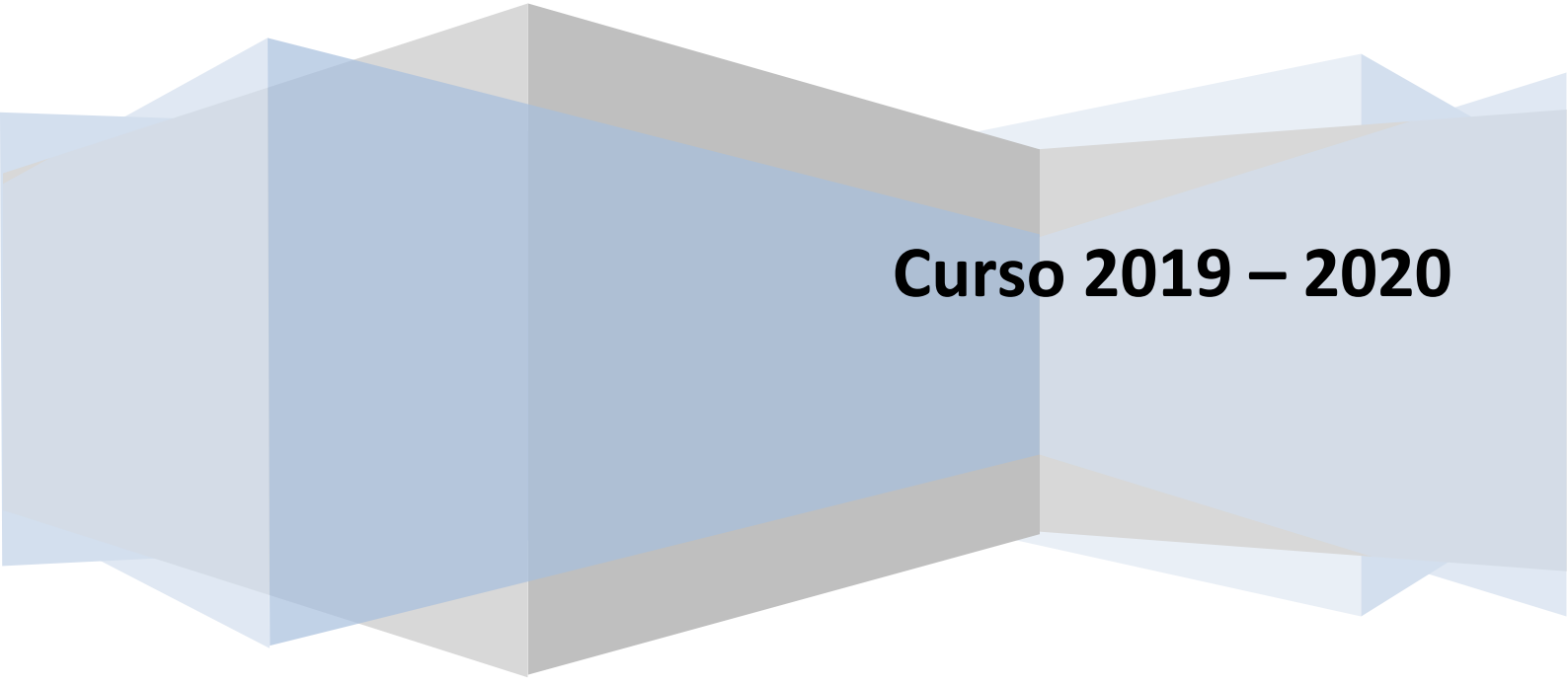
Universidad Nacional de Educación a Distancia
Departamento de Lenguajes y Sistemas Informáticos

Práctica de Procesadores del Lenguaje II

Directrices de implementación (v1.0)

Dpto. de Lenguajes y Sistemas Informáticos
ETSI Informática, UNED

Anselmo Peñas (coordinador)
Laura Plaza
Alvaro Rodrigo



Curso 2019 – 2020

Contenido

1 Introducción

La construcción de un compilador es una compleja labor que requiere atender a numerosas cuestiones técnicas como la implementación de estructuras de datos orientadas a objetos, la organización de éstas en paquetes, el uso de diversos patrones de diseño, etc.

Para el desarrollo de la práctica se proporciona al alumno un marco de trabajo de referencia. Este marco de trabajo se proporciona con el siguiente fin:

- Reducir el tiempo de puesta en marcha del entorno necesario para el desarrollo de la práctica.
- Acelerar la adquisición de buenas prácticas en el desarrollo de un compilador.
- Centrar el trabajo del alumno en el desarrollo de aquellos aspectos que inciden directamente con los contenidos propios de la asignatura.

En este documento se presenta el marco de trabajo software que se proporciona al alumno y se describe detalladamente el trabajo que el alumno debe realizar para superar satisfactoriamente la práctica. La corrección de la práctica se apoya en la organización software que prescribe este marco de trabajo y, por tanto, es normativo que el alumno respete completamente dicha organización. Cualquier violación a este respecto podría suponer un suspenso en el proceso de corrección.

Para la comprensión de este documento se presupone que el alumno ha estudiado previamente la teoría de la asignatura y que por tanto está capacitado para entender de forma correcta las explicaciones y terminología ofrecida.

Las normas de entrega y división del trabajo deben consultarse en el enunciado de la práctica. En este documento se dan unas indicaciones de cómo resolver el análisis semántico, generación de código intermedio y código final. Al final de todo el proceso el alumno habrá desarrollado un compilador capaz de tomar un código fuente y generar un fichero de texto con instrucciones ensamblador que pueda ser interpretado por el emulador ENS2001.

En caso de correcciones sobre el marco de trabajo, se anunciarán en el Tablón de Anuncios y los foros del curso virtual de la asignatura. *Es necesario por tanto que el estudiante visite con asiduidad ambos.*

2 Marco de trabajo y desarrollo de la práctica

El desarrollo de la práctica se realiza sobre un marco de trabajo tecnológico y por tanto constituye un recurso necesario para el desarrollo de la misma. El marco de trabajo se entrega como un fichero comprimido que proporciona dos elementos fundamentales:

- **Una estructura de directorios para que el alumno organice su código**. Estos directorios contiene ficheros fuente en java con código de andamiaje que el alumno deberá completar y / o modificar para implementar las distintas funcionalidades del compilador. Estas clases están organizadas jerárquicamente en paquetes siendo **compiler** el paquete raíz. Por tanto, cualquier clase dentro del paquete compiler (o alguno de sus paquetes hijos) es una clase abierta que el alumno puede y debe modificar. Consulte la API en el marco de trabajo y una descripción más detallada en la sección 2.1 para obtener más información.

- **Una librería en java que proporciona clases de soporte para el desarrollo del compilador**. Todas las clases abiertas del paquete compiler se apoyan en clases definidas dentro de la librería compiler-api.jar. Esta librería contiene, básicamente, clases de ayuda, clases abstractas e interfaces que utilizan, extienden o implementan la mayoría de las clases del paquete compiler (y sus paquetes hijos). Se trata de una familia de clases cerradas y compiladas, por lo que el alumno no dispone de su código fuente para alterarlas ya que NO debe ni puede modificarlas. Todas las clases de soporte, dentro de la liberia, están organizadas jerárquicamente en paquetes siendo **es.uned.lsi.compiler** el nombre del paquete padre. Por tanto toda clase dentro de éste paquete (o alguno de sus paquetes hijos) es una clase de soporte proporcionada por el equipo docente que no debe modificarse. Consulte la API en el marco de trabajo o el resto del documento para obtener una información más detallada.

Tanto la estructura de directorios como las implementaciones y clases de referencia están en inglés. Se presupone, por tanto, que el alumno está familiarizado con la terminología inglesa a la hora de programar. No es necesario que el alumno complete la práctica en inglés, pero no ha de traducir el código proporcionado.

Además, debe quedar claro que la práctica se desarrolla en Java y dentro del paradigma de orientación a objetos. Existe una amplia literatura sobre este lenguaje de programación cómo referencias podemos destacar http://www.oracle.com/technetwork/java/index.html y http://www.javahispano.org/. En esta última se puede encontrar documentación en castellano.

2.1 La estructura de directorios

A continuación se procede a explicar la estructura de directorios del marco de trabajo, deteniéndonos en profundidad en cada uno de los paquetes donde debe almacenarse el código.

- **src**. En este directorio debe almacenarse todo el código fuente del compilador. Las clases están organizadas en diferentes paquetes. El paquete principal se llama 'compiler' y dentro existen otros paquetes: uno por cada una de las fases conceptuales que constituyen el compilador. En este directorio deben almacenarse todo el código fuente del compilador organizado en paquetes tal y como describiremos a continuación. El contenido de este directorio contiene algunas clases de soporte de las que el alumno debe partir para el desarrollo de la práctica.

  - **/compiler/lexical**. Este directorio debe almacenar las clases del paquete 'compiler.lexical' que implementan el scanner de la práctica y todas sus clases asociadas (incluido scanner.java). En este paquete se incluye la clase Token.java que el alumno debe utilizar y puede completar y/o extender para el analizador léxico.

  - **/compiler/syntax**. Este directorio debe almacenar todas las clases del paquete 'compiler.syntax' que implementan el parser y todas sus clases asociadas (incluidas parser.java y sym.java).

  - **/compiler/syntax/nonTerminal.** En este directorio se almacenarán las clases que representan los símbolos no terminales necesarios para llevar a cabo la traducción dirigida por la sintaxis en cup como comentaremos más adelante. Se incluyen las clases Axiom y NonTerminal. La primera representa al axioma de la gramática. La segunda representa la clase abstracta de la que deben extender las clases de no terminales creadas por el estudiante.

  - **/compiler/semantic.** Contendrá las clases necesarias para el análisis semántico. Está compuesto de varios paquetes organizativos.

  - **/compiler/semantic/symbol.** En este directorio se encuentran las clases necesarias para utilizar la tabla de símbolos, incluyendo los diferentes símbolos que puede almacenar.

  - **/compiler/semantic/type.** En este directorio se almacenan las clases necesarias para utilizar el sistema de tipos del lenguaje.

  - **/compiler/intermediate.** En este directorio estarán las clases necesarias para la generación del código intermedio mediante cuartetos (también llamados cuádruplas) junto con los tipos de operandos que admiten.

  - **/compiler/code.** En este directorio se situarán las clases necesarias para la generación del código final.

  - **/compiler/test.** En este directorio se encuentran tres clases llamadas LexicalTestCase.java, SyntaxTestCase.java y FinalTestCase.java que sirven para

probar el analizador léxico, sintáctico y la entrega final respectivamente. Más adelante se detallará cómo ejecutar y probar la práctica.

- **classes**. En este directorio se almacenarán las clases (archivos de extensión '.class') que se generan mediante el proceso de compilación del código fuente del compilador. El alumno no debe modificar este directorio, ya que la herramienta Ant, que presentaremos más adelante, se encarga de automatizar las tareas de compilación. Es importante que los ficheros '.class' se sitúen en este directorio y no en otros creados automáticamente por algunos IDEs, como por ejemplo el directorio bin.

- **lib**. Este directorio contiene las librerías necesarias para la realización de la práctica (archivos con extensión .jar). Aquí se incluyen las librerías *jflex.jar* y *cup.jar* necesarias para llevar a cabo el proceso de generación del scanner y el parser respectivamente, y *compiler-api.jar* que contiene las clases e interfaces necesarias para el desarrollo del compilador. El contenido de esta última librería será discutida en detalle más adelante. Todas las librerías bajo el directorio /lib NO deben eliminarse bajo ningún concepto. Aunque no es necesario, si el alumno considera oportuno hacer uso de otras librerías de apoyo debe incluirlas en este directorio e indicarlo en la memoria.

- **doc**. Este directorio contiene diferentes tipos de documentos necesarios para la realización y documentación de la práctica organizados de acuerdo a una serie de subdirectorios que se explican a continuación:

  - **/api**. En este directorio se almacena una completa descripción de las clases (el API) que constituyen el marco de trabajo incluido en la librería 'compiler-api.jar'. El formato de la documentación responde al estándar 'javadoc' definido por Sun. Para visualizarlo en forma de página Web ha de accederse a index.html.

  - **/config.** En este directorio se proporcionan el fichero de configuración de la aplicación: 'build.xml' que incluye las descripción de las tareas Ant para realizar la práctica (véase más adelante una descripción en profundidad de este documento). Nótese, no obstante, que el alumno no requiere comprender el contenido de este fichero sino solamente saber utilizarlo con el uso de la herramienta Ant.

  - **/memoria**. Aquí debe incluirse la memoria de la práctica según las normas explicadas en el documento del enunciado de la práctica.

  - **/specs**. Este directorio incluye los ficheros para generar el analizador léxico o scanner (scanner.flex) y el analizador sintáctico o parser (parser.cup). Para esta práctica se proporciona por parte del Equipo Docente una implementación de estos ficheros. El alumno puede partir de esta implementación, modificarla o realizar la suya propia. Para un correcto funcionamiento NO se debe cambiar su ubicación a otro directorio.

  - **/test**. Contiene los ficheros de prueba para probar el compilador desde la tarea ant correspondiente (ver más adelante). Cualquier fichero fuente de prueba que desee utilizar el alumno deberá almacenarse en este directorio.

2.2 La librería compiler - api

La librería de referencia compiler-api.jar contiene una familia de clases organizadas en diferentes paquetes que deben ser utilizados por el alumno para implementar el compilador. En el directorio /doc/api puede encontrar una completa descripción técnica del contenido de este fichero jar. No obstante, a continuación describimos el uso, utilidad y organización en paquetes de cada una de las clases incluidas:

- Paquete **es.uned.lsi.compiler.lexical**. Este paquete contiene las clases que el alumno debe utilizar para desarrollar el analizador léxico de su compilador. Recordamos que se proporciona una implementación que el alumno puede utilizar.

- Paquete **es.uned.lsi.compiler.syntax**. Este paquete contiene las clases que el alumno debe utilizar para desarrollar el analizador sintáctico de su compilador.

- Paquete **es.uned.lsi.compiler.syntax.nonTerminal**. Este paquete contiene la interfaz NonTerminalIF de la que extiende la clase abstracta NonTerminal.

- Paquete **es.uned.lsi.compiler.semantic** Contiene la clase SemanticErrorManager (ver apartado de análisis semántico) usada para indicar los errores semánticos junto con métodos para ayudar a realizar trazas personalizadas (semanticDebug y semanticInfo). Su uso es similar al de LexicalErrorManager o SyntaxErrorManager. Además, en este paquete residen las clases Scope y ScopeManager necesarias para dar soporte a la gestión de ámbitos de visibilidad.

- Paquete **es.uned.lsi.compiler.semantic.symbol** Contiene la colección de clases necesarias para dar soporte a los símbolos que aparecen durante la compilación de un programa. En concreto nos referimos al interfaz SymbolIF y a la clase base que lo extiende SymbolBase y de la cual derivan todas las clases abiertas que el alumno deberá modificar en el paquete compiler.semantic.symbol para realizar su compilador tal y como se describirá más adelante. También se ofrece una *implementación completa de la tabla de símbolos denominada SymbolTable*.

- Paquete **es.uned.lsi.compiler.semantic.type** Contiene las clases necesarias para dar soporte al sistema de tipos del compilador. En concreto allí se puede encontrar el interfaz TypeIF y la clase abstracta que implementa dicho interfaz TypeBase. También hay definido en este paquete el interfaz TypeTableIF que es implementado por la clase TypeTable, también proporcionada. Por lo tanto, al igual que la tabla de símbolos, no es necesario implementar la tabla de tipos.

- Paquete **es.uned.lsi.compiler.intermediate**. Dentro de este paquete aparecen todos los artefactos necesarios para la generación de código intermedio. En este sentido se incluyen las interfaces e implementaciones para gestionar cuartetos, temporales, etiquetas y variables. También se incluye la clase IntermediateCodeBuilder necesaria para construir de forma sencilla las colecciones de cuartetos.

- Paquete **es.uned.lsi.compiler.code** En el paquete code residen los artefactos necesarios para general código final a partir de código intermedio. La clase FinalCodeFactory organiza la

generación de código final y debe usarse para crear el código final a partir de la lista de cuádruplas. Para la conversión de cuádruplas a código final se define el entorno de ejecución, a partir del interfaz ExecutionEnvironmentIF y que el alumno debe completar en la clase ExecutionEnvironmentEns2001, situada en el paquete compiler.code. También se ofrecen las interfaces MemoryDescriptorIF y RegisterDescriptorIF implementadas, igualmente, en el paquete compiler.code cuyo uso, opcional, asiste en las tareas de gestión de registros máquina y memoria

2.3 Instalación

La instalación del marco de trabajo es un proceso sencillo que puede resumirse en la secuencia de pasos que exponemos a continuación:

- Entre al entorno virtual de la asignatura
- Acceda al apartado correspondiente a la práctica
- Descargue el fichero zip correspondiente a la arquitectura (para cada grupo A o B se proporciona una arquitectura donde solo varían los ficheros scanner.flex y parser.cup).
- Cree una carpeta para albergar el desarrollo del proyecto, por ejemplo PL2.
- Descomprima el fichero de la arquitectura bajo ese directorio

Si desea utilizar algún entorno de desarrollo integrado puede hacerlo. En ese caso debería crear un nuevo proyecto y ubicar allí la estructura de directorios de la arquitectura asegurándose de indicar que el directorio /scr contiene los fuentes del proyecto y que la versión compilada de los mismos deberá redirigirse al directorio /classes.

2.4 Desarrollo

Las tareas de generación, compilación, ejecución y prueba que son necesarias para realizar el compilador se han automatizado mediante la herramienta Ant. Esta es una herramienta que permite automatizar diferentes tipos de tareas descritas en un script expresado como un documento en XML (normalmente llamado build.xml). Su uso resulta similar a los mecanismos de procesamiento por lotes proporcionados por algunos sistemas operativos, que usan ficheros '.bat' o '.sh' para describir un script de actuación o a las herramientas Makefile utilizadas por algunos compiladores para automatizar los procesos de compilación y enlazado de acuerdo a ficheros '.mak'.

El primer paso para utilizar la herramienta ANT es la instalación de la misma. Se puede encontrar toda la información acerca de ella en la siguiente dirección Web:

```
http://ant.apache.org
```

El documento build.xml proporcionado en el directorio doc/config contiene la descripción de todas las tareas necesarias para realizar el compilador. No obstante, para que éstas funcionen es necesario importar JFlex.jar y Cup.jar que se incluyen en el directorio /lib. Por ello, se recomienda copiar ambos archivos dentro del directorio 'lib' de Ant. Por ejemplo, si tenemos Ant instalado en C:/ant, debemos copiar los dos archivos en el directorio C:/ant/lib

Para invocar las tareas de Ant desde la línea de comandos debe abrirse una consola del sistema y situarse sobre el directorio que contenga el fichero con la especificación de las tareas (en nuestro caso build.xml) y escribir:

```
ant nombreTarea –Dpropiedad=valor
```

Donde 'nombreTarea' es el nombre de una tarea definida en el fichero de tareas XML, 'propiedad' el nombre de una propiedad definida en ese fichero y 'valor' el valor que tendrá esa propiedad, siendo este parámetro opcional. Además los ficheros de descripción de tareas de Ant permiten especificar una tarea por omisión. En ese caso no haría falta poner el nombreTarea tampoco[1].

El fichero build.xml que se incluye contiene la definición de las siguientes tareas, que están a disposición del alumno:

- clear. Borra los archivos generados por procesos anteriores de compilación.
- jflex. Lee el archivo 'doc/specs/scanner.flex' y genera el código fuente del analizador léxico asociado en el archivo 'src/compiler/lexical/Scanner.java'.
- cup. Lee el archivo 'doc/specs/parser.cup' y genera los archivos de código fuente asociados 'src/compiler/syntax/sym.java' y 'src/compiler/syntax/parser.java' que corresponden a la implementación del analizador sintáctico.
- build. Llama por orden a las tres tareas anteriores y compila todos los archivos fuente, dejando las clases generadas en el directorio 'classes'. Queda así generado el compilador.
- flexTest. Llama al método main de la clase 'LexicalTestCase' pasándola cómo parámetro el nombre de un fichero situado en el directorio /doc/test. Para indicar qué fichero queremos compilar se ha de modificar en el fichero build.xml la siguiente línea:

  ```
  <property name="test-file-name" value="testA.ha" />
  ```

  Cambiando 'value' con el nombre del fichero que queramos. Los casos de prueba que desarrolle el alumno deberán situarse en el directorio /doc/test y llamarlos tal y como se ha indicado. Otra opción es pasar este valor como parámetro en la llamada a Ant. Por ejemplo:

  ```
  Ant flexTest –Dtest-file-name=testB.ha
  ```

- cupTest. Llama al método main de la clase SyntaxTestCase de forma similar a lo descrito en la tarea 'flexTest'. En este caso se probará el analizador léxico y el sintáctico.
- finalTest. Llama al método main de la clase FinalTestCase de forma similar a flexTest y cupTest. Esta clase prueba el funcionamiento del compilador generando los ficheros con el código final.

[1] Existen también entornos de desarrollo integrados (IDEs) que disponen de la herramienta Ant incorporada lo que simplifica su uso. Un ejemplo de tales entornos es Eclipse (www.eclipse.org)

Estas tres últimas tareas de test invocan previamente a las tareas clear y build, por lo que no es necesario llamarlas previamente.

El alumno puede utilizar estas tareas y, si lo desea, diseñar otras nuevas. Sin embargo, para la **entrega final** es fundamental comprobar que la tarea '*finalTest*' funciona correctamente con nuestro compilador y genera el código final esperado.

3 Entrega

Todas las clases de soporte de la librería compiler-api.jar pueden ser utilizadas para la entrega. Asimismo, todas las clases abiertas dentro del paquete compiler (o alguno de sus paquetes hijos) deberían ser implementadas (probablemente junto con algunas otras definidas por el estudiante). Deben proporcionarse las especificaciones del analizador JFlex y Cup (que pueden partir de las proporcionadas por el Equipo Docente) junto con las clases escritas y sobrescritas por el alumno y aquellas generadas de forma automática por las herramientas. Adviértase no obstante que el alumno NO debe hacer ninguna modificación directa sobre éstas últimas (Scanner.java, sym.java y parser.java) ya que podrán ser sobrescritas durante el proceso de evaluación.

3.1 Análisis sintáctico

Para la realización de la práctica, tanto el análisis semántico como la generación de código intermedio están guiados por la sintaxis. Esto implica que se tendrán que hacer ciertas modificaciones dentro de la gramática de partida. En los siguientes apartados se entrará más en profundidad a describir tales modificaciones. Sin embargo, detallaremos aquí los mecanismos que proporciona Cup para soportar esquemas de traducción dirigidos por la sintaxis.

3.1.1 Gramáticas con atributos

Los esquemas de traducción dirigidos por la sintaxis que se estudian a nivel teórico para aportar semántica a las construcciones de un lenguaje de programación están soportados por la existencia de una colección de atributos asociados tanto a los símbolos terminales como los no terminales de la gramática.

Cup da soporte a esta idea, permitiendo que se implemente para cada símbolo (terminal y no terminal) de la gramática una clase java que contenga la colección de atributos necesarios para dar soporte a los aspectos semánticos y de generación de código.

Por tanto, lo primero que debe hacer el alumno para realizar esquemas de traducción dirigidos por la sintaxis es implementar la familia de clases que representarán cada uno de los no terminales de su gramática. Esto se resume en un proceso en 2 pasos:

- Implementar una clase para cada no terminal de la gramática que requiera atributos. Aunque no es prescriptivo, se recomienda que el nombre de cada clase corresponda al del no terminal asociado respetando los criterios de nombrado de clases impuestos por Java. Estas clases puede dejarse inicialmente vacías, ya que sus atributos necesarios se irán descubriendo a medida que el alumno avance con el desarrollo de la práctica. Todas estas clases deben almacenarse en el paquete compiler.syntax.nonTerminal.

- Tipificar cada no terminal con la clase apropiada. En la sección de declaraciones de no terminales es necesario insertar entre la palabra clave 'non terminal' y el nombre del no terminal, el nombre de la clase que implementa dicho no terminal. Para que Cup reconozca estas clases asegúrese de que en la sección de importaciones existe una clausula import del estilo import compiler.syntax.nonTerminal.*

A continuación se exponen algunos consejos y advertencias de carácter general que pueden ser de utilidad a la hora de implementar estas clases:

- **Constructores**. Para cada clase no terminal es aconsejable implementar un constructor sin argumentos que contenga la lógica de inicialización de los atributos (si la hubiere). Los demás constructores deberían apoyarse en este constructor invocando a 'this ()' como primera instrucción.

- **Constructores de copia**. Como discutiremos en la siguiente subsección, en muchas ocasiones las acciones semánticas se limitan a propagar ascendentemente un atributo o conjunto de atributos (objeto) que ha sido construido en nodos inferiores del árbol de análisis sintáctico. Para este caso es especialmente aconsejable implementar en cada no terminal cuantos constructores de copia sean necesarios. Un constructor de copia es un constructor que toma como parámetro a otro objeto (en este contexto un no terminal) y copia todos sus atributos.

- **Atributos compuestos**. En la teoría de la traducción dirigida por la sintaxis, los atributos son de tipo simple. Sin embargo, en Cup, y en esta práctica en particular, se permite utilizar atributos complejos, tales como pilas, listas o tablas hash, para los atributos de las clases no terminales. Por ejemplo, es posible asociar al no terminal que representa una lista de identificadores un atributo de tipo java.util.List para propagar la colección de lexemas de los identificadores que han sido recopilados al construir el árbol de análisis sintáctico.

- **Métodos de ayuda**. Asimismo es recomendable implementar en cada clase no terminal tantos métodos de ayuda como sean necesario para simplificar la especificación semántica en Cup. En el ejemplo anterior se puede implementar un método 'addIdentifier' que añada un nuevo lexema de identificador a la lista de lexemas interna.

3.1.2 Acciones semánticas

Para asignar valor a los atributos gramaticales de los no terminales y en general para realizar cualquier acción que permita definir la semántica de una construcción del lenguaje en cup se utilizan esquemas de traducción dirigidos por la sintaxis.

Los esquemas de traducción dirigidos por la sintaxis permiten incorporar fragmentos de código ejecutable en cualquier lugar en la parte derecha de una regla. En cup estos esquemas de traducción se especifican en java separados por los caracteres {: y :}. En el listado 1 se muestra un ejemplo abstracto de acción semántica insertada entre los terminales B y C de una regla de producción de X.

Listado 1. Ejemplo acción semántica

```
X ::= A B {: <<código java aquí>>  :} C D ;
```

A todos los efectos, Cup traduce este código a la construcción gramatical equivalente que aparece en el listado 2. Esto quiere decir que la inserción de acciones semánticas supone por definición una alteración de la gramática que puede desencadenar en el surgimiento de nuevos conflictos, aunque esto en la práctica es bastante poco probable.

Listado 2. Interpretación gramatical de acciones semánticas en Cup

```
X ::= A B Y C D;

Y ::= { : <<código java aquí>>  :} ;
```

3.1.3 Propagación de atributos

En los esquemas de traducción dirigidos por la sintaxis la existencia de acciones semánticas que propagan atributos entre los símbolos de una gramática permite hablar de atributos sintetizados o heredados en función de si dicha propagación es desde los hijos al padre o desde el padre a los hijos o entre hermanos. No obstante, Cup sólo da soporte a atributos sintetizados por tanto solo es posible incluir acciones semánticas que construyan un objeto de la clase esperada por el antecedente de la regla (según la tipificación en la sección de declaración de no terminales) y la propaguen ascendentemente en el momento de la reducción. Esto en Cup se hace con la palabra reservada RESULT. El listado 3 es un ejemplo de esta propagación ascendente aplicada a las expresiones.

Listado 3. Ejemplo de propagación de atributos sintetizados en Cup

```
expression ::= expression :e1 PLUS expression :e2

{ :

      Expression e = new Expression () ;

      <<completar e en función de atributos de e1 y e2>>

      RESULT = e;

:} ;
```

Obsérvese que *expression* debe ser declarada de tipo *Expression* en la sección de declaración de no terminales. De esta forma es posible acceder a los atributos para e1 y e2 definidos dentro de dicha clase y también es posible construir un objeto de tipo *Expression* y asignárselo al antecedente de la regla mediante el uso de la cláusula RESULT.

En ocasiones las acciones semánticas necesarias para alguna regla de producción se limitan a hacer una propagación ascendente de alguno de los objetos no terminales que se encuentra en la parte derecha de la misma pero respetando la tipificación. Por ejemplo, en el listado 4 aparece una regla de producción que indica que una expresión puede ser una referencia (una referencia representa o bien un identificador, o bien el acceso al campo un registro o un elemento de un array). En este caso la acción semántica se limita a copiar el contenido de los atributos de la referencia a la expresión. Para ello podemos hacer uso de los constructores de copia que describíamos anteriormente. En este caso, la clase Expression incluye un constructor de copia de referencias (clase Reference).

Listado 4. Ejemplo de propagación de atributos mediante constructores de copia

```
expression ::= reference : r
 { :
      Expression e = new Expression (r);
      RESULT = e;
 :} ;
```

3.1.4 La clase Axiom

En el paquete compiler.syntax.nonTerminal puede observarse la existencia de la clase Axiom que hereda de NonTerminal. Esta clase es necesaria para que el programa de evaluación automática funcione adecuadamente.

Al respecto de esta clase deben hacerse las siguientes advertencias:

- El alumno no debe cambiar el código de esta clase. Sí es posible ampliar el código de la misma añadiendo nuevos métodos o atributos, pero en ningún caso modificar el código proporcionado. De todas formas el alumno no debería tener ninguna necesidad para hacer cambios sobre esta clase.

- El alumno solo debe garantizar que el recorrido de la regla de producción para Axiom se realiza de acuerdo a lo descrito anteriormente ya que como puede verse, este objeto es explotado desde la regla 'program' que inicia el análisis.

3.2 Análisis semántico

En este apartado se describirá el trabajo que ha de desarrollar el estudiante para realizar el análisis semántico. Esta fase se encargará principalmente de:

- Comprobación de la unicidad de declaraciones y definiciones. No permitir que existan dos declaraciones con el mismo nombre en el mismo ámbito.

- Comprobación de tipos. Augurarse que todas las variables y constantes simbólicas referenciadas en el contexto de una expresión han sido previamente declaradas y que el tipo de cada construcción coincide con el previsto en su contexto.

- Comprobación de concordancia en las referencias. Asegurarse de que el tipo de las expresiones que sirven para referenciar el rango de un conjunto de tipo entero y que el rango cae dentro del rango de valores esperado según su declaración (cuando esto sea posible). En los registros comprobar que el acceso a un campo de un registro coincide con alguno de aquellos indicados en la declaración.

- Comprobación de paso de parámetros. Asegurarse de que el orden y tipo de los parámetros actuales pasados en una invocación a un subprograma coincide con el orden y tipo de los parámetros formales indicados en la declaración. En caso de funciones comprobar asimismo que el tipo de retorno coincide con el tipo esperado según la declaración.

- Comprobación de la existencia de la sentencia de retorno en las funciones. Asegurar que cada función contiene una sentencia de retorno.

Todas estas comprobaciones han de realizarse mediante acciones semánticas dentro de las reglas del analizador sintáctico. Para ello han de utilizarse las siguientes clases proporcionadas por el marco de trabajo tecnológico.

3.2.1 Gestor de errores semánticos

La clase SemanticErrorManager es el gestor de errores semántico. Se sitúa en el paquete *es.uned.lsi.compiler.semantic* y es un artefacto que permite informar de los errores semánticos que se encuentran durante el proceso de compilación de un programa fuente. Es similar al uso de las clases LexicalErrorManager y SyntaxErrorManager. A continuación se resume el significado de cada uno de sus métodos:

- **SemanticDebug**. Sirve para emitir mensajes de depuración por la salida estándar asociados con los aspectos semánticos. A la hora de implementar las acciones semánticas se encontrará frecuentemente con errores en tiempo de ejecución (del estilo ClassCastException o NullPointerException) que son difíciles de localizar si no utiliza un mecanismo de traza del código apropiado. Este método viene a cubrir esta necesidad.

- **SemanticInfo**. Emite mensajes de información relacionados con los aspectos semánticos. Utilice este método para informar del avance del procesamiento semántico de las acciones si lo desea.

- **SemanticError**. Este método sirve para informar de errores semánticos (típicamente warnings) que pueden detectarse durante la compilación. La invocación a este método emite un mensaje de error pero no detiene el proceso de compilación.

- **SemanticFatalError**. Este método sirve para informar de errores semánticos irrecuperables que pueden detectarse durante la compilación. La invocación a este método emite un mensaje de error y detiene el proceso de compilación.

El marco de trabajo proporciona un mecanismo de impresión de mensajes basado en niveles similar al proporcionado por otros sistemas de traza (Log4J). En el nivel más bajo de filtro, DEBUG, todos los mensajes son mostrados. En el nivel INFO son emitidos los mensajes de información y los de error. Utilizando el nivel ERROR solo se muestran mensajes de error. Con el Nivel FATAL sólo se informa de los errores irrecuperables. El nivel más alto, NONE, no hay traza en absoluto. Esta es una característica añadida a los tres gestores de errores (léxico sintáctico y semántico). Para una descripción más detallada sobre estos artefactos consulte la API.

3.2.2 Ámbitos

Un programa fuente está compuesto por diversos ámbitos gramaticales. Cada vez que se cree un ámbito (Scope) éste creará, de forma automática, una tabla de tipos y de símbolos asociadas a ese ámbito. En la siguiente sección se explicarán ambas tablas.

Al acceder a un nuevo ámbito, por ejemplo, el comienzo de una función, procedimiento o programa principal, se ha de crear mediante el método ScopeManager.openScope() un nuevo ámbito de visibilidad. ScopeManager permite alcanzar el ámbito en curso (getCurrentScope) que a su vez da acceso a las tablas de símbolos y tipos asociadas al mismo. A continuación se explica cada clase incluida en el paquete es.uned.lsi.compiler.semantic (ver Figura 1).

- **Interfaz ScopeIF**. Dentro de un programa existen diferentes ámbitos de visibilidad de manera que dos variables (o constantes) pueden ser declaradas con un mismo nombre siempre que esto no ocurra en el mismo ámbito. Para dar soporte a esta idea, el marco de trabajo proporciona el artefacto Scope, comentado anteriormente.

- **Clase Scope**. La clase Scope es una implementación completa del interfaz anterior. Un ámbito alberga un nombre, cadena de texto que identifica de manera única al ámbito, una tabla de símbolos, donde se almacenarán todos los símbolos declarados dentro de dicho ámbito, una tabla de tipos y una tabla de temporales, de la cual hablaremos más profundamente en la siguiente sección. Por construcción, los ámbitos creados tendrán un nivel de anidamiento que se asigna automáticamente dentro de la arquitectura del marco de trabajo (el nivel del módulo principal es 0). Con el método Scope.getLevel () se devuelve el nivel de un determinado ámbito. Al crearse un Scope éste crea a su vez una tabla de símbolos (SymbolTable) que sigue el interfaz SymbolTableIF.

- **Interfaz ScopeManagerIF**. Para evitar la colisión de nombres entre símbolos declarados en distintos ámbitos se utiliza un gestor de ámbitos cuya interfaz describe esta clase.

- **Clase ScopeManager**. Es una implementación completa del interfaz anterior. Básicamente esta implementación consiste en una estructura de datos que gestiona los ámbitos apilándolos en una pila interna según se entra y se sale de los mismos haciendo uso de las funciones openScope (name) y closeScope (). El gestor de ámbitos contiene otros métodos que permiten acceder a otros ámbitos como getParentScope (), getCurrentScope () o getScope (int index). Sin embargo, tal vez los más importantes son searchSymbol () y containsSymbol () junto con searchType() y constainsType() que permiten hacer una búsqueda en profundidad sobre la pila de ámbitos para obtener el símbolo o tipo declarado en el ámbito más próximo al ámbito en curso. Consulte la API y la teoría para obtener detalles a este respecto. A este respecto, el estudiante ha de descubrir en qué partes del programa fuente se crean ámbitos.

3.2.3 Tipos y tabla de tipos

Durante el proceso de compilación de un programa fuente es necesario registrar en una estructura de datos, llamada tabla de tipos, todos los tipos definidos por el usuario en el programa y también los tipos primitivos prescritos por el lenguaje. Esto es necesario para comprobar la concordancia de tipos en las expresiones cuando se usan referencias a variables, constantes o funciones previamente declaradas. En este sentido, el marco de trabajo suministra una colección de artefactos que han de ser apropiadamente conjugados con acciones semánticas codificadas por el alumno.

Todas las clases que el alumno necesite desarrollar para implementar los tipos han de situarse dentro del paquete *compiler.semantic.type*. A continuación se describen cada uno de estos elementos (ver figura 1). Para más detalles consulta la API:

- **Interfaz TypeIF**. La interfaz TypeIF, definida en el paquete es.uned.lsi.compiler.semantic.type, es la encargada de definir la colección mínima de métodos que toda clase de tipos (simple, conjuntos, registro, procedimiento y función) debe implementar.

- **Clase Abstracta TypeBase**. La clase abstracta TypeBase, contenida dentro del paquete es.uned.lsi.compiler.semantic.type, representa una implementación de referencia de los métodos prescritos en la interfaz TypeIF. Esta clase es utilizada por las demás clases que representan tipos como clase base de la que extienden una funcionalidad básica.

- **Clase Abierta TypeSimple.** La clase abierta TypeSimple representa tipos primitivos del lenguaje (Integer y Boolean). Reside en el paquete compiler.semantic.type (no confundir con es.uned.lsi.compiler.semantic.type) y constituye una clase abierta en el sentido de que se proporciona su código fuente para que el alumno la altere definiendo los atributos que serán necesarios para caracterizar los tipos primitivos del lenguaje[2].

- **Clase Abierta TypeArray**. Utilizada para representar tipos estructurados vectoriales.

- **Clase Abierta TypeRecord**. Utilizada para representar tipos Registro.

- **Clase Abierta TypeEnum**. Utilizada para representar tipos enumerados.

- **Clase Abierta TypeProcedure**. Utilizada para representar procedimientos.

- **Clase Abierta TypeFunction**. Utilizada para representar funciones.

- **Clase Abierta TypePointer**. Utilizada para representar tipos puntero.

- **Clase Abierta TypeSet**. Utilizada para representar tipos conjunto.

- **Clase Abierta TypeUnion**. Utilizada para representar uniones y registros variantes.

[2] De aquí en adelante debe entenderse que las clases abiertas son todas aquellas cuyo código fuente se proporciona en alguno de los subpaquetes del paquete compiler para que el alumno las complete o modifique. Nótese, no obstante, que estas alteraciones deben respetar en todo momento la sintaxis y semántica de los métodos heredados e implementados.

- **Interfaz TableTypeIF**. El interfaz TypeTableIF es una entidad que reside en el paquete es.uned.lsi.compiler.semantic.type y que prescribe la colección de métodos que es necesario implementar cuando se quiere construir una tabla de tipos.

- **Clase TypeTable**. Esta clase es una implementación completa del interfaz anterior, situada en el paquete es.uned.lsi.compiler.semantic.type. Las tablas de tipos son estructuras de diccionario (tablas hash) que permiten localizar tipos en función de su nombre. Los métodos para añadir tipos llevan el nombre 'addType'. Se pueden realizar consultas para saber si un tipo ya está incluido mediante los métodos 'containsType'

En la figura 1 se muestra el diagrama de clases correspondiente con las clases necesarias para representar tipos que pueden introducirse en la tabla de tipos. Como se comentaba antes, TypeIF corresponde con la interfaz que implementa TypeBase y define las funciones básicas que han de tener los tipos: asignar/devolver un nombre (get/setName) y un ámbito (get/setScope). TypeBase, por su parte, es la clase de la que heredan los tipos que han de utilizarse en el compilador. Todos estos tipos se registran en TableType que a su vez queda descrita por el interfaz TableTypeIF.

Para el desarrollo de la práctica hay que completar la implementación de aquellos tipos que se necesiten según la especificación (A o B) que tenga el alumno. Se pueden añadir clases auxiliares en este paquete pero *en ningún caso* se han de modificar las dependencias existentes.

Además del resto de métodos y atributos que el alumno desee añadir a las clases para completar la definición del tipo, existen varios métodos que se proporcionan como por ejemplo:

- **public final boolean equals (Object other).** Este método implementa la lógica de comparación de un objeto pasado como argumento (other) con el objeto en curso (this). Implementa la equivalencia de tipos garantizando que ésta es nominal. Es decir, dos tipos serán equivalentes si y solo si, corresponden a la misma entrada de la tabla de tipos.

3.2.4 Símbolos y tabla de símbolos

La gestión de símbolos es un proceso mediante el cual el compilador debe registrar en una estructura de datos llamada tabla de símbolos todos los símbolos del programa fuente que hayan sido declarados por el programador. Estos símbolos incluyen constantes simbólicas, variables, procedimientos y funciones. Esta información será utilizada para garantizar posteriormente que cada referencia a un símbolo dentro de una expresión ha sido previamente declarada.

Cuando se declara un símbolo, éste viene acompañado de información de tipo. Esto quiere decir que deberemos asegurar que cada entrada de la tabla de símbolos se enlaza, antes o después, con una entrada de la tabla de tipos correspondiente. Esto se puede hacer mediante el método setType que todas las clases de símbolos heredan de la clase SymbolBase (ver el API para más información)

El trabajo del alumno es incluir las acciones semánticas pertinentes para garantizar que el registro de símbolos se realiza apropiadamente. En este sentido, el marco de trabajo proporciona una serie de clases de ayuda (Figura 1). En caso de desarrollar nuevas clases, éstas han de situarse en el paquete *compiler.semantic.symbol* donde se encuentran el resto de las proporcionadas por el marco de trabajo:

- **Interfaz SymbolTableIF**. Definido dentro del paquete es.uned.lsi.compiler.semantic.symbol prescribe la colección de métodos que es necesario implementar para obtener una tabla de símbolos.

- **Clase SymbolTable**. Esta clase implementa la interfaz SymbolTableIF y es ofrecida en el mismo paquete. En ella existen, entre otros, métodos para añadir 'addSymbol', y buscar por nombre 'containsSymbol' (el lexema del identificador asociado al símbolo) símbolos dentro de la tabla.

- **Interfaz SymbolIF**. Esta interfaz, incluida dentro del paquete es.uned.lsi.compiler.semantic.symbol, describe la colección de métodos que es necesario implementar para construir un tipo de símbolo.

- **Clase Abstracta SymbolBase**. Esta clase base, contenida dentro del paquete es.uned.lsi.compiler.semantic.symbol, es una implementación abstracta de los métodos definidos en la interfaz anterior. Todas las clases abiertas siguientes extienden de esta clase de forma directa o indirecta.

- **Clase Abierta SymbolVariable**. Sirve para representar símbolos de tipo variable local, global o parámetros.

- **Clase Abierta SymbolConstant**. Representa símbolos de tipo constante.

- **Clase Abierta SymbolProcedure**. Representa símbolos de tipo procedimiento.

- **Clase Abierta SymbolFunction**. Representa símbolos de tipo función.

- **Clase Abierta SymbolParameter**. Representa símbolos de tipo parámetro de función o procedimiento.

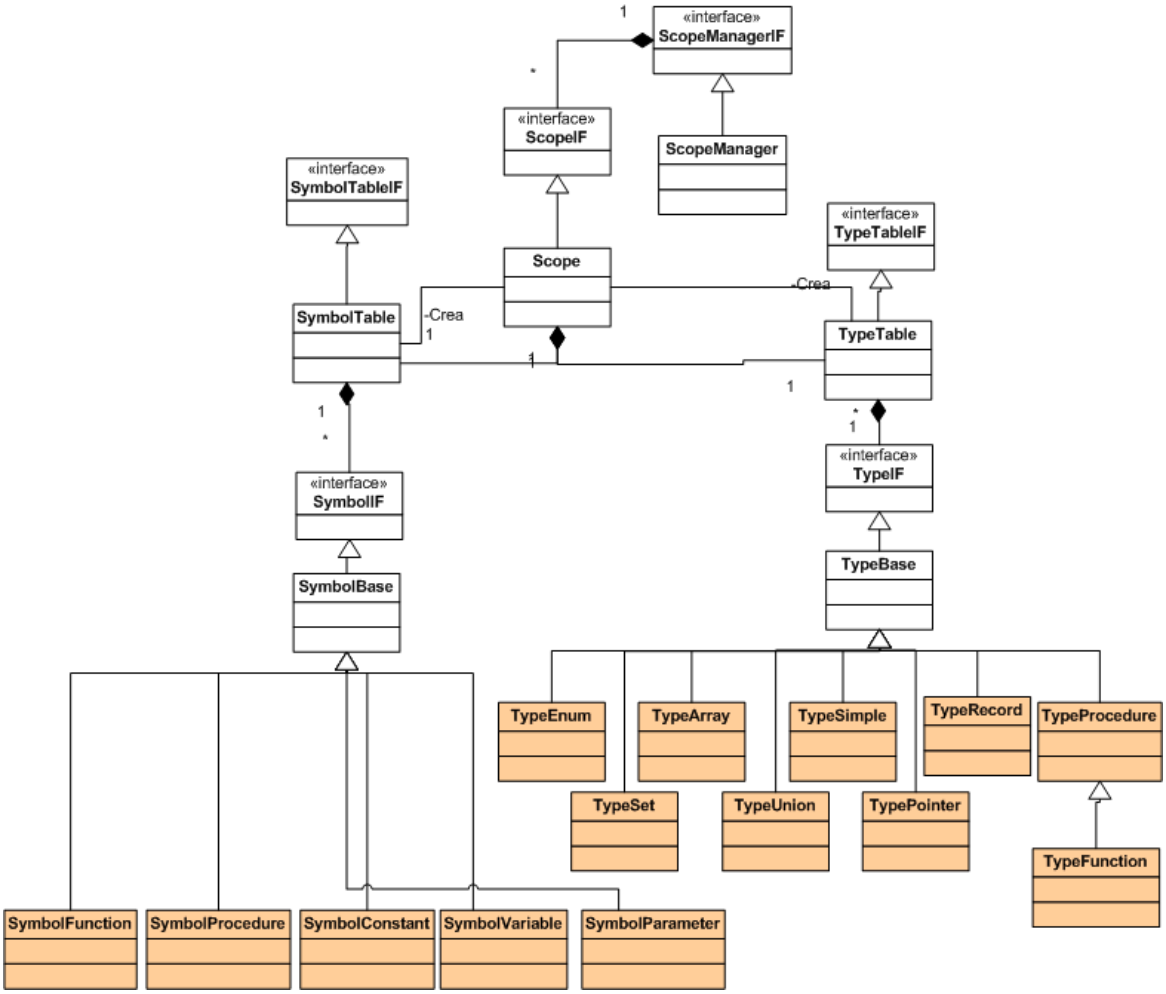


Figura 1. Diagrama ámbitos, tabla de símbolos y tabla de tipos

Tanto en la figura 1 como en la figura 2 las clases con relleno naranja (gris claro al imprimir) representan las clases abiertas, cuya implementación debe ser completada por el alumno.

3.3 Código intermedio

Una vez implementados todos los aspectos semánticos del compilador, el siguiente paso es incluir acciones semánticas en la especificación de Cup para que el proceso de análisis sintáctico vaya generando una secuencia de instrucciones de código intermedio. Estas instrucciones se van acumulando en los atributos a tal efecto de los no terminales de la gramática mediante sucesivos procesos de reordenación y reconstrucción hasta que al final se alcance en el axioma una representación en código intermedio de todo el árbol de análisis sintáctico.

En este sentido, lo que el alumno debe garantizar de cara a que el proceso de evaluación automática funcione correctamente es que dicho código pueda obtenerse mediante el método getIntermediateCode() de la clase no terminal Axiom. Este método está definido en el interfaz NonTerminalIF del que hereda Axiom, ya que implementa NonTerminal. Es decir, el alumno deberá propagar un objeto de tipo Axiom en la regla Axiom garantizando que el atributo intermediateCode contiene todo el código intermedio.

A continuación describimos los artefactos que son necesarios para general el código intermedio (figura 2). En general, las interfaces están situadas en el paquete *es.uned.lsi.compiler.intermediate* y las implementaciones que han de completar los estudiantes en el paquete *compiler.intermediate*. Para una descripción más detallada consulte la API:

- **Interfaz QuadrupleIF**. En esta práctica el lenguaje de código intermedio está formado por una colección ordenada de cuartetos. Un cuarteto o cuádrupla consta de un código de operación de tipo String y tres operandos (dos operandos y un resultado), alguno de los cuales puede no tener valor para representar el caso de los cuartetos con menos operandos. Esta interfaz, declarada dentro del paquete es.uned.lsi.compiler.intermediate, indica la colección de métodos que deben implementar las clases que representen cuartetos, básicamente son métodos de acceso y modificación (get/set) de los atributos anteriores.

- **Clase Quadruple**. Es una implementación completa del interfaz anterior que se encuentra definida dentro del paquete es.uned.lsi.compiler.intermediate. Esta clase no debe ser implementada ni completada por los estudiantes.

- **Interfaz OperandIF**. Cada uno de los tres operandos que forman parte de un cuarteto son de tipo OperandIF. Este interfaz, definido dentro del paquete es.uned.lsi.compiler.intermediate, no contiene ningún método, es un interfaz de marca que sirve para representar en abstracto los diferentes tipos de operandos que un cuarteto puede contener.

- **Interfaz ValueIF**. Representa operandos numéricos dentro de los cuartetos.

- **Clase Abierta Value**. Esta clase es una implementación abierta del interfaz ValueIF. Por ejemplo, en el cuarteto ADD t0 a 1 el tercer operando (1) debería ser un objeto de tipo Value. El alumno puede acceder al código de esta clase para extender su definición.

- **Interfaz VariableIF**. Representa operandos para representar variables (locales y globales) y parámetros dentro de los cuartetos.

- **Clase Abierta Variable**. Esta clase es una implementación abierta del interfaz VariableIF. Es decir, se utiliza para introducir referencias simbólicas en código intermedio a las variables declaradas en un programa fuente. Por ejemplo, en el cuarteto ADD t0 a 1 el segundo operando (a) debería ser un objeto de tipo Variable, suponiendo que a puede asociarse con un SymbolVariable contenido dentro de alguna tabla de símbolos. El alumno puede acceder al código de esta clase para extender su definición.

- **Interfaz LabelIF**. Representa operandos de tipo etiqueta dentro de los cuartetos para dar soporte a las instrucciones de salto condicional e incondicional e implementar así las estructuras de control del lenguaje o las llamadas a subrutinas entre otras características.

- **Clase Abierta Label**. Esta clase es una implementación abierta del interfaz LabelIF. Se utiliza principalmente en las instrucciones de salto y en las invocaciones a procedimiento o función. Por ejemplo, en el cuarteto BR L1 t0 el primer operando (L1) debería ser un objeto de tipo Label que anteriormente o posteriormente a esta instrucción habrá sido insertado en el código para apuntar a cierta instrucción. El alumno puede acceder al código de esta clase para extender su definición.

- **Interfaz ProcedureIF**. Define procedimientos y funciones dentro del código intermedio.

- **Clase Abierta Procedure**. Es una implementación del interfaz ProcedureIF.

- **Interfaz LabelFactoryIF**. Este interfaz definido dentro del paquete es.uned.lsi.compiler.intermediate describe la colección de métodos que aparecen en las implementaciones de las factorías de etiquetas.

- **Clase LabelFactory**. Esta clase, contenida dentro del paquete es.uned.lsi.compiler.intermediate, es una implementación del interfaz anterior que devuelve un nuevo objeto Label cada vez que se invoca al método create (). Este artefacto garantiza la unicidad de nombres de etiquetas cada vez que se invoca al método constructor

- **Interfaz TemporalIF**. Representa operandos de tipo temporal dentro de los cuartetos.

- **Clase Abierta Temporal**. Esta clase es una implementación abierta del interfaz TemporalIF. Un temporal es la representación simbólica de una dirección de memoria donde se almacenan cómputos de carácter intermedio para realizar un procesamiento complejo. Por ejemplo, en el cuarteto ADD t0 a 1 el primer operando (t0) debería ser un objeto de tipo Temporal, suponiendo que t0 hace referencia a una posición de memoria intermedia donde se acumulará el resultado de la suma de a + 1 para luego recogerlo más adelante. El alumno puede acceder al código de esta clase para extender su definición

- **Interfaz TemporalFactoryIF**. Este interfaz definido dentro del paquete es.uned.lsi.compiler.intermediate, describe la colección de métodos que aparecen en las implementaciones de las factorías de temporales.

- **Clase TemporalFactory**. Esta clase, contenida dentro del paquete es.uned.lsi.compiler.intermediate, es una implementación del interfaz anterior que devuelve un nuevo objeto Temporal cada vez que se invoca al método create (). Este artefacto garantiza la unicidad de nombres de temporales cada vez que se invoca al método constructor. Además, al crearse el temporal lo añade a la tabla de temporales asociada al Scope actual. No debe ser implementada por el estudiante.

- **Interfaz TemporalTableIF.** Interfaz que define una tabla de temporales.

- **Clase TemporalTable**. Implementa el interfaz TemporalTableIF. Al crear un objeto Scope, además de crearse una tabla de símbolos y tipos, se crea también una tabla de temporales (TemporalTable) que sirve para almacenar todos los temporales definidos en un determinado ámbito. De esto se encargan las fábricas de temporales. Al crear un temporal para el código intermedio, éste se almacenará también en su tabla de temporales asociada al ámbito en el cuál se creó (ver TemporalFactory).

- **Interfaz IntermediateCodeBuilderIF**. Este interfaz, definido dentro del paquete es.uned.lsi.compiler.intermediate, describe la colección de métodos del artefacto IntermediateCodeBuilder que describimos a continuación.

- **Clase IntermediateCodeBuilder**. Esta es una clase de ayuda que permite construir nuevos cuartetos y acumularlos internamente para conformar un fragmento de código intermedio. Para construir (añadir) nuevos cuartetos a dicho código esta clase dispone de diferentes métodos addQuadruple y addQuadruples (véase documentación de la clase en la API para más detalles al respecto). Asimismo, el método create () sirve para devolver la lista de cuartetos construida. El listado 5 es un ejemplo que ilustra el uso de esta clase para generar de forma cómoda el código asociado a una expresión de suma.

Listado 5. Ejemplo de generación de código intermedio y final

```
Expression ::= expression:e1 ADD expression:e2 {:
   Expression e = new Expression ();
   // Comprobación de tipos
   TypeIF e1Type = e1.getType ();
   TypeIF e2Type = e2.getType ();
   if (e1Type.equals (e2Type)) e.setType (e1Type);
   else semanticErrorManager ("Error de tipos");
   // Generación de código intermedio
   ScopeIF scope = scopeManager.getCurrentScope ();
   TemporalFactoryIF tf = new TemporalFactory ();
   TemporalIF eTemp = tf.create ();
   e.setTemporal (eTemp);
   TemporalIF e1Temp = e1.getTemporal ();
   TemporalIF e2Temp = e2.getTemporal ();
   IntermediateCodeBuilderIF cb = new IntermediateCodeBuilder (scope)
   cb.addQuadruples (e1.getIntemediateCode ());
   cb.addQuadruples (e2.getIntemediateCode ());
   cb.addQuadruple ("ADD", eTemp, e1Temp, e2Temp);
   List intermediateCode = cb.create ();
   e.setIntermediateCode (intermediateCode);
   ...
   RESULT = e; :};
```

Como puede apreciarse en este ejemplo, al seguir la traducción dirigida por la sintaxis, cuando se crea nuevo código, éste debe añadirse al código intermedio creado por otras reglas de producción.

Esto se puede hacer mediante el método addQuadruples() e invocando a getIntemediateCode () que poseen todas las clases NonTerminal para devolver el código intermedio parcial generado.

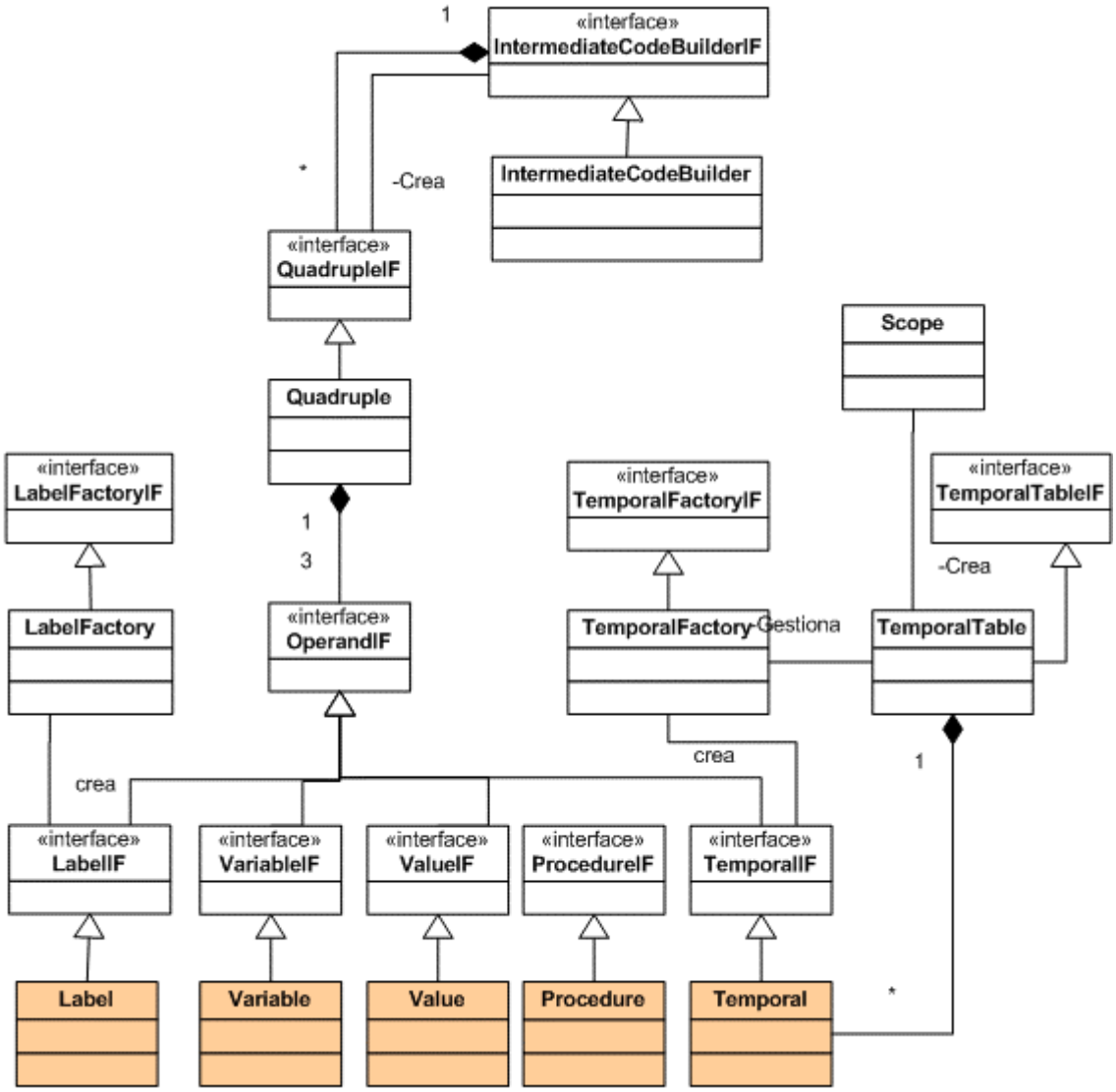


Figura 2 Diagrama código intermedio

El código intermedio ha de diseñarse de forma que su traducción a código final sea lo más directa posible. Es decir, la proximidad entre código intermedio y código final debe maximizarse de forma que pueda afirmarse que el código intermedio es casi una representación de código final donde aún no se han resulto las referencias simbólicas a memoria (variables y temporales) por direcciones físicas de memoria.

Obsérvese además que la creación de los registros de activación en las secuencias de llamada y retorno deberían prepararse adecuadamente en esta parte para permitir la existencia de funciones con recursividad directa y no delegarse a la fase siguiente de traducción a código final ya que este último paso debe ser lo más automático posible.

3.4 Código final

La última fase del compilador es la construcción del código final. Este código es una lista de sentencias en el lenguaje ensamblador de una arquitectura final generada a partir de los cuartetos obtenidos durante la fase anterior. En el caso de esta práctica el lenguaje ensamblador utilizado viene proporcionado por el emulador ENS2001. El manual de usuario de este emulador está disponible dentro del curso virtual. Las clases encargadas de generar el código final, que el estudiante ha de completar, estarán en el paquete *compiler.code***.** A continuación se describen las clases ofrecidas para desarrollar la práctica.

- **Interfaz FinalCodeFactoryIF**. Esta interfaz, definida dentro del paquete es.uned.lsi.compiler.code, describe la colección de métodos que implementa la clase FinalCodeFactory.

- **Clase FinalCodeFactory**. Esta clase implementa el interfaz FinalCodeFactoryIF y es la encargada de generar el fichero de texto que contendrá todas las instrucciones ensamblador. Está situada en el paquete es.uned.lsi.compiler.code. El fichero de salida puede proporcionarse como parámetro y cambiarse en cualquier momento. En el test proporcionado este fichero será nombrado igual que el fichero usado como código fuente pero cambiando su extensión a '.ens'. Por ejemplo, 'test.p' generará un fichero llamado 'testA.ens'. Para crear el fichero se invoca al método create (List intermediateCode) pasándole una lista de cuartetos (objetos tipo Quadruple).

- **Interfaz MemoryDescriptorIF.** Interfaz definida en el paquete es.uned.lsi.compiler.code. Proporciona los métodos necesarios para realizar la gestión de memoria durante la generación del código final.

- **Clase Abierta Memory Descriptor.** Clase que implementa el interfaz MemoryDescriptorIF.

- **Interfaz RegisterDescriptorIF.** Interfaz definida en el paquete es.uned.lsi.compiler.code. Proporciona los métodos necesarios para crear descriptores de registro durante la generación del código final.

- **Clase Abierta RegisterDescriptor**. Clase que implementa el interfaz RegisterDescriptorIF.

- **Interfaz ExecutionEnvironmentIF**. Esta interfaz representa los métodos que debe implementar una clase que represente un entorno de ejecución. Está situada en el paquete es.uned.lsi.compiler.code.

- **Clase Abierta ExecutionEnvironmentEns2001**. La traducción de los cuartetos a código ensamblador se realiza dentro la clase ExecutionEnvironmentEns2001 que implementa la interfaz ExecutionEnvironmentIF para la herramienta ENS2001. El estudiante ha de implementar el método translate (QuadrupleIF quadruple) que sirve para traducir cada cuarteto de código intermedio a un String java que representa la instrucción equivalente en el lenguaje ensamblador del entorno seleccionado (en este caso ENS2001). Este proceso de traducción es abierto. Es decir, el alumno puede utilizar nuevas clases o métodos para articular el mismo siempre y cuando no modifique ni la sintaxis ni la semántica de dicho método. Obsérvese además que un cuarteto de código intermedio puede dar lugar a una

secuencia ordenada de varias instrucciones en ensamblador. En este caso, el estudiante deberá asegurarse de que, dentro del String de retorno, cada instrucción esté separada por un retorno de carro '\n'

Si no hay errores léxicos, sintácticos ni semánticos en el código fuente pasado al compilador a desarrollar, se generará un fichero con el código ensamblador. Entonces ha de ejecutarse con el emulador ENS2001 para comprobar que su comportamiento es el esperado.

3.5 Cambios en el fichero CUP

Para que Cup soporte la arquitectura definida por el marco de trabajo de la práctica, es necesario realizar una serie de modificaciones sobre el fichero de la especificación gramatical de partida (parser.cup). A continuación se detallan los principales puntos a tener en cuenta:

- **Sección action code**. No se debe comentar ninguna de las líneas que se suministran en el fichero de partida. Deben extraerse de la clase CompilerContext todas las variables que referencian artefactos que son necesarios para realizar el compilador. Esta clase es instanciada en FinalTestCase y se trata de una clase con métodos estáticos para obtener en cualquier parte del código (incluidas las posibles clases creadas por los estudiantes) clases de utilidad. Por ejemplo, en la primera línea del action code se observa que desde la variable syntaxErrorManager se puede acceder a la clase SyntaxErrorManager. Al obtenerla desde el objeto CompilerContext siempre se devolverá la misma, por tanto no ha de crearse nuevos objetos manager. En concreto se pueden recoger del contexto los siguientes objetos únicos: SyntaxErrorManager, SemanticErrorManager, ScopeManager, y finalCodeFactory.

- **Producción inicial**. En la acción semántica que sigue al no terminal Axiom debe incluirse la secuencia de instrucciones indicada para que genere el código intermedio una vez finalizado el proceso de compilación. Recuerde que para que este proceso produzca resultados el método ax.getIntermediateCode () debe dar una lista de objetos QuadrupleIF no vacio. En efecto, dentro de las acciones semánticas de program se recoge el código intermedio almacenado en ax (sintetizado de Axiom) y se procede a crear el código final con la llamada finalCodeFactory.create (intermediateCode). Llegados a ese punto se generará el código final y el parser terminará.

- **Resto de producciones**. Se deben incorporar a las reglas de la gramática las acciones semánticas pertinentes para implementar los aspectos semánticos y de generación de código intermedio (ver listado 5).

3.6 Juegos de prueba

Para terminar la descripción del marco de trabajo de referencia vamos a exponer el contenido de la clase FinalTestCase. La tarea Ant para ejecutar la clase FinalTestCase se llama finalTest y recibirá un parámetro indicando el nombre del fichero de pruebas a compilar. Este fichero ha de situarse en el directorio 'doc/test'.