INDICE.

1.- El analizador semántico y la comprobación de tipos.

El analizador semántico es el encargado de comprobar que el código analizado es semánticamente correcto. Entre otras cosas, el analizador semántico comprueba la no duplicidad en la declaración de variables, constantes, nombres de procedimientos y funciones, etc... También realiza la comprobación de tipos, que las llamadas a subprogramas sean correctas, es decir, comprueba el numero de parámetros, orden y tipo de estos, la presencia de una sentencia de retorno de valor devuelto en una función.
El análisis semántico se realiza, principalmente, en el parser.cup del proyecto, apoyándose en clases de los paquetes compiler.semantic.symbol, compiler.semantic.type y compiler.syntax.nomTerminal, las cuales implementan las estructuras y métodos necesarios para manejar los no terminales de la gramática.
La gestión de errores se realiza mediante el gestor SemanticErrorManager. Cada error semántico se muestra con un código de error numérico único, facilitando de esta forma su localización dentro del parser.cup.
En cuanto a la comprobación de tipos, se encarga de comprobar que los tipos en expresiones, asignaciones, parámetros, etc... sean correctos.
En resumen, el analizador semántico, principalmente comprueba que los símbolos declarados no estén repetidos dentro del mismo ámbito, que los símbolos utilizados dentro de un ámbito (constantes, variables, subprogramas), estén previamente declarados, que los tipos de las expresiones sean correctos, se comprueba que los parámetros en llamadas a subprogramas sean correctos en cuanto al numero de parámetros, orden y tipo, se comprueba que exista una sentencia de retorno de valor en las funciones.

1.1.- Descripción de la tabla de símbolos implementada.

La tabla de símbolos se implementa y gestiona mediante las clases SymbolTableIF y SymbolTable, las cuales proporcionan métodos para añadir nuevos símbolos a la tabla de símbolos así como para buscar un símbolo, o recuperar un símbolo de la tabla.
Para cada ámbito creado se asocia su propia tabla de símbolos, la cual almacena los símbolos declarados en dicho ámbito (variables, subprogramas, parámetros..).
Cada símbolo almacenado en la tabla de símbolos, tiene asociado un tipo, previamente declarado y almacenado en la tabla de tipos.
A tabla de tipos almacena los tipos primitivos del lenguaje así como los tipos construidos y declarados por el usuario.
Para cada símbolo, se almacena en la tabla de símbolos la clase de símbolo que es (variable, parámetro, procedimiento, función, variable de retorno...), el ámbito en el que esta declarado, el nombre del símbolo, su tipo, ademas de su dirección relativa al ámbito en el que esta declarado y su tamaño en memoria.

2 .- Generación de código intermedio.

Para la generación del código intermedio, se utiliza un sistema de cuádruplas las cuales se construyen mediante el uso de las clases IntermediateCodeBuilder y Quadruple. Estas clases proporcionan los métodos necesarios para crear la lista de cuádruplas que formaran el programa en código intermedio, así como su gestión y manejo.
Por otro lado, en el paquete compiler.syntax.nonTerminal, en las clases creadas para el manejo de los no terminales, en aquellos que generan código intermedio, como las expresiones, se incluye un método, generateIntermediateCode, que sera el encargado de crear el conjunto de cuádruplas necesarias para cada situación del programa y añadirlas

a la lista de cuádruplas que forman todo el programa.

2.1 .- Descripción de la estructura utilizada.

Como se ha dicho en el punto anterior, el código intermedio se genera mediante cuádruplas que se almacenan en una lista, formando dicha lista el código intermedio del programa fuente. El código intermedio generado es parecido a ensamblador pero sin estar ligado a ningún hardware en concreto.
La lista que almacena el código intermedio es "List<QuadrupleIF> intermediateCode", que se define dentro de la clase NomTerminal en el paquete compiler.syntax. NomTerminal. El resto de clases de este paquete, heredan de la clase NomTerminal, permitiendo de esta forma que cada no terminal genere su código intermedio y poder propagarlo como un atributo sintetizado a través del árbol sintáctico, de manera que al alcanzar el nodo raíz, se tiene todo el código intermedio del programa fuente.
Por tanto, para crear y gestionar el código intermedio, se utilizan principalmente las clases definidas dentro de los paquetes compiler.syntax.nomterminal y compiler.intermediate.
Una cuádrupla en código intermedio tiene el siguiente formato:

INSTRUCCION operando1, operando2, operando3

Las instrucciones que forman el código intermedio son las siguientes:

Instrucción	Operando1	Operando2	Operando3	Explicación
ADD	res	op1	op2	res = op1 + op2
BR	dir			salto incondicional a dir.
BRF	dir	op1		si op1 es falso (0) salta a dir
BRI	dir	op1	op2	si op1 = op2 salta a dir.
CALL	subp			llama al subprograma subp.
EQ	res	op1	op2	si op1=op2 entonces res=1. si op1!=op2 entonces res=0.
HATL				fin de programa.
INIT	dir	label		dir → dirección de inicio del programa. Label → nombre del programa. prepara el inicio del programa. Prepara los textos de salida. Prepara el RA del programa principal. Inicia el puntero de pila SP y el puntero de frame FP(.IX). Salta a la dirección de la primera instrucción.
INL	label			Inserta la etiqueta label.
MQ	res	op1	op2	si op1>op2 entonces res=1. en otro caso res = 0.
MV	op1	op2		op1 = op2
MVA	op1	op2		op1 = &op2 Almacena en op1 la dirección de memoria de op2.
MVP	op1	op2		op1 = *op2

				Almacena en op1 el contenido de la dirección de memoria almacenada en op2.
OR	res	op1	op2	res = op1 OR op2
PARAM	op1			Introduce en la pila el parámetro op1.
RET	op1			Realiza el retorno de un subprograma al punto indicado por op1.
RETURN	op1			Devuelve el valor de retorno de una función, op1.
STP	op1	op2		*op1 = op2 Almacena op2 en la dirección de memoria almacenada en op1.
SUB	res	op1	op2	res = op1 – op2
WRITE	op1			Muestra en pantalla op1.
WRTLN				Muestra en pantalla un salto de linea.

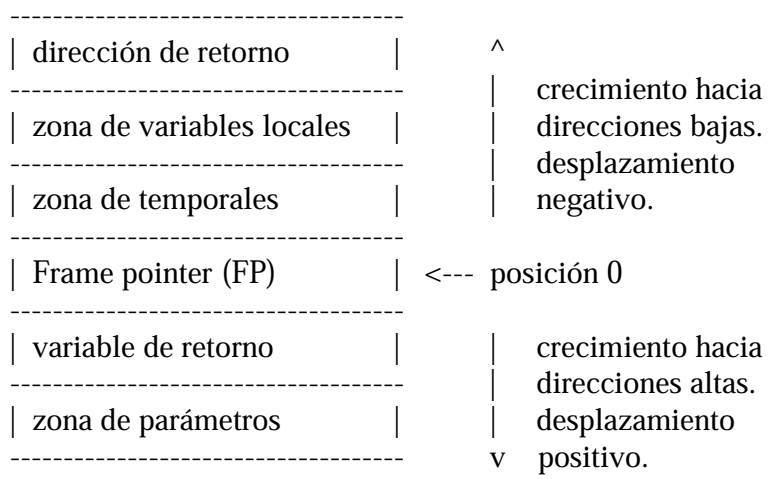
3 .- Generación de código final.

Para la creación del código final, se hace uso de las clases incluidas en el paquete compiler.code y compiler.traductor.
El método “translate” de la clase ExecutionEnviromentEns2001 es el encargado de traducir el código intermedio a código final, para ello hace uso de las clases incluidas en el paquete compiler.traductor, invocando para cada instrucción de código intermedio su traductor correspondiente.
La clase FrameManager del paquete compiler.code se encarga de gestionar los frames de cada registro de activación
La clase MemoryManager es la encargada de generar el mapa de memoria, para ello hace uso del método “mapAddresses”, este método, obtiene la lista de todos los ámbitos del programa, para cada ámbito obtiene su tabla de símbolos y para cada símbolo, comprueba el tipo de símbolo (variable, parámetro, registro, etc...), su tamaño y el ámbito al que pertenece y le asigna una dirección de memoria relativa al ámbito al que pertenece, dicha dirección de memoria, sera utilizada por el traductor para calcular la dirección de memoria real del símbolo
Después de mapear los símbolos de la tabla de símbolos, se obtiene la tabla de temporales del ámbito y se le asigna a cada temporal una dirección de memoria relativa al ámbito al que pertenece.
Una vez mapeada la memoria para símbolos y temporales, se hace la reserva de memoria para los textos que se utilizaran en las sentencias WRITE del programa, para ello se hace uso del método “memoryAllocationTextAddresses” de la clase MemoryManager.
La clase TextManager del paquete compiler.code es la encargada de gestionar los textos que serán utilizados por las sentencias WRITE del programa.
Para realizar la traducción de cada instrucción de código intermedio a código final, se ha creado el paquete compiler.traductor, que incluye una clase principal “Traductor” y una clase “Traductor_” especifica para cada una de las instrucciones de código intermedio. Cada clase “Traductor_” especifica de cada instrucción de código

intermedio hereda de la clase “Traductor” principal.
La clase “Traductor” principal, incluye un método abstracto “traducir(QuadrupleIF quadruple)” que sera implementado por cada traductor especifico y es el encargado de realizar la traducción de cada instrucción de código intermedio a código final.
La clase “Traductor” principal, también incorpora métodos para gestionar los operandos de cada instrucción de código intermedio (de cada cuádrupla), comprobar de que tipo es cada operando (variable, parámetro, etc...), así como determinar si cada operando es global o local y obtener su dirección de memoria relativa al ámbito al que pertenece

3.1 .- Descripción del registro de activación implementado.

El registro de activación (RA), tiene la siguiente estructura:



El elemento principal del RA es el frame pointer (FP), mediante el cual se accederá al resto de elementos del RA. El FP es la dirección de acceso al RA y estará ubicada en el registro .IX del ENS2001. Por tanto, usamos como referencia el registro .IX para acceder al resto de elementos. Los diferentes RA, se colocan en la pila, la cual va creciendo desde direcciones altas hacia direcciones bajas. Considerando FP como el elemento con posición 0 dentro del RA, los elementos que estén por encima de FP tendrán un desplazamiento negativo respecto a FP (temporales, variables locales y dirección de retorno), lo elementos por debajo de FP tendrán un desplazamiento positivo respecto a FP (valor de retorno y parámetros).
Por tanto, teniendo en cuenta que la dirección de memoria de FP esta almacenada en el registro .IX y que FP se considera como la posición 0 dentro del RA, al valor de retorno se accede sumando 1 al registro .IX. La posición del primer parámetro estará en la posición 2 respecto a FP, por lo tanto, la zona de parámetros tendrá un offset de 2, de esta forma, para acceder a un parámetro concreto, sumaremos a .IX el offset de la zona de parámetros y la dirección del parámetro concreto al que queremos acceder (dirección real de parámetro = .IX + offsetParametros + dirección de parámetro).
La zona de temporales esta justo encima de FP por lo que tiene un desplazamiento negativo, por tanto, para acceder a un temporal concreto se resta a .IX la dirección del temporal al que queremos acceder (dirección real de temporal = .IX – dirección de temporal).
La zona de variables locales se sitúa justo después de la zona de temporales, por lo que la zona de variables locales tendrá un desplazamiento negativo respecto a FP igual al tamaño de la zona de temporales, para calcular el offset de la zona de variables locales, la clase “Traductor” incorpora el método “getVariableMemoryOffset” que devuelve el tamaño de la zona de temporales o lo que es lo mismo, el desplazamiento de la zona de variables locales. De esta forma, para acceder a una variable local concreta, se resta a

.IX el offset de la zona de variables locales y la dirección relativa de la variable local a la que queremos acceder (dirección real de variable local = .IX – offsetVariables – dirección de variable local).
Por ultimo, la dirección de retorno, almacena la dirección de memoria a la que se devuelve la ejecución una vez haya terminado la ejecución del subprograma en curso.

4 .- Indicaciones especiales.

Puesto que el lenguaje no es sensible a mayúsculas y minúsculas, identificadores como "Variable", "variable" o "VaRiAbLe" son idénticos, por tanto, se ha optado por transformarlos a mayúsculas para su tratamiento.
En la dirección /0 de memoria se almacena la dirección de memoria donde comienza el código del programa.
La zona de variables globales comienza en la dirección de memoria /2, creciendo hacia direcciones altas.
Después de la zona de variables globales se ubica la zona de textos, donde se almacenan las cadenas de texto que serán utilizadas por sentencias WRITE.
Después se ubica la zona de código, que aloja el código ejecutable del programa y al igual que las anteriores, crece hacia direcciones altas.
La pila comienza en la dirección de memoria /64999 y crece hacia direcciones bajas. Se utiliza para almacenar los distintos registros de activación que se van generando durante la ejecución.
Por ultimo, esta la zona de stack, que almacena las direcciones de frame de cada registro de activación. La zona de stack crece hacia direcciones altas.
Para la generación de errores semánticos, se ha numerado cada error con un numero único desde 01 hasta 53, con el formato "error nº : descripción del error", facilitando de esta forma su localización dentro del parser.cup.

5 .- Conclusiones.

Se trata de una practica muy completa que hace comprender la complejidad del trabajo que realiza un compilador dentro de la sencillez del propio concepto de compilador.
Ha sido, sin ninguna duda, la practica mas compleja, difícil y larga de todo el grado, por otro lado, también ha sido la mas atractiva de hacer y la mas emocionante.
En cuanto a la practica en si, según se van completando fases, se hace mas complicada, fase de análisis semántico, fase de traducción a código intermedio y fase de traducción a código final, que desde mi punto de vista es la mas complicada. Ademas, ha sido frecuente tener que rehacer o rediseñar algunas partes de una fase ya realizada, durante la realización de otra fase, por ejemplo, rediseñar partes del código intermedio durante la realización del código final.
En cuanto al desarrollo de la practica en si, ha sido muy laborioso, principalmente a la falta de un entorno adecuado para la depuración del código, por ejemplo, no se puede hacer una ejecución paso a paso (o al menos yo no lo he descubierto), por lo que en la mayoría de las ocasiones, para descubrir un error en el código, he tenido que introducir "mensajes de control" en el código, o bien mediante semanticDebug o semanticInfo en el parser.cup o mediante system.out.print en las clases de java, de esta forma, si un "mensaje de control" que se espera ver en pantalla no aparece, es indicativo de que hay un error antes de ese mensaje de control, en resumen, la depuración del código es muy engorrosa y para realizarla se necesita tanto o mas tiempo que el empleado en realizar el código.

6 .- Gramática.

Se ha utilizado la gramática proporcionada por el equipo docente.

Terminales.

terminal Token LITERAL_ENTERO;
terminal Token LITERAL_CADENA;
terminal Token ID;
terminal Token BEGIN;
terminal Token BOOLEAN;
terminal Token CONST;
terminal Token DO;
terminal Token ELSE;
terminal Token END;
terminal Token FALSE;
terminal Token FOR;
terminal Token FUNCTION;
terminal Token IF;
terminal Token INTEGER;
terminal Token OR;
terminal Token PROCEDURE;
terminal Token PROGRAM;
terminal Token RECORD;
terminal Token THEN;
terminal Token TO;
terminal Token TRUE;
terminal Token TYPE;
terminal Token VAR;
terminal Token WRITE;
terminal Token WRITELN;
terminal Token PARENT_ABRIR;
terminal Token PARENT_CERRAR;
terminal Token COMA;
terminal Token PUNTO_Y_COMA;
terminal Token DOS_PUNTOS;
terminal Token IGUAL;
terminal Token MENOS;
terminal Token MAYOR_QUE;
terminal Token DOS_PUNTOS_IGUAL;
terminal Token PUNTO;
terminal Token CIRCUNFLEJO;
terminal Token ARROBA;

No terminales.

non terminal program;
non terminal Axiom axiom;
non terminal Declaraciones declaraciones;
non terminal Bloque bloque;
non terminal Secuencia_sentencias secuencia_sentencias;
non terminal Sentencia sentencia;
non terminal decl_tipos;

```
non terminal                                    decl_variables;
non terminal                                    decl_subprogramas;
non terminal                                    secuencia_constantes;
non terminal                                    constante;
non terminal   Literal_entero_o_logico          literal_entero_o_logico;
non terminal   Literal_logico                   literal_logico;
non terminal                                    secuencia_tipos;
non terminal                                    tipo;
non terminal                                    secuencia_variables;
non terminal   Variabl                          variabl;
non terminal   Secuencia_IDs                    secuencia_IDs;
non terminal   Tipo_datos                       tipo_datos;
non terminal   Tipo_primitivo                   tipo_primitivo;
non terminal   Secuencia_campos_registro        secuencia_campos_registro;
non terminal   Campo_registro                   campo_registro;
non terminal   Decl_procedimiento               decl_procedimiento;
non terminal                                    decl_funcion;
non terminal                                    decl_parametros;
non terminal   Secuencia_parametros             secuencia_parametros;
non terminal   Parametro                        parametro;
non terminal   Exp                              exp;
non terminal   Secuencia_parametros_llamada     ecuencia_parametros_llamada;
non terminal   Sentencia_asignacion             sentencia_asignacion;
non terminal   Sentencia_asignacion_izq         sentencia_asignacion_izq;
non terminal   LLamada_subprograma              llamada_subprograma;
non terminal   Sentencia_if                     sentencia_if;
non terminal   Sentencias_then_else_for         sentencias_then_else_for;
non terminal   Sentencia_for                    sentencia_for;
non terminal   Sentencia_write                  sentencia_write;
non terminal                                    parametro_write;
non terminal                                    parte_else;
```

Relaciones de precedencia.

```
precedence left         ELSE;
precedence nonassoc     MAYOR_QUE, IGUAL;
precedence left         MENOS, OR;
precedence left         PUNTO, PARENT_ABRIR, PARENT_CERRAR;
```

Reglas de producción.

```
start with program;

program ::=
  {:
      syntaxErrorManager.syntaxInfo ("Starting parsing...");
  :}
  axiom:ax
  {:
          // No modificar esta estructura, aunque se pueden añadir más acciones
          semánticas
          // Para la entrega de febrero pueden comentarse las sentencias siguientes:
                  List intermediateCode = ax.getIntermediateCode ();
```

```
        finalCodeFactory.setEnvironment(CompilerContext.getExecutionEnvironment());
          finalCodeFactory.create (intermediateCode);
          // En caso de no comentarse las sentencias anteriores puede generar una
          excepcion
          // en las llamadas a cupTest y finalTest. Esto es debido a que
          // aún no se tendrá implementada la generación de código intermedio ni final.
          // Para la entrega de junio y septiembre deberán descomentarse y usarse.
  :};

axiom ::= PROGRAM ID PUNTO_Y_COMA declaraciones bloque PUNTO;

// Las declaraciones incluyen, de forma opcional pero siempre en este orden:
// constantes, tipos, variables globales y subprogramas
declaraciones ::= CONST secuencia_constantes decl_tipos
                | decl_tipos;

decl_tipos ::= TYPE secuencia_tipos decl_variables
                | decl_variables;

decl_variables ::= VAR secuencia_variables decl_subprogramas
                | decl_subprogramas;

decl_subprogramas ::= decl_subprogramas decl_procedimiento
                | decl_subprogramas decl_funcion
                | ;

// Declaración de constantes (del programa principal o de un subprograma)
secuencia_constantes ::= secuencia_constantes constante | constante;

constante ::= ID IGUAL literal_entero_o_logico PUNTO_Y_COMA;

literal_entero_o_logico ::= LITERAL_ENTERO
                | literal_logico;

literal_logico ::= TRUE
                | FALSE;

// Declaración de tipos (del programa principal o de un subprograma)
secuencia_tipos ::= secuencia_tipos tipo | tipo;

tipo ::= ID IGUAL RECORD secuencia_campos_registro END PUNTO_Y_COMA ;

secuencia_campos_registro ::= secuencia_campos_registro campo_registro
                        | campo_registro;

campo_registro ::= ID DOS_PUNTOS tipo_primitivo PUNTO_Y_COMA;

tipo_primitivo ::= INTEGER
                | BOOLEAN
                | CIRCUNFLEJO INTEGER;

// Declaración de variables (del programa principal o de un subprograma)
secuencia_variables ::= secuencia_variables variabl | variabl;
```

```
variabl ::= secuencia_IDs DOS_PUNTOS tipo_datos PUNTO_Y_COMA;

secuencia_IDs ::= secuencia_IDs COMA ID
                | ID;

tipo_datos ::= tipo_primitivo
                | ID;

// Declaración de procedimientos
decl_procedimiento ::= PROCEDURE ID decl_parametros PUNTO_Y_COMA
                declaraciones bloque PUNTO_Y_COMA;

decl_parametros ::= PARENT_ABRIR secuencia_parametros PARENT_CERRAR
                | PARENT_ABRIR PARENT_CERRAR;

secuencia_parametros ::= secuencia_parametros PUNTO_Y_COMA parametro
                        | parametro;

parametro ::= secuencia_IDs DOS_PUNTOS tipo_primitivo;

// Declaración de funciones
decl_funcion ::= FUNCTION ID decl_parametros DOS_PUNTOS tipo_primitivo
        PUNTO_Y_COMA declaraciones bloque PUNTO_Y_COMA;

// Llamada a función o procedimiento (se diferencia en el análisis semántico de la
        llamada)
llamada_subprograma ::= ID PARENT_ABRIR secuencia_parametros_llamada
        PARENT_CERRAR
                | ID PARENT_ABRIR PARENT_CERRAR;

secuencia_parametros_llamada ::= secuencia_parametros_llamada COMA exp
                | exp;

// Expresiones
exp ::= LITERAL_ENTERO
                | ID
                | ID CIRCUNFLEJO
                | ID PUNTO ID
                | ID PUNTO ID CIRCUNFLEJO
                | ARROBA ID
                | ARROBA ID PUNTO ID
                | TRUE
                | FALSE
                | exp MENOS exp
                | exp OR exp
                | exp MAYOR_QUE exp
                | exp IGUAL exp
                | PARENT_ABRIR exp PARENT_CERRAR
                | llamada_subprograma;

// Bloque de sentencias
bloque ::= BEGIN secuencia_sentencias  END
                | BEGIN END ;
```

```
secuencia_sentencias ::= secuencia_sentencias sentencia
                       | sentencia;

sentencia ::= sentencia_asignacion
            | sentencia_if
            | sentencia_for
            | llamada_subprograma PUNTO_Y_COMA
            | sentencia_write;

sentencia_asignacion ::= sentencia_asignacion_izq DOS_PUNTOS_IGUAL exp
        PUNTO_Y_COMA ;

sentencia_asignacion_izq ::= ID
            | ID CIRCUNFLEJO
            | ID PUNTO ID
            | ID PUNTO ID CIRCUNFLEJO;

// Sentencia de control de flujo condicional if-then-else
sentencia_if ::= IF PARENT_ABRIR exp PARENT_CERRAR THEN
        sentencias_then_else_for parte_else;

sentencias_then_else_for ::= bloque PUNTO_Y_COMA
            | sentencia;

parte_else ::= ELSE sentencias_then_else_for
            | ;

// Sentencia de control de flujo iterativo for (las sentencias dentro de un bucle for siguen
        las mismas normas que para el if)
sentencia_for ::= FOR PARENT_ABRIR ID DOS_PUNTOS_IGUAL exp TO exp
        PARENT_CERRAR DO sentencias_then_else_for;

// Sentencias write y writeln
sentencia_write ::= WRITE PARENT_ABRIR parametro_write PARENT_CERRAR
        PUNTO_Y_COMA
         WRITELN PARENT_ABRIR PARENT_CERRAR PUNTO_Y_COMA;

parametro_write ::= exp
            | LITERAL_CADENA
            | ;
```