> ::= <ArgsActuales> | ε
```

Algo importante a notar es que se reusaron reglas de producción que antes no estaban conectadas debido a que al factorizar se dieron casos donde la producción quedaba igual, el ejemplo previo generó la producción de `<ArgsActualesOpcionales>` que luego se utilizó al factorizar las producciones de acceso a métodos y variables

```
<Primario> ::= <AccesoVar>
<Primario> ::= <AccesoMetodo>
<AccesoVar> ::= idMetVar
<AccesoMetodo> ::= idMetVar <ArgsActuales>
```

Se ve que `<AccesoMetodo>` con `<AccesoVar>` tenían una situación similar a la factorización previa, ambos difieren únicamente en que una regla de producción tenía `<ArgsActuales>` por eso se reutilizó la regla , de otra manera tendríamos dos reglas idénticas con distinto nombre, algo no deseable, por completitud mostramos que la factorización del ejemplo de los acceso terminó siendo :

```
<Primario> ::= <AccesoMetVar>
<AccesoMetVar> ::= idMetVar <ArgsActualesOpcional>
```

Un último caso que se dió fue aquel en el que diversas reglas compartían prefijo pero al intentar subirlas nos dábamos cuenta que no derivan inmediatamente de la misma regla o ni siquiera estaban conectadas una con la otra, esto ocurrió al intentar factorizar

```
<Atributo> ::= <VisibilidadOpcional> <EstaticoOpcional>
<TipoMiembro> idMetVar <InicializacionOpcional>;

<Metodo> ::= <VisibilidadOpcional> <EstaticoOpcional>
<TipoMiembro> idMetVar <ArgsFormales> <Bloque>
```

```
<EncabezadoMetodo> ::= <VisibilidadOpcional>
<EstaticoOpcional> <TipoMiembro> idMetVar <ArgsFormales> ;
```

Vemos que comparten el mismo prefijo hasta *idMetVar*, el problema es que tanto `<Atributo>` como `<Metodo>` derivan de la producción de `<Miembro>` pero `<EncabezadoMetodo>` no lo hace, entonces esta última no se puede subir a la regla de producción, en este caso no se la tuvo en cuenta, se opto por tomar a la producción

```
<Constructor> ::= <VisibilidadOpcional> idClase
<GenericidadOpcional> <ArgsFormales> <Bloque>
```

que aunque solo compartía un No Terminal como prefijo si deriva de la regla de producción <Miembro> haciendo más completa la factorización a izquierda, el resultado fue

```
<Miembro> ::= <VisibilidadOpcional>
<AtributoMetodoOConstructor> | ε

<AtributoMetodoOConstructor> ::= <Constructor> |
<EncabezadoAtributoMetodo>

<EncabezadoAtributoMetodo> ::= <EstaticoOpcional>
<TipoMiembro> idMetVar <FinAtributoMetodo>

<FinAtributoMetodo> ::= <InicializacionOpcional> ; |
<ArgsFormales> <Bloque>

<Constructor> ::= idClase <GenericidadOpcional> <ArgsFormales>
<Bloque>
```

Resultado

La gramatica resultante de todos los pasos anteriores, sumado a un arreglo en una ambigüedad que se generaba en miembro debido a agregar al visibilidad, fue :

```
<Inicial> ::= <ListaClases>

<ListaClases> ::= <Clase> <ListaClases> | ε
```

```
<Clase> ::= <ClaseConcreta> | <Interface>

<DeclaracionClases> ::= , idClase <DeclaracionClases> | ε

<ClaseConcreta> ::= class idClase <GenericidadOpcional>
<HerenciaOpcional> { <ListaMiembros> }

<Interface> ::= interface idClase <ExtiendeOpcional> {
<ListaEncabezados> }

<HerenciaOpcional> ::= <HeredaDe> | <ImplementaA> | ε

<GenericidadOpcional> ::= < idClase <DeclaracionClases> > | ε

<HeredaDe> ::= extends idClase <GenericidadOpcional>

<ImplementaA> ::= implements idClase <GenericidadOpcional>

<ExtiendeOpcional> ::= extends idClase <GenericidadOpcional> |
ε

<ListaMiembros> ::= <Miembro> <ListaMiembros> | ε

<ListaEncabezados> ::= <EncabezadoMetodo> <ListaEncabezados> |
ε

<Miembro> ::= <VisibilidadOpcional>
<AtributoMetodoOConstructor>

<AtributoMetodoOConstructor> ::= static
<EncabezadoAtributoMetodo> |
<EncabezadoAtributoMetodoConstructor>

<EncabezadoAtributoMetodoConstructor> ::= idClase
<GeneracidadOpcional> <PosibleConstructor> |
<TipoMiembroSinClase> <EncabezadoAtributoMetodoTipoDicho>

<EncabezadoAtributoMetodoTipoDicho> ::= idMetVar
<FinAtributoMetodo>
```

```
<PosibleConstructor> ::= idMetVar <FinAtributoMetodo> |
<ArgsFormales> <Bloque>


<EncabezadoAtributoMetodo> ::= <TipoMiembro> idMetVar
<FinAtributoMetodo>

<FinAtributoMetodo> ::= <DeclaracionVariableMultiple>
<InicializacionOpcional> ; | <ArgsFormales> <Bloque>

<EncabezadoMetodo> ::= <VisibilidadOpcional>
<EstaticoOpcional> <TipoMiembro> idMetVar <ArgsFormales> ;

<TipoMiembroSinClase> ::= <TipoPrimitivo> | void

<VisibilidadOpcional> ::= public | private | ε

<InicializacionOpcional> ::= = <Expresion> | ε

<TipoMiembro> ::= <Tipo> | void

<Tipo> ::= <TipoPrimitivo> | idClase

<TipoPrimitivo> ::= boolean | char | int | float

<EstaticoOpcional> ::= static | ε

<ArgsFormales> ::= ( <ListaArgsFormalesOpcional> )

<ListaArgsFormalesOpcional> ::= <ListaArgsFormales> | ε

<ListaArgsFormales> ::= <ArgFormal> <OtroArgFormal>

<OtroArgFormal> ::= , <ListaArgFormales> | ε

<ArgFormal> ::= <Tipo> idMetVar

<Bloque> ::= { <ListaSentencias> }

<ListaSentencias> ::= <Sentencia> <ListaSentencias> | ε
```

```
<Sentencia> ::= ;
<Sentencia> ::= <Expresion>;
<Sentencia> ::= <VarLocal> ;
<Sentencia< ::= <VarLocalConTipoPrimitivo>;
<Sentencia> ::= <Return> ;
<Sentencia> ::= <If>
<Sentencia> ::= <While>
<Sentencia> ::= <Bloque>

<VarLocal> ::= var idMetVar = <ExpresionCompuesta>

<VarLocalConTipoPrimitivo> ::= <TipoPrimitivo> idMetVar
<DeclaracionVariableMultiple> <InicializacionOpcional>

<DeclaracionVariableMultiple> ::= , idClase
<DeclaracionVariableMultiple> | ε


<Return> ::= return <ExpresionOpcional>

<ExpresionOpcional> ::= <Expresion> | ε

<If> ::= if (<Expresion>) <Sentencia> <Else>

<Else> ::= else <Sentencia> | ε

<While> ::= while ( <Expresion> ) <Sentencia>

<Expresion> ::= <ExpresionCompuesta> <InicializacionOpcional>

<ExpresionCompuesta>::=<ExpresionBasica><RExpresionCompuesta>

<RExpresionCompuesta>::=<OperadorBinario><ExpresionBasica>
<RExpresionCompuesta> | ε

<OperadorBinario> ::= || | && | == | != | < | > | <= | >= | +
| - | * | / | %

<ExpresionBasica> ::= <OperadorUnario> <Operando>
```

```
<ExpresionBasica> ::= <Operando>

<OperadorUnario> ::= + | - | !


<Operando> ::= <Literal>
<Operando> ::= <Acceso>

<Literal> ::= null | true | false | intLiteral | charLiteral |
stringLiteral | floatLiteral

<Acceso> ::= <Primario> <EncadenadoOpcional>

<Primario> ::= <AccesoThis>
<Primario> ::= <AccesoMetVar>
<Primario> ::= <AccesoConstructor>
<Primario> ::= <AccesoMetodoEstatico>
<Primario> ::= <ExpresionParentizada>

<AccesoThis> ::= this

<AccesoConstructor> ::= new idClase <GenericidadOpcional>
<ArgsActuales>

<AccesoMetVar> ::= idMetVar <ArgsActualesOpcional>

<ArgsActualesOpcionales> ::= <ArgsActuales> | ε

<ExpresionParentizada> ::= ( <Expresion> )

<AccesoMetodoEstatico> ::= idClase . idMetVar <ArgsActuales>

<ArgsActuales> ::= ( <ListaExpsOpcional> )

<ListaExpsOpcional> ::= <ListaExps> | ε

<ListaExps> ::= <Expresion> <ContinuaListaExps>

<ContinuaListaExps> ::= , <ListaExps> | ε
```

```
<EncadenadoOpcional>  ::= .idMetVar <ArgsActualesOpcionales>
<EncadenadoOpcional> | ε
```

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

Se le suman los reconocibles por 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

Un error que no es reconocible y no se pudo encontrar la razón es la siguiente :

```
class Class {
    void m1() {
          public int x = 3;
    }
}
```

Dado este código el analizador muestra como error al primer paréntesis, en vez de al modificador public, sin embargo si se hace

```
class Class {
    void m1() {
```

```
            int y;
            public int x = 3;
        }
    }
```

Ahi si marca el error correctamente, el modificador *public*, puede que se trate de un problema de la implementación avanzada.

Decisiones de diseño

Si bien el código en su mayoría representa las producciones de la gramática, esto no se cumple en el caso de poder declarar una variable con tipo clase, esto porque en la gramática generaría una ambigüedad difícil de factorizar, entonces se optó por usar una especie de flag en la producción de sentencia, Sentencia(), y si ve que se está tratando de un idClase primero revisa si este es o no una declaración de variables.

Existen dudas sobre la implementación de los *follow sets* de la gramática por lo que puede que existan casos donde abarquen de más o restringen innecesariamente.

Por último se decidió mostrar todas las combinaciones posibles que esperaba el analizador en el caso de lanzar una excepción, esto es preguntando a todos los primeros que no se pudieron capturar, puede que quede un poco verborragia la excepción pero no escatima en información.

Clases utilizadas

Se utilizaron en total 7 clases :

- **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.

- **Main**, encargada de mostrar los errores o ejecución exitosa, tomar los argumentos de consola , crear el manejador de archivo, analizador léxico y sintáctico.

Logros intentados

- Variables locales Clásicas
- Entrega Anticipada Sintáctica
- Implementación avanzada
- Imbatibilidad Sintáctica
- Genericidad
- Visibilidad Extendida
- Floats! - Sintacticos
- Atributos Inicializados

De la etapa previa se corrigieron errores relacionados algunos logros con el objetivo de conseguirlos, estos fueron :

- Columnas
- Reporte de Error Elegante

Otros cambios

Se agrego la palabra reservada float al analizador léxico debido a que necesitamos capturar si se declara un tipo de esta variable al intentar el logro de float, al igual que la palabra reservada private para el logro de visibilidad extendida.

Corrección acerca de la convención de tomar un intLiteral de 9 dígitos, antes se leía un intLiteral hasta 9 dígitos y si había más dígitos pertenecen a otro int o float literal, ahora por retroalimentación de la cátedra se implementó que sea un error léxico encontrar un intLiteral con más de 9 dígitos.

Por último antes el analizador léxico tomaba a classId como dos tokens distintos, uno era la palabra reservada *class* con id *rw_class* y a *Id* como un *idClase*. Ahora se cambió para respetar como hace este chequeo Java, donde se permite tener un *idMetVar* con lexema *classId* , entonces es un solo token identificador, así con todas las palabras reservadas.