Informe ETAPA 2 Compiladores e Intérpretes.

Alumno: Simón E. Fredes Hadad.

Compilación y Ejecución del Código Fuente en IntelliJ IDEA

Para compilar y ejecutar el código fuente del compilador de MiniJava en IntelliJ IDEA, sigue los pasos detallados a continuación:

1. Configuración del Proyecto en IntelliJ IDEA

- **Abrir IntelliJ IDEA**: Inicia IntelliJ IDEA en tu computadora.
- **Crear un Nuevo Proyecto o Importar**:
    - Si es un nuevo proyecto: Ve a `File > New > Project`, selecciona `Java`, configura el SDK (JDK 20 o el que estés utilizando), y luego presiona `Next`. Asigna un nombre y una ubicación al proyecto, y presiona `Finish`.
    - Si ya tienes un proyecto: Ve a `File > Open`, y selecciona la carpeta del proyecto existente con el código fuente del compilador de MiniJava.

2. Configurar el SDK

- Asegúrate de que el proyecto esté configurado con el SDK correcto. Ve a `File > Project Structure > Project`, y selecciona la versión del JDK que deseas usar (por ejemplo, JDK 20).

3. Compilación del Código

- **Compilar Todo el Proyecto**: Para compilar el proyecto completo, ve al menú y selecciona `Build > Build Project` (o usa el atajo `Ctrl + F9`).
- IntelliJ IDEA compilará todos los archivos de código fuente y generará los archivos `.class` en el directorio `out/production/Main`.

4. Ejecución del Código

- Para ejecutar los testers correspondientes al proyecto, ve a `Run > Run 'TesterDeCasosConErrores' o 'TesterDeCasosSinErrores'` (o usa el atajo `Shift + F10`).
- IntelliJ IDEA ejecutará la aplicación utilizando la configuración que especificaste y mostrará los resultados en la consola de salida.

5. Verificación de Errores y Depuración

- Si ocurren errores durante la compilación o ejecución, IntelliJ IDEA mostrará mensajes detallados en la consola de errores. Esto se debe a que se hizo uso de System.out como salida ya que así lo especificaba el enunciado.

- Puedes usar la herramienta de depuración (`Debug > Debug 'NombreDeConfiguración'` o `Shift + F9`) para ejecutar el código paso a paso y encontrar errores o verificar la lógica del programa.

GRAMÁTICA ORIGINAL, CON MODIFICACIONES.

<Inicial> ::= <ListaClases>
<ListaClases> ::= <Clase> <ListaClases> | ε
<Clase> ::= class idClase <HerenciaOpcional> { <ListaMiembros> }
<HerenciaOpcional> ::= extends idClase | ε
<ListaMiembros> ::= <Miembro> <ListaMiembros> | ε
<Miembro> ::= <EstaticoOpcional> <TipoMiemrbo> idMetVar
<RestoAtributoMetodo> **|** <Constructor>

~~<Miembro> ::= <Atributo> | <Metodo> | <Constructor>~~

Originalmente, el no terminal “miembro” poseía una ambigüedad, por lo tanto, se factoriza. Lo que se realizó fue una factorización por medio de obtener la intersección de los Primeros(Atributo) y Primeros(Metodo). Luego, se obtuvo el factor común y se dió lugar a la creación de “RestoAtributoMetodo”. Por ultimo, se eliminó el no terminal “Atributo” y “Metodo” ya que, quedaron obsoletos.

<RestoAtributoMetodo>:= ; | <ArgsFormales> <Bloque>

~~<Atributo> ::= <EstaticoOpcional> <TipoMiemrbo> idMetVar ;~~
~~<Metodo> ::= <EstaticoOpcional> <TipoMiembro> idMetVar <ArgsFormales> <Bloque>~~

<Constructor> ::= public idClase <ArgsFormales> <Bloque>
<TipoMiembro> ::= <Tipo> | void
<Tipo> ::= <TipoPrimitivo> | idClase
<TipoPrimitivo> ::= boolean | char | int
<EstaticoOpcional> ::= static | ε
<ArgsFormales> ::= (<ListaArgsFormalesOpcional>)
<ListaArgsFormalesOpcional>::= <ListaArgsFormales>| ε

<ListaArgsFormales> ::= <ArgFormal><RestoListaArgFormal>
<RestoListaArgFormal> ::= , <ListaArgsFormales> | ε

<ArgFormal> ::= <Tipo> idMetVar

<Bloque> ::= { <ListaSentencias> }

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

<Sentencia> ::= ;
~~<Sentencia> ::= <Asignacion> ;~~
~~<Sentencia> ::= <Llamada> ;~~
<Sentencia> ::= <Expresion> ;
<Sentencia> ::= <VarLocal> ;
<Sentencia> ::= <Return> ;
<Sentencia> ::= <Break> ;
<Sentencia> ::= <If>
<Sentencia> ::= <While>
<Sentencia> ::= <Switch>
<Sentencia> ::= <Bloque>

~~<Asignacion> ::= <Expresion>~~
~~<Llamada> ::= <Expresion>~~
Lo que sucedió aquí, es que la intersección de los primeros de Sentencia no era vacía, tanto "Asignación" como "Llamada" compartian Primeros. Se factorizó creando la sentencia: Sentencia::= Expresión, que engloba el factor comun de "Asignación" y "Llamada". Por ultimo, se eliminó los no-terminales ya que no se utilizaban (quedaron obsoletos).

<VarLocal> ::= var idMetVar = <ExpresionCompuesta>
<Return> ::= return <ExpresionOpcional>
<Break> ::= break
<ExpresionOpcional> ::= <Expresion> | ε
~~<If> ::= if (<Expresion>) <Sentencia>~~
~~<If> ::= if (<Expresion>) <Sentencia> else <Sentencia>~~
<If> ::= if (<Expresion>) <Sentencia><RestoIF>
<RestoIF>::= ε | else<Sentencia>
Lo que sucedió en la sentencia "IF" es que producía ambiguedad. Esto se debe a que en la sentencia original (que está tachada) se repetia el factor comun
if (<Expresion>) <Sentencia>. Lo que se hizo fue crear una nueva sentencia y se factorizó, dando lugar al no-terminal "RestoIF". Por ultimo, se eliminó la sentencia que producía ambiguedad.

<While> ::= while (<Expresion>) <Sentencia>
<Switch> ::= switch (<Expresion>) { <ListaSentenciasSwitch> }
<ListaSentenciasSwitch> ::= <SentenciaSwitch> <ListaSentenciasSwitch> | ε
<SentenciaSwitch> ::= case <LiteralPrimitivo> : <SentenciaOpcional>
<SentenciaSwitch> ::= default : <Sentencia>
<SentenciaOpcional>::= <Sentencia> | ε

~~<Expresion> ::= <ExpresionCompuesta> <OperadorAsignacion> <ExpresionCompuesta>~~
~~<Expresion> ::= <ExpresionCompuesta>~~

<Expresion>::= <ExpresionCompuesta> <ExpresionAUX>
<ExpresionAUX>::= <OperadorAsignacion> <ExpresionCompuesta> | ε

La sentencia Expresión también producía ambiguedad, ya que, si comparamos los Primeros(ExpresiónCompuesta) intersección Primeros(ExpresionCompuesta) vemos que son iguales, por lo tanto, hay ambiguedad y debemos resolverla para conservar la caracteristica de la gramatica LL(1). Lo que se hizo fue, como en pasos anteriores, obtener el factor común ("ExpresionCompuesta") y crear "ExpresiónAUX", cuyo arbol de derivación engloba el **OperadorAsignaciónExpresionCompuesta** ó **vacío**. Por ultimo, se eliminan las sentencias "Expresión" que producian ambiguedad.

<OperadorAsignacion> ::= = | += | -=
~~<ExpresionCompuesta> ::= <ExpresionCompuesta> <OperadorBinario> <ExpresionBasica>~~
<ExpresionCompuesta> ::= <ExpresionBasica><ExpresionCompuestaAUX>
<ExpresionCompuestaAUX>::= <OperadorBinario><ExpresionBasica>
<ExpresionCompuestaAUX> | ε

Lo que sucedió aquí fue que la gramatica contenía una recursión a izquierda. Esto hace que rompa las reglas de una gramatica LL(1). Se solucionó realizando una factorización. Lo primero que se hizo fue crear "ExpresionCompuestaAUX". Lo que se hace aquí es no perder el poder de producción de la gramatica y se logra también eliminar la recursión a izquierda.

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

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

<OperadorUnario> ::= + | - | !

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

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

<Acceso> ::= <Primario> <EncadenadoOpcional>
<Primario> ::= <AccesoThis>
<Primario> ::= <AccesoVarMet>
<Primario> ::= <AccesoConstructor>
<Primario> ::= <AccesoMetodoEstatico>
<Primario> ::= <ExpresionParentizada>
El problema identificado en "Primario" fue que su intersección de *primeros* no era null. Esto se debía a que AccesoVar n AccesoMetodo = idMetVar. Esto se solucionó creando un nuevo no-terminal llamado **"AccesoVarMet"** que engloba el factor común idMetVar y además, tiene una regla de producción que se llama **AccesoVarMetAUX** que, según se necesite puede derivar en reglas que coinciden con AccesoMetodo o en AccesoVar (vacio).

<AccesoThis> ::= this
<AccesoVarMet>::= idMetVar <AccesoVarMetAUX>
<AccesoVarMetAUX>::= <ArgsActuales> | ε
~~<AccesoVar> ::= idMetVar~~
<AccesoConstructor> ::= new idClase <ArgsActuales>
<ExpresionParentizada> ::= (<Expresion>)
~~<AccesoMetodo> ::= idMetVar <ArgsActuales>~~
<AccesoMetodoEstatico> ::= idClase . idMetVar <ArgsActuales>
<ArgsActuales> ::= (<ListaExpsOpcional>)
<ListaExpsOpcional> ::= <ListaExps> | ε

~~<ListaExps> ::= <Expresion>~~
~~<ListaExps> ::= <Expresion> , <ListaExps>~~

Al igual que en ocasiones anteriores, la intersección de las dos posibles ramas de derivación de ListaExps no son vacía. Sino que comparten "Expresión". Lo que se realizó fue crear un no-terminall ListaEXPSAUX y que deriva según corresponda, si vacío (primer caso) ó ,ListaExps (segundo caso).

<ListaExps> ::= <Expresion> <ListaExpsAUX>
<ListaExpsAUX> ::= ε | , <ListaExps>

~~<EncadenadoOpcional> ::= <VarEncadenada> | <MetodoEncadenado> | ε~~

~~<EncadenadoOpcional> ::= idMetVar <EncadenadoOpcional> |~~
~~idMetVar <ArgsActuales> <EncadenadoOpcional> | ϵ~~

<EncadenadoOpcional>::=. idMetVar<EncadenadoOpcionalAUX> | ~~ϵ~~
<EncadenadoOpcionalAUX>::= <EncadenadoOpcional> | <ArgsActuales> <EncadenadoOpcional>

Lo que sucedió aquí es que La intersección de los OR no es vacía, debido a que los PRIMEROS de VarEncadenada y MetodoEncadenado son idMetVar. Por lo tanto, se factorizó obteniendo los primeros en común, para luego crear el noTerminal EncadenadoOpcionalAUX. Esto, como en casos anteriores, deriva las dos posibles ramas de la gramatica original de EncadenadoOpcional, logrando que la intersección del ***or*** sea vacía, y manteniendo el poder expresivo de la gramatica. Por ultimo, eliminamos VarEncadenada y MetodoEncadenado, ya que no se utilizan (quedaron obsoletos).

~~<VarEncadenada> ::= . idMetVar <EncadenadoOpcional>~~
~~<MetodoEncadenado> ::= . idMetVar <ArgsActuales> <EncadenadoOpcional>~~