Universidad Central “Marta Abreu” de Las Villas.
Facultad Matemática, Física y Computación
Licenciatura en Ciencia de la Computación

TRABAJO DE DIPLOMA

Titulo: Aplicación de Reglas de Negocio y Base de Datos para el Trasplante Renal.

Autor: Alexander Garcell Roque.
Tutores: Msc. Martha Beatriz Boggiano Castillo
Lic. Alain Pérez.
Curso 2009-2010

Dictamen

Dictamen.
Hago constar que el presente trabajo fue realizado en la Universidad Central “Marta Abreu” de Las Villas como parte de la culminación de los estudios de la especialidad de Ciencia de la Computación, autorizando a que el mismo sea utilizado por la institución, para los fines que estime conveniente, tanto de forma parcial como total y que además no podrá ser presentado en eventos ni publicado sin la autorización de la Universidad.

Firma del autor

Los abajo firmantes, certificamos que el presente trabajo ha sido realizado según acuerdos de la dirección de nuestro centro y el mismo cumple con los requisitos que debe tener un trabajo de esta envergadura referido a la temática señalada.

Firma del tutor

Firma del jefe del Laboratorio

Dedicatoria.

A mi madre y abuela,
A mis tíos,
A mi hermano y
A mi novia Vivian.

Agradecimientos.

A Omar quien estuvo todo el tiempo a mi lado,
A mi tutora MSc. Martha Beatriz Boggiano Castillo y
A mí también tutor Lic. Alain Pérez Alonso, quien me asesoró en momentos críticos,
A todos los profesores que me formaron,
A Javier, Antonio, El Loko, y a todos mi amigos y amigas que no me alcanzaría mil páginas para mencionarlos.

Pensamiento

Los sueños han de ser grandes, para no perderlos de vista.

Resumen.
Los problemás en la concepción de Sistemás de Información, debido al tratamiento informal de los requerimientos, han permitido considerar como factor de esencial importancia la forma en que se manejan los requisitos, enfocados actualmente como Reglas de Negocio.

Por tanto el propósito de este trabajo es automatizar las reglas de restricción para la problemática de trasplante renal, ganando con ello agilidad, transparencia y reducción del costo en el negocio. Esto implica estudiar un patrón para reglas de restricción, analizar los recursos que posibiliten su implementación y determinar su forma de almacenamiento.

Se hace una revisión crítica de la base de datos, para cada una de las reglas editadas en la aplicación. Se modifica el compilador de la aplicación, implementándole el tratamiento de errores al analizador sintáctico, además de implementar el análisis semántico en este. Se concluye con la validación de la Aplicación para Reglas de Negocio de Restricción. Esta validación es mediante la generación de las reglas de restricción para la base de datos del negocio de trasplante renal. Implementando una serie de transformaciones al editor, y al compilador, que mejoran la utilización de la aplicación para el usuario común.

Abstract.

The problems in the conception of Systems of Information, due to the informal treatment of the requirements, they have allowed to consider as factor of essential importance the form in that the requirements are managed, focused at the moment like business rules. Therefore the purpose of this work is to automate the restriction rules for the problem of renal transplant, winning with it agility, transparency and reduction of the cost in the business. This implies to study a pattern for rules offered, to analyze the resources that facilitate their implementation and to determine their storage form.

A critical revision of the database is made, for each one of the rules published in the application. The compiler of the application modifies, implementing him the treatment of errors to the syntactic analyzer, besides implementing the semantic analysis in this.

You concludes with the validation of the Application for Rules of Business of Restriction developed for and modified . This validation is by means of the generation of the restriction rules for the database of the business of renal transplant. Implementing a series of transformations to the editor, and to the compiler that you/they improve the use of the application for the common user.

Tabla de Contenido.

Tabla de Contenido

Introducción.

La insuficiencia renal crónica es cada vez más común y de complejo seguimiento. Es conceptualizada como una enfermedad epidémica y catastrófica. El manejo de toda la información necesaria para los procesos de consulta de enfermedad renal crónica avanzada, hemodiálisis ambulatoria y trasplante renal en el Hospital Provincial "Arnaldo Milián Castro" es realmente engorroso. Se requiere contar con el diseño de estos procesos, de su base de datos para conformar un Sistema de Información que permita el manejo de los datos de forma precisa en el área de nefrología, un sistema regido por el comportamiento de las políticas de trabajo de este servicio médico, en forma de Reglas de Negocio, con el objetivo de que se garantice que nuevas políticas en la organización o el cambio de estas no afecte drásticamente el Sistema de Información y el trabajo de los analistas y programadores.

La base de datos para el control de las actividades asociadas al trasplante renal fue diseñada en (González Mena 2008) a partir de un conjunto de consideraciones básicas brindadas por los clientes a los analistas sobre qué datos e información debía persistir en la base de datos, es decir qué datos debían ser almacenados sobre los pacientes, trasplantes, donantes, etc.

La mayoría de los requerimientos del sistema y procesos de negocios involucrados en el comportamiento del negocio fueron expresadas en forma de regla y se implementaron siguiendo la filosofía de diseño usando tres capas: capa de datos, capa del negocio y capa de interfaz; en la capa del negocio se implementaron la mayoría de los procesos y reglas de este negocio: trasplante renal.

Reglas de Negocio o *Business rules*, que es el término anglosajón, se oye por vez primera en la voz de Daniel Appleton, por medio de su artículo *The Missing Link*. Desde 1984 ya hace más de dos décadas, se ha escrito bastante y desde enfoques diversos sobre este "enlace perdido". En él se reclamaba la formalización de los términos del negocio (Toledo, 2009).

Pronto, los científicos de la computación se dieron cuenta de la utilidad que sostenía aplicar recursos de bases de datos para recoger las reglas, tales como la modelación

conceptual y recursos de restricción en las bases de datos relacionales. Solo que tales recursos para captar las reglas estaban diluidos en la lógica de implementación. Así, se ideó la utilización de triggers, suerte de artificio del lenguaje procedural del SQL, para implementar la aplicación de reglas del negocio más complejas y cambiantes (Morgan, 2002).

Las Reglas de Negocio son un componente clave en cómo se toman las decisiones en una empresa o institución, estas se encuentran siempre presentes en la actuación de una organización (Lorenzo, 2009).

En el grupo de investigación de Base de Datos del CEI, se estudia la problemática de trabajar las Reglas de Negocio más cercanas a los datos como recursos de los gestores de bases de datos implementados automáticamente, a partir de *The Missing Link*, y se han desarrollado dos versiones de la herramienta de software para la generación automática de reglas de restricción en base de datos (Aplicación de Reglas de Negocio v1.0 ,v1.1).

Para estas aplicaciones la base de datos de prueba ha sido producto de un diseño sencillo con el tema de trasplante renal, pero que no tiene en cuenta la gran mayoría de las políticas reales asociadas a los datos que rigen el trabajo en el servicio, solo considera un conjunto incipiente y variado de reglas de negocio, clasificadas como de tipo restricción, para probar que puede lograrse una inserción automática de reglas de negocio expresadas con un "patrón" determinado donde se reconozcan en la formulación de la regla tablas, atributos y relaciones entre tablas de la base de datos del negocio(Pérez Alonso, 2008).

Planteamiento del problema.

- Se necesita conocer el comportamiento de la herramienta Aplicación de Reglas de Negocio (v 1.1) para generar reglas de restricción en una base de datos, que expresen realmente las características de los datos y sus relaciones en un entorno real.

Objetivo general.

- Validar el software Aplicación de Reglas de Negocio v1.1 para la generación de las reglas de negocio de restricción en la base de datos de un Sistema de

Información para el departamento de trasplante renal del Hospital Provincial de Villa Clara, para poder hacer un estudio de las características que exige esta herramienta para la base de datos. Definir y eliminar las limitaciones que puedan detectarse en esta herramienta.

Objetivos específicos.

- Transformar la base de datos de trasplante renal al gestor Microsoft SQL Server 2000 desde el MySQL, para realizar una revisión de la implementación física de la base de datos, de modo que se garantice la correspondencia necesaria del diseño físico con los requerimientos de los datos persistentes del Sistema de Información del control de trasplante renal.
- Expresar las reglas de negocio en lenguaje técnico y usarlas como entrada al software Aplicación de Reglas de Negocio v1.1. para detectar errores en la concepción de la base de datos y en el funcionamiento del software.
- Realizar transformaciones necesarias a la Aplicación de Reglas de Negocio v1.1 que garanticen un mejor desempeño y la generación adecuada de las reglas.

Preguntas de Investigación.

1- ¿Cuáles reglas de negocio de tipo restricción están asociadas al Trasplante Renal?

2- ¿Cuáles reglas de negocio de tipo restricción pueden ser generadas automáticamente en la base de datos de Trasplante Renal?

3- ¿Cuáles son las dificultades que presenta el software para realizar una adecuada generación automática?

4- ¿Es posible garantizar la correspondencia necesaria entre el diseño físico de la base de datos y los requerimientos de los datos persistentes del Sistema de Información del Control de Trasplante Renal, después de transformar la base de datos MySQL a SQL Server 2000?

Capitulo1 ***"Estudio sobre la generación de reglas de negocio de restricción para un Sistema de Información basado en reglas"***

Capítulo1 "Estudio sobre la generación de Reglas de Negocio de Restricción para un Sistema de Información basado en Reglas"

Este capítulo recoge aspectos generales sobre las reglas de negocios, así como lo necesario a considerar para su automatización en el negocio de trasplante renal Se exploran los diferentes tipos de reglas y sus disímiles formas de implementación, especialmente, profundizando sobre los recursos de Base de Datos.

Se muestra cómo se insertan las Reglas de Negocio en un sistema y las diferentes vías para ello, siempre conscientes de las ventajas potenciales de su utilización. Mostrando una panorámica de la problemática de trasplante renal.

1.1 Problemática del trasplante renal.

La práctica de la medicina de trasplante ha sido adoptada, como primera opción terapéutica, para un número creciente de enfermedades orgánicas durante los últimos 50 años. El trasplante renal es un procedimiento quirúrgico para implantar un riñón sano en un paciente con insuficiencia renal crónica, consiste en extraer un riñón de una persona (donante) y colocarlo en un paciente que presente insuficiencia renal crónica en fase terminal. Puede ser trasplantado un paciente con diagnóstico confirmado de insuficiencia renal crónica avanzada o que ya se encuentre en un programa de hemodiálisis o en diálisis peritoneal.

1.2 Reglas de Negocio en los Sistemas de Información.

Todo Sistema de Información trata de reflejar parte del funcionamiento del mundo real, para automatizar tareas que de otro modo serían llevadas a cabo de modo ineficiente, o bien no podrían realizarse. Para ello, es necesario que cada Sistema de Información refleje las restricciones que existen en el negocio dado, de modo que nunca sea posible llevar a cabo acciones no válidas. A las reglas que debe seguir el Sistema de Información para garantizar esto se las llama *Reglas de Negocio*, o *Business Rules*. Ejemplos de tales

Reglas son: no permitir crear facturas pertenecientes a clientes inexistentes, controlar que el saldo negativo de un cliente nunca sobrepase cierta cantidad, etc.

En realidad, la información puede ser manipulada por muchos programas distintos: así, una empresa puede tener un departamento de contabilidad que controle todo lo relacionado con compras, cobros, etc., y otro departamento técnico, que esté interesado en relacionar diversos parámetros de producción con los costes. La visión que ambos departamentos tendrán de la información y sus necesidades será distinta, pero en cualquier caso siempre se deberán respetar las reglas de negocio. El hecho de que la información sea manipulada por diversos programas hace más difícil garantizar que todos respeten las reglas, especialmente si las aplicaciones corren en diversas máquinas, bajo distintos sistemás operativos, y están desarrolladas con distintos lenguajes y herramientas.

La adquisición de las reglas del negocio no es tarea fácil ya que muchas de las reglas son difíciles de identificar. En particular muchas reglas tienen una representación no explícita.

Dependiendo del contenido de su información las reglas del negocio pueden ser basadas sobre el conocimiento explícito o táctico, es decir ambas formas de las reglas aunque representan restricciones difieren entre ellas por la forma en que lo hacen. El conocimiento explícito es formalizado, conocimiento que es fácil de expresar en forma de principios, procedimientos, hechos, figuras, reglas, fórmulas, etc. Contrario a esto, el conocimiento táctico no es fácil de expresar y ver.

En el desarrollo del Sistema de Información la transformación de las reglas siempre aparecen en documentos, procedimientos, políticas, regulaciones, manual de usuarios, etc. Esto, sin embargo, no es siempre el caso. Muchas de las reglas que son usadas en regulaciones del negocio no tienen una representación formal. Ellas se basan en conocimiento táctico de forma individual y difícil de expresar. Son altamente personales y subjetivas, basadas en la experiencia, ideas, emociones e intuiciones.

Mientras una serie de trabajos tiene que realizarse para soportar la adquisición de las reglas del negocio (herramientas de soporte, maquetas, etc.) sólo se presta atención a la captura de

los enlaces relacionales entre las reglas. Esto es muy visible en la práctica donde las reglas son fundamentalmente capturadas sin tener en cuenta ninguna fuente que la motive. Esto hace que las Reglas de Negocio sean de muy difícil tratamiento.

Lo novedoso en esta área, es identificar posibilidades de capturar los enlaces relacionales de las Reglas de Negocio. El conocimiento y la experiencia desde el campo de administración del conocimiento deben ser muy valorados.

1.3 Clasificación de Reglas de Negocio.

Existen muchos tipos de Reglas de Negocio. Debido a su diversidad y complejidad los autores tienden a agruparlas y clasificarlas siguiendo diferentes puntos de vista, pero con un objetivo común; por tanto un estudio de estas clasificaciones muestra que en realidad son similares y se complementan unas a otras en especial la clasificación que se utilizará es la brindada por (Morgan, 2002) que es una buena y detallada clasificación.

El primer grupo de Reglas de Negocio engloba todas aquellas reglas que se encargan de controlar que la información básica almacenada para cada atributo o propiedad de una entidad u objeto es válida: no hay precios de artículos negativos, el sexo de una persona solo puede ser másculino o femenino, una fecha siempre debe ser una fecha válida (no existe el 30 de Febrero, ¿cierto?), etc. Estas reglas se las llama *reglas del modelo de datos*.

Otro grupo importante de reglas incluye todas aquellas reglas que controlan las relaciones entre los datos. Estas reglas especifican, por ejemplo, que todo pedido debe ser realizado por un cliente, y que el mismo debe estar dado de alta en nuestro sistema: además, una vez que un cliente haya hecho algún pedido, se deberá garantizar que no es posible eliminarlo, a menos que previamente se eliminen todos sus pedidos. Estas reglas constituyen las *reglas de relación*.

Es frecuente que a partir de cierta información se pueda derivar otra: por ejemplo, el total de un pedido se puede calcular a partir de las distintas líneas que lo componen, mientras que el total de cada línea se puede calcular a partir del número de unidades vendidas y el precio por unidad. Al conjunto de reglas que especifican y controlan la obtención de

información que se puede calcular a partir de la ya existente se las llama *reglas de derivación*.

Otro grupo de Reglas de Negocio es el compuesto por las *reglas de restricción*, que restringen los datos que el sistema puede contener. Nótese que este grupo de reglas se solapan en cierto modo con las reglas del modelo de datos, dado que aquellas también impiden la introducción de datos erróneos, como se vio anteriormente. La diferencia estriba en que las reglas de restricción restringen el valor de los atributos o propiedades de una entidad más allá de las restricciones básicas que sobre las mismás existen: por ejemplo, para un saldo existe una regla básica (regla del modelo de datos) que indica que éste debe ser un número (¡no por obvia es menos regla!), pero además puede haber una regla que indique que el saldo nunca puede ser menor que cierta cantidad tope establecida para cierto tipo de clientes. Esta sería lo que aquí denominamos una regla de restricción, y la diferencia fundamental estriba en el hecho de que este tipo de reglas requiere para su verificación del acceso a otros fragmentos de información, algo que no sucede con las reglas del modelo de datos. Esto tiene ciertas consecuencias que se verán más adelante.

El último grupo de Reglas de Negocio incluye aquellas reglas que determinan y limitan cómo fluye la información a través de un sistema. Por ejemplo, un cliente puede hacer una petición de análisis a un laboratorio, que anota un encargado: hecho esto, se genera un parte para uno o más analistas, estos realizan las mediciones correspondientes y devuelven los partes con la información pertinente, a partir de la cual se genera un informe de análisis, que será un análisis válido solo cuando sea firmado por los responsables de garantizar su corrección. A las reglas que indican qué camino recorre la información y obligan a que se sigan solo los caminos válidos se las llama *reglas de flujo*.

1.3.1 Cuando ubicar las Reglas de Negocio de restricción en la base de datos.

Si se analiza un Sistema de Información como un esquema cliente/servidor no hay una única posibilidad a la hora de distribuir las Reglas de Negocio. Sin embargo, sí hay ciertas pautas que se pueden tener en cuenta a la hora de tomar una decisión, basadas en

una clasificación de las Reglas de Negocio. En especial para las reglas de negocio de restricción deben implementarse en el servidor, o en la capa intermedia. Dado que estas reglas contemplan restricciones en los datos que dependen casi siempre de información presente en varias tablas, llevar a cabo el control en el cliente puede implicar cierto tráfico de red, por lo que es más conveniente situar la implementación de la regla más cerca de los datos. Según (Solivares, 2009) el lugar ideal podría ser el servidor, pero aquí nos encontramos con las limitaciones del SQL de los gestores de base de datos, y con el posible problema del acceso a diversas bases de datos, por lo que ubicar estas reglas en la capa intermedia puede ser de nuevo una buena solución.

1.4 Como implementar las Reglas de Negocio en la Base de Datos.

Las reglas se pueden implementar de formas disímiles e incluso pueden existir diferentes técnicas para una misma regla. En (Morgan, 2002) muestra como seria para las bases de datos:

Es probable que las Reglas de Negocio sean más naturales dentro de la Base de Datos, donde pueden tener un contacto más directo con los datos. Los gestores de Base de Datos más usados son Oracle, SQL Server y Postgres. Otros productos de base de datos correlativos tienen ampliamente capacidades similares, por lo que no es nada complejo extender esta investigación a otros lenguajes. En el caso del Servidor SQL, puede programarse en un dialecto llamado Transact-SQL.

Lo más habitual es querer usar las Reglas de Negocio respecto a funcionalidades que estén alrededor de lo básico (CREATE, READ, UPDATE y DELETE). Estos mecanismos son comunes a todos los servicios de datos.

1.4.1 Recursos de Base de Datos para la implementación.

El lenguaje SQL estándar (SQL 99) y particularmente la implementación para SQL Server 2000 (Transact-SQL) ofrecen varios recursos de manejo de datos que se analizan

en esta sección para seleccionar a la postre los más convenientes en la implementación de reglas de negocio, tipo restricción.

Restricciones (Constraints) CHECK:

Las restricciones CHECK exigen la integridad del dominio mediante la limitación de los valores que puede aceptar una columna. Además, determinan los valores válidos a partir de una expresión lógica que no se basa en datos de otra columna. Se puede crear una restricción CHECK con cualquier expresión lógica (booleana) que devuelva TRUE (verdadero) o FALSE (falso) basándose en operadores lógicos.

Es posible aplicar varias restricciones CHECK a una sola columna. Éstas se evalúan en el orden que fueron creadas. También es permitido aplicar una sola restricción CHECK a varias columnas si se crea al nivel de la tabla (SQLServer2000, 2004).

Desencadenadores (Triggers):

Los desencadenadores de Microsoft SQL Server 2000 son una clase especial de procedimiento almacenado definido para la ejecución automática al emitirse una instrucción UPDATE, INSERT o DELETE en una tabla o una vista. Son una herramienta eficaz que pueden utilizar los sitios para exigir automáticamente las reglas comerciales cuando se modifican los datos. Amplían la lógica de comprobación de integridad, valores predeterminados y reglas de SQL Server, aunque se deben utilizar las restricciones y los valores predeterminados siempre que estos aporten toda la funcionalidad necesaria.

Los desencadenadores pueden consultar otras tablas e incluir instrucciones SQL complejas. Son especialmente útiles para exigir reglas o requisitos complejos. También son útiles para exigir la integridad referencial, que conserva las relaciones definidas entre tablas (SQLServer2000, 2004).

Vistas:

Las vistas suelen utilizarse para centrar, simplificar y personalizar la percepción de la base de datos para cada usuario. Pueden emplearse como mecanismos de seguridad, que permiten a los usuarios obtener acceso a los datos por medio de ella, pero no les conceden el permiso de obtener acceso directo a sus tablas subyacentes.

Las vistas también se pueden utilizar para realizar particiones de datos y para mejorar el rendimiento cuando estos se copian. Además, permiten a los usuarios centrarse en datos

de su interés y en tareas específicas de las que son responsables. Los innecesarios pueden quedar fuera de la vista; de ese modo, también es mayor la seguridad de los datos, dado que los usuarios sólo pueden ver los definidos en la vista y no los que hay en la tabla subyacente (SQLServer2000, 2004).

Funciones definidas por el usuario:

Microsoft SQL Server 2000 (SQLServer2000, 2004) permite crear funciones definidas por el usuario. Al igual que cualquier función, son rutinas que devuelven valores. Dependiendo del tipo de valor que devuelven, estas pueden entrar en una de las tres categorías siguientes:

Funciones que devuelve una tabla de datos actualizable: si una función definida por el usuario contiene una única instrucción SELECT y dicha instrucción es actualizable, el resultado tabular devuelto por la función también es actualizable.

Funciones que devuelve una tabla de datos no actualizable: si una función definida por el usuario contiene varias instrucciones SELECT, o una instrucción SELECT no actualizable, el resultado tabular devuelto por dicha función no es actualizable.

Funciones que devuelve un valor escalar: las funciones definidas por el usuario pueden devolver valores escalares.

Procedimientos Almacenados:

Los procedimientos almacenados pueden facilitar en gran medida la administración de la Base de Datos y la visualización de información sobre esta y sus usuarios. Es una colección pre compilada de instrucciones SQL e instrucciones de control de flujo opcionales, almacenadas bajo un solo nombre y procesadas como una unidad. Estos se guardan en una base de datos, se pueden ejecutar desde una aplicación y permiten variables declaradas por el usuario, ejecución condicional y otras funciones eficaces de programación.

Los procedimientos almacenados pueden contener flujo de programas, lógica y consultas a la base de datos; aceptar parámetros y proporcionar sus resultados, devolver conjuntos de resultados individuales o múltiples y devolver valores.

Se pueden utilizar procedimientos almacenados para cualquier finalidad que requiera la utilización de instrucciones SQL, con estas ventajas:

Ejecución de una serie de instrucciones SQL en un único procedimiento almacenado.

Referenciar a otros procedimientos almacenados desde el propio procedimiento almacenado, con lo que se puede simplificar una serie de instrucciones complejas.
El procedimiento almacenado se compila en el servidor cuando se crea, por tanto, se ejecuta con mayor rapidez que las instrucciones SQL individuales.
(SQLServer2000, 2004).

Assertions:

Las *assertions* son un tipo de restricción especial que se puede especificar en SQL sin que deba estar asociada a una tabla en particular, como es el caso de los *constraints* (Mota, 2005). Generalmente son utilizados para describir restricciones que afectan a más de una tabla. Como los *constraints* sólo pueden establecerse sobre tuplas de una tabla, las *assertions* son útiles cuando es necesario especificar una condición general de la base de datos que no se puede asociar a una tabla específica. Por esta característica de poder especificar una assertion para toda la base de datos y no para una tabla en particular, se le llama restricción autónoma. Esto tiene grandes ventajas a nivel conceptual, pero las hace difíciles de implementar.

1.5 Características de la herramienta usada para generar las Reglas de Negocio de restricción en la Base de Datos automáticamente.

La herramienta es en esencia, un editor de reglas de negocio en lenguaje técnico, cuenta con un compilador que genera el código SQL para una regla de negocio especificada en lenguaje técnico, este no realiza un chequeo de tipo, análisis sintáctico ni análisis semántico, siguiendo un patrón de edición. Se utilizan los *triggers* como recurso del SQL para implementar las reglas del negocio. La herramienta permite también activar las reglas en la base de datos. Todas las funcionalidades son a través de un repositorio el cual es un documento XML.

1.5.1 Representación de las Reglas de Negocio en lenguaje técnico.

El lenguaje técnico creado para captar las reglas tratadas en este estudio, se obtiene de la estructura que ofrece el patrón de restricción de (Morgan, 2002), en la cual mediante las <características> y los <hechos> se especifican las restricciones al <sujeto> (Pérez

Alonso). Para lograr entonces, reglas flexibles y poderosas, se debe contar con un lenguaje dentro de las <características> y los <hechos> que permita restringir correctamente al <sujeto>. Esto es posible navegando por la notación punto que brinda la posibilidad de establecer relaciones y acceder a los atributos deseados.

Es necesario también tener en cuenta la utilización de funciones como las anteriormente vistas para las operaciones con elementos múltiples, además del uso de las conjunciones "and" y "or" para tener varias restricciones en una misma regla. Esto, en conjunto con la lógica del patrón de restricción, ofrece un lenguaje con la potencia necesaria para establecer restricciones a los objetos del negocio, obteniendo reglas equivalentes al lenguaje OCL.

Una expresión de regla en lenguaje técnico sería:

Un Paciente no puede tener sizeof(Evolucion.idEvolucion) > 30

<Sujeto>: Paciente

<Características>: sizeof (Evolucion.idEvolucion) > 30

A esta regla le corresponde en lenguaje natural la siguiente:

Un paciente no puede tener más de 30 evoluciones.

1.5.2 Notación punto.

En (Pérez Alonso, 2008) se utilizó esta notación en el lenguaje técnico para acceder y navegar entre las relaciones.

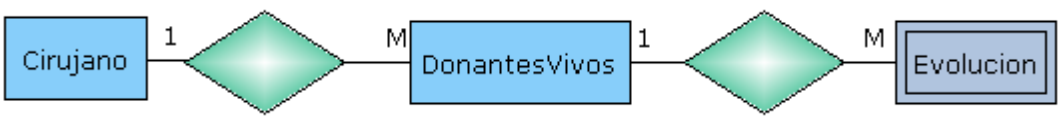
Acceso simple a un atributo:

Entidad_1.Atributo

Navegar entre relaciones de entidades:

Entidad_1.Entidad_2. (…) .Entidad_n.Atributo

Por ejemplo, si se desea acceder al nombre del paciente se puede alcanzar anotando Paciente.Nombre, pero también es posible navegar en las relaciones. En el siguiente esquema:



Al indicar Cirujano.DonantesVivos.Evolución se quiere expresar en lenguaje natural "Las Evoluciones de los Donantes Vivos atendidos por un Cirujano…".

Nótese que es importante el orden, pues al formular Evolución.DonantesVivos.Cirujano se estaría expresando en lenguaje informal "El Cirujano cuyo Donante Vivo tiene una Evolución… ".

Al utilizar la navegación de la notación punto para establecer relaciones entre entidades y acceder a los atributos de estas, se obtendrán dos tipos de resultados principales debido a la cardinalidad entre los objetos del negocio; pues como mismo se puede especificar un solo miembro del negocio, también es viable hacer referencia a un conjunto de estos y resulta necesario discernir ambos casos.

1.5.3 Patrón del Editor.

En (Pérez Alonso, 2008) se define el patrón de restricción siguiente:

<Determinante> <sujeto> (no puede tener <características>) |

(Puede tener <características> sólo si <hechos>).

Elemento	**Significado**
<determinante>	Es el determinante para cada sujeto, por ejemplo: Una, Uno, El, La, Cada, Todos. Según el mejor sentido en la redacción.
<sujeto>	Es un elemento de la Base de Datos del negocio, tal como entidades y objetos. La entidad puede ser cualificada por otros atributos descriptores, tales como la existencia en un estado particular o relacionada con una aplicación específica de la regla.
<características>	Describe las características del sujeto en el negocio, tanto internas como relacionadas con otras entidades.
<hechos>	Hechos relativos al estado o comportamiento de la Base de Datos del negocio, incluyendo o no al sujeto.

1.6 La representación de las Reglas de Negocio en lenguaje formal.

Varios son, según se ha visto, los mecanismos y recursos del SQL y las bases de datos para capturar las diferentes reglas de negocio. No obstante, la aplicación desarrollada por Pérez Alonso, solo utiliza los *triggers*. En su reporte anota cómo las restricciones de una columna son más eficientes si se implementan con un *check constraint*, pero no las materializa en la aplicación.

Aunque se ha mencionado en qué consiste un *trigger*, es de gran ayuda contar con la definición formal. En (SQLServer2000, 2004) se expresa como:

> […] una clase especial de procedimiento almacenado que se define para que se ejecute automáticamente cuando se emita una instrucción UPDATE, INSERT, O DELETE en una tabla o una vista.

Los *triggers* amplían la lógica de comprobación de integridad de restricciones. Son utilizados para implementar aquellas restricciones que no pueden ser captadas por otros recursos de las bases de datos.

Como se ha mencionado, los *triggers* se crean en tablas o vistas. Parte de la sintaxis de la creación es la siguiente:

CREATE TRIGGER *trigger_name*

ON {table | *view}*

Según se aprecia, es obligatorio especificar el lugar donde ocurre el evento, lo cual es uno de los objetos de análisis.

Los pacientes con enfermedad renal crónica pueden recibir terapia con diálisis para salvar la vida hasta tanto se pueda encontrar un donante de riñón. El riñón donado puede provenir de:

- Un donante familiar vivo: (genéticamente emparentado con el receptor: padres, hermanos o hijos).
- Un donante muerto (persona recientemente fallecida que no ha tenido enfermedad renal crónica).

Es así que la medicina de trasplante añade la figura de un tercer elemento en la relación médico-paciente: el donador de órganos, el cual es un sujeto sano que inesperadamente se ve involucrado en la solución de un problema que en realidad no le pertenece. Se enfrenta así uno de los escenarios más complejos en el quehacer médico: la relación médico-paciente-donador.

La compleja figura del donador vivo, en la práctica de la medicina de trasplante, obliga a la definición precisa de criterios de seguridad y respeto a la capacidad de decisión con relación a la donación (Reyes-Acevedo, 2005). Esta figura nace por necesidad y se mantiene también por necesidad, basada en tres requisitos esenciales (Reyes-Acevedo, 2005):

- Altas posibilidades de éxito en proveer de una mejor calidad de vida en el receptor que otras opciones disponibles.
- El riesgo de la donación debe ser bajo y aceptable para el donador, el receptor y el médico.
- La donación debe ser voluntaria y de un donador suficientemente informado.

En la actualidad, con la aparición de nuevos fármacos inmunosupresores, se está logrando una mínima mortalidad, aun así sigue siendo un procedimiento quirúrgico electivo o semi-electivo que se realiza a pacientes que han sido sometidos a una evaluación cuidadosa(Redín et al., 2006).

Básicamente la consulta de trasplante renal a nivel internacional se desarrolla de la siguiente manera. Un paciente acude a la consulta de nefrología con padecimiento renal. En esta consulta el médico examina al paciente. Mediante análisis y pruebas diversas se determina el estado general del paciente FGT.

En cuanto a la gravedad de su insuficiencia renal crónica (IRC), clasificada por cinco estados, I, II, III, IV y V. se remite al paciente al Área de Salud a recibir asistencia en caso de estadío I o II; a llevar su seguimiento en Consulta de Progresión si fuese estadío III o IV; o directamente a recibir Métodos Sustitutivos en caso de estadío V.

Para recibir Métodos Sustitutivos, el paciente debe someterse a un Análisis de Pre-requisitos para Métodos Sustitutivos comprendido dentro del protocolo de pre-trasplante.

En consulta de progresión se recepciona al paciente y se crea una nueva historia clínica en caso de que no exista. Posteriormente el especialista valora los análisis hechos al paciente antes de ingresar en consulta de progresión y procede a realizar examen físico.
Con los resultados de dicho examen físico, la enfermera llena los datos de la consulta y el médico orienta análisis, los cúales son realizados por el técnico de laboratorio.
En el caso en que el paciente comience con el protocolo de trasplante, lo primero es analizar el padecimiento a tratar con TR. Luego el especialista determina si seria efectivo y posible o no realizar el TR. Si el tratamiento adecuado al padecimiento es TR, se analizan las enfermedades que puedan interferir en dicho trasplante, tanto como el estado físico y mental del paciente.
Además se analizan las especificidades a tener en cuenta con respecto al sexo. En caso de alguna enfermedad que interfiera en el TR se asiste con el objetivo de curar y de no ser posible el tratamiento de dicha enfermedad se remite el paciente a recibir tratamiento de Hemodiálisis. Cuando el paciente pendiente a trasplante no cuenta con condiciones físicas y mentales favorables, se remite a recibir tratamiento de Hemodiálisis. Con respecto al sexo, en caso de ser mujer el embarazo constituye una importante atenuante para la operación de TR, por lo que, es necesario realizar prueba de embarazo y en el supuesto caso de estar embarazada, solo se procede con el TR o primeramente con el ingreso del paciente a la lista de espera si se obtiene el consentimiento del paciente, por escrito, para realizar la operación. Cuando tratamos con un paciente másculino es importante tener en cuenta los resultados de TR y PSA.
Luego de atravesar todo este proceso de requisitos, si el paciente es aceptado por sus condiciones para realizársele la operación de TR, ingresa a la Lista de Espera por Donante Candidato disponible, donde él espera por un donante asociado. Este donante puede ser Vivo o Cadavérico. Al encontrar dicho Donante asociado, comienza por parte del equipo médico la búsqueda de la pareja Donante-Receptor.
Cuando se obtiene una posible pareja Donante-Receptor, se prosigue con la segunda etapa del protocolo Pre-Trasplante, en la cual se informa a la pareja de todo el proceso y se espera el consentimiento del paciente para TR. Entonces se comienza Protocolo de Compatibilidad y se completa dicho protocolo si el paciente es Compatible Inicial para

determinar si finalmente es compatible. Al determinar que es compatible se continúa con Estudios de Protocolo Complementario.

Finalizados Estudios de Protocolo Complementario se selecciona órgano para trasplante. Cuando se cuenta con una pareja Donante-Receptor compatible, se procede a realizar el trasplante, donde lo primero es fijar una fecha. Fijada la fecha se ingresan Donador y Receptor y se procede con la operación. Luego de la operación el paciente pasa a post operatorio para su recuperación.

1.7 Conclusiones del capítulo

En este capítulo se ha brindado un marco teórico sobre las reglas de negocio, haciendo hincapié en las de tipo restricción, de las cúales se ofreció un patrón que las define para su posible implementación y almacenamiento adecuado. Esto proporciona un conocimiento preliminar sobre los objetivos generales de este trabajo, y facilita la comprensión de las decisiones a tomar posteriormente. Se estudiaron los recursos de Base de Datos que pudieran ser empleados para implementar reglas.

Capítulo2 “Propuesta de transformaciones”

Capítulo2 “Propuesta de transformaciones”

Es de interés estudiar el estado de la Base de Datos, así como las transformaciones que debe sufrir para que se pueda usar la Aplicación para generar Reglas de Negocio de Restricción. Se analizará el desempeño del editor para un grupo de reglas de restricción, las cúales fueron revisadas por los especialistas del negocio. Todo esto debe culminar con una propuesta de transformaciones a la aplicación. ¿Cúales transformaciones se deben implementar? A esta pregunta es a la que debe dar respuesta este capítulo.

2.1 Características de la Base de Datos en el Sistema de Información basado en reglas.

Al extraer un grupo de reglas de negocio tipo restricción que serán algunas de las que gobernarán el Sistema de Información se puede apreciar una de las principales deficiencias en el diseño de la base de datos, que es no haberse preguntado ¿Cuál es el propósito de la Base de Datos?, ¿Qué información debe tener?, ¿Qué información debe generar? , es decir el propósito de esta base de datos es convertirse en un componente de un Sistema de Información basado en reglas, por lo que debe tener en el diseño físico una correspondencia de las tablas que lo integran con las entidades del negocio que intervienen en las reglas extraídas, así como los atributos de cada tabla se deben corresponder con la información que se verificará en la regla en si para una entidad, por lo que no es un buen diseño para la base de datos aquel, en el que no se tengan en cuenta, las reglas que regirán el negocio.

2.1.1 Características que se revisan en el diseño físico de la Base de Datos de Trasplante Renal para utilizar la Aplicación para generar Reglas de Negocio de Restricción:

- Existencia de las tablas correspondientes por cada entidad del negocio.

- Existencia de los datos involucrados en las reglas del negocio.

- Existencia de las relaciones por las que se navegará en el lenguaje técnico de las reglas del negocio.

También se atiende a la presencia de llaves primarias y foráneas pues la base de datos con la que se cuenta es el resultado de una conversión de MYSQL a MSSQL SERVER 2000, donde se produjeron problemás con la generación adecuadas de estas.

2.1.2 Análisis de la persistencia de los datos en las Reglas de Negocio de Restricción especificadas para la problemática de Trasplante Renal.

Se aborda un grupo de Reglas de Negocios de Restricción con las que se cuenta donde se garantiza la persistencia de los datos involucrados en estas, se asegura no encontrar una regla de negocio que involucre una información que no debe estar plasmada en la Base de Datos. Un ejemplo es que los cirujanos deben lavarse las manos antes de entrar al salón.

2.1.3 Análisis del diseño físico de la Base de Datos para expresar las Reglas de Negocio en lenguaje técnico.

Como se menciona en el epígrafe 2.1 el diseño inicial de la base de datos no comprendía todas las reglas de restricción que controlarían el Sistema de Información, por lo que cuando una regla de negocio se expresa en lenguaje técnico es necesario verificar los requisitos expuestos en el epígrafe 2.1.1. Esto lleva consigo que la base de datos se ha modificado hasta satisfacer dichos requisitos. Esto se puede apreciar en el siguiente ejemplo:

Un Paciente Candidato a trasplante con evidencias de actividad bioquímica no puede estar en la lista de trasplantes**.**

Esta regla es bien sencilla de expresar en lenguaje técnico pero al revisar la base de datos se observa que la tabla Trasplante no estaba relacionada con PacienteCandidato y no

existía donde comprobar la actividad bioquímica del paciente candidato, por lo que fue necesario implementar la relación entre estas tablas y la inclusión de un campo ActivBio de tipo boolean, haciendo válida la sentencia en lenguaje técnico siguiente:

Un Trasplante no puede tener

Trasplante.PacienteCandidato.ActivBio = true

2.2 Algunas reglas que fueron probadas para Aplicación para generar Reglas de Negocio de Restricción.

Regla 1:

Un Paciente puede ser receptor con donante vivo sólo si su edad está entre 15 y 55 años.

Un DonanteVivo no puede tener

DonanteVivo.Receptor.Paciente.edad > 55 or Receptor.Paciente.edad< 15

Regla 2:

Un Paciente Candidato a trasplante con evidencias de actividad bioquímica no puede estar en la lista de trasplantes.

Un Trasplante no puede tener

Trasplante.PacienteCandidato.activbio = true

Regla 3:

Todo Paciente Candidato a trasplante no puede ingresar en la lista de trasplante A MENOS QUE se les realiza una evaluación completa**.**

Un Receptor no puede tener

Receptor.PacienteCandidato.evaluacion <> completa

Regla 4:

Un paciente mayor de 50 años y con Diabetes Mellitus no puede ser considerado Paciente Candidato a Trasplante sólo si es evaluado por coronariografía.

Un Receptor puede tener

Receptor.PacienteCandidato.EnfermedadesContraidas.Enfermedades.nombre = Diabetes Mellitus and Receptor.PacienteCandidato.Paciente.edad > 50
solo si
Receptor(@idsujeto).Paciente.Analisisrealizado.Analisis.nombre = coronariografia

Regla 5:
Un *trasplante* puede ser realizado sólo si el *receptor* tiene vasos adecuados.
Un Trasplante no puede tener
Trasplante.Receptor.Paciente.Analisisrelizado.Analisis.nombre = vasos and
Trasplante.Receptor.Paciente.Analisisrealizado.resultado <> adecuados

Regla 6:
Un trasplante puede establecerse sólo si la circulación de las extremidades del receptor no está comprometida.
Un Trasplante no puede tener
Trasplante.Receptor.paciente.analisisrelizado.analisis.nombre = Circulación and
Trasplante.Receptor.paciente.analisisrealizado.resultado = comprometida

Regla 7: Todo Trasplante de Riñón es con Donante Vivo o Donante Cadavérico.
Un Trasplante no puede tener
Trasplante.TipoDonante <>'Vivo' and Trasplante.TipoDonante <>'Muerto'

Regla 8: Todo Donante Potencial Vivo tiene que ser de 1ra línea de consanguinidad.
Un Receptor no puede tener
Receptor.DonanteVivo.idparentesco <> 1

Regla 9: La Edad para el Donante está entre 18 y 55 años.
Un DonantePotencial no puede tener
Donantepotencial.Paciente.edad < 18 or Donantepotencial.Paciente.edad > 55

Regla 10: Si Donante Potencial presenta alguna de estas Contraindicaciones entonces no puede ser Donante:

VIH Sida, Hepatitis B, Hepatitis C

Un DonantePotencial no puede tener
DonantePotencial.Paciente.EnfermedadesContraidas.Enfermedades.nombre = VIH or
DonantePotencial.Paciente.EnfermedadesContraidas.Enfermedades.nombre = Hepatitis B
or
DonantePotencial.Paciente.EnfermedadesContraidas.Enfermedades.nombre = Hepatitis C

Regla 11: Si existe un Riñón con Doble Arteria y la Función Renal no tiene Diferencia Significativa (<5%) para Donador Vivo entonces se toma Riñón Contra lateral.
Un Trasplante no puede tener
Trasplante.DonanteVivo.Riñón.arteria = doble and
Trasplante.DonanteVivo.Riñón.diferencia significativa < 5

Regla 12: Si ambos Riñones presentan Función Renal sin Diferencia Significativa (<5%) y Arteria Renal Única para Donador Vivo entonces se toma de presencia Riñón Izquierdo.
Un Trasplante no puede tener
Trasplante. DonanteVivo.Riñón.arteria = única and
Trasplante. DonanteVivo.Riñón. Riñóniz.significativa < 5 and
Trasplante. DonanteVivo.Riñón. Riñónde.significativa < 5 and
Trasplante. Riñón = derecho

Regla 13: La Cirugía se realiza preferiblemente martes o miércoles en Sala de Trasplante.
Un Trasplante no puede tener
Trasplante.fecha = martes or Trasplante.fecha = miércoles

Regla 14: Al ingreso de todo paciente que va a ser trasplantado, la enfermera comprobará que al paciente se le han realizado todas las pruebas establecidas en el protocolo "Receptor renal"

Un Trasplante no puede tener
Trasplante.Receptor.Analisisrealizados.Analisis.nombre <> "Receptor renal"

Regla 15: Un paciente con alguna de las siguientes características deben ser evaluados con coronariografía:
Diabetes
Mayores de 50 años
Historia sugerente de ángor
Un Paciente puede tener
Paciente.EnfermedadesContraidas.enfermedades.nombre = Diabetes Mellitus or
Paciente.edad > 50

or historia sugerente de ángor ..
sólo si
paciente(@idsujeto).analisisrealizado.analisis.nombre = coronariografia

Regla 16: Un donante potencial vivo solo puede ser donante de un solo órgano**.**
Un DonantePotencial no puede tener
DonantePotencial.DonanteVivo.CI = DonantePotencial.CI

Regla 17: Si Donante Potencial presenta algunas de estas Contraindicaciones entonces se puede valorar por parte de Equipo de Trasplante si puede ser Donante:
Familiar con Enfermedad Poliquística.
Edad > 30 y no tiene Expresión Clínica en Estudios de Laboratorio. (No tiene realizado análisis).
Paciente con Carga Genética para Diabetes Mellitus.
Un DonantePotencial no puede tener
DonantePotencial.Paciente.Edad > 30 and
DonantePotencial.EnfHereditaria = 'Poliquística'
DonantePotencial.Paciente.AnalisisRealizado = null

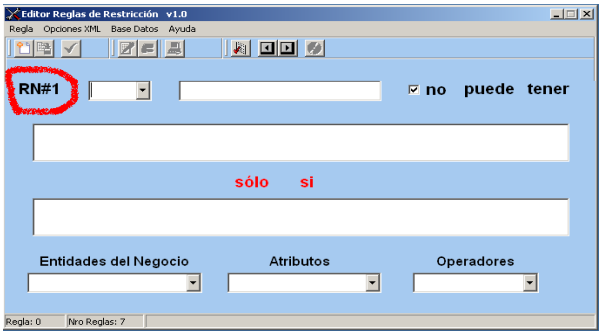
2.2.1 Características del uso de la Aplicación para Reglas de Negocio de Restricción.

Como se ha visto en este capítulo el uso de la aplicación es un momento crucial para el Sistema de Información, pues en la edición de una regla es importante conocer si la base de datos soporta lo planteado en epígrafe 2.1.1. La aplicación permite ver las tablas con sus atributos, lo que posibilita chequear dos de los aspectos planteados. El tercer aspecto consistente en saber si existen las relaciones entre las tablas por las que se navega atreves de la notación punto usada por el editor, no es posible chequearlo, haciéndose engorroso para el usuario el tener que ir primero con la base de datos y revisar que estas existen, otra limitación del compilador de la aplicación es el que no realice un chequeo de tipo lo que puede generar un problema a la hora de activar la regla en la base de datos.

2.2.2 Limitaciones de la aplicación para generar las Reglas de Negocio en la Base de Datos.

- No permite hacerle corresponder a cada regla en lenguaje técnico su lenguaje natural, aun cuando existe una tabulación en el repositorio para esta información, haciendo menos legible la sentencia en lenguaje técnico.
- Comienza la asignación del identificador de cada regla por RN#1 en cualquiera sea el repositorio en el que se trabaja, no siendo posible diferenciar por su nombre la regla en la base de datos.

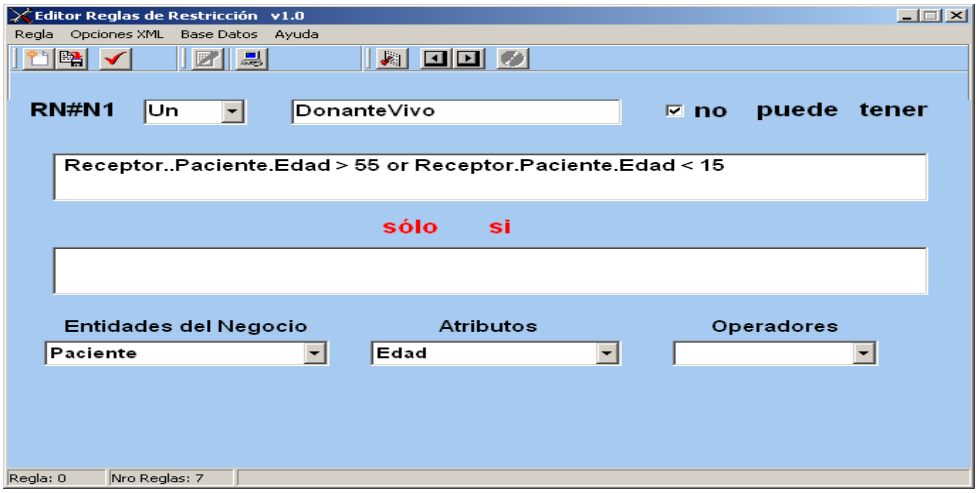
- El compilador no especifica donde o porque falla una regla en una característica o en un hecho, siendo difícil saber el tipo del error en la sentencia. (véase ejemplo 1)
- No realiza análisis semántico, genera la regla sin verificar que la relación exista, generando en ocasiones reglas con errores sintácticos en el código generado. (véase ejemplo 2)
- No realiza chequeo de tipos por lo que se puede generar bien el código y no ser posible generarse la regla para la base de datos por incompatibilidad de tipos, siendo difícil para el usuario identificar donde está el error. (véase ejemplo 2)



Ejemplo 1:

A continuación veremos un ejemplo de regla en el editor con problemás sintácticos:

Un Paciente puede ser receptor con donante vivo sólo si su edad está entre 15 y 55 años.



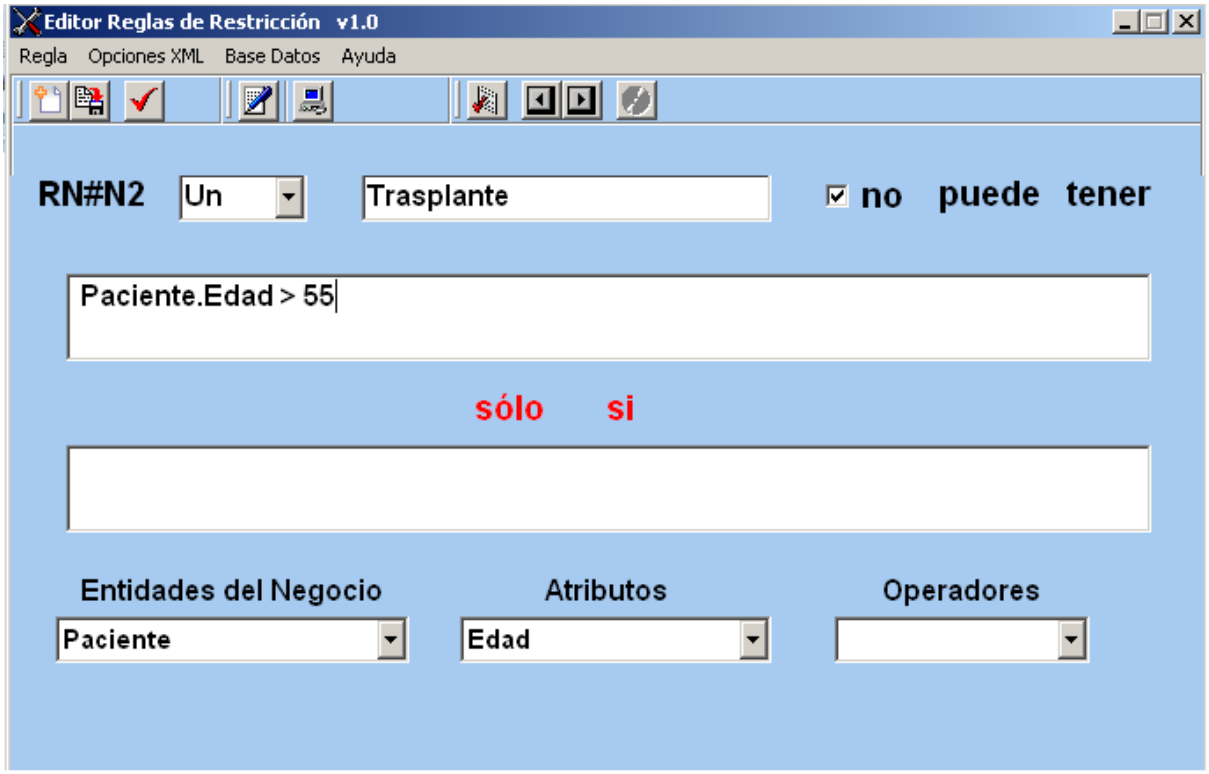
Nótese en las características la presencia de un error sintáctico al estar dos puntos seguidos. Al compilar la regla el mensaje de error no es específico en donde ocurre, siendo mostrado el texto siguiente:

Ejemplo 2:
En este ejemplo se mostrará como al no realizar un chequeo semántico, no se comprueba la existencia de la relaciones entre las entidades que intervienen en la regla, generando un código con errores de sintáxis. No siendo posible activar la regla, lo que puede no comprender el usuario que la editó.
La regla usada para este ejemplo es sencilla:

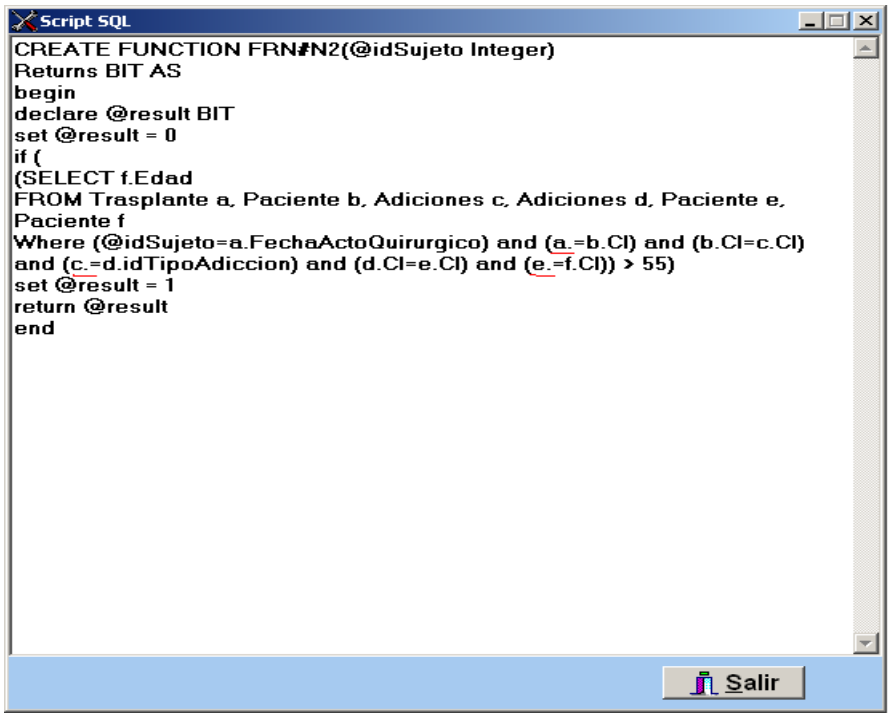
Un trasplante no se le puede realizar a un paciente mayor de cincuenta y cinco años.

En lenguaje técnico sería:
Un Trasplante no puede tener
PacienteTrasplante.Paciente.Edad > 55

Pero si se expresa de de la siguiente manera:



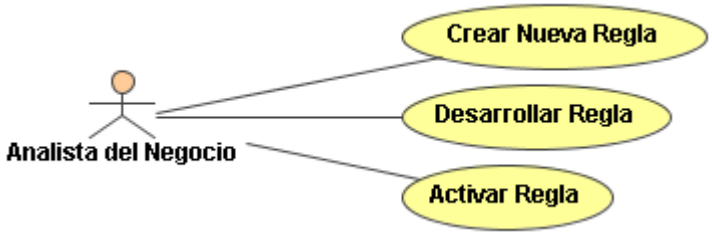
Al compilar esta regla el compilador no detecta error alguno y se genera el siguiente código con errores de sintáxis:



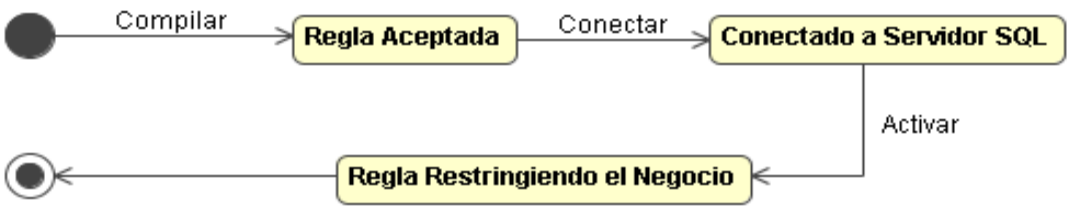
Este código mal generado es el resultado de no chequear la existencia de relación entre Trasplante y Paciente, además de consultar el catálogo en el momento que se necesita la información, en este caso la llave foránea de Paciente en la tabla Trasplante.

2.3 Análisis de los caso de usos de la aplicación.

En (Pérez Alonso, 2008) se definen los casos de uso de la siguiente manera:



Para el caso de uso activar regla el diagrama de estado sería este:



El problema comienza cuando se quiere desactivar la regla, puesto que las políticas del negocio pueden variar durante un tiempo. Es decir la necesidad de desactivar una regla durante un tiempo sin borrarla del repositorio no es atendida. En (Lorenzo, 2009)) se implementa el caso de uso modificar regla. Pero tampoco está integrado a la aplicación, por lo que se pudiera aumentar un caso de uso "Desactivar Regla", que respondiera a desactivarla de la Base de Datos pero no del repositorio, y permitir al usuario modificarla.

2.4 Obtención de la información necesaria del catálogo.

El catálogo del MS SQL Server provee de toda la información especificada. El catálogo de este proveedor comprende todas las tablas del sistema, las cuales contienen los metadatos. Ellas son:

- Tablas del sistema que sólo se encuentran en la base de datos máster
- Tablas del sistema que se encuentran en todas las bases de datos
- Tablas del Agente SQL Server de la base de datos msdb
- Tablas de la base de datos msdb
- Tablas para almacenar información de duplicación

De ellas son útiles aquí las tablas del sistema que se encuentran en cada base de datos. En su contenido recogen la información tan necesaria para el editor de reglas, a saber:

- Nombre de las tablas
- Nombre llaves primarias y foráneas
- Atributos

Esta información puede ser encontrada en las respectivas tablas del sistema:

- Sysobjects
- Sysobjects y sysforeignkeys
- Syscolumns

2.4.1 Cómo consultar el catálogo.

El catálogo puede ser consultado de varias formas, o al menos así se afirma en (Microsoft SQL Server, 2004). Ellas se enumeran a continuación:

- Vistas de esquema de información
- Conjuntos de filas de esquemás OLE DB
- Funciones del catálogo de ODBC
- Procedimientos almacenados y funciones del sistema

Aunque se pudiera, las tablas del catálogo del sistema no deben ser consultadas directamente. La arquitectura de tales tablas depende íntimamente de la versión de SQL Server que se utilice. Por tal razón se prefiere un método que nos aísle de la arquitectura. Los métodos anteriores emulan por su elección (Microsoft SQL Server, 2004).

El editor fue desarrollado en Delphi Pascal (Pérez Alonso, 2008), y una de las formas más fáciles de implementar y de trabajar en este ambiente de desarrollo son las componentes visuales. Dentro de las categorías que se agrupan en él, están las destinadas al trabajo con base de datos:

- Componentes InterBase nativos
- Biblioteca dbExpress
- BDE (Borland Database Engine)
- ADO (ActiveX Data Objects)

De ellos fue escogida la componente de ADO para Delphi nombrada dbGo.

El Delphi7 cuenta con una serie de componentes de ADO disponibles para su utilización en aplicaciones orientadas a bases de datos. ADOConnection para la conexión a una base de datos, ADOCommand ejecuta un comando SQL de acción, ADODataSet esta

desciende de TDataSet de propósito general, ADOTable para la encapsulación de una tabla, ADOQuery para la encapsulación de una sentencia SELECT o SQL, ADOStoreProc para la encapsulación de un procedimiento almacenado y RDSConnection para conexión a Remote Data Services.
Aunque todas son muy útiles, para la presente aplicación, en en el objetivo especifico del acceso al catálogo se utilizó la componente ADOConnection y ADOStoreProc.

2.5 Propuesta de transformaciones a la aplicación.

Como ha sido analizado en el presente capitulo, la aplicación presenta ciertas limitaciones que dificultan el trabajo a quien la utiliza en vista de generar las reglas para una base de datos de un negocio real. Se propone a continuación una serie de cambios a implementar en el compilador, este es una pieza clave para la aplicación, en las primeras versiones no fue tratado con demásiado interés puesto que el objetivo primario era lograr la edición de la regla en si, que harán más fácil y profesional el uso del editor.

- Para más comodidad en próximos estudios de la aplicación implementar la posibilidad de crear repositorio.
- Con el objetivo de eliminar la ambigüedad entre reglas generadas para la base de datos desde diferentes repositorios generar aleatoriamente los ID de las reglas.
- Se propone la implementación del caso de uso desactivar regla.
- Se propone que el acceso al catálogo sea solo al inicio de ejecutarse la aplicación, optimizando el tiempo de compilación y haciendo posible realizar un análisis semántico.
- Implementar tratamiento de errores en el análisis sintáctico.
- Implementar análisis semántico, teniendo en cuenta las siguientes condiciones:
    - Existencia de las entidades en la base de datos.
    - Existencia de los atributos en las entidades.
    - Existencia de las relaciones entre las entidades involucradas.

El tratamiento de errores y el análisis semántico, son indispensables para independizarnos de la base de datos, y evitarían la generación de código erróneo para la base de datos.

2.6 Conclusiones del capítulo.

Las principales interrogantes que motivan la presente investigación fueron presentadas en el capítulo que recién termina. Las condiciones que debe cumplir la base de datos para utilizar la Aplicación para reglas de negocio de restricción. Las transformaciones que se pueden implementar en vista de hacer más sencillo el uso del editor. Así como una propuesta de en qué momento acceder al catálogo. En todo lo respectivo se obtuvo que:

- La base de datos fue diseñada sin tener en cuenta la totalidad de las reglas que regirían el negocio.
- El diseño de la aplicación debe mejorarse del punto de vista de ingieneria de software.
- La aplicación es en estos momentos de difícil uso para un usuario común.
- El compilador debe ser modificado para justificar el uso de la herramienta. Pues es en el donde se brinda un lenguaje de interacción para el usuario.

Capitulo3 “Implementación de las transformaciones a la aplicación”

Capítulo3 "Implementación de las transformaciones a la aplicación"

Este capítulo trata el desarrollo de las modificaciones realizadas al editor para reglas de negocio, tipo restricción. Consta de tres secciones, una primera en la cual se aborda los cambios que se realizaron a la base de datos para un grupo de reglas de negocio en específico. La segunda sección es dedicada a los cambios del diseño del editor. La tercera y última sección discurre sobre la implementación de las transformaciones al compilador.

3.1 Revisión de la Base de Datos para cada una de las reglas en lenguaje natural y técnico.

La revisión de la base de datos atendiendo a las características brindadas en el capitulo anterior epígrafe 2.1.1, se realiza una vez la regla está en lenguaje natural. Pues de esta manera es que se conocen las entidades involucradas con sus respectivos atributos, y relaciones por las que se debe navegar. Estos son los motivos por los que la revisión se irá realizando a la vez se expresa la regla en lenguaje técnico. Solo se mostrarán algunos ejemplos por no extender demásiado el capítulo y las que resten se mostrarán en los anexos.

Regla 1:

Un Paciente puede ser receptor con donante vivo sólo si su edad está entre 15 y 55 años.

Un DonanteVivo no puede tener

DonanteVivo.Recptor.Paciente.edad > 55 or Receptor.Paciente.edad< 15

Las entidades DonanteVivo y Receptor no estaban relacionadas, se creó esta relación a través de un nuevo atributo en DonanteVivo, CIReceptor que es una llave foránea de CIReceptor de la tabla Receptor. Para esto se tuvo en cuenta que un DonanteVivo sólo puede donar un riñón.

Regla 2:

Un Paciente Candidato a trasplante con evidencias de actividad bioquímica no puede estar en la lista de trasplantes.

Un Trasplante no puede tener

Trasplante.PacienteCandidato.activbio = true

Para esto fue necesario crear el campo activbio de tipo bit, que brindaría la información de si un PacienteCandidato presenta actividad bioquímica o no.

Regla 3:

Todo Paciente Candidato a trasplante no puede ingresar en la lista de trasplante AL MENOS QUE se les realiza una evaluación completa.

Un Trasplante no puede tener

trasplante.PacienteCandidato.evaluacion <> 'completa'

Se creó el campo evaluación de tipo cadena en PacienteCandidato, para conocer el estado de la evaluación.

Regla 4:

Un trasplante puede establecerse solo si la circulación de las extremidades del receptor no está comprometida.

Un Trasplante no puede tener

Trasplante.Receptor.paciente.analisisrelizado.analisis.nombre = 'Circulación' and

Trasplante.Receptor.paciente.analisisrealizado.resultado = 'comprometida'

Aunque en el diseño primero implementado en MYSQL se relacionaban las tablas Trasplante y Receptor, en la base de datos con que se contaba no existía. Por lo que se aumento en Trasplante un campo CIReceptor que hace referencia a CIReceptor en la tabla Receptor.

Estos son varios ejemplos de las transformaciones que se le realizaron a la base de datos. Se puede apreciar que el resultado de estas transformaciones no es óptimo en un diseño de base de datos, esto reafirma lo señalado en el epígrafe 2.1.

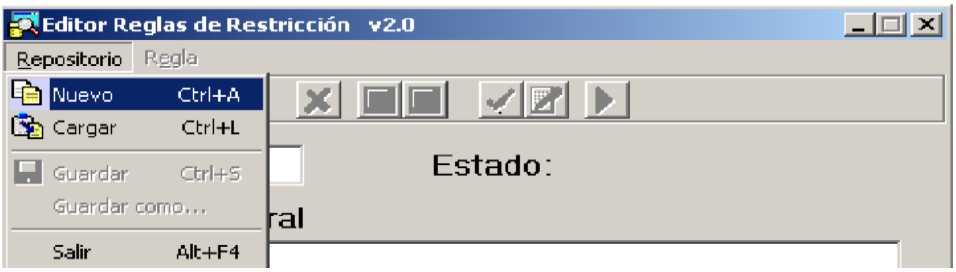
3.2 Implementación de las modificaciones al Editor de Reglas.

El Editor de Reglas fue sujeto a cambios en su interfaz y ello responde a la modificación en el comportamiento de uno de sus casos de uso. Los cambios no fueron tan drásticos, pero ameritan documentarse en aras de poner al corriente al programador de sus causas y consecuencias.

Las modificaciones al Editor de reglas responde a la propuesta de transformaciones que

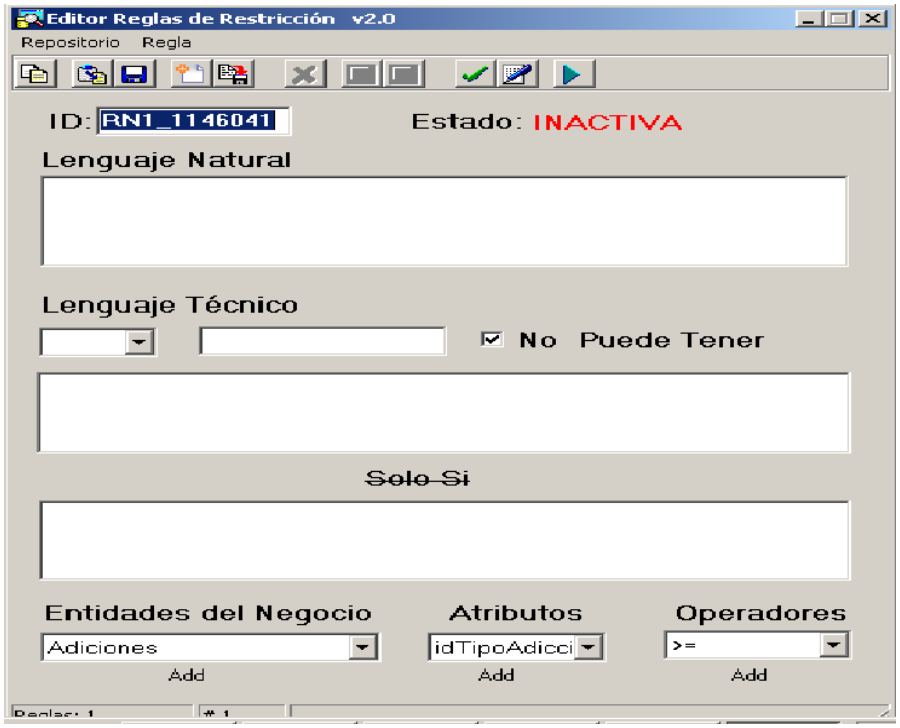
se hicieron en el capítulo 2 epígrafe 2.4. A continuación se muestran los cambios realizados:

- El usuario tiene la posibilidad de crear el repositorio en que se guardarán las reglas. Se crea un nuevo documento XML con la misma estructura definida en la primera versión de la aplicación



- Se cambió la forma en que se asigna el identificador de la regla, generándose de forma aleatoria un número de seis dígitos.
- Se muestra mediante etiquetas el estado de la regla, es decir si esta activa o no.
- Se incorpora al diseño una caja de texto donde se escribe la regla en lenguaje natural, para esto ya existía una tabulación en el documento XML pero el editor no la mostraba.

Estos cambios implementados mejoran el diseño del punto de vista de ingeniería de software, haciendo el trabajo con el Editor más sencillo.



3.2.1 Transformación del caso de uso desactivar regla.

Esta modificación del editor se estudia separada de las otras por tener un mayor grado de importancia en la aplicación. Es de entender que la mayor alteración al diseño anterior es esta. La implementación de esta funcionalidad se hizo modulada, porque el objetivo de la aplicación es ser utilizada como una capa de una herramienta mayor. Se implementó el método *DeactivateRule*(), que se invoca al hacer clic en el botón *Desactivar*. Se muestra una ventana para la confirmación de la acción, si se aprueba, se borra la regla de la base de datos pero no del repositorio, siendo posible modificarla, recompilarla y volver a activarla. El método es leído por una property de la clase *TDBDataModule*, y se implementó en la clase *TExtADOConnection*. Se muestra a continuación:

```
function TExtADOConnection.DeactivateRule(id: string): boolean;
var
  Query: TADOQuery;
begin
  Query := TADOQuery.Create(Owner);
  with Query do
  begin
```

```
    try
      Connection := Self;
      Close;
      SQL.Clear;
      SQL.Add('IF EXISTS (SELECT name FROM sysobjects');
      SQL.Add(Format('WHERE name = '#39'%s_%s'#39')',[id,'function']));
      SQL.Add(Format('DROP FUNCTION %s_%s',[id,'function']));
      ExecSQL;

      SQL.Clear;
      SQL.Add('IF EXISTS (SELECT name FROM sysobjects');
      SQL.Add(Format('WHERE name = '#39'%s'#39' AND type = '#39'TR'#39')',[id]));
      SQL.Add(Format('DROP TRIGGER %s',[id]));
      ExecSQL;

      Result := true;
    except
      Result := false;
    end;
  end;
end;
```

3.3 Implementación de las transformaciones al compilador.

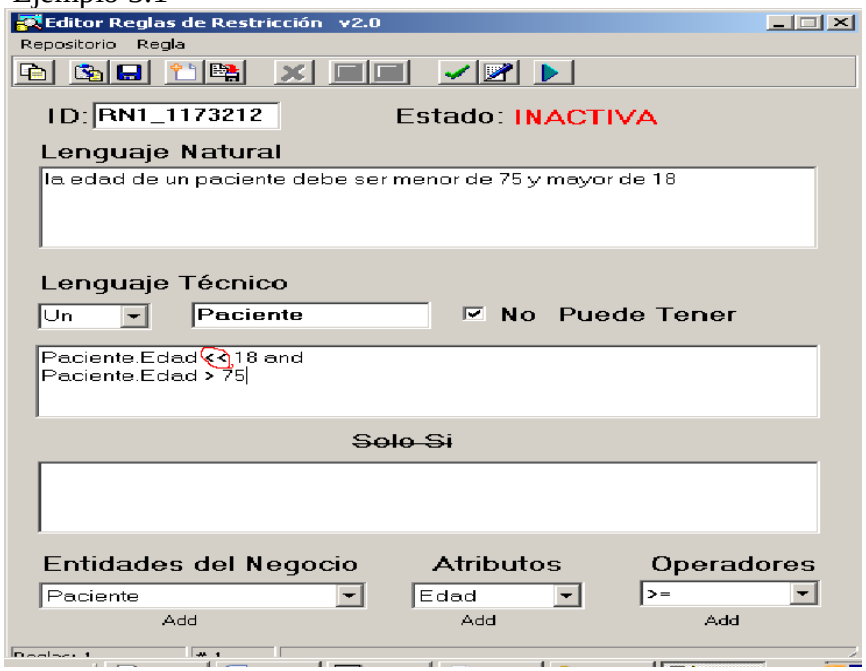
La implementación de las transformaciones al compilador se muestra en dos partes, la primera es el tratamiento de errores implementado para el analizador sintáctico, y la segunda el analizador semántico. Para realizar el análisis semántico es necesario conocer información del esquema; como los nombres de las tablas, el nombre y tipo de sus columnas, y las relaciones involucradas en la base de datos. Esta información será consultada y almacenada en estructuras de datos al inicio de la aplicación, y no a la hora de generar el código como está concebido en las versiones anteriores.

3.3.1 Tratamiento de errores en el análisis sintáctico.

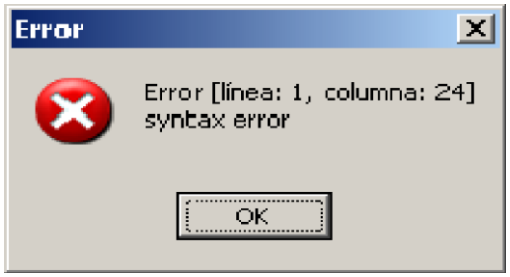
En el momento en que ocurre un error de compilación de tipo sintáctico, el módulo *TCompiler* captura el error e información adicional como la posición donde ocurre. Luego lo empaqueta un una estructura *TError* y lo envía al módulo principal, donde el formulario del editor notifica al usuario del fallo del proceso de compilación, mostrando la información adicional.

Como resultado de implementar el tratamiento de errores al analizador sintáctico, se obtiene mucha más claridad para la edición de una regla. Se humaniza más este proceso donde antes no se especificaba el error cometido, ni en que lugar. A continuación se muestra un ejemplo de las ventajas del tratamiento de los errores en el análisis sintáctico.

Ejemplo 3.1



Se puede ver claramente como el error es sintáctico, pues no se espera un operador detrás de un operador, en esta versión del compilador, la información que se brinda del error es más específica para el usuario.



3.3.2 Implementación del analizador semántico.

El analizador semántico es el encargado de velar por los errores que se cometen por desconocer cierta información, que no puede ser accedida en el editor. Un ejemplo de esto lo constituye el saber cuando existe relación entre dos entidades, cuando se compara con tipos incompatibles, etc. Para solucionar esta problemática es necesario contar con esa información. El acceso a esa información como se estudiaba en el capitulo dos era cuando se generaba la regla en lenguaje formal. Por lo que cambiará el momento en que se accede al catálogo de la base de datos. También será necesario implementar un chequeo de tipos y comprobar la existencia de dichas relaciones para que el usuario no necesite consultar la base de datos a cada momento, y lo más importante evitar se genere un código erróneo, lo que traería problemás para activar la regla.

Acceso a la información necesaria.

El acceso al catálogo es realizado al iniciar la aplicación. El objetivo de acceder al catálogo en el momento en que se inicia la aplicación es optimizar el tiempo de compilación, pues esta información será consultada constantemente. Desde que inicia el editor se muestra en los cuadros combinados las entidades del negocio con sus respectivos atributos, en el momento de compilar la regla editada se comprueba que el sujeto sea una entidad, que cada identificador que no sea un atributo sea una entidad, se comprueba también el que la lista de entidades concatenadas por puntos que representa la navegación por sus relaciones no termine en algo que no sea un atributo de la última entidad. Por último se comprueba que las entidades concatenadas por puntos estén relacionadas, para esto se implementó un método que valida cuando dos entidades están relacionadas. Este método utiliza la información que se extrajo del catálogo, que es almacenada en variables que se inicializan con la aplicación. Fue necesario crear nuevos tipos que serán mostrados a continuación:

```
type
  TDBField = class
  public
    Name, Type_Name: string;
    Data_Type, Nullable: integer;
```

```
end;
```

TDBField: Es la estructura donde se almacena la información necesaria de cada atributo: el nombre, el tipo de dato, el nombre del tipo de dato y si es permitido que no se le asigne valor.

```
TDBTable = class
public
  Name, Owner, Cualifier: string;
  FieldsList: THashedStringList;
  constructor Create();
  destructor Destroy;override;
end;
```

TDBTable: Es el tipo donde se guarda la información de cada tabla: el nombre, el propietario y una lista de atributos.

```
TDBRelation = class
public
  Primary, Foreing: string;
  PKTable, FKTable: TDBTable;
end;
```

TDBRelation: Es en esta estructura donde se almacenan las relaciones: el nombre de el campo que es llave primaria, el nombre del campo que es llave foránea y que referencia a esa llave primaria, y que tabla contiene la llave primaria y cual la llave foránea.

Estos tipos se inicializan al acceder al catálogo, obteniendo todas las tablas con sus atributos, y una lista de relaciones entre las tablas. Estos tipos fueron definidos en el módulo *DBManager*. A continuación se muestra como se extrae toda la información. Para lo que fue necesario extender la clase *TADOConnection* con un método llamado *RetrieveInformation* encargado de la extracción de la información.

TExtADOConnection = class (TADOConnection)

```
  private
    mDBName : string;
    mDBTables: THashedStringList;
    mDBRelations: array of TDBRelation;

    nTables, nRelation: integer;
  public
    constructor Create(AOwner: TComponent);
    destructor Destroy;override;

    procedure RetrieveInformation(ProgressBar: TProgressBar = nil);
    function GetTable(TableName: string): TDBTable;
    function GetRelation(k1, k2: string): TDBRelation;

    function IsRuleActive(id: string): boolean;
    function ActivateRule(id, table, body, errormsg: string): boolean;
    function DeactivateRule(id: string): boolean;

    function CompatibleTypes(t1, t2: string): boolean;

    property DBName: string read mDBName write mDBName;
    property DBTables: THashedStringList read mDBTables write mDBTables;
  end;
```

Método para comprobar la existencia de relaciones entre dos entidades.

Al contar con toda la información necesaria se implementa un método que devuelve un objeto TDBRelation con la relación entre dos tablas. Pues no es necesario buscar por todas las relaciones de una tabla, sino que solo nos importa cuando dos tablas están relacionadas entre ellas. Este método se implementó en la clase TExtADOConnection:

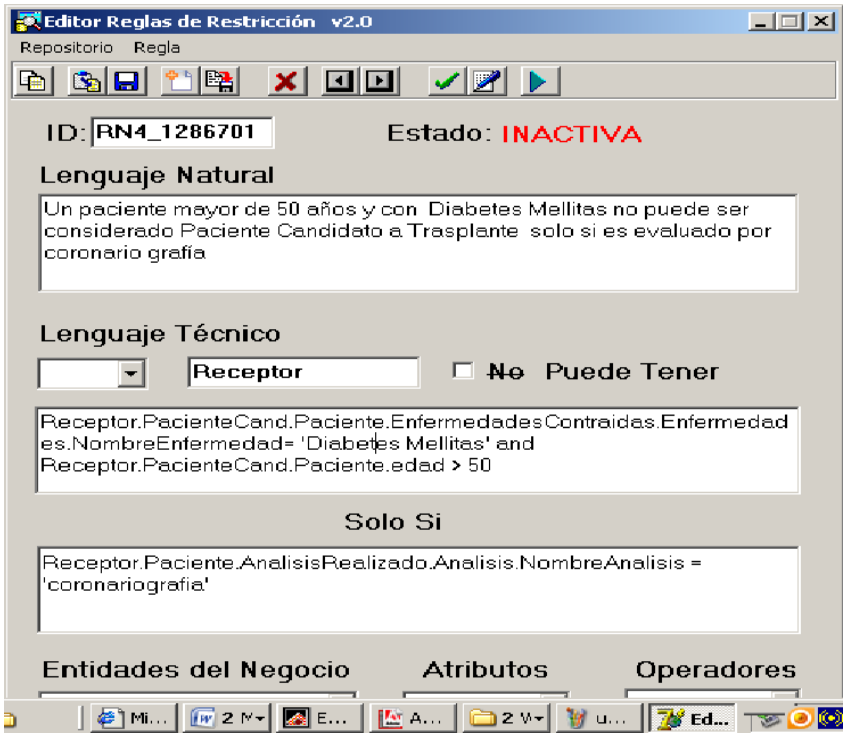
```
function TExtADOConnection.GetRelation(k1, k2: string): TDBRelation;
```

```
var
  i: integer;
  relation: TDBRelation;
begin
  for i := 0 to Length(mDBRelations) - 1 do
    if ((k1 = mDBRelations[i].PKTable.Name) and (k2 = mDBRelations[i].FKTable.Name))
      or ((k2 = mDBRelations[i].PKTable.Name) and (k1 = mDBRelations[i].FKTable.Name)) then
    begin
      relation := mDBRelations[i];
      break;
    end;
  Result := relation
end;
```

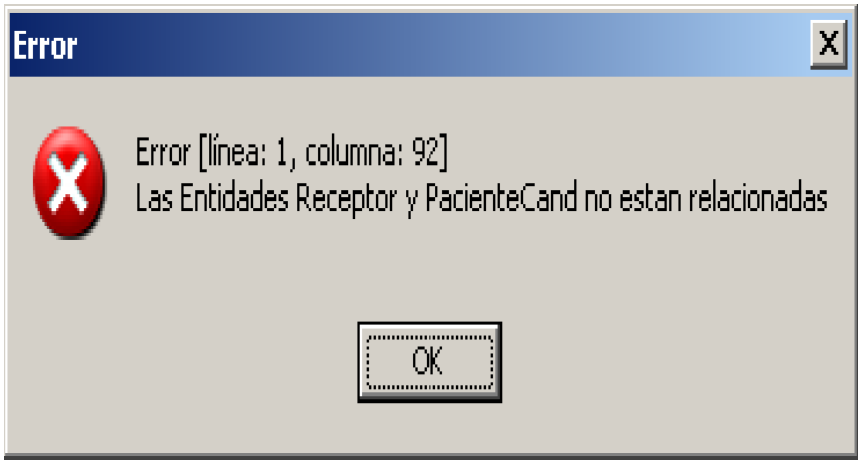
En esta clase se cuenta con un atributo que contiene una lista de DBRelation (mDBRelation) por lo que solo fue necesario buscar un relación que tuviera por llave primaria la llave primaria de la entidad1 y por llave foránea la llave foránea de la entidad2, si esta no existía se hacia la misma búsqueda pero con las entidades intercambiadas.

Para tener una idea del efecto de estas transformaciones se muestra un ejemplo donde que establece una comparación entre la antigua versión y esta, para cuando no existe relación entre las entidades involucradas.

Ejemplo 3.2



Para este ejemplo se viola el lenguaje técnico navegando por dos entidades que no están relacionadas. La respuesta del compilador para este ejemplo es informar que las entidades no se encuentran relacionadas.



Pero se observó en el capítulo 2 que cuando esto ocurría, se generaba un código con puntos vacios, resultado de nunca corroborar que existía relación entre estas tablas.

Chequeo de tipos.

El chequeo de tipos es una parte importante del analizador semántico. No sería posible eliminar la dependencia de consultar la base de datos sin este, pues en la aplicación no se

brinda la información del tipo de los atributos de cada entidad. Por esto fue necesario implementárselo al compilador. Al ser parte del análisis semántico, se mostrará un segmento de cómo se realizó el análisis semántico para las características pues es similar para los hechos, y hubiese sido extenso mostrarlo también cuando se estudio la primera parte del análisis semántico.

En este segmento se aprecia cómo se utiliza *TADOConnection* y las estructuras definidas con anterioridad, para conocer cuando un identificador no es una entidad:

```
{
        DBTable := DBDataModule.ADOConnection.GetTable(szIdentifier);
        if DBTable = nil then
        begin
                Location := 1;
                yyerror(Format('%s no es una Entidad',[szIdentifier]));
                yyerrlab;
                exit;
        end;
        EntList.add(szIdentifier);
}
```

Se muestra cuando no es un atributo de cierta entidad, impidiendo suceda un error al activar la regla:

```
{
        DBTable := DBDataModule.ADOConnection.GetTable(Sem[Top - 1]);

        index := DBTable.FieldsList.IndexOf(Sem[Top]);
        if index < 0 then
        begin
                Location := 2;
                yyerror(Format('%s no es un atributo de %s',[Sem[Top],Sem[Top - 1]]));
                yyerrlab;
                exit;
```

```
        end;

        DBField := TDBField(DBTable.FieldsList.Objects[index]);

        inc(Cab);
        Tipo[Cab] := DBField.Type_Name;

        if EntList.IndexOf(Sem[Top - 1]) < 0 then
                EntList.add(Sem[Top - 1]);

        inc(J);
        Polaca[J] := Sem[Top - 1] + '.' + Sem[Top];
}
```

Una funcionalidad importante como se ha destacado es el comprobar que las entidades involucradas estén relacionadas, es en este el momento que se logra la mayor eficiencia del compilador. Evitándose el constante acceso a la base de datos:

```
{
        DBRelation := DBDataModule.ADOConnection.GetRelation(Sem[Top - 1],Sem[Top - 2]);
        if DBRelation = nil then
        begin
                Location := 2;
                yyerror(Format('Las Entidades %s y %s no estan relacionadas',[Sem[Top - 2],Sem[Top - 1]]));
                yyerrlab;
                exit;
        end;

        if EntList.IndexOf(Sem[Top - 2]) < 0 then
                EntList.add(Sem[Top - 2]);
```

```
        inc(JIndex);
        Joins[JIndex] := DBRelation;

        dec(Top);
}
```

Por último se muestra un segmento de código en el que se comprueba la compatibilidad de los tipos comparados. Aquí eliminamos la posibilidad de fallo de en la Aplicación en el momento de activar la regla:

```
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab - 1],Tipo[Cab]) then
        begin
                Location := 2;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab - 1],Tipo[Cab]]));
                yyerrlab;
                exit;
        end;

        inc(J);
        Polaca[J] := 'LE';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
```

3.4 Conclusiones del capítulo.

De esta forma culmina este capítulo, obteniendo un compilador más fuerte que elimina en mayor grado la dependencia de un acceso constante a la Base de Datos, haciendo la aplicación de más fácil y eficiente uso para el usuario. Quedando validada la aplicación para el Sistema de Información para el área de nefrología del Hospital Provincial "Arnaldo Milián Castro".

Recomendaciones y conclusiones.

Conclusiones.

Para generar automáticamente las Reglas de Negocio, tipo Restricción en la base de datos con que se contaba para trasplante renal:

- Se modificó la Base de Datos, aumentándole entidades, atributos y relaciones.
- Se modificó el Editor en vista de mejorar su diseño para el uso de la aplicación.
- Se modificó el compilador para dar respuestas más certeras a los errores cometidos al usuario. Implementándosele un tratamiento de errores al analizador sintáctico, y haciendo un análisis semántico.
- Se validó la Aplicación de Reglas de Negocio de Restricción, para el Sistema de Información de trasplante renal.

Recomendaciones.
Para investigaciones y trabajos futuros se recomienda:

- Investigar cómo implementar todo tipo de navegación mediante la notación punto.
- Expandir la generación de reglas a otros tipos de clasificación.
- Extender la aplicación de forma que un usuario sea capaz de editar las reglas.
- Investigar sobre el posible empleo en la implementación de restricciones al crear la Base de Datos.
- Investigar cómo generar las reglas para cualquier gestor de base de datos.
- Investigar cómo se le podría aplicar las Reglas de Negocio generadas a datos ya existentes en la Base de Datos.

Bibliografia

Bibliografia.

GONZÁLEZ MENA , D. V. 2008. “Modelación de Reglas de Negocio como apoyo para sistemás de información en el área de nefrología”

LORENZO, I. M. 2009. Modificación de las Reglas de Negocio tipo restricción y su implementación”

MORGAN, T. 2002. Business Rules and Information Systems: Aligning IT with Business Goals.

.

MOTA, S. A. 2005. Bases de Datos Activas.

PEREIRA 2009. Tesis de Grado "Solución al problema de la cardinalidad en la generación automática de Reglas de Negocio en bases de datos relacionales.

".

PÉREZ ALONSO, A. 2008. *Aplicación para reglas de restricción en negocios.* Licenciado Trabajo de Diploma, Universidad Central "Marta Abreu" de Las Villas.

REDÍN, A., LARREA, B., OSÉS, B., LINACERO, M. C., BERANTEGUI, M. & LABAIRU, E. 2006. Informatización del protocolo de cuidados del trasplante renal. *Clínica universitaria de Navarra.Pamplona*.

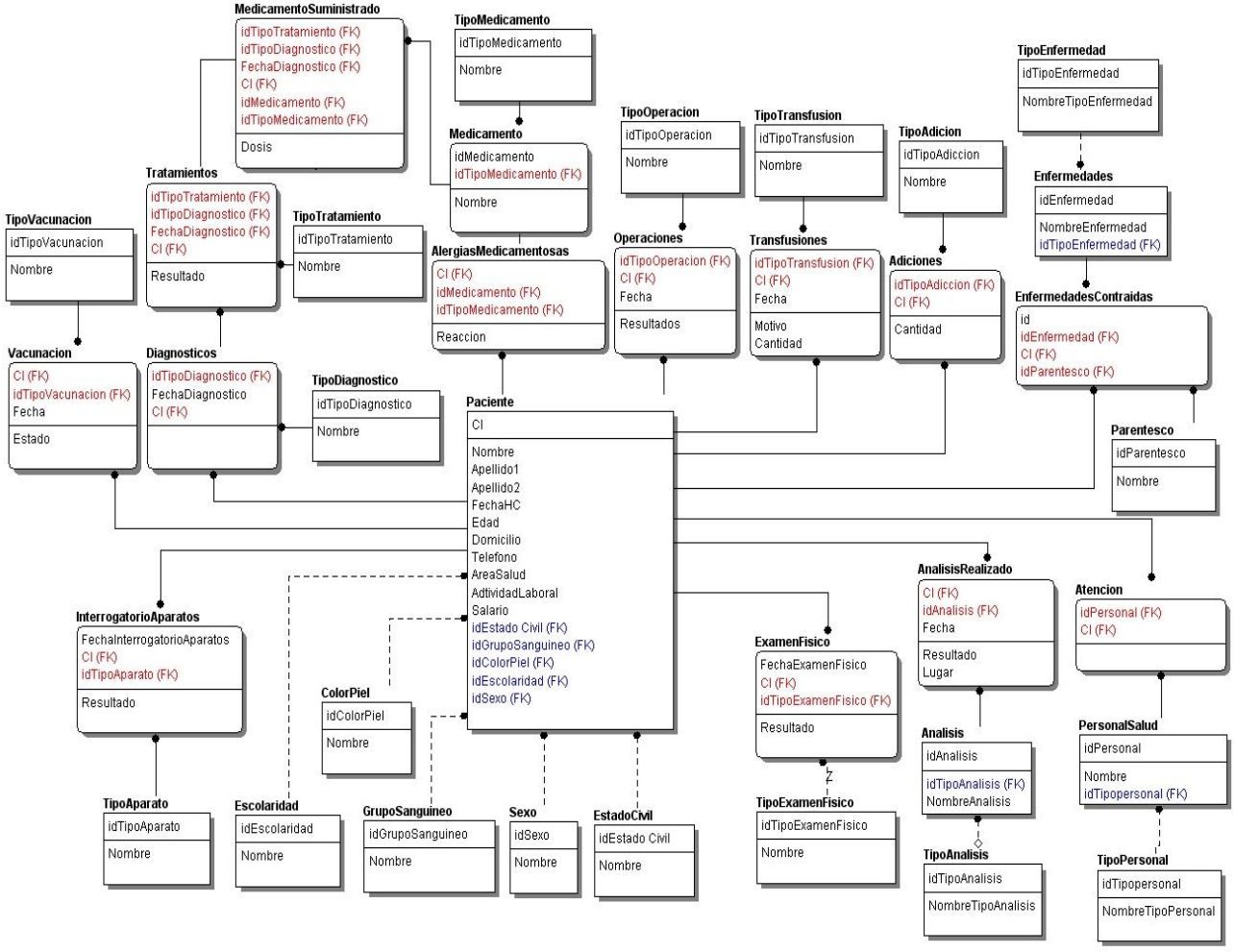
REYES-ACEVEDO, R. 2005. Ética y trasplante de órganos: búsqueda continua de lo que es aceptable. 57, Núm.2.

SOLIVERES, P. A. 2009. Ubicación de las Reglas de Negocio. *Publicado en Revista Profesional para Programadores (RPP)*

SQLSERVER2000, M. 2004. Libros en pantalla de Microsoft SQL Server. 8.0 ed.

Anexos.

Anexo1 Base de Datos de Trasplante Renal



Anexo2 Analizador Semántico

```
%{

uses YaccLib, LexLib, DBManager, Classes;

const
  OPEQU = 10;

var
  Location: integer;

  Sem: array[0..255] of Variant;
  Top: integer;

  Tipo: array[0..255] of Variant;
  Cab: integer;

  Polaca: array[0..255] of Variant;
  J, K: integer;

  Joins: array[0..255] of TDBRelation;
  JIndex, KIndex: integer;

  EntList: TStringList;
  EntList2: TStringList;

  DBTable: TDBTable;
  DBField: TDBField;
  DBRelation: TDBRelation;

  szIdentifier: string;
  index: integer;
%}

%start Regla

%token <String>  TSTRING
%token <Integer> TIDENT
%token <Integer> TNUMBER
%token <Real>    TREALNUM

%token TLPARENT
%token TRPARENT
%token TCOM
%token TPTO
```

```
%token TAND
%token TOR
%token TNOT

%token TLT
%token TGT
%token TLE
%token TGE
%token TEQ
%token TNE

%token TSIZEOF
%token TEXIST
%token TEMPTY

%token TAVG
%token TSUM
%token TMIN
%token TMAX
%token TAD
%token TSOD
%token TSD

%token ILLEGAL                  /* illegal token        */

%left  TGT TLT
%left  TGE TLE TEQ TNE
%left  TAND
%left  TLPAREN TRPAREN

%%
Regla: Entidad_CaracteristicasPrincipal Caracteristicas Hechos
{
}
        ;

Entidad_CaracteristicasPrincipal: TIDENT
{
        DBTable := DBDataModule.ADOConnection.GetTable(szIdentifier);
        if DBTable = nil then
        begin
                Location := 1;
                yyerror(Format('%s no es una Entidad',[szIdentifier]));
                yyerrlab;
                exit;
        end;
```

```
        EntList.add(szIdentifier);
}
  ;

Caracteristicas: {
        Location := 2;
        yyerror('Características vacias');
        yyerrlab;
} ;

Caracteristicas: Exp_Caracteristicas
{
        K := J;
        KIndex := JIndex;
}
        ;

Exp_Caracteristicas: TNOT Exp_Caracteristicas
{
        if Tipo[Cab] <> 'boolean' then
        begin
                Location := 2;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab],'boolean']));
                yyerrlab;
                exit;
        end;

        inc(J);
        Polaca[J] := 'NOT';
}
  | Exp_Caracteristicas TAND Exp_Caracteristicas
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 2;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
                yyerrlab;
                exit;
        end;

        inc(J);
        Polaca[J] := 'AND';

        dec(Cab);
```

```
        Tipo[Cab] := 'boolean';
}
  | Exp_Caracteristicas TOR Exp_Caracteristicas
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 2;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
                yyerrlab;
                exit;
        end;

        inc(J);
        Polaca[J] := 'OR';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
  | Elemento_Caracteristicas TGT Elemento_Caracteristicas
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 2;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
                yyerrlab;
                exit;
        end;

        inc(J);
        Polaca[J] := 'GT';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
  | Elemento_Caracteristicas TLT Elemento_Caracteristicas
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 2;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
```

```
                yyerrlab;
                exit;
        end;

        inc(J);
        Polaca[J] := 'LT';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
  | Elemento_Caracteristicas TGE Elemento_Caracteristicas
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 2;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
                yyerrlab;
                exit;
        end;

        inc(J);
        Polaca[J] := 'GE';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
  | Elemento_Caracteristicas TLE Elemento_Caracteristicas
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 2;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
                yyerrlab;
                exit;
        end;

        inc(J);
        Polaca[J] := 'LE';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
```

```
  | Elemento_Caracteristicas TEQ Elemento_Caracteristicas
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 2;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
                yyerrlab;
                exit;
        end;

        inc(J);
        Polaca[J] := 'EQU';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
  | Elemento_Caracteristicas TNE Elemento_Caracteristicas
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 2;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
                yyerrlab;
                exit;
        end;

        inc(J);
        Polaca[J] := 'NE';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
  | Token_Logico
  | TLPARENT Exp_Caracteristicas TRPARENT
  ;

Token_Logico: TEXISTS TLPAREN Caract_Multiple TRPAREN
{
        Tipo[Cab - 1] := 'boolean';
}
  | TEMPTY TLPAREN Caract_Multiple TRPAREN
{
```

```
        Tipo[Cab - 1] := 'boolean';
}

  ;

Elemento_Caracteristicas: TNUMBER
{
        inc(Cab);
        Tipo[Cab] := 'int';

        inc(J);
        Polaca[J] := szIdentifier;
}
  | TREALNUM
{
        inc(Cab);
        Tipo[Cab] := 'float';

        inc(J);
        Polaca[J] := szIdentifier;
}

  | TSTRING
{
        inc(Cab);
        Tipo[Cab] := 'char';

        inc(J);
        Polaca[J] := szIdentifier;
}

  | Token_Elemental_Caracteristicas
  | Caract_Multiple
  ;

Token_Elemental_Caracteristicas: TSIZEOF TLPARENT Caract_Multiple TRPARENT
{
        inc(J);
        Polaca[J] := 'SIZEOF';
}
        | TEMPTY TLPARENT Caract_Multiple TRPARENT
{
        inc(J);
        Polaca[J] := 'EMPTY';
}
        | TEXIST TLPARENT Caract_Multiple TRPARENT
```

```
{
        inc(J);
        Polaca[J] := 'EXIST';
}
  | TAVG TLPARENT Caract_Multiple TRPARENT
{
        inc(J);
        Polaca[J] := 'AVG';
}
  | TSUM TLPARENT Caract_Multiple TRPARENT
{
        inc(J);
        Polaca[J] := 'SUM';
}
  | TMIN TLPARENT Caract_Multiple TRPARENT
{
        inc(J);
        Polaca[J] := 'MIN';
}
  | TMAX TLPARENT Caract_Multiple TRPARENT
{
        inc(J);
        Polaca[J] := 'MAX';
}
  | TAD TLPARENT Caract_Multiple TRPARENT
{
        inc(J);
        Polaca[J] := 'AVGDIF';
}
  | TSOD TLPARENT Caract_Multiple TRPARENT
{
        inc(J);
        Polaca[J] := 'SIZEOFDIF';
}
  | TSD TLPARENT Caract_Multiple TRPARENT
{
        inc(J);
        Polaca[J] := 'SUMDIF';
}
  ;

Caract_Multiple: Entidad_Caracteristicas TPTO Caract_Multiple
{
        DBRelation := DBDataModule.ADOConnection.GetRelation(Sem[Top -
1],Sem[Top - 2]);
        if DBRelation = nil then
```

```
        begin
                Location := 2;
                yyerror(Format('Las Entidades %s y %s no estan relacionadas',[Sem[Top -
2],Sem[Top - 1]]));
                yyerrlab;
                exit;
        end;

        if EntList.IndexOf(Sem[Top - 2]) < 0 then
                EntList.add(Sem[Top - 2]);

        inc(JIndex);
        Joins[JIndex] := DBRelation;

        dec(Top);
}
  | Entidad_Caracteristicas TPTO Atrib_Multiple_Caracteristicas
{
        DBTable := DBDataModule.ADOConnection.GetTable(Sem[Top - 1]);

        index := DBTable.FieldsList.IndexOf(Sem[Top]);
        if index < 0 then
        begin
                Location := 2;
                yyerror(Format('%s no es un atributo de %s',[Sem[Top],Sem[Top - 1]]));
                yyerrlab;
                exit;
        end;

        DBField := TDBField(DBTable.FieldsList.Objects[index]);

        inc(Cab);
        Tipo[Cab] := DBField.Type_Name;

        if EntList.IndexOf(Sem[Top - 1]) < 0 then
                EntList.add(Sem[Top - 1]);

        inc(J);
        Polaca[J] := Sem[Top - 1] + '.' + Sem[Top];
}
  ;

Atrib_Multiple_Caracteristicas: TIDENT
{
        inc(Top);
        Sem[Top] := Copy(szIdentifier, 1, Length(szIdentifier));
```

```
}
  ;

Entidad_Caracteristicas: TIDENT
{
        DBTable := DBDataModule.ADOConnection.GetTable(szIdentifier);
        if DBTable = nil then
        begin
                Location := 2;
                yyerror(Format('%s no es una Entidad',[szIdentifier]));
                yyerrlab;
                exit;
        end;

        inc(Top);
        Sem[Top] := Copy(szIdentifier, 1, Length(szIdentifier));
}
  ;

Hechos: {
} ;

Hechos: Exp_Hechos;

Exp_Hechos: TNOT Exp_Hechos
{
        if Tipo[Cab] <> 'boolean' then
        begin
                Location := 3;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab],'boolean']));
                yyerrlab;
                exit;
        end;

        inc(K);
        Polaca[K] := 'NOT';
}
  | Exp_Hechos TAND Exp_Hechos
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 3;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
                yyerrlab;
```

```
                exit;
        end;

        inc(K);
        Polaca[K] := 'AND';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
  | Exp_Hechos TOR Exp_Hechos
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 3;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
                yyerrlab;
                exit;
        end;

        inc(K);
        Polaca[K] := 'OR';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
  | Elemento_Hechos TGT Elemento_Hechos
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 3;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
                yyerrlab;
                exit;
        end;

        inc(K);
        Polaca[K] := 'GT';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
  | Elemento_Hechos TLT Elemento_Hechos
```

```
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 3;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
                yyerrlab;
                exit;
        end;

        inc(K);
        Polaca[K] := 'LT';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
  | Elemento_Hechos TGE Elemento_Hechos
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 3;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
                yyerrlab;
                exit;
        end;

        inc(K);
        Polaca[K] := 'GE';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
  | Elemento_Hechos TLE Elemento_Hechos
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 3;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
                yyerrlab;
                exit;
        end;
```

```
        inc(K);
        Polaca[K] := 'LE';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
  | Elemento_Hechos TEQ Elemento_Hechos
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 3;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
                yyerrlab;
                exit;
        end;

        inc(K);
        Polaca[K] := 'EQU';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
  | Elemento_Hechos TNE Elemento_Hechos
{
        if not DBDataModule.ADOConnection.CompatibleTypes(Tipo[Cab -
1],Tipo[Cab]) then
        begin
                Location := 3;
                yyerror(Format('Tipos no compatibles: %s - %s',[Tipo[Cab -
1],Tipo[Cab]]));
                yyerrlab;
                exit;
        end;

        inc(K);
        Polaca[K] := 'NE';

        dec(Cab);
        Tipo[Cab] := 'boolean';
}
  | Token_Logico
  | TLPARENT Exp_Hechos TRPARENT
  ;
```

```
Token_Logico: TEXISTS TLPAREN Caract_Multiple TRPAREN
{
        Tipo[Cab - 1] := 'boolean';
}
  | TEMPTY TLPAREN Caract_Multiple TRPAREN
{
        Tipo[Cab - 1] := 'boolean';
}

  ;

Elemento_Hechos: TNUMBER
{
        inc(Cab);
        Tipo[Cab] := 'int';

        inc(K);
        Polaca[K] := szIdentifier;
}
  | TREALNUM
{
        inc(Cab);
        Tipo[Cab] := 'float';

        inc(K);
        Polaca[K] := szIdentifier;
}

  | TSTRING
{
        inc(Cab);
        Tipo[Cab] := 'char';

        inc(K);
        Polaca[K] := szIdentifier;
}

  | Token_Elemental_Hechos
  | Hechos_Multiple
  ;

Token_Elemental_Hechos: TSIZEOF TLPARENT Hechos_Multiple TRPARENT
{
        inc(K);
        Polaca[K] := 'SIZEOF';
```

```
}
  | TAVG TLPARENT Hechos_Multiple TRPARENT
  | TSUM TLPARENT Hechos_Multiple TRPARENT
  | TMIN TLPARENT Hechos_Multiple TRPARENT
  | TMAX TLPARENT Hechos_Multiple TRPARENT
  | TAD TLPARENT Hechos_Multiple TRPARENT
  | TSOD TLPARENT Hechos_Multiple TRPARENT
  | TSD TLPARENT Hechos_Multiple TRPARENT
  ;

Hechos_Multiple: Entidad_Hechos TPTO Hechos_Multiple
{
        DBRelation := DBDataModule.ADOConnection.GetRelation(Sem[Top -
1],Sem[Top - 2]);
        if DBRelation = nil then
        begin
                Location := 3;
                yyerror(Format('Las Entidades %s y %s no estan relacionadas',[Sem[Top -
2],Sem[Top - 1]]));
                yyerrlab;
                exit;
        end;

        if EntList2.IndexOf(Sem[Top - 2]) < 0 then
                EntList2.add(Sem[Top - 2]);

        inc(KIndex);
        Joins[KIndex] := DBRelation;

        dec(Top);
}
  | Entidad_Hechos TPTO Atrib_Multiple_Hechos
{
        DBTable := DBDataModule.ADOConnection.GetTable(Sem[Top - 1]);

        index := DBTable.FieldsList.IndexOf(Sem[Top]);
        if index < 0 then
        begin
                Location := 3;
                yyerror(Format('%s no es un atributo de %s',[Sem[Top],Sem[Top - 1]]));
                yyerrlab;
                exit;
        end;

        DBField := TDBField(DBTable.FieldsList.Objects[index]);
```

```
        inc(Cab);
        Tipo[Cab] := DBField.Type_Name;

        if EntList2.IndexOf(Sem[Top - 1]) < 0 then
                EntList2.add(Sem[Top - 1]);

        inc(K);
        Polaca[K] := Sem[Top - 1] + '.' + Sem[Top];
}
  ;

Atrib_Multiple_Hechos: TIDENT
{
        inc(Top);
        Sem[Top] := Copy(szIdentifier, 1, Length(szIdentifier));
}
  ;

Entidad_Hechos: TIDENT
{
        DBTable := DBDataModule.ADOConnection.GetTable(szIdentifier);
        if DBTable = nil then
        begin
                Location := 3;
                yyerror(Format('%s no es una Entidad',[szIdentifier]));
                yyerrlab;
                exit;
        end;

        inc(Top);
        Sem[Top] := Copy(szIdentifier, 1, Length(szIdentifier));
}
  ;
%%

{$I LEXER}

end.
```