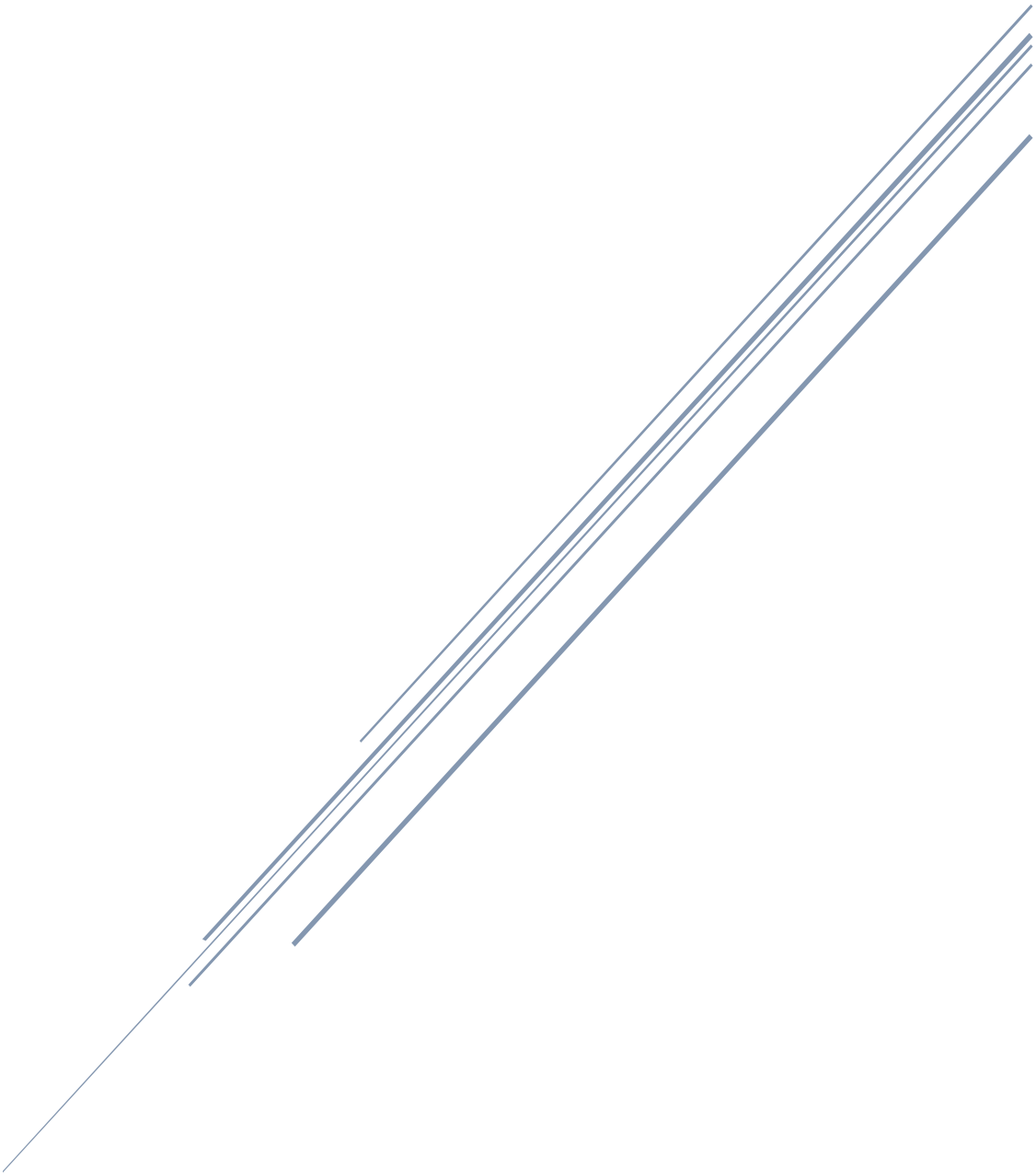
PROCESADORES DEL LENGUAJE

ENTREGA 2



Francisco Jesús Beltrán Moreno

ÍNDICE

Descripción de la gramática EBNF ampliada

CompilationUnit : (ImportClause)* TintoDecl
ImportClause : **import identifier semicolon**
TintoDecl : LibraryDecl | NativeDecl
LibraryDecl : **library identifier lbrace** (Function)* **rbrace**
Function : Access FunctionType **identifier** ArgumentDecl FunctionBody
NativeDecl : **library identifier lbrace** (NativeFunction)* **rbrace**
NativeFunction : Access FunctionType **identifier** ArgumentDecl **semicolon**
Access : **public** | **private**
FunctionType : Type | **void**
Type : **int** | **char** | **boolean**
ArgumentDecl : **lparen** (Argument (**comma** Argument)*)? **rparen**
Argument : Type **identifier**
FunctionBody : **lbrace** (Statement)* **rbrace**

Stm : Decl **semicolon** | IdStm **semicolon** | IfStm | SwitchStm | WhileStm | DoWhileStm | ForStm | BreakStm | ContinueStm | ReturnStm | NoStm | BlockStm
Decl : Type **identifier** Assignement (**comma identifier** Assignement)*
IdStm : **identifier** (Assignement | MethodCall | **dot identifier** MethodCall)
IfStm ::= **if lparen** Expr **rparen** Stm (**else** Stm)?
SwitchStm : **switch lparen** Expr **rparen lbrace** (CaseClause | DefaultClause)* **rbrace**
CaseClause : **case integer_literal colon** (Stm)*
DefaultClause : **default colon** (Stm)*
WhileStm : **while lparen** Expr **rparen** Stm
DoWhileStm : **do** Stm **while lparen** Expr **rparen semicolon**
ForStm : **for lparen** (ForInit)? **semicolon** (Expr)? **semicolon** (IdStmList)? **rparen** Stm
ForInit ::= (Decl | IdStmList)
IdStmList : IdStm (**comma** IdStm)*
BreakStm : **break semicolon**
ContinueStm : **continue semicolon**
ReturnStm : **return** (Expr)? **semicolon**
NoStm : **semicolon**
BlockStm : **lbrace** (Stm)* **rbrace**

Expr : AndExpr (**or** AndExpr)*
AndExpr : RelExpr (**and** RelExpr)*
RelExpr : SumExpr (RelOp SumExpr)?
RelOp : (**eq** | **ne** | **gt** | **ge** | **lt** | **le**)
SumExpr : UnOp ProdExpr (SumOp ProdExpr)*
UnOp : (**not** | **minus** | **plus**)?
SumOp : (**minus** | **plus**)
ProdExpr : Factor (MultOp Factor)*
MultOp : (**prod** | **div** | **mod**)
Factor : (Literal | Reference | **lparen** Expr **rparen**)
Literal : (**integer_literal** | **char_literal** | **true** | **false**)
Reference : **identifier** (FunctionCall | **dot identifier** FunctionCall)?
FunctionCall : **lparen** (Expr (**comma** Expr)*)? **rparen**

Descripción de la gramática BNF equivalente

CompilationUnit ::= ImportClauseList TintoDecl	**import**, **library, native**
ImportClauseList ::= ImportClause ImportClauseList	**import**
ImportClauseList ::= **lambda**	**library**
ImportClause ::= **import identifier semicolon**	**import**
TintoDecl ::= LibraryDecl	**library**
TintoDecl ::= NativeDecl	**native**
LibraryDecl ::= **library identifier lbrace** FunctionList **rbrace**	**library**
FunctionList ::= Function FunctionList	**public**, **private**
FunctionList ::= **lambda**	**rbrace**
Function ::= Access FunctionType **identifier** ArgumentDecl FunctionBody	**public**, **private**
NativeDecl ::= **native identifier lbrace** NativeFunctionList **rbrace**	**native**
NativeFunctionList ::= NativeFunction NativeFunctionList	**public**, **private**
NativeFunctionList ::= **lambda**	**rbrace**
NativeFunction ::= Access FunctionType **identifier** ArgumentDecl **semicolon**	**public**, **private**
Access ::= **public**	**public**
Access ::= **private**	**private**
FunctionType ::= Type	**int**, **char**, **boolean**
FunctionType ::= **void**	**void**
Type ::= **int**	**int**
Type ::= **char**	**char**
Type ::= **boolean**	**boolean**
ArgumentDecl := **lparen** ArgumentList **rparen**	**lparen**
ArgumentList ::= Argument MoreArguments	**int**, **char**, **boolean**
ArgumentList ::= **lambda**	**rparen**
MoreArguments ::= **comma** Argument MoreArguments	**comma**
MoreArguments ::= **lambda**	**lparen**
Argument ::= Type **identifier**	**int**, **char**, **boolean**
FunctionBody ::= **lbrace** StatementList **rbrace**	**lbrace**

StatementList ::= Statement StatementList	**int**, **char**, **boolean**, **identifier**, **if**, **while**, **return**, **semicolon**, **lbrace, do, for, switch, break, continue**
StatementList ::= **lambda**	**rbrace**
Statement ::= Decl **semicolon**	**int**, **char**, **boolean**
Statement ::= IdStm **semicolon**	**identifier**
Statement ::= IfStm	**if**
Statement ::= WhileStm	**while**
Statement ::= ReturnStm	**return**
Statement ::= NoStm	**semicolon**
Statement ::= BlockStm	**lbrace**
Statement ::= DoWhileStm	**do**
Statement ::= ForStm	**for**
Statement ::= SwitchStm	**switch**
Statement ::= BreakStm	**break**
Statement ::= ContinueStm	**continue**
Decl ::= Type **identifier** Assignement MoreDecl	**int**, **char**, **boolean**
MoreDecl ::= **comma identifier** Assignement MoreDecl	**comma**
MoreDecl ::= **lambda**	**semicolon**
Assignement ::= **assign** Expr	**assign**
Assignement ::= **lambda**	**comma**, **semicolon**
IfStm ::= **if lparen** Expr **rparen** Statement ElseStm	**if**
ElseStm ::= **else** Statement	**else**
ElseStm ::= **lambda**	**int**, **char**, **boolean**, **identifier**, **if**, **while**, **return**, **semicolon**, **lbrace, do, for, switch, break, continue**
SwitchStm ::= **switch lparen** Expr **rparen lbrace** OptionsSwitch **rbrace**	**switch**
OptionsSwitch ::= **case integer_literal colon** MoreStatement OptionsSwitch	**case**
MoreStatement ::= Statement MoreStatement	**int**, **char**, **boolean**, **identifier**, **if**, **while**, **return**, **semicolon**, **lbrace, do, for, switch, break, continue**
OptionsSwitch ::= **default colon** MoreStatement OptionsSwitch	**default**
MoreStatement ::= **lambda**	**case, default, rbrace**
OptionsSwitch ::= **lambda**	**rbrace**
DoWhileStm ::= **do** Statement **while lparen** Expr **rparen semicolon**	**do**
ForStm ::= **for lparen** ForInit **semicolon** Expr **semicolon** IdStmList **rparen** Statement	**for**
ForInit ::= Decl	**int, char, boolean**
ForInit ::= IdStmList	**identifier**
ForInit ::= **lambda**	**semicolon**
IdStmList ::= IdStm MoreIdStmList	**identifier**
IdStmList ::= **lambda**	**rparen**
MoreIdStmList ::= **comma** IdStm MoreIdStmList	**comma**
MoreIdStmList ::= **lambda**	**rparen**

WhileStm ::= **while lparen** Expr **rparen** Statement	**while**
ReturnStm ::= **return** ReturnExpr **semicolon**	**return**
ReturnExpr ::= Expr	**not**, **minus**, **plus**, **integer_literal**, **char_literal**, **true**, **false**, **identifier**, **lparen**
ReturnExpr ::= **lambda**	**semicolon**
BreakStm ::= **break semicolon**	**break**
Continue ::= **continue semicolon**	**continue**
NoStm ::= **semicolon**	**semicolon**
IdStm ::= **identifier** IdStmContinue	**identifier**
IdStmContinue ::= **assign** Expr	**assign**
IdStmContinue ::= FunctionCall	**lparen**
IdStmContinue ::= **dot identifier** FunctionCall	**dot**
BlockStm ::= **lbrace** StatementList **rbrace**	**lbrace**

Expr ::= AndExpr MoreOrExpr	**not**, **minus**, **plus**, **integer_literal**, **char_literal**, **true**, **false**, **identifier**, **lparen**
MoreOrExpr ::= **or** AndExpr MoreOrExpr	**or**
MoreOrExpr ::= **lambda**	**comma**, **rparen**, **semicolon**
AndExpr ::= RelExpr MoreAndExpr	**not**, **minus**, **plus**, **integer_literal**, **char_literal**, **true**, **false**, **identifier**, **lparen**
MoreAndExpr ::= **and** RelExpr MoreAndExpr	**and**
MoreAndExpr ::= **lambda**	**or**, **comma**, **rparen**, **semicolon**
RelExpr ::= SumExpr MoreRelExpr	**not**, **minus**, **plus**, **integer_literal**, **char_literal**, **true**, **false**, **identifier**, **lparen**
MoreRelExpr ::= RelOp SumExpr	**eq**, **ne**, **gt**, **ge**, **lt**, **le**
MoreRelExpr ::= **lambda**	**and**, **or**, **comma**, **rparen**, **semicolon**
RelOp ::= **eq**	**eq**
RelOp ::= **ne**	**ne**
RelOp ::= **gt**	**gt**
RelOp ::= **ge**	**ge**
RelOp ::= **lt**	**lt**
RelOp ::= **le**	**le**
SumExpr ::= UnOp ProdExpr MoreSumExpr	**not**, **minus**, **plus**, **integer_literal**, **char_literal**, **true**, **false**, **identifier**, **lparen**
MoreSumExpr ::= SumOp ProdExpr MoreSumExpr	**plus**, **minus**
MoreSumExpr ::= **lambda**	**eq**, **ne**, **gt**, **ge**, **lt**, **le**, **and**, **or**, **comma**, **rparen**, **semicolon**
UnOp ::= **not**	**not**
UnOp ::= **minus**	**minus**
UnOp ::= **plus**	**plus**
UnOp ::= **lambda**	**integer_literal**, **char_literal**, **true**, **false**, **identifier**, **lparen**
SumOp ::= **minus**	**minus**
SumOp ::= **plus**	**plus**

ProdExpr ::= Factor MoreProdExpr	**integer_literal**, **char_literal**, **true**, **false**, **identifier**, **lparen**
MoreProdExpr ::= MultOp Factor MoreProdExpr	**prod**, **div**, **mod**
MoreProdExpr ::= **lambda**	**plus**, **minus**, **eq**, **ne**, **gt**, **ge**, **lt**, **le**, **and**, **or**, **comma**, **rparen**, **semicolon**
MultOp ::= **prod**	**prod**
MultOp ::= **div**	**div**
MultOp ::= **mod**	**mod**
Factor ::= Literal	**integer_literal**, **char_literal**, **true**, **false**
Factor ::= Reference	**identifier**
Factor ::= **lparen** Expr **rparen**	**lparen**
Literal ::= **integer_literal**	**integer_literal**
Literal ::= **char_literal**	**char_literal**
Literal ::= **true**	**true**
Literal ::= **false**	**false**
Reference ::= **identifier** ReferenceContinue	**identifier**
ReferenceContinue ::= FunctionCall	**lparen**
ReferenceContinue ::= **dot identifier** FunctionCall	**dot**
ReferenceContinue ::= **lambda**	**prod**, **div**, **mod**, **plus**, **minus**, **eq**, **ne**, **gt**, **ge**, **lt**, **le**, **and**, **or**, **comma**, **rparen**, **semicolon**
FunctionCall ::= **lparen** ExprList **rparen**	**lparen**
ExprList ::= Expr MoreExpr	**integer_literal**, **char_literal**, **true**, **false**, **identifier**, **lparen**, **not**, **minus**, **plus**
ExprList ::= **lambda**	**rparen**
MoreExpr ::= **comma** Expr MoreExpr	**comma**
MoreExpr ::= **lambda**	**rparen**

Descripción de los métodos asociados a los nuevos símbolos de la gramática

```java
/*
 * Analiza el s@mbolo <StatementList>
 * @throws SintaxException
 */
private void parseStatementList() throws SintaxException
{
    // AÑADIDO DO, FOR, SWITCH, BREAK, CONTINUE
    int[] expected = { INT, CHAR, BOOLEAN, IDENTIFIER, IF, WHILE,
                       RETURN, SEMICOLON, LBRACE, RBRACE, DO,
                       FOR, SWITCH, BREAK, CONTINUE };
    switch(nextToken.getKind())
    {
        case INT:
        case CHAR:
        case BOOLEAN:
        case IDENTIFIER:
        case IF:
        case WHILE:
        case RETURN:
        case SEMICOLON:
        case FOR:
        case DO:
        case SWITCH:
        case LBRACE:
            parseStatement();
            parseStatementList();
            break;
        case BREAK:
            parseBreakStm();
            break;
        case CONTINUE:
            parseContinueStm();
            break;
        case RBRACE:
            break;
        default:
            throw new SintaxException(nextToken,expected);
    }
}
```

Modificado el método **parseStatementList()** con los nuevos tipos {**do, for, switch, break, continue**}.

```java
/**
 * Analiza el s@mbolo <Statement>
 * @throws SintaxException
 */
private void parseStatement() throws SintaxException
{
    // AÑADIDO MATCH(SEMICOLON), DO, FOR, SWITCH, BREAK, CONTINUE
    int[] expected = { INT, CHAR, BOOLEAN, IDENTIFIER, IF, WHILE,
                       RETURN, SEMICOLON, LBRACE, DO, FOR, SWITCH,
                       BREAK, CONTINUE };
    switch(nextToken.getKind())
    {
        case INT:
        case CHAR:
        case BOOLEAN:
            parseDecl();
            match(SEMICOLON);
            break;
        case IDENTIFIER:
            parseIdStm();
            match(SEMICOLON);
            break;
        case IF:
            parseIfStm();
            break;
        case WHILE:
            parseWhileStm();
            break;
        case RETURN:
            parseReturnStm();
            break;
        case SEMICOLON:
            parseNoStm();
            break;
        case LBRACE:
            parseBlockStm();
            break;
        case DO:
            parseDoWhileStm();
            break;
        case FOR:
            parseForStm();
            break;
        case SWITCH:
            parseSwitchStm();
            break;
        case BREAK:
            parseBreakStm();
            break;
        case CONTINUE:
            parseContinueStm();
            break;
        default:
            throw new SintaxException(nextToken,expected);
    }
}
```

Modificado el método **parseStatement()** con los nuevos tipos y además la modificación del semicolon {**;**} para poder usarlo en el for.

```java
/**
 * Analiza el símbolo <MoreStatement>
 * @throws SintaxException
 */
private void parseMoreStatement() throws SintaxException {
    // AÑADIDO PARA USAR EN EL SWITCH
    int[] expected = { INT, CHAR, BOOLEAN, IDENTIFIER, IF, WHILE,
                       RETURN, SEMICOLON, LBRACE, DO, FOR, SWITCH,
                       CASE, DEFAULT, RBRACE };

    switch(nextToken.getKind()) {
        case INT:
        case CHAR:
        case BOOLEAN:
        case IDENTIFIER:
        case IF:
        case WHILE:
        case RETURN:
        case SEMICOLON:
        case LBRACE:
        case FOR:
        case DO:
        case SWITCH:
        case BREAK:
        case CONTINUE:
            parseStatement();
            parseMoreStatement();
            break;
        case CASE:
        case DEFAULT:
        case RBRACE:
            break;
        default:
            throw new SintaxException(nextToken,expected);
    }
}
```

Añadido el método **parseMoreStatement()** requerido en el tipo switch.

```java
/**
 * Analiza el símbolo <Decl>
 * @throws SintaxException
 */
private void parseDecl() throws SintaxException
{
    // ELIMINADO match(SEMICOLON);
    int[] expected = { INT, CHAR, BOOLEAN };
    switch(nextToken.getKind())
    {
        case INT:
        case CHAR:
        case BOOLEAN:
            parseType();
            match(IDENTIFIER);
            parseAssignement();
            parseMoreDecl();
            break;
        default:
            throw new SintaxException(nextToken,expected);
    }
}
```

Eliminación del semicolon en el método **parseDecl()**, para poder usarlo en el for.

```java
/**
 * Analiza el símbolo <ElseStm>
 * @throws SintaxException
 */
private void parseElseStm() throws SintaxException
{
    // AÑADIDO DO, FOR, SWITCH, BREAK, CONTINUE
    int[] expected = { ELSE, INT, CHAR, BOOLEAN, IDENTIFIER, IF, WHILE,
                       RETURN, SEMICOLON, LBRACE, RBRACE, DO, FOR, SWITCH,
                       BREAK, CONTINUE };
    switch(nextToken.getKind())
    {
        case ELSE:
            match(ELSE);
            parseStatement();
            break;
        case INT:
        case CHAR:
        case BOOLEAN:
        case IDENTIFIER:
        case IF:
        case WHILE:
        case DO:
        case FOR:
        case SWITCH:
        case CONTINUE:
        case BREAK:
        case RETURN:
        case SEMICOLON:
        case LBRACE:
        case RBRACE:
            break;
        default:
            throw new SintaxException(nextToken,expected);
    }
}
```

Modificación del método **parseElseStm()** para reconocer los nuevos métodos añadidos {**do, for, switch, break, continue**}.

```java
/**
 * Analiza el símbolo <SwitchStm>
 * @throws SintaxException
 */
private void parseSwitchStm() throws SintaxException {
    // AÑADIDO SWITCH
    int[] expected = { SWITCH };

    switch(nextToken.getKind()) {
        case SWITCH:
            match(SWITCH);
            match(LPAREN);
            parseExpr();
            match(RPAREN);
            match(LBRACE);
            parseOptionsSwitch();
            match(RBRACE);
            break;
        default:
            throw new SintaxException(nextToken,expected);
    }
}

/**
 * Analiza el símbolo <OptionsSwitch>
 * @throws SintaxException
 */
private void parseOptionsSwitch() throws SintaxException {
    // AÑADIDO PARA USAR EN EL SWITCH
    int[] expected = { CASE, DEFAULT, RBRACE };

    switch(nextToken.getKind()) {
    case CASE:
        match(CASE);
        match(INTEGER_LITERAL);
        match(COLONS);
        parseMoreStatement();
        parseOptionsSwitch();
        break;
    case DEFAULT:
        match(DEFAULT);
        match(COLONS);
        parseMoreStatement();
        parseOptionsSwitch();
        break;
    case RBRACE:
        break;
    default:
        throw new SintaxException(nextToken,expected);
    }
}
```

Añadidos los métodos **parseSwitchStm()** y **parseOptionsSwitch()**, para los tipos switch y case o default.

```java
/**
 * Analiza el símbolo <DoWhileStm>
 * @throws SintaxException
 */
private void parseDoWhileStm() throws SintaxException {
    // AÑADIDO DO
    int[] expected = { DO };

    switch(nextToken.getKind()) {
        case DO:
            match(DO);
            parseStatement();
            match(WHILE);
            match(LPAREN);
            parseExpr();
            match(RPAREN);
            match(SEMICOLON);
            break;
        default:
            throw new SintaxException(nextToken,expected);
    }
}
```

Añadido el método **parseDoWhileStm()** para el reconocimiento del tipo **Do-While**.

```java
/**
 * Analiza el símbolo <ForStm>
 * @throws SintaxException
 */
private void parseForStm() throws SintaxException {
    // AÑADIDO FOR
    int[] expected = { FOR };

    switch(nextToken.getKind()) {
    case FOR:
        match(FOR);
        match(LPAREN);
        parseForInit();
        match(SEMICOLON);
        parseExpr();
        match(SEMICOLON);
        parseIdStmList();
        match(RPAREN);
        parseStatement();
        break;
    default:
        throw new SintaxException(nextToken,expected);
    }
}

/**
 * Analiza el símbolo <ForInit>
 * @throws SintaxException
 */
private void parseForInit() throws SintaxException {
    // AÑADIDO PARA USAR EN EL FOR
    int[] expected = { INT, CHAR, BOOLEAN, IDENTIFIER, SEMICOLON };

    switch(nextToken.getKind()) {
        case INT:
        case CHAR:
        case BOOLEAN:
            parseDecl();
            break;
        case IDENTIFIER:
            parseIdStmList();
            break;
        case SEMICOLON:
            break;
        default:
            throw new SintaxException(nextToken,expected);
    }
}

/**
 * Analiza el símbolo <IdStmList>
 * @throws SintaxException
 */
private void parseIdStmList() throws SintaxException {
    // AÑADIDO PARA USAR EN EL FOR
    int[] expected = { IDENTIFIER, RPAREN };

    switch(nextToken.getKind()) {
        case IDENTIFIER:
            parseIdStm();
            parseMoreIdStmList();
            break;
        case RPAREN:
            break;
        default:
            throw new SintaxException(nextToken,expected);
    }
}
```

```java
/**
 * Analiza el símbolo <MoreIdStmList>
 * @throws SintaxException
 */
private void parseMoreIdStmList() throws SintaxException {
    // AÑADIDO PARA USAR EN EL FOR
    int[] expected = { COMMA, IDENTIFIER, RPAREN };

    switch(nextToken.getKind()) {
        case COMMA:
            match(COMMA);
            parseIdStm();
            parseMoreIdStmList();
            break;
        case RPAREN:
            break;
        default:
            throw new SintaxException(nextToken,expected);
    }
}
```

Añadidos los métodos **parseForStm()**, **parseForInit()**, **parseIdStmList()** y **parseModeIdStmList()** para usarlos en el tipo for.

```java
/**
 * Analiza el símbolo <BreakStm>
 * @throws SintaxException
 */
private void parseBreakStm() throws SintaxException {
    // AÑADIDO BREAK
    int[] expected = { BREAK };

    switch(nextToken.getKind()) {
        case BREAK:
            match(BREAK);
            match(SEMICOLON);
            break;
        default:
            throw new SintaxException(nextToken,expected);
    }
}

/**
 * Analiza el símbolo <ContinueStm>
 * @throws SintaxException
 */
private void parseContinueStm() throws SintaxException {
    // AÑADIDO CONTINUE
    int[] expected = { CONTINUE };

    switch(nextToken.getKind()) {
        case CONTINUE:
            match(CONTINUE);
            match(SEMICOLON);
            break;
        default:
            throw new SintaxException(nextToken,expected);
    }
}
```

Añadidos el **parseBreakStm()** y **parseContinueStm()** para detectar tanto el break como el continue.

Pruebas de Funcionamiento

Se ha elegido este código de prueba en el que se ha metido un elemento de cada nuevo tipo añadido. Con un resultado completamente satisfactorio.

```
import Console;

/**
 * Aplicación para mostrar los números primos menores de 100.
 * Biblioteca principal.
 */
library Main {

  /**
   * Punto de entrada de la aplicación
   */
  public void Main()
  {
    int b=1, p=1, c=9;

    switch(b) {
      case 0:
        imprimir(b);
        imprimir(p);
        break;
      case 1:
        imprimir(c);
        break;
      default:
        imprimir(p);
    }

    if ( p== 0) {
      imprimir(p);
    } else {
      for(int i=0; i<10; i = i+1)
        imprimir(i);
    }

    for (int i=1; i<100; i = i+1)
    {
      if( esPrimo(i) ) {
        imprimir(i);
        imprimir(b);
      }
        else
        imprimir(i);
    }

    do {
      imprimir(c);
      imprimir(b);
    } while( c==9 );

    if( c == 9)
      break;
    else
      continue;
  }
```

```
  /**
   * Imprime un número entero en la consola
   */
  private void imprimir(int i)
  {
    Console.print(i);
    Console.print('\n');
  }

  /**
   * Verifica si un número es primo
   */
  private boolean esPrimo(int i)
  {
    int j = 2;
    while(j<i)
    {
      if(i%j == 0) return false;
      j = j+1;
    }
    return true;
  }
}
```

Con el resultado: