Universidad de Huelva

Universidad de Huelva

Grado en Ingeniería Informática

Trabajo. Operadores de Bit

Trabajo Final de la Asignatura

Autor: Alberto Fernández Merchán
Profesor: Francisco José Moreno Velo
Asignatura: Procesadores del Lenguaje

Año: 2022

Índice

1. Introducción

Tinto es un lenguaje de programación imperativo orientado a procesos. En esta práctica se utilizará el compilador que hemos visto durante el curso para agregarle varias funcionalidades extra. Estos nuevos operadores serán los siguientes:

- **Operador NOT a nivel de bit** ($\sim$): Es un operador unario que invierte el valor de cada uno de los bits de su operando.
- **Operador AND a nivel de bit** (&): Es un operador binario que combina los bits de sus operandos de manera que se obtiene un 1 si ambos bits son 1 y un 0 en caso contrario.
- **Operador OR a nivel de bit** (|): Es un operador binario que combina los bits de sus operandos de manera que se obtiene un 0 si ambos bits son 0 y un 1 en caso contrario.
- **Operador XOR a nivel de bit** ($\wedge$): Es un operador binario que combina los bits de sus operandos de manera que se obtiene un 1 si ambos bits son diferentes y un 0 si son iguales.
- **Desplazamiento a la derecha con signo** (>>): Es un operador binario que mueve a la derecha los bits del primer operando tantas veces como indique el segundo operando. Si el primer operando es negativo, se insertan por la izquierda 1's, en caso contrario, se insertan 0's
- **Desplazamiento a la derecha sin signo** (>>>): Es un operando binario que mueve a la derecha los bits del primer operando tantas veces como indique el segundo operando. Se insertan 0's por la izquierda independientemente del signo del primer operando.
- **Desplazamiento a la izquierda** (<<): Es un operador binario que mueve a la izquierda los bits del primer operando tantas posiciones como indique el segundo operando. Los bits que excedan de la posición 31 se pierden.

Para poder añadir estos operadores al lenguaje Tinto tendremos que modificar el compilador que nos proporciona la página web de la asignatura [1]. A este compilador tendremos que modificarle el **analizador léxico**, el **analizador sintáctico**, el **analizador semántico** y, por último, **generar el código** ensamblador que permite utilizar dichos operadores.

Para realizar esta tarea utilizaremos la herramienta **JavaCC**, un árbol de **sintaxis abstracta (AST)** y varias clases para generar el código **ensamblador MIPS**.

Finalmente, se comprobará el correcto funcionamiento del lenguaje proponiendo varios ejemplos donde podría fallar el compilador y un último ejemplo que agrupe todos los operadores añadidos y muestre el resultado por la consola.

A continuación se muestra la estructura del proyecto. Esta se encuentra dividida en cuatro partes:

- **Programa Principal**. Lo forman las clases *TintoCompiler.java* y *TintoCompilerOptions.java*. Es el programa principal que llama al resto de clases.
- **Árbol de Sintáxis Abstracta**. Es la estructura de datos que almacena la sintaxis del lenguaje Tinto. Está formado por los paquetes: *tinto.ast.expression*, *tinto.ast.statement* y *tinto.ast.struct*.
- **Código MIPS**. Son las clases que se encargan de generar el código intermedio y código ensamblador MIPS. Está formado por los paquetes: *tinto.code*, *tinto.mips*, *tinto.mips.registers* y *tinto.mips.instructions*.
- **Tinto Parser**. Por último, el compilador de Tinto. Las clases se han generado utilizando los ficheros *TintoParser.jj* y *TintoHeaderParser.jj* que utilizan la herramienta **JavaCC**.

2. Modificaciones Léxicas y Sintácticas

2.1. Modificaciones Léxicas

Para incluir los nuevos operadores, primero debemos añadir nuevas categorías léxicas que reconozcan los símbolos asociados a esos operadores. Esto se realiza en el fichero *TintoHeaderParser.jj* y en *TintoParser.jj*.

```
                    /* OPERATORS */

TOKEN :
{
  < ASSIGN: "=" >
| < EQ: "==" >
| < LE: "<=" >
| < GT: ">" >
| < LT: "<" >
| < GE: ">=" >
| < NE: "!=" >
| < OR: "||" >
| < AND: "&&" >
| < NOT: "!" >
| < PLUS: "+" >
| < MINUS: "-" >
| < PROD: "*" >
| < DIV: "/" >
| < MOD: "%" >
| < TILDE: "~" >
| < BIT_AND: "&" >
| < BIT_OR: "|" >
| < XOR: "^" >
| < RS_SHIFT: ">>" >
| < RU_SHIFT: ">>>" >
| < L_SHIFT: "<<" >
}
```

2.2. Modificaciones Sintácticas

Para incluir los operadores de bit al lenguaje Tinto tenemos que añadir a la gramática de Tinto algunas reglas y modificar otras que ya estaban establecidas.

2.2.1. Modificaciones en la Gramática de Tinto

Las nuevas reglas que describen las expresiones gramaticales son las siguientes:

Expr ::= AndExpr (or AndExpr)*
AndExpr ::= BitOrExpr (and BitOrExpr)*
BitOrExpr ::= BitXOrExpr (BIT-OR BitXOrExpr)*
BitXOrExpr ::= BitAndExpr (XOR BitAndExpr)*
BitAndExpr ::= RelExpr (BIT-AND RelExpr)*
RelExpr ::= ShiftExpr (RelOp ShiftExpr)?
RelOp ::= (eq | ne | gt | ge | lt | le)
ShiftExpr ::= SumExpr (ShiftOp SumExpr)*
ShiftOp :: = (L-SHIFT | RS-SHIFT | RU-SHIFT)
SumExpr ::= UnOp ProdExpr (SumOp ProdExpr)*
UnOp ::= (not | plus | minus)?
SumOp ::= (plus | minus)
ProdExpr ::= CompExpr (ProdOp CompExpr)*
ProdOp ::= (prod | div | mod)
CompExpr ::= (TILDE)? Factor
Factor ::= Literal | Reference | lparen Expr rparen
Literal ::= integer-literal | char-literal | true | false
Reference ::= identifier (MethodCall | dot identifier MethodCall)?
MethodCall ::= lparen (Exp (comma Expr)*)? rparen

Con este orden establecido en las reglas gramaticales se establece también un orden en la prioridad de aplicación de los diferentes operadores del lenguaje. De esta forma, el operador de complemento ($\sim$) sería el más prioritario mientras que, por el contrario el operador OR (||) sería el menos prioritario.

2.2.2. Modificaciones Sintácticas en el fichero TintoParser.jj

En el fichero *TintoParser.jj* tendremos que añadir las reglas gramaticales del lenguaje. Para ello tendremos que añadir las siguientes funciones:

AndExpr ::= BitOrExpr (AND BitOrExpr)*

```
void AndExpr() :
{ }
{
  BitOrExpr() ( <AND> BitOrExpr()  )*
}
```

BitOrExpr ::= BitXOrExpr (BIT-OR BitXOrExpr)*

```
void BitOrExpr() :
{ }
{
        BitXORExpr() (<BIT_OR> BitXORExpr() )*
}
```

BitXORExpr ::= BitAndExpr (XOR BitAndExpr)*

```
void BitXORExpr() :
{ }
{
        BitAndExpr() (<XOR> BitAndExpr())*
}
```

BitAndExpr ::= RelExpr (BIT-AND RelExpr)*

```
void BitAndExpr() :
{ }
{
        RelExpr() (<BIT_AND> RelExpr())*
}
```

RelExpr ::= ShiftExpr (RelOp ShiftExpr)?

```
void RelExpr() :
{ }
{
   ShiftExpr() ( RelOp() ShiftExpr() )?
}
```

ShiftExpr ::= SumExpr (ShiftOp SumExpr)*

```
void ShiftExpr() :
{ }
{
        SumExpr() ( ShiftOp() SumExpr() )*
}
```

ShiftOp :: = (L-SHIFT | RS-SHIFT | RU-SHIFT)

```
void ShiftOp() :
{ }
{
   <L_SHIFT>
 | <RS_SHIFT>
 | <RU_SHIFT>
}
```

ProdExpr ::= CompExpr (ProdOp CompExpr)*

```
void ProdExpr() :
{ }
{
        CompExpr() ( MultOp() CompExpr() )*
}
```

CompExpr ::= (TILDE)? Factor

```
void CompExpr() :
{ }
{
        (<TILDE>)? Factor()
}
```

3. Descripción de las modificaciones de las clases que desarrollan el Árbol de Sintaxis Abstracta del compilador

Como el objetivo de la práctica es añadir nuevos operadores al lenguaje Tinto, debemos modificar las clases donde se especifican las expresiones binarias (*tinto.ast.expression.BinaryExpression.java*) y unarias (*tinto.ast.expression.UnaryExpression.java*).

3.1. Expresiones Binarias

Este tipo de expresiones son las que implican dos operandos. Las nuevas expresiones binarias que vamos a añadir son las siguientes:

- **BIT-AND**. Operador AND a nivel de bit.
- **BIT-OR**. Operador OR a nivel de bit.
- **XOR**. Operador XOR a nivel de bit.
- **RS-SHIFT**. Operador de desplazamiento hacia la derecha con signo.
- **RU-SHIFT**. Operador de desplazamiento hacia la derecha sin signo.
- **L-SHIFT**. Operador de desplazamiento hacia la izquierda con signo.

A continuación se muestra el código de la clase *BinaryExpression.java* donde se le da un identificador a cada una de las expresiones binarias:

```
/**
 * Operador: BIT_AND
 */
public static final int BIT_AND = 14;
/**
 * Operador: BIT_OR
 */
public static final int BIT_OR = 15;
/**
 * Operador: XOR
 */
public static final int XOR = 16;
```

```
/**
 * Operador: L_SHIFT
 */
public static final int L_SHIFT = 17;
/**
 * Operador: RS_SHIFT
 */
public static final int RS_SHIFT = 18;
/**
 * Operador: RU_SHIFT
 */
public static final int RU_SHIFT = 19;
```

También debemos modificar el método privado de la clase ***computeType***, que se encarga de devolver el tipo de dato que resulta tras evaluar la expresión. En el caso de los nuevos operadores devolverán el mismo tipo de dato de la expresión de la izquierda del operador (ya que ambas expresiones deberán de ser del mismo tipo (entero)):

```
/**
 * Calcula el tipo de dato de la expresión binaria
 */
private static int computeType(int op, Expression left, Expression right)
{
        switch(op)
        {
        case AND:
        case OR:
        case EQ:
        case NEQ:
        case LT:
        case LE:
        case GT:
        case GE:
                return Type.BOOLEAN_TYPE;
        case PLUS:
        case MINUS:
        case PROD:
        case DIV:
        case MOD:

        case BIT_AND:
        case BIT_OR:
        case XOR:
        case L_SHIFT:
        case RS_SHIFT:
        case RU_SHIFT:
                return left.getType();
        default:
                return Type.MISMATCH_TYPE;
        }
}
```

3.2. Expresiones Unarias

Este tipo de expresiones son aquellas que solo involucran un operando. En esta práctica tan solo hay un operador que sea de este tipo, el operador de complemento (*TILDE*). A continuación se muestra el código de la clase *UnaryExpression.java*, donde se le asigna un identificador a cada expresión unaria:

```
/**
 * Operador: TILDE
 */
public static final int TILDE = 4;
```

Y, como en las expresiones binarias, también tenemos que modificar el método privado ***computeType*** que se encarga de devolver el tipo de dato del resultado de evaluar la expresión:

```
/**
 * Calcula el tipo de dato de la expresión unaria
 */
private static int computeType(int op, Expression exp)
{
        switch(op)
        {
        case NOT:
                return Type.BOOLEAN_TYPE;
        case MINUS:
                return exp.getType();
        case TILDE:
                return Type.INT_TYPE;
        default:
                return exp.getType();
        }
}
```

4. Descripción de las modificaciones semánticas realizadas sobre los ficheros JavaCC del compilador

Debido a que el lenguaje Tinto no usa ficheros de cabecera para predeclarar el contenido de una librería, debemos compilar un fichero Tinto en dos pasadas. La primera se hará para obtener la declaración de las cabeceras de las funciones definidas en el fichero analizado. La segunda pasada se encargará de analizar el cuerpo de las funciones para obtener una lista de instrucciones de la función.

4.1. Analizador Semántico de Cabecera

Como hemos dicho anteriormente, esta primera pasada se encargará de analizar la cabecera de las funciones. Para ello utilizamos el analizador *TintoHeaderParser.jj*. En este fichero tendremos que añadir los *tokens* que definen los nuevos operadores que vamos a añadir al lenguaje.

4.2. Analizador Semántico Completo

A continuación se realizará una segunda pasada con el objetivo de analizar semánticamente el cuerpo de las funciones. Para este analizador se usa el fichero *TintoParser.jj*.

4.2.1. Expresiones Semánticas

En este fichero hemos tenido que actualizar las funciones que hemos especificado en el analizador sintáctico. Le hemos añadido control de errores y atributos sintetizados y heredados. Las funciones quedarían de la siguiente forma:

AndExpr ::= BitOrExpr (AND BitOrExpr)*

```
Expression parseAndExpr(SymbolTable symtab) :
{
  Expression exp1, exp2;
  Token tk;
}
{
  exp1 = tryBitOrExpr(symtab)
  (
    tk = < AND >
    exp2 = tryBitOrExpr(symtab)
    {
      exp1 = actionAndExpression(tk, exp1, exp2);
    }
  )*
  {return exp1;}
}
```

BitOrExpr ::= BitXOrExpr (BIT-OR BitXOrExpr)*

```
Expression parseBitOrExpr(SymbolTable symtab) :
{
  Expression exp1, exp2;
  Token tk;
  int op = BinaryExpression.BIT_OR;
}
{
  exp1 = tryXORExpr(symtab)
  (
    tk = < BIT_OR >
    exp2 = tryXORExpr(symtab)
    {exp1 = actionBitORExpression(tk, exp1, exp2);}
  )*
  {return exp1;}
}
```

BitXORExpr ::= BitAndExpr (XOR BitAndExpr)*

```
Expression parseXORExpr(SymbolTable symtab) :
{
  Expression exp1, exp2;
  Token tk;
  int op = BinaryExpression.XOR;
}
{
  exp1 = tryBitAndExpr(symtab)
  (
    tk = < XOR >
    exp2 = tryBitAndExpr(symtab)
    { exp1 = actionXORExpression(tk, exp1, exp2);}
  )*
  {
    return exp1;
  }
}
```

BitAndExpr ::= RelExpr (BIT-AND RelExpr)*

```
Expression parseBitAndExpr(SymbolTable symtab) :
{
  Expression exp1, exp2;
  Token tk;
}
{
  exp1 = tryRelExpr(symtab)
  (
    tk = < BIT_AND >
    exp2 = tryRelExpr(symtab)
    {exp1 = actionBitAndExpression(tk, exp1, exp2);}
  )*
  {
    return exp1;
  }
}
```

RelExpr ::= ShiftExpr (RelOp ShiftExpr)?

```
Expression parseRelExpr(SymbolTable symtab) :
{
  Expression exp1, exp2;
  int op;
  Token tk;
}
{
  exp1 = tryShiftExpr(symtab)
  (
    {tk = getToken(1);}
    op = RelOp()
    exp2 = tryShiftExpr(symtab)
    {exp1 = actionRelExpression(tk, op, exp1, exp2);}
  )?
  {return exp1;}
}
```

ShiftExpr ::= SumExpr (ShiftOp SumExpr)*

```
Expression parseShiftExpr(SymbolTable symtab) :
{
  Expression exp1, exp2;
  int op;
  Token tk;
}
{
  exp1 = trySumExpr(symtab)
  (
    {tk = getToken(1);}
    op = ShiftOp()
    exp2 = trySumExpr(symtab)
    {exp1 = actionShiftExpression(tk, op, exp1, exp2);}
  )*
  {
    return exp1;
  }
}
```

ShiftOp :: = (L-SHIFT | RS-SHIFT | RU-SHIFT)

```
int ShiftOp() :
{
  int op;
}
{
  (
    < L_SHIFT >
    {op = BinaryExpression.L_SHIFT;}
  | < RS_SHIFT >
    {op = BinaryExpression.RS_SHIFT;}
  | < RU_SHIFT >
    {op = BinaryExpression.RU_SHIFT;}
  )
  {return op;}
}
```

ProdExpr ::= CompExpr (ProdOp CompExpr)*

```
Expression parseProdExpr(SymbolTable symtab) :
{
  Expression exp1, exp2;
  int op;
  Token tk;
}
{
  exp1 = tryCompExpr(symtab)
  (
    {tk = getToken(1);}
    op = MultOp()
    exp2 = tryCompExpr(symtab)
    {exp1 = actionProdExpression(tk, op, exp1, exp2);}
  )*
  {return exp1;}
}
```

CompExpr ::= (TILDE)? Factor

```
Expression parseCompExpr(SymbolTable symtab) :
{
  Expression exp1;
  int op = UnaryExpression.NONE;
  Token tk;
}
{
  (< TILDE >
  { op = UnaryExpression.TILDE;}
  )?
  {    tk = getToken(1); }
  exp1 = tryFactor(symtab)
  {    exp1 = actionCompExpression(tk, op, exp1);}
  {
    return exp1;
  }
}
```

4.2.2. Acciones Semánticas

Cada una de las expresiones anteriores llama a una función que se encargará de crear la expresión correspondiente. A estas funciones se les conoce como **acciones semánticas** y están escritas en lenguaje Java.

Las acciones semánticas que he añadido son las siguientes:

```
private Expression actionBitOrExpression(Token tid1, Expression exp1, Expression exp2)
{
  if (exp1 == null || exp2 == null) return null;
  verifyNumericTypes(tid1, exp1, exp2);
  int op = BinaryExpression.BIT_OR;
  Expression exp = new BinaryExpression(op, exp1, exp2);
  return exp;
}
```

```
private Expression actionBitAndExpression(Token tid1, Expression exp1, Expression exp2)
{
  if (exp1 == null || exp2 == null) return null;
  verifyNumericTypes(tid1, exp1, exp2);
  int op = BinaryExpression.BIT_AND;
  Expression exp = new BinaryExpression(op, exp1, exp2);
  return exp;
}

private Expression actionXORExpression(Token tid1, Expression exp1, Expression exp2)
{
  if (exp1 == null || exp2 == null) return null;
  verifyNumericTypes(tid1, exp1, exp2);
  int op = BinaryExpression.XOR;
  Expression exp = new BinaryExpression(op, exp1, exp2);
  return exp;
}

private Expression actionCompExpression(Token tk, int op, Expression exp1)
{
  switch (op)
  {
    case UnaryExpression.NONE :
    return exp1;
    case UnaryExpression.TILDE :
    verifyNumericType(tk, exp1);
    return new UnaryExpression(op, exp1);
  }
  return exp1;
}

private Expression actionShiftExpression(Token tid1, int op, Expression exp1, Expression exp2)
{
  if (exp1 == null || exp2 == null) return null;
  verifyNumericTypes(tid1, exp1, exp2);
  Expression exp = new BinaryExpression(op, exp1, exp2);
  return exp;
}
```

```
  private Expression actionBitORExpression(Token tk, Expression exp1, Expression exp2)
  {
    if (exp1 == null || exp2 == null) return null;
    verifyNumericTypes(tk, exp1, exp2);
    int op = BinaryExpression.BIT_OR;
    Expression exp = new BinaryExpression(op, exp1, exp2);
    return exp;
  }
}
```

5. Descripción de las modificaciones de las clases que generan el código intermedio del compilador.

El compilador de Tinto utiliza un código intermedio propio formado por un total de 23 instrucciones. Se trata de un código de tres direcciones que se almacenan en los campos target, source1 y source2. Para añadir los operadores de bit he tenido que añadir nuevas instrucciones.

Para ello, en la clase *CodeConstants.java* he tenido que añadir nuevos identificadores para asignarlos a los nuevos operadores. El código es el siguiente:

```
...
/**
 * Instrucciones de Operadores de BIT y DESPLAZAMIENTO.
 */
public int TILDE = 24;
public int BIT_AND = 25;
public int BIT_OR = 26;
public int XOR = 27;
public int RS_SHIFT = 28;
public int RU_SHIFT = 29;
public int L_SHIFT = 30;
```

Además, también se han añadido las instrucciones a la clase *CodeInstruction.java*. Esta clase representa a una instrucción en código intermedio. Además, también se le puede añadir un comentario que indica cómo funciona la instrucción. El código modificado es el siguiente:

```
 /**
 * Obtiene el nombre correspondiente al tipo
 * de la instrucción
 */
private String getInstructionName()
{
  switch(kind)
   {
   case LABEL:  return "";
...
   case BIT_AND: return "and";
   case BIT_OR: return "or";
   case XOR: return "xor";
   case TILDE: return "nor";
   case L_SHIFT: return "sllv";
   case RS_SHIFT: return "srav";
   case RU_SHIFT: return "srlv";
        default: return "";
    }
}
```

```
private String getCode()
{
        String tg = (target == null? "": target.toString());
        String s1 = (source1 == null? "": source1.toString());
        String s2 = (source2 == null? "": source2.toString());

        String inst = "\t"+getInstructionName();
        switch(kind) {
        case LABEL:
                return tg+":";
        ...
        case TILDE:
        case BIT_AND:
        case BIT_OR:
        case XOR:
        case L_SHIFT:
        case RS_SHIFT:
        case RU_SHIFT:

                return inst+" "+tg+", "+s1+", "+s2;
        ...
        default:
                return "";
        }
}

private String getComment()
{
        String tg = (target == null? "": target.getDescription());
        String s1 = (source1 == null? "": source1.getDescription());
        String s2 = (source2 == null? "": source2.getDescription());
        switch(kind)
        {
        case LABEL:
                return "";
        ...
        case TILDE:
                return tg+" <- ~"+s1;
        case BIT_AND:
                return tg+" <- "+s1+" & "+s2;
        case BIT_OR:
                return tg+" <- "+s1+" | "+s2;
        case XOR:
                return tg+" <- "+s1+" ^ "+s2;
        case L_SHIFT:
                return tg+" <- "+s1+" << "+s2;
        case RS_SHIFT:
                return tg+" <- "+s1+" >> "+s2;
        case RU_SHIFT:
                return tg+" <- "+s1+" >>> "+s2;
        default:
                return "";
        }
}
```

Por último, en la clase *CodeGenerator.java* se ha modificado el método *generateCodeForUnaryExpression* y se ha añadido el método *getUnaryCode*. De esta forma se selecciona el identificador de la instrucción de cada uno de los operadores unarios.

```
        /**
         * Genera el código de una operación aritmética unaria
         */
        private CodeVariable generateCodeForUnaryExpression(FunctionCodification mc,
            UnaryExpression exp, CodeInstructionList codelist)
        {
                Expression operand = exp.getExpression();

                CodeInstructionList code = new CodeInstructionList();
                CodeVariable source = generateCodeForExpression(mc,operand,code);
                CodeVariable target = mc.getNewTemp();

                int op = getUnaryCode(exp.getOperator());

                code.addInstruction(new CodeInstruction(op,target,source,null));
                codelist.addInstructionList(code.getList());

                return target;
        }

        private int getUnaryCode(int operator) {

                switch(operator) {
                case UnaryExpression.MINUS: return INV;
                case UnaryExpression.NOT: return NOT;
                case UnaryExpression.TILDE: return TILDE;
                }
                return 0;
        }
```

Además, se han añadido los nuevos operadores binarios al método *getBinaryCode* para poder obtener sus identificadores correspondientes:

```
private int getBinaryCode(int op)
{
        switch(op)
        {
        case BinaryExpression.EQ: return JMPEQ;
        ...
        case BinaryExpression.BIT_AND: return BIT_AND;
        case BinaryExpression.BIT_OR: return BIT_OR;
        case BinaryExpression.XOR: return XOR;
        case BinaryExpression.RS_SHIFT: return RS_SHIFT;
        case BinaryExpression.RU_SHIFT: return RU_SHIFT;
        case BinaryExpression.L_SHIFT: return L_SHIFT;

        default: return 0;
        }
}
```

6. Descripción de las modificaciones de las clases que generan el código ensamblador MIPS del compilador

El último paso es generar el código ensamblador del procesador MIPS a partir del código intermedio que hemos generado en el apartado anterior.

Para ello tendremos que modificar las clases que pertenecen a los paquetes *tinto.mips.instructions* y *tinto.mips*. Concretamente las clases: *InstructionFactory.java* y *FunctionAssembler.java*.

6.1. Modificación InstructionFactory.java

Esta clase contiene un conjunto de métodos estáticos para crear las instrucciones de MIPS. Sin embargo, no contiene todas las instrucciones, sino solo las que el lenguaje Tinto soporta. En este proyecto tendremos que añadir cuatro instrucciones más referentes a los desplazamientos y al operador de complemento. Las operaciones de AND, OR y XOR ya están añadidas (ya que usan la misma instrucción que para los operadores lógicos correspondientes). El código es el siguiente:

```
//-------------------------------------------------------------------//
//                        BIT OPERATORS: RS_SHIFT                    //
//-------------------------------------------------------------------//

public static Instruction createRS_SHIFT(Register target, Register source1,
Register source2)
{
        return new RRRInstruction(SRAV, target, source1, source2);
}
//-------------------------------------------------------------------//
//                        BIT OPERATORS: RU_SHIFT                    //
//-------------------------------------------------------------------//

public static Instruction createRU_SHIFT(Register target, Register source1,
Register source2)
{
        return new RRRInstruction(SRLV, target, source1, source2);
}


//-------------------------------------------------------------------//
//                        BIT OPERATORS: L_SHIFT                     //
//-------------------------------------------------------------------//

public static Instruction createL_SHIFT(Register target, Register source1,
Register source2)
{
        return new RRRInstruction(SLLV, target, source1, source2);
}

//-------------------------------------------------------------------//
//                        BIT OPERATORS: TILDE                      //
//-------------------------------------------------------------------//

public static Instruction createTILDE(Register target, Register source1,
Register source2)
{
        return new RRRInstruction(NOR, target, source1, source2);
}
```

6.2. Modificación FunctionAssembler.java

Esta clase es la encargada de generar el código ensamblador a partir del código intermedio. Para incluir los operadores de bit, tendremos que crear nuevos métodos para que cree las instrucciones de desplazamiento y complemento. Las instrucciones de AND y OR ya están creadas (ya que se usa la misma instrucción para las operaciones lógicas homónimas).

El código que hay que añadir es el siguiente:

```
/**
 * Genera la descripción en ensamblador de cada una de las instrucciones del
 * código intermedio
 */
private void createAssembler(Vector<Instruction> vector, CodeInstruction inst) {
        int kind = inst.getKind();
        CodeAddress target = inst.getTarget();
        CodeAddress source1 = inst.getSource1();
        CodeAddress source2 = inst.getSource2();
        switch (kind) {
        case CodeConstants.LABEL:
                translateLabel(vector, target);
                break;
        ...
        case CodeConstants.AND:
                translateAND(vector, target, source1, source2);
                break;
        case CodeConstants.BIT_OR:
                translateOR(vector, target, source1, source2);
                break;
        case CodeConstants.XOR:
                translateXOR(vector, target, source1, source2);
                break;
        case CodeConstants.RU_SHIFT:
                translateRUSHIFT(vector, target, source1, source2);
                break;
        case CodeConstants.RS_SHIFT:
                translateRSSHIFT(vector, target, source1, source2);
                break;
        case CodeConstants.L_SHIFT:
                translateLSHIFT(vector, target, source1, source2);
                break;
        case CodeConstants.TILDE:
                translateTILDE(vector, target, source1);
                break;
        }
}
```

También hay que hacer los métodos *translate...* correspondientes a cada instrucción nueva:

```
private void translateTILDE(Vector<Instruction> vector, CodeAddress target,
        CodeAddress source) {
    Register source_reg = translateLoadIntValue(vector, source, RegisterSet.a0);
    Register target_reg = getTargetRegister(target, RegisterSet.v0);
    if(needsNOP(source_reg)) vector.add(InstructionFactory.createNOP());
    vector.add(InstructionFactory.createTILDE(target_reg, source_reg , RegisterSet.r0));
    setFetched(null,null,0);
    translateStoreIntValue(vector, target, target_reg);
}
```

```
    private void translateLSHIFT(Vector<Instruction> vector, CodeAddress target,
            CodeAddress source1, CodeAddress source2) {

    Register source1_reg = translateLoadIntValue(vector, source1, RegisterSet.a0);
    Register source2_reg = translateLoadIntValue(vector, source2, RegisterSet.a1);
    Register target_reg = getTargetRegister(target, RegisterSet.v0);
    if (needsNOP(source1_reg, source2_reg))
            vector.add(InstructionFactory.createNOP());
    vector.add(InstructionFactory.createL_SHIFT(target_reg, source1_reg, source2_reg));
    setFetched(null, null, 0);
    translateStoreIntValue(vector, target, target_reg);

}

    private void translateRSSHIFT(Vector<Instruction> vector, CodeAddress target,
            CodeAddress source1, CodeAddress source2) {

    Register source1_reg = translateLoadIntValue(vector, source1, RegisterSet.a0);
    Register source2_reg = translateLoadIntValue(vector, source2, RegisterSet.a1);
    Register target_reg = getTargetRegister(target, RegisterSet.v0);
    if (needsNOP(source1_reg, source2_reg))
            vector.add(InstructionFactory.createNOP());
    vector.add(InstructionFactory.createRS_SHIFT(target_reg, source1_reg, source2_reg));
    setFetched(null, null, 0);
    translateStoreIntValue(vector, target, target_reg);

}

    private void translateRUSHIFT(Vector<Instruction> vector, CodeAddress target,
            CodeAddress source1, CodeAddress source2) {

    Register source1_reg = translateLoadIntValue(vector, source1, RegisterSet.a0);
    Register source2_reg = translateLoadIntValue(vector, source2, RegisterSet.a1);
    Register target_reg = getTargetRegister(target, RegisterSet.v0);
    if (needsNOP(source1_reg, source2_reg))
            vector.add(InstructionFactory.createNOP());
    vector.add(InstructionFactory.createRU_SHIFT(target_reg, source1_reg, source2_reg));
    setFetched(null, null, 0);
    translateStoreIntValue(vector, target, target_reg);
}

    private void translateXOR(Vector<Instruction> vector, CodeAddress target,
            CodeAddress source1, CodeAddress source2) {

    Register source1_reg = translateLoadIntValue(vector, source1, RegisterSet.a0);
    Register source2_reg = translateLoadIntValue(vector, source2, RegisterSet.a1);
    Register target_reg = getTargetRegister(target, RegisterSet.v0);
    if (needsNOP(source1_reg, source2_reg))
            vector.add(InstructionFactory.createNOP());
    vector.add(InstructionFactory.createXOR(target_reg, source1_reg, source2_reg));
    setFetched(null, null, 0);
    translateStoreIntValue(vector, target, target_reg);

}
```

7. Pruebas De Funcionamiento

7.1. Funcionamiento del Analizador Léxico

Para comprobar que el analizador léxico funciona correctamente he creado un fichero donde exista un símbolo que el lenguaje Tinto no acepte, como por ejemplo el símbolo '?'. El resultado esperado tras compilar el fichero será un fichero de *TintocErrors.txt* donde especifique que el símbolo '?' es un error léxico.

El código que usaremos será el siguiente:

```
import Console;

/**
 * Aplicación para probar los operadores
 * de bit.
 */
library Main {

  /**
   * Punto de entrada de la aplicación
   */
  public void Main()
  {
    ?; // Símbolo no reconocido por el lenguaje Tinto.
    int a = ~42;
    int b = 15;

  }


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

Como habíamos previsto, al compilar el fichero nos genera un documento en el que se especifica el tipo de error que se ha cometido (Figura 1):

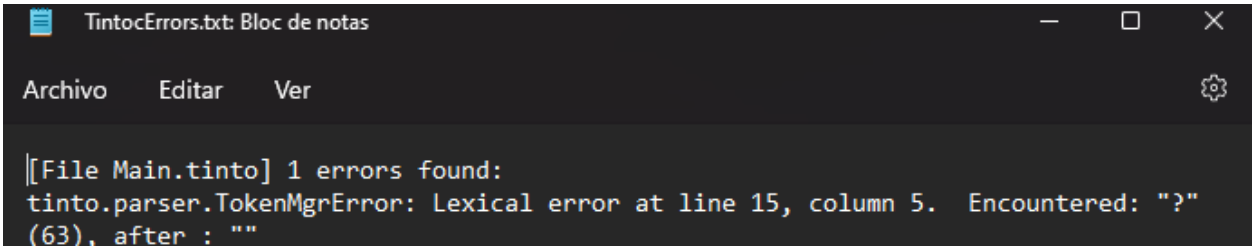

Figura 1: Error Léxico en el fichero Main.tinto

7.2. Funcionamiento del Analizador Sintáctico

Para probar el correcto funcionamiento del analizador sintáctico usaremos un fichero que contenga una expresión que no se encuentre definida en la gramática del lenguaje. Por ejemplo colocar un operador binario con un solo operando o, por el contrario, utilizar un operador unario haciendo uso de dos operadores. El resultado esperado tras compilar el fichero será un fichero de errores llamado *TintocErrors.txt* que especifique en qué línea se ha cometido el error sintáctico.

El código que he utilizado para comprobar el funcionamiento del analizador sintáctico es el siguiente:

```
import Console;

/**
 * Aplicación para probar los operadores
 * de bit.
 */
library Main {

  /**
   * Punto de entrada de la aplicación
   */
  public void Main()
  {

    int a = 10~42; // Al ser un operador unario, no debería aceptar dos operandos.
    int b = 15;

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

Como habíamos previsto, el compilador genera un fichero de texto donde especifica el tipo de error y algunas opciones para poder arreglar el fallo (Figura 2).

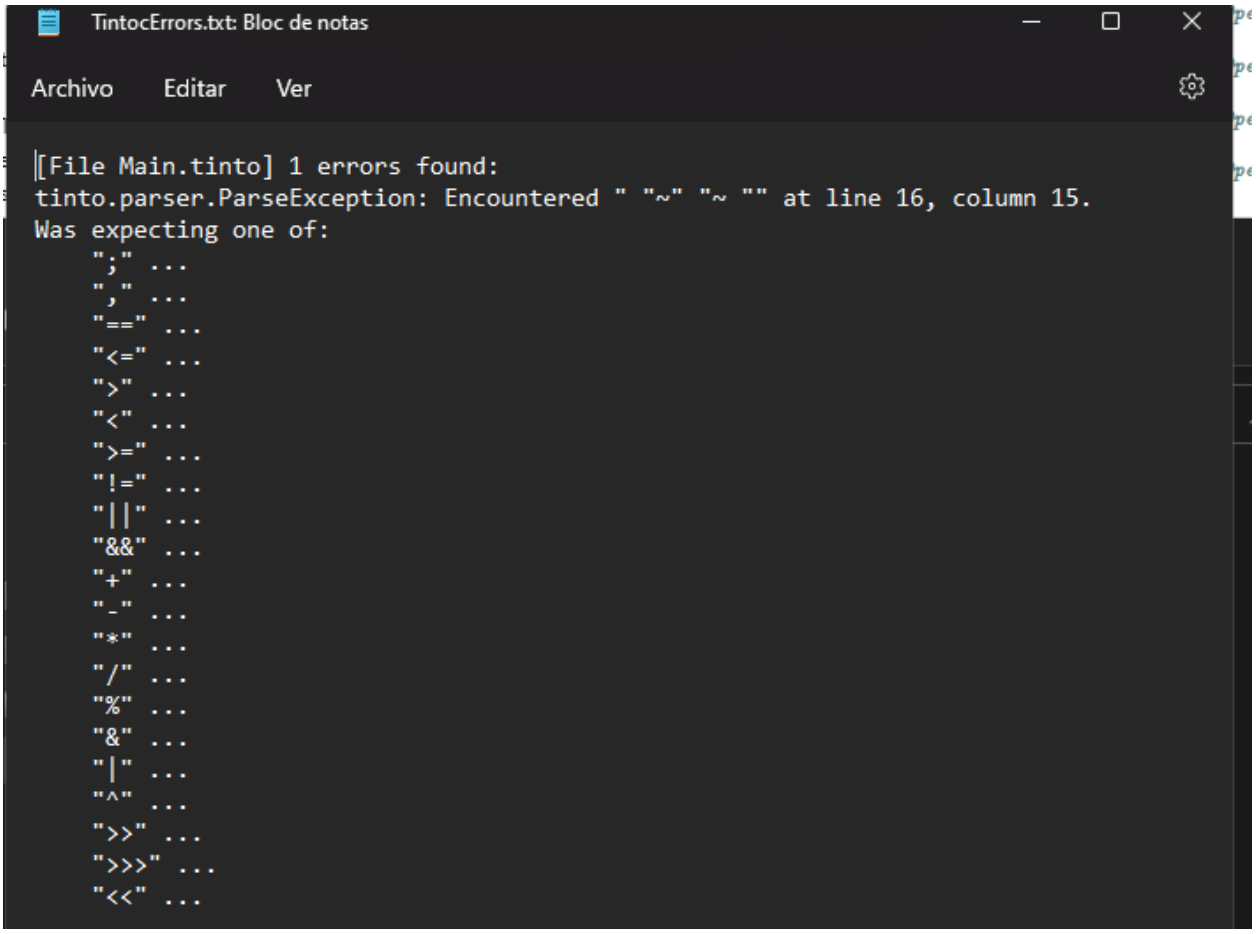

Figura 2: Error Sintáctico en el fichero Main.tinto

7.3. Funcionamiento del Analizador Semántico

Por último, para comprobar que el analizador semántico funciona correctamente, podemos cambiar el tipo de operando que utilizan los operadores que hemos añadido a la gramática. Por ejemplo, haciendo una operación AND a nivel de bit entre dos caracteres. El resultado esperado al compilar el fichero será un fichero de errores llamado *TintocErrors.txt* donde especifique el tipo de error semántico que se ha cometido.

El código que he utilizado para comprobar el funcionamiento del analizador semántico es el siguiente:

```
import Console;

/**
 * Aplicación para probar los operadores
 * de bit.
 */
library Main {

  /**
   * Punto de entrada de la aplicación
   */
  public void Main()
  {
    int a = ~42;
    int c = 'c'&2; // Es correcto sintácticamente, sin embargo,
                   //  el tipo de los operandos no es correcto.
  }

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

El fichero de error que genera el compilador es el siguiente (Figura 3):

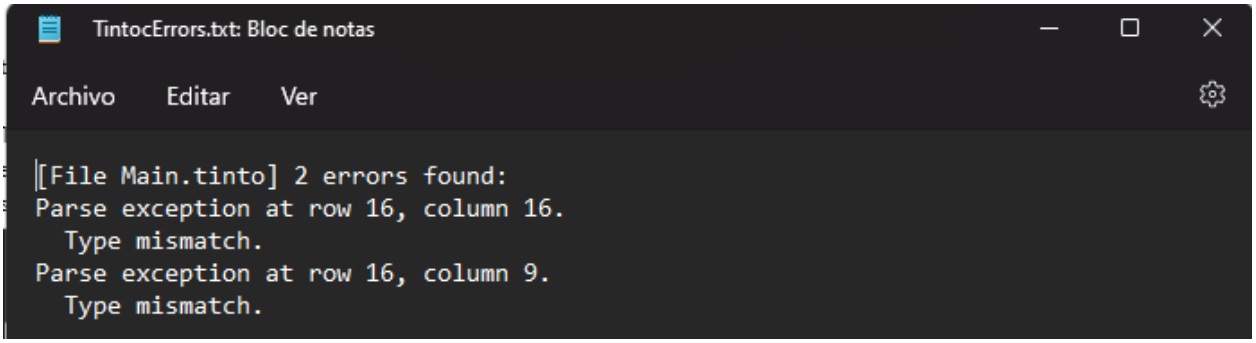

Figura 3: Error semántico en el fichero Main.tinto

7.4. Funcionamiento de Nuevos Operadores

Para comprobar que los nuevos operadores que he introducido al lenguaje Tinto funcionan correctamente, he creado un fichero *Main.tinto* donde se pone a prueba cada uno de los operadores. Las pruebas que se han realizado son las mismas que ofrece el enunciado de la práctica. El código del programa principal es el siguiente:

```
import Console;

/**
 * Aplicación para probar los operadores
 * de bit.
 */
library Main {

  /**
   * Punto de entrada de la aplicación
   */
  public void Main()
  {
    int a = ~42;
    int b = 15;

    imprimir(a); // Prueba del Operador TILDE
    a = 42;

    int c = a&b;
    imprimir(c); // Prueba del Operador BIT_AND

    c = a|b;
    imprimir(c); // Prueba del Operador BIT_OR

    c = a^b;
    imprimir(c); // Prueba del Operador XOR

    c = a<<5;
    imprimir(c); // Prueba del Operador L_SHIFT
    a = -42;

    c = a>>3;
    imprimir(c); // Prueba del Operador RS_SHIFT

    c = a>>>3;
    imprimir(c); // Prueba del Operador RU_SHIFT
  }


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

7.4.1. Resultados

Tras ejecutarlo en el simulador *QtSpim* del procesador MIPS, obtenemos los siguientes resultados:

Figura 4: Resultados de las operaciones a nivel de bit en el procesador MIPS.

8. Conclusiones

En esta práctica hemos aprendido como programar un compilador que hace uso de un analizador léxico, uno sintáctico y uno semántico utilizando la herramienta JavaCC. Además, también hemos generado el código ensamblador para el procesador MIPS.

Para poder utilizar los operadores de bit y de desplazamiento en el lenguaje Tinto, hemos tenido que añadirlos a la práctica que estaba subida en la página web de la asignatura [1]. Debido a que los operadores eran parecidos a otros que ya estaban creados en la práctica no he tenido que realizar grandes cambios sobre el programa.

Para los operadores binarios he utilizado como base el operador de suma que ya incluía el lenguaje Tinto, en cambio, para el operador de complemento no he podido utilizar ese debido a que es un operador unario. En su lugar he utilizado el operador de signo negativo (MINUS) como base.

Ha sido muy interesante aprender como funcionan los compiladores que utilizamos en otras asignaturas para programar. Además, también es útil saber hacer un compilador de cualquier gramática regular ya que, para poder hacer programas que requieran leer ficheros, es necesario tener dichos ficheros estructurados de una manera que pueda ser reconocible y poder generar el código necesario.

En cuanto a los resultados de la práctica he comprobado, utilizando una calculadora de operaciones binarias, que dichos resultados son correctos, por lo que podemos concluir con que he añadido correctamente los operadores de bit y de desplazamiento.

Referencias

[1] Francisco Moreno Velo. Trabajo de procesadores del lenguaje. http://www.uhu.es/francisco.moreno/gii_pl/practicas/practica01.htm, 2022.