TESIS CON
FALLAS DE ORIGEN

2.30

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

ESTUDIO DE DIVERSOS METODOS DE ESPECIFICACION FORMAL DE PROGRAMAS

TESIS PROFESIONAL

QUE PARA OBTENER EL TITULO DE:

A C T U A R I O

P R E S E N T A :

GUSTAVO ARTURO MARQUEZ FLORES

México, D. F. 1988

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

DERECHOS RESERVADOS © PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis está protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

2, 30

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

ESTUDIO DE DIVERSOS METODOS DE ESPECIFICACION FORMAL DE PROGRAMAS

TESIS PROFESIONAL

QUE PARA OBTENER EL TITULO DE:

A C T U A R I O

P R E S E N T A :

GUSTAVO ARTURO MARQUEZ FLORES

México, D. F. 1988

C O N T E N I D O

Contenido i

Introducción. 1

Capitulo 1. La Dificultad para Probar y Especificar Programas. 3

1.1 La Creciente Necesidad de Probar y Especificar Programas y las Primeras Investigaciones y Trabajos. 3

Capitulo 2. El Lenguaje de Especificación Formal Larch. 11

2.1 Generalidades. 11

2.1.1 Objetivo de los Lenguajes Larch. 11

2.1.2 Filosofía de los Lenguajes Larch. 12

2.1.3 Características de los Lenguajes Larch. 13

2.2 Descripción del Lenguaje Larch. 14

2.2.1 El lenguaje Larch Compartido. 14

2.2.2 El lenguaje Larch de Interfaz. 23

2.3 Aplicando el lenguaje Larch. 28

Capitulo 3. El lenguaje de Especificación Formal OBJ. 41

3.1 Generalidades. 41

3.2 Descripción del Lenguaje. 42

3.2.1 Tipos de Funciones en OBJ. 43

3.2.2 Variables y Ecuaciones en OBJ. 46

3.3 Algunas Características Relevantes de OBJ. 49

3.4 Implantación de OBJ. 55

3.5 Aplicando el lenguaje OBJ. 57

Capítulo 4. Desarrollo e Implantación de un Ejemplo Aplicando el Lenguaje Larch. 65

4.1 Especificando un Ejemplo. 65

4.1.1 Edición del Archivo de Nombres de Grupos. 68

4.1.2 Edición del Archivo de Alumnos. 70

4.1.3 Especificación de la Característica Nombres y del TDI Cadena. 87

4.2 Implantando la Especificación. 89

Conclusiones. 93

Apéndice A. Especificación de una Relación de Orden Total en el Lenguaje Larch Compartido. 95

Apéndice B. Especificación Completa del Ejemplo de un Directorio en el Lenguaje Larch Compartido y de Interfaz. 96

Apéndice C. Especificación de la Característica Archivo-Alumnos en el Lenguaje Larch Compartido y de Interfaz. 101

Apéndice D. Implantación en el Lenguaje Pascal del Ejemplo Desarrollado en al Capítulo 4. 108

Apéndice E. Sintaxis del Lenguaje Larch Compartido. 117

Apéndice F. Sintaxis del Lenguaje OBJ. 119

Bibliografía. 121

I N T R O D U C C I O N.

A lo largo de los últimos años, la programación ha sido objeto de un importante trabajo de investigación. Universalmente se ha admitido que la programación es una actividad importante que implica un trabajo riguroso, con la aplicación de métodos y técnicas sofisticadas, así como de herramientas eficientes. Esta actitud ha dado origen a diversas áreas de investigación dentro de la programación, como la de los Lenguajes de Programación, la Validación y Prueba de Programas, la Teoría y Metodología de la Programación, la Especificación de Programas así como el empleo de Estructuras de Datos y Tipos de datos Abstractos.

Esencialmente, se puede considerar la programación como el paso del enunciado de un problema, a un programa de computadora escrito en un cierto lenguaje de programación. El punto de partida de la escritura de este programa, es un conjunto de enunciados que expresan relaciones entre un conjunto de datos y uno o varios resultados esperados. Este conjunto de enunciados constituye la especificación del programa, la cual muestra en esencia lo que va a hacer el programa o sistema, pero no como lo va a hacer. Una especificación, básicamente debe ser precisa, sin ambiguedad y de tal manera que no origine malentendidos ni malas interpretaciones en los requerimientos de un programa.

El escribir los requerimientos de un programa a través de especificaciones informales, en lenguaje natural por ejemplo, tiene las ventajas de poder organizar mejor las ideas del problema a resolver, ya que es un lenguaje simple de entender y de manejar.

Los tradicionales métodos de diseño de sistemas aplicados en la industria e institutos de investigación, han logrado algunos avances al incluir la especificación informal en el desarrollo de programas durante algún tiempo.

La desventaja que tiene la aplicación de especificaciones informales, es que muchas veces es imprecisa y vaga, como el lenguaje natural, pudiendo ocasionar ambigüedades en la especificación de un programa.

A diferencia de la especificación informal, una especificación formal está escrita en un lenguaje con sintaxis y semántica precisa y definida. Por consiguiente, una particularidad de una especificación formal es su precisión y exactitud. Además, tiene las ventajas de poder justificar decisiones llevadas a cabo en el diseño. e intentar ir probando la validación y correctividad del

programa. Sin embargo es más difícil y complicada de entender.

Desde hace algunos años, las investigaciones acerca de métodos de especificación de programas, así como la especificación formal, han adquirido un carácter significativo e importante. Estas investigaciones se han centrado especialmente en la presentación y evaluación de una clase de lenguajes muy particulares para especificar programas. Un lenguaje de especificación es un conjunto de expresiones muy especiales que incluye sintaxis y semánticas propias para presentar y precisar especificaciones de programas.

El interés que condujo a la realización de estas investigaciones, fue la necesidad de introducir un formalismo de carácter matemático desde la descripción de las demandas y solicitudes de los usuarios de programas y sistemas (como se verá en el primer capítulo), así como sentar las bases para permitir la generación automática de programas.

Algunos de los lenguajes de especificación formal más conocidos son: CLEAR [Burstall y Goguen, 77, Burstall y Goguen, 81], Iota [Nakajima et al., 80], Act-one [Ehrig y Mahr, 85], Special [Robinson y Roubine, 77], OBJ [Goguen y Tardo, 77], Z [Abrial, 80], VDM'SMetal-iv [Bjorner y Jones, 78], Inajo [Scheid y Anderson, 85], Gypsy [Good et al., 78] y la Familia de Lenguajes de Especificación Formal Larch [Guttag et al., 85].

La finalidad de esta tesis es señalar la importancia que ha tenido la especificación y prueba de programas, así como mostrar algunos de los trabajos e investigaciones importantes realizados para resolverlos. También se revisará la constante necesidad de especificar y probar programas, expresada por científicos e investigadores desde poco después de los inicios de la computación, debido a la deficiencia e inseguridad de los sistemas y programas que se venían desarrollando, y que aún hasta en nuestros días siguen desarrollándose así, debido en parte a la poca difusión que han tenido los métodos y formalismos de especificación.

Con el objeto de introducir y difundir los métodos formales para la especificación de programas y sistemas, y en especial los lenguajes de especificación formal, en seguida se presentan y se estudian la familia de lenguajes de especificación formal Larch así como el lenguaje de especificación formal OBJ. Por último, se presenta un ejemplo de una especificación e implantación en el lenguaje de programación Pascal, aplicando alguno de estos lenguajes de especificación.

C A P I T U L O   P R I M E R O.

La Dificultad Para Probar Y Especificar Programas.

Desde el surgimiento de la computación como una actividad exclusivamente científica, hace ya aproximadamente 35 años, científicos e investigadores imaginaron el enorme desarrollo y aplicación que tendría en las áreas de la ciencia, la investigación y en algunas otras áreas de aplicación. Sospechaban que esta nueva ciencia reportaría grandes avances en estas áreas por la rapidez y exactitud con que se podrían resolver problemas numéricos, estadísticos, administrativos, de proceso de información y algunos otros realizados muy frecuentemente en éstas y otras áreas.

Sin embargo, después de aproximadamente 10 años de haber surgido, empezaron a aparecer serias complicaciones en esta nueva ciencia, que tuvieron como consecuencia obstaculizar el desarrollo de la computación. Estas complicaciones se concentraron muy especialmente en dos áreas: la dificultad para probar programas y la dificultad para especificarlos, las cuales estan muy estrechamente relacionadas.

A continuación, se describen algunos aspectos y antecedentes importantes que se han aportado para intentar resolver estos problemas, así como la creciente necesidad histórica para resolverlos.

1.1 La Necesidad de Probar y Especificar Programas y las Primeras Investigaciones y Trabajos.

La necesidad de probar y especificar programas correctamente, surge a consecuencia de la deficiente manera de programar, diseñar e implantar programas y sistemas.

En su inicio, al principio de la década de los 50's, las computadoras fueron enormes máquinas que ocupaban cuartos o salas enteras por sus dimensiones, lo que les hacía parecer laboratorios. Eran difíciles de manejar y requerían de personas muy especializadas para su operación. Además, la lentitud y poca precisión de estas máquinas hacía a los programadores de aquel tiempo enfrentarse a verdaderos problemas. Ellos tenían que realizar sorprendentes malabarismos y trucos, así como tener una mente hábil y astuta para resolver problemas simples de programación.
Además, la programación en este tiempo era considerada como una actividad de optimización eficiente de recursos (por el tamaño limitado de la memoria de estas computadoras y de otros recursos) más que de aplicación de téncnicas propias de programación, como

la estructuración de programas, el planteamiento de casos de prueba, el empleo de métodos de especificación, etc.

Era evidente que la situación anterior conduciría a la construcción de programas y sistemas inseguros, de enorme tamaño, difíciles de probar y que en muchas ocasiones no cumplían con la especificación establecida, dado que ni siquiera se tenía claro lo que se quería. Todo esto debido a que aún no se promovía el desarrollo de la computación como ciencia.

Pero la situación de aquel tiempo se empeoró poco después aún más, con el surgimiento de las nuevas computadoras de la llamada segunda generación, que surgió a principios de la década de los 60's, pues estas máquinas eran aún más complicadas de operar. Por una parte por los nuevos implementos y avances tecnológicos logrados en esta época, como la sustitución de los tubos de vacío por transistores, la invención de cintas magnéticas, la construcción de sistemas operativos, la construcción de lenguajes de programación más fáciles que el tradicional y complicado lenguaje ensamblador y la existencia de bibliotecas, entre otros avances.
Y por otra parte, por la inexperiencia de los programadores en su manejo.

Pero quizá la causa principal que empeoró esta situación fue que estas nuevas máquinas se volvieron aún más poderosas en la solución de problemas numéricos y de cálculo, pudiéndose ahora resolver programas y sistemas más complejos.
Y fue entonces, que la ambición de la sociedad y la ciencia creció al querer extender las aplicaciones de las computadoras a muy diversas áreas, lo cual dió como consecuencia más dificultad para el programador al enfrentarse a nuevos problemas, como el tener que saber administrar internamente la memoria de la computadora, el tener que saber también un lenguaje más poderoso de programación (que básicamente en este tiempo era Fortran, Cobol y Algol) y así como el conocimiento y manejo de sistemas operativos y compiladores.
Sin embargo, no hay que negar que estas máquinas lograron incrementar la confianza y seguridad en los programas y sistemas, obteniendo soluciones más factibles y confiables.

Otra de las razones importantes que originó esta situación era que por un lado el usuario y el programador tienen formas distintas de hablar y comunicarse, ya que el usuario plantea su problema al programador en lenguaje natural y el programador intenta traducir estos requerimientos en un lenguaje más sofisticado que es el de programación, el cual es difícil de entender para el usuario. Surgen así conflictos y ambigüedades en lo que se quiere que haga el programa o sistema, pués el lenguaje que utiliza el usuario es vago e informal, ocasionando que el programador entendiera poco su problema y desarrollara entonces programas no útiles y poco eficientes.

Se requiere entonces un 'lenguaje' o un método tal que: 1) usara como herramienta a las matemáticas, la lógica, el álgebra, etc. (o sea que fuera formal, preciso y no ambiguo) para especificar

programas y sistemas, 2) no fuera tan complicado como los lenguajes de programación, 3) pudiera representar de manera expresiva y analítica las características y propiedades de los requerimientos de los usuarios, 4) su principal objetivo fuera contestar el "qué" debe hacer un programa y no el "cómo" lo va a hacer y 5) fuera además entendido por ambos.

Por otra parte, una véz que se lograba terminar de diseñar un programa y se implantaba, no había forma de probar que el programa hiciera lo que se supone debiera de hacer y que no hiciera lo que no debiera de hacer, así como de saber si los resultados eran correctos o no.

Algunos de los métodos y técnicas que se usaban y que actualmente siguen usándose para probar un programa, es decir, demostrar que cumpliera con las características antes establecidas, son informales. Como la de releer varias veces detenidamente el programa, la de probar con algunos casos de prueba el programa y la de depurar el programa a través de "parches" de otros programas. Estos métodos dependen mucho del ingenio e intuición del programador y se ha visto que aún después de aplicar estas técnicas, es posible encontrar errores, lo que comprueba que son inadecuados e ineficientes.

Lo que hacía falta y se requería en aquel tiempo y en consecuencia actualmente, es un método formal tal que pudiera garantizar al momento de escribir los requerimientos y especificaciones de programas, que el programa o sistema que genere este conjunto de expresiones u oraciones sea correcto aun antes de implantarlo y que no exista ambigüedad en él.

Pero a las causas anteriores que originaron lo que algunos científicos e investigadores llamaron más tarde la "crisis del software", habría que agregarle el hecho que la computación era considerada una actividad "artística", más que como ciencia o rama de investigación, porque se consideraba innecesario, la aplicación de técnicas matemáticas así como del formalismo y rigor matemático que rige a muchas ciencias.

En 1965 en un congreso llevado a cabo por la IFIP (International Federation for Information Processing) en Inglaterra por Stanleys Gill, se anuncia el primer problema al que se enfrentarían los investigadores y científicos de este tiempo y que ofrecía pocas perspectivas de solución inmediata: la corrección y prueba de programas. Hubo algunos investigadores teóricos que atacaron este problema reduciéndolo a probar formalmente la equivalencia de dos diferentes programas, resultando inútil desde un punto de vista práctico.

En 1968 se llevó a cabo una conferencia sobre ingeniería de software en Garmisch, Alemania, para tratar de hallar una solución a la crisis del software, donde se reunieron investigadores, profesores y gente de la industria. La conclusión a que se llegó fue que, por primera véz se reconoció formalmente que existía realmente una crisis del software: la programación estaba entonces por

un rumbo equivocado.

Una de las personas que sugirió importantes opiniones y aspectos de la computación, con el propósito de mejorar esta situación fue Edsger W. Dijkstra, quien analizó la situación de la computación en esta década en 1972, [Dijkstra, 72] y obtuvo como conclusión importantes consideraciones y argumentos que deben tomarse en cuenta actualmente, para la especificación y prueba de programas, así como de la computación en general para su correcto desarrollo.
Una de las consideraciones que hace, establece reconocer la situación de la época, la necesidad de una mayor seguridad y confiabilidad en los programas y sistemas, para saber así donde se requieren soluciones inmediatas y factibles.
Otra consideración importante se refiere a la manera de programar, la cual repercute en la economía; es decir, según él, en los años siguientes, los precios del hardware bajarán considerablemente (como sucede actualmente) y si los sistemas desarrolados siguen siendo costosos e inseguros, la situación se hallaría fuera de balance.
Por lo tanto es preciso desarrollar programas más eficientes, acordes con el nivel del hardware.
Por otra parte, en uno de sus argumentos él sugiere seguir un método para probar un programa, que difiere de los usados hasta ese momento, en que éstos sólo muestran la presencia de errores, pero desafortunadamente inadecuados para mostrar su ausencia. Este método consiste primero en preguntarse por qué construcción de una prueba convincente optar y habiéndola encontrado, entonces construir el programa satisfaciendo los requerimientos de la prueba.
Sin embargo, esta tendencia para probar programas, es sólo aplicable cuando nos hallemos restringido a lo que el llama "programas intelectualmente manejables", proporcionando aún así con esta limitante, los medios efectivos para encontrar una prueba contundente de manera satisfactoria, entre otras muchas posibles.

El restringirse a diseñar e implantar programas intelectualmente manejables, constituye otra de sus consideraciones. Estos programas tienen como característica cumplir con dos clases de reglas.
El primer tipo de reglas son aquéllas que se imponen de manera mecánica. Por ejemplo, del lenguaje de programación, la exclusión de la proposición "go-to", el eludir procedimientos con más de un parámetro de salida, etc.
Y la segunda clase de reglas son aquéllas impuestas en la disciplina de un programador. Por ejemplo, estipular las condiciones necesarias para evitar un ciclo sin control, establecer una estructuración correcta de un programa, etc.
Esta restricción llevaría a solucionar problemas capaces de plantearse algorítmicamente e implantarse fácilmente en un programa para hallar la solución correcta del problema.

Con el fin de incrementar la confianza y seguridad de los programas y sistemas, así como iniciar una nueva área de investigación dentro de la computación orientada a la corrección y prueba de programas, y a la construcción de modelos y métodos matemáticos de especificación formal de programas, en esta década se empe

zaron a realizar lo que serían las primeras investigaciones y trabajos realizados en estas áreas y que vale la pena mencionar algunos de ellos por la importancia que han tenido posteriormente.

Una de las primeras ideas aportadas a la corrección y prueba de programas, fue un artículo publicado en 1961 y más tarde en 1962 en un congreso de la IFIP por John McCarthy [McCarthy, 61]. En él establecía que en lugar de probar programas con casos de prueba hasta depurarlos, debería probarse mejor que cumplieran con las propiedades solicitadas.

En esta década se publicaron otros tres trabajos importantes que tuvieron una profunda importancia en estas áreas.

El primer trabajo fue debido a Peter Naur publicado en el año de 1966 en la Universidad de Copenhagen. El insistió en la importancia de probar programas y propuso una técnica informal para especificarlos [Naur, 66].

El segundo trabajo, realizado por Robert Floyd y presentado en un evento de la American Matematical Society en el año de 1967, colocó una pieza importante en esta área [Floyd, 67]. Floyd planteó la manera de probar programas colocando aseveraciones en un diagrama de flujo de programa y conforme se iba recorriendo éste, las aseveraciones deberían ser concluyentes y ciertas, de manera que llevaran a la ejecución correcta del programa.
Así para un ciclo del diagrama, al colocar una aseveración cierta al principio si al recorrerlo se alcanza de nuevo este punto y continúa siendo cierta esta aseveración, el ciclo es correcto.
Esta idea dió origen al 'Invariante del Ciclo'. También sugirió que un conjunto de técnicas de prueba podría proporcionar una definición de un lenguaje de programación.

El tercer trabajo correspondió a C. A. R. Hoare de la Universidad de Belfast, quien tomó la sugeriencia de Floyd como parte central de su artículo y definió un pequeño lenguaje en términos de un sistema lógico de axiomas y reglas de inferencia, para probar parcialmente programas [Hoare, 69]. Con esto, intentó mostrar cómo definiendo un lenguaje de programación, en términos de cómo probar un programa, en lugar de cómo lo ejecuta, se puede guiar a diseños más simples, pretendiendo con esto una definición axiomática del lenguaje.

Durante la década de los 70's, hubo también varios trabajos importantes que destacan en el área de prueba y corrección de programas, los cuales se mencionan a continuación.

A partir de 1972, un artículo de Hoare sobre pruebas de corrección de representaciones de datos, estimuló la investigación de tipos de datos abstractos [Hoare, 72]. Las especificaciones algebraicas de tipos de datos aparecieron más tarde en un trabajo publicado en 1974 [Liskov y Zilles, 74], y posteriormente en la referencia [Guttag y Horning, 78].
En 1973 ciertas reglas de prueba fueron escritas para la mayor parte del lenguaje Pascal [Hoare y Wirth, 73].
En 1975 se desarrolla un sistema automático de verificación para el lenguaje Pascal, basado en axiomas y reglas de inferencia [Igarashi et al., 75].

Sin embargo, a pesar de las investigaciones, trabajos y esfuerzos realizados hasta ahora para resolver el problema de probar y especificar programas, aún no parece tan fácil hacerlo, pues la teoría que presentaron los trabajos e investigaciones anteriores era difícil de entender y manejar, por ejemplo, aún no se sabía cómo funcionaba y se implantaba un invariante del ciclo. Algunos investigadores creían que la única manera de probar un programa formalmente era a través del uso de un probador de teoremas o verificador. Otros arguían que las pruebas mecánicas eran inútiles, por la complejidad y detalle que requerían.

Pero en 1975, cuando Edsger W. Dijkstra publicó un artículo en el que introdujo ciertas proposiciones para un pequeño lenguaje, publicado en 1962 [Dijkstra, 62], en el que mostraba cómo pueden ser usados como 'cálculo para la derivación de programas', se empezó a clarificar la manera de desarrollar un invariante del ciclo.
Se aclaró también que el empleo de teorías y formalismos matemáticos, acompañados de un sentido común, podrían guiar el desarrollo de programas más confiables y fáciles. Empezaron así a emerger entonces conceptos y principios que daban a la programación un carácter de ciencia bien definida.

En esta década investigadores y profesores, tomando en cuenta que es lo que se requería para solucionar el problema de especificación y prueba de programas y sistemas, conceptualizaron el uso de modelos, a través de los cuales se podían establecer de manera precisa y sin ambigüedad, las especificaciones y demandas de los usuarios y que podían ser fácilmente comprensible por el programador, así como diseñar programas y sistemas ya probados y que cumpliera con aquellas propiedades aun antes de construirlos e implantarlos.
Concluyeron que para lograr esto, el modelo debería de cumplir al menos con las siguientes características:

a) Abstracción: implica remover aquellos detalles insignificantes de la especificación, a fin de manipular sólo aquellas propiedades y características importantes;

b) Representación: requiere que el modelo en sí tenga un poder expresivo, por ejemplo gráfico, esquemático, simbólico, etc. y analítico, a fin de que pueda manejarse fácilmente y de manera casi natural y

c) Manipulación: implica poder manipular y operar sus representaciones, cualquiera que fuera.

Para poder darle un carácter formal a este modelo, se requirió de un sistema formal como respaldo. De esta manera se formalizaba ya ahora el concepto de especificación, que debería de tomarse en cuenta en adelante y que consiste en un conjunto S de proposiciones de L, el lenguaje definido por el sistema formal, o sea $S \subseteq L$.
De acuerdo al lenguaje definido por el sistema formal, que puede

ser algebraico, lógico, funcional, natural, etc., surgen diversos lenguajes de especificación, los cuales se pueden utilizar de acuerdo a las necesidades y naturaleza requerida por el problema del usuario.
Esto trajo como consecuencia la introducción de nuevos formalismos para la especificación y prueba de programas. Al parecer se vislumbraba ahora una nueva solución al problema de la especificación y prueba de programas.

Por último, para terminar con esta exposición, Hoare ha sugerido recientemente mayor disciplina matemática en estas áreas y en general en la computación misma. Para él, la computación es una actividad completamente matemática. La aplicación de las matemáticas sería desde la aplicación de pruebas basadas en la lógica formal, hasta el uso de axiomas algebraicos.
Como ejemplo de esta aplicación, muestra cómo a través de un conjunto de axiomas construidos por él, los programadores podrían verificar y contruir programas más eficientes, pudiéndose también especificar más fácilmente. Muestra también cómo las reglas de la programación son tán sencillas, obvias y útiles como los axiomas de la aritmética.
El define lo que serían los operadores y operandos en su conjunto de axiomas. Los operadores pueden constituir símbolos y expresar acciones o condiciones a realizar con los operandos, los cuales pueden ser comandos, programas o especificaciones de programas.
De esta manera sería muy fácil precisar y cambiar la especificación de un programa, porque la estructura de éste sería muy similar a la estructura de la especificación.
En general, según él, la incursión de las matemáticas en la computación abarcaría entre otras áreas las siguientes:

- En el área de especificación de programas, los manuales y guias que estén basados bajo un criterio o contexto matemático, estarán mejor estructurados, más completos y entendibles, garantizando así la efectividad y confiabilidad del software.

- Con respecto a los sistemas de software, éstos podrán adquirirse y usarse aún con un menor número de errores y correcciones, si éstos están construidos con técnicas basadas en teoremas o axiomas matemáticos que garanticen, aun antes de probarlos, que alcanzan la especificación señalada de manera eficaz, ahorrando con esto mucho dinero y tiempo.

- Por otro lado, la seguridad y confianza de los programas y sistemas en el futuro se hará estricta y tenderá a incrementarse conforme aumentan las mùtiples aplicaciones de éstos. Los métodos matemáticos ofrecen una mejor esperanza para ampliar el control, inspección o revisión de los sistemas de soft-

ware, que las tradicionales técnicas rigurosas de la ingeniería sobre análisis y control de sistemas.

En los siguientes capítulos se presentarán dos lenguajes de especificación formal, llamados Larch y OBJ. Ambos son lenguajes de tipo algebraico, pues el lenguaje definido por el sistema formal que los respalda es algebraico. Esto significa que en este tipo de lenguajes la manera de especificar un tipo de datos abstracto es definiendo las operaciones que se pueden efectuar en él a través de funciones, dando tanto el dominio y rango de cada una de ellas (especificación sintáctica), así como un conjunto de axiomas algebraicos que determinan, en forma precisa, la semántica de cada función.

C A P I T U L O   S E G U N D O.

EL Lenguaje de Especificación Formal Larch.

2.1 Generalidades.

Como parte del proyecto Larch, surgido a mediados de la década de los años 70's y encabezado por John V. Guttag, James J. Horning y por Jeannette M. Wing, surge una metodología de especificación de programas, que se compone básicamente de una familia de lenguajes de especificación formal llamada Larch. Estos lenguajes de especificación tienen entre otros objetivos, el facilitar la escritura de especificaciones algebraicas y hacer que éstas sean más fáciles de leer y entender. El proyecto Larch es la continuación formal de investigaciones llevadas a cabo por J. Guttag en 1975 [Guttag, 75]. El proyecto tiene la finalidad de aplicar y desarrollar herramientas y técnicas para la especificación formal de programas en los próximos años. Actualmente se está desarrollando una familia de lenguajes de especificación y un conjunto de herramientas para apoyar su uso, incluyendo editores sensibles al lenguaje y examinadores de semántica basados en un eficaz demostrador de teoremas.

2.1.1 Objetivos de los Lenguajes Larch.

Para llevar a cabo la construcción y diseño de la familia de lenguajes Larch, se tomaron en cuenta algunas consideraciones y necesidades que más tarde influirían en él y que le darían características y objetivos definidos. Estas fueron:

- La dificultad de especificar grandes sistemas. Algunos métodos de especificación son adecuados para especificaciones pequeñas, pero pueden fallar totalmente para especificaciones más grandes. Por ello, estas últimas deben ser construidas a partir de pequeñas especificaciones que puedan ser entendidas separadamente.

- El problema de las especificaciones incompletas. En ocasiones, las especificaciones incompletas reflejan descuidos de detalles o aspectos irrelevantes para algun fin particular de la misma, como son las especificaciones de uso del sistema, que pueden ser indicadas separadamente. Pero algunas veces reflejan la clara intención de retrazar decisiones en la especificación u omitir importantes aspectos en el diseño de

la especificación. Por esto los métodos de especificación deben poder detectar estas omisiones, sin afectar a aquellos aspectos irrelevantes en esta etapa de especificación.

- La necesidad de bibliotecas de especificaciones. Es impráctico volver a escribir especificaciones que alguna vez afanozamente se escribieron. De aquí que sea necesario tener un compendio de especificaciones comúnes que puedan servir como modelos o guias cuando se enfrenta a situaciones similares, como son las especificaciones para datos abstractos, tales como Colas, Conjuntos, Pilas (Stacks), Conjuntos, Arboles, etc.

- La corrección automática de errores en las especificaciones. En el proceso de escritura de especificaciones se pueden cometer tantos errores como en el proceso de programación. Por esto, las especificaciones formales deben ellas mismas a ayudar lo más posible a corregir estos errores al momento de irse escribiendo. También se desea que los lenguajes de especificación incorporen procedimientos o instrucciones que permitan verificar automáticamente las especificaciones, para detectar así los errores más comúnes en cada paso de la especificación.

2.1.2 Filosofía de los Lenguajes Larch.

Los lenguajes de especificación Larch tienen un estilo particular. El principio de este estilo, es la separación de la especificación en dos partes, donde cada una constituye un lenguaje de especificación a su vez. Una parte, formada por los lenguajes Larch de Interfaz, sirve para la especificación de programas y sistemas en un lenguaje particular de programación. Mientras que la otra parte, el lenguaje Larch Compartido, construye una especificación más abstracta y común a cualquier lenguaje de programación.

En general, una especificación describe las ideas y aspectos relevantes de lo que se va a especificar. El lenguaje Larch Compartido proporciona un conjunto de cláusulas, que son útiles para describir o abstraer las propiedades y características de datos abstractos, a través de funciones y axiomas algebraicos. Los lenguajes Larch de Interfaz utilizan esta representación simbólica del dato abstracto para describir cómo estas propiedades y características pueden ser implantadas en un lenguaje de programación.
La razón por la cual se sugirió la separación del lenguaje de

especificación Larch en dos partes fue la siguiente. Por un lado, era común encontrarse con especificaciones que requerían de construcciones tales como llamados secuenciales, excepciones, disposiciones de entrada y salida, etc., es decir, un lenguaje de especificación debería de estar estrechamente relacionado con el lenguaje de programación, de lo cual surgió la idea de construir los lenguajes Larch de interfaz. Y por otro lado, también era común encontrarse con estructuras abstractas, tales como conjuntos, tablas, colas, pilas, etc., las cuales resultaban ser útiles en la construcción de varios tipos de especificaciones y que fueran independientes del lenguaje de programación y que se podían compartir en muchas aplicaciones en las que se usaran, de donde surgió como consecuencia la idea de diseñar al lenguaje Larch Compartido.

El motivo que sugirió utilizar el termino "Interfaz", fue que estos lenguajes de especificación se encargan de describir el comportamiento de un dato abstracto, según el lenguaje de programación en concreto donde se va a implantar. Mientras que el término "Compartido", fue elegido porque todos los lenguajes Larch de Interfaz comparten este mismo lenguaje para describir a aquellos datos abstractos, (sin importar el lenguaje de programación elegido).

2.1.3 Características de los Lenguajes Larch.

Como se mencionó anteriormente, la familia de lenguajes Larch se divide en dos lenguajes de especificación: en el lenguaje Larch Compartido, el cual engloba un conjunto de cláusulas generales para especificar datos abstractos y el conjunto de lenguajes Larch de Interfaz, los cuales establecen también cláusulas para la especificación de estos datos abstractos en un lenguaje de programación particular.

A continuación se mencionan algunos aspectos y características de ambos lenguajes.

- Permiten la modularidad entre ellos. Ambos lenguajes están diseñados para permitir la construcción de especificaciones provenientes de otras especificaciones.

- Acentúan la presentación de la especificación. Los lenguajes Larch por su construcción son legibles. Además, los mecanismos de composición de Larch son definidos como operaciones sobre las especificaciones más que sobre teorías o modelos.

- Integran herramientas con influencia recíproca. Los lenguajes Larch están diseñados para facilitar la construcción de especificaciones con influencia o afectación sobre otra u o-

tras especificaciones.

- Permiten una verificación automática de la especificación. Los lenguajes Larch están construidos para permitir una verificación de la especificación conforme va siendo construida, incrementando con esto la seguridad y confianza de éste.

- Dependencia estrecha con el lenguaje de Programación. Los lenguajes Larch de Interfaz comprenden expresiones que están muy estrechamente relacionadas con el lenguaje de programación, las cuales son necesarias para escribir de manera concisa y comprensible las especificaciones.

Los lenguajes Larch de Interfaz están diseñados especialmente para especificar módulos de programas, los cuales son también útiles cuando son necesitados por otras estructuras de datos.
Cuando el lenguaje de especificación refleja la estructura de un lenguaje de programación, tal como lo hacen los lenguajes Larch de Interfaz, la especificación es generalmente más pequeña y entendible para los programadores, además de facilitar así la comunicación entre estos módulos. Cada uno de estos lenguajes de especificación se encarga de mostrar lo que se puede aseverar en el comportamiento de un dato abstracto, para un lenguaje particular de programación. Proporcionan un conjunto de cláusulas para escribir estas aseveraciones, incorporando expresiones para contrucciones tales como efectos laterales, manejo de excepciones y de iteraciones.

El lenguaje Larch Compartido, es más bien algebraico, pues define, a través de ecuaciones e igualdades algebraicas, relaciones entre funciones, las cuales establecen o definen las propiedades y características de un dato abstracto.

2.2 Descripción del Lenguaje Larch.

El lenguaje Larch Compartido, se constituye básicamente de la especificación de pequeños módulos, donde se describen las propiedades de un dato abstracto a través de funciones, las cuales son restringidas aquí mismo de acuerdo también a ciertas propiedades. Este mismo lenguaje de especificación describe la manera de construir y relacionar estos bloques de especificación a través de reglas y cláusulas sintácticas. Los lenguajes Larch de Interfaz indican la manera de implantar estos módulos de especificación en un lenguaje de programación.
A continuación se hará una descripción de cada uno de estos lenguajes.

2.2.1 El Lenguaje Larch Compartido.

Como se mencionó anteriormente, el lenguaje Larch Compartido se compone de módulos de especificación, llamados Característica (Trait). Esta unidad básica de especificación establece un conjunto de funciones, posiblemente vacío, para las cuales define su sintaxis y semántica. Este conjunto de funciones define las operaciones permisibles en el dato abstracto a especificar, es decir, sus propiedades y características.
Como ejemplo, en la siguiente especificación, que corresponde a una tabla que almacena valores a través de índices, se muestra la estructura general de una característica (los puntos suspensivos y números del lado derecho son sólo para hacer referencia a los axiomas más tarde) :

```
1 { Tabla-De-Valores : Trait
    Introduces
           Crea :  => Tabla
         Agrega : Tabla, Indice, Valor => Tabla
        Elimina : Tabla, Indice => Tabla
  2{      # ∈ # : Indice, Tabla => Booleano
         Evalúa : Tabla, Indice => Valor
      EstaVacío : Tabla => Booleano
         Tamaño : Tabla => Entero
    Constrains  Crea, Agrega, ∈ , Elimina, Evalúa, EstaVacío,
                Tamaño  so that
              Tabla generated by [Crea, Agrega]
              Tabla partitioned by [ ∈ , Evalúa]
      for all [ind, ind1 : Indice; val : Valor; t : Tabla]

      Elimina( Agrega( t, ind, val), ind1) =
         If ind1 = ind1
           Then Elimina( t, ind1 )  ........................ 1
         Else
           Agrega( Elimina( t, ind1), ind, val)  ........... 1'
      ind ∈ Crea = Falso  .................................. 2
      ind ∈ Agrega( t, ind1, val) =
                               (ind = ind1) | (ind ∈ t)  ... 3
      Evalúa( Agrega( t, ind, val), ind1) =
         If ind = ind1
 3{        Then val  ....................................... 4
         Else
           Evalúa( t, ind1)  ............................... 4'
      EstaVacío( Crea ) = Verdadero  ....................... 5
      EstaVacío( Agrega( t, ind, val )) = Falso  ........... 6
      Tamaño( Crea ) = 0  .................................. 7
      Tamaño( Agrega( t, ind, val)) =
         If ind ∈ t Then
           Tamaño( t )  .................................... 8
         Else
           Tamaño (t) + 1  ................................. 8'
    Exempts[ Evalúa( Crea, ind ), Elimina( Crea, ind ) ]
    Implies Converts[ ∈ , Evalúa, EstaVacío, Tamaño ]
```

Una característica siempre está dividida en tres partes.

Al inicio de la primera parte (1) se define el nombre de la especificación, indicando que se trata de una característica. En el ejemplo anterior, esto está indicado con la frase 'Tabla-De-Valores : Trait'.
A continuación, en la segunda parte (2) y en seguida de la palabra 'Introduces' (Introduce), se declara la sintaxis de un conjunto de funciones, las cuales indican las operaciones deseadas a realizarse en el dato abstracto.
Para cada una de estas funciones, se debe especificar, si se requiere: su dominio, es decir, la información que requiere para llevar a cabo su propósito, el cual debe definirse al lado izquierdo de este símbolo: =>; y su rango, o sea el resultado esperado al aplicar en el dato abstracto esta función y que debe figurar del lado derecho de este mismo símbolo.

Estas funciones, por su manera de definirse en la tercera parte, describen la especificación completa del dato abstracto por medio de axiomas.

En la tercera parte de una característica (3), se definen los valores de las funciones antes definidas, así como que es lo que deben de hacer, a través de la cláusula 'Constrains <funciones a restringir>' (Restringe), por medio de ecuaciones algebraicas que las relacionan, llamados axiomas, que definen la semántica de cada función.

En el ejemplo anterior, las funciones definidas en esta parte describen las operaciones deseadas para manipular una tabla de valores a través de índices como son: en principio crear la tabla (inicializarla), agregar un valor a ella a través de un índice, eliminar un valor de ella también a través de un índice, saber si un índice dado está en la tabla, obtener un valor de la tabla a través de un índice, saber si está vacía y así como saber el número de índices que contiene (o sea, el tamaño de la tabla).
En esta característica, la función Crea no tiene dominio, ya que no se requiere de ninguna información para crear el dato abstracto Tabla. Los símbolos # de la cuarta función, indican el lugar correspondiente donde deben sustituirse (de izquierda a derecha) los elementos del dominio especificados para aplicar esta función.

En una característica, a los tipos de datos abstractos u objetos que se pretenden especificar a través de las funciones y axiomas, se les llama Tipos de Datos de Interés (TDI). En la característica anterior, el TDI es Tabla (t).

En general, las funciones que pueden figurar en una característica se dividen en dos clases principales, que son: constructoras y observadoras. Las funciones constructoras son aquellas que crean objetos del TDI definido, o sea cuyo rango es de este tipo.
Y las funcionas observadoras son las funciones que dicen algo respecto al TDI. El rango de estas funciones son elementos de tipos que no se están especificando en la característica.
El conjunto de funciones constructoras se divide a su vez en dos clases: generadoras y extensiones. Las funciones generadoras son el conjunto mínimo de funciones constructoras que permiten crear

o generar todos los objetos del TDI. Estas funciones son importantes en la característica, para un mejor entendimiento del TDI, por ello deben declararse explícitamente en ella. Esto se hace a través de la cláusula generated by [<func. generadoras del TDI>]. Las funciones extensiones son las funciones que se pueden definir en términos de las generadoras y son una extensión de aquéllas.
Por ejemplo, en la característica anterior, las funciones observadoras son ∈, Evalúa, EstaVacío y Tamaño. Las funciones constructoras generadoras son Crea y Agrega. Y la única función constructora extensión es Elimina, pues todas las tablas que resultan de aplicar esta función, se pueden obtener usando solamente las funciones generadoras Crea y Agrega.

Existen dos cláusulas que pueden figurar al principio de la tercera parte de una característica: TDI generated by <funciones generadoras> (TDI generado por ...) y TDI partitioned by <funciones observadoras> (TDI particionado por ...). La primera fue explicada ya anteriormente. La segunda indica las funciones que sirven para diferenciar dos objetos del TDI, esto es, para crear clases de equivalencia entre los objetos generados, lo cual resulta a veces útil. Esta cláusula afirma que estos dos objetos pertenecen a la misma clase de equivalencia si son indistintos los resultados de aplicar todas estas funciones observadoras sobre ellos, pero si al aplicar al menos una en ambos se obtiene una respuesta distinta, entonces pertenecen a diferente clase de equivalencia.

En esta misma tercera parte, también se deben establecer (si es que las hubiera), aquellas funciones con sus argumentos, las cuales al ser aplicadas con estos argumentos, pueden originar problemas por ambigüedad en la especificación del TDI, o que carecen de significado. La especificación de estas funciones en la característica se hace a través de la cláusula Exempts[<funciones con argumentos sin significado>] (Eximir las funciones ...).
Si alguna otra característica hace referencia (como se verá más adelante) a la característica que define estas funciones, entonces éstas seguirán siendo válidas en ella.

Por último, también aquí deben establecerse aquellas funciones, que han quedado completamente definidas en la característica y que no requieren de ningún otro axioma que las restringa aún más, aunque pueden servir para construir otras características de datos abstractos.
Ya que sucede comúnmente (como se verá más adelante), que al definir una función en una característica, esta función puede servir para definir o completar la especificación de otro dato abstracto imponiendo, posiblemente, otros axiomas sobre ella.

Para la especificación en la característica de estas funciones, se hace uso de la cláusula Implies Converts[<funciones completamente definidas>] (Implica Definidas las funciones ...).
Así, en la característica anterior Tabla-De-Valores, la cláusula Exempts[Evalúa(Crea, ind), Elimina(Crea, ind)] indica que no es válido hacer referencia a este conjunto de estas funciones

de esta forma.

En base a los axiomas definidos en una característica, se pueden proponer nuevos axiomas, a los que se les llamará teoremas, los cuales deben poderse demostrar a partir de los axiomas definidos. El conjunto de axiomas y de estos teoremas y reglas de inferencia del cálculo de predicados de primer orden, es llamado Teoría de la característica.

Algunos teoremas de la teoría de la característica anterior son los siguientes (suponiendo que val1 : Valor y que ind2 : Indice):

TEOREMA 1. Demostrar si es válida la siguiente igualdad:

Tamaño(Agrega(Agrega(Crea, ind, val), ind1, val1)) = 2

Demostración.

Aplicando el axioma 8' se tiene que:
Tamaño(Agrega(Agrega(Crea, ind, val), ind1, val1)) =
Tamaño(Agrega(Crea, ind, val) + 1
y aplicando de nuevo este axioma se tiene que:
= Tamaño(Crea) + 1 + 1 y por el axioma 7:
= 0 + 1 + 1 = 2. □

TEOREMA 2. Demostrar si es válida la siguiente igualdad:

ind2 ∈ Agrega(Agrega(Crea, ind, val), ind1, val1) = Falso

Demostración.

Aplicando el axioma 3, segunda opción se tiene que:
ind2 ∈ Agrega(Agrega(Crea, ind, val), ind1, val1) =
Ind2 ∈ Agrega(Crea, ind, val)
aplicando de nuevo este axioma:
= ind2 ∈ Crea y por el axioma 2:
= Falso. □

TEOREMA 3. Demostrar si es válido la siguiente igualdad:

Evalúa(Agrega(Agrega(Agrega(Crea, ind, val), ind1, val1), ind, val2), ind) = val2

Demostración.

Aplicando el axioma 4 se sigue de inmediato que:
Evalúa(Agrega(Agrega(Agrega(Crea, ind, val), ind1, val1), ind, Val2), ind) = val2. □

TEOREMA 4. Demuestre si es válida la propiedad conmutativa de la función Agrega, es decir, compruebe si:

∀t : Tabla --> Agrega(Agrega(t, ind1, val1), ind2, val2) =
Agrega(Agrega(t, ind2, val2), ind1, val1)

Demostración.

Sea t1 el miembro izquierdo de la igualdad y t2 el derecho Como se explicó en la cláusula Tabla partitioned by [∈, Evalúa] de esta característica, estas funciones son suficientes para de

cidir si t1 = t2.

Se tiene que t = t, luego:

```
ind1 ∈ t1 = verdadero   aplicando el axioma 3 dos veces
ind2 ∈ t1 = verdadero        "        "      "  3
Evalúa( t1, ind1) = val1  aplicando los axiomas 4 y 4'
Evalúa( t1, ind2) = val2        "        el axioma 4
pero:
ind1 ∈ t2 = verdadero   aplicando el axioma 3 dos veces
ind2 ∈ t2 = verdadero        "        "      "  3
Evalúa( t2, ind2) = val1  aplicando los axiomas 4 y 4'
Evalúa( t2, ind2) = val2        "        el axioma 4
```

entonces, como al aplicar estas funciones sobre los objetos t1 y t2 no hubo ninguna distinción entre ellos, o sea se obtuvieron los mismos resultados, entonces t1 = t2 y por lo tanto es válida la propiedad conmutativa de la función Agrega. □

En la práctica, al definir especificaciones con Larch, no es suficiente escribirlas a través de funciones con restricciones, sino que a veces es necesario escribir varias características para poder completar una sola especificación.
Surge entonces la necesidad de poder relacionar o hacer referencias entre éstas características, a través de un conjunto de cláusulas reservadas que indiquen las relaciones entre características o funciones.
El lenguaje Larch Compartido ofrece un conjunto de cláusulas para poder relacionar características en una especificación o provenientes de otra especificación, así como cláusulas generales que indican hacer algo específico sobre axiomas, características o funciones. Por ejemplo, incluir características ya definidas en otra característica, las cuales no están presentes en ella, o restringir aún más una función de una característica asumida por otra característica. En seguida se mencionarán algunas de estas cláusulas.

La siguiente característica, "Caja", especifica las operaciones comúnes de aquellas estructuras de datos que almacenan elementos, que pueden ser Conjuntos, Colas, Pilas, Arreglos, etc.
Esto es posible ya que una característica puede corresponder a objetos de cualquier tipo. Estas operaciones son: crear el dato abstracto e insertar elementos a él.

```
Caja : Trait
   Introduces
         Crea :  => C
      Inserta : C, E => C
   Constrains C so that
      C generated by [Crea, Inserta]
```

Esta característica especifica el TDI parametrizado C, esto significa que C es un parámetro que puede ser sustituido por Conjuntos, Colas, Pilas, Arreglos, etc. los cuales tienen la particu

laridad de almacenar elementos, que en la caracteristica son llamados objetos de tipo E.

Una cláusula muy importante y útil es 'Assumes' (Asumir), la cual indica que la caracteristica asume a otra u otras caracteristicas señaladas en esta cláusula.
Por ejemplo, la caracteristica Esta-Vacio, la cual especifica cuando un objeto de tipo 'C' tiene al menos uno o ningún elemento, asume a la caracteristica 'Caja', es decir, hace referencia a ella.

```
Esta-Vacio: Trait
   Assumes Caja
   Introduces
      EstaVacio : C => Booleano
   Constrains  EstaVacio, Crea, Inserta  so that
            for all [c : C, e : E]
      EstaVacio( Crea ) = Cierta
      EstaVacio( Inserta( c, e)) = Falso
   Implies Converts[ EstaVacio ]
```

Como puede verse, la caracteristica 'Esta-Vacio' también restringe las funciones definidas en la caracteristica asumida, 'Caja', en su definición, siendo esto la propiedad principal de la cláusula 'Assumes'.

Las caracteristicas 'El-Siguiente' y 'Los-Restantes' especifican, respectivamente, cuál es el siguiente elemento y cuáles son los restantes elementos (que puede ser uno) de un objeto de tipo 'C', después de haber hecho una inserción a este objeto.

```
El-Siguiente : Trait
   Assumes Caja
   Introduces
      Siguiente : C => E
   Constrains  Siguiente, Inserta  so that
            for all [e : E]
      Siguiente( Inserta( Crea, e)) = e
   Exempts[ Siguiente( Crea ) ]

Los-Restantes : Trait
   Assumes Caja
   Introduces
      Resto : C => C
   Constrains  Resto, Inserta  so that
            for all [e : E]
      Resto( Inserta( Crea, e)) = Crea
   Exempts[ Resto( Crea ) ]
```

Estas caracteristicas muestran, respectivamente, las proposiciones Exempts[Siguiente(Crea)] y Exempts[Resto(Crea)].
La primera significa que la expresión 'Siguiente(Crea)' no tiene significado, porque no se puede hacer referencia a los siguientes elementos de un objeto de tipo 'C', luego de haberlo creado, sin haber insertado antes algún elemento a él. Y por una razón muy similar, la segunda expresión tampoco tiene significado.

Una cláusula similar a 'Assumes' es la cláusula 'Imports' (Im-

portar), la cual tiene el mismo efecto que esta cláusula: el hacer referencia a la característica que indica, excepto que con esta cláusula no es posible restringir aún más las funciones de la característica o características importadas por esta cláusula. Estas pueden ser completamente entendidas independientemente del contexto en cual son traidas.

Otra cláusula también similar a 'Assumes' es la cláusula 'Include' (Incluir). Esta cláusula, además de hacer referencia a la característica o características a asumir, es decir heredar a la característica todas las funciones restringidas por éstas, tiene la particularidad también de heredar a la característica las funciones restringidas por las características que éstas asuman o aun mismo incluyan, pudiendo la característica (donde esta escrita esta cláusula) también restringir las funciones de estas características, que a diferenrencia de la claúsula Asumes, no puede restringir.
La característica siguiente muestra esta cláusula:

```
Distigue-Cajas : Trait
   Includes  El-Siguiente, Los-Restantes, Esta-Vacío
   Constrains C so that C Partitioned by
               [Siguiente, Resto, EstaVacío]
```

Esta característica especifica cuando dos objetos de tipo C, los cuales mantienen un orden definido en sus elementos, tales como pilas, arreglos, colas, colas con prioridad, etc. son iguales. O sea, si:

```
Siguiente( t1 ) = Siguiente( t2 )
    Resto( t1 ) = Resto( t2 )
EstaVacío( t1 ) = EstaVacío( t2 )
```

entonces t1 = t2, donde t1 y t2 son dos objetos de tipo "C". Pero si al menos una de estas igualdades no se cumple, entonces el objeto t1 es distinto del t2.

Como se podrá concluir ahora, las cláusulas Assumes, Imports e Include permiten estructurar mejor la especificación de un tipo de datos abstracto, ya que permiten ir construyendo características más complejas a partir de otras más simples, haciendo a la vez que sean más fáciles de leer y entender. De esta forma se van componiendo las propiedades especificadas en cada uno de ellas, hasta tener una estructura más compleja que va tomando lo definido anteriormente.

Una cláusula más es la cláusula "With" (Con), la cual se muestra en la siguiente característica. Esta especifica, considerando las características anteriores, una cola con prioridad definida en un objeto de tipo C:

```
Cola-Con-Prioridad : Trait
   Assumes  Orden-Total With [E for T]
   Includes  Distinque-Cajas
   Constrains  Siguiente, Resto, Inserta
```

```
so that for all [q : C, e : E]
Siguiente( Inserta( q, e)) = If EstaVacío( q ) then e
                            else
                              If Siguiente( q ) ≲ e
                                 then Siguiente( q )
                              else e
Resto( Inserta( q, e) = If EstaVacío( q ) then Crea
                        else
                          If Siguiente( q ) ≲ e then
                            Inserta( Resto( q ), e)
                          else q
Implies Converts[ Siguiente, Resto, EstaVacío ]
```

La cláusula 'With' indica que la característica 'Orden-Total', no descrita aquí, es asumida con el parámetro 'E' en sustitución de su correspondiente parámetro 'T' (de tipo general). Esto se hace con el fin de ser más formal y consistente con la representación de tipos de datos en ambas características.
Esta característica especifica el criterio para saber cuál de los elementos agregados al objeto de tipo C, tiene menor o mayor prioridad (para ser eliminados u obtener información de ellos). Este criterio se especifica como una relación entre los elementos agregados, tal que indusca en ellos un orden total (esta relación se especifica por el símbolo ≲). Esta característica se mustra en el apéndice A.

Aunque la proposición Implies Converts[EstaVacío] de la característica incluida Esta-Vacío es heredada aquí, la proposición 'Implies Converts[Siguiente, Resto, EstaVacío]' es una proposición más explícita para indicar que estas funciones han quedado completamente definidas con la característica 'Cola-Con-Prioridad', y ningún otra característica que haga referencia a ellas puede definir nuevos axiomas sobre ellos, ya que entonces afectarían a la especificación de esta característica.

Para finalizar con la descripción del lenguaje Larch Compartido, a continuación se muestra la característica Iguales-Elementos, la cual establece cuando dos objetos de un tipo son iguales, a través del tipo parametrizado (T).
Esta característica es asumida por la característica 'Multi-Conjuntos'. Esta especifica las operaciones a realizarse en un dato abstracto, el cual reune varios conjuntos con distintos tipos de elementos:

```
Iguales-Elementos : Trait
   Introduces
      # == # : T, T => Boleano
   Constrains  T so that T partitioned by [ == ]
            for all [x, y, z : T]
        ( x == x ) = Verdadero
        ( x == y ) = Falso
        ( x == y ) = ( y == x )
        ( ( x == y ) & ( y == z ) ) = ( x == z )

Multi-Conjuntos : Trait
```

```
Assumes  IgualesElementos With [E for T]
Includes  Esta-Vacío, Tamaño-Caja
          With [Mconj for C, {} for Crea]
Introduces
          Cuántos : Mconj, E => Número Cardinal
          Elimina : Mconj, E => Mconj
     NúmConjuntos : Mconj => Número Cardinal
Constrains  Mconj so that Mconj partitioned by [Cuántos]
            for all [Mc : Mconj; e, e1, e2 : E]
   Cuántos( {}, e1 ) = 0
   Cuántos( Inserta( Mc, e1 ), e2) = Cuántos( Mc, e2 ) +
                                  ( If e1 == e2 then 1 else 0 )
   Tamaño( Inserta( Mc, e ) ) = Tamaño( Mc ) + 1
   NúmConjuntos( {} ) = 0
   NúmConjuntos( Inserta( Mc, e ) ) = NúmConjuntos( Mc ) +
                                    ( If Cuántos( Mc, e ) > 0
                                        then 0
                                      else 1 )
   Elimina( {}, e1 ) = {}
   Elimina( Inserta( Mc, e1 ), e2 ) = If e1 == e2 then Mc
                                        else
                                 Inserta( Elimina( Mc, e2 ), e1 )
Implies Converts[ EstaVacío, Cuántos, Elimina,
                  NúmConjuntos, Tamaño]
```

La característica "Tamaño-Caja" no descrita aquí, asume la característica Caja e introduce la función: Tamaño : C => Entero, con el axioma Tamaño(Crea) = 0. Su finalidad es fijar como cero el tamaño (que en el caso del objeto Mconj, es el número de elementos que contiene) de un objeto de tipo Mc recien creado.

La frase "Constrains Mconj" en la característica anterior, es una forma de indicar también la cláusula "Constrains" abarcando todas las funciones cuyo dominio o rango sea de tipo "Mconj". La expresión "Mconj partitioned by [Cuántos]", indica que la función "Cuántos" es suficiente para distinguir objetos de tipo Mconj, a través del parámetro dado "E" en su dominio. Esta función cuenta el número de elementos de una clase dada "E" que hubiera en el objeto MConj. En cambio, la función "NúmConjuntos" cuenta el número de conjuntos (subconjuntos de tipo E) contenidos en el objeto Mconj. Aquí, la cláusula "With" establece una sustitución de funciones válida también; la sustitución de la función "{}" por la función "Crea", de la característica Caja, así como la sustitución de un objeto de tipo Mconj por la de un objeto de tipo "C".

2.2.2 El Lenguaje Larch de Interfaz.

En la descripción del lenguaje de especificación Larch hecha hasta este momento, no se ha mencionado nada con respecto a cómo van a ser representadas las estructuras de datos de las características, ni el algoritmo para manejar estas estructuras. Estas dos decisiones se llevan a cabo durante la implantación de la especificación, pero las funciones o procedimientos (rutinas) para

manejar estas estructuras de datos, son definidas en el lenguaje Larch de Interfaz.

La finalidad de los languajes Larch de Interfaz, es describir las rutinas a ser implantadas en un programa. Y el objetivo del lenguaje Larch Compartido al establecer las características, es definir un conjunto de funciones que den significado a las rutinas que aparecerán en la especificación de Interfaz.

Como ya se dijo, los lenguajes Larch de Interfaz también están diseñados para la representación, en un lenguaje de programación, de las especificaciones de un tipo de datos abstracto.
En un lenguaje Larch de interfaz, desde los mecanismos de comunicación entre rutinas hasta la elección de palabras reservadas, está influenciada por el lenguaje de programación. Los únicos lenguajes de Interfaz hasta ahora bien desarrollados, corresponden a los lenguajes de programación Pascal y Clu, llamados Larch/Pascal y Larch/Clu, respectivamente.

La especificación de un dato abstracto, en ambos lenguajes de de Interfaz, tiene tres partes (véase el ejemplo siguiente).
La primera parte (1) contiene un encabezado que define el nombre del dato abstracto a especificar, a través de la cláusula 'Type <Nombre del TDI>', así como los nombres de las rutinas especificadas en éste lenguaje, indicándose éstas a través de la cláusula 'Exports <Rutinas especificadas>'. La segunda parte (2) hace referencia a una característica; describe el nombre de ésta y el TDI que especifica, a través de la cláusula 'based on <Nombre del TDI de la ...> From <Caracteristica asumida>.
Y la tercera parte (3) comprende la especificación de Interfaz de cada rutina.
La especificación de Interfaz de una rutina tiene a su vez dos partes. La primera parte (3.1) declara un encabezado dando el nombre de la rutina y los nombres y tipos de sus parámetros formales. Y la segunda parte (3.2) se compone de un conjunto de frases en las cuales se establece, siguiendo la idea de Hoare de pre y pos condiciones, que condición o condiciones se requieren antes de modificar posiblemente un parámetro formal (pre-condiciones), así como las pos-condiciones que indican cómo se debe comportar la rutina para asegurarlas y cuáles de sus parámetros se verán modificados.
La siguiente especificación en Larch/Pascal es del TDI de la característica Tabla-De-Valores, el cual se definió al principio de este capítulo (sección 2.2.1). En ella figuran tres procedimientos y cuatro funciones.

```
 1  Type Tabla Exports IniciaTab, AgregaInd, PerteneceInd,
              ValorInd, EliminaInd, TablaVacia, TamañoTab
       2  Based on Sort Tabla From Tabla-De-Valores
                          With [ { } for Crea ]

   3.1  Procedure IniciaTab( Var t : Tabla );
                  modifies at most [ t ]
            3.2   ensures t_post  = { }
        Procedure AgregaInd( Var t : Tabla; ind : Indice;
```

```
                    val : Valor )
          requires Tamaño( Agrega( t, ind, val )) <= 100
          modifies at most [ t ]
          ensures t_post  = Agrega( t, ind, val )

Function PerteneceInd( t : Tabla; ind : Indice) : boolean
         ensures PerteneceInd = ind ∈ t

Function ValorInd( t : Tabla; ind : Indice) : Valor;
         requires t <> { }
         ensures ValorInd_post = Evalda( t, ind )

Procedure EliminaInd( Var t : Tabla; ind : Indice;
                      val : Valor);
          requires t <> { }
          modifies at most [ t ]
          ensures t_post = Elimina( t, ind )

Function TablaVacia( t : Tabla ) : boolean;
         ensures TablaVacia = EstaVacio( t )

Function TamañoTab( t : Tabla ) : Integer;
         ensures TamañoTab = Tamaño( t )


End Tabla
```

3

Seguiremos las partes antes descritas en esta especificación. En ella, el nombre del dato abstracto a especificar es Tabla. Las rutinas que especifica se refiren a las operaciones a realizarse en la tabla que almacena valores a través de índices, establecidas en la característica Tabla-De-Valores.

La cláusula 'Based on' establece una relación entre tipos de objetos, que en este ejemplo es entre el dato abstracto 'Tabla', el cual se desea especificar y el TDI 'Tabla' especificado en la característica Tabla-De-Valores.
Es decir, que los objetos de tipo 'Tabla' que aparecen en esta característica, son usados en el lenguaje de Interfaz como parámetros para representar objetos de este tipo. Esta asociación de tipos no necesariamente debe coincidir, ya que es posible especificar un tipo de datos abstracto en Larch/Pascal a partir de otra característica más general.

Los parámetros formales de una rutina de una especificación de Larch/Pascal pueden ser modificados o no, indicándose con la palabra 'post' como subíndice, en aquellos parámetros que podrán sufrir algún cambio en la rutina, y que mantendrán este valor después de que regrese de donde fue llamada (de hecho estos parámetros son transferidos al procedimiento o función por variable).
Un parámetro formal no modificado es aquel parámetro que mantiene su valor original aun después de que el procedimiento o función fue llamado.
Como 't_{post}' en este ejemplo, donde 't' es un parámetro formal de tipo Tabla, al cual después de haberle agregado o eliminado algún

índice o inicializado con vacío, se requiere que aún después de regresar la rutina de donde fue llamada, siga manteniendo ese valor.

Analicemos ahora en particular el segundo procedimiento llamado AgregaInd. Este tiene como finalidad agregar a la tabla un elemento en un lugar específico, a través de un índice. Esta información es transferida al procedimiento a través de parámetros formales. Supongamos que de acuerdo a lo especificado por un usuario, se desea que el número de elementos después de agregar uno nuevo, sea menor o igual que 100.
Esta pre-condición se debe establecer al principio del procedimiento, a través de la cláusula 'requieres' (se requiere que ...) Las funciones que siguen a esta cláusula, Tamaño y Agrega son las establecidas en la característica Tabla-De-Valores y verifican esta pre-condición.

Este tipo de consideraciones sobre las limitaciones o restricciones de un objeto o parámetro, como el tamaño de un arreglo o los posibles valores que puede tomar un parámetro, se establecen en la especificación de Interfaz, y no que en las características, y son restricciones dadas por la implantación en concreto.

Si se cumple la condición anterior, entonces se deseará agregar a la tabla este elemento y entonces, la tabla de valores quedará modificada. Este tipo de modificaciones que pudiera sufrir una variable, se deben especificar aquí a través de la cláusula 'modifies at most [V1, v2, ..., Vn]' (modificar al menos [V1, V2, ...,Vn]), donde V1, v2, ..., Vn son variables de algún tipo.
Esta cláusula establece que la rutina cambiará, posiblemente, el valor de algunas o todas las variables V1, V2, etc., las cuales están en la rutina o programa que la llaman.
La cláusula que especifica todo lo contrario, es decir, afirma que al final de la ejecución de una rutina ningún parámetro formal ha sufrido modificación alguna, es la cláusula 'modifies nothing'.

En esta rutina entonces, al agregar un elemento en la tabla ésta quedará modificada, lo cual está indicado por la cláusula modifies at most [T].

Las cláusulas anteriores, así como algunas otras que se mencionarán, son propias del lenguaje Larch/Pascal, ya que cada lenguaje Larch de Interfaz contiene sus propias cláusulas y proposiciones, heredadas del mismo lenguaje de programación.

En este procedimiento (AgregaInd) falta indicar cuál es la finalidad que tiene, lo que debe de hacer o asegurar si se cumple la pre-condición establecida. Esto se hace con la cláusula 'ensures' (Asegurar). Esta cláusula establece la pos-condición, que dice cuáles parámetros quedaron modificados después de la ejecución de la rutina y cómo.

Así, en este procedimiento, la proposición ensures t_{post} = Agrega (t, ind, val) asegura que al finalizar la ejecución de la ru

tina y aún después de regresar de donde fue llamada, el objeto "t", de tipo Tabla, deberá contener al elemento cuyo índice es "ind" y valor "Val," porque el propósito de la rutina es agregarlo.

Las demás cláusulas que aparecen en las otras rutinas, tienen un propósito semejante a las explicadas anteriormente.

Como se podrá concluir, a diferencia de las características en una especificación de Interfaz, se establecen las rutinas que deben ser implantadas y en cada una de ellas se establecen las condiciones o requerimientos sobre los parámetros de entrada y variables globales, así como también si ellos han quedado probablemente modificados al finalizar la ejecución de la rutina.

Con respecto al diseño de una especificación de Interfaz, el usuario es quien establece las variables de la cláusula "requieres", de acuerdo a sus requerimientos. También ellos deben presuponer las variables contenidas en las cláusulas "ensures" y "modifies at most", sin importarle como la rutina cambiará estas variables ni a través de que algoritmo.
En cambio, los programadores en principio asumen la cláusula "requieres", después deben establecer la cláusula "ensures" respetando la cláusula "modifies at most". De esta manera, al construir una especificación de Interfaz queda establecido un acuerdo entre los especificadores y los usuarios de un dato abstracto.

El lenguaje de Interfaz Larch/CLU es diferente a Larch/Pascal, porque Larch/CLU contiene cláusulas relativas al lenguaje de programación CLU, así como la misma construcción de datos abstractos se hace de manera diferente. Este lenguaje no se describirá aquí por ser Clu un lenguaje de programación poco común.

Para una descripción más completa y detallada de ambos lenguajes de Interfaz puede consultarse la referencia bibliográfica [Guttag et al., 85].

2.3 Aplicando el Lenguaje de Especificación Formal Larch

En esta última parte de este capítulo se verá un ejemplo aplicando el lenguaje Larch. La finalidad de este ejemplo es mostrar los conceptos expuestos anteriormente, así como mostrar la manera de construir una especificación formal en el lenguaje Larch a partir de su especificación en lenguaje natural.
El lenguaje de Interfaz que se usará será Larch/Pascal.

Primeramente, se describen los pasos que se sugieren para construir una especificación formal en el lenguaje Larch.

1.- Expresar los requerimientos del usuario a través de frases concisas (operaciones a realizar).

2.- Desarrollar una intuición aproximada del problema. Esto requiere primero de una conversación verbal con el usuario.

3.- Identificar los TDI involucrados.

4.- Escribir la información componente del lenguaje de Interfaz para cada TDI, así como la interfaz con la característica correspondiente. Esto sugiere considerar cada operación requerida como un procedimiento, función o conjunto de ellos, y escribir los encabezados y parámetros formales de éstos.

5.- Escribir la información sintáctica y semántica de los componentes del lenguaje Compartido para cada característica que respaldará el significado de cada TDI, así como la interacción entre cada una de ellas (si existe). Se deben establecer las funciones necesarias para caracterizar las operaciones requeridas sobre el TDI que define cada característica.

6.- Completar la especificación de Interfaz, escribiendo aseveraciones en las especificaciones de las rutinas a través de las cladsulas 'requieres', 'modifies at most' y 'ensures', de acuerdo a las operaciones definidas por el usuario sobre los TDI.
Estos dos últimos pasos se llevan a cabo simultáneamente.

7.- Mostrar y explicar al usuario si la especificación está de acuerdo a lo que él espera; si existe algun error o mal entendido repetir los pasos anteriores.

Durante el proceso de escritura de una especificación, se deberá también evaluar en ella la presencia de ciertas propiedades, que son su completéz y totalidad. Estas tienen como fin incrementar la confianza de que la especificación está correcta. La primera se refiere a las funciones definidas en una característica, y consiste en que éstas deben estar especificadas para cualquier argumento que pueda contener, ya sea a través de axiomas o definidas bajo la cladsula exempts. La segunda se explicará y se mostrará más adelante.

Siguiendo los pasos antes mencionados para escribir la especificación, en lenguaje Larch, de un directorio de una ciudad, el cual contendrá el nombre, la dirección y el teléfono de las personas que viven en ella. Se supondrá que una persona sólo puede tener una sola dirección y teléfono registrados, y que a su vez está información registrada corresponde exactamente a una sola persona.

Siguiendo los pasos 1 y 2 antes descritos, se consulta con el usuario del directorio lo que espera de él, o sea las operaciones o requerimientos que desea hacer, obteniendo las siguientes funciones:

a) Agregar el nombre, la dirección y el teléfono de una persona al directorio, siempre que no esté.

b) Obtener la dirección y el teléfono de una persona que está en el directorio.

c) Saber si una persona está o no en el directorio.

d) Eliminar el nombre, la dirección y el teléfono de una persona del directorio.

e) Modificar la dirección y el teléfono de una persona que está en el directorio.

f) Saber el ndmero de personas que están en el directorio.

Intuitivamente se puede pensar el directorio como una tabla que almacena llaves con sus correspondientes valores, donde los nombres de las personas son las llaves y las direcciones y teléfonos son los valores.

Siguiendo el paso 3 con la información antes descrita, se empezará ahora a construir la especificación abstrayendo estos requerimientos en el lenguaje Larch. Se eligirá como tipos de datos de interés el directorio (Direct), el nombre de las personas (NomPers), la dirección y el teléfono de éstas (DirecTel).
Esto sugiere una primera aproximación de la especificación, para escribir grupos de procedimientos o funciones (rutinas) para cada uno de estos tres tipos de datos, de acuerdo al paso 4, que constituirá la especificación de Interfaz (véase fig. 1 de la siguiente hoja), así como las características que definirán las funciones de cada uno de ellos. Cada una de estas características

se llamará, respectivamente, Directorio, Nombre-De-Personas y Direcc-Tel-De-Personas (véase fig. 2).
Los puntos suspensivos indican que falta por definirse o completarse el concepto.

```
Type Direct Exports ....
     Based on Sort .... From Directorio
          ....
     End Direct

Type NomPers Exports ....
     Based on Sort .... From Nombre-De-Personas
          ....
     End NomPers

Type DirecTel Exports ....
     Based on Sort .... From Direcc-Tel-De-Personas
          ....
     End DirecTel
```

Fig. 1 Componentes del Lenguaje de Interfaz.

```
Directorio : Trait
   Introduces
      ...
   Constrains
      ...

Nombre-De-Personas : Trait
   Introduces
      ...
   Constrains
      ...

Direcc-Tel-De-Personas: Trait
   Introduces
      ...
   Constrains
```

Fig. 2 Componentes del Lenguaje Compartido.

Se empezará primero a construir el grupo de rutinas que comprende el lenguaje de Interfaz para el TDI Direct, así como a especificar la característica correspondiente, de acuerdo a los pasos 5 y 6. Más tarde se definirán los demás tipos de datos junto con sus respectivas características. Este grupo de rutinas dependerá de las operaciones a realizarse en el directorio.

Se definirá entonces la especificación de cada una de estas operaciones a través de procedimientos y funciones, definiendo para ello también, las funciones que deberán figurar en la característica Directorio, las cuales darán significado a estas rutinas. Empecemos por construir la especificación de la operación para crear el directorio, a través de un procedimiento llamado CreaDirectorio. En él no se desea restringir ningún parámetro de entra-

da, sino garantizar que el tipo de dato Direct, que es el valor que regresará el procedimiento, no contenga ningún elemento y además esté listo para admitir nombres, direcciones y teléfonos de personas. La especificación de este procedimiento es la siguiente:

```
Procedure CreaDirectorio( Var D : Direct )
          ensures ( Dpost = SecreaDirect )
      end;
```

Para crear el objeto Direct, se usa la función SecreaDirect que aparece en la cláusula "ensures", la cual indica solo una inicialización de él. Esta función debe definirse en la característica Directorio, describiendo cual debe ser su dominio y rango.
La función no tendrá dominio (porque no requiere información alguna para ser creado) pero tendrá como rango el objeto "Dir" (que es de tipo Direct).
Con esta función ya definida, se puede ahora tener una referencia de la especificación de Interfaz de este objeto con respecto a la característica Directorio. En la frase Based on Sort .... es donde se establece esta relación.
Con esta información se tiene ahora la siguiente especificación de este TDI, incorporando en la especificación de Interfaz los encabezados de las rutinas a emplear para las demás operaciones a especificar, que corresponden (descritas en el mismo orden) a las operaciones definidas por el usuario:

```
Directorio : Trait
    Introduces
       SecreaDirect : => Dir
         ...
    Constrains
         ...
```

Especificación de la característica Directorio.

```
Type Direct Exports  CreaDirectorio, AgregaNomDirecTel,
          DireccTel, EstáPersona, EliminaPersona,
          ModificaDireccTel, TamañoDirectorio
     Based on Sort Dir From Directorio

     Procedure CreaDirectorio( Var D : Direct )
            ensures ( Dpost = SecreaDirect )
         end;

     Procedure AgregaNomDirecTel( Var D : Direct;
                              Np : NomPers; Dt : DirecTel );
            requieres ....
            modifies at most ....
            ensures ....
         end;
```

```
        Function DireccTel( D : Direct; Np : NomPers ) : DirecTel
                 requieres ....
                 modifies at most ....
                 ensures ....
             end;

        Function EstáPersona( D : Direct; Np : NomPers ) : Boolean
                 requieres ....
                 modifies at most ....
                 ensures ....
             end;


        Procedure EliminaPersona( Var D : Direct; Np : NomPers)
                  requieres ....
                  modifies at most ....
                  ensures ....
             end;

        Procedure ModificaDireccTel( Var D : Direct;
                                     Np : NomPers; Dt : DirecTel);
                  requieres ....
                  modifies at most ....
                  ensures ....
             end;

        Function TamañoDirectorio( D : Direct ) : Integer;
                 requieres ....
                 modifies at most ....
                 ensures ....
             end;

   End Direct
```

Especificación de Interfaz del TDI DIRECT.

Ahora se definirá la especificación de la operación para agregar el nombre, la dirección y el teléfono de una persona al directorio, también a través de un procedimiento llamado AgregaNomDirecTel.

En él se desea agregar esta información, solo si la persona no está en él. Esto se puede indicar a través de la clásula 'requieres' a la entrada del procedimiento, estipulando primero como precondición que el nombre de esta persona no debe estar.

Se observará que el objeto Direct es un tipo de dato que puede cambiarse o modificarse a lo largo de su uso, por la posible adición de nuevas personas en él, lo cual será indicado por la clásula 'modifies at most'.

Con esto tenemos la especificación de esta operación a través del siguiente procedimiento:

```
Procedure AgregaNomDirecTel( Var D : Direct; Np : NomPers;
                             Dt : DirecTel );
        requieres   ¬ Estéen( D, Np )
        modifies at most  [ D ]
        ensures D_post = AgregaPers( D, Np,Dt )
    end;
```

La definición de las funciones Estéen y AgregaPers se hace también en la característica Directorio. La primera función booleana tiene como finalidad establecer la pre-condición del procedimiento, es decir, saber si el nombre de una persona está en el directorio, mientras que la segunda asegura (por la cladsula ensures) el propósito del procedimiento: agregar al directorio el Nombre, el teléfono y la dirección de una persona.
Escribiendo los correspondientes axiomas para definir el comportamiento de estas funciones, se tiene completamente definido el procedimiento AgregaNomDirecTel.

Continuando con la especificación de las demás operaciones, a través de las rutinas antes establecidas e introduciendo las funciones y axiomas correspondientes para dar significado a estas rutinas, se tiene ahora la especificación completa del TDI Direct

```
Directorio : Trait
    Include Nombre-De-Personas with [ NomPers for Cadena ]
    Include Direcc-Tel-De-Personas with [ DirecTel for Cadena ]
    Introduces
        SecreaDirect : => Dir
          AgregaPers : Dir, NomPers, DirecTel => Dir
      ObtenDireccTel : Dir, NomPers => DirecTel
              Estéen : Dir, NomPers => Booleano
         EliminaPers : Dir, NomPers => Dir
      ModifDireccTel : Dir, NomPers, DirecTel => DirecTel
           Dimensión : Dir => Número Cardinal
    Constrains Dir so that
        Dir generated by [ SecreaDirect, AgregaPers ]
        Dir partitioned by [ ObtenDirecctel ]
        for all [D : Dir; N1, N2 : NomPers; Dt1, Dt2 : DirecTel]

        ObtenDireccTel( AgregaPers( D, N1, Dt1 ), N2 ) =
                                    If NomIguales( N1, N2 ) Then
                                       Dt1
                                    else
                                       ObtenDireccTel( D, N2 )
        Estéen( AgregaPers( D, N1, Dt1 ), N2 ) =
                           NomIguales( N1, N2 ) | Estéen( D, N2 )
        EliminaPers( AgregaPers( D, N1, Dt1 ) N2 ) =
                               If NomIguales( N1, N2 ) Then
                                  D
                               else
                      AgregaPers( EliminaPers( D, N2 ), N1, Dt1 )
        ModifDireccTel( AgregaPers( D, N1, Dt1 ), N2, Dt2 ) =
                                   If NomIguales( N1,  N2 ) Then
                                      AgregaPers(  D, N2, Dt2 )
                                   else
```

```
            AgregaPers( ModifDireccTel( D, N2, Dt2 ), N1, Dt1)
        Dimensión( SecreaDirect ) = 0
        Dimensión( AgregaPers( D, N1, Dt1)) = 1 + Dimensión( D )
        Estéen( SecreaDirect, N1 ) = Falso

    Exempts[ ObtenDireccTel( SecreaDirec, N1 ),
             EliminaPers( SecreaDirec, N1 ),
             ModifDireccTel( SecreaDirec, N1, Dt1 ) ]
    Implies Converts[ ObtenDireccTel, EliminaPers, Estéen,
                      ModifDireccTel, Dimensión ]
```

Especificación de la característica Directorio.

```
 Type Direct Exports  CreaDirectorio, AgregaNomDirecTel,
            DireccTel, EstáPersona, EliminaPersona,
            ModifDirecTel, TamañoDirectorio
      Based on Sorts Dir From Directorio

      Procedure CreaDirectorio( Var D : Direct )
              ensures ( D post = SecreaDirect )
         end;

      Procedure AgregaNomDirecTel( Var D : Direct;
                                   Np : NomPers; Dt : DirecTel );
              requieres   ¬ Estéen( D, Np )
              modifies at most  [ D ]
              ensures D post = AgregaPers( D, Np,Dt )
         end;

      Function DireccTel( D : Direct; Np : NomPers ) : DirecTel
              requieres Estéen( D, Np )
              ensures DireccTel = ObtenDireccTel( D, Np )
         end;

      Function EstáPersona( D : Direct; Np : NomPers ) : Boolean
              ensures EstáPersona = Estéen( D, Np )
         end;

      Procedure EliminaPersona( Var D : Direct; Np : NomPers)
              requieres Estéen( D, Np )
              modifies at most [ D ]
              ensures D post = EliminaPers( D, Np )
         end;

      Procedure ModificaDireccTel( Var D : Direct;
                                     Np : NomPers; Dt : DirecTel);
              requieres Estéen( D, Np )
              modifies at most [ D ]
              ensures D post = ModifDireccTel( D, Np, Dt )
         end;
```

```
      Function TamañoDirectorio( D : Direct ) : Integer;
               ensures TamañoDirectorio = Dimensión( D )
         end;

   End Direct
```

Especificación de Interfaz del TDI DIRECT.

Al definir el resto de las operaciones, como se tuvieron que especificar funciones que involucraban los tipos NomPers y DirecTel, se tuvieron que incluir al principio de la característica directorio, las características correspondientes a estos tipos, haciendo una sustitución apropiada de tipos (el tipo Cadena será el nombre del TDI que especificarán estas características, como se verá más adelante). La función NomIguales se especificará más tarde en la característica Nombre-De-Personas.

Ahora bien, generalmente después de haber escrito la especificación de un dato abstracto, es recomendable revisarla y discutirla. Es en esta inspección donde se evalúa la propiedad de Totalidad, mencionada en un principio.
Si se analiza la especificación anterior, se observará que el procedimiento AgregaNomDireccTel no está completamente definido, ya que no dice que hacer si al intentar agregar el nombre de una persona al directorio ésta ya está, cosa que sería muy común que sucediera.

Cuando una especificación como ésta no considera uno o varios casos posibles a suceder, se dice que la especificación no es total. Para que este procedimiento cumpla con esta propiedad, es deseable que contemple el caso de cuando se intenta agregar una persona ya registrada en el directorio.
Para que el procedimiento lo haga así, es necesario agregar en el encabezado del procedimiento (o función si es el caso) la instrucción Signals <variable booleana>, la cual indica que la variable contenida en ella, en este caso definida como Ya-Está-Persona, es booleana. Está, posiblemente se modificará en el procedimiento de acuerdo a las circunstancias, informando al programador lo que sucede cuando se intenta agregar una persona al directorio, que ya está en él.
De esta manera se tiene modificada la especificación de este procedimiento y es la siguiente:

```
   Procedure AgregaNomDirecTel( Var D : Direct;
                                Np : NomPers; Dt : DirecTel );
                          Signals( Ya-Está-Persona )
            modifies at most  [ D ]
            ensures If ¬Estáen( D, Np ) Then
                              Dpost = AgregaPers( D, Np,Dt )
                      else
                        Signals( Ya-Está-Persona ) &
                        modifies nothing
        end;
```

Por una razón similar, entonces las funciones DireccTel, la cual obtiene la dirección y el teléfono de una persona del direc-

```
      Function TamañoDirectorio( D : Direct ) : Integer;
               ensures TamañoDirectorio = Dimensión( D )
         end;

   End Direct
```

Especificación de Interfaz del TDI DIRECT.

Al definir el resto de las operaciones, como se tuvieron que especificar funciones que involucraban los tipos NomPers y DirecTel, se tuvieron que incluir al principio de la característica directorio, las características correspondientes a estos tipos, haciendo una sustitución apropiada de tipos (el tipo Cadena será el nombre del TDI que especificarán estas características, como se verá más adelante). La función NomIguales se especificará más tarde en la característica Nombre-De-Personas.

Ahora bien, generalmente después de haber escrito la especificación de un dato abstracto, es recomendable revisarla y discutirla. Es en esta inspección donde se evalúa la propiedad de Totalidad, mencionada en un principio.
Si se analiza la especificación anterior, se observará que el procedimiento AgregaNomDireccTel no está completamente definido, ya que no dice que hacer si al intentar agregar el nombre de una persona al directorio ésta ya está, cosa que sería muy común que sucediera.

Cuando una especificación como ésta no considera uno o varios casos posibles a suceder, se dice que la especificación no es total. Para que este procedimiento cumpla con esta propiedad, es deseable que contemple el caso de cuando se intenta agregar una persona ya registrada en el directorio.
Para que el procedimiento lo haga así, es necesario agregar en el encabezado del procedimiento (o función si es el caso) la instrucción Signals <variable booleana>, la cual indica que la variable contenida en ella, en este caso definida como Ya-Está-Persona, es booleana. Está, posiblemente se modificará en el procedimiento de acuerdo a las circunstancias, informando al programador lo que sucede cuando se intenta agregar una persona al directorio, que ya está en él.
De esta manera se tiene modificada la especificación de este procedimiento y es la siguiente:

```
    Procedure AgregaNomDirecTel( Var D : Direct;
                                     Np : NomPers; Dt : DirecTel );
                                Signals( Ya-Está-Persona )
              modifies at most  [ D ]
              ensures If ¬Estáen( D, Np ) Then
                                     Dpost = AgregaPers( D, Np,Dt )
                      else
                        Signals( Ya-Está-Persona ) &
                        modifies nothing
         end;
```

Por una razón similar, entonces las funciones DireccTel, la cual obtiene la dirección y el teléfono de una persona del direc-

torio; EliminaPersona, cuyo fin es eliminar del directorio esta misma información y además el nombre de una persona, y el procedimiento ModificaDireccTel, que permite modificar también esta misma información, tampoco cumplen con esta propiedad, porque además de no considerar el caso de cuando una persona no está en el directorio y se desea información de ella, modificarla o borrarla, tampoco consideran el caso cuando se desea hacer esto y el directorio está vacío. Obsérvese que esto dltimo si se especificó en la característica a través de la cladsula Exempts.
En todas estas rutinas es necesario agregar las variables booleanas No-Está-Persona y Directorio-Vacío a través de la instrucción Signals.
La modificación de estas rutinas se muestra a continuación.

```
Function DireccTel( D : Direct; Np : NomPers ) : DirecTel
               Signals( No-Está-Persona, Directorio-Vacío )
        ensures If D = SecreaDirect Then
                     Signals( Directorio-Vacío ) &
                     modifies nothing
                  else
                    If Estéen( D, Np ) Then
                       DireccTel = ObtenDireccTel( D, Np)
                     else
                       Signals( No-Está-Persona ) &
                       modifies nothing
     end;

Procedure EliminaPersona( Var D : Direct; Np : NomPers)
                  Signals( No-Está-Persona, Directorio-Vacío)
          modifies at most [ D ]
          ensures If D = SecreaDirect Then
                       Signals( Directorio-Vacío ) &
                       modifies nothing
                    else
                      If Estéen( D, Np ) Then
                         D post = EliminaPers( D, Np )
                      else
                        Signals( No-Está-Persona ) &
                        modifies nothing
    end;

Procedure ModificaDireccTel( Var D : Direct;
                             Np : NomPers; Dt : DirecTel);
               Signals( No-Está-Persona, Directorio-Vacío )
               modifies at most [ D ]
               ensures If D = SecreaDirect Then
                            Signals( Directorio-Vacío ) &
                            modifies nothing
                         else
                           If Estéen( D, Np ) Then
                              D post = ModifDireccTel( D, Np, Dt )
                            else
                              Signals( No-Está-Persona ) &
                              modifies nothing
    end;
```

Si una especificación de un procedimiento o función no es total, entonces son los programadores quienes están en libertad de escoger el comportamiento de éste, para aquellos casos no especificados. Sin embargo, muchas veces ellos no saben como guiar este comportamiento o de la existencia de casos no especificados, lo cual puede originar ambigüedades en la especificación.

Continuando con la especificación de los TDI NomPers y direcTel, se confirma con el usuario su finalidad, la cual es, respectivamente, registrar los nombres de las personas a aparecer en el directorio, así como sus direcciones y teléfonos, a la vez de verificar aún antes de incorporarlos al directorio, que su nombre, dirección y teléfono estén correctamente escritos con los caracteres válidos que sugiere el usuario: letras, números y los caracteres especiales "#", "-" y ".".

Para el primer TDI también se debe especificar la función booleana NomIguales, la cual dice cuando los nombres de dos personas son iguales, ya que ésta fue asumida por la característica Diccionario. Se especificará primero el TDI NomPers.
Las operaciones que se desean realizar con este TDI son:

a) Crear un objeto que almacene los caracteres del nombre de una persona.

b) Verificar como válidos estos caracteres, antes de agregarlos a este objeto.

c) Saber cuando el nombre de dos personas es el mismo.

La primera operación se especifica de manera similar a la operación especificada para crear o inicializar el TDI Direct, a través de la función Nombre. Esta introduce el tipo Cadena (que es asumido en la característica Directorio como NomPers), el cual es un dato abstracto que almacena los caracteres válidos del nombre de una persona.

La segunda operación tiene como objetivo agregar a la cadena inicializada anteriormente, los caracteres válidos del nombre de una persona y detectar si existe alguno inválido. Esta operación se especificará a través de la siguiente función :

AgregaCarN : Cadena, Caracter => Cadena

La tecera operación tiene como objetivo comparar caracter por caracter los nombres de dos personas, para saber si son iguales o no. Esta operación se especificará a través de la siguiente función:

NomIguales : Cadena, Cadena => Booleano

Antes de mostrar la especificación de este TDI, primero se identificarán los caracteres válidos que señala el usuario a través de una característica, la cual introducirá la función booleana CaracterVálido para identificar estos caracteres. Esta característica será asumida por las características Nombre-De-Personas y Direcc-Tel-De-Personas.

Esta se muestra a continuación:

```
Caracteres-Válidos : Trait
    Imports Iguales-Elementos with [ Caracter  For T ]
    Introduces
       CaracterVálido : Caracter => Booleano
    Constrains CaracterVálido so that
       Caracter partitioned by # == #
       Caracter generated by [ A,B,....Z,0,1,....9,....,
                               #,$,^,[,(,....+,! ]
       for all [ Car : Caracter ]
           CaracterVálido( Car ) = Car == A | Car == B | ... |
                                   Car == Z | Car == 0 |
                                   Car == 1 | ... | Car == 9 |
                                   Car == , | Car == # |
                                   Car == -
```

En esta característica se asume el tipo caracter e importa la característica Iguales-Elementos, para distinguir caracteres válidos de inválidos a través de la función # == #.

La especificación de Interfaz del TDI NomPers, así como de la característica Nombre-De-Personas es entonces la siguiente:

```
Nombre-De-Personas : Trait
   Asumes Caracteres-Válidos
   Imports Iguales-Elementos with [ Caracter For T ]
   Introduces
         Nombre : => Cadena
      AgregaCarN : Cadena, Caracter => Cadena
      NomIguales : Cadena, Cadena =>  Booleano
   Constrains NomIguales so that
      Cadena generated by [ Nombre, AgregaCarN ]
      Cadena partitioned by [ NomIguales ]
      for all [ n1, n2 : Cadena; @, $ : Caracter ]
      NomIguales(AgregaCarN(n1, @), AgregaCarN(n2, $)) =
                               @ = $  &  NomIguales( n1, n2 )
      NomIguales( Nombre, Nombre ) = Cierto
      NomIguales( AgregaCarN( n1, @), Nombre) = Falso .... 1
      NomIguales( Nombre, AgregaCarN( n2, $)) = Falso .... 2
   Implies Converts[ NomIguales ]
```

Especificación de la característica Nombre-De-Personas.

```
Type NomPers Exports IniciaNombre, RegistraNombre,
         MismaPersona
    Based on Sort Cadena From Nombre-De-Personas

    Procedure IniciaNombre( Var Np : NomPers );
            ensures( Np post = Nombre )
      end;
```

```
Procedure RegistraNombre( Var Np : NomPers; @ : Caracter);
                        Signals( Caracter-InVálido )
          modifies at most [ Np ]
          ensures If CaracterVálido( @ ) Then
                      Np post = AgregaCarN( Np, @ )
                  else
                     Signals( Caracter-Inválido )
   end;
Function MismaPersona( Np1, Np2 : NomPers ) : Boolean;
         ensures MismaPersona = NomIguales( Np1, Np2 )
   end;

End NomPers
```

Especificación de Interfaz del TDI NomPers.

Esta característica también importa Iguales-Elementos, con el fin de saber si los caracteres de los nombres de dos personas son iguales o no.

Por último, para la especificación del TDI DirecTel, las operaciones a realizar en él son las mismas a las dos primeras realizadas para el TDI anterior. Por lo tanto, la especificación de este TDI es muy similar a aquélla. El objeto que almacenará como caracteres la dirección y teléfono de una persona se llamará también Cadena.

El nombre de la función que indica la inicialización del TDI Cadena, será llamada DirTel y el de la función que especifica la operación de agregar la dirección y el teléfono a este tipo, será llamada AgregaCarDT.

La especificación de este TDI, así como de la característica correspondiente Direcc-Tel-De-Personas se muestra en seguida:

```
Direcc-Tel-De-Personas : Trait
    Imports Caracteres-Válidos
    Introduces
            DirTel : => Cadena
       AgregaCarDt : Cadena, Car => Cadena
     Constrains Cadena so that
        Cadena generated by [ DirTel, AgregaCarDt ]
```

Especificación de la característica Direcc-Tel-De-Personas.

```
Type DirecTel Exports IniciaDirecTel RegistraDirecTel
     Based on Sort Cadena From Direcc-Tel-De-Personas

     Procedure IniciaDirecTel( Var Dt : DirecTel );
               modifies at most [ Dt ]
               ensures( Dt post = DirTel )
        end;
```

```
Procedure RegistraDirecTel( Var Dt : DirecTel;
                                 @ : Caracter );
                            Signals( Caracter-Inválido )
              modifies at most [ Dt ]
              ensures If CaracterVálido( @ ) Then
                           Dtpost = AgregaCarDt( Dt, @ )
                      else
                        Signals( Caracter-Inválido )
      end;

   End DirecTel
```

Especificación de Interfaz del TDI Directel.

Con la especificación de este último TDI ha concluido por completo la especificación, en el lenguaje Larch, de un directorio de nombres, direcciones y de teléfonos de personas de una ciudad. La especificación completa se muestra en el apéndice B.

El siguiente paso es mostrarla al usuario y explicarle detenidamente que es lo que se va haciendo con respecto a los TDI considerados para satisfacer sus operaciones deseadas con ellos. Finalmente, si no hay ningún error o mal entendido, se implanta en el lenguaje de programación Pascal.

A través de este ejemplo, se ha intentado mostrar la manera de construir una especificación de un tipo abstracto utilizando el lenguaje de especificación Larch/Pascal. Obsérvese como en ella se muestra claramente las operaciones a realizar en un directorio telefónico sugeridas por el usuario. Estas significan lo mismo que estableció el usuario; simplemente se han traducido sus requerimientos en un lenguaje más formal que el natural, para así eliminar ambigüedades en ellos y hacer la implantación del programa que los ajecutará.

En la especificación no se dice que algoritmo utilizar para realizar cierta operación, por ejemplo, para buscar en el directorio el nombre de una persona. Tampoco se menciona ni se sugiere alguna estructura de datos a utilizar. Estos aspectos son resueltos al momento de implantar la especificación.

C A P I T U L O   T E R C E R O.

El Lenguaje de Especificación Formal OBJ.

3.1 Generalidades.

OBJ es un lenguaje de especificación formal y es además, un lenguaje de programación. El hecho que le permite tener además esta última cualidad, es debido a que por una parte, además de permitir escribir y probar especificaciones algebraicas de programas, permite también la ejecución de estas especificaciones tal como si fueran programas.

OBJ surge en 1977 con la versión OBJ-0, como una versión formal a la idea planteada por Joseph A. Goguen en la UCLA en 1974, consistente en implantar en un lenguaje, reglas de reescritura para abstraer las caracteríísticas y propiedades de tipos de datos abstractos, a partir de álgebras y que además puedan ser probadas. Mas tarde, Joseph J. Tardo hizo la primera implantación en una computadora IBM 360/91 de la UCLA en 1976.

Después de esta primera versión, han surgido otras similares. Actualmente está siendo implantada parcialmente una versión llamada OBJ-T, y empezandose a planear y a diseñar una nueva versión llamada OBJ-1. Aquí se describirán algunas características de la primera versión.

OBJ presenta ciertas características, las cuales permiten construir especificaciones correctas de programas y sistemas. Estas son:

- Permite construir grandes especificaciones a partir de especificaciones mas pequeñas, permitiendo también establecer una comunicación entre ellas;

- Permite probar bloques de especificación de manera independiente así como su comunicación, a través de la ejecución de casos de prueba;

- Destaca la representación de las especificaciones a través de una notación flexible y que puede ser definida por el usuario;

- Permite la manipulación de condiciones de error de manera sistemática;
- Proporciona un conjunto de cladsulas de verificación sintácticas y semánticas que

permiten realizar inspecciones en la especificación.

Por otra parte, sistemas importantes e interesantes han sido desarrollados utilizando el lenguaje OBJ, lo cual respalda lo poderoso de este lenguaje. D. Harm en 1977 especificó un lenguaje de programación simple con expresiones, iteradores, 'go-to's' y asignadores, corriendo algunos programas en el. T. Kaufman especificó y probó en ese mismo año un editor simple de texto. Goguen, Tardo, M. Zamfir y N. Williamson especificaron una estructura de un archivo de índices, incluyendo las operaciones de ordenación y recuperación, en 1978.

Lo anterior muestra que es posible que sistemas complejos, tales como sistemas operativos, editores y sistemas de base de datos, puedan ser especificados algebraicamente y entonces simulados en OBJ. Hay dos maneras en que OBJ ayudaría a establecer la correctividad de estas especificaciones: proporcionando al usuario cladsulas e instrucciones para permitirle definir condiciones de error y realizando ejecuciones de las especificaciones a través de casos de prueba.

3.2 Descripción del Lenguaje.

Por la manera en que son definidas y construidas las especificaciones, el lenguaje de especificación OBJ se asemeja en mucho a los lenguajes de programación LISP y APL, pues permite la definición de funciones y la evaluación de ellas a través de expresiones. Sin embargo, a diferencia de aquellos lenguajes, las funciones que define OBJ se establecen a través de módulos o bloques. Ellas son definidas por medio de ecuaciones algebraicas (llamados axiomas, como en el lenguje Larch). Estas funciones no denotan procedimientos, sino mas bien funciones algebraicas con dominio y rango definido (como en el lenguaje Larch). A pesar de ser OBJ también un lenguaje de programación, en él no existen llamados de procedimientos y funciones, ni asignaciones de variables, efectos laterales, 'goto's', ni en general ninguna estructura de control como while's, repeat's, etc. Aunque algunas de estas construcciones e instrucciones, como efectos laterales y asignación de variables pueden ser simuladas y especificadas en OBJ.

Los módulos a través de los cuales se especifican las propiedades y características de un dato abstracto, o aún de una operación algebraica, por medio de axiomas, se llaman 'Objetos' (similares a las Características definidas en el lenguaje Larch).

OBJ-O proporciona tres objetos ya incorporados que pueden utilizarse para construir especificaciones. Estos son: INT para manejo de enteros, BOOL para operaciones booleanas y el objeto ID para la comparación de identificadores. Las funciones que contiene el objeto INT son: INC para el incremento de un número entero (p. ej. INC(5) = 6); DEC para el decremento de un número entero (p. ej. DEC(8) = 7); los operadores infijos de relación '<', '>',

'>=', '<=', '='; los operadores infijos aritméticos '=', '+', '*', '-', '(', ')' y el operador negativo unario '-' y así como el conjunto de números enteros que son constantes.
El objeto BOOL, proporciona las funciones de los operadores infijos '+' o '|' para disyunción, '*' o '&' para conjunción, el operador prefijo '-' o ' ' para negación y los operadores infijos '=>' y '<=>' de implicación y 'si y solo si' respectivamente, así como las constantes booleanas 'T' (True) y 'F' (False) de verdadero y falso, respectivamente.
Y el objeto ID tiene el operador infijo '=', cuyo propósito es saber si sus dos operandos (identificadores o constantes) son iguales. Estos deben ser inhibidos por un apostrofe (como lo hace LISP, p. ej. 'Atomo).

Para especificar un objeto, se comienza por definir el nombre del objeto precedido por la palabra reservada OBJ (u OBJECT, objeto) y se finaliza la especificación escribiendo la palabra también reservada JBO (o TCEJBO, otejbo), para indicarle al intérprete de OBJ que se ha definido un módulo de especificación.
OBJ está implantado en el lenguaje de programación LISP, como se explicará más adelante.

Un objeto está dividido en cinco secciones: 1) un encabezado en donde se especifica el nombre del objeto; 2) una sección de declaración de tipos (SORTS, Tipos); 3) una sección de declaración de funciones (OPS, Operadores); 4) una sección de declaración de variables (VARS, Variables) y 5) una más de definición de ecuaciones (EQNS, Ecuaciones). El siguiente esquema muestra la estructura de un objeto:

```
1 { OBJ <Nombre del objeto>
2 {     SORTS <tipos introducidos> / <tipos asumidos>
3 {     OK-OPS ....
  {     ERR-OPS ....
4 {     VARS ....
  {     OK-EQNS ....
5 {     EQNS ....
  {     ERR-EQNS ....
    JBO
```

La instrucción 'SORTS ....' (en la sección 2) describe los nombres de los tipos involucrados en el objeto, la linea diagonal ('/') separa los tipos introducidos por el objeto de los tipos asumidos (ya declarados en otros objetos). Esto significa que OBJ permite ir construyendo objetos mas complicados a partir de otros más simples.

3.2.1 Tipos de Funciones en OBJ.

Existen dos clases de funciones en OBJ (especificadas en la sección 3). Las funciones correctas, usadas en situaciones normales o correctas (OK-OPS) y las funciones errones, usadas en situa

ciones excepcionales, para recuperar situaciones y para conformar los posibles mensajes de error (ERR-OPS).

Para la definición de las funciones, se usan expresiones. Una expresión es la llamada de una función constante o de una función con sus argumentos, los cuales pueden también ser expresiones (expresiones complejas). Cada expresión tiene un tipo, que es el tipo de la función. Una expresión es correcta o erronea. Una expresión cuya función es erronea es una expresión erronea, así como también si alguno de sus argumentos es una expresión erronea. Y de manera similar, una expresión es correcta si la función que hace referencia es correcta o si tiene como argumentos funciones y éstas resultan ser correctas.

Una declaración de una función muestra los tipos de su dominio (argumentos o aridad) y su rango, es decir, el tipo de valor de la función, así como su forma sintáctica (como en el lenguaje Larch). Un ejemplo de una declaración de una función cuyo tipo es INT y cuya aridad es (INT INT) es la siguiente:

G : INT INT -> INT

Una función que no tiene dominio, es una función constante del tipo declarado, por ejemplo:

CREA : -> PILA

La sintaxis de una función está determinada por la forma en que se especifica, la cual está indicada a la izquierda de los dos puntos ":". Asi, las funciones declaradas de la forma anterior (<Identificador> : <Dominio> -> <rango>), se asume una sintaxis prefija, figurando los argumentos entre paréntesis y separados por comas. Por lo tanto es válida la siguiente expresión:

G (2, G (4, 5))

OBJ permite modificar la sintaxis de una función de acuerdo a las circunstancias para un mejor significado de la función. Por ejemplo, la función:

METE _ EN _ : INT PILA -> PILA

puede ser usada para evaluar la expresión:

METE 5 EN CREA, en lugar de la función:

_ _ : INT PILA -> PILA para evaluar la misma expresión:

5 Crea, la cual no es muy explicativa.

Una operación importante que figura en ciertos lenguajes de especificación es la coerción entre tipos, es decir, transformar un

identificador de un tipo a otro. Por ejemplo, la función:

_ : S1 -> S2

indica que todos los identificadores de tipo S1 son también de ti
po S2, es decir, S1 va a ser un subtipo de S2.

Un ejemplo claro del uso de una coerción es la siguiente. Supongamos que se define la operación división entre números reales, a través de la siguiente función:

_ / _ : REAL REAL -> REAL

y que se define la operación "N módulo M" entre enteros, con la siguiente función:

_ MOD _ : INT INT -> INT

entonces, si se tuviera que evaluar la expresión:

8 / (5 MOD 2)

al momento de que el intérprete de OBJ quisiera efectuarla, desplegaría un mensaje diciendo que no se puede efectuar la división, porque existe una incopatibilidad entre tipos, ya que no se definió la división de reales entre enteros. Por lo tanto, para evi
tar esto, es necesario agregar la coerción:

_ : INT -> REAL

y así, ahora el intérprete al terminar de efectuar la operación (5 MOD 2), con esta coerción, el resultado (1) es real y podra realizarse la división.

Otra característica importante de OBJ es la de permitir declarar funciones con la misma forma sintáctica, pero con distinto do
minio. A este tipo de funciones se les llama frecuentemente funciones "sobrecargadas". Por ejemplo, OBJ-0 proporciona algunas de estas funciones, las cuales están ya incorporadas en él:

_ + _ : BOOL BOOL -> BOOL

_ + _ : INT INT -> INT

Aunque este tipo de funciones podría originar confusiones en una especificación misma, OBJ esta diseñado para manipularlas correctamente, ya que en una cierta situación, OBJ identifica el ti
po del argumento de cada función y busca en las funciones correspondientes, aquella que concuerde con el tipo.

Estos dos últimos conceptos, la coerción entre tipos y las fun
ciones sobrecargadas, dan a OBJ un gran poderio sobre otros lenguajes similares para especificar datos abstractos y operaciones algebraicas, ya que permiten soportar una sintaxis flexible y útil.

3.2.2 Variables y Ecuaciones en OBJ.

Después de haber declarado en un objeto las funciones utilizadas, se declara la sección de definición de variables (y su tipo) auxiliares para la descripción de los axiomas de las funciones, después de la palabra reservada 'VARS' (sección 4).
Por ejemplo:

```
VARS
        I, J : INT
      R1, R2 : REAL
           A : LIST
```

La semántica de las funciones se hace a través de ecuaciones (axiomas), las cuales se indican entre paréntesis y a continuación de la sección anterior (VARS). Una ecuación es una igualdad entre expresiones (constantes o variables) del mismo tipo. La expresión del lado derecho, que puede ser constante o variable, es el valor de la expresión del lado izquierdo de la ecuación.
Por ejemplo:

```
( G( 0, J ) = J )  ..... (1)
```

En esta sección existen dos tipos de ecuaciones, las ecuaciones correstas y las erroneas, las cuales van respectivamente en las secciones 'OK-EQNS' y 'ERR-EQNS'. Las ecuaciones correctas son aquéllas que contienen expresiones correctas, y las ecuaciones erroneas son aquéllas que contienen del lado derecho expresiones erroneas. Existe una tercera subsección aquí llamada 'EQNS', en la cual se declaran aquellas ecuaciones las cuales pueden ser correctas o erroneas, dependiendo de sus argumentos.

OBJ proporciona ciertas proposiciones condicionales para permitir la definición de funciones condicionales. Una función condicional, es aquélla cuyo valor está condicionado por una expresión que debe aparecer a la derecha de la popocisión 'IF'. Esta debe ser tal que pueda ser evaluada a través de las funciones de los objetos incorporados BOOL o ID, por ejemplo:

```
( G(I, J) = J   IF   ( I = 0 ) )
```

Nótese que èsta es una forma condicional de escribir la ecuación anterior (1).

En seguida se muestra la especificación de la función Máximo Comun Divisor (MCD), para ejemplificar las ideas antes mostradas:

```
OBJ MAXIMO-COMUN-DE-DIVISOR
    SORTS    / INT BOOL
    OK-OPS
       MCD : INT INT -> INT
       GCD : INT INT -> INT
       RES : INT INT -> INT
    ERR-OPS
       ARGUMENTOS-NEGATIVOS : -> INT
```

```
VARS
   A, B : INT
OK-EQNS
   ( MCD( A, B ) = IF ( A < B ) THEN
                      GCD( B, A )
                   ELSE GCD( A, B ) ) FI
   ( GCD( A, B ) = IF ( RES( A, B ) <> 0 ) THEN
                      GCD( B, RES( A, B ) )
                   ELSE B FI
   ( RES( A, B ) = IF ( A >= B ) THEN
                      RES( A - B, B )
                   ELSE A FI )
ERR-EQNS
   ( MCD( A, B ) = ARGUMENTOS-NEGATIVOS IF
                             ( A < 0 | B < 0 ) )

JOB
```

Como puede verse, este objeto no introduce ningún tipo nuevo, sólo invoca los tipos ya implantados por OBJ-0 (INT, BOOL), y hace uso de la estructura condicional "IF-THEN-ELSE-FI" (IF-THEN-ELSE), implantada en OBJ-0, ya que en él se usan sólo operaciones de números enteros y booleanas. Obsérvese cómo la función "ARGUMENTOS-NEGATIVOS" tiene efecto cuando se intenta calcular el MCD de algún número negativo.

La manera de evaluar una expresión en un objeto, como en el MCD, es la siguiente:

RUN MCD(18, 36) NUR

de esta manera se pide al intérprete que calcule el MCD de los nu meros 18 y 36. El intérprete a continuación responderá:

AS INT: 18

(como entero : 18) debido a que el tipo de la última expresión evaluada es entero. Una forma alternativa es encerrar la expresión entre paréntesis:

(MCD(18, 36))

Si se deseara evaluar la expresión: MCD (8, -55), se obtendría como respuesta esperada:

AS INT: >> ERROR >> ARGUMENTOS-NEGATIVOS

Y si, por equivocación se intenta evaluar la expresión:

RUN MCF(6, 97) NUR

OBJ respondería:

?WARNING: REWRITTEN, EXPRESSION CANNOT BE DEPARSED ..... (2)

la cual significa que la expresión no puede ser analizada porque no exite alguna función especificada en el objeto como MCF.

La forma en que OBJ evalúa una expresión, es decir, obtiene el valor de la expresión, es la siguiente. Primeramente el intérprete debe encontrar una ecuación que unifique la expresión dada, es decir, debe encontrar alguna ecuación cuyo identificador de la función en el lado izquierdo, sea igual al de la expresión dada y además cuyos argumentos coincidan, o sea sean del mismo tipo.
Esta evaluación se hace primero intentando la unificación con las ecuaciones erroneas y luego con las ecuaciones correctas.
En caso de no encontrar esta ecuación, intenta buscarla (el intérprete) en los objetos de los tipos asumidos en el mismo archivo fuente o en los archivos u objetos asumidos con la clausula GET (que se explicará mas adelante). Y en caso de tampoco encontrarla aquí, entonces se desplegará un mensaje similar al mostrado en (2).
Una vez encontrada esta ecuación que unifica la expresión a evaluar, comprueba si esta ecuación es condicional. Si lo es, evalúa la condición y si es cierta, sustituye los respectivos argumentos de la ecuación dada en el lado derecho de la ecuación. De ser falsa la condición, busca otra ecuación que unifique.
Pero si no es una ecuación condicional, simplemente sustituye los argumentos respectivos de la ecuación dada en el lado derecho de la ecuación.
Una vez sustituidos estos argumentos en ambos casos, se efectúan las operaciones o si es el caso, las nuevas evaluaciones correspondientes.

Cuando se trata de evaluar una expresión compleja, por ejemplo:

```
( MCD( 8, MCD( 21, 76 ) ) - MCD( 5, 117 )
```

OBJ reduce primero las expresiones, es decir, evalúa (o si es el caso, reduce), los argumentos (subexpresiones), en este caso la expresión MCD(21 76) . O sea, la evaluación de expresiones complejas se realiza de adentro hacia afuera y de izquierda a derecha, hasta agotar todas las expresiones correctas (subexpresiones) y la expresión misma que las contiene.

Si al evaluar una expresión compleja, se encuentra que en una subexpresión se tiene que aplicar una ecuación erronea, entonces el intérprete dejará de seguir evaluando esta expresión, y desplegará entonces una función erronea como argumento de la siguiente expresión (o subexpresión) que se iba a evaluar. El siguiente conjunto de evaluaciones, muestra los pasos que seguiría el intérprete para evaluar la expresión siguiente (entre llaves se establecen los comentarios referentes a cada paso de evaluación):

```
MCD( 25, MCD( 3 - 7, 9 ) ) * MCD( 758, 242 )

   MCD( 25, MCD( -4, 9 ) ) * MCD( 758, 242 )
      {Se realiza la operación 3 - 7}
```

```
MCD( 25, ARGUMENTOS-NEGATIVOS) * MCD( 758, 242 )
   {Se evalúa la expresión MCD( -4, 9) cuyo valor es
    ARGUMENTOS-NEGATIVOS}

MCD( 25, ARGUMENTOS-NEGATIVOS) * GCD( 758, 242 )
   {Se evalúa la expresión MCD( 758, 242 ) cuyo fin es
    ordenarlos de mayor a menor}

MCD( 25, ARGUMENTOS-NEGATIVOS) * GCD( 242, 32 )
   {Se evalúla expresión GCD( 758, 242 ) }

MCD( 25, ARGUMENTOS-NEGATIVOS) * GCD( 32, 18 )
   {Se evalúa la expresión GCD( 242, 32 ) }

MCD( 25, ARGUMENTOS-NEGATIVOS) * GCD( 18, 14 )
   {Se evalúa la expresión GCD( 32, 18 ) }

MCD( 25, ARGUMENTOS-NEGATIVOS) * GCD( 14, 4 )
   {Se evalúa la expresión GCD( 18, 14 ) }

MCD( 25, ARGUMENTOS-NEGATIVOS) * GCD( 4, 2 )
   {Se evalúa la expresión GCD( 14, 4 ) }

MCD( 25, ARGUMENTOS-NEGATIVOS) * 2
   {Se evalúla expresión GCD( 4, 2 ) obteniendo como
    resultado 2}
```

y entonces, al no haber mas expresiones correstas se obtiene como resultado:

```
AS INT: >> ERROR >>  MCD( 25, ARGUMENTOS-NEGATIVOS) * 2
```

La ventaja que representa el que el intérprete dé como resultado de la aplicación de una o varias funciones erroneas, una expresión o expresiones conteniendo funciones erroneas, en lugar de desplegar una serie de errores que es lo que normalmente un compilador haría, es que permite hacer depuraciones en la especificación misma y hallar la razón de tal mensaje erroneo, comparando esta expresión con la expresión que se evaluó.

3.3 Algunas Características Relevantes de OBJ.

En esta parte se ilustrarán algunas de las características más distintivas de OBJ, como son: mensajes de error, funciones escondidas, coerciones entre tipos, funciones asociativas y memoria de resultados.

Dentro del lenguaje OBJ, existen algunas instrucciones que le indican al intérprete que ciertas funciones tendrán algunas características particulares. Una de estas instrucciones es "ASSOCIATIVE" (Asociar), la cual al aparecer del lado izquierdo de una función y entre paréntesis, le indica al intérprete que los argumentos de esta función, obedecerán la propiedad asociativa cada vez

que sea llamada. Esta instrucción se muestra en el siguiente objeto, cuyo fin es agrupar un conjunto de números en una lista.

```
OBJ  LISTA_DE_ENTEROS
     SORTS  LIST / INT
     OK-OPS
         _ _ :  LIST LIST -> LIST (ASSOCIATIVE)
           _ :  INT -> LIST
JBO
```

Obsérvese cómo las dos funciones no tienen una sintaxis definida (entrefija, postfija o antefija), ambas se aplican ante cualquier circunstancia que lo requiera el intérprete. La segunda ecuación establece una coerción de un tipo entero a un tipo lista. Si se hace la siguiente evaluación:

```
RUN  1 2 3  NUR
```

OBJ responderá:

```
AS  LIST:  (1, 2, 3)
```

Los paréntesis indican que el elemento constan de una lista. También es válido hacer la siguiente evaluación:

```
AS LIST RUN  ( 1, 2, 3 ), ( 4, 5 ), ( 6, 7, 8 )  NUR
```

Aquí se pide al intérprete que evalúe la expresión como del tipo especificado (List). En este caso se obtiene:

```
AS LIST: ( ( 1, 2, 3 ), ( 4, 5 ), ( 6, 7, 8 ) )
```

Supóngase ahora que se desea obtener la cabeza (el último elemento que entro de una lista) y el resto de una lista de números. La especificación siguiente muestra la manera de especificar estas operaciones:

```
OBJ  CABEZA_Y_RESTO
     SORTS    / LIST INT
     OK-OPS
        CABEZA _ : LIST -> INT
         RESTO _ : LIST -> LIST
     ERR-OPS
         NINGUN-RESTO : -> LIST
     VAR
          I : INT
          L : LIST
     OK-EQNS
        AS INT ( CABEZA( L, I ) = I )
        ( RESTO( L, I ) = L )
        AS INT (  CABEZA( I ) = I )
     ERR-EQNS
         ( RESTO ( I ) = NINGUN-RESTO )
JBO
```

Este objeto muestra la manera de manipular una coerción entre tipos de una función. La frase 'AS INT', o en general 'AS <Tipo>'

precedida en una ecuación, indica que el tipo resultante debe ser de este tipo, ya sea que se tenga que hacer una coerción o evitán
dola.

Debe observarse aquí que el orden de los objetos especificados es importante; primero se debe especificar el objeto o los obje- tos que introducen los tipos que luego otro u otros objetos asumi rán en el mismo archivo. Un objeto que asuma el tipo o los tipos de otro objeto puede hacer uso de las funciones definidas en ese objeto.

Supóngase ahora que se desea hacer la siguiente evaluación:

RUN CABEZA(2, 3, 4 + CABEZA(6)) NUR

el intérprete responderá entonces:

AS INT: 10

y si, se quisiera hacer la siguiente evaluación:

RUN CABEZA(2, 3, 4 + CABEZA(RESTO(6))) NUR

se obtendría como respuesta obvia:

AS INT: >> ERROR >> (CABEZA(2, 3, 4 + CABEZA(NINGUN-RESTO)))

Unos ejemplos más:

RUN RESTO(5, 8, 15, 11) NUR
AS LIST: (5, 8, 15)

RUN RESTO(CABEZA(9, 10, 11)) NUR
AS INT: RESTO(NINGUN-RESTO)

Si en este objeto (y en cualquier otro que presente una situa- ción similar) se eliminan las proposiciones 'AS INT' precedidas de las funciones, y si se deseara hacer la primera evaluación an- terior, se obtendría como respuesta:

WHICH SORT WOULD YOU LIKE IT ?

que pregunta de que tipo se desea analizar la función, ya que de bido a la coerción existente en el objeto asumido 'LISTA_DE_ENTE- ROS', la función CABEZA tiene dos tipos (INT y LIST). Si existie- ra duda sobre el tipo de la función, se puede responder ahora:

> HOW ?   (Como ?) a lo cual el intérprete responderá ahora:

CAN BE PARSED AS ONE OF (INT LIST)   (puede ser analizada como una función de tipo INT o LIST) pudiéndose responder:

> AS INT   y obteniéndose como respuesta:
AS INT: 10   o respondiendo:
> AS LIST y se obtiene como respuesta:

AS LIST: 10

Pero si se desea evitar este diálogo, las expresiones pueden evaluarse del tipo deseado si ésta precede a la palabra RUN (p. ej. AS INT RUN <función a evaluar>). Por ejemplo:

```
        AS LIST RUN  CABEZA( 15, 20, 50, 13 )  NUR
           AS LIST: ( 13 )
```

Para poder asumir objetos y tipos de otros archivos fuente, o todo un archivo de objetos a un objeto, se debe declarar la claúsula GET <Nombre del Archivo> que asume los objetos del archivo especificado, o la clausula GET <Nombre del Objeto> From <Nombre del Archivo> si se desea asumir un objeto en particular de un archivo. Ambas deben ser escritas inmediatamente después de especificar el nombre del objeto que las empleará.

Existe un tipo de funciones muy importante y útil en algunos lenguajes de especificación, llamadas funciones escondidas (Hidden, escondidas) o auxiliares. Estas funciones, aunque no son estrictamente necesarias, tienen como propósito simplificar y clarificar una especificación. Además, permiten realizar una depuración mas fácil y eficiente de la especificación. Estas funciones forman parte de la especificación de la abstracción de un tipo abstracto, pero no de la abstracción misma. Pueden no aparecer de manera directa en el momento de implantar la especificación, pero sí deben ser tomadas en cuenta.

Para declarar una función escondida en OBJ, debe escribirse la instrucción 'HIDDEN' entre peréntesis al lado derecho de la función que se desea tenga esta propiedad. La desventaja que tienen estas funciones en OBJ, es que éstas sólo pueden usarse en el mismo objeto, pero no por otros.

El siguiente objeto muestra el uso de estas funciónes y algunas otras características de un objeto. Este especifica las operaciones deseables a realizar en una pila de números, la cual se puede recorrer en ambas direcciones, haciendo uso de un sólo apuntador a la pila:

```
OBJ  RECORRIDO_DE_UNA_PILA
     SORTS  PILA, HPILA  /  INT BOOL
     OK-OPS

           CREA : -> PILA
     *** Crea la pila de números ***
           PUSH : PILA INT -> PILA
     *** Agrega un número a la pila y coloca el apuntador a
         este número ***
     -      POP : PILA -> PILA
     *** Saca un número de la pila y coloca el apuntador al
         número de abajo. El apuntador debe estar en el tope
         de la pila ***
          ABAJO : PILA -> PILA
     *** Recorre el apuntador un lugar abajo ***
         ARRIBA : PILA -> PILA
```

```
*** Recorre el apuntador un lugar arriba ***
       LEE : PILA -> INT
*** Lee el número al que señala el apuntador ***
       _ _ : HPILA INT -> PILA
*** Coerción ***
      HPUSH : HPILA INT -> HPILA  (HIDDEN)
*** Coerción ***
      PROF_ : HPILA -> INT  (HIDDEN)
*** Cuenta el número de elementos que contiene la
    pila ***
       BASE : -> HPILA   (HIDDEN)
*** Inicializa la pila con cero elementos ***
ERR-OPS

         PILA-VACIA : -> PILA
   *** POP cuando la pila está vacía ***
    NINGUN-ELEMENTO : -> INT
   *** Lee cuando la pila está vacía ***
      APUNT-NO-TOPE : -> PILA
   *** Push y Pop cuando el apuntador no señala al tope
       de la pila ***
      APUNT-EN-BASE : -> PILA
   *** Abajo cuando se está en la base de la pila ***
      APUNT-EN-TOPE : -> PILA
   *** Arriba cuando se está en el tope de la pila ***
VARS

    I, J, K : INT
         HS : HPILA
OK-EQNS

   ( CREA = BASE 0 )
   ( LEE( HPUSH( HS, I ) J ) = I IF J = INC( PROF HS ) )
   ( = LEE( HS, J ) IF NOT (J = INC( PROF HS ) )
   ( PUSH( ( HS J ) k ) = ( HPUSH( HS, K ), INC( J ) )
                          IF  J = ( PROF HS ) )
   ( PROF BASE = 0 )
   ( PROF( HPUSH( HS, J ) ) = INC( PROF HS)
   ( ABAJO( HS, J ) = ( HS, DEC( J ) )  IF J > 0 )
   ( ARRIBA( HS, J ) = ( HS, INC( J ) )
                      IF ( INC( J ) <= PROF HS ) )
   ( POP( HPUSH( HS K ) J ) = ( HS DEC( J ) )
                             IF J = INC( PROF HS )
ERR-EQNS
    ( LEE( BASE 0 ) = NINGUN-ELEMENTO )
    ( POP( BASE 0 ) = PILA-VACIA )
    ( PUSH( HS J, K ) = APUNT-NO-TOPE
                        IF NOT ( J = PROF HS ) )
    ( ABAJO( HS 0 ) = APUNT-EN-BASE )
    ( ARRIBA( HS J ) = APUNT-EN-TOPE IF
                       ( INC( J ) > PROF HS ) )
    ( POP( HS J ) = APUNT-NO-TOPE IF NOT ( J = PROF HS ))
JBO
```

Este objeto muestra como documentar una especificación en OBJ;

empleando tres asteriscos para delimitar un comentario en la especificación. Muestra también la forma de indicar que una función tiene una posible respuesta entre dos posibles, dependiendo de una ecuación condicional. Las ecuaciones que empiezan con el signo '=' a la izquierda y comprendidas entre peréntesis, indican que es otra posible respuesta de la función anterior, si no se cumple la ecuación condicional, es una forma abreviada de la proposición condicional IF-THEN-ELSE-FI.

En esta especificación, las funciones escondidas 'BASE' y 'PROF' sirven, respectivamente, para establecer una indexación desde el inicio de cada uno de los elementos que contendrá la pila y para saber la posición del apuntador, para así entender cómo la pila es afectada cuando se realizan operaciones en ella.

Las siguientes evaluaciones muestran la manera en que trabaja la pila, así como los posibles mensajes que emitirá el intérprete al encontrar algún error:

```
RUN LEE( PUSH( PUSH( PUSH( CREA, 1 ), 2 ), 3 ) )   NUR
    AS INT: 3

RUN LEE( ABAJO( PUSH( PUSH( PUSH( CREA, 1 ), 2 ), 3 ) ) )  NUR
    AS INT: 2

RUN LEE( POP( ABAJO( PUSH( PUSH( PUSH( CREA, 1 ), 2), 3 ) ) ) NUR
    AS INT:  >> ERROR >>  LEE( APUNT-NO-TOPE )

RUN HPUSH( 1, BOTTOM )  NUR
    ?WARNING:  REWRITTEN,  EXPRESSION  CANNOT  BE  DEPARSED
```

(Porque las funciones escondidas no se pueden evaluar)

```
RUN PUSH( ABAJO( PUSH( CREA, 1 ), 2 )  NUR
    AS PILA:  >> ERROR >>  APUNT-NO-TOPE

RUN ABAJO( PUSH( POP( PUSH( PUSH( PUSH( CREA, 5 ), 4), 6),
          7 ) ) ) NUR
    AS PILA: 5 4 7

RUN LEE( PUSH( 1, CREA ) )
    ?WARNING:  REWRITTEN,  EXPRESSION  CANNOT  BE  DEPARSED
```

(Porque no se definió alguna función cuyo primer argumento fuera de tipo Entero y segundo de tipo pila)

Frecuentemente es deseable que en una especificación de un dato abstracto, o de una operación algebraica, OBJ 'recuerde' o preserve los resultados parciales de una o varias evaluaciones, es decir, que el intérprete cree un area reservada de memoria para almacenar estos resultados y que los actualice cada vez que se ejecute una nueva evaluación. Por ejemplo, en el objeto anterior, sería deseable manipular una Pila de números aplicando varias operaciones en él, sin perder en cada evaluación su contenido.
Para lograr esta facilidad, OBJ proporciona una instrucción llama

da "PERMUTING" (permutar), la cual seguida entre perénteşis de una función, indica que el intérprete almacenará en memoria el resultado de la evaluación cuando sea aplicada esta función y lo actualizará cada vez que sea aplicada de nuevo esta función.
Una forma alternativa de esta instrucción, es el modo de evaluación con memoria, el cual se ejecuta al evaluar una expresión y funciona como si todas las funciones del objeto fueran declaradas con el atributo "PERMUTING". Para lograr esta forma de evaluación, en lugar de evaluar una expresión con la instrucción normal "RUN", se evalúa con la instrucción "RUM".

En seguida se muestran algunas evaluaciones utilizando la forma de evaluación con memoria, utilizando el objeto anterior como referencia.

```
RUM LEE( ABAJO( PUSH( PUSH( PUSH( CREA 1 ), 9 ), 15 ) ) )  MUR
    AS INT: 9

RUM  PUSH( ARRIBA( PILA ), 25 )  MUR
     AS PILA: 1 9 15 25

RUM  LEE( ABAJO( ABAJO( POP( PILA ) ) ) ) MUR
     AS INT: 1

RUM  LEE( PUSH( PUSH( CREA, 8 ), 22 ) )  MUR
     AS INT: 22
```

La pila cuyos elementos eran 1, 9, 15 y 25 fue totalmente borrada al ejecutar ahora esta última evaluación.

3.4 Implantación de OBJ.

En esta parte del capítulo, se describirá brevemente los mecanismos y componentes que permiten a OBJ-0, la facilidad de leer una especificación y evaluar las expresiones que se derivan de ella.

OBJ-0 está implantado en el lenguaje de programación LISP. Los principales componentes de la implantación de OBJ-0 como lenguaje de programación son: un analizador no determinístico, un evaluador de expresiones, un intérprete de objetos de entrada y algunos archivos de funciones de entrada y salida.
La finalidad que tiene el analizador, es traducir las expresiones de entrada de un objeto (que constituyen cadenas de átomos de LISP), a representaciones de expresiones de LISP. Se encarga también de buscar dentro del objeto declarado, aquella expresión que coincida en aridad y tipos con la expresión a evaluar, pudiendo de esta manera detectar expresiones ambiguas definidas por el usuario.

El intérprete de objetos de entrada es invocado al momento de encontrarse la palabra "OBJ" (u OBJECT). En seguida lee todo el contenido del objeto hasta encontrar la palabra "JBO" (o TCEJBO)

o la intrucción de fin de archivo. Luego intenta interpretar este "registro lógico" como un objeto. Si por alguna razón no puede interpretar completamente el objeto, desplegará un mensaje (Warning) y, procedera a leer el siguiente objeto, si lo hay.

El mecanismo de evaluación de expresiones, es inicializado cuando se escriben las instrucciones "RUN", "AS", "RUM" o cuando un peréntesis izquierdo es encontrado. En seguida se leen todos los símbolos siguientes, hasta encontrar el respectivo símbolo terminal "NUR", "MUR" o peréntesis derecho y si entonces, el análisis de esta expresión resulta ser correcto, se evalúa. Cuando se usa la opción "RUM" para evaluar alguna expresión se alerta al evaluador de expresiones a mantener en memoria los resultados de evaluaciones de expresiones anteriores.

Una expresión que contenga funciones escondidas, es tratada como una expresión no analizable durante una evaluación normal. Si una expresión, al evaluarla, contiene funciones escondidas, el intérprete desplegará el siguiente mensaje:

?WARNING: REWRITTEN, EXPRESSION CANNOT BE DEPARSED

3.5 Aplicando el Lenguaje de Especificación Formal OBJ

Se desarrollará en esta última parte de este capítulo, un ejemplo aplicando el lenguaje OBJ. El ejemplo que se utilizará para ilustrar el empleo de OBJ, será el mismo que el que se utilizó en el capítulo anterior para explicar y mostrar los conceptos del lenguaje Larch, con el fin de obtener interesantes conclusiones.

Al igual que en Larch, para construir una especificación en OBJ, se sugieren también seguir una serie de pasos, los cuales se enuncian a continuación. Como se espera, algunos de estos son similares a los sugeridos para Larch.
Estos pasos guian una metodología de diseño de Arriba-Abajo (Top-Down).

1) Expresar los requerimientos del usuario a través de frases concisas (operaciones a realizar).

2) Desarrollar una intuición aproximada del problema. Esto requiere primero de una conversación verbal con el usuario.

3) Identificar los tipos de datos que participarán en la especificación.

4) Establecer las operaciones a realizarse o ejecutarse con cada uno de los tipos de datos participantes, de acuerdo a lo acordado en el inciso (1).

5) Identificar (si los hay) el tipo o los tipos independientes involucrados en la especificación, es decir, aquellos cuya definición requiere sólo de los tipos de los objetos ya incorporados o ya especificados. Los demás tipos que no cumplan con esta propiedad, serán llamados tipos dependientes, que pasarán a ser independientes una vez que hayan estado especificados y estén a disposición de ser requeridos por otros objetos.

6) Construir la especificación de los tipos a través de objetos, empezando por el más dependiente.
   Primero se indicarán los tipos asumidos por este objeto (si le son necesarios) luego se empezarán a 'traducir' las operaciones sobre estos tipos como funciones correctas ('OK-OPS') y se escribirá su semántica a través de ecuaciones (OK-EQNS). Si es necesario, se utilizan funciones fijas, escondidas, coerciones, etc.. Por último, se deben especificar las posibles acciones erroneas en que se pueden ope

rar estos tipos o para aquellos casos excepcionales, escribiendo para esto ecuaciones erroneas (ERR-EQNS) y estableciendo los posibles mensajes o funciones de error (ERR-OPS). En ambos casos es necesario ir declarando las variables utilizadas por los tipos (VARS).

7) Diseñar las pruebas necesarias para probar la especificación.

8) Ejecutar la especificación en el intérprete de OBJ y realizar las pruebas diseñadas. De haber algún error o incongruencia hacer las correcciones necesarias, repitiendo los pasos anteriores.

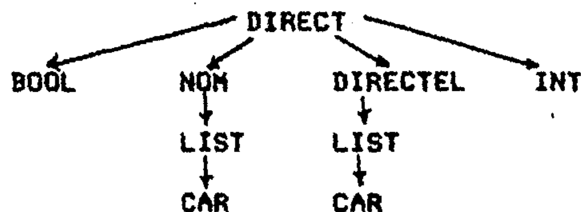
9) Implantar la especificación en algún lenguaje de programación.

Se construirá ahora la especificación del directorio en OBJ.

Los incisos (1) y (2) ya fueron desarrollados en la sección 2.3 del capítulo 2.

De acuerdo a las operaciones definidas por el usuario se deduce que los tipos involucrados en la especificación son: Directorio (DIRECT), el cual se considerará como un tipo abstracto con las mismas características mencionadas en el capítulo anterior; el Nombre de las personas (NOM); la Dirección y el Teléfono de las personas (DIRECTEL), estos dos últimos tipos también tienen las mismas características mencionadas en el capítulo anterior; Booleanos (BOOL), para obtener la respuesta acerca de que si una persona está o no en el directorio, así como para saber si éste estáo no vacío y para algunos otros casos que se mostrarán en la misma especificación; Enteros (INT), para saber el número de personas que están en el directorio, asícomo para identificar su número telefónico y su dirección; Caracter (CAR), para identificar los caracteres que constituirán el nombre y dirección de una persona; y el tipo llamado Lista (LIST), el cual será un conjunto de carateres.

El orden creciente de independencia entre estos tipos, es:



Se empezará primero a construir el objeto DIRECTORIO, que introducirá el tipo DIRECT, especificando las operaciones involucradas en él (véase sección 2.3). Obsérvese que la especificación de

este objeto es muy similar a la caracteristica Directorio.

```
OBJ DIRECTORIO
    SORTS DIRECT / BOOL, NOM, DIRECTEL, INT
    OK-OPS
                 CREADIRECT : -> DIRECT
        AGREGANOM _ , _ , _ : DIRECT, NOM, DIRECTEL -> DIRECT
          OBTDIRECTEL _ , _ : DIRECT, NOM -> DIRECTEL
          ESTAPERSONA _ , _ : DIRECT, NOM -> BOOL
           ELIMINAPER _ , _ : DIRECT, NOM -> DIRECT
      MODDIRECTEL _ , _ , _ : DIRECT, NOM, DIRECTEL -> DIRECT
                  TAMDIRECT : DIRECT -> INT
                          _ : DIRECTEL -> DIRECT (HIDDEN)

      ERR-OPS
              DIRECTORIO-VACIO : -> DIRECTEL
         PERSONA-NO-EN-EL-DIRECT : -> DIRECTEL
      VARS
              D : DIRECT
          N, N1 : NOM
        DT, DT1 : DIRECTEL
      OK-EQNS
        ( ESTAPERSONA( CREADIRECT, N ) = F )
        ( ESTAPERSONA( AGREGANOM( D, N, DT ) N1 ) =
                       N ES-IGUAL N1 ! ESTAPERSONA( D, N1 ) )
        ( TAMAÑODIRECT( CREADIRECT ) = 0 )
        ( TAMAÑODIRECT( AGREGANOM( D, N, DT ) ) = 1 +
                                 TAMAÑODIRECT( D ) )
      EQNS
        ( OBTENDIRECTEL( AGREGANOM( D, N, DT ) N1 ) =
                        IF ESTAPERSONA( D, N1 ) THEN
                           IF ( N ES-IGUAL-A N1 ) THEN DT
                           ELSE
                             OBTENDIRECTEL( D, N1 )
                        ELSE ( PERSONA-NO-EN-EL-DIRECT ) )
        ( ELIMINAPER( AGREGANOM( D, N, DT ) N1 ) =
                       IF ESTAPERSONA( D, N1 ) THEN
                          IF ( N ES-IGUAL-A N1 ) THEN D
                          ELSE
                            AGREGANOM( ELIMINAPER( D, N1 ),
                                       N, DT )
                       ELSE
                         ( PERSONA-NO-EN-EL-DIRECT ) )
        ( MODDIRECTEL( AGREGANOM( D, N, DT ), N1, DT1 ) =
                       IF ESTAPERSONA( D, N1 ) THEN
                          IF ( N ES-IGUAL-A N1 ) THEN
                             AGREGANOM( D, N, DT1 )
                          ELSE
                            AGREGANOM( MODDIRECTEL( D, N1,
                                       DT1 ), N, DT )
                       ELSE
                         ( PERSONA-NO-EN-EL-DIRECT ) )
       ERR-EQNS
        ( OBTENDIREC( CREADIRECT, N ) = DIRECTORIO-VACIO )
        ( ELIMINAPER( CREADIRECT, N ) = DIRECTORIO-VACIO )
```

```
    ( MODDIRECTEL( CREADIRECT, N, DT ) =  DIRECTORIO-VACIO )
JBO
```

A diferencia de la característica Directorio, este objeto en seguida de dar las funciones que corresponden a las operaciones a realizarse en el directorio, especifica la semántica también de aquellas situaciones excepcionales, a través de las ecuaciones erroneas, en lugar de considerarlas como un conjunto de expresiones inválidas como lo hace Larch, con la cladsula Exempts.
La coerción en este objeto tiene como fin convertir el tipo de las funciones erroneas en el tipo DIRECT, por ejemplo en la función: ELIMINAPER(CREADIRECT, N), cuyo tipo es DIRECT, el rango que es DIRECTORIO-VACIO es de tipo DIRECTEL.

Ahora se construirán los objetos NOMBRE-DE-PERSONAS y DIREC-TEL-DE-PERSONAS, los cuales introducen, respectivamente, los tipos NOM y DIRECTEL. Ellos identifican el nombre, la dirección y el teléfono de una persona a través de una secuencia de caracteres, de acuerdo a lo que especificó el usuario (véase sección 2.3).

```
OBJ NOMBRE-DE-PERSONAS
    SORTS  NOM / LIST
    OK-OPS
      _ : LIST -> NOM
JBO
```

```
OBJ DIREC-TEL-DE-PERSONAS
    SORTS  DIRECTEL / LIST
    OK-OPS
      _ : LIST -> DIRECTEL
JBO
```

Estos objetos asumen el tipo LIST, el cual falta por especificarse. Su fin es identificar una cadena de caracteres como el nombre de una persona o la dirección y teléfono de éstas. En cierta forma pueden considerarse las funciones que introducen como coerciones de tipo LIST al tipo NOM y DIRECTEL, respectivamente.

Por dltimo, se construirán los objetos COMPARA-NOMBRES, SECUENCIA-DE-CARACTERES y CARACTERES. Estos objetos complementan la especificación de los dos objetos anteriores, ya que en ellos se especifican las operaciones adicionales a realizarse sobre los tipos que asumen. Estas operaciones son (véase sección 2.3): agregar los caracteres válidos sugeridos por el usuario a un tipo de datos abstractos, para formar el nombre, dirección y teléfono de una persona; comparar los nombres de dos personas e identificar los caracteres válidos e inválidos sugeridos por el usuario antes de formar el nombre, dirección y teléfono.

El primer objeto no introduce ningdn tipo, sólo enriquece el objeto BOOL introduciendo la función booleana _ES-IGUAL-A_, utilizada en el objeto DIRECTORIO para determinar si los nombres de dos personas son iguales o no. Este objeto asume al objeto incorporado ID para determinar si dos caracteres (identificadores) a su vez son iguales o no.

El segundo objeto introduce el tipo LIST, el cual es considerado nuevamente como un dato abstracto que guarda secuencia de caracteres, para poder permitir definir e identificar el nombre, dirección y teléfono de una persona.

Y el tercer objeto introduce el tipo CAR e indica cuales son los caracteres válidos e inválidos sugeridos por el usuario.
A continuación se muestran estos objetos.

```
OBJ COMPARA-NOMBRES
    SORTS    / ID BOOL LIST CAR
    OK-OPS
       _ ES-IGUAL-A _ : LIST LIST -> BOOL
    VARS
        L, L1 : LIST
         J, I : CAR
    OK-EQNS
      ( (L J) ES-IGUAL-A (L1 I) =  J = I & L ES-IGUAL-A L1 )
      ( (L J) ES-IGUAL-A I = F )
      ( J ES-IGUAL-A (L1 I) = F )
      ( J ES-IGUAL-A I =  I = J )

JBO


OBJ SECUENCIA-DE-CARACTERES
    SORTS  LIST / CAR
    OK-OPS
         _ : CAR -> LIST
         _ : INT -> LIST
       _ _ : LIST LIST -> LIST  ( ASSOCIATIVE )
JBO


OBJ CARACTERES
    SORTS  CAR /
    OK-OPS
       _ : -> CAR
    ERR-OPS
       ( CARACTER-INVALIDO : -> CAR )
    OK-EQNS
       ( A = CAR );            ( 0 = CAR )
       ( B = CAR );            ( 1 = CAR )
         .  .  .                 .  .  .
         .  .  .                 .  .  .
         .  .  .                 .  .  .

       ( Z = CAR );            ( 9 = CAR )
       ( . = CAR )
       ( # = CAR )
       ( - = CAR )
       ( ' ' = CAR )
    ERR-EQNS
       ( ( = CARACTER-INVALIDO );  ( ) = CARACTER-INVALIDO )
       ( < = CARACTER-INVALIDO );  ( > = CARACTER-INVALIDO )
       ( ? = CARACTER-INVALIDO );  ( / = CARACTER-INVALIDO )
       ( % = CARACTER-INVALIDO );  ( & = CARACTER-INVALIDO )
       ( * = CARACTER-INVALIDO );  ( + = CARACTER-INVALIDO )
       ( ! = CARACTER-INVALIDO )
JBO
```

El primer objeto, a través de la función recursiva _ES-IGUAL-A_, determina cuando los nombres de dos personas son iguales, comparando caracter por caracter empezando por el último de ellos insertado.
Se dice que este objeto enriquece al objeto BOOL, ya que no introduce ningún tipo y asume este objeto incorporando en él la función booleana _ ES-IGUAL-A_. De la misma forma enriquece los objetos SECUENCIA-DE-CARACTERES y CARACTERES, ya que también los asume y define una función con los tipos que introducen.

El segundo objeto introduce el tipo LIST como una coerción entre éste y los tipos INT y CAR, luego, a través de la función con el atributo ASSOCIATIVE, se establece la inserción de caracteres a una secuencia de caracteres (de tipo LIST), para formar el nombre, dirección y teléfono de una persona.

Y en el tercer objeto (CARACTERES) se introducen de manera diferente los caracteres sugeridos por el usuario. Primero se especifica una función con dominio indeterminado, pero cuyo rango introduce el tipo CAR. Luego, a través de ecuaciones correctas y erroneas se introducen los caracteres válidos e inválidos, respectivamente, sin necesidad de especificar una función que los diferencíe como se hizo en la característica Caracteres-Válidos.

Con la especificación de estos tres últimos objetos, ha quedado concluida la especificación de un directorio telefónico en OBJ.

El orden en que deben aparecer los objetos anteriores para ser interpretados por OBJ es, desde aquel que introduce el tipo más independiente, hasta aquel que introduce el más dependiente.
En este caso, el orden de los objetos es: CARACTERES, SECUENCIA-DE-CARACTERES, COMPARA-NOMBRES, NOMBRE-DE-PERSONAS, DIREC-TEL-DE-PERSONAS y DIRECTORIO.
Esto se hace con el fin de que el intérprete identifique y reconosca los objetos que serán asumidos por otro u otros objetos, así como aquellos que se van enriqueciendo por la especificación de otros objetos.

De acuerdo a los pasos descritos, lo que a continuación sigue, es realizar y diseñar las pruebas de validación de la especificación (paso 7).

En general, el especificador es quien decidirá que tipos de pruebas efectuar en ella, por ejemplo de Caja Negra, de Caja Blanca o a través de casos de prueba, etc. Desafortunadamente, no se tiene disponible el intérprete de OBJ, por lo que no es posible mostrar la ejecución de estas pruebas en la especificación del directorio.
En seguida se describen algunos casos de prueba que muestran la manera en que OBJ interpreta esta especificación.

```
RUM  OBTDIRECTEL( AGREGANOM( CREADIRECT, JUAN GONZALEZ,
                  DIV. NTE. #23), JUAN GONZALEZ )  MUR

     AS DIERCTEL: DIV. NTE. #23
```

```
RUM  ESTALAPERSONA( DIRECT, RAFAEL MARTINEZ )  MUR
     ?WARNING : REWRITEEN, EXPRESION CANNOT BE DEPARSED

RUM  ESTAPERSONA( DIRECT, RAFAEL MARTINEZ )  MUR
     AS BOOL : F

RUM  OBTDIRECTEL( MODDIRECTEL( AGREGANOM( DIRECT, MARTIN LOPEZ,
                                   XOLA #12 ), JUAN GONZALEZ,
                       LIMA #23 TEL. 5-34-56-78), JUAN GONZALEZ ) MUR
     AS DIRECTEL: LIMA #23 TEL. 5-34-56-78

RUM  TAMDIRECT( DIRECT )  MUR
     AS INT : 2

RUM  ESTAPERSONA( ELIMINAPER( DIRECT, MARTIN LOPEZ ),
                  MARTIN LOPEZ )  MUR
     AS BOOL : F

RUM  AGREGANOM( DIRECT, JAVIER ESTRAZA, AV. SEIS # 2
                TEL 4-55-66-56 )  MUR
     AS DIRECT: >> ERROR >> AGREGANOM( DIRECT, JAVIER ESTRA
                                 CARACTER-INVALIDO )

RUM  OBTDIRECTEL( CREADIRECT, RAFAEL MARTINEZ )  MUR

     AS DIRECTEL : >> ERROR >> DIRECTORIO-VACIO

RUM  MODDIRECTEL( CREADIREC, MARTIN LOPEZ, PTE. 116 # 45 ) MUR

     AS DIRECTEL : >> ERROR >> DIRECTORIO-VACIO

RUM  ELIMINAPER( AGREGANOM( CREADIRECT, JUAN ANTONIO,
                            AV. SUR # 23 TEL. 5-66-45-45 ),
                 CESAR FLORES ) MUR

     AS DIRECT : >> ERROR >> PERSONA-NO-EN-EL-DIRECT
```

Si al haber realizado las pruebas anteriores, cualquiera que haya sido su aspecto o tipo, se encuentra algún error, se debe buscar la función o funciones que lo produjeron y examinar la semántica y sintaxis de éstas, para intentar buscar algúna incongruencia en ella. En ciertos casos, sirve a veces también dialogar con el usuario acerca del problema encontrado, ya que muchas veces su descripción de la especificación es incompleta o ambigua y el problema hallado puede ayudarle a conformar mejor las ideas plantedas y de esta manera solucionarlo.

Una véz realizadas las pruebas anteriores, en seguida se debe mostrar al usuario la especificación explicándole cual es el objetivo de cada función especificada, lo cual debe de coincidir con lo que él propuso. Es recomendable que aquí el usuario sea quien evalúe, a través de algunos casos que el considere convenientes, la correctividad de la especificación así como lo que él espera deba hacer, haciendo algunas evaluaciones con el intérpre

te.

De haber algún error encontrado por el usuario en la especificación, se deben repetir o seguir en la especificación los pasos descritos con el usuario.

Por último, una vez aprobada la especificación del directorio por el usuario, se debe implantar en algún lenguaje de programación, siguiendo la misma estructura funcional de la especificación.

Con este ejemplo, se pretendió mostrar la manera de construir la especificación de un problema, que en este caso fue de un directorio telefónico de nombres y direcciones de una ciudad, en un lenguaje más formal que es OBJ, a partir de su especificación en lenguaje natural que es el que empleó el usuario.
Las conclusiones a que se llegaron en el capítulo anterior al ejemplificar el empleo de Larch, son aplicables aquí también.

Existen algunas diferencias sobresalientes entre los lenguajes Larch y OBJ que se hicieron notar en la especificación del directorio telefónico y que es necesario volver a mencionar.

OBJ parece ser más bien un lenguaje de programación que de especificación, ya que permite ver como trabaja la especificación a través de las evaluaciones de las funciones, para lo que es necesario realizar pruebas como las que se harían si ya estuviera desarrollada en un lenguaje de programación, antes de implantarla en el lenguaje que lo ejecutará realmente.
Surge entonces la pregunta ¿Porque no aceptar la especificación del directorio telefónico en OBJ, como el sistema que lo ejecutará ?. La respuesta es porque el intérprete realiza la evaluación de las funciones de manera rudimentaria, ya que no sigue por ejemplo, un algoritmo eficiente de búsqueda en el directorio, para encontrar el nombre de una persona, ni tiene una adecuada estructura de datos para representar el directorio como una tabla indexada, ni suficiente memoria que se desearía para almacenar los nombres y direcciones de una persona, etc.. Muestra sólo que es lo que deben hacer las funciones al operar con ellas, pero no cómo deben de hacerlo, que es lo que no interesa ahora.

Otra de las diferencias principales, es que OBJ se limita sólo a especificar, por medio de funciones, tipos de datos abstractos, pero no dice cuál debe ser la implantación en un lenguaje de programación. En cierta forma dice cómo debe ser la implantación, ya que establece las funciones necesarias para especificar un tipo de datos abstractos, y que deben de hacer ellas. El algoritmo y la estructura de datos a emplear se seleccionan en la implantación.

En cambio, Larch está más cercano, por decirlo así, a la implantación de la especificación en un lenguaje de programación (p. ejem., a través de Larch\Pascal o Larch\Clu). Porque establece cuales son las rutinas a emplear y en ellas indica en forma precisa, que deben de hacer.

C A P I T U L O   C U A T R O.

Desarrollo e Implantación de un Ejemplo Aplicando el Lenguaje de Especificación Formal Larch.

En este último capítulo de este trabajo se desarrollará un ejemplo más aplicando el lenguaje Larch. La intención de este ejemplo es mostrar la relación entre una especificación de Interfaz y su implantación en el lenguaje de programación Pascal.

La razón por la cual se decidió realizar este ejemplo en el lenguaje Larch, y no en OBJ, fue porque con Larch es posible mostrar la relación entre un lenguaje de especificación formal y el sistema que ejecutará esta especificación a través del lenguaje Pascal.

4.1 Especificando un Ejemplo.

La especificación que se desea escribir en Larch, consiste en un conjunto de operaciones a realizar en varios archivos de alumnos de distintos grupos para llevar a cabo un control de sus calificaciones. Este archivo almacenará los nombres de los alumnos de un grupo, sus calificaciones parciales y el promedio de éstas.
El nombre de cada uno de estos archivos de alumos se requiere que esté contenido a su vez en un archivo de nombres de grupos de alumnos, para poderlos identificar y volver a usar.

Antes de describir el conjunto de operaciones que se desean realizar en un archivo dado de alumnos, se describen en seguida las operaciones que también se desean realizar en un archivo de nombres de grupos:

a) Crear el archivo de grupos que contendrá los nombres de diversos archivos de grupos de alumnos. Esta operación se debe realizar sólo una vez;

b) Editar un archivo de alumnos a través de su nombre. Si éste no está en el archivo de grupos, entonces se agrega en él y se crea un archivo de alumnos bajo este nombre.
En ambos casos, que esté o no el nombre, se debe indicar que éste es el archivo de alumnos el cual se va a editar y

c) Desplegar los nombres de los archivos de alumnos contenidos en el archivo de grupos.

Con respecto al conjunto de operaciones que se desean realizar en un archivo de alumnos, éstas son las siguientes:

a) Agregar un número dado de alumnos al archivo, sólo cuando recien se ha creado el archivo o se han agregado, modificado o eliminado nombres de alumnos;

b) Modificar, un número dado de veces, los nombres de los alumnos a través de un número que tienen asociados éstos en el archivo, sólo cuando se han agregado, modificado o eliminado nombres de alumnos;

c) Eliminar, un número dado de veces, los nombres de algunos alumnos del archivo, también a través de un número, sólo cuando se han agregado, modificado o eliminado nombres de alumnos;

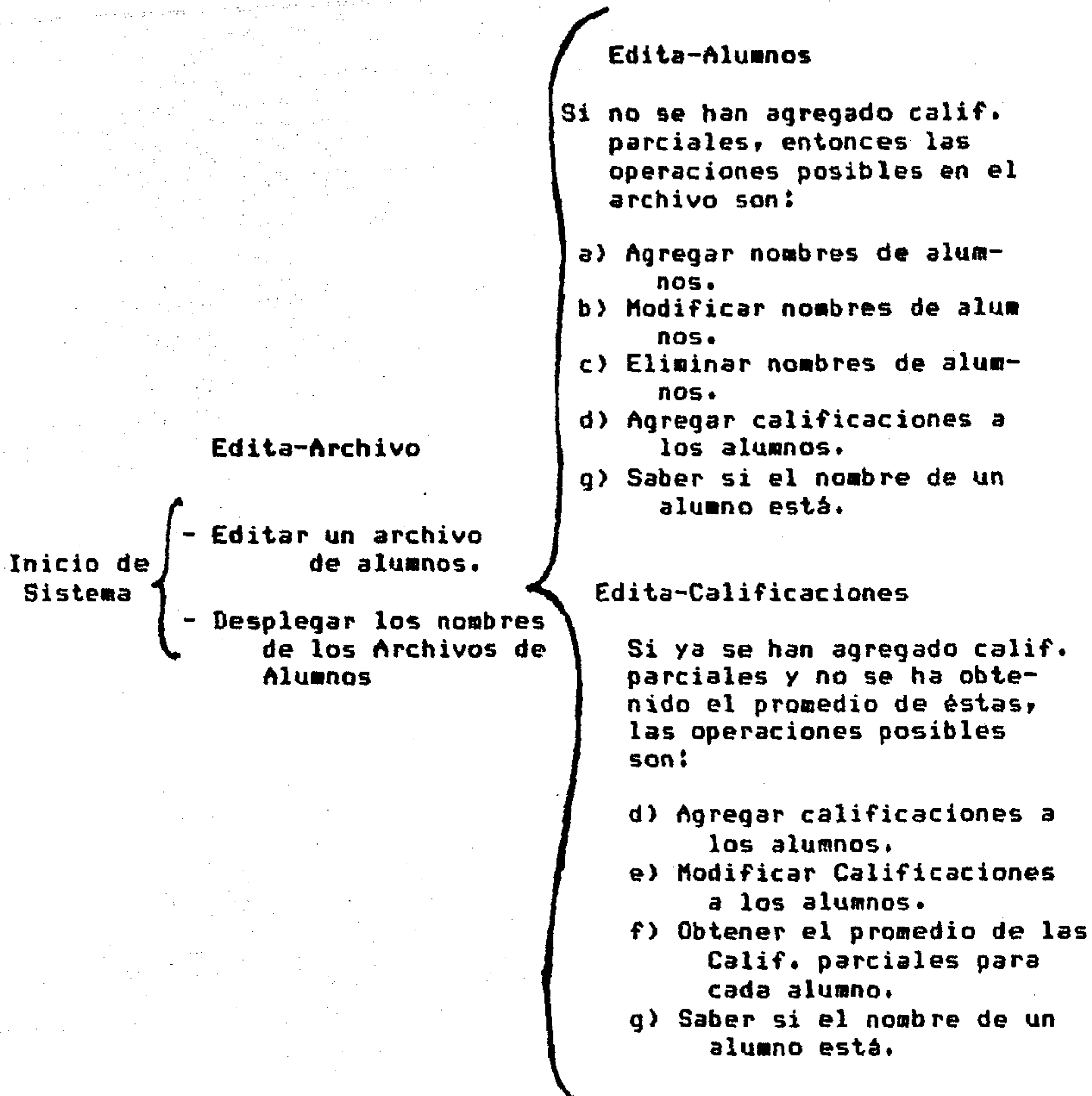
d) Agregar una calificación parcial a cada alumno, sólo cuando antes se han agregado, modificado o eliminado nombres de alumnos al archivo. Una vez realizada esta operación no es posible realizar todas las anteriores operaciones descritas;

e) Modificar, un número dado de veces, las calificaciones parciales de algunos alumnos de un parcial dado, a través del número que tienen asociados estos alumnos en el archivo. Las modificaciones de estas calificaciones sólo se podran hacer cuando se han agregado al archivo las calificaciones correspondientes a este parcial, y no cuando recien se ha creado el archivo, agregado, eliminado o modificado nombres de alumnos u obtenido ya el promedio de las calificaciones;

f) Obtener el promedio de las calificaciones parciales para cada alumno y agregarlo al archivo. Esta operación solo deberá efectuarse cuando antes se han agregado calificaciones parciales de los alumnos.
Una vez realizada esta operación ya no es posible realizar alguna de las operaciones anteriores ni aún ésta y

g) Saber en cualquier momento (excepto cuando ya se ha obtenido el promedio del grupo) si el nombre de un alumno está en el archivo.

El siguiente esquema muestra una estructura general de la especificación requerida. En ella se indica más bien cuando se deben ejecutar las operaciones a realizar en el archivo de alumnos:



Siguiendo los pasos descritos en la sección 2.3 para construir una especificación en Larch, corresponde ahora identificar el o los TDI involucrados en la especificación.

Se eligirá como TDI el archivo de grupos (ArchGrups); el archivo de alumnos (ArchAlums); los nombres de los alumnos (NomAlum) y los nombres de los grupos de alumnos (NomGrup). Estos dos últimos TDI serán asumidos como tipos de datos abstractos que contendrán los caracteres que permitirán formar el nombre de un alumno así como el de un archivo de alumnos, respectivamente. Siendo estos TDI definidos para un mismo fin y con las mismas características (un conjunto de caracteres), se considerarán como subtipos del TDI Cadena, el cual tendrá las mismas características y propiedades de cada uno de ellos.

En consecuencia, se tienen entonces que especificar las características que describirán las operaciones a realizarse en los

TDI: ArchGrups, ArchAlums y Cadena. Estas serán, respectivamente llamadas: Archivo-Grupos, Archivo-Alumnos y Nombres.

Primeramente se especificará cada una de las operaciones a realizar en un archivo de grupos, a través de la característica Archivo-Grupos y la correspondiente especificación de Interfaz.

4.1.1 Edición del Archivo de Nombres de Grupos.

A diferencia de la operación para crear un archivo de alumnos, que se realiza varias veces debido a la apertura de nuevos grupos, la operación para crear el archivo de grupos se debe realizar sólo una vez y en el momento de ser implantado el sistema correspondiente a esta especificación, debido a que no constituye una operación a ser realizada por el usuario, por lo que no debe figurar en la especificación de Interfaz, pero si en la característica Archivo-Grupos para indicar su implantación.

Debido a que el TDI ArchGrups es un dato abstracto que va a contener objetos (los nombres de los archivos de grupos de alumnos) y estos van a ser agregados a él, se pueden incluir en Archivo-Grupos, la característica Caja, la cual fue descrita en el segundo capítulo en la sección 2.2.1, haciendo sólo aqui una sustitución de nombres de funciones y tipos, y la característica Nombres, para indicar los objetos (de tipo NomGrup) que va a contener este TDI.
Por lo tanto una tentativa a la definición de la característica Archivo-Grupos puede ser la siguiente:

```
Archivo-Grupos : Trait
   Includes Nombres With [ NomGrup For Cadena ]
   Includes Caja With [ Crea For CreaArchGrups,
                        Inserta For AgregaNomGrup,
                        E For NomGrup,
                        C For ArchGrups ]
   Introduces
       ...
```

Con esta característica ha quedado especificado la operación para crear el archivo de grupos (a través de la función CreaArchGrups), así como la operación para agregar estos nombres al archivo (a través de la función AgregaNomGrup), si es que éstos no están en él. Para realizar esta última operación, se requiere entonces la especificación de la función booleana auxiliar:

EstáNomGrup : ArchGrups, NomGrup => Booleano

cuya finalidad es saber si el nombre de un archivo de alumnos está en el archivo de grupos.

La operación para editar un archivo de alumnos se especificará en la característica Archivo-Alumnos, por ser una operación a realizar con el TDI ArchAlum.

La operación para desplegar los nombres de los grupos contenidos en el archivo, se especificará a través de la función:

DespliegGrups: ArchGrups => ArchGrups

la cual indica que se debe obtener del archivo de grupos, cada nombre del grupo que se haya creado y una vez obtenido, escribirlo. Esta última operación se especificará por medio de la función:

Escribe : NomGrup => NomGrups

cuyo fin es sólo indicar que debe escribirse el nombre de un grupo sin modificarlo.
La primera función debe advertir, cuando se desea desplegar el nombre de un grupo, si el archivo de grupos esta vacío, por lo que ahora se requiere de la especificación de una función auxiliar para saber cuando el objeto ArchGrups esta vacío. Como se recordará, en la misma sección 2.2.1 del segundo capitulo se especificó la característica Está-Vacío que introduce esta función respecto al objeto tipo C, por lo que esta característica será incluida en Archivo-Grupos para indicar esta misma operación sobre el TDI ArchGrup.

A continuación se muestra la especificación completa de la característica Archivo-Grupos, así como la especificación de interfaz del TDI ArchGrups, la cual especificará la implantación de las operaciones a realizarse en el archivo de grupos, a través de procedimientos y funciones:

```
Archivo-Grupos : Trait
   Includes Nombres With [ NomGrup For Cadena ]
   Includes Caja, EstáVacío With [ Crea For CreaArchGrups,
                                   Inserta For AgregaNomGrup,
                                   E For NomGrup,
                                   C For ArchGrups ]
   Introduces
      EstáNomGrup : ArchGrups, NomGrup => Booleano
      DespliegGrups : ArchGrups => ArchGrups
      Escribe : NomGrup => NomGrup
   Constrains ArchGrup so that
      ArchGrup  partitioned by [ EstáNomGrup ]
      for all [ AG : ArchGrups; NG, NG1 : NomGrup ]
      EstáNomGrup( CreaArchGrups, NG ) = Falso
      EstáNomGrup( AgregaNomGrup( AG, NG ), NG1 ) =
           NombresIguales(NG, NG1) | EstáNomGrup(AG, NG1)
      DespliegGrups( AgregaNomGrup( AG, NG )) =
         If Not Está-Vacío( AG ) Then
            AgregaNomGrup( DespliegGrups( AG ), Escribe( NG ))
         else
            AgregaNomGrup( AG, Escribe( NG ))
   Exempts [ DespliegGrups( CreaArchGrups ) ]
   Implies Converts [ EstáNomGrup, Escribe, DespliegGrups,
                      EstáVacío ]
```

Especificación de la Característica Archivo-Grupos.

```
Type ArchGrups Export DesplNombreGrupos
    Based on sort ArchGrups From Archivo-Grupos
            with [ AG' : ArchGrups ]
    Based on sort Cadena From Nombres
            with [ NG : NomGrup For Cadena ]

    Procedure Crea-Archivo-Grupos( Var AG : ArchGrups );
            ensures AG post  = CreaArchGrups
       end;

    Procedure DesplNombreGrupos( Var AG : ArchGrups )
                       Signals( Archivo-Vacío )
            requieres AG = CreaArchGrups or
                       AG = AgregaNomGrup( AG', NG)
            ensures If EstáVacío( AG ) Then
                       Signals( Archivo-Vacío ) &
                          modifies nothing
                    else
                       DespliegGrups( AG )
       end;

    Function EstáNombreGrupo( AG : ArchGrups;
                              NG : NomGrup) : Boolean;
            ensures EstáNombreGrupo = EstáNomGrup( AG, NG)
       end;

    Function ArchGrups-Vacío( AG : ArchGrups ) : Boolean;
            ensures Archivo-Vacío = EstáVacío( AG )
       end;

    Procedure Agrega-Nombre-Grupo( Var AG : ArchGrups;
                                   NG : NomGrup );
            requieres  ¬EstáNomGrup( AG, NG )
            modifies at most[ AG ]
            ensures AG post  = AgregaNomGrup( AG, NG )
       end;


End ArchGrups
```

Especificación de Interfaz del TDI ArchGrups.

Obsérvese comó en la especificación de Interfaz no se especifica como función o procedimiento la operación Escribe, ya que ésta está implantada ya en el lenguaje Pascal como un procedimiento particular (Write).

La función NombresIguales se especificará más adelante en la característica Nombres.

4.1.2 Edición del Archivo de Alumnos.

Se especificarán ahora las operaciones a realizar en un archivo de alumnos, o sea en el TDI ArchAlums a través de la caracterís

tica Archivo-Alumnos y su correspondiente especificación de Interfaz. Primero se especificarán las operaciones para editar y crear un archivo de alumnos y más tarde aquéllas para manejar la información que contendrá.

La especificación de la operación para crear un archivo de alumnos (cuando el nombre de este archivo no está en el de grupos) se hará a través de la siguiente función:

CreaArchAlum : NomGrup => ArchAlums

ya que cada archivo de alumnos creado debe ser identificado por su nombre. Esta función establece que el archivo de alumnos creado bajo este nombre, está listo para realizar en él algunas de las operaciones a realizar con la información de los alumnos.

Para editar un archivo de alumnos, se requiere primeramente que sea proporcionado su nombre, esto se especificará a través de la función:

Lee: NomGrup => NomGrup.

La especificación de la operación para indicar cuál es el archivo a editar se hará a través de la siguiente función:

EditArchAlum : NomGrup => ArchAlums

Esta función supone que el nombre del archivo a editar está en el archivo de grupos. Por lo tanto la especificación de Interfaz de esta operación debe tener en cuenta la función EstáNomGrup.
Con la especificación de estas operaciones, la especificación de la característica Archivo-Alumnos, así como de la correspondiente especificación de Interfaz, que establece la implantación de la operación para editar un archivo de alumnos a través de un procedimiento llamado EditaArchAlums, es la siguiente:

```
Archivo-Alumnos : Trait
   Includes Archivo-Grupos
   Includes Nombres With [ NomGrup For Cadena ]
   Introduces
      CreaArchAlum : NomGrup => ArchAlums
      EditArchAlum : NomGrup => ArchAlums
      Lee : NomGrup => NomGrup
  Constrains ArchAlum so that
     ArchAlum Generated by [ CreaArchAlum ]
  Implies Converts[ Lee ]
```

Especificación de la característica Archivo-Alumnos.

```
Type ArchAlums Exports EditaArchAlums
     Based on sort ArchAlums From ArchivoAlumnos
     Based on sort ArchGrups From Archivo-Grupos
              with [ AG' : ArchGrups ]

     Based on sort Cadena From Nombres With
              [ NG : NomGrup For Cadena ]
```

```
Procedure CreaArchivoAlumnos( Var AA : ArchAlums;
                                  NG : NomGrup );
        ensures AA post  = CreaArchAlum( NG );
   end;

Procedure EditaArchAlums( Var AG : ArchGrups;
                              Var AA : ArchAlums );
        requieres (AG = CreaArchGrups or
                   not EstáVacío( AG ) )
        modifies at most [ AG ]
        ensures If Not EstáNomGrup(AG,Lee(NG)) Then
                   AgregaNomGrup( AG,NG ) &
                   AA post  = CreaArchAlum( NG )
                else
                   AA post  = EditArchAlum( NG )
   end;

End ArchAlums
```

Especificación de Interfaz del TDI ArchGrups.

Nuevamente en la especificación de Interfaz del TDI ArchGrups, no se especificará como función o procedimiento la operación LEE ya que ésta es otra función implantada en el lenguaje Pascal como un procedimiento particular (Read).

Ahora se especificarán, en el orden en que fueron descritas, las operaciones deseadas a realizar con la información contenida en un archivo de alumnos.

Todas estas operaciones suponen ya elegido (a través del procedimiento EditaArchAlumns) el archivo de alumnos que se va a editar (llamado de aquí en adelante AA, de tipo ArchAlums).

La operación para agregar n nombres de alumnos al archivo, (establecida en el inciso a de la sección 4.1) se especificará en la característica a través de un conjunto de funciones, las cuales indicarán qué es lo que debe de hacerse para lograr el objetivo de esta operación. Estas son las siguientes:

AgregaAlumN-Vec(AA, n) : Enumera las veces que deben agregarse al archivo AA, n alumnos, (n es tipo entero positivo distinto de cero);

AgregaAlum(AA, Ali) : Indica que debe agregarse al archivo AA, el nombre de un alumno (llamado Ali, de tipo NomAlum).

Una vez definido que es lo que deben de hacer estas funciones, la semántica de la primera función queda definida de la siguiente manera:

```
AgregaAlumN-Vec( AA, n ) =
   If n = 1 Then
      AgregaAlum( AA, LeeCad( Ali ) )
   else
     AgregaAlumN-Vec( AgregaAlum( AA, LeeCad( Ali )), n - 1 )
```

Nuevamente se utiliza aquí la función Lee para indicar que deben ser proporcionados los nombres de los alumnos. La sintaxis de esta función será ahora: LeeCad : Cadena => Cadena debido a que se utilizará para ambos tipos NomGrup y NomAlum. La especificación de la operación para indicar que debe ser proporcionado n, el número de alumnos a agregar al archivo, se hará a través de la función:

LeeEnt : Entero => Entero

Más adelante se describirá formalmente la sintaxis de estas funciones.

Antes de mostrar la especificación de Interfaz de esta operación, debe observarse que cuando se definió, se dijo que ésta sólo deberá aplicarse cuando el archivo esta vacío o se obtuvo como última versión de agregar, modificar o eliminar nombres de alumnos, pero no de haber agregado calificaciones parciales u obtenido el promedio de éstas.
Estos requerimientos deben establecerse en la característica correspondiente a través de la cláusula Exempts y en la especificación de Interfaz como pre-condiciones (que se especificarán a través de la cláusula requieres).
En la carácterística deben especificarse así:

```
ExemptsC AgregaAlumN-Vec( AgregaCalif(...), n)
         AgregaAlumN-Vec( AgregaProm(...), n) ]
```

Se ha supuesto que se ha definido n como un tipo entero positivo en la característica y especificado las funciones AgregaCalif y AgregaProm, cuyo propósito es agregar una calificación parcial de un alumno al archivo y obtener el promedio de las calificaciones parciales de un alumno, respectivamente, (estas funciones tienen como rango un objeto de tipo ArchAlums y se especificarán más adelante). Los puntos suspensivos seguidos de ellas, indican que falta aún por definirse su dominio.
En la especificación de Interfaz se especificará esta operación a través de un procedimiento llamado Agrega-N-Alumnos y es el siguiente:

```
Procedure Agrega-N-Alumnos( Var AA : ArchAlums );
          Signals( Número-No-Válido )
          requieres AA <> AgregaCalif(...) &
                    AA <> AgregaProm(...)
          modifies at mostC AA ]
          ensures If LeeEnt( n ) <= 0 Then
                       Signals(Número-No-Válido) &
                          modifies nothing
                  else
                    AApost  = AgregaAlumN-Vec( AA, n )
    end;
```

La siguiente operación a realizar es modificar del archivo n nombres de alumnos (inciso b), a través de un número que deberán tener asociados estos en el archivo. La especificación de esta operación se realizará también a través de un conjunto de funcio-

nes. Estas son las siguientes:

    ModifNomAlumN-Vec(AA, n) : Enumera las veces que deben modificarse del archivo AA, n alumnos;

    ModifNomAlum(AA, NoAi, Ali) : Localiza en el archivo AA la vez en que fue agregado el nombre del alumno correspondiente al número NoAi (de tipo entero positivo distinto de cero) y sustituye este nombre por Ali y lo agrega al archivo.

Como debido a que la segunda función debe localizar en el archivo el nombre del alumno correspondiente al número NoAi, es necesario especificar una función auxiliar, que será llamada NúmAlum, la cual dirá cual es el número que le corresponde a un determinado alumno del archivo, para así comparar este número con el número del nombre del alumno que se quiere modificar.
La especificación de esta función requiere comparar el alumno dado con el último alumno agregado al archivo y si son iguales, el número de este alumno será el total de alummos hasta ahora agregados al archivo, si no, se busca de nuevo en el archivo en que fue agregado el último alumno.
En consecuencia, para especificar esta última función, se requiere especificar cual es el total de alumnos agregados al archivo, que se especificará a través de la función:

TotalAlum : AA => Entero.

La semántica de esta función es la siguiente:

```
    TotalAlum( CreaArchAlum) = 0

    TotalAlum( AgregaAlum( AA, Ali)) = 1 + TotalAlum( AA )
```

Ahora se puede definir la semántica de la función NúmAlum de la siguiente manera:

```
    NúmAlum( AgregaAlum( AA, Ali), Alj) =
       If NomIguales( Ali, Alj ) Then
          TotalAlum( AA ) + 1
       else
         NúmAlum( AA, Alj)
```

Por lo tanto la semántica de las funciones para realizar esta operación es la siguiente:

```
    ModifNomAlumN-Vec( AgregaAlum( AA, Ali ), n ) =
       If n = 1 Then
         ModifNomAlum( AgregaAlum( AA, Ali), LeeEnt( NoAi ),
                       LeeCad( Alj ) )
       else
         ModifNomAlumN-Vec( ModifNomAlum( AgregaAlum( AA, Ali )
                            LeeEnt(NoAi), LeeCad(Alj) ), n - 1 )

    ModifNomAlum( AgregaAlum( AA, Ali ), NoAj, Alj ) =
```

```
      If NúmAlum( AgregaAlum( AA, Ali ), Ali ) = NoAj Then
         AgregaAlum( AA, Alj )
      else
         AgregaAlum( ModifNomAlum( AA, NoAj, Alj ), Ali )
```

La especificación de esta última función supone que NoAi está en el archivo. Para saber si este número está o no, debe especificarse la función booleana auxiliar EstáNúmAlum, cuyo fin es establecer que debe de hacerse para saber si el número correspondiente a un alumno está o no en el archivo.
La semántica de esta función es la siguiente:

```
   EstáNúmAlum( AgregaAlum( AA, Ali ), NoAi ) =
      If 0 < NoAi <= TotalAlum( AA ) + 1 Then Verdadero
      else Falso
```

Por el momento esta función tiene estos argumentos, ya que posiblemente se utilice también al momento de eliminar un alumno o modificar alguna de sus calificaciones parciales también a través de su número en el archivo.
Mas adelante se describirá formalmente la sintaxis de las funciones antes descritas.

Nuevamente, la especificación de estas funciones debe advertir cuando se desea modificar del archivo el nombre de un alumno cuando este está vacío, o se ha obtenido de agregar una calificación parcial de un alumno o de haber obtenido el promedio de éstas, ya que no se puede modificar el nombre de un alumno cuando estas operaciones se han realizado.
Estas también deben especificarse en la característica Archivo-Alumnos con la cláusula Exempts.
La primera pre-condición se establecerá en la especificacion de Interfaz a través de la cláusula: Signals(Archivo-Vacío), ya que de acuerdo a lo especificado, (en el diagrama mostrado de esta especificación) esta operación puede realizarse aún cuando el archivo de alumnos esta vacío, mientras que las otras dos (agregar una calificación parcial y obtener el promedio de éstas) se establecerán a través de la cláusula requieres, ya que esta operación no podra realizarse cuando se hayan realizado antes estas dos operaciones.
La especificación de Interfaz de esta operación se hará a traves de un procedimiento llamado Modifica-N-Nombres y es el siguiente:

```
   Procedure Modifica-N-Nombres( Var AA : ArchAlums );
             Signals( Archivo-Vacío, Número-No-Válido )
             requieres AA <> AgregaCalif(...) &
                       AA <> AgregaProm(...)
             modifies at most [ AA ]
             ensures If LeeEnt( n ) <= 0 Then
                        Signals(Número-No-Válido) &
                           modifies nothing
                     else
                       If TotalAlum( AA ) = 0 Then
                          Signals(Archivo-Vacío) &
```

```
                         modifies nothing
                 else
                   AA post  = ModifNomAlumn-Vec( AA, n )
    end;
```

La especificación de la operación para eliminar del archivo n nombres de alumnos, (inciso c) se hará también a través de un conjunto de funciones, que son las siguientes:

EliminaAlumN-Vec(AA, n) : Enumera las veces que deben eliminarse del archivo AA, n alumnos;

EliminaAlum(AA, NoAi) : Localiza en el archivo AA la vez en que fue agregado el nombre del alumno correspondiente al número NoAi y lo elimina.

La semántica de estas funciones es la siguiente:

```
EliminaAlumN-Vec( AgregaAlum( AA, Ali ), n ) =
   If n = 1 Then
      EliminaAlum( AgregaAlum( AA, Ali ), LeeEnt( NoAj) )
   else
      EliminaAlumN-Vec( EliminaAlum( AgregaAlum( AA, Ali ),
                     LeeEnt( NoAj ) ), n - 1 )

EliminaAlum( AgregaAlum( AA, Ali ), NoAj ) =
   If NúmAlum( AgregaAlum( AA, Ali ), Ali ) = NoAj Then
      AA
   else
      AgregaAlum( EliminaAlum( AA, NoAj ), Ali )
```

Esta última función también supone que el número de alumno a eliminar está en el archivo, por lo que debe también de tomarse en cuenta la función EstáNúmAlum en la implantación de esta operación.

La restricciones que tiene esta operación al ser aplicada a través de las funciones anteriores son las mismas que de la operación para modificar los nombres de n alumnos, por lo que éstas se especificarán en la característica y en la especificación de Interfaz de la misma forma.

La especificación de Interfaz de esta operación se hará a través de un procedimiento llamado Elimina-N-Nombres y es el siguiente:

```
 Procedure Elimina-N-Nombres( Var AA : ArchAlums );
           Signals( Archivo-Vacío, Número-No-Válido)
           requieres AA <> AgregaCalif(...) &
                     AA <> AgregaProm(...)
           modifies at most [ AA ]
           ensures If LeeEnt( n ) <= 0 Then
                        Signals(Número-No-Válido) &
                          modifies nothing
                   else
                     If TotalAlum( AA ) = 0 Then
                        Signals(Archivo-Vacío) &
```

```
                         modifies nothing
                   else
                     AA post = EliminaAlumN-Vec( AA, n )
        end;
```

Antes de continuar con la especificación de las demás operaciones, se muestra enseguida formalmente cual es la especificación de Interfaz de las operaciones ya especificadas, así como la sintaxis de las funciones introducidas.

```
    Archivo-Alumnos : Trait
       Includes Archivo-Grupos
       Includes Nombres With [ NomGrup, NomAlum For Cadena ]
       Introduces
          CreaArchAlum : NomGrup => ArchAlums
          EditArchAlum : NomGrup => ArchAlums
          LeeCad : Cadena => Cadena
          LeeEnt : Entero => Entero
          AgregaAlumN-Vec : ArchAlums, Entero => ArchAlums
          ModifNomAlumN-Vec : ArchAlums, Entero => ArchAlums
          EliminaAlumN-Vec : ArchAlums, Entero => ArchAlums
          AgregaAlum : ArchAlums, NomAlum => ArchAlums
          ModifNomAlum : ArchAlums, Entero, NomAlum => ArchAlums
          EliminaAlum : ArchAlums, Entero => ArchAlums
          NúmAlum : ArchAlums, NomAlum => Entero
          EstáNúmAlum : ArchAlums, Entero => Booleano
          TotalAlum : ArchAlums => Entero.
       Constrains ArchAlums so that
             ArchAlum Generated by [ CreaArchAlum, AgregaAlum,
                      AgregaCalif, AgregaProm ... ]
             For all [ AA : ArchAlums; Ali, Alj : NomAlum;
                       NoAi : Entero ( >0 ); n : Entero ( >0 ) ]

             AgregaAlumN-Vec( AA, n ) =
                If n = 1 Then
                   AgregaAlum( AA, Lee( Ali ))
                else
                   AgregaAlumN-Vec( AgregaAlum( AA, LeeCad(Ali) ),
                                    n - 1)

             ModifNomAlumN-Vec( AgregaAlum( AA, Ali ), n ) =
                 If n = 1 Then
                    ModifNomAlum( AgregaAlum( AA, Ali ),
                                  LeeEnt( NoAi ), LeeCad( Alj ) )
                 else
                   ModifNomAlumN-Vec( ModifNomAlum( AgregaAlum(
                                      AA, Ali ), LeeEnt( NoAi ),
                                      LeeCad( Alj ) ), n - 1 )

             ModifNomAlum( AgregaAlum( AA, Ali ), NoAi, Alj ) =
                If NúmAlum( Agrega( AA, Ali ), Ali ) = NoAi Then
                   AgregaAlum( AA, Alj )
                else
                  AgregaAlum( ModifNomAlum( AA, NoAi, Alj ), Ali )
```

```
EliminaAlumN-Vec( AgregaAlum( AA, Ali ), n ) =
   If n = 1 Then
      EliminaAlum( AgregaAlum( AA, Ali ),
                   LeeEnt( NoAi ) )
   else
     EliminaAlumN-Vec( EliminaAlum( AgregaAlum(
                       AA, Ali), LeeEnt( NoAi ) ),
                       n - 1 )

EliminaAlum( AgregaAlum( AA, Ali ), NoAi ) =
   If NúmAlum( AgregaAlum( AA, Ali ), Ali ) = NoAi
      Then
          AA
   else
     AgregaAlum( EliminaAlum( AA, NoAi ), Ali )

NúmAlum( AgregaAlum( AA, Ali), Alj) =
  If NombresIguales( Ali, Alj ) Then
     TotalAlum( AA ) + 1
  else
     NúmAlum( AA, Alj)

EstáNúmAlum( SeCreaArchivo, NoAi ) = Falso

EstáNúmAlum( AgregaAlum( AA, Ali ), NoAi ) =
  If 0 < NoAi <= TotalAlum( AA ) + 1 Then
     Verdadero
  else
     Falso

TotalAlum( CreaArchAlum ) = 0

TotalAlum( AgregaAlum( AA, Ali)) = 1 + TotalAlum( AA)

Exempts[ AgregaAlumN-Vec( AgregaCalif( ... ), n),
         AgregaAlumN-Vec( AgregaProm( ... ), n),
         ModifNomAlumN-Vec( CreaArchAlum, n),
         ModifNomAlumN-Vec( AgregaCalifs( ... ), n),
         ModifNomAlumN-Vec( AgregaProm( ... ), n),
         EliminaAlumN-Vec( CreaArchAlum, n),
         EliminaAlumN-Vec( AgregaCalifs( ... ), n),
         EliminaAlumN-Vec( AgregaProm( ... ), n), ... ]
Implies Converts[ Lee ... ]
```

Especificación de la característica Archivo-Alumnos.

```
Type ArchAlums Exports EditaArchAlums, Agrega-N-Alumnos,
                       Modifica-N-Nombres, Elimina-N-Nombres.

    Based on sort ArchAlums From Archivo-Alumnos
    Based on sort ArchGrups From Archivo-Grupos
             with [ AG' : ArchGrups ]
    Based on sort Cadena From Nombres
             with [ NG : NomGrup For Cadena ]
```

ESTA TESIS NO DEBE
SALIR DE LA BIBLIOTECA

```
    Procedure CreaArchivoAlumnos( Var AA : ArchAlums;
                                      NG : NomGrup );
              ensures AA post  = CreaArchAlum( NG );
       end;

    Procedure EditaArchAlums( Var AG : ArchGrups;
                                  Var AA : ArchAlums );
              requieres (AG = CreaArchGrups or
                         Not EstáVacío( AG )
              modifies at most [ AG ]
              ensures If Not EstáNomGrup(AG,LeeCad(NG)) Then
                         AgregaNomGrup( AG,NG ) &
                         AA = CreaArchAlum( NG )
                      else
                         AA post  = EditArchAlum( NG )
       end;

Procedure Agrega-N-Alumnos( Var AA : ArchAlums );
          Signals(Número-No-Válido)
          requieres AA <> AgregaCalif(...) &
                    AA <> AgregaProm(...)
          modifies at most[ AA ]
          ensures If LeeEnt( n ) <= 0 Then
                      Signals(Número-No-Válido) &
                         modifies nothing
                  else
                    AA post= AgregaAlumN-Vec( AA, n )
   end;

Procedure Modifica-N-Nombres( Var AA : ArchAlums );
          Signals( Archivo-Vacío, Número-No-Válido )
          requieres AA <> AgregaCalif(...) &
                    AA <> AgregaProm(...)
          modifies at most [ AA ]
          ensures If TotalAlum( AA ) = 0 Then
                       Signals(Archivo-Vacío) &
                           modifies nothing
                  else
                    If LeeEnt( n ) <= 0 Then
                      Signals(Número-No-Válido) &
                          modifies nothing
                    else
                      AA post  = ModifNomAlumn-Vec( AA, n )
   end;

Procedure Elimina-N-Nombres( Var AA : ArchAlums );
          Signals( Archivo-Vacío, Número-No-Válido)
          requieres AA <> AgregaCalif(...) &
                    AA <> AgregaProm(...)
          modifies at most [ AA ]
          ensures If TotalAlum( AA ) = 0 Then
                       Signals(Archivo-Vacío) &
                          modifies nothing
                  else
                    If LeeEnt( n ) <= 0 Then
```

```
                    Signals(Número-No-Válido) ;
                        modifies nothing
                else
                  AA      = EliminaAlumN-Vec( AA, n )
  end;

End ArchAlums
```

Especificación de Interfaz del TDI ArchAlums.

Respecto a esta especificación, puede preguntarse por ejemplo, ¿porqué la función con el argumento: ModifNomAlum(CreaArchAlum, NoAi, Ali), así como algunas otras que resultan ser ambiguas también, no aparecen dentro de la claúsula Exempts ?. La razón es porque esta ambigüedad es detectada antes cuando con estos argumentos es llamada la función que a su vez llamará a éstas que pudieran resultar ambiguas; las primeras son las que figuran en la claúsula Exempts, en este ejemplo es a través de la función ModifNomAlumN-Vec(CreaArchAlum, n).

Continuando con la especificación, la siguiente operación a especificar es agregar una calificación parcial a cada uno de los alumnos que aparecen en el archivo (inciso d). El número de estas calificaciones para cada alumno se supondrá que estará limitado a 5 y serán distinguidas por el número de calificación que se ha agregado al archivo.

Esta operación se especificará de nuevo a través de un conjunto de funciones que son las siguientes:

AgregaCalifs(AA, NoAi, Pi) : Indica que debe recorrerse todo el archivo de alumnos para agregarle al alumno cuyo numero es NoAi, la calificación parcial número Pi (de tipo entero positivo distinto de cero);

AgregaCalif(AA, Ali, Ci, Pi) : Indica que debe de agregarse al archivo AA, al alumno cuyo nombre es Ali, la calificación Ci (de tipo real), que corresponde al parcial Pi.

Debido a que la primera función establece que debe recorrerse todo el archivo secuencialmente, debe saber cuando ha llegado al final de éste, lo cual se sabrá comparando el número de alumno NoAi con el número total de alumnos.
Respecto a la segunda función, debido a que se desea escribir el nombre del alumno para identificar la calificación que le corresponde y no su número correspondiente, se requiere especificar una función auxiliar, que se llamará Alum, la cual proporcionándole el archivo de alumnos y un número de alumno, indicará cual es el nombre del alumno al que le corresponde este número. La semántica de esta función es la siguiente:

```
Alum( AgregaAlum( AA, Ali), NoAi) =
    If TotalAlum( AA ) + 1 = NoAi Then
       Ali
```

```
        else
          Alum( AA, NoAi)
```

Y debido a que esta función tiene como fin agregar una calificación parcial de un alumno al archivo, se debe proporcionar ésta, lo cual se especificará a través de la siguiente función:

```
LeeReal : Real => Real
```

Por lo tanto ahora puede definirse la semántica de la función requeridas para establecer la operación de agregar calificaciones parciales al archivo:

```
 AgregaCalifs( AA, NoAi, Pi) =
       If NoAi = ToTalAlum( AA ) Then
          AgregaCalif( AA, Escribe( Alum( AA, NoAi ) ),
                       LeeReal( Ci ), Pi )
       else
          AgregaCalifs( AgregaCalif( AA, Escribe( Alum( AA, NoAi)),
                        LeeReal( Ci ), Pi ), NoAi + 1, Pi )
```

La sintaxis de la función Escribe es similar a la especificada en la característica Archivo-Grupos (Escribe : NomAlum => NomAlum).

Debido a que a partir del momento en que se agrega la segunda calificación parcial al archivo, las funciones TotalAlum y Alum, especificadas anteriormente, ya no se pueden utilizar por el tipo de argumento que tienen, que no corresponde a la última función aplicada en el archivo, se requiere agregar los siguientes axiomas:

```
    TotalAlum( AgregaCalif( AA, Alum( AA, NoAi), Ci, Pi)) =
             TotalAlum( AA )

    Alum( AgregaCalif( AA, Ali, Ci, Pi), NoAi ) = Alum( AA )
```

La igualdad de estos axiomas se hace obvia si se toma en cuenta el hecho de que aunque se hayan agregado calificaciones al archivo, no implica que el número de alumnos en el archivo y el número asociado a cada uno de ellos cambien.

De acuerdo a lo especificado, en relación a la operación de agregar una calificación parcial, se requiere que ésta no se lleve a cabo cuando apenas se ha creado el archivo de alumnos u obtenido el promedio de las calificaciones parciales de los alumnos, por lo que entonces es necesario advertir en la característica lo siguiente:

```
    Exempts[ AgregaCalifs( CreaArchAlum, NoAi, Pi ),
             AgregaCalifs( AgregaProm( ... ), NoAi, Pi ) ]
```

Teniendo en cuenta ahora la descripción anterior acerca de lo que debe de hacerse para agregar una calificación parcial, se tiene la siguiente especificación de Interfaz de esta operación, que se realizará a través de un procedimiento llamado Agrega-Calif-Parc:

```
Procedure Agrega-Calif-Parc( Var AA : ArchAlums );
          Signals( Archivo-Vacío, Calif-Parc-No-Procede)
          requieres AA <> AgregaProm( ... )
          modifies at most[ AA ]
          ensures If TotalAlum( AA ) = 0 Then
                        Signals( Archivo-Vacío ) &
                          modifies nothing
                  else
                    If LeeEnt( Pi ) <= 0  Or  Pi > 5 Then
                        Signals( Calif-Parc-No-Procede ) &
                           modifies nothing
                    else
                      AApost  = AgregaCalifs( AA, 1, Pi)
   end;
```

La siguiente operación consiste en modificar n veces una calificación parcial correspondiente a un número de examen parcial y número de alumno dados (inciso e). Para ejecutar esta operación se requiere del archivo de alumnos, el número de calificación parcial a modificar, el número de veces que serán modificadas las calificaciones sobre este parcial, el número de alumno al cual se va a modificar su calificación y la calificación que se desea tenga ahora.

Para especificar esta operación, se requiere definir de nuevo un conjunto de funciones, las cuales indicarán que debe de hacerse para realizar esta operación:

ModifCalifsParcN-Vec(AA, Pi, n) : Localiza en el archivo AA la vez que fue agregado el parcial número Pi, para modificar estas calificaciones n veces;

ModifCalifN-Vec(AA, n) : Enumera las n veces en que al archivo AA, dado por la función anterior, le serán modificadas las calificaciones correspondientes al parcial Pi;

ModifCalif(AA, NoAi, Ci) : Localiza en el archivo AA, dado por la función anterior, la vez en que fue agregada la calificación parcial correspondiente al alumno cuyo número es NoAi y la sustituye por la calificación Ci.

La primera función supone que el número de calificación parcial a modificar está en el archivo. Para asegurarse de esto, debe especificarse la función booleana definida como EstáParc, que establece lo que debe de hacerse para saber si el número de parcial Pi está en el archivo AA.
La semántica de esta función debe establecer comparar Pi con el número de parciales agregados hasta ahora en el archivo AA.
Esto requiere la especificación de la función auxiliar NúmParc, la cual cuenta el número de examenes parciales agregados al archivo AA. La semántica de estas funciónes es la siguiente:

```
NúmParc( AgregaCalif( AA, Ali, Ci, Pi ) ) =
   If Pi = 1 Then 1
```

```
   else
     1 + NúmParc( AA )

EstáParc( AgregaCalif( AA, Ali, Ci, Pi ) ), Pj) =
   If NúmParc( AgregaCalif( AA, Ali, Ci, Pi ) ) >= Pj Then
     Verdadero
   else
     Falso
```

La semántica entonces de las funciones para realizar la operación de modificar calificaciones de alumnos correspondiente a un parcial es la siguiente:

```
ModifCalifsParcN-Vec( AgregaCalif( AA, Ali, Ci, Pi ), Pj,
                      n ) ) =
    If Pi = Pj Then
       ModifCalifN-Vec( AgregaCalif( AA, Ali, Ci, Pi ), n)
    else
       AgregaCalif( ModifCalifsParcN-Vec( AA, Pj, n), Ali,
                    Ci, Pi )

ModifCalifN-Vec( AgregaCalif( AA, Ali, Ci, Pi ), n) =
    If n = 1 Then
       ModifCalif( AgregaCalif( AA, Ali, Ci, Pi ),
                   LeeEnt( NoAj ), LeeReal( Cj ) )
    else
       ModifCalifN-Vec( ModifCalif( AgregaCalif( AA, Ali, Ci,
                        Pi ), LeeEnt( NoAj ), LeeReal( Cj ) ),
                        n - 1)

ModifCalif( AgregaCalif( AA, Alum( AA, NoAi), Ci, Pi), NoAj,
            Cj) =
     If NoAi = NoAj Then
        AgregaCalif( AA, Alum( AA, NoAi), Cj, Pi)
     else
        AgregaCalif( ModifCalif( AA, NoAj, Cj),
                     Alum( AA, NoAi), Ci, Pi)
```

La especificación de la segunda función supone nuevamente que el número de alumno al cual se va a modificar su calificación parcial, está en el archivo. Para asegurarse de esto, debe tomarse en cuenta al momento de implantar esta función, la siguiente, ahora con el argumento de la última operación realizada en el archivo:

```
EstáNúmAlum( AgregaCalif( AA, Ali, Ci, Pi), NoAi ) =
             EstáNúmAlum( AA, NoAi )
```

Obviamente, la especificación de esta operación a través de la función ModifCalifsParcN-Vec tiene argumentos con los cuales no es válido hacer referencia, como se especificó en un principio. Esta función con estos argumentos se muestra en seguida:

```
ExemptsC ModifCalifsParcN-Vec( CreaArchAlum, Pi, n),
```

```
ModifCalifsParcN-Vec( AgregaAlum( AA, Ali), Pi, n )
ModifCalifsParcN-Vec( AgregaProm( ... ), Pi, n ) ]
```

La especificación de Interfaz de esta operación que se realizará a través de un procedimiento llamado Modif-N-CalifsParc es la siguiente (AA' es un objeto de tipo ArchAlums):

```
Procedure Modif-N-CalifsParc( Var AA : ArchAlum);
          Signals( Parcial-No-Está, Número-Negativo)
          requieres AA <> CreaArchAlum,
                    AA <> AgregaAlum( AA', Ali ) &
                       AA <> AgregaProm( ... )
          modifies at most [ AA ]
          ensures If not EstáParc( AA, LeeEnt( Pi ))
                       Then
                         Signals( Parcial-No-Está ) &
                            modifies nothing
                  else
                    If LeeEnt( n ) <= 0 Then
                      Signals( Número-Negativo ) &
                         modifies nothing
                    else
                      AA post  = ModifCalifsParcN-Vec( AA, Pi, n)
   end;
```

Para realizar la operación de obtener el promedio de las calificaciones parciales para cada alumno (inciso f), es decir, sumar todas las calificaciones parciales correspondientes a cada alumno del archivo y luego dividir este resultado entre el número de exámenes parciales realizados, se requiere que con solo proporcionar el archivo de alumnos, se obtega el promedio de las calificaciones parciales de cada alumno a partir del primero y hasta el último alumno del archivo automáticamente.
Para especificar esta operación, se especificará el siguiente conjunto de funciones:

ObtenProm(AA, NoAi) : Obtiene el promedio de las calificaciones parciales del alumno cuyo número en el archivo es NoAi;

AgregaProm(AA, NoAi, Pr) : Indica que debe de agregarse al archivo AA, el promedio Pr (de tipo real) de las calificaciones parciales correspondientes al alumno número NoAi;

SumaCalif(AA, NoAi) : Localiza en el archivo AA, cada una de las calificaciones parciales correspondientes al alumno número NoAi, las suma y el resultado lo asigna a esta función.

La definición de la primera función se hará recursivamente incrementando cada vez el número de alumno en uno, para indicar que al ejecutar esta función, se deben obtener el promedio de las calificaciones parciales de cada alumno del archivo.

La semántica de las funciones anteriores es entonces la si-

guiente:

```
    ObtenProm( AA, NoAi) =
       If NoAi = TotalAlum( AA) Then
          AgregaProm( AA, NoAi, SumaCalif( AA, NoAi) /
          NúmParc( AA))
       else
          ObtenProm( AgregaProm( AA, NoAi, SumaCalif( AA, NoAi) /
                     NúmParc( AA)), NoAi + 1 )

    SumaCalif( AgregaCalif( AA, Alum( AA, NoAj), Ci, Pi ), NoAi) =
       If Pi = 1 Then
          If NoAi = NoAj Then
             Ci
          else
            SumaCalif( AA, NoAi)
       else
         If NoAi = NoAj Then
            Ci + SumaCalif( AA, NoAi)
         else
           SumaCalif( AA, NoAi)
```

Debido a que a partir del momento en que se agrega al archivo el promedio del segundo alumno no se pueden aplicar las funciones descritas anteriormente: TotalAlum, SumaCalif y NúmParc, por el tipo de argumento que tienen, que no corresponde a la última función aplicada en el archivo (AgregaProm), es necesario agregar entonces los siguientes axiomas:

```
    TotalAlum( AgregaProm( AA, NoAi, Pr ) ) = TotalAlum( AA )
    SumaCalif( AgregaProm( AA, NoAi, Pr ) ) = SumaCalif( AA )
    NúmParc( AgregaProm( AA, NoAi, Pr ) ) = NúmParc( AA )
```

La descripción formal de la sintaxis de estas funciones se hará más adelante.

Nuevamente, de acuerdo a lo especificado, la ejecución de esta operación a través de la función ObtenProm tiene argumentos con los cuales no es válido hacer referencia a ella.
Estas restricciones son las siguientes:

```
    Exempts[ ObtenProm( CreaArchAlum, NoAi ),
             ObtenProm( AgregaAlum( AA, Ali ), NoAi ),
             ObtenProm( ObtenProm( AA, NoAi), NoAi ) ... ]
```

La especificación de Interfaz de esta operación, que se realizará a través de un procedimiento llamado ObtenPromedios, es la siguiente:

```
    Procedure Obten-Promedios( Var AA : ArchAlums );
              requieres AA <> CreaArchAlum,
                        AA <> AgregaAlum( AA', Ali ) &
                        AA <> AgregaProm( AA', NoAi, Pr)
```

```
            modifies at most [ AA ]
            ensures AA post = ObtenProm( AA, 1 )
      end;
```

Por último, una operación más que se desea efectuar en el archivo de alumnos, es saber en cualquier momento si el nombre de un alumno está en él (inciso g).
Esta operación se puede especificar a través de una función booleana llamada EstáNomAlum, cuya semántica establecerá localizar en el archivo el nombre del alumno dado.
Para definir la semántica de esta función, es necesario utilizar la función NombresIguales. Esta función tiene como propósito establecer si los nombres de dos alumnos son iguales o no, ya que para saber si el nombre de un alumno está en el archivo, es necesario compararlo con cada uno de los nombres del archivo.
Los argumentos que debe tener la función EstáNomAlum son el nombre del alumno a buscar y el archivo de alumnos, obtenido después de ejecutar cualquiera de las operaciones anteriores (excepto la de obtener el promedio de las calificaciones del grupo) especificando de esta manera que esta operación se puede realizar después de ejecutar algunas de estas operaciones.
La semántica de esta función es entonces la siguiente:

```
EstáNomAlum( CreaArchAlum, Ali) = Falso

EstáNomAlum( AgregaAlum( AA, Ali), Alj) =
            NombresIguales( Ali, Alj) | EstáNomAlum( AA, Alj)

EstáNomAlum( AgregaCalif( AA, Ali, Ci, Pi ), Alj ) =
         NombresIguales( Ali, Alj ) | EstáNomAlum( AA, Alj )
```

Debido a que esta operación no se desea realizar después de haber obtenido el promedio de las calificaciones de los alumnos, es necesario entonces declarar la siguiente restricción:

```
Exempts[ EstáNomAlum( AgregaProm( AA, NoAi, Pr ), NoAj ) ... ]
```

La especificación de Interfaz de esta operación, que se realizará a través de un procedimiento llamado Está-Nombre-Alumno, es la siguiente:

```
 Procedure Está-Nombre-Alumno( AA : ArchAlums, Ali : NomAlum);
             Signals( Está-el-Alumno, No-Está-el-Alumno)

          ensures If EstáNomAlum( AA, Ali) then
                       Signals( Está-el-Alumno )
                     else
                       Signals( No-Está-el-Alumno )
      end;
```

La especificación completa de la característica Archivo-Alumnos, así como la especificación de Interfaz del TDI ArchAlums se muestra en el apendice C.

4.1.3 Especificación de la Característica Nombres y del TDI Cadena.

Se especificará ahora la característica Nombres y la especificación de Interfaz correspondiente al TDI Cadena.
Como se mencionó en un principio, este TDI corresponde a los tipos NomAlum y NomGrup asumidos bajo estos nombres por las características Archivo-Alumnos y Archivo-Grupos, respectivamente. La finalidad de estos tipos es registrar y distiguir los nombres de los alumnos y de un archivo de alumnos, respectivamente, como un conjunto de caracteres.
La característica Nombres tiene como objetivo especificar las operaciones que se desean realizar sobre el TDI Cadena.

Las operaciones que se desean efectuar sobre este TDI son:

- Crear un objeto el cual contendrá los caracteres que forman el nombre de un alumno y el de un archivo de Alumnos;

- Agregar al objeto estos caracteres y

- Saber cuando el nombre de dos alumnos o archivos de ellos es el mismo. Esta fue una operación utilizada en las características Archivo-Alumnos y Archivo-Grupos, definida con la función NombresIguales.

Ahora bien, como se recordará en la sección 2.3 del segundo capítulo, al desarrollar la especificación del directorio, se tuvo ahí también la necesidad de especificar los nombres de las personas inscritas al directorio como cadenas de caracteres, (a través del TDI llamado también Cadena) introduciendo primero los caracteres válidos que contendría un nombre y luego las operaciones que se deseaban efectuar en él, (que coinciden con las descritas anteriormente), además de la operación para diferenciar los nombres de dos personas del directorio. Por lo tanto esta especificación es la misma que para el TDI Cadena de nuestro ejemplo.
A diferencia de aquí, no se especificarán los caracteres válidos que conforman los nombres de un alumno y de un archivo.

Por lo tanto, la especificación de la característica Nombres, así como de la de Interfaz del TDI Cadena, es la siguiente:

```
Nombres : Trait
  Imports Iguales-Elementos With [ Caracter For T ]
  Introduces
         Nombre : => Cadena
     AgregaCarN : Cadena, Caracter => Cadena
     NombresIguales : Cadena, Cadena =>  Booleano
  Constrains NombresIguales so that
      Cadena generated by [ Nombre, AgregaCarN ]
      Cadena partitioned by [ NombresIguales ]
      For all [ n1, n2 : Cadena; @, $ : Caracter ]
```

```
        NombresIguales(AgregaCarN(n1, @), AgregaCarN(n2, $)) =
                        @ == $  & NombresIguales( n1, n2 )
        NombresIguales( Nombre, Nombre ) = Cierto
        NombresIguales( AgregaCarN( n1, @), Nombre) = Falso ....1
        NombresIguales( Nombre, AgregaCarN( n2, $)) = Falso ....2
    Implies Converts[ NombresIguales ]
```

Especificación de la característica Nombres.

```
  Type Cadena Export IniciaNombre, RegistraNombre,
       MismaPersona.

       Based on Sort Cadena From Nombres

       Procedure Inicia-Nombres( Var Nom : Cadena );
                 ensures ( Nom post= Nombre )
          end;

       Procedure Registra-Nombre( Var Nom : Cadena;
                                  @ : Caracter );
                modifies at most[ Nom ]
                ensures ( Nom post = AgregaCarN( Nom, @) )
          end;

       Function Mismo-Nombre( Nom1, Nom2 : Cadena ) : Boolean;

           ensures ( MismoNombre = NombresIguales( Nom1, Nom2) )
          end;
  End Cadena
```

Especificación del TDI Cadena.

4.2 Implantando la Especificación.

Una vez descritas las especificaciones de Interfaz y características que conforman la especificación y realizadas en ella las pruebas correspondientes, se realizará ahora la implantación en el lenguaje Pascal (Versión Turbo 3.0) de nuestro ejemplo.
Es en esta etapa en donde los progamadores establecen como deben ser realizadas las operaciones requeridas sobre los TDI especificados a través de instrucciones programables de Pascal.

Primeramente, para realizar la implantación de la especificación en Pascal, se debe elegir la Estructura de Datos más apropiada para representar los TDI, tomando en consideración las operaciones a realizar en él. Posiblemente aquí existan limitaciones para representar completamente él o los TDI, así como restricciones para manejar libremente las funciones.
Estas limitaciones y restricciones deben reportarse claramente en el manual de especificación del usuario.

Después, deben programarse las funciones y procedimientos definidos en la especificación de Interfaz en Pascal, asumiendo primero las proposiciones establecidas en la claúsula requires.
En ocaciones es necesario agregar aquí uno o varios procedimientos o funciones adicionales, con el fin de hacer más clara la implantación de la especificación, como por ejemplo los que se realizan para inicialización de variables y registros o para llevar un control de flujo del programa, etc.

Finalmente, se deben programar las funciones establecidas en estas rutinas y definidas en la característica correspondiente al TDI a implantar. Es decir, una vez entendida la semántica de cada una de las funciones, se debe elegir el algoritmo apropiado para representar estas funciones (algunas de ellas sirvieron sólo como auxiliares, es decir, para contribuir a la especificación de otras funciones y que, no requieren ser implantadas o programadas directamente). Finalmente se elige el conjunto de variables o estructuras de datos auxiliares que representarán este algoritmo en el programa.
Aquí, los programadores están en libertad de elegir la implantación de estas funciones, para lograr la ejecución de las operaciones sobre el TDI. Pero las decisiones importantes sobre las funciones y su comportamiento se hicieron antes por los diseñadores en la especificación.

El éxito de la implantación de la especificación, dependerá de la habilidad y experiencia de los programadores para elegir la correcta estructura de datos para representar los TDI, implantarla correctamente y programar adecuadamente las funciones de las características.

En nuestro ejemplo, se eligirá como estructura de datos para representar el TDI ArchGrups, un archivo de registros, donde cada registro contendrá el nombre de un archivo de alumnos; para repre

sentar el TDI ArchAlums, también se eligirá un archivo de registros donde cada registro contendrá la información referente a un alumno: su nombre, el número de calificaciones parciales que tiene así como sus calificaciones parciales y el promedio de éstas.

Se eligieron estas estructuras de datos en ambos casos, porque resultan ser fáciles de emplear en Pascal y las más apropiadas para representar las operaciones que se desean realizar sobre estos TDI.
Por ejemplo, para editar un archivo de alumnos se requiere que al proporcionar su nombre se cree, si no existe y después se asigne al programa como un archivo de trabajo. Estas operaciones pueden ser implantadas con la instrucción Assign de Pascal (todos los archivos de Alumnos estarán almacenados en disco).
También si se desea saber en cualquier momento el tamaño del archivo o si está vacío, con esta implantación basta con considerar la función de Pascal FileSize, la cual tiene el mismo propósito que la función TotalAlum. Y si se deseara agregar el nombre de un archivo o el de un alumno, sus calificaciones y promedio al archivo correspondiente, puede realizarse esta operación con la función Write(<Nombre del archivo>, <elemento>), que escribe (agrega) en el archivo dado el tipo de elemento (en este caso un registro) a insertar.

Otra ventaja más que representa esta implantación, es que permite tener enumerados secuencialmente conforme fueron agregados, los registros que contienen la información referente a un alumno o del nombre de un grupo, lo que da la ventaja de poder recorrer el archivo fácilmente. Además permite accesar, dado un número de registro contenido en el archivo (que se realiza a través de un apuntador), cualquier registro a través de la función implantada Seek. Esto permite facilitar las operaciones para eliminar un alumno del archivo, modificar su nombre así como sus calificaciones parciales, ya que todas ellas se realizan a través de un número, que será precisamente el de su registro en el archivo.

Para representar al TDI Cadena se eligirá al tipo String, implantado por Pascal.
Esta implantación permite almacenar en una localidad de este tipo, secuencias de caracteres, que es lo requerido por el TDI Cadena. La ventaja que tiene esta implantación, es que no requiere de un procedimiento o función para inicializarse ni para agregar en él los caracteres que uno desea. Simplemente basta con declarar una localidad de este tipo y ser asignada a una secuencia de caracteres (a través de una lectura o asignación) para realizar estas dos operaciónes. Por lo tanto en la implantación no se programarán los procedimientos definidos en la especificación de Interfaz correspondiente al TDI Cadena (IniciaNombres y RegistraNombre).

Las limitaciones que reportan estas estructuras de datos sobre los TDI considerados, son las siguientes.
Respecto a la representación de los TDI ArchGrups y ArchAlum, el número de registros que contendrá cada archivo estará limitado por el tamaño de espacio disponible que haya en el disco donde vayan a ser almacenados. Y respecto a la implantación del tipo

String, es necesario especificar un número fijo de caracteres que contendrán los TDI NomAlum y NomArch. Para el primero se especificará como máximo 30 caracteres y para el segundo 10.

Respecto a la programación de los procedimientos en la implantación, estos serán llamados a través de menús, los cuales llevarán el control de la secuencia del programa que ejecutará la especificación. El esquema de esta especificación mostrado al principio de este capítulo, permite distinguir los tres menús, llamados de acuerdo a cada llave descrita, Edita-Archivo, Edita-Alumnos y Edita-Calif. Estos deberán implantarse (lo cual no se hará aquí) como procedimientos auxiliares para hacer más clara la implantación y ejecución de las operaciones en los archivos.
Las cláusulas requieres al ser especificadas en estos procedimientos, definen las pre-condiciones bajo las cuales deberan ser utilizados y en consecuencia tomadas en cuenta para elegir el menú que los llamará.
Por ejemplo, en el procedimiento de la especificación de Interfaz de la operación para obtener el promedio de las calificaciones parciales para cada alumno· la cláusula requieres establece como pre-condición para usar este procedimiento, que la última operación ejecutada en el archivo de alumnos, no haya sido la creación de él, ni la de agregado alumnos (que implica la ejecución también de las operaciones de modificar y eliminar alumnos) ni obtenido ya el promedio de las calificaciones parciales, por lo que este procedimiento sólo deberá de ser llamado por el menú Edita-Calif, ya que en él sólo se describen las operaciones a realizar en el archivo que cumplen las pre-condiciones para aplicar este procedimiento.

En relación a la programación de las funciones especificadas en nuestro ejemplo, no hay realmente un algoritmo definido para realizar alguna de ellas que deba ser programado. Por ejemplo de ordenar alfabéticamente los nombres de los alumnos, o de seleccionar y buscar alguno de ellos, etc. En el caso de la operación para saber si el nombre de un archivo está, la búsqueda en el archivo se hará secuencialmente. Las funciones definidas recursivamente, como AgregaAlumN-Vec, ModifNomAlumN-Vec, EliminaAlumN-Vec, etc. que establecen ser utilizadas un número dado de veces (n), se programarán a través de la proposición: For ... To ... Do.

La programación de la función EliminaAlum, que se hace a través del procedimiento EliminaN-NomAlum, y cuyo fin es eliminar del archivo especificado un alumno, requiere de un algoritmo especial debido a la implantación. Debido a que no existe alguna función incorporada de Pascal que permita eliminar un registro aleatoriamente, es decir un alumno del archivo, se seguira el siguiente algoritmo.
Para eliminar un registro (un alumno) si éste no es el último, se recorren un lugar hacia arriba a partir del siguiente, todos los registros, borrando el registro a eliminar 'encimando' en él otro registro. Luego se realiza en el archivo la operación dada por la función incorporada de Pascal Truncate, cuyo propósito es eliminar del archivo todos los registros a partir de donde señala actualmente el apuntador de registros del archivo, en este caso

el último registro que sobra, (ya que supuestamente se dejo señalado el apuntador aquí).
Si el alumno a eliminar es el último, se realiza sólo esta última operación. Para permitir una estructuración más clara de la implantación de esta función, un procedimiento llamado Actualiza-Archivo, se encargará de realizar el proceso de recorrer cada registro y realizar la eliminación del último registro del archivo.

Finalmente, una última observación respecto a la implantación de la especificación de nuestro ejemplo, es que se desea que el programa que ejecutará la especificación sea 'amistoso' al usuario, es decir, debido a la iteracción entre éste y el usuario, ya que de acuerdo a lo especificado, el programa debe saber cuantos nombres de alumnos se van agregar, eliminar o modificar, etc. o saber el nombre y calificación de un alumno, se quiere que esta información sea transferida al programa a través de preguntas concisas y amistosas. Además, se desea se que desplieguen mensajes de cuando ha terminado de realizar una operación sobre los archivos o cuando no es posible realizar alguna de estas operaciones en los archivos (establecidas por las cláusulas signals).

La implantación de las rutinas de las operaciones a realizar en la especificación de este ejemplo se muestra en el apéndice D.

CONCLUSIONES.

Como conclusiones de este trabajo se pueden enunciar las siguientes características de la especificación formal:

La especificación formal de programas es una área de investigación de la computación recientemente abordada y que lleva algunos años de esfuerzo en concepción y formación.

Esta manera de especificar programas comprende la aplicación de métodos y técnicas rigurosas para describir de manera clara y concisa lo que debe hacer un programa o sistema.

Otra característica es que permite detectar un entendimiento incompleto o confuso que pudiera existir sobre los requerimientos de un programa, advirtiendo a los diseñadores de programas y sistemas de estos problemas a un nivel de usuario más que de programación. Al especificar un sistema o programa se tiene la ventaja de poder desarrollar conceptos abstractos o ideas para su construcción y uso, que son frecuentemente más generales que el sistema o programa mismo, permitiendo así la especificación generalizada de éstos, los cuales pueden volverse a emplear para desarrollar futuros sistemas.

Además, la especificación formal proporciona herramientas y medios para verificar lo correcto de un programa antes de realizar la implantación del sistema, a través de pruebas y métodos formales de verificación.

Una ventaja más de esta manera de especificación, es que sirve para establecer un 'contrato' o acuerdo entre el usuario y el programador, ya que en él se establecen de manera clara y precisa los requerimientos y demandas del usuario, evitando así problemas de implantación y desarrollo de sistemas inútiles.

Los lenguajes de especificación constituyen uno de los métodos de especificación formal más usual, por su no muy difícil entendimiento y amplia flexibilidad en la especificación de programas y sistemas, así como por el lenguaje y rigor matemático que emplean, lo cual los hace ser eficientes y confiables.

Un lenguaje de especificación es un conjunto de expresiones e instrucciones especiales que permiten representar especificaciones de programas, es decir, las características y propiedades relevantes de lo que debe de hacer, bajo un cierto contexto y con formalismo matemático.

En este trabajo se estudiaron la familia de lenguajes de especi

ficación Larch y el lenguaje OBJ. Ambos lenguajes permiten construir especificaciones algebraicas de programas en los cuales los datos abstractos juegan un papel prominente.

Se emplearon ambos lenguajes para especificar un sistema en particular (un directorio telefónico), y además se utilizó el lenguaje Larch para especificar un sistema más complejo hasta su implantación en Pascal (el de un archivo de alumnos y sus calificaciones). En ambos casos, el lograr construir una especificación 'correcta' no fue sencillo. No fue sino hasta después de haber realizado varios intentos de especificación, que se obtuvo una especificación satisfactoria.

Esta experiencia me permite concluir que:

- En principio, es indespensable entender bien y completamente lo que se desea especificar, ya que durante la especificación surgieron algunos problemas (como algunas funciones sin sentido o incompletas) debido a que aún a través de frases concisas, no se tenía claro lo que se quería especificar, por lo que fue conveniente realizar una discusión más detallada con el usuario.

- Existen algunas funciones que por su simple definición a través de frases concisas, parecen fáciles de especificar (a través de una sola función), pero que realmente requieren de varias funciones para su especificación. Esto originó algunas complicaciones en el desarrollo de las especificaciones, pero se solucionaron al considerar que una operación puede requerir de varias funciones para su especificación.

- Resulta más fácil emplear especificaciones ya construidas y separar la especificación en módulos. La escritura y definición de ambos sistemas desarrollados y en los dos lenguajes de especificación, se facilitó bastante al construir las especificaciones de manera estructurada utilizando cosas u objetos ya construidos o definidos.

En general, la dificultad que en algunos casos ofreció la especificación de ambos sistemas, no se debió al lenguaje de especificación empleado. Ambos lenguajes ofrecen construcciones y expresiones útiles para la especificación, lo que les hace ser claros y fáciles de utilizar. Aunque noté que por ejemplo en Larch, por sus mecanismos de asumir características ya definidas, fue más fácil construir especificaciones que en OBJ, quien realiza más bien estos mecanismos a través de tipos ya construidos, lo cual hizo dificultarme un poco la identificación de éstos en el desarrollo de las especificaciones.

A P E N D I C E   A

Especificación de una Relación de Orden Total en el Lenguaje Larch Compartido.

```
Relación : Trait
   Introduces
      # r # : T, T  ->  Booleano


Reflexiva : Trait
   Includes  Relación
   Constrains for all [ x : T ]
      x r x  =  Cierto


Antisimétrica : Trait
   Includes  Relación
   Constrains for all [ x, y : T ]
      ( x r y ) & ( y r x )  =>  ( y = x )


Transitiva : Trait
   Includes  Relación
   Constrains for all [ x, y, z : T ]
   ( x r y ) & ( y r z )  =>  ( x r z )


Orden-Parcial : Trait
   Imports  Reflexiva With [ ≲  For r ]
            Antisimétrica With [ ≲  For r ]
            Transitiva With [ ≲  For r ]


Relación-Total : Trait
   Includes Relación
   Constrains for all [ x, y : T ]
       x r y | y r x   =   Cierto


Orden-Total : Trait
   Imports  Orden-Parcial, Relación-Total With [ ≲  For r ]
```

A P E N D I C E B

Especificacioń Completa del Ejemplo de un Directorio en el lenguaje Larch Compartido y de Interfaz.

Especificación en el lenguaje Larch compartido del TDI Dir

```
Directorio : Trait
   Include Nombre-De-Personas with [ NomPers for Cadena ]
   Include Direcc-Tel-De-Personas with [ DirecTel for Cadena ]
   Introduces
        SecreaDirect : => Dir
          AgregaPers : Dir, NomPers, DirecTel => Dir
       ObtenDirecTel : Dir, NomPers => DirecTel
           Estéen : Dir, NomPers => Booleano
         EliminaPers : Dir, NomPers => Dir
       ModifDirecTel : Dir, NomPers, DirecTel => DirecTel
        Dimensión : Dir => Numero Cardinal
   Constrains Dir so that
      Dir generated by [ SecreaDirect, AgregaPers ]
      Dir partitioned by [ ObtenDirecctel ]
   For all [D : Dir; N1, N2 : NomPers; Dt1, Dt2 : DirecTel]

      ObtenDireccTel( AgregaPers( D, N1, Dt1 ), N2 ) =
                                         If NomIguales( N1, N2 ) Then
                                            Dt1
                                         else
                                           ObtenDireccTel( D, N2 )
      Estéen( AgregaPers( D, N1, Dt1 ), N2 ) =
                            NomIguales( N1, N2 ) ! Estéen( D, N2 )
      EliminaPers( AgregaPers( D, N1, Dt1 ) N2 ) =
                                 If NomIguales( N1, N2 ) Then
                                   D
                                 else
                         AgregaPers( EliminaPers( D, N2 ), N1, Dt1 )
      ModifDireccTel( AgregaPers( D, N1, Dt1 ), N2, Dt2 ) =
                                        If NomIguales( N1,  N2 ) Then
                                           AgregaPers(  D, N2, Dt2 )
                                      . else
                AgregaPers( ModifDireccTel( D, N2, Dt2 ), N1, Dt1)
      Dimensión( SecreaDirect ) = 0
      Dimensión( AgregaPers( D, N1, Dt1)) = 1 + Dimensión( D )
      Estéen( SecreaDirect, N1 ) = Falso

   Exemp̃ts[ ObtenDireccTel( SecreaDirec, N1 ),
            EliminaPers( SecreaDirec, N1 ),
              ModifDireccTel( SecreaDirec, N1, Dt1 ) ]
   Implies Converts[ ObtenDireccTel, EliminaPers, Estéen,
                       ModifDireccTel, Dimensión ]
```

Especificación de Interfaz del TDI Direct.

```
Type Direct Exports  CreaDirectorio, AgregaNomDirecTel,
          DireccTel, EstáPersona, EliminaPersona,
          ModifDirecTel, Tamañ oDirectorio
     Based on Sorts Dir From Directorio

     Procedure CreaDirectorio( Var D : Direct )
                  ensures ( D post = SecreaDirect )
        end;


     Procedure AgregaNomDirecTel( Var D : Direct; Np : NomPers;
                                  Dt : DirecTel );
                             Signals( Ya-Está-Persona )
               modifies at most  [ D ]
               ensures If ¬Estáen( D, Np ) Then
                                 D post = AgregaPers( D, Np,Dt )
                       else
                         Signals( Ya-Está-Persona ) &
                         modifies nothing
         end;

     Function DireccTel( D : Direct; Np : NomPers ) : DirecTel
                  Signals( No-Está-Persona, Directorio-Vacío )
              ensures If D = SecreaDirect Then
                         Signals( Directorio-Vacío ) &
                         modifies nothing
                       else
                         If Estéen( D, Np ) Then
                           DireccTel = ObtenDireccTel( D, Np)
                         else
                           Signals( No-Está-Persona ) &
                           modifies nothing
        end;

     Function EstáPersona( D : Direct; Np : NomPers ) : Boolean
               ensures EstáPersona = Estéen( D, Np )
        end;

     Procedure EliminaPersona( Var D : Direct; Np : NomPers)
                 Signals( No-Está-Persona, Directorio-Vacío)
               modifies at most [ D ]
               ensures If D = SecreaDirect Then
                          Signals( Directorio-Vacío ) &
                          modifies nothing
                       else
                          If Estéen( D, Np ) Then
                            D post = EliminaPers( D, Np )
                          else
                            Signals( No-Está-Persona ) &
                            modifies nothing
        end;
```

```
  Procedure ModificaDireccTel( Var D : Direct; Np : NomPers;
                               Dt : DirecTel);
          Signals( No-Está-Persona, Directorio-Vacío )
          modifies at most [ D ]
          ensures If D = SecreaDirect Then
                       Signals( Directorio-Vacío ) &
                       modifies nothing
                  else
                     If Estéen( D, Np ) Then
                        D post = ModifDireccTel( D, Np, Dt )
                       else
                        Signals( No-Está-Persona ) &
                        modifies nothing
     end;

  Function TamañoDirectorio( D : Direct ) : Integer;
          ensures TamañoDirectorio = Dimensión( D )
     end;

End Direct
```

Especificación en el lenguaje Larch Compartido del TDI Caracter.

```
Caracteres-Válidos : Trait
    Imports Iguales-Elementos with [ Caracter  For T ]
    Introduces
       CaracterVálido : Caracter => Booleano
    Constrains CaracterVálido so that
       Caracter partitioned by # == #
       Caracter generated by [ A,B,....Z,0,1,....9,....,
                               #,$,^,[,(....+,! ]
       for all [ Car : Caracter ]
           CaracterVálido( Car ) = Car == A | Car == B | ... |
                                   Car == Z | Car == 0 |
                                   Car == 1 | ... | Car == 9 |
                                   Car == . | Car == # |
                                   Car == -
```

Especificación en el lenguaje Larch Compartido del TDI Cadena.

```
Nombre-De-Personas : Trait
   Asumes Caracteres-Validos
   Imports Iguales-Elementos with [ Caracter For T ]
   Introduces
         Nombre : => Cadena
      AgregaCarN : Cadena, Caracter => Cadena
      NomIguales : Cadena, Cadena =>  Booleano
   Constrains NomIguales so that
       Cadena generated by [ Nombre, AgregaCarN ]
       Cadena partitioned by [ NomIguales ]
       for all [ n1, n2 : Cadena; @, $ : Caracter ]
```

```
      NomIguales(AgregaCarN(n1, @), AgregaCarN(n2, $)) =
                                 @ == $  &  NomIguales( n1, n2 )
      NomIguales( Nombre, Nombre ) = Cierto
      NomIguales( AgregaCarN( n1, @), Nombre) = Falso .... 1
      NomIguales( Nombre, AgregaCarN( n2, $)) = Falso .... 2
   Implies Converts[ NomIguales ]
```

Especificación de Interfaz del TDI NomPers.

```
Type NomPers Exports IniciaNombre, RegistraNombre,
           MismaPersona
     Based on Sort Cadena From Nombre-De-Personas

     Procedure IniciaNombre( Var Np : NomPers );
               ensures( Np post = Nombre )
        end;

     Procedure RegistraNombre( Var Np : NomPers; @ : Caracter);
                                 Signals( Caracter-Inválido )
               modifies at most [ Np ]
               ensures If CaracterVálido( @ ) Then
                             Np post = AgregaCarN( Np, @ )
                         else
                             Signals( Caracter-Inválido )
        end;

     Function MismaPersona( Np1, Np2 : NomPers ) : Boolean
              ensures MismaPersona = NomIguales( Np1, Np2 )
        end;

   End NomPers
```

Especificación en el lenguaje Larch Compartido del TDI DirTel.

```
Direcc-Tel-De-Personas : Trait
    Imports Caracteres-Válidos
    Introduces
            DirTel : => Cadena
       AgregaCarDt : Cadena, Car => Cadena
     Constrains Cadena so that
         Cadena generated by [ DirTel, AgregaCarDt ]
```

Especificación de Interfaz del TDI DirecTel.

```
Type DirecTel Exports IniciaDirecTel RegistraDirecTel
     Based on Sort Cadena From Direcc-Tel-De-Personas

     Procedure IniciaDirecTel( Var Dt : DirecTel );
               modifies at most [ Dt ]
               ensures( Dt post = DirTel )
        end;
```

```
     Procedure RegistraDirecTel( Var Dt : DirecTel;
                                  @ : Caracter );
                               Signals( Caracter-Inválido )
               modifies at most [ Dt ]
               ensures If CaracterVálido( @ ) Then
                           Dt post = AgregaCarDt( Dt, @ )
                        else
                          Signals( Caracter-Inválido )
          end;

  End DirecTel
```

A P E N D I C E C

Especificación de la Característica Archivo-Alumnos en el Lenguaje Larch Compartido y de Interfaz.

Especificación en el lenguaje Larch Compartido del TDI ArchAlums.

```
Archivo-Alumnos : Trait
   Includes Archivo-Grupos
   Includes Nombres With [ NomGrup, NomAlum For Cadena ]
   Introduces

      CreaArchAlum : NomGrup => ArchAlums
      EditArchAlum : NomGrup => ArchAlums
      LeeCad : Cadena => Cadena
      LeeEnt : Entero => Entero
      LeeReal : Real => Real
      AgregaAlumN-Vec : ArchAlums, Entero => ArchAlums
      ModifNomAlumN-Vec : ArchAlums, Entero => ArchAlums
      EliminaAlumN-Vec : ArchAlums, Entero => ArchAlums
      AgregaAlum : ArchAlums, NomAlum => ArchAlums
      ModifNomAlum : ArchAlums, Entero, NomAlum => ArchAlums
      EliminaAlum : ArchAlums, Entero => ArchAlums
      NúmAlum : ArchAlums, NomAlum => Entero
      EstáNúmAlum : ArchAlums, Entero => Booleano
      TotalAlum : ArchAlums => Entero
      Alum : ArchAlums, Entero => NomAlum
      AgregaCalifs : ArchAlums, Entero, Entero => ArchAlums
      AgregaCalif : ArchAlums, NomAlum, Entero,
                    Entero  => ArchAlum
      Escribe : NomAlum => NomAlum
      NúmParc : ArchAlums => Entero
      EstáParc : ArchAlums, Entero => Booleano
      ModifCalifsParcN-Vec : ArchAlums, Entero,
                             Entero => ArchAlums
      ModifCalifN-Vec : ArchAlums, Entero => ArchAlums
      ModifCalif : ArchAlums, Entero, Entero => ArchAlums
      ObtenProm : ArchAlums, Entero => ArchAlums
      AgregaProm : ArchAlum, Entero, Real => ArchAlums
      SumaCalif : ArchAlums, Entero => Real
      EstaNomAlum : ArchAlums, NomAlum => Boolean

   Constrains ArchAlums so that
      ArchAlum Generated by [ CreaArchAlum, AgregaAlum,
                              AgregaCalif, AgregaProm ]
      For all [ AA : ArchAlums; Ali, Alj : NomAlum;
                NoAj, NoAi : Entero ( >0 ); n : Entero ( >0 );
                Pi, Pj : Entero ( >0 ); Ci, Cj : Real;
                Pr : Real ]
      AgregaAlumN-Vec( AA, n ) =
```

```
   If n = 1 Then
      AgregaAlum( AA, LeeCad( Ali ) )
   else
     AgregaAlumN-Vec( AgregaAlum( AA, LeeCad(Ali) ), n - 1 )

ModifNomAlumN-Vec( AgregaAlum( AA, Ali ), n ) =
   If n = 1 Then
      ModifNomAlum( AgregaAlum( AA, Ali ), LeeEnt( NoAi ),
                    LeeCad( Alj ) )
   else
     ModifNomAlumN-Vec( ModifNomAlum( AgregaAlum( AA, Ali ),
                        LeeEnt(NoAi), LeeCad(Alj) ), n - 1 )

ModifNomAlum( AgregaAlum( AA, Ali ), NoAi, Alj ) =
   If NumAlum( Agrega( AA, Ali ), Ali ) = NoAi Then
      AgregaAlum( AA, Alj )
   else
     AgregaAlum( ModifNomAlum( AA, NoAi, Alj ), Ali )

EliminaAlumN-Vec( AgregaAlum( AA, Ali ), n ) =
   If n = 1 Then
      EliminaAlum( AgregaAlum( AA, Ali ), LeeEnt(NoAi) )
   else
     EliminaAlumN-Vec( EliminaAlum( AgregaAlum( AA, Ali ),
                       LeeEnt( NoAi ) ), n - 1 )

EliminaAlum( AgregaAlum( AA, Ali ), NoAi ) =
   If NumAlum( AgregaAlum( AA, Ali ), Ali ) = NoAi Then
      AA
   else
     AgregaAlum( EliminaAlum( AA, NoAi ), Ali )

NúmAlum( AgregaAlum( AA, Ali), Alj) =
   If NombresIguales( Ali, Alj ) Then
      TotalAlum( AA ) + 1
   else
     NumAlum( AA, Alj)

EstáNúmAlum( AgregaAlum( AA, Ali ), NoAi ) =
   If 0 < NoAi <= TotalAlum( AA ) + 1 Then
      Verdadero
   else
     Falso

Alum( AgregaAlum( AA, Ali), NoAi) =
   If TotalAlum( AA ) + 1 = NoAi Then
      Ali
   else
     Alum( AA, NoAi)

TotalAlum( CreaArchAlum ) = 0

TotalAlum( AgregaAlum( AA, Ali)) = 1 + TotalAlum( AA)

AgregaCalifs( AA, NoAi, Pi) =
```

```
      NoAi = ToTalAlum( AA ) Then
      AgregaCalif( AA, Escribe( Alum( AA, NoAi)),
                   LeeReal( Ci ), Pi )
   else
     AgregaCalifs( AgregaCalif( AA, Escribe( Alum( AA,
                   NoAi) ), LeeReal(Ci), Pi), NoAi + 1, Pi )

TotalAlum( AgregaCalif( AA, Alum( AA, NoAi), Ci, Pi) ) =
           TotalAlum( AA )

Alum( AgregaCalif( AA, Ali, Ci, Pi), NoAi) = Alum( AA )

ModifCalifsParcN-Vec( AgregaCalif( AA, Ali, Ci, Pi ),
                      Pj, n ) =
   If Pi = Pj Then
      ModifCalifN-Vec( AgregaCalif( AA, Ali, Ci, Pi ), n )
   else
     AgregaCalif( ModifCalifsParcN-Vec( AA, Pj, n), Ali,
                  Ci, Pi )

ModifCalifN-Vec( AgregaCalif( AA, Ali, Ci, Pi ), n) =
   If n = 1 Then
      ModifCalif( AgregaCalif( AA, Ali, Ci, Pi ),
                  LeeEnt( NoAj ), LeeReal( Cj ) )
   else
     ModifCalifN-Vec( ModifCalif( AgregaCalif( AA, Ali, Ci,
                      Pi ), LeeEnt( NoAj ), LeeReal( Cj ) ),
                      n - 1 )

ModifCalif( AgregaCalif( AA, Alum( AA, NoAi), Ci, Pi),
            NoAj, Cj) =
   If NoAi = NoAj Then
      AgregaCalif( AA, Alum( AA, NoAi), Cj, Pi)
   else
     AgregaCalif( ModifCalif( AA, NoAj, Cj), Alum( AA,
                  NoAi), Ci, Pi )

NúmParc( AgregaCalif( AA, Ali, Ci, Pi ) ) =
   If Pi = 1 Then 1
   else
     1 + NúmParc( AA )

EstáParc( AgregaCalif( AA, Ali, Ci, Pi ) ), Pj) =
   If NúmParc( AgregaCalif( AA, Ali, Ci, Pi ) ) >= Pj
      Then  Verdadero
   else
     Falso

EstáNúmAlum( AgregaCalif( AA, Ali, Ci, Pi), NoAi ) =
   EstáNúmAlum( AA, NoAi )

ObtenProm( AA, NoAi) =
   If NoAi = TotalAlum( AA) Then
      AgregaProm( AA, NoAi, SumaCalif( AA,
                  NoAi) / NúmParc( AA))
```

```
      else
        ObtenProm( AgregaProm( AA, NoAi, SumaCalif( AA,
                         NoAi) / NúmParc( AA ) ), NoAi + 1 )

   SumaCalif( AgregaCalif( AA, Alum( AA, NoAj), Ci, Pi ),
           NoAi) =
      If Pi = 1 Then
         If NoAi = NoAj Then
            Ci
         else
           SumaCalif( AA, NoAi)
      else
        If NoAi = NoAj Then
           Ci + SumaCalif( AA, NoAi)
        else
          SumaCalif( AA, NoAi)

   TotalAlum( AgregaProm( AA, NoAi, Pr ) ) = TotalAlum( AA )

   SumaCalif( AgregaProm( AA, NoAi, Pr ) ) = SumaCalif( AA )

   NúmParc( AgregaProm( AA, NoAi, Pr ) ) = NúmParc( AA )

   EstáNomAlum( CreaArchAlum, Ali) = Falso

   EstáNomAlum( AgregaAlum( AA, Ali), Alj) =
      NombresIguales( Ali, Alj) | EstáNomAlum( AA, Alj)

   EstáNomAlum( AgregaCalif( AA, Ali, Ci, Pi), Alj) =
      NombresIguales( Ali, Alj ) | EstáNomAlum( AA, Alj )


Exempts[ AgregaAlumN-Vec( AgregaCalif( AA, Ali, Ci, Pi ), n),
         AgregaAlumN-Vec( AgregaProm( AA, NoAi, Pr ), n),
         ModifNomAlumN-Vec( CreaArchAlum, n),
         ModifNomAlumN-Vec( AgregaCalif( AA, Ali, Ci, Pi ), n),
         ModifNomAlumN-Vec( AgregaProm( AA, NoAi, Pr ), n),
         EliminaAlumN-Vec( CreaArchAlum, n),
         EliminaAlumN-Vec( AgregaCalif( AA, Ali, Ci, Pi ), n),
         EliminaAlumN-Vec( AgregaProm( AA, NoAi, Pr ), n),
         AgregaCalifs( CreaArchAlum, NoAi, Pi ),
         AgregaCalifs( AgregaProm( AA, NoAi, Pr ), NoAi, Pi ),
         ModifCalifsParcN-Vec( CreaArchAlum, Pi, n ),
         ModifCalifsParcN-Vec( AgregaAlum( AA, Ali), Pi, n ),
         ModifCalifsParcN-Vec( AgregaProm( AA, NoAi, Pr ),
                               Pi, n ),
         ObtenProm( CreaArchAlum, NoAi ),
         ObtenProm( AgregaAlum( AA, Ali ), NoAi),
         ObtenProm( AgregaProm( AA, NoAi, Pr), NoAi )
         EstáNomAlum( AgregaProm( AA, NoAi, Pr ), NoAj ) ]



Implies Converts[ LeeCad, LeeEnt, LeeReal, NúmAlum,
```

```
                EstáNúmAlum, TotalAlum, Alum, Escribe,
                NúmParc, EstáParc, ObtenProm, AgregaProm,
                SumaCalif, EstáNomAlum ]
```

Especificación en el lenguaje de Interfaz del TDI ArchAlums.

```
Type ArchAlums Exports EditaArchAlums, Agrega-N-Alumnos,
               Modifica-N-Nombres, Elimina-N-Nombres,
               Agrega-Calif-Parc, Modif-N-CalifsParc,
               Obten-Promedios, Está-Nombre-Alumno.

     Based on sort ArchAlums From Archivo-Alumnos
              With [ AA' : ArchAlums ]
     Based on sort ArchGrups From Archivo-Grupos
              With [ AG' : ArchGrups ]
     Based on sort Cadena From Nombres
              With [ NG : NomGrup For Cadena ]

     Procedure CreaArchivoAlumnos( Var AA : ArchAlums;
                                       NG : NomGrup );
               ensures AA post  = CreaArchAlum( NG )
        end;

     Procedure EditaArchAlums( Var AG : ArchGrups;
                               Var AA : ArchAlums );
               requieres (AG = CreaArchGrups or
                             Not EstaVacío( AG )
               modifies at most [ AG ]
               ensures If Not EstáNomGrup(AG, LeeCad( NG )) Then
                          AgregaNomGrup( AG,NG ) &
                             AA post  = CreaArchAlum( NG )
                       else
                         AA post = EditArchAlum( NG )
        end;

     Procedure Agrega-N-Alumnos( Var AA : ArchAlums );
               Signals(Número-No-Válido)
               requieres AA <> AgregaCalif( AA', Ali, Ci, Pi ) &
                            AA <> AgregaProm( AA', NoAi, Pr )
               modifies at most[ AA ]
               ensures If LeeEnt( n ) <= 0 Then
                          Signals(Número-No-Válido) &
                             modifies nothing
                       else
                         AA post = AgregaAlumN-Vec( AA, n )
        end;

     Procedure Modifica-N-Nombres( Var AA : ArchAlums );
               Signals( Archivo-Vacío, Número-No-Válido )
               requieres AA <> AgregaCalif( AA', Ali, Ci, Pi ) &
                            AA <> AgregaProm( AA', NoAi, Pr )
               modifies at most [ AA ]
               ensures If LeeEnt( n ) <= 0 Then
```

```
                    Signals(Número-No-Válido) &
                       modifies nothing
                 else
                   If TotalAlum( AA ) = 0 Then
                        Signals(Archivo-Vacío) &
                           modifies nothing
                   else
                     AA post  = ModifNomAlumn-Vec( AA, n )
   end;

Procedure Elimina-N-Nombres( Var AA : ArchAlums );
          Signals( Archivo-Vacío, Número-No-Válido)
          requieres AA <> AgregaCalif( AA', Ali, Ci, Pi ) &
                       AA <> AgregaProm( AA', NoAi, Pr )
          modifies at most [ AA ]
          ensures If LeeEnt(n) <= 0 Then
                    Signals(Número-No-Válido) &
                       modifies nothing
                 else
                   If TotalAlum( AA ) = 0 Then
                        Signals(Archivo-Vacío) &
                           modifies nothing
                   else
                     AA post  = EliminaAlumN-Vec( AA, n )
   end;

Procedure Agrega-Calif-Parc( Var AA : ArchAlums );
          Signals( Archivo-Vacío, Calif-Parc-No-Procede)
          requieres AA <> AgregaProm( AA', NoAi, Pr )
          modifies at most[ AA ]
          ensures If TotalAlum( AG ) = 0 Then
                       Signals( Archivo-Vacío ) &
                          modifies nothing
                 else
                   If LeeEnt(Pi) <= 0  Or  Pi > 5 Then
                        Signals( Calif-Parc-No-Procede ) &
                           modifies nothing
                   else
                     AA post  = AgregaCalifs( AA, i, Pi)
   end;

Procedure Modif-N-CalifsParc( Var AA : ArchAlum);
          Signals( Parcial-No-Está, Número-Negativo)
          requieres AA <> CreaArchAlum,
                       AA <> AgregaAlum( AA', Ali ) &
                       AA <> AgregaProm( AA', NoAi, Pr )
          modifies at most [ AA ]
          ensures If LeeEnt( n ) <= 0 Then
                       Signals( Número-Negativo ) &
                          modifies nothing
                 else
                   If Not EstáParc( AA, LeeEnt( Pi ) ) Then
                      Signals( Parcial-No-Está ) &
                         modifies nothing
                   else
```

```
                          AA post  = ModifCalifsParcN-Vec( AA,
                                      Pi, n)
    end;

 Procedure Obten-Promedios( Var AA : ArchAlums );
           requieres AA <> CreaArchAlum,
                        AA <> AgregaAlum( AA', Ali ) &
                        AA <> AgregaProm( AA', NoAi, Pr)
           modifies at most [ AA ]
           ensures AA post = ObtenProm( AA, 1 )
    end;

 Procedure Está-Nombre-Alumno( AA : ArchAlums, Ali : NomAlum);
           Signals( Está-el-Alumno, No-Está-el-Alumno)

           ensures If EstáNomAlum( AA, Ali) then
                      Signals( Está-el-Alumno )
                   else
                     Signals( No-Está-el-Alumno )
    end;

End ArchAlums
```

A P E N D I C E   D

Implantación en el Lenguaje Pascal del Ejemplo Desarrollado en el Capitulo Cuatro.

```
Const

  LongNomAr = 10;  {Longitud máxima del nombre de un archivo}
  LongNomAl = 30;  {Longitud máxima del nombre de un alumno}
  NumCalifs = 5;   {Núm. de calif. parciales máximo para un alumno}

Type

  NombreGr = String[ LongNomAr ];
  NombreAl = String[ LongNomAl ];
  RegNomGrup = Record
                 NomGrupo : NombreGr; {Nombre de un grupo de alumnos}
               end;

  RegAlum = Record                   {Información referente a un alumno: }
              NomAlum : NombreAl;         {Nombre}
              NumParc : Byte;             {Núm. de exámen parcial}
              Califs : Array[ 1 .. NumCalifs ] of Real;
                                          {Lista de calificaciones}
              Prom : Real;                {Promedio}
            end;

  ArchivoGrups = File of RegNomGrup; {Archivo de noms. de grupos}
  ArchivoAlums = File of RegAlum;    {Archivo de alumnos}

Procedure Crea_Archivo_Grupos( Var AG : ArchivoGrups );
         {Crea el archivo de nombres de grupos: AG}

  Begin
    ReWrite( AG )
  end;

Procedure Crea_Archivo_Alumnos( Var AA : ArchivoAlums );
         {Crea el archivo de alumnos: AA}

  Begin
    ReWrite( AA )
  end;
```

```
Function Está_Nombre_Grupo( NomArch : NombreGr ) : Boolean;
{Verifica si el nombre de un archivo de grupos (NomArch) }
{ha sido creado}

  Var
    Arch : File;
  Begin
    Assign( Arch, NomArch );
    {$I-}
    Reset( Arch );
    Close( Arch );
    {$I+}
    Esta_Nombre_Grupo :=  Ioresult = 0
  end;

Procedure Agrega_Nombre_Grupo( Var AG : ArchivoGrups;
                               NomGrup : NombreGr );
{Agrega en el archivo AG el nombre de un archivo de }
{alumnos (NomGrup)}

  Var
    unRegNomGrup : RegNomGrup;
                   {Registro del nombre del archivo de alumnos}
      Begin
        unRegNomGrup.NomGrupo := NomGrup;
        SeeK( AG, FileSize( AG ) );
  {Coloca el nombre del archivo simpre al final del archivo AG}
        Write( AG, unRegNomGrup );
        Writeln; Writeln;
        Write(' ' : 25, 'El  ARCHIVO  A  EDITAR  ES  NUEVO')
      end;

Procedure Despl_Nombre_Grupos( Var AG : ArchivoGrups );
{Despliega los nombres de los grupos contenidos en el archivo AG}

  Var
    reg : Integer;  {Enumera cada registro del archivo AG}
    unregNomGrup : RegNomGrup;
     {Selecciona un registro del archivo AG en particular}
  Begin
    Assign( AG, 'NomGrs.Dat' );
    If Not Está_Nombre_Grupo( 'NomGrs.Dat' ) Then
      Crea_Archivo_Grupos( AG )
    else
      Reset( AG );
    If Arch_Grups_Vacio( AG ) Then
       Escr_Archivo_Vacío
    else
      Begin
        For reg := 0 To ( FileSize( AG ) - 1 ) Do
          Begin
            SeeK( AG, reg );
            Read( AG, unRegNomGrup );
            Writeln(unRegNomGrup.NomGrupo );
            Writeln;
```

```
            Delay( 1000 );
          end
      end;
      Close( AG )
  end; {Despl_Nombre_Grupos}

Procedure Edita_Arch_Alums( Var AG : ArchivoGrups;
                            Var AA : ArchivoAlums );
  {Asigna del archivo AG el archivo de alumnos (AA) a trabajar}

  Var
    NomGrup : NombreGr; {Nombre del archivo de alumnos a editar}
  Begin
    Assign( AG, 'NomGrs.Dat');
    If Not Está_Nombre_Grupo( 'NomGrs.Dat' ) Then
      Crea_Archivo_Grupos( AG )
    else
      Reset( AG );
    Write('Nombre del Archivo de Alumnos a Editar: ');
    Read( NomGrup );
    Assign( AA, NomGrup );
    If Not Está_Nombre_Grupo( NomGrup ) Then
    {Verifica si el nombre del grupo dado ya existe}
       Begin
         Agrega_Nombre_Grupo( AG, NomGrup );
         CreaArchAlums( AA );
       end
    else
      Reset( AA )
  end; {Edita_Arch_Alums}

Procedure Escr_Archivo_Vacío;
          {Escribe Archivo Vacio}

  Begin
    Writeln( ' ' : 28, 'A R C H I V O   V A C I O');
    Delay( 5000 )
  end;

Function Arch_Grups_Vacio( Var AG : ArchivoGrups ) : Boolean;
         {Verifica si el Archivo de Grupos está vacío}

  Begin
    Arch_Grups_Vacío := ( FileSize( AG ) = 0 )
  end;

Function Arch_Alums_Vacio( Var AA : ArchivoAlums ) : Boolean;
         {Verifica si el Archivo de Alumnos está vacío}

  Begin
    Arch_Alums_Vacío := (FileSize( AA ) = 0 )
  end;
```

```
Procedure Numero_Invalido;
          {Escribe Número Inválido}

  Begin
    Writeln; Writeln;
    Write(' ' : 26, 'N U M E R O   I N V A L I D O' );
    Delay( 5000 )
  end;

Procedure Num_Alum_No_Esta;
   {Escribe que el número dado de un alumno del archivo}
   {de alumnos, no está en él}

  Begin
    Writeln; Writeln;
    Write(' ' : 24, 'NO  ESTA  EL  NUMERO  DE  ALUMNO');
    Delay( 5000 )
  end;

Function Está_Número_Alumno( Var AA : ArchivoAlums;
                             NoAlum : Integer ) : Boolean;
  {Verifica si el número dado de un alumno (NoAlum) está}
  {en el archivo de alumnos (AA)}

  Begin
    Está_Número_Alumno := ( ( NoAlum > 0 ) And
                          ( NoAlum <= FileSize( AA ) ) )
  end

Procedure Agrega_N_Alumnos( Var AA : ArchivoAlums );
          {Agrega n nombres de alumnos al archivo de alumnos AA}

  Var
    unRegAlum : RegAlum; { Información referente a un alumnos}
    Alum : Integer; {enumera los alumnos a agregar}
    NAlum : Integer; {Número de alumnos a agregar al archivo}
    NúmAlum : Integer; {Número actual de alumnos en el archivo}
  Begin
    Write('Cuántos  Alumnos  Quieres  Agregar ?  ');
    Read(NAlum);
    If ( NAlum <= 0 ) Then
      Número_Invalido
    else
      Begin
        NúmAlum := FileSize( AA );
        For  Alum  := NumAlum To ( NumAlum + NAlum - 1 ) Do
          Begin
 {Agrega los NAlum a partir del último alum. (registro) agregado}
            ClrScr;
            Write('Cual es el Nombre del Alumno');
            Writeln(' a Agregar al Archivo ?');
            Writeln; Writeln;
            With unRegAlum Do
              Begin
                Read( NomAlum );
```

```
              NumParc := 0;
              Prom := -1.0;
            end;
          Seek( AA, Alum );
          Write( AA, unRegAlum )
        end
    end
  end; {Agrega_N_Alumnos}

Procedure Modifica_N_Nombres( Var AA : ArchivoAlums );
  {Modifica n nombres de alumnos en el archivo de alumnos AA}

  Var
    unRegAlum : RegAlum; {Un reg. del archivo AA en particular}
    NúmAlMod : Integer;  {Alumno al cual se va a modif. su nombre}
    Alum : Integer; {Enumera los alumnos a modificar su nombre}
    NAlum : Integer; {Número de alumnos a modificar su nombre}
  Begin
    If Arch_Alums_Vacío( AA ) Then
       Escr_Archivo_Vacío
    else
      Begin
        Write('Cuantos Alumnos Quieres Modificar ? ');
        Read( NAlum );
        If NAlum <= 0 Then
           Número_Invalido
        else
          For Alum := 1 To NAlum Do
            Begin
              ClrScr;
              Writeln;
              Write('Cual es el Numero de Alumno a Modificar ?');
              Read(NumAlMod);
              If Está_Número_Alumno( AA, NumAlMod ) Then
                Begin
                  Seek( AA, NumAlMod - 1 );
                  Read( AA, unRegAlum );
                  Writeln; Writeln;
                  Write('Cuál es el Nuevo Nombre ? ');
                  Read(unRegAlum.NomAlum );
                  {Se modifica su nombre dando otro nombre}
                  Seek( AA, NumAlMod - 1 );
                  Write( AA, unRegAlum );
                end
              else
                Núm_Alum_No_Está
            end
      end
   end; {Modifica_N_Nombres}
```

```
Procedure Actualiza_Archivo( Var AA : ArchivoAlums;
                              Numalum : Integer);
  {Cuando un alumno a través de su número (NúmaLum) en el}
  {archivo AA,  es eliminado se actualiza el archivo recorriendo}
  {un lugar hacia arriba cada registro del archivo}

  Var
    RegAux : RegAlum; {Información de un reg. momentáneamente}
    unRegAlum : RegAlum; {Un registro del archivo AA en particular}
    reg : Integer; {Accesa cada registro por su número}
  Begin
    Seek( AA, NumAlum - 1 );
    If ( NumAlum <> FileSize( AA ) ) Then
      Begin
        reg := FilePos( AA );
        While reg <> ( FIleSize( AA ) - 1 ) Do
          Begin
            Seek( AA, reg + 1 );
            Read(AA, unRegAlum );
            RegAux := unRegAlum;
            Seek( AA, reg );
            Write( AA, RegAux );
            reg := reg + 1;
          end
      end;
    Truncate( AA )  {Se borra el último reg. del arch. de alums.}
  end;  {Actualiza_Archivo}

Procedure Elimina_N_Nombres( Var AA : ArchivoAlums );
        {Elimina del archivo de alumnos AA n nombres de alumnos}

  Var
    NúmAleli : Integer; {Número de alumno a eliminar}
    Alum : Integer; {Enumera los alumnos a eliminar}
    NAlum : Integer; {Número de alumnos a eliminar del archivo}
  Begin
    If Arch_Alums_Vacío( AA ) Then
       Escr_Archivo_Vacío
    else
      Begin
        Write('Cuántos Alumnos Quieres Eliminar ? ');
        Read(NAlum );
        If NAlum <= 0 Then
          Número_Inválido
        else
          For Alum := 1 To NAlum Do
            Begin
              ClrScr;
              Write('Cual es el Numero de Alumno a Eliminar ? ');
              Read( NumAleli );
              If Está_Número_Alumno( AA, NumAleli ) Then
                Actualiza_Archivo( AA, NumAleli )
              else
                Num_Alum_No_Esta
            end
```

```
      end
  end; {Elimina_N_Nombres}

Procedure Agrega_Calif_Parc( Var AA : ArchivoAlums );
   {Agrega a cada alumno del archivo de alumnos (AA) una }
   {calificación parcial}

  Var
    unRegAlum : RegAlum;  {Un reg. del archivo AA en particular}
          reg : Integer;  {Accesa un reg. del archivo por su núm.}
         Parc : Byte;     {Núm. de calif. parcial a agregar}
  Begin
    If Arch_Alums_Vacío( AA )  Then
       Escr_Archivo_Vacío
    else
      Begin
        Write('Cuál es el Número de Calificacion Parcial ?  ');
        Read( Parc );
        If ( Parc <= 0 ) Or ( Parc > NumCalifs ) Then
          Begin
            Writeln; Writeln;
            Write(' ' : 18, 'NUMERO DE PARCIAL FUERA DE');
            Writeln(' RANGO (1-5)');
            Delay( 5000 );
          end
        else
          For reg := 0 To ( FileSize( AA ) - 1 ) Do
            Begin
              ClrScr;
              Seek( AA, reg );
              Read( AA, unRegAlum );
              Write('Alumno: ');
              Writeln(unRegAlum.NomAlum);
              Writeln;
              Write('Calificacion: ');
              Read(unRegAlum.Califs[ Parc ] );
              unRegAlum.NumParc := unRegAlum.NumParc + 1;
              Seek( AA, reg );
              Write( AA, unRegAlum )
            end
      end
  end; {Agrega_Calif_Parc}

Procedure Modif_Calif_Parc( Var AA : ArchivoAlums );
   {Modifica  n calificaciones parciales de un número de parcial}
   {dado del archivo AA}

  Var
    unRegAlum : RegAlum;  { Un reg. del archivo AA en particular}
    Parmodif : Integer;   {Núm. de calif. parcial a modificar}
    NúmAlumMod : Integer;
           {Alumno al cual se desea modificar su calificación}
    NAlum : Integer;
 {Enumera los alums. a los cuales se les va a modif. su calif.}
    NoCalMod : Integer; {Núm. de calif. parcial a modificar}
```

```
Begin
  Write('Cuál es el Núm. de Calif. Parcial a Modificar ? ');
  Read( Parmodif );
  SeeK( AA, 0 );
  Read( AA, unRegAlum );
  If (Parmodif >= 1) And (Parmodif <= unRegAlum.NumParc) Then
    Begin
      Writeln; Writeln;
      Write('Cuántas Califs. Parciales Quieres Modificar ? ');
      Read(NoCalMod );
      If NoCalMod <=  0 Then
         Número_Inválido
      else
        Begin
          For NAlum := 1 To NoCalMod Do
            Begin
              ClrScr;
              Write('Cual es el Número de Alumno a Quien');
              Writeln(' Deseas Modificar su Calificacion ? ');
              Read(NúmAlumMod );
              If Está_Número_Alumno( AA, NumAlumMod ) Then
                Begin
                  SeeK( AA, NumAlumMod - 1);
                  Read( AA, unRegAlum );
                  ClrScr;
                  Writeln('Alumno : ', unRegAlum.NomAlum );
                  Writeln;
                  Write('Calificación: ');
                  Read(unRegAlum.Califs[ ParModif ] );
                  SeeK( AA, NumAlumMod - 1 );
                  Write( AA, unRegAlum);
                end
              else
                Num_Alum_No_Está
            end
        end
    end
  else
    If (Parmódif <= 0 ) Then
      Número_Inválido
    else
     Begin
       Writeln; Writeln;
       Write(' ' :20, 'NUMERO DE CALIFICACION NO REGISTRADA');
       Delay( 5000 )
     end
 end; {Modif_Calif_Parc}

Procedure Obten_Promedios( Var AA : ArchivoAlums );
   {Obtiene el promedio de las calificaciones parciales para }
   {cada alumno del archivo AA}

  Var
    unRegAlum : RegAlum; {Un reg. del archivo AA en particular}
    Alum : Integer;  {Enumero los alumnos del archivo}
```

```
    Calif : Integer; {Accesa cada calificación de un alumno}
    SumCalif : Real; {Suma de las califs. parciales de un alumno}
  Begin
    For Alum := 0 To ( FileSize( AA ) - 1 ) Do
      Begin
        Seek( AA, Alum );
        Read( AA, unRegAlum );
        SumCalif := 0;
        For Calif := 1 To unRegAlum.NumParc Do
          SumCalif := SumCalif + unRegAlum.Califs[ Calif ];
        unRegAlum.Prom := SumCalif / unRegAlum.NumParc;
        Seek( AA, Alum);
        Write( AA, unRegAlum )
     end
  end; {Obten_Promedios}

Procedure Está_Nombre_Alumno( Var AA : ArchivoAlums );
   {Verifica si el nombre de un alumno está o no en el}
   {archivo de alumnos}

  Var
    unRegAlum : RegAlum; {Un reg. del archivo AA en particular}
    NomAlum : NombreAl;  {Nombre del alum. a buscar en el archivo}
    Alum : Integer;      {Enumera los alumnos del archivo}
    noestá : Boolean;  {Cierto cuando el nom. del alumno no está}
  Begin
    If Arch_Alums_Vacío( AA ) Then
       Escr_Archivo_Vacío
    else
      Begin
        noestá := True; {Sup. que el nombre del alumno no está}
        Write('Nombre  del  Alumno  a  Buscar: ');
        Read( NomAlum );
        Alum := 0;
        While noestá And ( Alum < FileSize ( AA ) ) Do
          Begin
            Seek( AA, Alum );
            Read( AA, unRegAlum );
            If ( unRegAlum.NomAlum = NomAlum ) Then
               noestá := False;
            Alum := Alum + 1;
          end;
        Writeln; Writeln;
        If noestá Then
          Begin
            Write(' ': 15,'EL ALUMNO NO ESTA EN EL ARCHIVO');
            Delay( 5000 );
          end
        else
          Begin
            Writeln(' ': 15,'EL ALUMNO ESTA EN EL ARCHIVO');
            Delay( 5000 )
          end
      end
  end; {Está_Nombre_Alumno}
```

A P E N D I C E   E

Sintaxis del Lenguaje Larch Compartido.

```
Trait ::= <NomTrait> | Trait <Bloque><Excepto><Consec>
NomTrait ::= <Identif>
Identif ::= <AlfNum> | <AlfNum><Identif>
AlfNum ::= a | b | c | ... | z | A | B | C | ... | Z |
           0 | 1 | 2 | ... | 9 | -
Bloque :: = <TraitsExtern><TraitSimple>
TraitsExtern ::= <Asume><Importa><Incluye>
Asume ::= Assumes <TraitRefer> | ε
Importa ::= Imports <TraitRefer> |ε
Incluye ::= Includes <TraitRefer> |ε
TraitRefer ::= <NomTrait>, <TraitRefer> | <NomTrait> |
               <NomTrait><Renombre>,<TraitRefer> | <NomTrait><Renombre>
Renombre ::= With [ <Sustit> ]
Sustit ::= <RenomTipo><Sustit> | <Renomtipo> | <RenomFunc> |
                <RenomFunc>, <Sustit>
Renomtipo ::= <NuevTipo> for <ViejTipo>
NuevTipo ::= <IdentTipo>
IdentTipo ::= <Identif>
ViejTipo ::= <IdentTipo>
RenomFunc ::= <NuevFunc> for <ViejFunc>
NuevFunc ::= <IdentFunc>
IdentFunc ::= <Identif> | <FormaFunc>
FormaFunc ::= <LugarDom><SimbFunc><RestoFunc><LugarDom>
LugarDom ::= # |ε
SimbFunc ::= <Identif> | <SimbEspecFunc>
SimbEspecFunc ::= <SimbEspec> | <SimbEspec><SimbEspecFunc> |
                   <AlfNum><SimbEspecFunc> | <AlfNum>
SimbEspec ::= @ | $ | % | ^ | & | * | ε | ? | ~
RestoFunc ::= <LugarDom><SimbEspecFunc><RestoFunc> |
                 <LugarDom><SimbEspecFunc> | ε
ViejFunc ::= <IdentFunc>
TraitSimple ::= <Funcs><Propocs>
Funcs ::= Introduces <DeclFuncs> | ε
DeclFuncs ::= <IdentFunc> : <DomRang><DeclFuncs> |
              <IdentFunc> : <DomRang>
DomRang ::= <Dominio> => <Rango>
Dominio ::= <IdentTipo>, <Dominio> | <IdentTipo>
Rango ::= <IdentTipo>
Propocs ::= <Restricc><Propc> | ε
Restricc ::= Constrains <ObjtsRestring> so that
ObjtsRestring ::= <IdentTipo> | <ListaFuncs>
ListaFunc ::= <IdentFunc>, <ListaFunc> | <IdentFunc>
Propc ::= <Generado><Particionado><Axiomas>
Generado ::= <IdentTipo> generated <PorLista><Generado> | ε
PorLista ::= by [ <ListaFunc> ]
```

```
Particionado ::= <IdentTipo> partitioned <PorLista><Particionado> | ε
Axiomas ::= for all [ <DeclVar> ] <DeclEcuacs>
DeclVar ::= <ListaIdent> : <IdentTipo> ; <DeclVar> |
            <ListaIdent> : <IdentTipo>
ListaIdent ::= <IdentVar>, <ListaIdent> | <IdentVar>
IdentVar ::= <Identif>
DeclEcuacs ::= <Ecuacion><DeclEcuacs> | <Ecuacion> | ε
Ecuacion ::= Term | Term = Term
Term ::= <Expr1> | If <Expr1> Then <Term> Else <Term>
Expr1 ::= <Operador><Expr2><Argumentos><Operador>
Operador ::= <SimbEspFunc> | ε
Expr2 ::= <IdentFunc> ( <ListaTerm> ) | <IdentVar> | ( <Term> )
ListaTerm ::= <Term>, <ListaTerm> | <Term>
Argumentos ::= <SimbEspFunc><Expr2><Argumentos> | ε
Excepto ::= exempts <TermExept> | ε
TermExept ::= for all [ <DeclVar> ] <ListaTerm> |
              <ListaTerm> | for all [ <DeclVar> ]
Consec ::= implies <PropConsec><Convertir> | ε
PropConsec ::= <ListaTraits><Props> | ε
ListaTraits ::= <TraitRefer><ListaTraits> | <TraitRefer>
Convertir ::= Converts [ <ListaFunc> ] | ε
```

A P E N D I C E   F

Sintaxis del Lenguaje OBJ.

```
Objeto ::= <IniciaObj><NomObj><Cuerpo><FinObj>
IniciaObj ::= OBJ | OBJECT
NomObj ::= <Identif>
Identif ::= <AlfNum> | <AlfNum> <Identif>
AlfNum ::= a | b | c | ...  | z | A | B | C | ... | Z |
           0 | 1 | 2 | ... | 9 | - | _ | / | + | @ | # | $ | ~ |
           % | *
Cuerpo ::=  GET <NomArch><Cuerpo> |
            GET <NomObj> From <NomArch><Cuerpo> | <Bloque>
NomArch ::= <Identif>
Bloque ::= SORTS <TiposIntrod> / <TiposAsum><Funcs><DeclVar><Ecuacs>
TiposIntrod ::= <ListaTipos>
ListaTipos ::= <IdentTipo><ListaTipos> | <IdentTipo>
IdentTipo ::= <Identif>
TiposAsum ::= <ListaTipos>
Funcs ::= <FuncCorrect><FuncFijas><FuncError>
FuncCorrect ::= OK-OPS <DeclFuncs> | ε
DeclFuncs ::=  <IdentFunc> : <DomRang><DeclFunc> |
               <IdentFunc> : <DomRang>
IdentFunc ::= <Identif>
DomRang ::= <Dominio> -> <Rango><Atributo>
Dominio ::= <ListaTipos>
Rango ::= <IdentTipo>
Atributo ::= (ASSOCIATIVE) | (HIDDEN) | (PERMUTING) | ε
FuncFijas ::= FIX-OPS <DeclFuncs> | ε
FuncError ::= ERR-OPS <DeclFuncs> | ε
DeclVar ::= VARS <ListaVars> | ε
ListaVars ::= <IdentVars>, <ListaVars> | <IdentVars> : <IdentTipo>
IdentVars ::= <Identif>
Ecuacs ::= <EcuacsCorrect><Ecuaciones><EcuacsError>
EcuacsCorrect ::=  OK-EQNS <DeclEcuacs> | ε
DeclEcuac ::= ( <ExprDer> = <ExprIzq> ) <DeclEcuac> |
              AS <IdentTipo> ( <ExprDer> = <ExprIzq> ) <DeclEcuac> |
              ( = <ExprIzq> ) <DeclEcuac> | ( <ExprDer> = <ExprIzq> ) |
              AS <IdentTipo> ( <ExprDer> = <ExprIzq> ) |
              ( = <ExprIzq> )
ExprDer ::= <IdentFunc> ( <Argumentos> ) | <IdentFunc>
Argumentos ::= <IdentVars> | <ExprDer> | <CadNum> |
               <IdentVars>, <Argumentos> | <ExprDer>, <Argumentos> |
               <Numero>, <Argumentos>
CadNum ::= <Numero><CadNum> | <Numero>
Numero ::= 0 | 1 | 2 | ... | 9
ExprIzq ::= <Expresion> | <Identif> |
            <Expresion> If <Condicion><EcuacCond><RestoCond> |
            If <Condicion><EcuacCond><RestoCond>
```

```
Expresion ::= <ListaTipos> | <ExprDer>
Condicion ::= Not | ε
EcuacCond ::= <ExprDer><SigRelac><ExprIzq>
SigRelac ::= > | < | >= | <= | =
RestoCond ::= THEN <ExprIzq> | THEN <ExprIzq> ELSE <ExprIzq> FI | ε
Ecuaciones ::= EQNS <DeclEcuacs> | ε
EcuacsError ::= ERR-EQNS <DeclEcuac> | ε
FinObj ::= JBO | TCEJBO
```

B I B L I O G R A F I A.

[Abrial, 80] J. Abrial.
'The Specification Languaje Z: Syntax and Semantics'
Oxford University Computing Laboratory, Programming Research
Group, 1980.

[Bjorner y Jones, 78] D. Bjoner y C. G. Jones.
'The Vienna Development Method: The Meta-Language',
Springer-Verlag Lectures Notes in Computer Science,
Vol. 61, 1978.

[Burstall y Goguen, 77] R. M. Burstall y J. A. Goguen.
'Putting Theories Together to Make Specifications'
Proc. 5th International Joint Conference on Artificial
Intelligence 1977, p.p. 1045-1058.

[Burstall y Goguen, 81] R. M. Burstall y J. A. Goguen.
'An Informal Introduction Specification Using CLEAR'
Academic Press, New York, 1981, p.p. 185-213.

[Dijkstra, 62] E. W. Dijkstra.
'Guarded Commands NonDeterminacy and the Formal Derivations
of Programs', Comm. of the ACM 18 (1962), p.p. 453-457.

[Dijkstra, 72] E. W. Dijktra.
'The Humble Programmer', Comm. of the ACM 10 (1972),
p.p. 859-866.

[Ehrig y Mahr, 85] H. Ehrig y B. Mahr.
'Fundamentals of Algebraic Specification 1: Equations and
Initial Semantics', Springer-Verlag, EATCS Monographs on
Theoretical Computer Science, Vol. 6, 1985.

[Floyd, 67] R. Floyd.
'Assigning Meaning to Programs', In Mathematical Aspects of
Computer Science, XIX Americal Mathematical Society (1967),
p.p. 19-32.

[Good et al., 78] D. I. Good, R. M. Cohen, C. G. Hoch,
L. W. Hunter y D. F. Hare.
'Report on the Language Gypsy, Version 2.0',
Technical Report ICSCA-CMP-10, Certifiable Minicomputer
Project, University of Texas at Austin, 1978.

[Guttag, 75] J. V. Guttag.
'The Specification and Application to Programming of Abstract
Data Types', Computer Science Department, Canada, 1975.

[Guttag y Horning, 78] J. V. Guttag y J. J. Horning.
'The Algebraic Specification of Data Types',
Acta Informatica 10 (1978), p.p. 27-52.

[Guttag et al., 85] J. V. Guttag, J. J. Horning y J. M. Wing.
'Larch in Five Easy Pieces', Digital Equipment Corporation Systems Research Center, Report 5, 1985.

[Hoare, 69] C. A. R. Hoare.
'An Axiomatic approach to Computer Programming', Comm. of the ACM 12 (1969), p.p. 576-580, 583.

[Hoare, 72] C. A. R. Hoare.
'Proof of Correctness of Data Representations', Acta Informatica 1 (1972), p.p. 271-281.

[Hoare y Wirth, 73] C. A. R. Hoare y N. Wirth.
'An axiomatic Definition of the Programming Language Pascal', Acta Infomatica 2 (1973) p.p. 335-355.

[Hoare, 86] C. A. R. Hoare.
'Mathematics of Programming', Byte 8, (1986), p.p. 115-126.

[Igarashi et al., 75] S. Igarashi, R. L. London y D. C. Luckham.
'Automatic Program Verification: a Logical Basis and its Implementation', Acta Informatica 4 (1975), p.p. 145-182.

[Liskov y Zilles, 74] B. Liskov y S. Zilles.
'Programming with Abstract Data Types', Proc. ACM SIGPLAN Conf. on Very High Level Languages, SIGPLAN Notices 9 (1974) p.p. 50-60.

[McCarthy, 61] J. McCarthy.
'A Basic for a Mathematical Theory of Computation'. Proc. Western Joint Computer Conf.', Los Angeles, 1961, p.p. 225-238 y Proc. IFIP Congress 1962, North Holland Publ. Co., Amsterdam, 1963.

[Nakajima et al., 80] R. Nakajima, M. Honda y H. Nakahara.
'Hierarchical Program Specification and Verification -A Many-Sorted Logical Approach' Acta Informatica, Vol. 14, 1980, p.p. 135-155.

[Naur, 66] P. Naur.
'Proofs of Algorithms by General Snapshots', BIT 6 (1969), p.p. 310-316.

[Robinson y Roubine, 77] L. Robinson y O. Roubine.
'Special -A Specification and Assertion Language' Technical Report CSL-46, Stanford Research Institute, 1977.

[Scheid y Anderson, 85] J. Scheid y S. Anderson.
'The Ina Jo Specification Language Reference Manual', Technical Report TM-(L)-6021/001/01, System Development Corporation, 1985.