Generación de una representación intermedia

(número de línea, tipo, salto verdadero, salto falso, parámetros)

```
inicio-programa
    leer numeroDeElementos                  (0,Leer, 1, 1, [numeroDeElementos])
    promedio=0                              (1,Asignacion,2,2,[prom,0])
    i=0                                     (2,Asignacion,3,3,[i,0])

    mientras (i<numeroDeElementos)          (3,Comparacion,4,7,[i,<,numeroDeElementos])
        leer elemento                       (4,Leer,5,5,[elemento])
        promedi=promedio+elemento           (5,Asignacion,6,6,[prom,prom,+,elemento])
        i=i+1                               (6,Asignacion,3,3,[i,i,+,1])
    fin-mientras

    promedio=promedio/numeroDeElementos     (7,Asignacion,8,8,[promedio,promedio,/,numeroDeElementos])
    escribir "El promedio es ",promedio     (8,Escribir,9,9,["El promedio es ", promedio])
fin-programa                                (9,-1,-1,FinPrograma,[])
```

```java
public class Tupla {
    protected int saltoVerdadero, saltoFalso;

    public Tupla(int sv, int sf) {
        saltoVerdadero = sv;
        saltoFalso = sf;
    }

    public void setSaltoVerdadero(int sv) {
        saltoVerdadero = sv;
    }

    public int getSaltoVerdadero() {
        return saltoVerdadero;
    }

    public void setSaltoFalso(int sf) {
        saltoFalso = sf;
    }

    public int getSaltoFalso() {
        return saltoFalso;
    }

    public String toString() {
        return this.getClass().getName()+ ", " +
                saltoVerdadero + ", " + saltoFalso;
    }
}
```

La clase **Tupla** contiene los parámetros que sea requieren para cada enunciado o instrucción y los saltos verdadero y falso que indican donde continuará el flujo después de esta tupla.

Para representar los diferentes tipos de tuplas, se define una jerarquía de clases con la superclase **Tupla** y las subclases **Leer**, **Escribir**, **Asignación**, **Comparación** y **FinPrograma**.

```java
public class Escribir extends Tupla {
    Token cadena, variable;

    public Escribir(Token variableCadena, int sv, int sf) {
        super(sv, sf);

        if (variableCadena.getTipo().getNombre().equals(TipoToken.CADENA))
            cadena = variableCadena;
        else
            variable = variableCadena;
    }

    public Escribir(Token cadena, Token variable, int sv, int sf) {
        super(sv, sf);
        this.cadena = cadena;
        this.variable = variable;
    }

    public String toString() {
        if (variable == null)
            return  "( " + super.toString() + ", [ " + cadena + " ] )";

        if (cadena == null)
            return  "( " + super.toString() + ", [ " + variable + " ] )";

        return  "( " + super.toString() + ", [ " + cadena + ", " + variable + " ] )";
    }
}
```

La clase **Escribir**, contiene constructores que permiten crear una tupla para el enunciado escribir, considerando las tres opciones que hay para utilizar este enunciado.

```java
public class Leer extends Tupla {
    Token variable;

    public Leer(Token variable, int sv, int sf) {
        super(sv, sf);
        this.variable = variable;
    }

    public String toString() {
        return  "( " + super.toString() + ", [ " + variable + " ] )";
    }
}
```

La clase **Leer**, contiene un constructor que permite crear una tupla para el enunciado leer.

```java
public class Asignacion extends Tupla {
    Token variable, valor1, valor2, operador;

    public Asignacion(Token variable, Token valor, int sv, int sf) {
        super(sv, sf);
        this.variable = variable;
        this.valor1 = valor;
    }

    public Asignacion(Token variable, Token valor1, Token operador, Token valor2, int sv, int sf) {
        super(sv, sf);
        this.variable = variable;
        this.valor1 = valor1;
        this.valor2 = valor2;
        this.operador = operador;
    }

    public String toString() {
        if (operador == null)
            return  "( " + super.toString() + ", [ \"" + variable + ", " + valor1 + "\" ] )";
        else
            return  "( " + super.toString() + ", [ " + variable + ", " + valor1 + ", " + operador +
                    ", " + valor2 + " ] ) ";
    }
}
```

La clase **Asignacion**, contiene constructores que permiten crear una tupla para un enunciado de asignación, considerando las dos opciones que hay para utilizar este enunciado.

```java
public class Comparacion extends Tupla {
    Token valor1, valor2, operador;

    public Comparacion(Token valor1, Token operador, Token valor2, int sv, int sf) {
        super(sv, sf);
        this.valor1 = valor1;
        this.valor2 = valor2;
        this.operador = operador;
    }

    public String toString() {
        return  "( " + super.toString() + ", [ " + valor1 + ", " + operador + ", " +
                valor2 + " ] ) ";
    }
}
```

La clase **Comparacion**, contiene un constructor que permiten crear una tupla para la comparación que es parte de los enunciados si-entonces y mientras.

```java
public class FinPrograma extends Tupla {
    public FinPrograma() {
        super(-1, -1);
    }

    public String toString() {
        return "( " + super.toString() + ", [ ], " + " )";
    }
}
```

La clase **FinPrograma**, contiene un constructor que permiten crear una tupla para indicar el fin de un programa. Esta tupla se agrega al final de las tuplas generadas y sirve para indicar que ya no hay más tuplas.

```java
public class PseudoGenerador {
    private ArrayList<Tupla> tuplas = new ArrayList<>();
    ArrayList<Token> tokens;

    public PseudoGenerador(ArrayList<Token> tokens) {
        this.tokens = tokens;
    }

    public void crearTuplaAsignacion(int indiceInicial, int indiceFinal) {
        if (indiceFinal - indiceInicial == 3)
            tuplas.add(new Asignacion(tokens.get(indiceInicial),
                                      tokens.get(indiceInicial+2),
                                      tuplas.size()+1, tuplas.size()+1));
        else if (indiceFinal - indiceInicial == 5)
            tuplas.add(new Asignacion(tokens.get(indiceInicial),
                                      tokens.get(indiceInicial+2),
                                      tokens.get(indiceInicial+3),
                                      tokens.get(indiceInicial+4),
                                      tuplas.size()+1, tuplas.size()+1));
    }
```

Para la generación de las tuplas se implementó la clase **PseudoGenerador**.

Se llevará a cabo el análisis sintáctico y la generación de la representación intermedia como una sola tarea.

```
public void crearTuplaLeer(int indiceInicial) {
    tuplas.add(new Leer(tokens.get(indiceInicial),
                        tuplas.size()+1, tuplas.size()+1));
}

public void crearTuplaEscribir(int indiceInicial, int indiceFinal) {
    if (indiceFinal - indiceInicial == 1)
        tuplas.add(new Escribir(tokens.get(indiceInicial),
                                tuplas.size()+1, tuplas.size()+1));
    else if (indiceFinal - indiceInicial == 3)
        tuplas.add(new Escribir(tokens.get(indiceInicial),
                                tokens.get(indiceInicial+3),
                                tuplas.size()+1, tuplas.size()+1));
}
```

La generación de la representación intermedia, es decir, las tuplas, se lleva a cabo a través de los métodos **crearTuplaLeer**, **crearTuplaEscribir**, **crearTuplaAsignación**, **crearTuplaComparación**, **crearTuplaFinPrograma**.

```
public void crearTuplaComparacion(int indiceInicial) {
    tuplas.add(new Comparacion(tokens.get(indiceInicial),
                               tokens.get(indiceInicial+1),
                               tokens.get(indiceInicial+2),
                               tuplas.size()+1, tuplas.size()+1));
}

public void crearTuplaFinPrograma() {
    tuplas.add(new FinPrograma());
}

public void conectarSi(int tuplaInicial) {
    int tuplaFinal = tuplas.size()-1;

    if (tuplaInicial >= tuplas.size() || tuplaInicial >= tuplaFinal)
        return;

    tuplas.get(tuplaInicial).setSaltoFalso(tuplaFinal+1);
}
```

Estos métodos van generando de manera secuencial las tuplas correspondientes con cada tipo de enunciado y para llevar a cabo la conexión de las tuplas **Comparacion** (que pueden provenir de un enunciado si-entonces o de un enunciado mientras) se incluyeron los métodos **conectarSi** y **conectarMientras**.

```java
    public void conectarMientras(int tuplaInicial) {
        int tuplaFinal = tuplas.size()-1;

        if (tuplaInicial >= tuplas.size() || tuplaInicial >= tuplaFinal)
            return;

        tuplas.get(tuplaInicial).setSaltoFalso(tuplaFinal + 1);
        tuplas.get(tuplaFinal).setSaltoVerdadero(tuplaInicial);
        tuplas.get(tuplaFinal).setSaltoFalso(tuplaInicial);

        for (int i = tuplaFinal; i > tuplaInicial; i--) {
            Tupla t = tuplas.get(i);

            if (t instanceof Comparacion && t.getSaltoFalso() == tuplaFinal + 1)
                t.setSaltoFalso(tuplaInicial);
        }
    }

    public ArrayList<Tupla> getTuplas() {
        return tuplas;
    }
}
```

```java
public class PseudoParser  {
    private ArrayList<Token> tokens;
    private int indiceToken = 0;
    private SyntaxException ex;

    private TablaSimbolos ts;
    private TipoIncorporado real;
    private PseudoGenerador generador;

    public PseudoParser(TablaSimbolos ts, PseudoGenerador generador) {
        this.ts = ts;
        this.generador = generador;
    }

    public void analizar(PseudoLexer lexer) throws SyntaxException {
        tokens = lexer.getTokens();
        real = new TipoIncorporado("real");
        ts.definir(real);

        if (Programa()) {
            if (indiceToken == tokens.size()) {
                System.out.println("\nLa sintaxis del programa es correcta");
                return;
            }
        }

        throw ex;
    }
```

La clase **PseudoParser** se modifica para incorporar el generador.

```java
// <Programa> -> inicio-programa <Enunciados> fin-programa
private boolean Programa() {
    if (match(TipoToken.INICIOPROGRAMA))
        if (Enunciados())
            if (match(TipoToken.FINPROGRAMA)) {
                generador.crearTuplaFinPrograma();
                return true;
            }

    return false;
}

// <Asignacion> -> VARIABLE = <Expresion>
private boolean Asignacion() {
    int indiceAux = indiceToken;

    if (match(TipoToken.VARIABLE))
        if (match(TipoToken.IGUAL))
            if (Expresion()) {
                generador.crearTuplaAsignacion(indiceAux, indiceToken);
                return true;
            }

    indiceToken = indiceAux;
    return false;
}
```

```java
// <Leer> -> leer VARIABLE
private boolean Leer() {
    int indiceAux = indiceToken;

    if (match(TipoToken.LEER))
        if (match(TipoToken.VARIABLE)) {
            generador.crearTuplaLeer(indiceAux+1);
            return true;
        }


    indiceToken = indiceAux;
    return false;
}
```

```java
// <Escribir> -> escribir CADENA , VARIABLE | escribir CADENA | escribir VARIABLE
private boolean Escribir() {
    int indiceAux = indiceToken;

    if (match(TipoToken.ESCRIBIR))
        if (match(TipoToken.CADENA))
            if (match(TipoToken.COMA))
                if (match(TipoToken.VARIABLE)) {
                    generador.crearTuplaEscribir(indiceAux+1, indiceToken);
                    return true;
                }

    indiceToken = indiceAux;

    if (match(TipoToken.ESCRIBIR))
        if (match(TipoToken.CADENA)) {
            generador.crearTuplaEscribir(indiceAux+1, indiceToken);
            return true;
        }

    indiceToken = indiceAux;

    if (match(TipoToken.ESCRIBIR))
        if (match(TipoToken.VARIABLE)) {
            generador.crearTuplaEscribir(indiceAux+1, indiceToken);
            return true;
        }

    indiceToken = indiceAux;
    return false;
}
```

```java
// <Si> -> si <Comparacion> entonces <Enunciados> fin-si
private boolean Si() {
    int indiceAux = indiceToken;
    int indiceTupla = generador.getTuplas().size();

    if (match(TipoToken.SI))
        if (Comparacion())
            if (match(TipoToken.ENTONCES))
                if (Enunciados())
                    if (match(TipoToken.FINSI)) {
                        generador.conectarSi(indiceTupla);
                        return true;
                    }

    indiceToken = indiceAux;
    return false;
}
```

```java
// <Mientras> -> mientras <Comparacion> <Enunciados> fin-mientras
private boolean Mientras() {
    int indiceAux = indiceToken;
    int indiceTupla = generador.getTuplas().size();

    if (match(TipoToken.MIENTRAS))
        if (Comparacion())
            if (Enunciados())
                if (match(TipoToken.FINMIENTRAS)) {
                    generador.conectarMientras(indiceTupla);
                    return true;
                }


    indiceToken = indiceAux;
    return false;
}
```

```java
// <Comparacion> -> ( <Valor> <Operador relacional> <Valor> )
private boolean Comparacion() {
    int indiceAux = indiceToken;

    if (match(TipoToken.PARENTESISIZQ))
        if (Valor())
            if (match(TipoToken.OPRELACIONAL))
                if (Valor())
                    if (match(TipoToken.PARENTESISDER)) {
                        generador.crearTuplaComparacion(indiceAux+1);
                        return true;
                    }

    indiceToken = indiceAux;
    return false;
}
```

```java
public class PruebaTuplas {
    public static void main(String[] arg) throws LexicalException, SyntaxException {
        String entrada = leerPrograma("/.../ejemplo.alg");
        PseudoLexer lexer = new PseudoLexer();
        lexer.analizar(entrada);

        System.out.println("*** Análisis léxico ***\n");

        for (Token t: lexer.getTokens()) {
            System.out.println(t);
        }

        System.out.println("\n*** Análisis sintáctico ***\n");

        TablaSimbolos ts = new TablaSimbolos();
        PseudoGenerador generador = new PseudoGenerador(lexer.getTokens());
        PseudoParser parser = new PseudoParser(ts, generador);
        parser.analizar(lexer);

        System.out.println("\n*** Tabla de símbolos ***\n");

        for (Simbolo s: ts.getSimbolos())
            System.out.println(s);

        System.out.println("\n*** Tuplas generadas ***\n");

        for (Tupla t: generador.getTuplas()) {
            System.out.println(t);
        }
    }
```

Para probar el funcionamiento del **PseudoParser** (con generación de representación intermedia) se implementa una clase de prueba, la cual extiende la clase que se implementó para probar las fases anteriores del intérprete.

```
inicio-programa
    leer numeroDeElementos
    promedio = 0
    i = 0

    mientras (i < numeroDeElementos)
        leer elemento
        promedio = promedio + elemento
        i = i + 1
    fin-mientras

    promedio = promedio / numeroDeElementos
    escribir "El promedio es ", promedio
fin-programa
```

```
*** Tuplas generadas ***

( Leer, 1, 1, [ <VARIABLE, numeroDeElementos> ] )
( Asignacion, 2, 2, [ "<VARIABLE, promedio>, <NUMERO, 0>" ] )
( Asignacion, 3, 3, [ "<VARIABLE, i>, <NUMERO, 0>" ] )
( Comparacion, 4, 7, [ <VARIABLE, i>, <OPRELACIONAL, <>, <VARIABLE, numeroDeElementos> ] )
( Leer, 5, 5, [ <VARIABLE, elemento> ] )
( Asignacion, 6, 6, [ <VARIABLE, promedio>, <VARIABLE, promedio>, <OPARITMETICO, +>, <VARIABLE, elemento> ] )
( Asignacion, 3, 3, [ <VARIABLE, i>, <VARIABLE, i>, <OPARITMETICO, +>, <NUMERO, 1> ] )
( Asignacion, 8, 8, [ <VARIABLE, promedio>, <VARIABLE, promedio>, <OPARITMETICO, />, <VARIABLE, numeroDeElementos> ] )
( Escribir, 9, 9, [ <CADENA, "El promedio es ">, <VARIABLE, promedio> ] )
( FinPrograma, -1, -1, [ ],  )
```

Ejercicio

- En ejercicios de capítulos anteriores se extendió el lenguaje de pseudocódigo para permitir la utilización del enunciado repite como estructura de iteración, por ejemplo:

```
repite (i, 0, numeroDeElementos)

    leer elemento

    promedio = promedio + elemento

fin-repite
```

- El enunciado **repite** requiere tres elementos: una variable que se utiliza como índice, el valor inicial y el valor final para la variable índice.

- ¿Cómo se podría extender el generador de tuplas y/o parser si el lenguaje de pseudocódigo incluyera el enunciado **repite** ?

- Modifica la clase **PseudoParser** y **PseudoGenerador** y las clases relacionadas que se requieran para generar las tuplas correspondientes.