UNIVERSIDAD NACIONAL AUTONOMA
DE MEXICO

Unidad Academica de los Ciclos Profesional y de
Posgrado del C. C. H.

5
2ej.

UNA BIBLIOTECA DE MODULOS REUSABLES POLIMORFICOS EN ML

T E S I S

PARA OBTENER EL GRADO DE:

MAESTRA EN CIENCIAS

P R E S E N T A :

MARIA GUADALUPE ELENA IBARGUENGOITIA GONZALEZ

Dirección: Dra. Hanna Oktaba

MEXICO, D. F. JUNIO DE 1991

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

ÍNDICE

Introducción

Capítulo I 1

Especificaciones algebraicas como herramienta para la creación de bibliotecas.

1. Necesidad de software reusable.

2. Especificaciones algebraicas.

3. Construcción de una biblioteca de especificaciones algebraicas de estructuras de datos.

Capítulo II 21

Aspectos del lenguaje ML útiles para la construcción de bibliotecas.

1. Requisitos en los lenguajes de programación para implementar módulos reusables.

2. Conceptos generales de ML.

3. Construcción de una biblioteca en ML.

Capítulo III 49
Uso de las bibliotecas.

1. Metodología para el uso de las dos bibliotecas.

2. Ejemplo de aplicación.

Conclusiones 61

Apéndice A 63
Biblioteca de especificaciones algebraicas.

Apéndice B 87
Diccionario de functores y signaturas.

Apéndice C 99
Biblioteca de implementaciones en ML.

Referencias bibliográficas. 130

INTRODUCCION

Desde los orígenes de la programación se ha buscado facilitar la solución de problemas con la ayuda de una computadora, así como aumentar la confiabilidad en los resultados obtenidos. Sin embargo, a pesar de los adelantos en el desarrollo de metodologías para la elaboración de programas como fue "La Revolución Estructurada" de los años setentas y otras técnicas, no es suficiente trabajar en esa sola línea de investigación. Hace falta desarrollar herramientas cada vez más poderosas que apoyen a dichas metodologías e incrementen la productividad de los programadores al permitirles usar lo ya desarrollado.

Esto ha ocasionado dedicar también esfuerzos en la investigación de lenguajes de programación que permita al programador concentrarse en resolver un problema, así como tener la posibilidad de reusar las operaciones, tipos de datos o programas ya desarrollados.

Un concepto que ha permitido avances en estas dos ramas de la Ciencia de la Computación es el de *Abstracción*. Abstraer es concentrarse en los detalles relevantes, olvidando los irrelevantes. Las abstracciones de datos o de funciones son medios idóneos para el reuso de software. La abstracción de funciones se logra definiendo "Que" se pretende hacer, sin importar los detalles de implementación. Así se puede, por ejemplo, abstraer la función *ordenar objetos* poniendo la atención en el algoritmo de ordenación más que en el tipo de objetos a manejar o la forma en que se encuentran almacenados. Este tipo de abstracciones fue conseguida desde los orígenes de los lenguajes de programación por el uso de subrutinas, procedimientos y funciones. En la actualidad casi todos los lenguajes de programación cuentan con bibliotecas de funciones útiles que se pueden incorporar en los programas.

Desde mediados de los años setentas, se inició la investigación sobre la abstracción de datos. Liskov (Liskov 74) define: "Un tipo de datos abstracto es una clase de objetos abstractos completamente caracterizado por las operaciones disponibles para su manejo"..."Es preocuparse en como se comporta el objeto y no en como se implementa". De manera que un tópico de investigación actual es el desarrollo de mecanismos en los lenguajes de programación que permitan la abstración de datos a fin de poder tener bibliotecas de estructuras de datos mas complejas que las que ofrece el lenguaje.

Una rama más es la investigación de metodologías de programación mas formales que permitan facilitar la programación; aumentar la confiabilidad del software; apoyar el desarrollo de programas cada vez mas complejos y aumentar la productividad de los programadores. Así, hay investigación actual encaminada a desarrollar programas partiendo de especificaciones formales, las que se van refinando y verificando hasta obtener código confiable. Al mismo tiempo, incorporar bibliotecas tanto de especificaciones como de implementaciones permite facilitar el desarrollo de programas complejos y aumentar la productividad de los programadores.

En el presente trabajo se muestra la conjunción de estas tres ramas al desarrollar una biblioteca de especificaciones algebraicas de las estructuras de datos mas comunes como pilas, colas, árboles, etc., una biblioteca de implementaciones para dichos tipos en ML y una metodología para usar ambas bibliotecas en el desarrollo de programas en ese lenguaje.

En el capítulo I se presenta un panorama del por qué es necesario y útil contar con software reusable y como las especificaciones algebraicas apoyan dicho objetivo al ser una herramienta para construir una biblioteca de especificaciones de las estructuras de datos.

En el capítulo II se analizan las características que deben tener los lenguajes de programación a fin de permitir el reuso. Se selecciona al lenguaje funcional ML para construir una biblioteca de las implementaciones corespondientes y se indica el procedimiento tanto para la construcción de implementaciones en ML basadas en especificaciones algebraicas, como para el uso de la biblioteca resultante.

Con estas dos bibliotecas es posible construir programas y en el capítulo III se da el procedimiento a seguir y se muestra el desarrollo de algunas aplicaciones.

El trabajo tiene tres apéndices, en el A se encuentra la biblioteca de especificaciones algebraicas; en el B un diccionario de los módulos de que consta la biblioteca de implementaciones, la cual se encuentra en el apéndice C, para facilitar su uso.

Finalmente se dan algunas conclusiones y referencias bibliográficas.

Capítulo I.

Especificaciones algebraicas como herramientas para la creación de bibliotecas de software reusable.

1. Necesidad de bibliotecas de software reusable.

Poder volver a usar programas desarrollados con anterioridad, ha sido una necesidad de los programadores desde los inicios de la Computación.

En los años cincuentas se buscó la manera de facilitar la programación. Por ejemplo en esos años en Estados Unidos se desarrollo la investigación encaminada a liberar a los programadores del trabajo a nivel de lenguaje de máquina. Se desarrollaron los lenguajes ensambladores primero, y los primeros lenguajes de programación después. En Inglaterra se hacía investigación encaminada a construir bibliotecas con las rutinas más comunes para las aplicaciones de esos tiempos.

A partir de los años sesentas se incrementa el esfuerzo encaminado a la construcción de bibliotecas como fue el caso de la ACM que desarrolla una principalmente con rutinas de apoyo para los métodos numéricos. Por otro lado la investigación en esos años se avoca al diseño de mejores lenguajes de programación con un cierto grado de abstracción buscando que fueran útiles para todo tipo de aplicaciones y pudieran funcionar en cualquier computadora.

Las tendencias en la investigación en los años setentas fueron, por un lado el mejoramiento de los lenguajes, por ejemplo para que permitieran hacer extensiones tanto a las operaciones como a los tipos de datos que proveían (ALGOL, SIMULA); y el desarrollo de metodologías que permitieran ayudar a resolver los cada vez más complicados problemas y consiguiendo software con un nivel aceptable de calidad y confiabilidad.

Con el concepto de *tipos de datos abstractos*, a mediados de los años setenta (Liskov 74), se ha dado una nueva línea de búsqueda para el desarrollo de bibliotecas, enfocandose más en las propiedades de los datos y ampliando el conjunto de tipos de datos que ofrecen los lenguajes. Es esta línea la que se desarrolla en este trabajo, se analizan las características que debe tener el software para poder usarse en diversas circunstancias y construir una biblioteca de estructuras de datos.

Tracz, en el artículo "Where Does Reuse Start?" (Tracz 90), vierte las siguientes opiniones sobre el reuso del software: "el reuso del software es el proceso de volver a usar lo que ha sido diseñado para ese fin". Esto es importante pues establece que el reuso no es algo que se improvisa o se puede hacer con cualquier componente de software, sino que se debe pensar en que ese es el objetivo. Esto ocasiona la pregunta de cuándo se debe establecer ese propósito al definirlo, al diseñarlo, o en el nivel de código?

En dicho artículo, la tesis es que al pensar en un componente reusable "la clave es separar el contexto del concepto y del contenido". O sea, trabajar a tres niveles: en la definición conceptual del componente, en el diseño del contenido y en la implementación dentro del contexto. Trabajar al nivel del concepto es hacer la especificación funcional, lo que significa definirlo que hará el componente, pensar de la manera más abstracta posible sobre las funciones que se requieren y a través del uso de parámetros (como en las funciones matemáticas), adaptar dicha funcionalidad a todo un conjunto de diferentes problemas.

El contenido, dice el autor, es el objeto genérico que se desarrolla en el diseño, en el que se concreta la abstracción realizada en la especificación, pero sin llegar al nivel de implementación para un problema particular. Es pues, al definir el contenido que se diseña en qué consiste el componente, se piensa en el algoritmo, se diseñan las estructuras de datos, se decide lo que se pondrá en un paquete, módulo o procedimiento. Todo esto se hace a un nivel intermedio, ya que no es ni el abstracto, ni el concreto de la implementación o codificación. Sin embargo, se empieza a considerar el lenguaje de programación que se va usar busca que presente características que permitan el reuso del software.

Por último, la implementación dentro de un contexto dado consiste en hacer las instancias concretas de los parámetros para resolver un problema en particular.

De todo esto se concluye que al pensar en software reusable se trabaja a varios niveles: la especificación del componente a nivel abstracto, el diseño de un módulo en algún lenguaje de programación que soporte este tipo de mecanismos y la implementación concreta.

Una vez definida la necesidad de trabajar a varios niveles, surge otra cuestión importante Qué técnicas existen actualmente para la reusabilidad? Biggerstaff (Biggerstaff 84) clasifica en dos enfoques: la generación y la composición.

La técnica de generación se basa en componentes activos en el sentido de que se define un patrón de código y se tiene un generador que lo transforma a código fuente mediante una serie de reglas. Ejemplos del uso de esta técnica son los llamados lenguajes de Cuarta Generación.

La composición parte de la idea de tener bloques que se *arman* según reglas bien definidas. Los bloques o componentes son, por ejemplo, procedimientos, funciones, esqueletos de código u objetos (en el sentido de los lenguajes orientados a objetos), que son atómicos o autocontenidos y con una estructura interna relativamente inmutable, pero que permiten la conexión de unos con otros a través de parámetros o de cláusulas de tipo *exporta* e *importa*.

Dentro de la línea de la composición, se puede considerar la idea de construir una serie de módulos o componentes para formar una biblioteca . Goguen (Goguen 86) opina que una biblioteca debe tener: componentes útiles, información de como usarlos en el diseño de sistemas, y código ya compilado. También apoya la idea de construir bibliotecas de tipos de datos abstractos, que es precisamente la idea de este trabajo.

Al construir una biblioteca de tipos de datos abstractos, se parte de un conjunto de especificaciones algebraicas que corresponden al establecimiento del concepto de Tracz. Por medio de las especificaciones algebraicas se obtiene una definición precisa, clara y no ambigua. Además por medio de ellas se puede desarrollar software productivo o pueden ser herramientas para el diseño del software por medio de prototipos que permiten validar lo que se construirá. Por esto se revisan a continuación algunos conceptos necesarios al construir una biblioteca de especificaciones.

2. Especificaciones algebraicas.

La teoría de las especificaciones algebraicas surge a partir de los trabajos de (Zilles 74), (Guttag 7)5 y (Goguen 78), quienes son los primeros en definir las especificaciones algebraicas con un enfoque matemático riguroso.

A continuación se da una definición formal de lo que es una especificación algebraica, (Ehrig 85).

Una especificación algebraica ESPEC = ((S, Op), E) consiste de:

-una signatura SIG = (S, Op), que es conjunto de géneros (nombres de portadores) y otros tipos S, y un conjunto de nombres de operaciones con su dominio y rango y de símbolos constantes Op definidos en los géneros.

- un conjunto de axiomas o propiedades E, definidas por medio de ecuaciones o predicados lógicos, que determinan el comportamiento de las operaciones Op.

Un álgebra (en sentido matemático: conjuntos y operaciones sobre ellos (Lipson 81)),de una especificación ESPEC, es aquella que corresponde a la signatura SIG y que satisface todas las ecuaciones E. En general hay muchas álgebras que satisfacen una especificación. Al álgebra que satisface una cierta signatura se le llama SIG-álgebra (o Σ-álgebra).

Entre las ventajas en el uso de especificaciones algebraicas como base para la construcción de módulos reusables están: se destacan las propiedades matemáticas, permiten obtener descripciones más simples y generales, lo que a su vez facilita la comprobación de que son correctas. Además como se verá las especificaciones algebraicas son un medio muy poderoso de diseñar software modular y reusable.

Goguen en el artículo "Reusing and Interconnecting Software Components" (Goguen 86), define varios conceptos importantes sobre la especificación de programas y provee mecanismos que permiten que se organicen las especificaciones en estructuras a través de herencias y otras relaciones.

En el mismo artículo, Goguen hace una cuidadosa distinción entre lo que es la composición horizontal y la vertical. Con estos conceptos, se tiene una noción poderosa del reuso de software a través de la composición.

La composición horizontal se refiere a imponer una estructura a un cierto nivel de abstracción; y la vertical a la forma de componer a diferentes niveles de abstracción.

La composición vertical permite obtener una biblioteca desde especificaciones hasta implementaciones, o sea en varios niveles de abstracción.

Dentro de las actividades que se pueden efectuar en la composición horizontal, al nivel de definición, con las especificaciones algebraicas están: enriquecer, derivar, y combinar (Sannella 86). La primera consiste en poder aumentar los géneros, las operaciones o los axiomas de una especificación existente. La segunda permite que algunas operaciones o géneros se oculten o renombren para obtener otra especificación. Combinarlas permite armar una nueva especificación a partir de una o varias. Para obtener una biblioteca de módulos reusables es importante trabajar tanto en forma horizontal como vertical.

Según (Van Horebeek 89) y (Bergstra 89), estas operaciones definidas por Sannella se pueden reducir a tener en una especificación la posibilidad de importar y exportar signaturas y operaciones, así como usar la parametrización. La exportación permite que ciertos tipos y operaciones definidas como exportables, puedan accesarse fuera de la especificación que las define y heredarse.

La importación es una operación de composición entre especificaciones. Con ella se obtiene la unión de las signaturas y operaciones.

La parametrización consiste en tener uno o más parámetros en la especificación, a los que se llaman parámetros formales, formales a los cuales se les asignará un valor concreto en el momento de desarrollar la instancia (a estos parámetros se les denomina actuales o reales). Este mecanismo es muy poderoso ya que permite generalizar una especificación y posponer decisiones de diseño al hacer una instancia concreta, permitiendo un diseño de lo general a lo particular. Por ejemplo, se puede tener una especificación de listas parametrizada por el tipo de los elementos en ella.

Por otra parte, por medio de la importación y la parametrización es posible establecer una jerarquía de especificaciones, partiendo de unas muy primitivas sobre las que se construyen otras. Cada nivel jerárquico debe respetar ciertas restricciones que se pueden verificar para facilitar la producción de software.

A continuación se define el formalismo que se usa para describir la biblioteca de especificaciones está basado en los trabajos de los autores citados anteriormente.

(Se usa una notación basada en BNF en donde, lo que aparece entre <>, es sustituible por lo indicado en esa categoría gramatical).

Una especificación consta de :

<especificación> =

```
     ESPEC  <nombre> ;
      < cláusula de parámetros >
      < cláusula de importación >
      < cláusula de exportación >
      < definición de géneros >
      < definición de operaciones >
      < axiomas >
FIN DE  < nombre > ;
```

La cláusula de parámetro define los requerimientos de los parámetros formales a través de la definición de géneros y operaciones que deben satisfacer tanto sintáctica como semánticamente los parámetros actuales.

Para la instanciación se establecen dos restricciones: que se preserve el rango de las operaciones y que se cumplan los axiomas.

En la cláusula de importación se puede indicar si de otras especificaciones se importa todo (TODO), todo salvo alguna parte (EXCEPTO) o se lista expresamente qué se importa.

La exportación lista los géneros y las operaciones que pueden usarse fuera del módulo. Un género puede ser derivación de otro ya especificado o ser uno nuevo. Las operaciones pueden ser definidas en la especificación, en cuyo caso hay que dar los axiomas posteriormente, o ser renombradas, lo que indica que el comportamiento es el mismo, pero se cambia de nombre para mayor claridad. Lo que no se exporta se le llama *oculto* y se impide su uso fuera de la especificación.

El usar cláusulas de importación y exportación da una protección extra al diseñador y a los usuarios de los módulos, pues permite saber que se incluye de otra especificación y qué se puede accesar desde fuera, lo que ocasiona especificaciones de mayor calidad y efectividad.

La definición de géneros y operaciones corresponde a la signatura S. Losaxiomas forman el conjunto E de propiedades.

A continuación se da un ejemplo de especificacion algebraica, el caso de las estructuras lineales. Son aquellas secuencias lineales de elementos en que se introducen y eliminan por algún extremo de la secuencia.

```
ESPEC  Estructuras Lineales;
PARAMETRO  Elementos
     Género  Elemento
FIN DE  Elementos;

IMPORTA TODO DE Booleanos;

EXPORTA Estrlineal, estrvacia, inserta, inicial, resto, esvacia;

     GENERO Estrlineal

     OPERACIONES
          estrvacia : -> Estrlineal
          inserta   : Elemento, Estrlineal -> Estrlineal
          inicial   : Estrlineal -> Elemento
          resto     : Estrlineal -> Estrlineal
          esvacia   : Estrlineal -> Bool

AXIOMAS
     VAR    e: Elemento       l,m : Estrlineal
E1:  inicial ( estrvacia ) = error
E2:  inicial ( inserta ( e,l )) = e
E3:  resto ( estrvacia ) = error
E4:  resto ( inserta (e,l )) = l
E5:  esvacia (estrvacia ) = cierto
E6:  esvacia ( inserta (e,l )) = falso

FIN DE Estructuras Lineales;
```

En esta especificación una estructura lineal está parametrizada por *Elemento*, lo que indica que se puede poner en ella cualquier tipo de datos que cumpla con la especificación del tipo *Elemento*. En la clausula de importación se indica que se dispone de todo lo que aparece en la especificación llamada *Booleanos*. Después se exporta el género *Estrlineal* y las operaciones definidas a continuación; no hay géneros ni operaciones ocultas, pues todo lo definido aparece dentro de la clausula de exportación.

Enseguida se declaran las variables auxiliares para la formulación de los axiomas, los que aparecen al final, numerados y en forma de ecuaciones.

Dentro de las operaciones se distinguen las llamadas constructoras, aquellas cuyo codominio es portador del género que se está definiendo y que permiten obtener nuevas estructuras lineales, como es el caso de *estrvacia* e *inserta*. A la operación *resto* se le llama auxiliar, pues aunque tiene codominio en el género estrlineal no construye una nueva sino que sólo la modifica. Las operaciones *inicial* y *esvacia* son observadoras, dan información sobre el tipo.

Para obtener una instacia de estructuras lineales, por ejemplo de naturales, se define el parámetro actual indicando cada uno de los requerimiento como se instancía. Por ejemplo, el parámetro de las estructuras lineales es *Elementos*, el cual es instanciado por la especificación de *Naturales*, y el género *Elemento*, con *Nat*:

```
INSTANCIA EstrNaturales;

    CON Elementos COMO Naturales
        Elemento  COMO Nat

FIN DE ENSTANCIA ;
```

De esta forma se pueden obtener diferentes instancias para cualquier especificación, con sólo comprobar que satisfagan los requerimientos del parámetro.

Se puede obtener una biblioteca de especificaciones algebraicas con los tipos de datos más usuales, como los que se estudian en un curso de Estructura de Datos: varios tipos de pilas, varias colas, árboles, etc., en la que se van componiendo diversas especificaciones. Otra ventaja es que se puede demostrar que todas las especificaciones son consistentes y tan completas como sea necesario.

3. Construcción de una biblioteca de especificaciones algebraicas de estructuras de datos.

A partir del formalismo y de los conceptos presentados en la sección anterior, se construye una biblioteca de especificaciones algebraicas para las estructuras de datos más comunes.

Primero se hacen las especificaciones de los parámetros necesarios para el resto de las especificaciones. Se inicia con las especificaciones llamadas *Elementos*, *Elemcompar* y *Elemorden*, que definen el contenido mínimo que se requiere como parámetro en las demás especificaciones. En *Elementos* sólo se necesita un género al que se llama *Elemento*. En *Elemcompar* además del género se requiere una operación de igualdad entre los elementos del género; en *Elemorden* además hace falta una operación que permita establecer un orden.

Enseguida se tiene la especificación de algunos tipos básicos como los booleanos, los naturales, caracteres (aunque en realidad no son necesarios debido a que son parte de los tipos predefinidos en muchos lenguajes de programación), a fin de tener en dicha biblioteca todos los tipos a que se hace referencia.

Se van construyendo en forma ascendente tipos más complejos. Además cualquier especificación puede ser el parámetro actual siempre que cumpla al menos con los requerimientos del parámetro formal. Para maximizar los beneficios al construir especificaciones jerárquicas, se debe asegurar que cada nuevo nivel preserve la estructura algebraica definida en el nivel anterior, por lo que se establecen dos restricciones (Bergstra 89 p 75):

Restricción de no Confusión: " Dos objetos definidos en una jerarquía no deben igualarse al agregar una nueva especificación".

Esta restricción significa que dos elementos que son diferentes antes de agregar una nueva especificación, no deben resultar iguales como consecuencia de las ecuaciones agregadas.

Restricción de no Extensión: " Un conjunto de objetos definidos en una jerarquía, no se debe extender con nuevos objetos después de agregar una nueva especificación".

Lo que se establece, es que si un cierto conjunto de objetos de un tipo t se tiene en una jerarquía, al agregar otra especificación, no se generen más objetos que sean también del tipo t por razonamientos con las ecuaciones agregadas.

Un ejemplo de cómo asegurar estas restricciones se presenta en el caso de las *Estructuras Lineales* de la sección anterior, en que la jerarquía anterior esta formada por los *Booleanos* y se extiende con esta nueva especificación. La primera restricción se cumple al observar que en los axiomas C5 y C6 *cierto* y *falso* son distintos, ya que *estrvacia* e *inserta (e,c)* por su definición nunca van a ser iguales. Por lo que aunque se agregue la especificación de estructuras lineales a la jerarquía se preserva la desigualdad de *cierto* y *falso* de *Bool*.

De la misma manera, para comprobar que se cumple la segunda restricción, hay que ver que se podrían agregar nuevos objetos al tipo *Bool* por la operación *esvacia* de las *Estrucutras Lineales*, pero de nuevo por los axiomas C5 y C6 y la propiedad antes mencionada, se asegura que el resultado de *esvacia* se reduce siempre a *cierto* o *falso*, por lo que no se introducen nuevos objetos al tipo *Bool*.

Se puede comprobar que la biblioteca que se presenta cumple con ambas restricciones.

A continuación se muestra la estructura jerárquica de la biblioteca usando una gráfica en la que si una especificación es importada por otra, se establece una flecha de la primera a la segunda. De forma que las raíces de los árboles son las especificaciones más generales. La biblioteca completa se puede consultar en el apéndice A.

Como se parte de las especificaciones básicas que pueden ser importadas por todas, están fuera de la jerarquía por simplicidad. Mas bien se muestra la jerarquización entre variantes de un mismo tipo de datos abstractos, pilas, colas, árboles, etc.

En la figura 3.1 se muestra la estructura y componentes de la biblioteca.

A continuación se analiza la estructura y las especificaciones de la biblioteca.

3.0 Especificaciones para los parámetros.

La especificación *Elementos* exporta sólo un género y se usa en aquellas especificaciones que requieren al menos de un género como parámetro. Como de hecho todas las especificaciones satisfacen sobradamente este requerimiento, es posible que lo instancien y sirvan de parámetro actual en cualquier momento.

Por otro lado, *Elemcompar* necesita además una operación de igualdad que cumpla con las condiciones clásicas de ser una relación de equivalencia definida en los elementos. *Elemorden* requiere de un orden lineal indicado en la operación "<" y que es cualquiera que cumpla con las propiedades de ser irrefexiva y transitiva.

3.1 Elementos básicos.

En las especificaciones de los *Booleanos*, *Naturales* y *Caracteres* se indica que se exporta el género y las operaciones más comunes, aunque no son especificaciones completas pues se deben incluir todo lo que se requiere de ellas. Pero se prefirió omitirlo para conseguir claridad en las especificaciones (Para verlas completas se puede revisar la bibliografía). Si se tuviera un probador automático de teoremas para comprobar las especificaciones sería necesario incluirlas completas.

3.2 Estructuras lineales.

Dentro de las estructuras lineales, siguiendo con la idea de construcción jerárquica, primero se presenta la especificación de *Estructuras Lineales*, en general, en ella se exporta el género *Estrlineal* y las operaciones básicas a toda estructura lineal que son : *estrvacia*, *inserta*, *inicial*, *resto* y *esvacia*. Son constructoras *estrvacia* e *inserta*, la primera permite iniciar como vacia una estructura y la segunda inserta un elemento. La operación *inicial* es una observadora del elemento recién insertado; *resto* permite observar todos los elementos anteriores (menos el inicial) y *esvacia* regresa un booleano indicando si la estructura lineal esta vacia. En los axiomas se define el comportamiento de cada una de ellas. Requiere como parámetro *Elementos* con el género *Elemento*.

```
ESPEC  Estructuras Lineales;

PARAMETRO  Elementos
     Género  Elemento

FIN DE  Elementos;

IMPORTA TODO DE Booleanos;

EXPORTA  Estrlineal, estrvacia, inserta, inicial, resto, esvacia;
     GENERO Estrlineal
     OPERACIONES
          estrvacia : -> Estrlineal
          inserta   : Elemento, Estrlineal -> Estrlineal
          inicial   : Estrlineal -> Elemento
          resto     : Estrlineal -> Estrlineal
          esvacia   : Estrlineal -> Bool

AXIOMAS
     VAR     e: Elemento        l,m : Estrlineal
E1:  inicial ( estrvacia ) = error
E2:  inicial ( inserta ( e,l )) = e
E3:  resto ( estrvacia ) = error
E4:  resto ( inserta (e,l )) = l
E5:  esvacia (estrvacia ) = cierto
E6:  esvacia ( inserta (e,l )) = falso

FIN DE Estructuras Lineales;
```

Para obtener *Listas* Se importa todo de las *Estructuras Lineales*,se renombra el género *Estrlineal* como *Lista* y a la constante *estrvacia* como *listavacia*. Exporta todo lo heredado, más las operaciones que se introducen que son *enesimo*, *longitud* y *concatena* que se consideran importantes para el manejo de listas y cuyo comportamiento se define en los axiomas como se ve a continuación.

```
ESPEC  Listas;
IMPORTA TODO DE Estructuras Lineales, Booleanos, Naturales;

     Renombra Estrlineal como Lista
              estrvacia  como listavacia

EXPORTA  Lista, listavacia, inserta, inicial, resto, longitud,
enésimo, esvacía;

     OPERACIONES
          enésimo        : Nat, Lista -> Elemento
          longitud       : Lista -> Nat
          concatena      : Lista, Lista -> Lista
```

```
AXIOMAS
     VAR    e: Elemento       l, l1 : Lista      n:Nat
L1:  enésimo (n, listavacia) = error
L2:  enésimo (n, inserta (e,l)) = Si n=0 entonces error
                                  si_no Si n=1 entonces inicial(l)
                                        si_no enésimo(n-1,resto(l))
L3:  longitud ( listavacia )) = 0
L4:  longitud ( inserta (e,l )) = 1 + longitud (l)
L5:  concatena (listavacia, l) = l
L6:  concatena (inserta (e,l),l1) = inserta (e, concatena (l,l1))

FIN DE   Listas;
```

Para *Pilas*, se parte de las estructuras lineales y se renombra el género *Estrlineal* como *Pila*, renombrando las operaciones para tener la nomenclatura típica de las pilas. Se puede observar que no hay axiomas en esta especificación pues el comportamiento de las operaciones definidas para las estructuras lineales es exactamente el mismo que el de las pilas.

En *Pila acotada*, se requiere de un parámetro que indique la cota. Importa todas las operaciones de las *Pilas*, exporta el género, todas las operaciones que importa y agrega *estallena* y *tamaño* además de hacer una redefinición de mete para considerar el acotamiento.

Para las Colas, se parte de una especificación llamada *Estr Colas* que define, a partir de las *Estructuras Lineales*,el género *Cola* importa todas las operaciones que permiten introducir elementos por un extremo de la estructura y sacarlos por ese mismo extremo, y agrega las operciones necesarias para hacerlo por el otro extremo. Se renombran las operaciones para indicar por que extremo se trabaja de forma que *inserta* se llama *meteult* y *resto* se vuelve *sacaprim*. De esto se obtiene inmediatamente la especificación de las Colas Dobles que son la clase de colas que incluyen todas esas operaciones, sin dar nuevos axiomas, como se aprecia a continuación.

```
ESPEC  Cola doble;

IMPORTA TODO DE Estr colas;

EXPORTA     Cola,  colavacia,  meteult,  meteprim,  sacault,
sacaprim, primero, último, esvacia;

FIN DE  Cola-doble;
```

Sin embargo, para tener *Colas*, con inserción por un extremo y salida por el otro, hay que ocultar algunas operaciones y exportar *meteult, sacaprim, primero, colavacia y esvacia*. Para *Cola acotada* se procede igual que en *Pila acotada*. En *Cola prior*,(cola con prioridad) importa *Colas*, exportando todo más una operación que busca el *máximo* y se redefinen en los axiomas algunas operaciones para tratar la prioridad, por lo que requiere como parámetro además del elemento una operación de orden, por lo que usa como parámetro la especificación de *Elemorden*.

3.3 Arboles.

El tipo de datos abstracto árbol binario se define como un conjunto de nodos tal que :

i) es vacio es un árbol binario, o

ii) consiste en un elemento de datos llamado raiz y dos subárboles binarios ajenos, llamados subárbol izquierdo y derecho, así como operaciones de acceso. (Harrison 89)

Por eso se parte de una especificación básica para árboles binarios parametrizada por *Elemorden*,(este tipo de datos no requiere más que a *Elemento* como parámetro, pero los que lo importan si necesitan un orden,lo que hace preferible que sobre aquí), importa a los *Booleanos* y *Naturales* y exporta el género *Arbolbin* y las operaciones que se consideraron más generales sobre los árboles, como son: *árbolvacio, creaárbol* las constructoras, *izq, der, raiz, altura*, y *esvacio*, que se definen en los axiomas.

```
ESPEC  Arbolesbinarios;

PARAMETRO    Elemorden
     Género Elemento
     operación
          igual
FIN DE  Elemorden;

IMPORTA TODO DE Booleanos, Naturales;

EXPORTA Arbolbin, árbolvacio, creaárbol, izq, der, raiz, altura,
esvacio;

     GENERO Arbolbin
```

```
OPERACIONES
        árbolvacío     : -> Arbolbin
        creaárbol     :Arbolbin,Elemento,Arbolbin -> Arbolbin
        izq             : Arbolbin -> Arbolbin
        der             : Arbolbin -> Arbolbin
        raiz            : Arbolbin -> Elemento
        altura          : Arbolbin -> Nat
        esvacio        : Arbolbin -> Bool

AXIOMAS
     VAR     e: Elemento          ai,ad : Arbolbin
AB1: izq (árbolvacio ) = error
AB2: izq (creaárbol (ai,e,ad )) = ai
AB3: der ( árbolvacio ) = error
AB4: der (creaárbol (ai,e,ad )) = ad
AB5: raiz (árbolvacio ) = error
AB6: raiz (creaárbol (ai,e,ad )) = e
AB7: altura (árbolvacio ) = 0
AB8: altura (creaárbol(ai,e,ad)) =1 + max(altura(ai),altura(ad))
AB9:  esvacio (árbolvacio ) = cierto
AB10:esvacio ( creaárbol (ai,e,ad )) = falso

FIN DE  Arbolesbinarios;
```

Después se construye una especificación Extiende árboles que importa todo lo anterior y exporta otras operaciones útiles al manejar árboles. En particular *eshoja, numhojas, numnodos* y *estalleno* que ayudan a construir árboles piramidales.

Un árbol piramidal (Heap) se define como un árbol binario completo con un orden parcial entre sus nodos, de forma que el valor del nodo raiz es menor o igual que los valores de los nodos de sus subárboles y ambos son a su vez piramidales.(Harrison 89) Por eso en la especificación de este tipo de árboles se requiere una operación de orden entre los elementos y una operación para saber si el árbol está lleno, ya que al insertar elementos se busca mantener las condiciones piramidales. La especificación de árboles piramidales importa *Extiencde árboles*, ocultando la constructora *creaarbol* y en su lugar exporta *inserta, max, sacamax* y *esvacio.*

Se define a los árboles balanceados por la altura según (Adel'son-Vel'skii y Landis 62).Un árbol se dice AVL o balanceado por la altura si

i) es vacio, o

ii) sus subárboles izquierdo y derecho son ambos AVL

iii) la diferencia de las alturas de sus subárboles es menor o igual a uno. (Harrison 89).

En la especificación *Arboles AVL* se importan las operaciones definidas en *Arboles binarios* pero ocultando la constructora *creaarbol*, ya que sólo inserta elementos en un árbol pero no tiene cuidado de mantener la condición necesaria para el balanceo, y en su lugar exporta las operaciones *inserta* y *sustrae* que añaden y quitan elementos del árbol sin perder la condición de balanceo por altura. Además exporta *estabalanceado*.

Un árbol de búsqueda se define como un árbol binario con un orden impuesto sobre los nodos de forma que:

i) Los valores del subárbol izquierdo son menores que el valor del nodo raíz, o

ii) los valores del subárbol derecho son mayores que el del nodo.

En la especificación *Arboles búsqueda* se importa todo de *Arboles binarios* y exporta con las operaciones de *inserta*, *sustrae*, *miembro* y *minimo* definidos en los axiomas y permiten cumplir con las restricciones de un árbol de búsqueda.

3.4 Conjuntos.

El tipo de datos abstracto conjunto lo define Harrison como una colección arbitraria de elementos distintos, con operaciones de acceso.

Así para la especificación de *Conjuntos* se tiene el parámetro usual, se importan también los *Booleanos* y *Naturales*. Se exporta el género *Conjunto* y las operaciones *conjvacio* e *inserta* que son las constructoras y tienen la característica de no permitir elementos repetidos según la definición. Las operaciones de acceso*remover*, *union*, *interseccion*, *diferencia*, *pertenece*, *tamaño* y *esvacio* se definen en los axiomas.

ESPEC Conjuntos;

PARAMETRO Elementos;
Género Elemento

FIN DE Elementos;

IMPORTA TODO DE Booleanos, Naturales;

EXPORTA Conjunto, conjvacio, inserta, remover, unión, intersección, diferencia, pertenece, tamaño, esvacio;

```
GENERO Conjunto

OPERACIONES
     conjvacio      : -> Conjunto
     inserta    : Elemento, Conjunto -> Conjunto
     remover    : Elemento, Conjunto -> Conjunto
     unión      : Conjunto, Conjunto -> Conjunto
     intersección    : Conjunto, Conjunto -> Conjunto
     diferencia      : Conjunto, Conjunto -> Conjunto
     pertenece       : Elemento, Conjunto -> Bool
     tamaño     : Conjunto -> Nat
     esvacio    : Conjunto -> Bool

AXIOMAS
     VAR    e,f: Elemento      c,d: Conjunto

C1:  pertenece (e, conj vacio) = falso
C2:  pertenece (e, inserta(f,c)) = Si e=f entonces  cierto
                                   si_no  pertenece (e,c)
C3:  es vacio ( conj vacio ) = cierto
C4:  es vacio ( inserta(e,c)) = falso
C5:  tamaño ( conj vacio) = 0
C6:  tamaño ( inserta (e,c)) = 1 + tamaño (remover(e,c))
C7:  remover (e, conj vacio ) = conj vacio
C8:  remover (e,inserta (f,c)) = Si e=f entonces  remover (e,c)
                                 si_no inserta ( f,remover (e,c) )
C9:  unión (c, conj vacio) = c
C10: unión (c, inserta (e,d)) = inserta (e, unión (c,d))
C11: intersección (c,conj vacio) = conj vacio
C12: intersección (c,inserta(e,d)) = Si pertenece (e,c) entonces
                                       inserta (e,intersección(c,d))
                                     si_no intersección (c,d)
C13: diferencia (c, conj vacio ) = c
C14: diferencia (c,inserta(e,d)) = Si pertenece( e,c) entonces
                                     remover (e, diferencia(c,d))

FIN DE  Conjuntos;
```

El tipo de datos abstracto Bolsa es una colección arbitraria de elementos que no necesariamente son distintos.

La especificación *Bolsas* importa la de *Conjuntos*, exporta el nuevo género *Bolsa* y modifica las operaciones *remover*, *diferencia* y *tamaño* para permitir repeticiones.

3.5 Gráficas.

El tipo abstracto Gráfica consiste en un conjunto finito de nodos, un conjunto de aristas, donde una arista es la conexión entre dos nodos, y operaciones de acceso.

En la especificación de *Gráficas* el parámetro es *Nodos*. Importa a los *Booleanos* y exporta el género *Gráfica*. Las constructoras de *Gráficas* son *graficavacia*, *agreganodo* y *agregaarista*. Además se tienen las operaciones *borranodo*, *borraarista*, *contiene*, *esadyacente* y *esvacia*, que se definen en los axiomas.

```
ESPEC  Gráficas;

PARAMETRO  Nodos
     Género Nodo
     operación
          igual      :Nodo, Nodo ->Bool

FIN DE  Nodos;

IMPORTA TODO DE Booleanos;

EXPORTA Gráfica, gráficavacia, agrega nodo, agrega arista,
borra nodo, borra arista, contiene, es adyacente, esvacia;

     GENERO Gráfica
     OPERACIONES
          gráficavacia  : -> Gráfica
          agrega nodo     : Nodo, Gráfica -> Gráfica
          agrega arista   : Nodo, Nodo, Gráfica -> Gráfica
          borra nodo      : Nodo, Gráfica -> Gráfica
          borraarista     : Nodo, Nodo, Gráfica -> Gráfica
          contiene        : Nodo, Gráfica -> Bool
          es adyacente    : Nodo, Gráfica -> Bool
          esvacia         : Gráfica -> Bool

AXIOMAS
     VAR    n,m,p,q : Nodo          g : Gráfica
G1:  borra nodo (n, gráficavacia ) = gráficavacia
G2:  borra nodo (m, agraga nodo(n,g)) = Si n= m entonces
                                          borra nodo (n,g)
                                   si_no agraga nodo (m,borra nodo(n,g))
G3:  borra nodo (n, agrega arista (p, q, g)) = Si n=p or n=q
                  entonces borra nodo(n,g)
                  si_no agrega arista(p,q,borra nodo(n,g))
G4:  borra arista (n,m, gráficavacia ) = gráficavacia
G5:  borra arista (n,m, agrega nodo(p,g)) =
                  agrega nodo (p, borra arista (n,m,g))
```

```
G6:  borra arista (n,m, agrega arista (p,q,g)) =
                Si ( n=p y m=q ) or ( n=q y m=p ) entonces
                borra arista (n,m, g)
                si_no agrega arista (p,q,borra arista (n,m,g,))
G7:  contiene (gráficavacia ) = falso
G8:  contiene (n, agrega nodo (m,g)) = Si n = m entonces cierto
                                       si_no contiene (n,g)
G9:  contiene (n, agrega arista (p, q,g)) = Si n=p or n=q
                                       entonces cierto
                                       si_no contiene (n,g)

G10: es adyacente (n,m, gráficavacia) = falso
G11: es adyacente (n,m, agrega nodo (p,g)) = es adyacente (n,m, g)
G12: es adyacente (n,m, agrega arista(p,g)) =
                              Si (n=p y m=q) or (n=q y m=p)
                              entonces cierto
                              si_no es adyacente (n,m, g)
G13: esvacia (gráficavacia ) = cierto
G14: esvacia (agrega nodo (n,g)) = falso
G15: esvacia (agrega arista (n,m, g) = falso

FIN DE Gráficas;
```

Una digráfica o gráfica dirigida es aquella en que las aristas tienen dirección, de modo que un nodo es el origen y otro el destino.

En *Digráficas* se importa todo de *Gráficas*, se exporta el género *Digráfica* y las operaciones *essucesor* y *espredecesor* que sustituyen a *esadyacente*, pues en este caso interesa la orientación de las aristas.

6. Tablas de dispersión o *hash*.

Estas tablas, aunque son también estructuras lineales, se tratan por separado. Una tabla abstracta es una estructura lineal de parejas de elementos (*llave*, *dato*) en la que los datos son homogéneos y la *llave* identifica en forma única a cada elemento.

Es por eso que al construir una tabla, primero se especifican sus elementos, dando lugar al *Diccionario*, que es la estructura lineal de parejas llave y dato. Así pues, en la especificación de *Diccionario* se requiere como parámetro a *Datos*, que provee la información sobre los tipos *llave* y *dato*, así como una operación *igual* de comparación entre llaves.

No se importa nada, exporta el género *Dicc* y las operaciones *DicVacio*, *inserta*, *elimina*, *recupera*, *pertenece* y *esVacio*, cuyas signaturas y comportamiento se pueden consultar en el apéndice A.

Tabla requiere de dos parámetros: *Elem*, que indica el género de los elementos a almacenar y *Rango*, que da el tamaño de la tabla. Exporta al género *Tabla* y las operaciones *TabVacia*, *asigna* e *inf* definidas en la especificación.

Con estas dos especificaciones se puede construir la de tabla de dispersión o *hash*, la cual tiene por objetivo hacer en el menor tiempo posible, del orden O(1) constante, la búsqueda de elementos sin necesidad de recorrerla. Para eso se auxilia de una función *funhash* que dada una llave encuentra el rango donde colocarla. (Ver figura 3.2)

La especificación *Tablahash* tiene como parámetro a dicha función e importa todo de *Tablas*, con *Elem* como *Diccionario* de *Funhash* y renombra el género *Tabla* como *Tabhash*. Las operaciones que exporta son *incluye*, *excluye*, *inf*, *existe* y *asignado*.

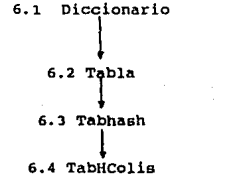


Figura 3.2 Tabla *Hash*.

Finalmente como la función *hash* puede dar colisiones, esto es que dadas dos llaves k1 y k2 diferentes *hash(k1) = hash(k2)*, se requiere de alguna función *sig*, como parámetro, que resuelva que hacer en esas situaciones, ejemplos de las cuales se pueden consultar en los libros de estructuras de datos.

Esta especificación requiere como parámetro los géneros *llave*, *dato* y *rango* así como las funciones *hash* y *sig*. Importa todo de *Tabhash* con *Elem* como la pareja (*llave*, *dato*) y exporta el género *Tabhash* y las operaciones *incluye* e *inf* modificadas para manejar situaciones de colisión.

CAPITULO II.

Aspectos del lenguaje ML útiles para la construcción de bibliotecas.

1. Requisitos en los lenguajes para implementar módulos reusables.

En este capítulo se hace un estudio de las características que debe tener un lenguaje de programación para permitir la creación de software reusable.

Hay dos características fundamentales para permitir la reusabilidad: la *modularidad* y la *parametrización*. Los lenguajes de programación en que se puedan encontrar estas facilidades son idóneos para construir la biblioteca de las implementaciones correspondientes a la anterior biblioteca de especificaciones algebraicas. Por este motivo se investigan estos conceptos a fin de justificar la selección de un lenguaje de programación.

Modularidad:

En los primeros lenguajes de programación surge la idea de crear procedimientos, subrutinas y macros a fin de descomponer el código en piezas más simples, evitar repeticiones de textos y extender los operadores del lenguaje. En los lenguajes actuales se tiene la idea de módulos que es como una caja negra que interactua con otros módulos o programas a través de una interfaz. Con la idea de modularidad surge una serie de nuevos conceptos en los lenguajes de programación que se revisan a continuación.

Uno de los objetivos del uso de módulos es regular la visibilidad, esto es, tener mecanismos que permitan distintos niveles de visión de un módulo, por ejemplo a un usuario sólo le interesa saber que contiene un módulo y no como está implementado. De esto se parte para que un sistema de módulos deba tener mecanismos para separar la interfaz de la implementación. La interfaz es la parte visible de un módulo en la que se definen los tipos de datos y las operaciones que se exportan. La implementación contiene la definición del comportamiento concreto de las entidades.

Otra característica importante es tener mecanismos de comunicación entre módulos. Esto es poder definir qué se exporta y se importa entre módulos. Un módulo puede importar a otro, lo que significa que puede hacer uso de todas las entidades exportables definidas en la interfaz del segundo módulo. Otro mecanismo de comunicación es la herencia, esta se define como la posibilidad de que un módulo pueda pasar no sólo las entidades que exporta y también aquellas que importó. Así un sistema de módulos con estas características permite la construcción de programas en forma ascendente y jerárquica.

En la actualidad varios lenguajes de programación permiten hacer la distinción entre la interfaz y la implementación. En ADA a los módulos se les llama paquetes (package), la interfaz es la especificación del paquete y se tiene el cuerpo del paquete. En Modula-2 se llaman módulos y constan de la definición y la implementación. ML tiene a las signaturas (signature) para la interfaz y estructuras (structure) para la implementación.

Pero en general el uso de estos mecanismos está restringido por el manejo de tipos. De aquí que los conceptos de los módulos están muy relacionados con los de *tipo*, ya que un sistema fuertemente tipificado permite detectar inconsistencias entre módulos, previniendo el mal manejo en la comunicación.

De esta discusión se hace importante profundizar un poco en algunos conceptos fundamentales de la teoría de tipos. En el artículo de Cardelli y Wagner "On Understanding Types, Data Abstraction and Polymorphism" (Cardelli 85), se define que los tipos surgen para categorizar conjuntos de objetos de acuerdo a su uso y comportamiento.

Los tipos se asocian, en los lenguajes de programación, a constantes, variables, operadores y símbolos de funciones. Un lenguaje en el que el tipo de una expresión se puede inferir haciendo un análisis al momento de compilación se llama *tipificado estáticamente*. Y aquellos en los que todas las expresiones deben ser consistentes en el tipo se les llama *fuertemente tipificados*. Si se tienen estas dos características, el compilador puede garantizar un programa sin errores de tipo, lo que hace más eficiente la corrida.

Pascal es un ejemplo de un lenguaje tipificado pero no fuertemente, ya que permite la mezcla de tipos como al sumar enteros con reales.

En relación al concepto de tipo, hay tres líneas de búsqueda en el desarrollo de lenguajes de programación para permitir la reusabilidad, ellas son *seguridad*, *expresividad* y *generalidad* o *flexibilidad*.

La línea de la seguridad, se refiere a tener una disciplina de tipos que se validen estáticamente de forma que se eviten los errores al momento de ejecución. Esto significa tener lenguajes fuertemente tipificados y estáticos.

La expresividad, parte de la idea de tener la posibilidad de definir nuevos tipos de datos que sean más adecuados para las aplicaciones de los usuarios. A estos lenguajes se les llama *sistemas de tipos extendibles* y deben tener mecanismos que permitan la definición de los nuevos tipos con las operaciones para manipularlos. Una manera de hacerlo es permitir encapsular en una sola entidad sintáctica datos y operaciones ocultando la representación concreta (como están implementados). Esto aumenta la expresividad de los lenguajes, facilita la portabilidad de los tipos de datos y también provee de seguridad.

La tercera línea parte de los lenguajes monomórficos, aquellos en que los datos y funciones tienen un sólo tipo, a fin de permitir mayor flexibilidad o generalidad por medio de funciones polimórficas o genéricas en las que el tipo es un parámetro. El polimorfismo se refiere a que las variables y funciones pueden referenciar o ser aplicadas a valores de distintos tipos. El ejemplo típico de una función así , es la que calcula la longitud de una lista ya que su dominio puede ser cualquier clase de lista y regresa el número correspondiente a su longitud.

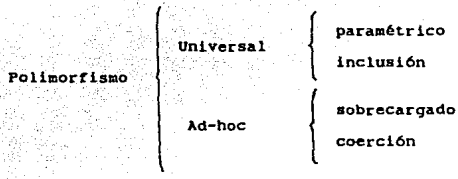
Parametrización:

Otra característica fundamental para la reusabilidad es la *parametrización* de los módulos. En un módulo se puede declarar uno o varios parámetros formales, los cuales se instancian en el momento en que se hace uso del módulo. Por ejemplo, un módulo que define a un árbol puede estar parametrizado por el tipo de elementos que contiene. Al hacer uso del árbol se indica que los elementos son por ejemplo, enteros o reales o cualquier otro tipo.

Según Goguen (Goguen 87) la programación parametrizada permite maximizar el reuso de programas, ya que se tienen en la forma más general posible, pues al sustituir diferentes parámetros se puede reusar el programa o módulo en diversas situaciones.

Pero es importante establecer con exactitud (esto es, incluyendo información sintáctica y semántica) cuáles son los requerimientos que deben cumplir los parámetros actuales al instanciar a los formales. Esta información se dá a través del tipo que debe satisfacer el argumento. De esta forma la programación parametrizada se facilita si se permite el manejo de algún polimorfismo en las funciones.

Según Cardelli se tienen varios tipos de polimorfismo, que se muestran en el siguiente esquema:



El *polimorfismo universal* permite que una función trabaje con un número infinito de tipos (todos los que tengan una estructura común). Al hacer la implementación de la función se hace a través de un sólo código que funciona con cualquiera de los tipos permisibles.

Polimorfismo ad-hoc trabaja con un conjunto finito de tipos diferentes y potencialmente sin relación entre ellos. Puede considerársele como un conjunto de funciones monomórficas, ya que la implementación es en realidad, un conjunto de códigos que se ejecutan según el tipo del argumanto.

El *polimorfismo paramétrico* se tiene cuando una función trabaja uniformemente en un rango de tipos, de forma que se puede usar en diferentes contextos sin hacer cambios a representaciones especiales o pruebas al momento de ejecución. Esta es la forma, según Cardelli, más pura de polimorfismo.

En el *polimorfismo por inclusión* un objeto puede pertenecer a diferentes clases que no necesariamente son ajenas, esto es, hay inclusión en las clases. En Smalltalk, y otros lenguajes orientados a objetos se encuentra el polimorfismo por inclusión como una relación entre tipos que permite que las operaciones se apliquen a objetos relacionados por inclusión en las clases. Por ejemplo se tiene una clase llamada polígono en la que se dan las definiciones básicas y las operaciones para este tipo de objeto, se pueden definir dos subclases, como rectángulos y triángulos, que heredan las propiedades de polígono.

Polimorfismo por sobrecargo de nombres es el que a un operador asocia dos o más funciones distintas con el mismo nombre y es en el contexto de ejecución cuando se define en concreto a que función se está haciendo referencia en una operación particular. Ejemplos típicos de esta clase de polimorfismo son las funciones para sumar o restar (+, -) que se pueden referir a números enteros o reales según sea el tipo de los operandos.

Polimorfismo por coerción es el que requiere de alguna acción de transformación para convertir un argumento al tipo esperado por una función. Ejemplo de esta situación es cuando se suman reales y enteros, como por ejemplo en Pascal, en que se transforma la representación del entero a una de real antes de realizar la suma.

El lenguaje paradigmático del polimorfismo paramétrico es ML, desarrollado con esta filosofía.

Existen en este momento distintos estilos de programación que incorporan estas líneas. Por un lado están los lenguajes imperativos, que a su vez se dividen en *de procedimientos* como Modula-2, Ada y Pascal; *concurrentes* como Pascal Concurrente; y *orientados a objetos*, entre los que están Smalltalk y Eiffel.

Por otro lado se tienen a los lenguajes declarativos, que a su vez se dividen en *funcionales* entre cuyos representantes están ML y Lisp 1.5 y *lógicos* como Prolog.

2. Conceptos generales de ML.

ML tiene varias propiedades interesantes para la experimentación, presenta muchos de los requerimientos hechos a los lenguajes a fin de facilitar la construcción de software reusable. A continuación se hace una revisión somera de sus características principales y se ahonda en aquellas que facilitan la reusabilidad.

2.1 ML es un lenguaje funcional.

ML (Meta Lenguaje) es un lenguaje funcional que surge como parte de un proyecto de Gordon en 1979, que consistía en desarrollar un sistema generador de pruebas llamado LCF, para razonar sobre funciones recursivas en el contexto de los lenguajes de programación. Dicho sistema tenía dos partes un cálculo deductivo PPλ (Cálculo de predicados polimórficos) y un lenguaje interactivo de programación llamado ML o Metalenguaje, el cual se encontró interesante por sí mismo, por lo que se continuó su desarrollo como lenguaje funcional. Posteriormente Milner hace el esfuerzo de estandarizarlo dando origen a SML o Standard ML (en este trabajo se le menciona como ML en realidad es SML), el cual se encuentra reseñado en el trabajo de (Harper 85), quien dice que es el resultado de dos años de trabajo por parte de 15 personas en Edimburgo.

ML es un lenguaje funcional que pertenece a la línea de los lenguajes declarativos cuyo objetivo es definir *qué* se va a calcular, en vez de decir *cómo*. Esto se logra porque su acción principal es la aplicación de funciones a argumentos. Por esto sus entidades de primera clase son las funciones, las cuales pueden ser recursivas (de hecho este es el mecanismo principal de iteración en el lenguaje), de alto orden o polimórficas, y se tienen mecanismos que facilitan su definición y uso.

Sintácticamente las funciones tienen formas ecuacionales, del lado izquierdo se tiene el nombre y un patrón, del lado derecho la regla para la aplicación de la función. La evaluación se hace primero con un ajuste de patrones con los argumentos y después regresando el valor determinado por el lado derecho.

Otras características que hacen interesantes a los lenguajes funcionales (HudaK 89) en general son:

-En ellos se evita todo tipo de construcciones que provoquen efectos laterales, ya que cada función no tiene otro efecto que calcular un resultado. Los lenguajes funcionales *puros* no contienen asignaciones, así que una vez que se da un valor a una variable, éste no cambia. Eso elimina errores y permite que se pueda poner en una expresión a una variable o su valor. A esta característica se le llama *transparencia referencial*.

-Los programas funcionales se escriben más rápido, pues sólo se define qué hacer sin preocuparse de detalles de más bajo nivel.

-Son de más alto nivel, pues están más cercanos a la notación matemática.

-Son más adecuados para razonamientos formales y análisis, lo que facilita la verificación de los programas.

-Se pueden ejecutar más fácilmente en arquitecturas paralelas, ya que no llevan ninguna noción de secuenciación explícita. Las funciones se pueden ejecutar en paralelo.

-Los lenguajes funcionales modernos incluyen funciones de alto nivel, evaluaciones flojas (del inglés lazy), apareamiento de patrones y varias formas de abstracción de datos.

ML tiene todas estas características y un sofisticado sistema de módulos y excepciones. Sin embargo, no es 100% funcional, ya que permite tener referencias a localidades (de forma parecida a la noción de variables en otros lenguajes), y un sistema de entrada salida que permite los efectos laterales y que lo hace No referencialmente transparente. Sin embargo en este trabajo se hace uso del núcleo puro.

2.2 ML es interactivo

El lenguaje ML, como se vió por su origen, es interactivo. Si se declara una función, lo que se evalúa inmediatamente es la declaración, no la función. Si esta evaluación determina que efectivamente se trata de una función, entonces si se procede a evaluarla mediante otra llamada al intérprete. Así al recibir una expresión ML la analiza, evalua y regresa tanto el valor de la expresión como su tipo. Por ejemplo si se le da la expresión

```
- val x = 4 * 5;
```

contesta

```
> val x = 20 : int
```

(el *símbolo de solicitud* de ML es un guión y al contestar lo hace indicándolo con ">"). Sin embargo es posible construir un archivo con varias expresiones que ML va leyendo y respondiendo a cada una.

2.3 ML y su sistema de tipos y polimorfismo.

Otra característica importante es su sistema de tipos, hace una validación estática y estricta de tipos por lo que no hay errores de tipo al momento de ejecución. ML es el primer lenguaje de programación en incluir a su semántica un sistema de inferencia de tipos, lo que hace que no se requiera declaración explícita de tipo para cada expresión. Por ejemplo si se declara la función:

```
fun f(x)  = x + 1;
```

el sistema responde que f es una función de los enteros a los enteros (f : int -> int), lo que infiere del hecho de que en la regla se suma 1, que es entero, y puesto que no hay mezcla de tipos x debe ser entero, y también de este tipo debe ser el resultado de la suma. Existen algunos casos como al definir

```
fun g(x) = x + y;
```

en los que el sistema no tiene forma de saber que tipo tienen los operandos x y y, por lo tanto no puede inferir el tipo de la función, por lo que solicita más información, la cual se puede proveer así

```
fun g(x:int) = x + y;  o  fun g(x) =x + y:int;
```

con lo que ya puede inferir que g es función de los enteros a los enteros (g: int -> int).

ML puede hacer inferencias no sólo para los tipos tradicionales, sino también para los polimórficos. Por ejemplo, al definir la función long para calcular la longitud de una lista de cualquier tipo de elementos, se define

```
fun long(x::xs) = if x = nil then 0
                  else 1 + long (xs);
```

en este caso no se sabe que tipo de elementos tiene la lista (x::xs), sólo se sabe que está formada por un primer elemento x y resto xs. El sistema responde long : 'a list -> int, (en donde 'a es la forma en que se indica un tipo variable o polimórfico). Al pedir que se evalúe dicha función, por ejemplo mediante

```
-long [1,2,3,4]
 >4 : int
```

el sistema infiere ahora que es una función de listas de enteros a enteros, la aplica y contesta

```
-long;
>long : int list -> int
```

Cuando se tienen funciones polimórficas, siempre se infiere el tipo más general posible. Por otro lado, hay que recalcar que al tener funciones polimórficas paramétricas como *long*, en realidad se tiene un conjunto de funciones con representación uniforme, de las cuales se elige una al definir el tipo concreto del parámetro. Este mecanismo corresponde al *polimorfismo paramétrico*.

A continuación se presentan algunos ejemplos de definiciones en ML para analizar algunas de estas características.

En primer lugar se da la definición de un tipo de datos polimórfico, paramétrizado y recursivo al que se denomina *'a arbol*. Se dice parametrizado porque *'a* juega el papel de parámetro al ser instanciado por tipos particulares, por ejemplo *int* y *string*, se obtienen las instancias de *int arbol* (árbol de números enteros) y *string arbol* (árbol de cadenas).

Es polimórfico porque las operaciones que a continuación se definen en términos del parámetro *'a* tienen sentido para cualquier árbol sin importar el tipo de sus elementos. Es además recursivo pues sus constructoras son las funciones *vacio* para el árbol vacio, *hoja of* para definir como son las hojas y *nodo of* que indica que el árbol puede tener subárboles del mismo tipo:

```
datatype 'a arbol=vacio |hoja of 'a |nodo of 'a arbol* 'a arbol
```

Al recibir esta definición ML contesta:

```
>datatype 'a arbol
   con  vacio : 'a arbol
   con  hoja  : 'a -> 'a arbol
   con nodo   : 'a arbol * 'a arbol -> 'a arbol
```

Enseguida se dan dos funciones definidas para dicho tipo. La primera es la función frontera con la que se obtiene la lista de los valores de las hojas del árbol, la definición es:

```
fun frontera vacio = ()
 |  frontera (hoja (x)) = ( x)
 |  frontera (nodo (t1,t2)) = frontera (t1) @ frontera(t2);
```

a lo que contesta

```
val frontera = fn : 'a arbol -> 'a list
```

indicando que es una función que recibe un 'a arbol y regresa una 'a lista, o sea es una función polimórfica. La frontera del árbol vacio es la lista vacia, la frontera de una hoja es la lista conteniendo el valor de dicha hoja y la frontera de un nodo con dos subárboles es la lista de la concatenación de las fronteras de ambos.

A continuación se define la función que indica el número de nodos:

```
fun numnodos vacio = 0
 |  numnodos (hoja(x)) =1
 |  numnodos (nodo (t1,t2)) = numnodos (t1) + numnodos (t2);
```

a lo que contesta ML:

```
> val numnodos = fn : 'a arbol -> int
```

Una vez definidos el tipo de datos y las funciones se pueden aplicar a algún caso concreto, como por ejemplo

```
val arbolito= nodo(hoja("a"),nodo (hoja("b"),hoja("c")));
```

a lo que ML contesta

```
>val arbolito = nodo(hoja("a"),nodo(hoja("b"), hoja("c"))) :
                                                    string arbol
```

y se le pueden aplicar las funiones anteriores:

```
-frontera arbolito;
> ("a","b","c")

-numnodos arbolito;
> 3 : int
```

2.4 ML y el manejo de excepciones.

ML tiene un mecanismo para el manejo de situaciones excepcionales. Su propósito es tener un medio para manejar situaciones en las cuales no está definida una función sin tener violaciones a la disciplina de los tipos, activando una excepción. Esto se hace por medio de los comandos: *exception* que define a la excepción y *raise* que al presentarse la condición indeseable activa la excepción.

Un ejemplo de una situación en que se requiere definir una excepción se da en el manejo de las estructuras lineales. Al querer efectuar la función que regresa el primer elemento, si ésta es vacía da un error (ver especificación algebraica). En ML se define una excepción para este caso como se ilustra a continuación:

```
exception Inicial
fun inicial Estrvacia                = raise Inicial
  | inicial (Inserta (e , l ))       = inicial (l)
```

De esta manera la función está definida para todos los casos; al presentarse la situación especial, el manejador de excepciones de ML activa la excepción Inicial sin problemas en la consistencia del manejo de tipos, ya que por omisión en la definición las excepciones tienen tipo *unit* que es el más elemental en ML. Para mayores detalles en el manejo de las excepciones se puede consultar la bibliografía.

2.5 ML y el manejo de módulos.

Como se mencionó una de las facilidades más novedosas de ML es su sistema de módulos, que permite la programación modular genérica. La gestación de este sistema , según (Mac Queen 85, 90), primero estuvo influenciada por el lenguaje Clear de Burstall y Goguen. Fue diseñado para manejar descomposición modular de las especificaciones algebraicas. Luego fue influenciado con las ideas de la Teoría de Tipos, por lo que se pensó en que fuera una generalización natural del sistema de tipos polimórficos del mismo ML por medio de módulos paramétricos.

Los tres componentes centrales del sistema de módulos son las estructuras (structures), las signaturas (signatures) y los functores (functors).

Una estructura es un ambiente (*environment structure*) que encapsula una colección de tipos relacionados y operaciones sobre ellos, recibe un nombre y puede ser manipulada por otras partes del programa formando un ambiente.

Una signatura es un modelo de una clase de estructuras que comparten un esquema común, las signaturas juegan el papel de tipo de las estructuras. Consiste en una secuencia formada por identificadores de tipos, por identificadores de valores individuales junto con sus tipos y por las excepciones, todos ellos correspondientes a los componentes de la estructura.

Un functor es una función que mapea una estructura en otra, permite construir sistemas mediante su aplicación. Se puede escribir un sistema usando sólo signaturas y functores y se ejecuta al aplicar los functores.

Las signaturas corresponden a las interfaces o signaturas en las especificaciones algebraicas. Los functores son similares a los módulos parámetricos o paquetes genéricos de otros lenguajes

Así, ML es un lenguaje estratificado que tiene, por un lado tipos, valores y funciones (nivel del núcleo) y por otro lado signaturas, estructuras y functores (sistema de módulos). Los dos niveles están relacionados pero no se mezclan. Las funciones no se pueden aplicar a estructuras y los functores no regresan valores. Se puede pensar en la signatura de una estructura como el tipo de un valor, y los functores son como las funciones. Además las estructuras son objetos de primera clase, como los valores, por lo que pueden pasarse como argumentos a los functores, los cuales a su vez las regresan como resultados. Sin embargo, los functores no son objetos de primera clase por lo que no pueden ser parámetros, ni obtenerse de la aplicación de otro functor.

En el nivel de manejo de módulos, lo análogo a la validación de tipo, es la satisfación que se da en términos de la correspondencia de componentes entre una estructura y una signatura. Una estructura *satisface* una signatura si tiene al menos todos los componentes requeridos por la signatura, y los tipos de estos componentes son al menos tan generales como los tipos dados en la signatura. Una estructura no necesariamente debe corresponder exactamente a la signatura, basta que al menos tenga el mismo número de componentes y que haya una correspondencia de tipos, aunque la estructura tenga más componentes.

Otra discrepancia permitida es que los valores declarados en una estructura tengan tipos polimórficos que instancien al tipo especificado en la signatura. Una estructura puede corresponder a varias signaturas dependiendo del grado de polimorfismo; de cuál es el componente formal de tipo considerado; o del número de componentes. Para cada estructura existe una signatura *más general* (igual que con los tipos), que representa la información completa y estática asociada a una estructura y es la inferida para alguna estructura que no tenga una signatura definida en forma explícita.

Por ejemplo, en la signatura *ELEMENTO* se especifica que debe tener como mínimo la declaración de un tipo llamado *elemento*.

```
signature  ELEMENTO =
sig
     type   elemento
end;
```

Las siguientes estructuras satisfacen dicha signatura:

```
structure  Enteros :  ELEMENTO =
struct
     type  elemento = int
end;
```

```
structure  Cadenas : ELEMENTO =
struct
     type  elemento = string
end;
```

Existe una infinidad de estructuras que satisfacen la signatura *ELEMENTO*, ya que la estructura puede tener más componentes de los indicados.

A continuación se retoma el ejemplo de las *Estructuras Lineales* que se analizó en el capítulo I y se muestra como se pueden construir los módulos correspondientes en ML.

Como se vió anteriormente, para el manejo de estructuras lineales se tiene una especificación parametrizada por *Elementos*. En ML se implementa por una signatura *ESTRLINEALB* que corresponde a la signatura en la especificación y un functor al que se denomina *Estrlineal* parametrizado por una estructura que debe satisfacer una signatura *ELEMENTO*.

```
signature ESTRLINEALB =
sig
   structure Elem : ELEMENTO
   type estrlineal
(*  operaciones *)
   val EstrVacia: estrlineal
   val Inserta:( Elem.elemento * estrlineal) -> estrlineal
   val resto: estrlineal -> estrlineal
   val inicial: estrlineal -> Elem.elemento
   val esVacia: estrlineal -> bool
end;
```

(Se acostumbra poner el nombre de las signatura con letras mayúsculas y los nombres de estructuras y functores con la inicial en mayúscula).

Al comparar esta signatura con la de la especificación algebraica se puede ver que se corresponden. Se definen el tipo del parámetro, el género con las constructoras que aparecen en la declaración *datatype* y demás operaciones con sus dominios y codominios.

A continuación se define al functor que mapea una estructura de signatura *ELEMENTO* a una estructura *Estrlineal* correspondiendo a la anterior signatura *ESTRLINEALB*. (Se numeran las líneas para facilitar su explicación posterior).

```
1.          functor Estrlineal ( E:ELEMENTO ) : ESTRLINEALB =
2           struct
3              strucutre Elem   = E
4              datatype estrlineal = EstrVacia | Inserta of
                                     (Elem.elemento *estrlineal)

5              exception Resto
6              fun resto  EstrVacia      = raise Resto
                | resto (Inserta (e,l))  = l

7              exception Inicial
8              fun inicial EstrVacia       = raise Inicial
                | inicial (Inserta (e , l ))    = e

9              fun esVacia EstrVacia      = true
                | esVacia  _               = false
10          end;
```

Este functor se adecua a la parte de las ecuaciones de la especificación algebraica. En la línea 3 se hace la identificación de la estructura Elem con el de la estructura que entra como parámetro. En la línea 4 se define el tipo *estrlineal* de la signatura, que tiene como constructoras a Estr*Vacia* e Inserta.

En seguida se definen las funciones correspondientes a las otras operaciones que son una reescritura de las ecuaciones de la especificación. Por ejemplo:

La definición de la operación *resto* es : El resto de una estructura vacia genera una excepción *Resto*, que es el caso indicado como error en la especificación y que garantiza que la definición de la operación esté completa. El axioma C2 de la especificación está implementado por el caso alternativo, denotado por "|" en ML.

De igual manera, la operación *esVacia* es verdadera en el caso de *Estrvacia* (axioma C5) y falso para cualquier otro (axioma C6) que en ML se puede indicar por "_" .

Para construir una estructura lineal de enteros o de cadenas, se llama al functor con la estructura correspondiente. Por ejemplo, se puede obtener una estructura llamada *EstrEnt* aplicando el functor a la estructura *Enteros* definida antes:

```
structure  EstrEnt  = Estrlineal (Enteros );
```

la estructura obtenida es una estructura lineal de enteros. De igual modo para tener una de cadenas se invoca al functor con la estructura *Cadenas*:

```
structure  EstrCad  = Estrlineal (Cadenas);
```

3. Construcción de una biblioteca de estructuras de datos en ML.

3.1 Metodología para la construcción de una biblioteca de implementaciones en ML para la biblioteca de especificaciones algebraicas.

El sistema de manejo de módulos de ML es una herramienta poderosa y adecuada para el desarrollo de una biblioteca de implementaciones basada en especificaciones algebraicas. Una de las ventajas de trabajar con ML es que se le puede considerar como un lenguaje de especificación funcional, en el que las especificaciones se pueden ejecutar. Además hay una identificación con las especificaciones algebraicas, de forma que a la signatura de una especificación, corresponde casi directamente a una signatura en ML.

El comportamiento de las operaciones se traduce en funciones agrupadas en un functor que además permite implementar la parametrización, para generar la estructura deseada. También es posible mantener la jerarquización de las especificaciones por medio de los functores.

Por lo tanto el método para construir la biblioteca de implementaciones en ML consiste en:

1. Preservar la estructura de la biblioteca de especificaciones, lo que se logra al aplicar functores que llevan como parámetro lo indicado en la cláusula de importación correspondiente a la estructura de nivel superior en la gráfica de la jerarquía de las especificaciones (ver fig. 1.1).

2. Cada especificación se implementa por:

- Una signatura (nombrada como la especificación pero con mayúsculas) que define el tipo y la signatura de las operaciones que se exportan. Cuando hay operaciones que se quieren ocultar se omiten en la signatura. Una signatura puede incluir una estructura importando todo lo definido en ella.

- Un functor (con igual nombre) parametrizado por la estructura que toma de entrada . En el cuerpo del functor se define la subestructura del parámetro y todas funciones a exportar.

Cuando en una estructura de datos se quiere exportar todo lo heredado del parámetro más algunas operaciones, puede dejarse al functor sin asociarle una signatura explícita, permitiendo al sistema inferir la más general. Otra situación en que no se asocia una signatura a un functor se presenta cuando se exporta todo lo importado, pero redefiniendo algunas operaciones. En este caso puede asignársele una signatura al momento de ejecutarlo, lo que le da versatilidad. Otra manera de lograr este mismo efecto es definir la signatura repetiendo todo lo de la signatura que se desea exportar pero puede resultar engorroso.

Las operaciones correspondiente a los axiomas se definen por medio de funciones. El renombramiento se logra indicando cual es el nombre actual de la función y de cual procede.

Para describir la estructura de la biblioteca en ML se usarán unas gráficas de la siguiente manera:

Se encierra en un círculo el nombre de la estructura de datos,el tipo que genera (si lo hay) y las operaciones exportables (lo que se define en la signatura correspondiente). Si hay algún renombre se indica por una igualdad.

Estos círculos se conectan entre sí por varios tipos de flechas:

indica la aplicación de un functor cuyo parámetro es la estructura de origen y que genera la estructura a la que apunta la flecha.

Significa que la signatura de la estructura de origen, satisface a la signatura señalada. Es de notar que estos dos tipos de flechas van en sentidos opuestos. Con esto se pretende indicar que la nueva signatura tiene mas restricciones que la original, debido a las operaciones agregadas.

Indica que toma como parámetro a otra estructura no mostrada en la gráfica.

Igual que en la biblioteca de especificaciones, aquí se tiene una gráfica para cada tipo de datos con sus variaciones: elementos, estructuras lineales, árboles, conjuntos y gráficas. A continuación se describen con mayor detalle los módulos de la biblioteca en ML, la cual se puede consultar en los apéndices B y C.

3.2 Descripción de la biblioteca en ML.

3.2.1 Elementos.

Se construye un archivo llamado *elementos.sml* que contiene la definición de las signaturas correspondientes a las especificaciones de los parámetros *ELEMENTOS*, *ELEMCOMPAR* y *ELEMORDEN*. En la signatura *ELEMENTO* se describe la familia de álgebras que consisten de un solo tipo genérico de nombre *elemento*.

ELEMCOMPAR corresponde a las álgebras que en su signatura incluyen un tipo y una operación binaria de valores booleanos, a ésta se le designa con el nombre genérico *igual* porque se espera que vista como un predicado tenga las propiedades de una relación de igualdad, en el sentido lógico.

En *ELEMORDEN* además del tipo y del valor *igual* se requiere de otro valor *menor*, nombre genérico de una operación que se espera corresponda a un orden lineal irreflexivo, similar a la relación < entre números. Como se aprecia *ELEMORDEN* satisface a *ELEMCOMPAR* y ésta a *ELEMENTO* (ver fig 3.1.).

En ese mismo archivo se definen las estructuras correspondientes a los elementos básicos de las especificaciones. Estas estructuras satisfacen las signaturas siguientes: los Booleano satisface a *ELEMCOMPAR*, se instancia el tipo genérico *elemento* con el tipo bool de ML y la función genérica *igual* con la igualdad de bool.

Las estructuras *Natural* y *Real* satisfacen ambas la signatura *ELEMORDEN*; en aquellas aparecenrespectivamente los tipos *int* y *real* de ML como instencias del tipo genérico *elemento* de *ELEMORDEN*; las funciones (genéricas) *igual* y *menor* de *ELEMORDEN*, son instanciadas por las relaciones = y < usuales entre números enteros y reales que ML proporciona.

De igual forma se construye la estructura *CADENA*, usando el tipo *string* de ML en lugar de *int* o *real*. De esta no aparece la especificación algebraica en la biblioteca de especificaciones, pero se ha incluido porque ML no proporciona el tipo *char* que permitiría construir directamente la estructura correspondiente a la especificación *Caracteres*. Esta se puede construir a partir de *Cadena*, pero no se presenta aquí una construcción tal por ser innecesaria para este trabajo.

En la figura 3.1 se presentan entre círculos las signaturas y en rectángulos las estructuras.

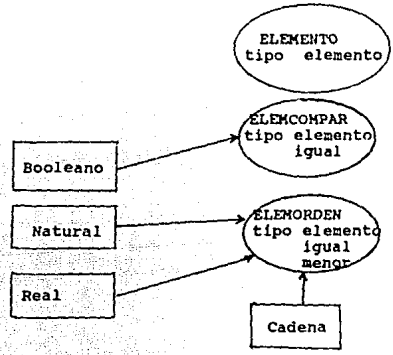


Fig. 3.1. Estructura Archivo *elementos.sml*

3.2.2 Estructuras Lineales.

Para implementar a las estructuras lineales, se hace uso del archivo *elementos.sml*

Primeramente se define una signatura *ESTRLINEALB* que corresponde a la signatura de la especificación en la que se define el parámetro requerido que es una estructura que debe satisfacer la signatura *ELEMENTO*. Define el tipo *estrlineal* y las operaciones *Estrvacia, Inserta, resto, inicial* y *esvacia* con sus dominios y rangos. (Ver ejemplo de la sección anterior).

Enseguida se define un functor *Estrlineal* que toma a una estructura E que satisfaga a *ELEMENTO* y regresa una estructura lineal. De esta forma si se aplica el functor a cualquier estructura que satisfaga mínimamente el requisito de definir un tipo, se puede obtener una estructura lineal de esos elementos.

Para obtener una lista, se define la signatura *LISTA* que requiere de una *Estructura lineal* con todas sus operaciones, definiendo además las operaciones *longitud, enésimo* y *concatena*. Posteriormente se define el functor *Lista* que toma una estructura lineal como parámetro y regresa la estructura de *Lista* en la que se renombra *EstrVacia* como *ListaVacia* y se proporciona el comportamiento de las funciones agregadas. (ver figura 3.2)

El functor Pila requiere de una estructura lineal y regresa una estructura que satisface la signatura PILA, renombrando las funciones que se tienen en las estructuras lineales a las usuales en este tipo de datos.

Para obtener una Pila acotada, se tiene un functor que requiere de una *Pila* y una *Cota* como parámetros. por lo que requiere de un archivo llamado *cotas.sml* que se usa para las pilas y colas acotadas y en el que se define una signatura *COTA* que requiere de una constante *max* de tipo *int* y varias estructuras con valores diferentes para ese valor. no se le asigna signatura alguna ya que exporta lo que hereda de las pilas con la función Mete modificada y las operaciones *tamaño* y *estallena*. (ver fig. 3.2.)

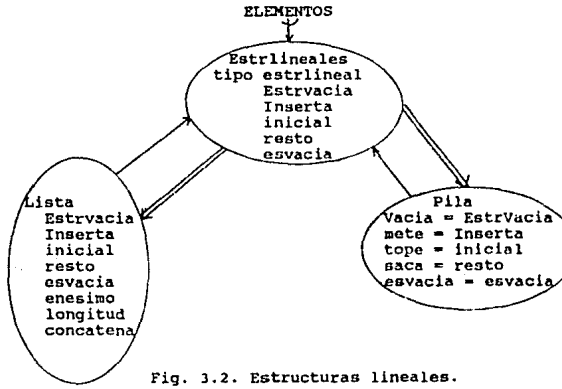


Fig. 3.2. Estructuras lineales.

Para obtener colas se construye una signatura *ESTRCOLA* que tiene como parámetro una estructura lineal y define operaciones para insertar y extraer los elementos por ambos extremos (conforme a la especificación *Estrcolas*). De *ESTRCOLA* se seleccionan las funciones necesarias para cada tipo de colas.

```
signature ESTRCOLA =
sig
   structure Ec : ESTRLINEALB
   val ColaVacia : Ec.estrlineal
   val Meteult : (Ec.Elem.elemento * Ec.estrlineal)-> Ec.estrlineal
   val ultimo : Ec.estrlineal -> Ec.Elem.elemento
   val sacault : Ec.estrlineal -> Ec.estrlineal
   val esvacia  : Ec.estrlineal -> bool
   val Meteprim : (Ec.Elem.elemento * Ec.estrlineal)->Ec.estrlineal
   val primero  : Ec.estrlineal -> Ec.Elem.elemento
   val sacaprim : Ec.estrlineal -> Ec.estrlineal
end;
```

El functor *Estrcola* renombra las operaciones a fin de tener la nomenclatura usual en las colas, *EstrVacia* es *ColaVacia*, Inserta se vuelve *Meteult*, *inicial* es *ultimo*, etc.

```
functor Estrcola (E: ESTRLINEALB ) : ESTRCOLA =
struct
   structure Ec = E
   open Ec
(* se renombran las operaciones de estructuras lineales  *)
   val ColaVacia = EstrVacia
   val Meteult   = Inserta
   val ultimo    = inicial
   val sacault   = resto
   val esvacia   = esVacia
(*  se definen las nuevas operaciones   *)
   fun Meteprim (e,c) = if esvacia (c) then Meteult (e,ColaVacia)
                  else Meteult (ultimo(c), Meteprim (e,sacault(c)))
   exception Primero
   fun primero (c) = if esvacia(c) then raise Primero
                   else if esvacia(sacault (c)) then ultimo (c)
                        else primero (sacault(c))
```

```
   exception Sacaprim
   fun sacaprim (c) =if esvacia (c) then raise Sacaprim
                     else if esvacia (sacault (c)) then ColaVacia
                       else Meteult(ultimo(c),sacaprim(sacault(c)))
end;
```

La signatura *COLA* y el functor *Cola* sirven para generar estructuras correspondientes a las colas simples, con las operaciones necesarias para que se pueda introducir por un extremo y sacar por el otro.

```
signature COLA =
sig
   structure Eco  : ESTRCOLA
   val ColaVacia : Eco.Ec.estrlineal
   val Meteult : (Eco.Ec.Elem.elemento*Eco.Ec.estrlineal)->
                                                  Eco.Ec.estrlineal
   val primero   : Eco.Ec.estrlineal -> Eco.Ec.Elem.elemento
   val sacaprim   : Eco.Ec.estrlineal -> Eco.Ec.estrlineal
   val esvacia   : Eco.Ec.estrlineal -> bool
end;


functor COLA (E:ESTRCOLA) : COLA =
struct
   structure Eco = E
   open Eco
   val ColaVacia = Ec.EstrVacia
   val Meteult  = Ec.Inserta
   val primero  = ultimo
   val sacaprim  = sacault
   val esvacia   = Ec.esVacia
end;
```

Las colas dobles requieren de todas las funciones definidas en *ESTRCOLA*, por lo que al functor *Coladoble* no se le asocia una signatura en forma explícita, dejando que el sistema le asocie la signatura más general, que incluye todas las operaciones definidas en *ESTRCOLA*.

```
functor Coladoble (E:ESTRCOLA ) =
struct
   open E

end;
```

El caso de Colas Acotadas, es semejante al de pilas acotadas y las colas con prioridad, parecidas a aquellas. (Consultar la fig. 3.3)

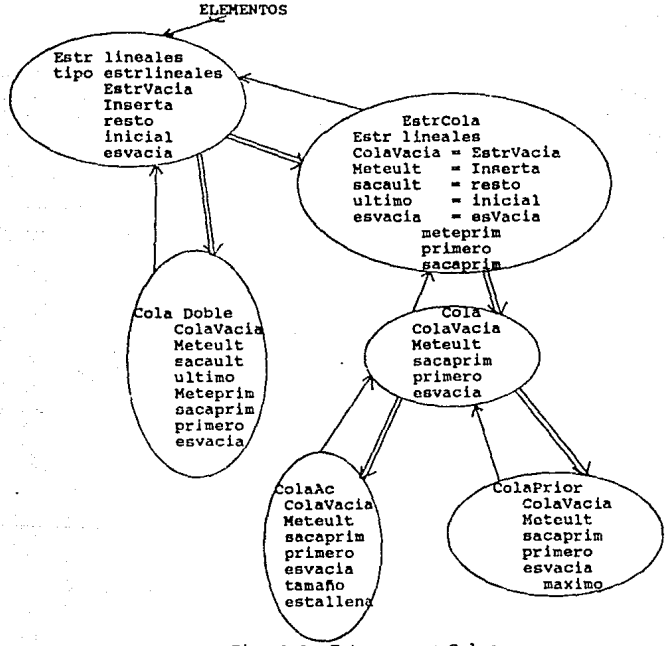


Fig. 3.3. Estrucruras Colas.

3.2.3 Arboles.

La estructura de los árboles es semejante a la correspondiente en la biblioteca de especificaciones. Primero se construye la signatura *ARBOLBIN*, en la que se define el parámetro que es un tipo *elemento*, enseguida el tipo de datos *arbolbin* con las constructoras *ArbolVacio* y *CreaArbol*, luego se da la signatura de las operaciones auxiliares que son *izq*, *der*, *raiz*, *altura* y *esVacio*.

En el functor *Arbolbin* se tiene como parámetro a una estructura que satisfaga a *ELEMORDEN* (en este caso no se requiere del orden entre elementos, pero los functores que lo requieren como parámetro sí) y regresa una estructura satisfaciendo la signatura anterior. En el cuerpo del functor se dan las funciones para cada operación según los axiomas de la especificación.

La siguiente signatura corresponde a *EXTIENDEARBOL* que como su nombre lo indica , importa y exporta las operaciones de árbol binario más otras útiles en el manejo de los árboles, como son *eshoja*, *numnodos*, *numhojas* y *estalleno*. El functor *Extarbol* toma una estructura de árbol binario y regresa otra de árbol extendido.

Para la implementación de árboles *piramidales* o *heap* se usan los dos functores anteriores, ya que el functor *Heap* toma como parámetro a una estructura satisfaciendo *EXTIENDEARBOL* y regresa un árbol piramidal. (Ver fig. 3.4).

En la signatura *HEAP* se definen las operaciones que son exportadas. En este caso, hay algunas que es importante que queden ocultas como la constructora *CreaArbol*, la cual es remplazada por *inserta* que agrega elementos al árbol vigilando que se sigan manteniendo las condiciones de árbol piramidal. En el functor *Heap* hay algunas operaciones que no están definidas en la signatura correspondiente, dichas operaciones son ocultas y sirven para facilitar la definición de las exportables.

A partir de los árboles binarios se construyen los árboles balanceados por la altura o *AVL* (ver fig. 3.4). En la signatura *ARBOLAVL* se indican las operaciones exportables, en donde también hay ocultamiento de *CreaArbol* a fin de definir nuevas funciones como inserta que mantiene la condición de balanceo. El functor *ArbolAVL* toma un árbol binario y regresa una estructura que satisface a la signatura *ARBOLAVL*. En este caso también hay en el functor definición de funciones ocultas, como *rotaizq*, *rotader*, *balanceizq* y *balanceder*, auxiliares en la definición de *inserta* y *sustrae*.

El último tipo de árbol en la biblioteca es el árbol de búsqueda, que es también un árbol binario, pero con la condición de que los valores en los nodos del subárbol izquierdo de cualquier nodo sean menores que él, y los del derecho mayores. Así la signatura *ARBOLBUSQ* indica las operaciones que exporta para mantener dichas condiciones. El functor *Arbolbusq* toma un árbol binario y lo convierte en uno de búsqueda (ver fig. 3.4)

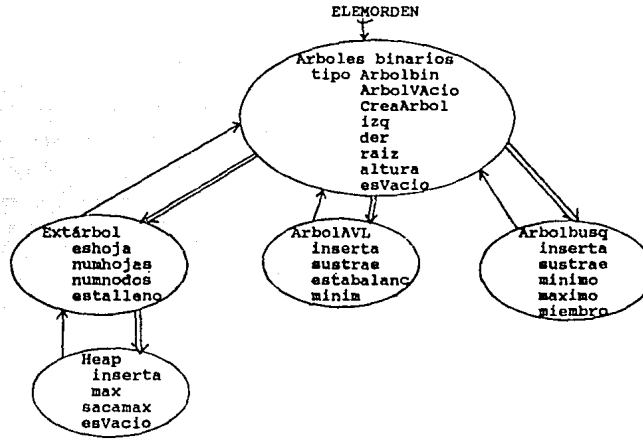


Fig 3.4. Arboles

3.2.4. Conjuntos.

El tipo de datos conjunto se implementa por una signatura *CONJUNTO* que toma como parámetro un tipo *elemento*, define al tipo conj con las constructoras *ConjVacio e Inserta*, y las operaciones para el manejo de conjuntos. El functor *Conjunto* toma una estructura que satisfaga a *ELEMCOMPAR* para poder establecer igualdad entre los elementos y evitar repeticiones, y regresa una estructura conjunto con sus operaciones. (Ver fig. 3.5)

Las bolsas son conjuntos con repetición de elementos, por lo que para implementarlas, no se da una signatura, lo que le permite heredar todo de la subestructura de conjunto que importa. En el functor *Bolsa*, el parámetro es una estructura satisfaciendo a *CONJUNTO*, de la cual se modifican las operaciones *remover*, *diferencia* Y *tamaño*, para permitir repeticiones. (Ver fig. 3.5)

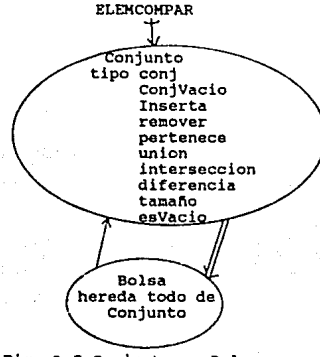


Fig. 3.5 Conjuntos y Bolsas.

3.2.5 Gráficas.

Las gráficas tienen como parámetro a *Nodo*, que es una estructura que satisface a *ELEMCOMPAR*. En la signatura se define el tipo con las constructoras *GrafVacia*, *Insertanodo* e *Insertaarista*. Además se da la signatura de las operaciones auxiliares. En el functor *Grafica* están las funciones que las implementan según los axiomas. (ver fig. 3.6).

A partir de las gráficas se pueden definir las digráficas que son aquellas cuyas aristas tienen dirección, por lo que se implementan por medio de un functor que toma una gráfica y regresa una digráfica en la que se oculta la operacion *adyacente* y en su lugar se exportan *essucesor* y *espredecesor*. (Ver fig. 3.6).

Fig 3.5. Gráfica y Digráfica.

3.2.6. Tablas *Hash*.

La estructura de la implementación de las tablas *hash* está dada en las figuras 3.6 y 3.7.

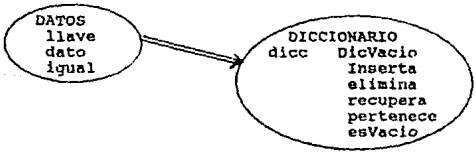


Figura 3.6 Diccionario.

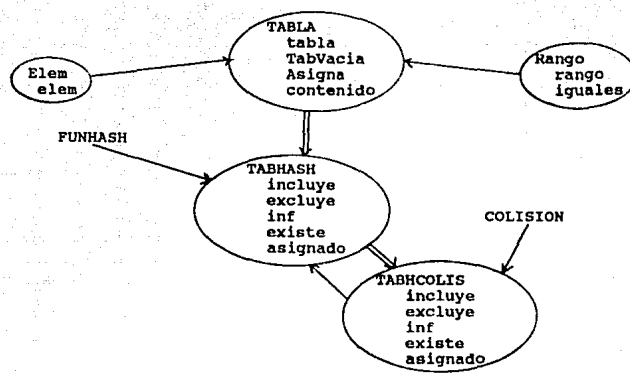


Figura 3.6. Tabla "hash".

Como se ve en la figura 3.6, se define una signatura *DATOS*, que contiene a los tipos *llave, dato* y una función de igualdad entre llaves, que es el parámetro para el functor *Dicc*, en el que se define el tipo *dicc* con las constructoras *DicVacio* e *Inserta* y las operaciones *elimina, recupera, pertenece* y *esVacio*. Este functor es útil por sí mismo, pero, como se indicó en el capítulo 1, es un auxiliar para implementar a las tablas *hash*.

Para las tablas, se requiere de un parámetro *Rango* y otro *Elem*, contienen a los tipos *rango* y *elem* así como una igualdad entre rangos, se dan las signaturas necesarias según la especificación y para las cuales se pueden construir diversas estructuras al momento de usarlas, pero no existen en la biblioteca.

Por medio de la signatura *TABLA* y el functor *Tabla* se define al tipo *tabla* con las constructoras *TabVacia* Y *Asigna* y la observadora *contenido*.

Se da la signatura *FUNHASH* que tiene una estructura de signatura *DATOS*, otra de signatura *RANGO* y una función *hash* o de dispersión, la cual puede implementarse al escribir la estructura correspondiente, según las que se pueden encontrar en la literatura.

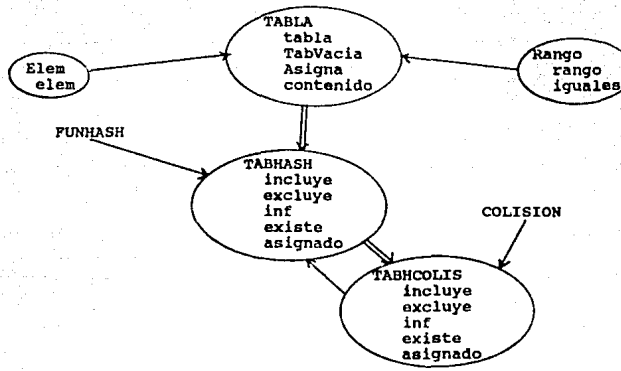


Figura 3.6. Tabla "hash".

Como se ve en la figura 3.6, se define una signatura *DATOS*, que contiene a los tipos *llave*, *dato* y una función de igualdad entre llaves, que es el parámetro para el functor *Dicc*, en el que se define el tipo *dicc* con las constructoras *DicVacio* e *Inserta* y las operaciones *elimina*, *recupera*, *pertenece* y *esVacio*. Este functor es útil por sí mismo, pero, como se indicó en el capítulo 1, es un auxiliar para implementar a las tablas *hash*.

Para las tablas, se requiere de un parámetro *Rango* y otro *Elem*, contienen a los tipos *rango* y *elem* así como una igualdad entre rangos, se dan las signaturas necesarias según la especificación y para las cuales se pueden construir diversas estructuras al momento de usarlas, pero no existen en la biblioteca.

Por medio de la signatura *TABLA* y el functor *Tabla* se define al tipo *tabla* con las constructoras *TabVacia* Y *Asigna* y la observadora *contenido*.

Se da la signatura *FUNHASH* que tiene una estructura de signatura *DATOS*, otra de signatura *RANGO* y una función *hash* o de dispersión, la cual puede implementarse al escribir la estructura correspondiente, según las que se pueden encontrar en la literatura.

Para las tabhash en sí, se da la signatura y el functor que corresponden a la especificación.

Por último para el manejo de colisiones se requiere además de la función *hash*, una operación para las colisiones la cual tiene su dominio y codominio definido en la signatura *COLISION*, y que es parámetro del functor *TabHColis*, que extiende a las tablas *hash* para manejar las colisiones.

3.3 Organización y uso de la biblioteca.

En el apéndice B se presenta un diccionario con el nombre del archivo en el que se encuentra cada tipo de datos y el nombre exacto del functor que lo genera, así como la signatura de la estructura que requiere el parámetro, y la signatura que describe lo que exporta.

En ese mismo apéndice se tiene un diccionario de signaturas en el que se encuentra el nombre de cada una de las que hay en la biblioteca, sus componentes tanto tipos como funciones en las que las constructoras del tipo tienen el nombre con la inicial mayúscula. Es importante consultar este diccionario para saber que exporta un functor.

En el caso de los functores que no tienen signatura asignada, exportan todo incluyendo lo importado de la estructura parámetro. Tal es el caso por ejemplo, de las *Bolsas* que importa todo de *Conjunto* y sólo redefine algunas operaciones para permitir repeticiones.

En el apéndice C se presentan todos los archivos que componen la biblioteca, cada uno con signaturas y functores. Para hacer uso de los módulos de la biblioteca se puede proceder de la siguiente manera:

- En el momento en que se quiere usar un tipo de datos se escribe la instrucción *use* ["*nombrearchivo.sml*"]; (consultarlo en el apéndice B) para incorporar el archivo de ese nombre que contiene al functor requerido.

- Se consulta el apéndice B para saber de que otros tipos de datos depende y llamar a esos otros archivos también. En general, se requiere a las estructuras básicas de caracteres, naturales, reales, etc. por lo que se debe llamar al archivo *elementos.sml*.

-Cuando se quiere poner nombre a una estructura generada por un functor, se recurre a una definición de la forma

```
structure <nombre> = <functor con sus parámetros>;
```

ejemplo:

```
structure Pilanat = Pila ( Natural);
```

Para introducir los valores que debe contener la estructura se hace uso de las constructoras del tipo y se declara como valor. Ejemplo:

```
val Pilanat = pilanat.Mete(2,pilanat.Mete(3,pilanat.PilaVacia));
```

en donde se pone el valor 3 en la pila vacia y luego se mete el valor 2 en el tope de *Pilanat*.

De esta forma se esta listo para usar una estructura con valores concretos por medio de todas las funciones que se ofrecen para el tipo de datos.

CAPITULO III.

Uso de las bibliotecas.

1. Metodología para el uso de las bibliotecas.

El desarrollo de un programa consta de varias fases como son:

- *Especificación del problema* : En esta fase se hace una descripción precisa, clara y concisa de los requerimientos. Es un medio de comunicación entre quien tiene un problema y quien desarrolla la solución por medio de un programa. Usualmente se hace en lenguaje natural, pero como se vió en el capítulo I, existen buenas razones para hacerla por medio de especificaciones formales, ya que éstas tienen la ventaja de que es posible introducirlas en la computadora a través de un lenguaje de especificación formal, y obtener un prototipo de la implementación del programa.

- *Programación* : En este caso se trata de escribir código en algún lenguaje de programación que satisfaga la especificación. Es deseable que al escribir un programa se cuente con herramientas que faciliten la tarea. Esto puede lograrse por medio de módulos reusables o con bibliotecas confiables tanto de tipos de datos como de funciones de uso común.

Hay una gran variedad de métodos de trabajo para la programación, que se pueden seguir en forma individual o combinados, como son :

La programación ascendente que consiste en ir construyendo programas a partir de datos o funciones simples ya preconstruidas hasta obtener componentes mas complejos. La programación descendente que es descomponer problemas complejos en otros cada vez mas simples. Crecimiento estructurado cuyo objetivo es desarrollar las partes mas importantes primero y luego agregar otras características. La derivación de programas que es construir una especificación que cada vez se va bajando en el nivel de abstracción hasta tener código para un cierto lenguaje de programación.

-*Verificación y prueba*: consiste en corroborar que un programa cumple con su especificación. Se pueden hacer dos tipos de pruebas:

La verificación formal o matemática que es bastante tediosa y complicada pero que asegura una mayor confiabilidad. Y la prueba por casos que es más simple pero menos segura.

Partiendo de estas fases se propone a continuación una metodología que permita construir un programa confiable en ML apoyandose en las dos bibliotecas desarrolladas en este trabajo.

(Abbott 83) propone una metodología interesante como punto de partida para lo que aquí se pretende, por lo que se hace una pequeña revisión de ella. Consiste en partir de una descripción informal del problema a resolver. De ella se define una estrategia informal para la solución del problema. Esta estrategia debe estar descrita en el mismo nivel de abstracción del problema, esto es en términos del dominio del problema.

El segundo paso consiste en formalizar dicha estrategia, lo que se logra en tres pasos:

1. Identificar los tipos de datos que se requieren para la solución. Esto se logra identificando en la estrategia informal, todos los sustantivos comunes los cuales sugieren los tipos de datos.

2. Identificar los objetos que corresponden a dichos tipos de datos. Los sustantivos propios son los candidatos a ser objetos.

3. Se definen los operadores o funciones que se aplican a los objetos. Un verbo, atributo, predicado o expresión descriptiva sugiere a un operador.

4. Organizar los operadores en la estructura de control sugerida por la estrategia informal.

Por otra parte para construir un sistema o programa complejo en ML (Harper 86) propone construir una colección jerárquica de estructuras interrelacionadas en la que la dinámica para la construcción de programas es el uso de functores, que permiten componer programas a partir de unidades independientes que se pueden probar y compilar por separado. Propone seguir las siguientes pautas:

1. Describir las signaturas de las estructuras a desarrollar sin preocuparse de algoritmos y representaciones. Según (MacQueen 85), un buen estilo de programación en ML es mantener las signaturas del menor tamaño posible.

2. Enseguida se escriben las estructuras que corresponden a las signaturas anteriores, lo que significa que es aquí donde se entra al detalle del comportamiento de las funciones. Estas estructuras deben compilarse y probarse por separado. Por lo que se organiza el sistema como un conjunto de functores, cada uno con sus dependientes como argumentos.

3. Para armar el sistema a partir de los functores, simplemente se les invoca. ML tiene la posibilidad de poner los functores con sus signaturas en un archivo al que se llama por la función *use* donde se requiera.

Basada en estas dos metodologías, y apoyándose en las bibliotecas se propone el siguiente procedimiento para la construcción de programas:

METODOLOGIA:

1. Hacer una descripción informal del problema y buscar una estrategia para su solución.

2. En dicha estrategia se identifican los tipos de datos que se requieren por medio del análisis de los sustantivos así como las operaciones a efectuar en ellos.

3. Una vez identificados los tipos de datos abstractos se tienen dos posibilidades: si algunos se encuentran en la biblioteca de especificaciones algebraicas (consultar apéndice A), se hace la especificación concreta con la instanciación de los parámetros y si hace falta el renombre de tipo u operaciones.

Si no está en la biblioteca, se hace la especificación algebraica, tratando de usar algo de lo ya especificado por medio de importaciones, exportaciones y otros mecanismos de composición de especificaciones.

4. Construir la especificación algebraica de las operaciones identificadas en la estrategia informal, poniendo atención a que tipos de datos se tienen que importar como parámetros y que se debe exportar.

5. Partiendo de todas las especificaciones algebraicas desarrolladas, construir las signaturas de las estructuras que se generarán y que correspondan a las especificaciones tanto de los tipos de datos como de las operaciones. Dichas signaturas deben incluir a los tipos, subestructuras y signaturas de las funciones que exportan. Si algún tipo de datos debe importar todo de otro y solo redefine operaciones, entonces no es necesario escribir la signatura pues es suficiente con ponerla en el parámetro del functor correspondiente.

6. Construir los functores que tienen como parámetros a las estructuras de las dependen en la jerarquía, y el detalle del comportamiento de las funciones. Un functor puede no tener parámetros (en cuyo caso se pone solo "()"), además puede no requerir ocultar nada de lo definido en él, es ese caso puede no asignarsele signatura alguna y ML le asigna la más general posible.

7. Cada functor junto con su signatura (si la tiene) se coloca en un archivo con extensión *.sml*. Si hace uso de la biblioteca en ML (ver apendice C) se debe indicar la inclusión del archivo correspondiente por medio de la función *use ["nombrearch.sml"];* (los nombres de los archivos se pueden consultar en el apéndice B). De la misma manera se pueden incorporar otros archivos generados para la aplicación en particular.

8. Por último, para unir el sistema se invocan los functores tanto los que se requieran de la biblioteca como los desarrollados para la aplicación. Hay que tener cuidado en mantener el orden de las dependencias, primero se llaman a los functores que no requieran nada o a módulos de la biblioteca y se va construyendo en forma ascendente el sistema.

Por suspuesto que como toda metodología, ésta no es infalible ni aplicable en forma "automática" a todos los problemas, sino que es una pauta de apoyo en el desarrollo de sistemas en ML, donde algo muy importante sigue siendo la creatividad e intuición del programador.

Una línea de investigación en la dirección de tener herramientas que permitan obtener programas en forma automática a partir de especificaciones, y eluso de bibliotecas es la creación de sistemas capaces de ayudar a recorrerlas y seleccionar los módulos necesarios para una aplicación particular. Este trabajo es parte de un proyecto de investigación encaminado a esos objetivos, dirigido por la Dra Hanna Oktaba que incluye la elaboración de bibliotecas en distintos lenguajes como Ada, Modula 2, etc, así como de un sistema para la selección de la más apropiada.

2. Ejemplo de aplicación.

Se presenta el desarrollo de una aplicación sencilla siguiendo la metodología anterior, en donde el objetivo central es desarrollar un programa en ML usando varios de los tipos de datos de las bibliotecas y no tanto resolverlo de la manera más general y eficiente posible.

1. La descripción informal del problema es : Hacer un programa en ML que evalue expresiones aritméticas. Se hacen restricciones al planteamiento del problema real como que los operandos son sólo de un dígito entero y se tienen las operaciones binarias +, -, *, /. pero si se permite el uso de paréntesis en las expresiones. Asi el objetivo es dada una expresión aritmética con estas restricciones obtener su valor.

La estrategia informal para obtener la solución es:

En una expresión aritmética hay dos tipos de elementos, los *operandos* que son dígitos, y *operadores* incluyendo los paréntesis. Para poder evaluar la expresión en una computadora es necesario suprimir los paréntesis pero manteniendo la prioridad de los operadores. Por esto se requiere hacer una reescritura de la expresión que está en notación infija a notación posfija, la cual tiene primero los dos operandos y después al operador.

Una vez que se tiene la expresión de esta manera, se puede evaluar con la ayuda de una pila que guarda los operandos, para que al encontrar un operador se extraiga de la pila los dos últimos guardados y efectuar la operación indicada en ellos, así hasta tener en el tope el valor de la expresión.

2. A partir de la estrategia anterior se pueden identificar los siguientes tipos de datos y las operaciones a efectuar en ellos:

-*operandos* : son enteros de un dígito los que se deben poder identificar y tomar su valor al evaluarlos.

-*operadores* : son +, -, *, / y los paréntesis que se deben identificar, sacar su prioridad y aplicarlos a los operandos apropiados.

-una *expresión en infija* que se debe poder transformar a posfija.

-la *expresión en posfija* se recorre al evaluarla.

-una pila para la evaluación en la que se meten, sacan y consultan los valores almacenados.

Una vez identificados los tipos y sus operaciones se puede estudiar la forma de representarlos. Los operandos conviene tratarlos como los enteros del lenguaje y usar las operaciones en ellos definidos.

Para los operadores se pueden considerar elementos de un conjunto, además una función para obtener la prioridad y reglas para la evaluación.

Las expresiónes infija y posfija se pueden representar por colas de caracteres en las que se insertan y extraen en el mismo orden. Pero hacen falta funciones que transformen de infija a sufija o posfija y evaluen esta última.

3. Consultando el apéndice A se encuentran especificaciones algebraicas para todos los tipos por lo que solo se instancían apropiadamente como sigue:

Se inicia por la cola que representa a las infijas y posfijas:

Especificación de expresiones infijas:

```
INSTANCIA Colas
CON Elementos COMO Caracteres
    Elemento COMO  Car
RENOMBRA Cola COMO expi
FIN DE INSTANCIA Colas;
```

Especificación de expresiones posfijas:

```
INSTANCIA Colas
CON Elementos COMO Caracteres
    Elemento COMO  Car
RENOMBRA Colas COMO exprpos
FIN DE INSTANCIA Colas;
```

Para los operadores se tiene la especificación siguiente:

```
ESPEC operador
INSTANCIA Elementos
CON Elemento COMO Caracteres
IMPORTA TODO DE Conjunto;
     Renombra conjunto COMO operador
     Operación
          prior : caracter -> natural
AXIOMAS
     Var      c: caracter
O1: prior c =     Si c = "*" o c="/"
                  entonces 2
                  si_no Si c = "+" o c = "-"
                  entonces 1
                  si_no Si c = "("
                  entonces 0
FIN DE operador;
```

4. Dados los tipos de datos se identifican las operaciones a efectuar *conv* y *evalua* y se hace la especificación completa del problema:

```
ESPEC Evalua expresiones aritméticas.
IMPORTA INSTANCIA Colas
     CON Elementos COMO Caracteres
         Elemento COMO caracter
     Renombra cola como expi
     renombra cola como exprpos
FIN DE INSTANCIA Colas;
     Operaciones
          conv : expi -> exprpos
          evalua : exprpos -> entero
FIN DE Evalua expresiones aritméticas;
```

los algoritmos para las funciones son: Para conv recibe una expresión aritmética válida en una cola y auxiliandose de una pila de caracteres, se invierte el orden de los operadores según la prioridad, se suprimen los paréntesis según algoritmo conocido y se regresa otra cola con la expresión en posfija.

Evalua recibe en una cola tipo exprpos, una expresión posfija válida, se auxilia de una pila de enteros en la que va guardando operandos que se evaluan al tener un operador hasta que en el tope se tiene ya el valor de la expresión.

5. De los pasos anteriores se deduce que expi y exprpos son colas de caracteres como se definió en las especificaciones anteriores. Por lo que se declaran las estructuras correspondientes auxiliandose del functor *Cola* y de la estructura *Cadena* de la biblioteca:

```
structure expi = Cola (Cadena);

structure exprpos = Cola (Cadena);
```

Para la función de conversión se requiere como auxiliar, una pila de caracteres en la que se almacenan temporalmente los operadores. Para eso se declara una estructura p que es el resultado de aplicar el functor *Pila* a una estructura de caracteres

```
structure p = Pila (Cadena);
```

También se requiere de una estructura *operador* resultado de la aplicación del functor *Conjunto* con parámetro *Cadena* y en la que se introducen los operadores a considerar, usando las constructoras de conjuntos.

```
        structure operador = Conjunto (Cadena);

val operador = operador.Inserta("+",operador.Inserta ("-",
operador.Inserta ("*",operador.Inserta ("/",operador.ConjVacio))));
```

Para la evaluación se regresa el tope de la pila de enteros, por lo que es necesario declarar una estructura *pilent* resultado de invocar al functor *Pila* a la estructura *Naturales* y el resultado de la función de evaluación estará en el tope.

```
        structure Pilaent = Pila ( Natural);
```

6. A continuación se construyen las funciones auxiliares para ejecutar tanto la conversión como la evaluación puedan ver en las figuras 3.2.1 y 3.2.2.

```
(*   funciones auxiliares   *)

local
   fun esdigito c = c >= "0" andalso c <= "9";

   fun esoperador c = operador.pertenece (c, operador);
   .
   exception Prior
   fun prior c = if c = "*" orelse c = "/"
                 then 2
                 else if c = "+" orelse c = "-"
                      then 1
                      else if c = "("
                           then 0
                           else raise Prior;

   fun sacaprior (p, d, c) = if P.esvacia p
                     then (P.Mete(C.primero d, p ), c)
                     else if prior(P.tope p) >= prior (C.primero c)
                          then  sacaprior (P.saca p, d, C.Meteult
                                           (P.tope p, c))
                          else (P.Mete (C.primero d, p), c);
```

```
fun sacapila (p, c) = if P.esvacia p then p
                      else sacapila (P.saca p,C.Meteuit (P.tope p,c));
fun conv' (expi, exprp) p = if C.esvacia expi
                      then if P.esvacia p
                           then exprp
                           else conv' (C.sacaprim expi, exprp)
                                      (sacapila (p, exprp))
                      else let val primC = C.primero expi
                           in if esdigito primC
                           then conv' (C.sacaprim expi, C.Meteuit
                                       (primC, exprp)) p
                           else if primC = "("
                                then conv' (C.sacaprim expi, exprp)
                                           (P.Mete (primC, p))
                                else if esoperador primC
                                then let val (p',c') =
                                           sacaprior (p, expi, exprp)
                                     in conv' (C.sacaprim expi,c') p'
                                     end
                                     else if primC = ")"
                                          then if P.esvacia p
                                               then exprp
                                               else if P.tope p = "("
                                     then conv' (C.sacaprim expi,exprp)
                                                (P.saca p)
                                     else conv' (expi, C.Meteuit
                                          (P.tope p,exprp)) (P.saca p)
                                else conv' (C.sacaprim expi, C.Meteuit
                                           (P.tope p,exprp)) (P.saca p)
                      end
in fun conv (expi, exprp) = conv' (expi, exprp) P.Vacia
end;
```

Figura 3.2.1 Función conv en ML.

```
exception Calcula
fun calcula (x,oper,y) = if oper = "+" then y+x
                         else if oper = "-" then y-x
                         else if oper = "*" then y*x
                         else if oper = "/" then y div x
                         else raise Calcula;
fun evalua' p e = if CD.esvacia e
                  then PE.tope p
                  else let val primC = CD.primero e
                       in if esdigito primC
                          then evalua' (PE.Mete (ord (primC)-48,p))
                                       (CD.sacaprim e)
                          else let val primP = PE.tope p
                                   val segP = PE.tope (PE.saca p)
                               in evalua' (PE.Mete(calcula(primP,primC,
                                          segP), p)) (CD.sacaprim e)
                                  end
                       end;
in val eval = evalua' Pilaent.Vacia epos;
end;
```

Figura 3.2.2 Función evalua en ML.

7. A continuación se tiene el encabezado del programa con las llamadas a los archivos de la biblioteca que se requieren:

```
use["elementos.sml"];

use["estrlineales.sml"];

use["ecolas.sml"];

use["pilas.sml"];

use["conjunto.sml"];
```

8. Para construir el sistema que hace la evaluación se definen dos estructuras ent y epos ambas de tipo cola de cadenas, en la primera se introduce la expresión a convertir (como ejemplo se tiene la expresión (2+3) * 5) por medio de las constructoras de colas que son ColaVacia y Meteult. En la segunda se obtendrá de la conversión y que sirve de entrada a la evaluación. En la figura 3.2.3 se encuentra el programa que llama los archivos necesarios, genera las estrucuras y la invocación a las funciones con los parámetros adecuado.

```
(*   estructura con los datos de entrada  *)
structure ent = Cola(EstrCola(Estrlineal (Cadena)));
(*  aqui se meten los datos de prueba (2+3)*5   *)
val ent = ent.Meteult ("5",ent.Meteult ("*",ent.Meteult (")",
          ent.Meteult ("3",ent.Meteult ("+",ent.Meteult ("2",
          ent.Meteult ("(",ent.ColaVacia)))))));

   structure exprp = Cola (Cadena);

(*  structuras auxiliares   *)
   structure P = Pila (Cadena);
   structure C = Cola (Cadena);
   structure operador = Conjunto (Cadena);

(*  se inicializan las estructuras  *)
val operador = operador.Inserta("+",operador.Inserta ("-",
 operador.Inserta ("*", operador.Inserta ("/",operador.ConjVacio))));


   structure ent = Cola (Cadena);
   val ent = ent.Meteult("5", ent.Meteult ("*", ent.Meteult (")",
                  ent.Meteult ("3", ent.Meteult ("+", ent.Meteult ("2",
                  ent.Meteult ("(",ent.ColaVacia)))))));

   structure epos = Cola (Cadena);


(*  llamada a la función conv con los argumentos necesarios   *)
     conv (ent,epos.ColaVacia);
```

```
(*   se evalua una expresion en posfija    *)
   structure Pilaent = Pila ( Natural);
   structure epos = Cola (Cadena);

(*   inicializacion de estructuras    *)
(*  aqui va la función evalua de la figura 3.2.2    *)
(*  se llama la función de evaluación     *)
     eval;
```

Figura 3.2.3 Programa de evaluación de expresiones aritmáticas.

CONCLUSIONES.

Construir la especificación algebraica de un sistema grande o complejo no es tarea fácil. Por eso es importante el concepto de dividir el trabajo en módulos más fáciles de especificar. Pero aun así, tener que llegar a niveles elementales, como la especificación de hasta las estructuras de datos más usuales, cada vez que se desarrolla un sistema es como "inventar el agua tibia".

Es por esto que es útil contar con una biblioteca de especificaciones que permitan ayudar a que a partir de ella se construyan nuevos tipos de datos abstractos mas complejos. Es en ese sentido que la biblioteca de especificaciones algebraicas aquí presentada es un buen aporte.

La biblioteca de especificaciones que se presenta en este trabajo, como se dijo anteriormente, tiene un nivel de abstracción alto lo que permite definir las propiedades de las estructuras de datos usuales sin pensar en los problemas de implementación alguna.

Otro aspecto importante es que dado que en general, el parámetro es un género, se hace posible pensar que se puede instanciar por cualquiera de los tipos especificados. Así es posible tener pilas de listas, árboles de conjuntos o colas de colas de cadenas, etc.

Al hacer una revisión bibliográfica se encontró que hay muchas bibliotecas de tipos de datos abstractos pero, en general, muy ligadas a implementaciones. Por ejemplo en la colección de cuatro libros de Lins (Lins 90) se presenta una biblioteca muy completa de implementaciones en Modula-2 de tipos de datos abstractos en la que se da su especificación en una adaptación del lenguaje desarrollado por Guttag y Liskov llamado Larch, así como los módulos en Modula-2.

Sin embargo, estas especificaciones no están parametrizadas, lo que ocasiona perdida en el nivel de abstracción.

Al revisar la estructura de las clases en Smalltalk/V, hay dos clases que comprenden varios de los tipos presentados en este trabajo, la clase de las Colecciones y la de Magnitudes. En la primera se encuentran estructuras ordenadas y sin orden, con y sin índice y que permiten o no repeticiones de sus elementos. Entre estas se tiene a las bolsas, conjuntos, pilas, colas, etc. con una jerarquización diferente a la presentada aquí, debido particularmente a que en Smalltalk el criterio de indexación es preponderante.

En conclusión, se puede decir que la biblioteca de especificaciones algebraicas aquí presentada tiene la ventaja de la parametrización, un alto nivel de abstracción y jerarquización que permite servir de base para implementaciones en diversos lenguajes de programación.

Por otro lado la biblioteca de implementaciones algebraicas en ML tiene dos ventajas. Por un lado, la biblioteca en sí que facilita la construcción de sistemas en ML. Además el hecho de mostrar como se pueden construir signaturas y functores a partir de especificaciones algebraicas, lo que es fácil por el hecho de que al diseñar ML se buscó aplicar las ideas de este tipo de especificaciones en el ámbito de los lenguajes de programación.(Sannella y Tarlecki 84).

Sin embargo, esos mismo autores dicen que la versión actual de ML, no es posible construir un sistema partiendo de la especificación y trabajando formalmente a través de refinamientos sucesivos, obtener código en ML verificado. Ellos se encuentran trabajando en esa línea por medio de Extended ML que pretende lograr esos objetivos.

Así pues la metodología aquí presentada para construir sistemas, partiendo de especificaciones algebraicas y con auxilio de las bibliotecas es un avance al facilitar la tarea del programador de ML.

La verificación de que las implementaciones cumplen con las especificaciones que es uno de los siguientes problemas a atacar.

APENDICE A

Biblioteca de Especificaciones Algebraicas.

0. Especificaciones para parámetros.

```
0.1 ESPEC  Elementos;
EXPORTA  Elemento;
FIN DE Elementos;
```

```
0.2 ESPEC  Elemcompar;

IMPORTA TODO DE Elementos, Booleanos;
EXPORTA  Elemento, "=";

OPERACIONES
          "="   : Elemento , Elemento -> Bool

AXIOMAS
     VAR   e,f :Elemento
EC1: "=" (e,f) = e "igual" f
FIN DE Elemcompar;
```

```
0.3 ESPEC  Elemorden;

IMPORTA TODO DE Elemcompar, Booleanos;
EXPORTA  Elemento, "=", "<";

OPERACIONES
          "<"  : Elemento , Elemento -> Bool

AXIOMAS
     VAR        e,f:Elemento
EO1: "<" (e,f) = e "menor" f
FIN DE Elemorden;
```

```
1.1 ESPEC  Booleanos;

EXPORTA  Bool, cierto, falso, y, o;

     GENERO  Bool

     OPERACIONES
          cierto     : -> Bool
          falso      : -> Bool
          y          : Bool, Bool -> Bool
          o          : Bool, Bool -> Bool

AXIOMAS
B1:  y(cierto,falso) = falso
B2:  y(falso,cierto) = falso
B3:  y(cierto,cierto) = cierto
B4:  y(falso,falso)  = falso
B5:  o(cierto,falso) = cierto
B6:  o(falso,cierto) = cierto
B7:  o(cierto,cierto) = cierto
B8:  o(falso,falso)  = falso
FIN DE Booleanos;
```

```
1.2 ESPEC  Naturales;

EXPORTA  Nat, cero, succ, suma;

     GENERO Nat

     OPERACIONES
          cero       : -> Nat
          succ       : Nat -> Nat
          "+"       : Nat, Nat -> Nat


AXIOMAS
     VAR  x,y : Nat
N1:  "+" (x,cero) = x
N2:  "+" (x,succ(y)) = succ ("+" (x,y))
FIN DE Naturales;
```

```
1.3 ESPEC  Caracteres;

EXPORTA  Car, 'a'..'Z', '0'..'9',...;

     GENERO Car

     OPERACIONES
          'a' .. 'z', 'A' .. 'Z' : -> Car
          '0' .. '9' : -> Car
          ' ', ...: -> Car
FIN DE Caracteres;
```

2. Estructuras Lineales.

```
2.1 ESPEC  Estructuras Lineales;

PARAMETRO  Elementos
     Género  Elemento
FIN DE  Elementos;

IMPORTA TODO DE Booleanos;
EXPORTA  Estrlineal, estrvacia, inserta, inicial, resto, esvacia;

     GENERO Estrlineal

     OPERACIONES
          estrvacia : -> Estrlineal
          inserta   : Elemento, Estrlineal -> Estrlineal
          inicial   : Estrlineal -> Elemento
          resto     : Estrlineal -> Estrlineal
          esvacia   : Estrlineal -> Bool

AXIOMAS
     VAR    e: Elemento       l,m : Estrlineal
E1:  inicial ( estrvacia ) = error
E2:  inicial ( inserta ( e,l )) = e
E3:  resto ( estrvacia ) = error
E4:  resto ( inserta (e,l )) = l
E5:  esvacia (estrvacia ) = cierto
E6:  esvacia ( inserta (e,l )) = falso
FIN DE Estructuras Lineales;
```

```
2.2 ESPEC  Listas;

IMPORTA TODO DE Estructuras Lineales, Booleanos, Naturales;
     Renombra Estrlineal como Lista
              estrvacia  como listavacia

EXPORTA  Lista, listavacia, inserta, inicial, resto, longitud,
enésimo, esvacia;

     OPERACIONES
          enésimo        : Nat, Lista -> Elemento
          longitud       : Lista -> Nat
          concatena      : Lista, Lista -> Lista

AXIOMAS
     VAR    e: Elemento       l, l1 : Lista      n:Nat
L1:  enésimo (n, listavacia) = error
L2:  enésimo (n, inserta (e,l)) = Si n=0 entonces error
                                  si_no Si n=1 entonces inicial(l)
                                        si_no enésimo(n-1,resto(l))
L3:  longitud ( listavacia )) = 0
L4:  longitud ( inserta (e,l )) = 1 + longitud (l)
L5:  concatena (listavacia, l) = l
L6:  concatena (inserta (e,l),l1) = inserta (e, concatena (l,l1))
FIN DE   Listas;
```

```
2.3 ESPEC  Pilas;
IMPORTA TODO DE Estructuras Lineales;
     Renombra
          Estrlineal como Pila
          estrvacia  como   pilavacia
          inserta    como  mete
          resto      como   saca
          inicial    como  tope

EXPORTA  Pila, pilavacia, mete, saca, tope, esvacia;
FIN DE  Pilas;
```

```
2.4 ESPEC  Pila acotada;

PARAMETRO  Cota
     CONST  max : Nat
FIN DE  Cota;

IMPORTA TODO DE Pilas, Booleanos, Naturales;

EXPORTA    Pila, pilavacia, mete, saca, tope, esvacia, tamaño
estallena;

     OPERACIONES
          tamaño     : Pila -> Nat
          estallena : Pila -> Bool

AXIOMAS
     VAR    e: Elemento       p: Pila
PA1:  tamaño ( pilavacia ) = 0
PA2:  tamaño ( mete ( e,p )) = 1 + tamaño ( p )
PA3:  estallena ( p ) = Si tamaño ( p ) = Max
                        entonces  cierto
                        si_no     falso
PA4:  mete ( e,p ) = Si estallena (p )
                     entonces     error
                     si_no mete ( e,p )
FIN DE  Pila acotada;
```

```
2.5 ESPEC  Estr Colas;

IMPORTA TODO DE Estructuras Lineales, Booleanos;
     Renombra  Estrlineales  como  Cola
               estrvacia  como  colavacia
               inserta    como  meteult
               inicial    como  último
               resto      como  sacault

EXPORTA   Cola, colavacia, meteult, sacault, último, esvacia,
meteprim, primero, sacaprim;

    OPERACIONES
          meteprim  : Elemento , Cola  -->  Cola
          primero   : Cola -> Elemento
          sacaprim  : Cola -> Cola

AXIOMAS
     VAR     e:Elemento          c: Cola
EC1: primero (colavacia ) = error
EC2: primero ( meteprim (e,c)) = e
EC3: primero ( meteult (e,c)) = Si esvacia (c) entonces e
                                si_no primero (c)
EC4: sacaprim (colavacia ) = error
EC5: sacaprim (meteprim (e,c)) = c
EC6: sacaprim ( meteult (e,c)) = Si esvacia (c) entonces colavacia
                                 si_no meteult (e,sacaprim (c))
FIN DE Estr Colas;
```

```
2.6 ESPEC  Colas;

IMPORTA TODO DE Estr Colas, Booleanos

EXPORTA  Cola, colavacia, meteult, primero, sacaprim, esvacia;

FIN DE Colas;
```

```
2.7 ESPEC  Cola acotada;

PARAMETRO  Cota
     CONST  Max : Nat
FIN DE  Cota;

IMPORTA TODO DE Colas;

EXPORTA   Cola, colavacia, meteult, primero, sacaprim, esvacia,
tamaño, estallena;

     OPERACIONES
          tamaño     : Cola -> Nat
          estallena : Cola -> Bool

AXIOMAS
     VAR     e: Elemento        c: Cola
CA1:  tamaño ( colavacia ) = 0
CA2:  tamaño ( meteult ( e,c )) = 1 + tamaño ( c )
CA3:  estallena ( c ) = Si tamaño ( c ) = Max
                        entonces  cierto
                        si_no     falso
CA4: meteult ( e,c ) = Si estallena ( c )
                        entonces  error
                        si_no     meteult ( e,c )
FIN DE  Cola acotada;
```

```
2.8 ESPEC  Cola prior;

PERAMETRO Elemorden
     (* Se refiere a la especificación de Elemorden que aparece en
las especificaciones para parámetros al principio de la biblioteca
*)
FIN DE Elemorden;

IMPORTA TODO DE Cola;

EXPORTA  Cola, colavacia, meteult, primero, sacaprim, esvacia,
maximo;

     OPERACIONES
          maximo : Cola -> Elemento

AXIOMAS
     VAR  e:Elemento    c:Cola
CP1:  sacaprim ( meteult (e,c )) = Si esvacia ( c )
                                   entonces  colavacia
                                   si_no Si maximo (c) < e
                                      entonces  c
                                      si_no meteult (e,sacaprim(c))
CP2:  maximo ( colavacia ) = error
CP3:  maximo ( meteult (e,c )) = Si esvacia (c) o maximo (c) < e
                                 entonces  e
                                 si_no  maximo (c)
FIN DE  Cola-prior;
```

```
2.9 ESPEC  Cola doble;

IMPORTA TODO DE Estr colas;

EXPORTA     Cola,  colavacia,  meteult,  meteprim,  sacault,
sacaprim, primero, último,  esvacia;


FIN DE  Cola-doble;
```

3. Arboles Binarios.

3.1 ESPEC Arboles binarios;

```
PARAMETRO    Elemorden
     (* Ver la especificación de Elemorden en las especificaciones
de lo parámetros *)
FIN DE  Elemorden;

IMPORTA TODO DE Booleanos, Naturales;

EXPORTA  Arbolbin, árbolvacio, creaárbol, izq, der, raiz, altura,
esvacio;

     GENERO Arbolbin

     OPERACIONES
          árbolvacio     : -> Arbolbin
          creaárbol    :Arbolbin,Elemento,Arbolbin -> Arbolbin
          izq             : Arbolbin -> Arbolbin
          der             : Arbolbin -> Arbolbin
          raiz            : Arbolbin -> Elemento
          altura          : Arbolbin -> Nat
          esvacio        : Arbolbin -> Bool

AXIOMAS
     VAR     e: Elemento          ai,ad : Arbolbin
AB1: izq (árbolvacio ) = error
AB2: izq (creaárbol (ai,e,ad )) = ai
AB3: der ( árbolvacio ) = error
AB4: der (creaárbol (ai,e,ad )) = ad
AB5: raiz (árbolvacio ) = error
AB6: raiz (creaárbol (ai,e,ad )) = e
AB7: altura (árbolvacio ) = 0
AB8: altura (creaárbol(ai,e,ad)) =1 + max(altura(ai),altura(ad))
AB9:  esvacio (árbolvacio ) = cierto
AB10:esvacio ( creaárbol (ai,e,ad )) = falso

FIN DE  Arboles binarios;
```

```
3.2 ESPEC  Extiende árboles;

IMPORTA TODO DE  Arboles binarios;

EXPORTA  Arbolbin, árbolvacio, creárbol, izq, der, raiz, altura,
numnodos, esvacio, eshoja, numhojas, estalleno;

     OPERACIONES
          eshoja    :Arbolbin -> Bool
          numhojas : Arbolbin -> Nat
          nummodos       : Arbolbin -> Nat
          estalleno : Arbolbin -> Bool

AXIOMAS
     VAR   e:Elemento       ai,ad: Arbolbin
EA1: eshoja ( árbolvacio ) = falso
EA2: eshoja (creaárbol(ai,e,ad))= esvacio(ai) y esvacio(ad)
EA3: numhojas (árbolvacio ) = 0
EA4: numhojas(creaárbol(ai,e,ad))=Si eshoja(creaárbol(ai,e,ad)
                                     entonces  1
                                  si_no numhojas(ai) + numhojas (ad)
EA5: numnodos (árbolvacio ) = 0
EA6:numnodos (creaárbol(ai,e,ad))=1+numnodos(ai)+numnodos(ad)
EA7: estalleno (árbolvacio) = cierto
EA8: estalleno (creaárbol (ai,e,ad)) = Si altura(ai) = altura(ad)
                                       y estalleno(ai) y estalleno(ad)
                                       entonces cierto
                                       si_no falso
FIN DE  Extiende árboles;
```

3.3 ESPEC Arboles piramidales;

IMPORTA TODO DE Extiende árboles;

EXPORTA Arbolbin, árbol-vacio, inserta, sacamax, max, esvacio;

```
     OPERACIONES
          inserta          : Elemento, Arbolbin -> Arbolbin
          sacamax          : Arbolbin -> Arbolbin
          max              : Arbolbin -> Elemento

          inscomp          : Elemento, Arbolbin -> Arbolbin
          hazheap          : Arbolbin, Elemento, Arbolbin -> Arbolbin
          creaheap         : Arbolbin -> Arbolbin
          ult              : Arbolbin -> Elemento
          eliminault       : Arbolbin -> Arbolbin

AXIOMAS
     VAR        e,e1 : Elementos       a,ai,ad : Arbolbin


H1:  inserta (e,a) = creaheap(inscomp(e,a))

H2:  sacamax (a) = Si esvacio (a) entonces error
                   si_no hazheap( izq(eliminault(a)),ult(a),
                                  der(eliminault(a)))

H3:  max (árbolvacio) = error

H4:  max (creaárbol (ai,e,ad)) = e

H5:inscomp(e,árbolvacio)=creaárbol(árbolvacio,e,árbolvacio)

H6:inscomp (e,creaárbol(ai,e1,ad)) =
                Si ( altura(ai) = altura (ad) +1
                  y (estalleno(ai)) or ((altura(ai)=altura(ad))
                   y Not ( estalleno (ad) ) entonces
                      creaárbol(ai,e1,inscomp(e,ad))
                si_no creaárbol (inscomp(e,ai),e1, ad)

H7:  hazheap (árbolvacio,e,árbolvacio)=creaárbol(árbolvacio,
                                        e, árbolvacio)

H8:  hazheap (ai,e,árbolvacio) = Si not e < raiz (ai) entonces
                                 creaárbol(ai,e,árbolvacio)
     si_no
     creaárbol(creaárbol(árbolvacio,e,árbolvacio),raiz(ai),
               árbolvacio)
```

```
H9:  hazheap(ai,e,ad) = Si raiz(ai) < e y raiz(ad) < e
          entonces creaárbol(ai,e,ad)
          si_no
            Si not raiz(ai) < raiz(ad)
            entonces creaárbol(hazheap(izq(ai),e,der(ai)),
                               raiz(ai),ad)
            si_no creaárbol(ai,raiz(ad),
                            hazheap(izaq(ad),e,der(ad))


H10: creaheap (árbolvacio) = árbolvacio

H11: creaheap (creaárbol(ai,e,ad)) = Si altura(ai)altura(ad)<=1
              entonces hazheap(creaheap(ai), e, creaheap(ad))
             si_no error

H12: ult (árbolvacio) = error

H13: ult (creaárbol(ai,e,ad)) = Si esvacio (ai) entonces e
                                si_no
                                   Si altura(ai) > altura(ad)
                                   entonces ult(ai)
                                   si_no ult(ad)

FIN DE Arboles piramidales;
```

```
3.4 ESPEC  Arboles AVL;

IMPORTA TODO DE Arboles binarios;

EXPORTA  Arbolbin, árbolvacio, inserta, sustrae, estabalanceado;

     OPERACIONES
          inserta    : Elemento, Arbolbin -> Arbolbin
          sustrae    : Elemento, Arbolbin -> Arbolbin
          rotaizq  : Arbolbin -> Arbolbin
          rotader  : Arbolbin -> Arbolbin
          estabalanceado: Arbolbin -> Bool


AXIOMAS
     VAR  e,e1 : Elemento          ai,ad :Arbolbin

AA1: inserta(e,árbolvacio)= creaárbol(árbolvacio,e,árbolvacio)
AA2: inserta (e, creaárbol(ai,e1,ad)) =
      Si e < e1 entonces
        Si abs( altura(ai) - altura (ad) ) = 1 entonces
           Si estabalanceado (ai) entonces
                 rotaizq(creaárbol(ai,e,inserta(e1,ad)))
           si_no rotaizq(rotader(creaárbol(inserta(e,ai),e1,ad))
        si_no  creaárbol(inserta(e,ai),e1,ad)

      si_no Si e1 < e  entonces
         Si abs( altura(ai) - altura(ad) ) = 1 entonces
           Si estabalanceado( ad ) entonces
                 rotader(creaárbol(ai,e,inserta(e1,ad)))
           si_no rotader(rotaizq(creaárbol(ai,e1,inserta(e,ad)))
         si_no creaárbol(ai,e1,inserta(e,ad))


AA3: sustrae ( e, árbolvacio ) = error

AA4: sustrae ( e, creaárbol (ai,e,ad)) =
      Si e < e1 entonces
           creaárbol (sustrae( e,ai), e1,ad)
           Si Not ( estabalanceado (ai) entonces
                 Si (altura(ai) - altura(ad) ) >= 0 entonces
                    rotader (creaárbol( ai,ei,ad ))
                 si_no rotader(rotaizq(creárbol(ai,e1,ad)))

      Si e1 < e entonces
           creaárbol (ai,e1,sustrae(e,ad))
           Si Not (estabalanceado( ad)) entonces
                 Si ( altura(ad) - altura(ai) ) >= 0 entonces
                    rotaizq (creaárbol(ai,e1,ad))
                 si_no rotader (creaárbol (ai,e1,ad))
```

```
AA5: rotaizq ( árbolvacio ) = árbolvacio

AA6: rotaizq (creaárbol(ai,e,ad)) = creaárol(izq(ai),raiz(ai),
                                    creaárbol(der(ai),e,ad))

AA7: rotader (árbolvacio ) = árbolvacio

AA8: rotader (creaárbol(ai,e,ad))=
          creaárbol (creaárbol(ai,e,izq(ad)),raiz(ad),der(ad) )

AA9: estabalanceado (árbolvacio ) = cierto

AA10: estabalanceado ( creaárbol (ai,e,ad )) =
                  Si altura (ai) = altura (ad) entonces  cierto
                  si_no falso

FIN DE Arboles AVL;
```

```
3.5 ESPEC  Arboles busqueda;

IMPORTA TODO DE Arboles binarios;

EXPORTA  Arbolbin, árbolvacio, inserta, sustrae, miembro, min;

     OPERACIONES
          inserta    : Elemento, Arbolbin -> Arbolbin
          sustrae    : Elemento, Arbolbin -> Arbolbin
          miembro    : Elemento, Arbolbin -> Bool
          min        : Arbolbin -> Elemento


AXIOMAS
     VAR   e,e1 : Elemento        ai,ad : Arbolbin

AB1: inserta(e,árbolvacio)= creaarbol(árbolvacio,e,árbolvacio)

AB2: inserta (e,creaárbol(ai,e1,ad )) = Si e < e1
                              entonces creaárbol(inserta (e,ai),e1,ad)
       si_no   Si e1 < e entonces creaárbol (ai,e1,inserta (e,ad))
                           si_no  creaárbol (ai,e1,ad)

AB3: sustrae(e,árbolvacio) = error

AB4: sustrae (e,creaárbol (ai,e1,ad)) = Si e < e1
      entonces creaárbol (sustrae (e,ai),e1,ad)
         si_no  Si e1 < e entonces creaárbol(ai,e1,sustrae (e,ad))
             si_no Si esvacio(ai) entonces ad
                 si_no Si esvacio (ad) entonces ai
                   si_no creaárbol(ai,min(ad),sustrae(min(ad),ad))

AB5: miembro (e,árbolvacio ) = falso

AB6: miembro ( e, creaárbol (ai,e,ad)) = Si e < e1
                                    entonces miembro (e,ai)
                             si_no Si e1 < e entonces miembro (e,ad)
                                   si_no cierto

AB7: min (árbolvacio) = error

AB8: min (creaárbol (ai,e,ad)) = Si esvacio (ai)
                                  entonces  e
                                  si_no min (ai)

FIN DE  Arboles busqueda;
```

4. Conjuntos.

```
4.1 ESPEC  Conjuntos;

PARAMETRO  Elementos;
     Género Elemento

FIN DE  Elementos;

IMPORTA TODO DE Booleanos, Naturales;

EXPORTA  Conjunto, conjvacio, inserta, remover, unión, intersección,
diferencia, pertenece, tamaño, esvacio;

     GENERO Conjunto

     OPERACIONES
          conjvacio      : -> Conjunto
          inserta    : Elemento, Conjunto -> Conjunto
          remover    : Elemento, Conjunto -> Conjunto
          unión      : Conjunto, Conjunto -> Conjunto
          intersección    : Conjunto, Conjunto -> Conjunto
          diferencia      : Conjunto, Conjunto -> Conjunto
          pertenece       : Elemento, Conjunto -> Bool
          tamaño     : Conjunto -> Nat
          esvacio   : Conjunto -> Bool

AXIOMAS
     VAR     e,f: Elemento      c,d: Conjunto

C1:  pertenece (e, conjvacio) = falso
C2:  pertenece (e, inserta(f,c)) = Si e=f entonces  cierto
                                   si_no  pertenece (e,c)
C3:  esvacio ( conjvacio ) = cierto
C4:  esvacio ( inserta(e,c)) = falso
C5:  tamaño ( conjvacio) = 0
C6:  tamaño ( inserta (e,c)) = 1 + tamaño (remover(e,c))
C7:  remover (e, conjvacio ) = conjvacio
C8:  remover (e,inserta (f,c)) = Si e=f entonces  remover (e,c)
                                 si_no inserta ( f,remover (e,c) )
C9:  unión (c, conjvacio) = c
C10: unión (c, inserta (e,d)) = inserta (e, unión (c,d))
C11: intersección (c,conjvacio) = conjvacio
C12: intersección (c,inserta(e,d)) = Si pertenece (e,c) entonces
                                       inserta (e,intersección(c,d))
                                     si_no intersección (c,d)
C13: diferencia (c, conjvacio ) = c
C14: diferencia (c,inserta(e,d)) = Si pertenece( e,c) entonces
                                     remover (e, diferencia(c,d))

FIN DE  Conjuntos;
```

4.2 ESPEC Bolsas;

```
IMPORTA TODO DE  Conjuntos;
     Renombra Conjunto  como  Bolsa

EXPORTA   Bolsa, conjvacio, inserta, remover, unión, intersección,
diferencia, pertenece, tamaño, esvacio;

     OPERACIONES
          remover : Elemento, Bolsa -> Bolsa
          diferencia    : Bolsa, Bolsa -> Bolsa
          tamaño  : Bolsa -> Nat


AXIOMAS
     VAR  e,d : Elemento    b,c: Bolsa

C6:  tamaño ( inserta (e,b)) = 1 + tamaño (b)
C8:  remover ( d, inserta(e,b)) = Si e = d  entonces  b
                                  si_no inserta (e,remover(d,b))
C14: diferencia (b, inserta (e,c)) = remover (e,diferencia(b,c))

FIN DE  Bolsas;
```

5. Gráficas.

```
5.1 ESPEC  Gráficas;

PARAMETRO  Nodos
     Género Nodo
     operación
          "="       : Nodo, Nodo ->Bool

FIN DE  Nodos;

IMPORTA TODO DE Booleanos;

EXPORTA Gráfica, gráficavacia, agreganodo, agregaarista,
borranodo, borraarista, contiene, esadyacente, esvacia;

     GENERO Gráfica

     OPERACIONES
          gráficavacia  : -> Gráfica
          agreganodo    : Nodo, Gráfica -> Gráfica
          agregaarista  : Nodo, Nodo, Gráfica -> Gráfica
          borranodo     : Nodo, Gráfica -> Gráfica
          borraarista   : Nodo, Nodo, Gráfica -> Gráfica
          contiene       : Nodo, Gráfica -> Bool
          esadyacente   : Nodo, Gráfica -> Bool
          esvacia       : Gráfica -> Bool


AXIOMAS
     VAR    n,m,p,q : Nodo            g : Gráfica

G1:  borranodo (n, gráficavacia ) = gráficavacia

G2:  borranodo (m, agraganodo(n,g)) = Si n= m entonces
                                          borranodo (n,g)
                                 si_no agraganodo (m,borranodo(n,g))

G3:  borranodo (n, agregaarista (p, q, g)) = Si n=p or n=q
                  entonces borranodo(n,g)
                  si_no agregaarista(p,q,borranodo(n,g))

G4:  borraarista (n,m, gráficavacia ) = gráficavacia
```

```
G5:  borraarista (n,m, agreganodo(p,g)) =
                 agreganodo (p, borraarista (n,m,g))

G6:  borraarista (n,m, agregaarista (p,q,g)) =
                 Si ( n=p y m=q ) or ( n=q y m=p ) entonces
                 borraarista (n,m, g)
                 Si no agregaarista (p,q,borraarista (n,m,g,))

G7:  contiene (gráficavacia ) = falso

G8:  contiene (n, agreganodo (m,g)) = Si n = m entonces cierto
                                      si_no contiene (n,g)

G9:  contiene (n, agregaarista (p, q,g)) = Si n=p or n=q
                                      entonces cierto
                                      si_no contiene (n,g)

G10: esadyacente (n,m, gráficavacia) = falso

G11: esadyacente (n,m, agreganodo (p,g)) = esadyacente (n,m, g)

G12: esadyacente (n,m, agregaarista(p,q,g)) =
                              Si (n=p y m=q) or (n=q y m=p)
                              entonces cierto
                              si_no esadyacente (n,m, g)

G13: esvacia (gráficavacia ) = cierto

G14: esvacia (agreganodo (n,g)) = falso

G15: esvacia (agregaarista (n,m, g) = falso

FIN DE Gráficas;
```

```
5.2 ESPEC  Digráficas;

IMPORTA TODO DE Gráficas;

EXPORTA   Gráfica, gráficavacia, agreganodo, agregaarista,
borranodo,  borraarista,  contiene,  essucesor,  espredecesor,
esvacia;

     OPERACIONES
          essucesor      : Nodo, Nodo, Gráfica -> Bool
          espredecesor   : Nodo, Nodo, Gráfica -> Bool

AXIOMAS
     VAR   n,m,p,q : Nodo            g : Gráfica

G10: espredecesor (n,m, gráficavacia) = falso

G11: espredecesor (p,q, agreganodo (n,g))=
                                Si ( n=p or n=q )
                                entonces cierto
                                si_no espredecesor (p,q, g)

G12: espredecesor (p,q,agregaarista (n,m, g)) =
                                Si ( n=p y m=q )
                                entonces cierto
                                si_no espredecesor (p,q, g)

G16: essucesor (n,m, gráficavacia) = falso

G17: essucesor (p,q, agreganodo (n,g)) =
                                Si ( n=p or n=q )
                                entonces falso
                                si_no essucesor (p,q, g)

G18: essucesor (p,q, agregaarista (n,m, g))
                                Si (n = q y m = p )
                                entonces cierto
                                si_no essucesor (p,q, g)

FIN DE  Digráficas;
```

6. Especificaciones de Tablas de Dispersión.

6.1 ESPEC Diccionario;

```
PARAMETRO Datos
   GENERO llave, dato
   OPERACIONES
      igual : llave, llave -> bool
   AXIOMAS
     igual (j,j) = cierto
     igual (j,k) = igual (k,j)
     igual (j,k) = cierto  And  igual (k,l) = cierto
                   => igual (j,l) = cierto
FIN DE Datos;


EXPORTA  Dicc, DicVacio, inserta, elimina, recupera, pertenece,
esVacio;

     OPERACIONES
          DicVacio: -> Dicc
          inserta : llave, dato, Dicc -> Dicc
          elimina : llave, Dicc -> Dicc
          recupera : llave, Dicc -> dato
          pertenece : llave, Dicc -> bool
          esVacio  : Dicc -> bool

AXIOMAS
     VAR    k,j : llave      d : dato      dic : Dicc

D1:  elimina (k,DicVacio) = DicVacio

D2:  elimina (k,inserta(j,d,dic)) = Si igual (k,j) entonces
                                       elimina (k,dic)
                                 si_no inserta(j,d,elimina(k,dic))

D3:  recupera (k,DicVacio) = error

D4:  recupera (k,inserta(j,d,dic)) = Si igual (k,j) entonces d
                                     si_no recupera(k,dic)

D5:  pertenece (k,DicVacio) = falso

D6:  pertenece (k,inserta(j,d,dic)) = Si igual(k,j) entonces cierto
                                      si_no pertenece (k,dic)

D7:  esVacio (DicVacio) = cierto

D8:  esVacio (inserta(j,d,dic)) = falso

FIN DE Diccionario;
```

```
6.2 ESPEC  Tablas;

PARAMETRO Elem
     Género elem
FIN DE Elem;


PARAMETRO Rango
     Género rango
     operación
          iguales : rango, rango -> bool
FIN DE Rango;


EXPORTA  Tabla, TabVacia, asigna, inf;

     GENERO Tabla

     OPERACIONES
          TabVacia : -> Tabla
          asigna   : elem, rango, Tabla -> Tabla
          inf      : rango, Tabla -> elem


AXIOMAS
     VAR   e : elem    i,j : rango    t : Tabla

T1:  inf (i,TabVacia) = error

T2:  inf (i,asigna (e,j,t)) = Si iguales (i,j) entonces e
                              si_no inf (i,t)

FIN DE Tablas;
```

```
6.3 ESPEC TablaHash;

PARAMETRO FunHash
     Género dato, llave, rango
     operacion
          hash : llave -> rango
FIN DE FunHash;

IMPORTA TODO DE Tablas con Elem como Diccionario de Funhash

     Renombra Tabla como Tabhash

EXPORTA  Tabhash, incluye, excluye, inf, existe, asignado;

     OPERACIONES
          incluye : llave, dato, tabhash -> tabhash
          excluye : llave, tabhash -> tabhash
          inf     : llave, tabhash -> dato
          existe  : llave, tabhash -> bool
          asignado: llave, tabhash -> bool

AXIOMAS
     VAR    k :llave  d:dato  i:rango  dic:Dicc  t:Tabla

TH1: incluye (l,d,TabVacia) = asigna (inserta(k,d,DicVacio),hash(k)
                                 , TabVacia)

TH2: incluye (k,d,asigna(dic,i,t)) = Si hash(k) = i entonces
                          asigna(inserta(k,d,dic),i,t)
                          si_no asigna(dic,i,incluye(k,d,t))

TH3: excluye (l,TabVacia) = TabVacia

TH4: excluye (k,asigna(dic,i,t)) = Si hash(k) = i entonces
                          asigna(elimina(k,dic),i,t)
                          si_no asigna(dic,i,excluye(k,t))

TH5: inf(k,TabVacia) = error

TH6: inf (k,asigna(dic,i,t)) = Si hash(k) = i entonces
                          recupera(k,dic)
                          si_no inf (k,t)

TH7: existe(k,TabVacia) = falso

TH8: existe(k,asigna(dic,i,t)) = Si hash(k) = i entonces
                          pertenece (k,dic)
                          si_no existe (k,t)

TH9: asignado (i,TabVacia) = falso

TH10:asignado (i,asigna(dic,j,t)) = Si i = j entonces cierto
                          si_no asignado (i,t)
FIN DE TablaHash;
```

```
6.4 ESPEC TablaHColis;

PARAMETRO Colisión
     GENERO llave, rango, dato
     OPERACIONES hash, sig
          hash : llave -> rango
          sig: rango -> rango
FIN DE Colisión;

IMPORTA TODO DE TablaHash con Elem como  (llave * dato)

EXPORTA  tabhash, incluye, inf;

     OPERACIONES
          incluye : llave, tabhash -> tabhash
          inf     : llave, tabhash -> dato

          añade : llave, dato, tabhash -> tabhash
          busca : llave, rango, tabhash -> dato

AXIOMAS
     VAR   k : llave    d : dato   i : rango   t: Tablahash

TC1: incluye (k,d,t) = Si asignado (hash(k),t) entonces
                        añade(k,d,sig(hash(k)),t)
                       si_no asigna ((k,d(,hash(k),t)

TC2: inf (k,t) = Si Not asignado(hash(k),t) entonces error
                 si_no Si #1(inf(hash(k),t)) = k entonces
                       #2(inf(hash(k),t)
                       si_no busca (k,sig(hash(k)),t)

TC3: añade (k,d,i,t) = Si hash(k) = i entonces error
                       si_no Si asignado(i,t) entonces
                             añade(k,d,sig(i),t)
                             si_no asigna ((k,d),i,t)

TC4: busca (k,i,t) = Si hash(k) = i or not asignado(i,t)
                     entonces error
                     si_no Si #1(inf(i,t)( = k entonces
                           #2(inf(i,t))
                           si_no busca (k,sig(i),t)

FIN DE TablahColis;
```

APENDICE B

Diccionario de Functores y Estructuras Básicas.

En el presente diccionario se presentan todos los archivos que componen la biblioteca; a continuación del nombre de un archivo se listan las estructuras básicas y los identificadores de los functores definidos en el archivo, así como las signaturas sus parámetros y la signatura correspondiente a la estructura que genera cada functor si la tiene, si no hay una definida, el sistema asigna la mas general. Cada programa indica las llamadas a los otros programas que requiera, por lo que solo hay que invocarlos.

Para hacer uso de cualquier functor se debe incluir en el programa el nombre del archivo por medio de la cláusula *use*.

elementos.sml

```
Booleano : ELEMCOMPAR
Natural  : ELEMORDEN
Real     : ELEMORDEN
Cadena   : ELEMORDEN
```

ecolas.sml

```
Estrlineal (E: ELEMENTO ) : ESTRLINEALB
EstrCola   (E: ELEMENTO ) : ESTRCOLA
Cola       (E: ELEMENTO ) : COLA
```

ecolasd.sml

```
Coladoble  (E: ELEMENTO ) : COLADOBLE
```

ecolasa.sml

```
ColaAc     ( Co : COLA , Cota : COTA ) : COLAAC
```

ecolasp.sml

```
Colaprior (E : ELEMORDEN ) : COLAPRIOR
```

pilas.sml

```
Pila (E : ELEMENTO ) : PILA
```

pilaac.sml

```
PilaAc ( Pi : PILA, Cota : COTA ) : PILAAC
```

listas.sml

```
Lista ( E : ELEMENTO ) : LISTA
```

arbol.sml

```
Arbolbin ( E: ELEMORDEN ) : ARBOLBIN
```

extarbol.sml

```
Extarbol (E: ELEMORDEN ) : EXTIENDEARBOL
```

heap.sml

```
Heap (E: ELEMORDEN ) : HEAP
```

arbus.sml

```
arbolbusq (E: ELEMORDEN ) : ARBOLBUSQ
```

avl.sml

ArbolAVL (E: ELEMORDEN) : ARBOLAVL

conjunto.sml

Conjunto (E: ELEMCOMPAR) : CONJUNTO

bolsas.sml

Bolsa (E: ELEMCOMPAR) : BOLSA

graficas.sml

Grafica (N : ELEMCOMPAR) : GRAFICA

digraf.sml

Digrafica (N: ELEMCOMPAR) : DIGRAFICA

tablah.sml

Dicc (D: DATOS) : DICCIONARIO

Tabla (E : ELEM, R :RANGO) : TABLA

Tabhash (Ta :TABLA, F : FUNHASH) : TABHASH

TabHColis (T : TABHASH, C :COLISION)

DICCIONARIO DE SIGNATURAS.

En el presente diccionario se encuentran las signaturas de todos los functores de la biblioteca que tienen una asignada, mostrando las funciones que exporta cada tipo, las constructoras estan indicadas con la inicial en mayúscula.

SIGNATURAS

```
signature ELEMENTO =
sig
    type elemento
end;
```

```
signature ELEMCOMPAR =
sig
    type elemento
    val igual : (elemento * elemento ) -> bool
end;
```

```
signature ELEMORDEN =
sig
    type elemento
    val igual : ( elemento * elemento ) -> bool
    val menor : ( elemento * elemento ) -> bool
end;
```

Estructuras lineales básicas

```
signature ESTRLINEALB =
sig
   structure Elem : ELEMENTO
   type estrlineal
(*  operaciones *)
   val EstrVacia: estrlineal
   val Inserta:( Elem.elemento * estrlineal) -> estrlineal
   val resto: estrlineal -> estrlineal
   val inicial: estrlineal -> Elem.elemento
   val esVacia: estrlineal -> bool
end;
```

Colas

```
signature ESTRCOLA =
sig
   structure Ec : ESTRLINEALB
   val ColaVacia : Ec.estrlineal
   val Meteult : (Ec.Elem.elemento * Ec.estrlineal) -> Ec.estrlineal
   val ultimo : Ec.estrlineal -> Ec.Elem.elemento
   val sacault : Ec.estrlineal -> Ec.estrlineal
   val esvacia  : Ec.estrlineal -> bool
   val Meteprim : (Ec.Elem.elemento * Ec.estrlineal) -> Ec.estrlineal
   val primero  : Ec.estrlineal -> Ec.Elem.elemento
   val sacaprim : Ec.estrlineal -> Ec.estrlineal
end;
```

```
signature COLA =
sig
   structure Eco  : ESTRCOLA
   val ColaVacia : Eco.Ec.estrlineal
   val Meteult   : (Eco.Ec.Elem.elemento*Eco.Ec.estrlineal)->Eco.Ec.estr
   val primero   : Eco.Ec.estrlineal -> Eco.Ec.Elem.elemento
   val sacaprim   : Eco.Ec.estrlineal -> Eco.Ec.estrlineal
   val esvacia   : Eco.Ec.estrlineal -> bool
end;



signature COLADOBLE =
sig
   structure Eco : ESTRCOLA
end;
```

Pilas

```
signature PILA =
sig
   structure Es : ESTRLINEALB
   val Vacia : Es.estrlineal
   val Mete  : (Es.Elem.elemento * Es.estrlineal )-> Es.estrlineal
   val saca  : Es.estrlineal -> Es.estrlineal
   val tope  : Es.estrlineal -> Es.Elem.elemento
   val esvacia : Es.estrlineal -> bool
end;
```

Listas

```
signature LISTA =
sig
   structure Es  : ESTRLINEALB
   val ListaVacia : Es.estrlineal
   val Inserta    : (Es.Elem.elemento * Es.estrlineal ) -> Es.estrlineal
   val resto      : Es.estrlineal -> Es.estrlineal
   val inicial    : Es.estrlineal -> Es.Elem.elemento
   val esvacia    : Es.estrlineal -> bool
   val longitud   : Es.estrlineal -> int
   val enesimo    : (int * Es.estrlineal) -> Es.Elem.elemento
   val concatena : (Es.estrlineal * Es.estrlineal) -> Es.estrlineal
end;
```

Arboles binarios

```
signature ARBOLBIN =
sig
(*  parametro  *)
   type elemento
   val menor    : (elemento * elemento) -> bool
   val igual    : (elemento * elemento) -> bool
(*  genero que define  *)
   datatype arbolBin = ArbolVacio | CreaArbol of (arbolBin * elemento *
                                    arbolBin)
(*  auxiliares  *)
   val izq       : arbolBin -> arbolBin
   val der: arbolBin -> arbolBin
   val raiz: arbolBin -> elemento
   val altura : arbolBin -> int
   val esVacio: arbolBin -> bool
end;
```

Arboles binarios extendidos

```
signature EXTIENDEARBOL =
sig
   structure Arbol : ARBOLBIN

(*  operaciones que se agregan  *)
   val eshoja : Arbol.arbolBin -> bool
   val numhojas: Arbol.arbolBin -> int
   val numnodos : Arbol.arbolBin -> int
   val estalleno: Arbol.arbolBin -> bool
end;
```

Arboles piramidales (heap)

```
signature HEAP =
sig
   type elemento
   val menor : (elemento * elemento ) -> bool
   type arbolBin
(*  operaciones visibles de arboles amontonados  *)
   val Arbolvacio : arbolBin
   val inserta    : (elemento * arbolBin) -> arbolBin
   val max        : arbolBin -> elemento
   val esVacio    : arbolBin -> bool
   val sacamax    : arbolBin -> arbolBin
end;
```

Arbol balanceado o AVL

```
signature ARBOLAVL =
sig
   type arbolBin
   type elemento
   val menor      : (elemento * elemento ) -> bool
   val Arbolvacio : arbolBin
   val estabalan  : arbolBin -> bool
   val minim      : arbolBin -> elemento
   val inserta    : (elemento * arbolBin) -> arbolBin
   val sustrae    : (elemento * arbolBin) -> arbolBin

end;
```

Arboles de búsqueda

```
signature ARBOLBUSQ =
sig
   structure Ar : ARBOLBIN
(*  operaciones que se exportan  *)
   val ArbolVacio : Ar.arbolBin
   val esVacio    : Ar.arbolBin -> bool
   val insertab   : (Ar.elemento * Ar.arbolBin) -> Ar.arbolBin
   val minimo     : Ar.arbolBin -> Ar.elemento
   val maximo     : Ar.arbolBin -> Ar.elemento
   val sustraeb   : (Ar.elemento * Ar.arbolBin) -> Ar.arbolBin
   val miembro    : (Ar.elemento * Ar.arbolBin) -> bo
end;
```

Conjuntos

```
signature CONJUNTO =
sig
   (*  parametro  *)
   type elemento
   val iguales : (elemento * elemento) -> bool
   (*  genero que se define  *)
   datatype conj = ConjVacio | Inserta of (elemento * conj)
    (*     funciones auxiliares       *)
   val remover      : (elemento * conj) -> conj
   val pertenece    : (elemento * conj) -> bool
   val union: (conj * conj ) -> conj
   val interseccion : (conj* conj) -> conj
   val diferencia : (conj * conj) -> conj
   val tamano     : conj -> int
   val esVacio    : conj -> bool
end;
```

Gráficas

```
signature GRAFICA =
sig
   (*   parametros     *)
   type nodo
   val igual : (nodo * nodo) -> bool
(*   tipo que se define  *)
   datatype grafica = GrafVacia | Insertanodo of (nodo * grafica)
                | Insertaarista of (nodo * nodo* grafica)
   val borranodo : (nodo * grafica) -> grafica
   val borraarista : (nodo*nodo*grafica) -> grafica
   val adyacente : (nodo*nodo*grafica) -> bool
   val contienenodo : (nodo * grafica) -> bool
   val esVacia  : grafica -> bool
end;
```

```
signature DATOS =
sig
     type llave
     type dato
     val igual : (llave * llave ) -> bool
end;


signature DICCIONARIO =
sig
(*   parametros      *)
   structure Da : DATOS
(*   definicion del tipo dic   *)
   type dicc
   val DiccVacio : dicc
   val Inserta   : (Da.dato * Da.llave * dicc) -> dicc
   val elimina   : (Da.llave * dicc) -> dicc
   val recupera : (Da.llave * dicc) -> Da.dato
   val pertenece: (Da.llave * dicc) -> bool
   val esVacio   : dicc -> bool
end;

signature ELEM =
sig
   type elem
end;

signature RANGO =
sig
   type rango
   val iguales : (rango * rango) -> bool
end;
```

```
signature TABLA =
sig
(*   parametros    *)
   structure E : ELEM
   structure R : RANGO
(*   definicion del tipo tabla   *)
   datatype tabla = TabVacia | Asigna of ( E.elem * R.rango * tabla)
   val contenido : (R.rango * tabla ) -> E.elem
end;


signature FUNHASH =
sig
   structure D : DATOS
   structure R : RANGO
   val hash : D.llave -> R.rango
end;


signature TABHASH =
sig
   structure T : TABLA
   structure F : FUNHASH
   structure Di : DICCIONARIO
   val incluye : (T.E.elem * Di.Da.llave *T.tabla) -> T.tabla
   val excluye : ( Di.Da.llave * T.tabla) -> T.tabla
   val inf     : ( Di.Da.llave * T.tabla) -> T.E.elem
   val existe  : ( Di.Da.llave * T.tabla) -> bool
   val asignado: ( Di.Da.llave * T.tabla) -> bool
end;

signature COLISION =
sig
   structure F : FUNHASH
   val sigue : F.rango -> F.rango
end;
```

APENDICE C

Biblioteca de implementaciones en ML.

En este apéndice se muestran los contenidos de cada uno de los archivos de la biblioteca en ML, en los que se encuentran las signaturas y functores que implementan cada estructura de datos.

```
(*        archivo de elementos   y estructuras basicas        *)
(* su nombre es elementos.sml  *)


signature ELEMENTO =
sig
    type elemento
end;


signature ELEMCOMPAR =
sig
    type elemento
    val igual : (elemento * elemento ) -> bool
end;


signature ELEMORDEN =
sig
    type elemento
    val igual : ( elemento * elemento ) -> bool
    val menor : ( elemento * elemento ) -> bool
end;

(*                estructuras basicas            *)


structure Booleano : ELEMCOMPAR =
struct
      type elemento = bool
      fun igual (a:bool ,b) = (a = b)

end;
```

```
structure Natural : ELEMORDEN =
struct
      type elemento = int
      fun igual (x:int,y) = (x = y)
      fun menor (x,y:int) = (x < y)
end;


structure Real :ELEMORDEN=
struct
      type elemento = real
      fun igual (x:real,y) = (x = y)
      fun menor (x:real,y) = (x < y)
end;



structure Cadena : ELEMORDEN =
struct
      type elemento = string
      fun igual (x:string,y) = (x = y)
      fun menor (x:string,y) = (x < y)
end;



(*   archivo estrlineales.sml : estructuras lineales     *)
(*  se requiere toda la informacion sobre los elementos  *)
(*  poner use["elementos.sml"]; antes de llamarlo        *)

signature ESTRLINEALB =
sig
   structure Elem : ELEMENTO
   type estrlineal
(*  operaciones *)
   val EstrVacia: estrlineal
   val Inserta:( Elem.elemento * estrlineal) -> estrlineal
   val resto: estrlineal -> estrlineal
   val inicial: estrlineal -> Elem.elemento
   val esVacia: estrlineal -> bool
```

```
end;

functor Estrlineal (E : ELEMENTO ) : ESTRLINEALB =
struct
   structure  Elem = E

  datatype estrlineal =EstrVacia | Inserta of (Elem.elemento*estrlineal)

(* definicion de las operaciones  *)

   exception Resto
   fun resto EstrVacia = raise Resto
    |   resto (Inserta (e,l)) = l

   exception Inicial
   fun inicial EstrVacia = raise Inicial
    | inicial (Inserta (e,l)) = e

   fun esVacia EstrVacia = true
    | esVacia _          = false
end;


(* archivo  ecolas.sml:  colas basicas y estructuras tipo  colas     *)
(*  debe llamarse antes a estrlineales.sml         *)
(* poner use["estrlineales.sml"];       *)

signature ESTRCOLA =
sig
   structure Ec : ESTRLINEALB
   val ColaVacia : Ec.estrlineal
   val Meteult : (Ec.Elem.elemento * Ec.estrlineal) -> Ec.estrlineal
   val ultimo : Ec.estrlineal -> Ec.Elem.elemento
   val sacault : Ec.estrlineal -> Ec.estrlineal
   val esvacia  : Ec.estrlineal -> bool
   val Meteprim : (Ec.Elem.elemento * Ec.estrlineal) -> Ec.estrlineal
```

```
   val primero  : Ec.estrlineal -> Ec.Elem.elemento
   val sacaprim : Ec.estrlineal -> Ec.estrlineal
end;

functor ESTRCOLA (E: ESTRLINEALB ) : ESTRCOLA =
struct
   structure Ec = E
   open Ec

   val ColaVacia = EstrVacia
   val Meteult   = Inserta
   val ultimo    = inicial
   val sacault   = resto
   val esvacia   = esVacia

   fun Meteprim (e,c) = if esvacia (c) then Meteult (e,ColaVacia)
                        else Meteult (ultimo(c), Meteprim (e,sacault(c)))

   exception Primero
   fun primero (c) = if esvacia(c) then raise Primero
                     else if esvacia(sacault (c)) then ultimo (c)
                          else primero (sacault(c))


   exception Sacaprim
   fun sacaprim (c) =if esvacia (c) then raise Sacaprim
                     else if esvacia (sacault (c)) then ColaVacia
                          else Meteult(ultimo(c),sacaprim(sacault(c)))
end;

functor EstrCola (E : ELEMENTO ) : ESTRCOLA =
struct
structure S = ESTRCOLA( Estrlineal (E))
open S
end;
```

```
signature COLA =
sig
   structure Eco  : ESTRCOLA
   val ColaVacia : Eco.Ec.estrlineal
   val Meteult   : (Eco.Ec.Elem.elemento*Eco.Ec.estrlineal)->
                                                    Eco.Ec.estrlineal
   val primero   : Eco.Ec.estrlineal -> Eco.Ec.Elem.elemento
   val sacaprim   : Eco.Ec.estrlineal -> Eco.Ec.estrlineal
   val esvacia   : Eco.Ec.estrlineal -> bool
end;


functor COLA (E:ESTRCOLA) : COLA =
struct
   structure Eco = E
   open Eco
   val ColaVacia = Ec.EstrVacia
   val Meteult  = Ec.Inserta
   val primero  = ultimo
   val sacaprim  = sacault
   val esvacia   = Ec.esVacia
end;


functor Cola (E : ELEMENTO) : COLA =
struct
structure S =  COLA( ESTRCOLA ( Estrlineal(E)))
open S
end;

(*  archivo ecolasd.sml:          colas dobles          *)
```

```
(*  se requieren las estructuras lineales basicas y las colas simples *)
(*  poner antes : use["estrlineales.sml"];
                  use["ecolas.sml"];              *)

functor COLADOBLE (E:ESTRCOLA ) =
struct
   open E

end;


functor Coladoble ( E: ELEMENTO ) =
struct
structure S = COLADOBLE ( ESTRCOLA ( Estrlineal (E)))
open S
end;



(*  archivo ecolasa.sml:  colas acotadas        *)
(* se traen las cotas para las colas acotadas  *)
use["cotas.sml"];
(*  se traen las estructuras lineales basicas y las colas simples *)
(*  poner use["estrlineales.sml"];     y
         use["ecolas.sml"];                    *)

functor ColaAc (Co :COLA, Cota: COTA ) =
struct
   open Co
   val max = Cota.max
   exception Emeteult
   fun Meteult (e,c) = if estallena (c) then raise Emeteult
                       else Meteult (e,c)
   and
       tamano (c) = if  esvacia (c)  then 0
                    else (1+ tamano (sacaprim (c)))
   and
```

```
      estallena (c) = (tamano(c) = max)
end;


(*  archivo ecolasp.sml:  colas con prioridad          *)
(*  se traen las estructuras lineales basicas y colas basicas *)
(*  poner   use["estrlineales.sml"];    y
           use["ecolas.sml"];                *)

functor Colaprior (Co:COLA, E:ELEMORDEN sharing structure
                  Co.Eco.Ec.Elem = E ) =
struct
   open Co

   val igual = E.igual
   val menor = E.menor

   fun maximo (c) = if esvacia (sacaprim(c)) andalso
                    menor(maximo(c),primero(c)) then primero(c)
                    else maximo(sacaprim(c))
end;



(*   archivo  pilas.sml       *)
(*     requiere de llamada a estrlineales.sml      *)
(*  poner use["estrlineales.sml"];                  *)


signature PILA =
sig
   structure Es : ESTRLINEALB
   val Vacia : Es.estrlineal
   val Mete  : (Es.Elem.elemento * Es.estrlineal )-> Es.estrlineal
   val saca  : Es.estrlineal -> Es.estrlineal
   val tope  : Es.estrlineal -> Es.Elem.elemento
```

```
   val esvacia : Es.estrlineal -> bool
end;


functor PILA (E:ESTRLINEALB) : PILA =
struct
   structure Es=E
   open Es
   val Vacia = EstrVacia
   val Mete  = Inserta
   val saca  = resto
   val tope  = inicial
   val esvacia = esVacia
end;

functor Pila (E : ELEMENTO ) : PILA =
struct
structure S = PILA (Estrlineal(E))
open S
end;

(*   archivo pilaac.sml:        pilas acotadas      *)
(*   se llama al archivo de cotas                   *)
use["cotas.sml"];
(*  requiere que se llame a estrlineales y a pilas asi  *)
(*  use["estrlineales.sml"];     y
    use["pilas.sml"];                                 *)

functor PilaAc (Pi: PILA , Cota: COTA) =
struct
   open Pi
   val max = Cota.max
   exception Emete
   fun Mete (e,p) = if estallena (p) then raise Emete
                    else Mete (e,p)
   and
```

```
      tamano (p) = if esvacia (p) then 0
                   else (1 + tamano (saca (p)) )
   and
      estallena (p) = (tamano(p) = max )
end;

(*        archivo  listas.sml        *)
(*    requiere de llamar a estrlineales.sml        *)
(*  poner use["estrlineales.sml"];                 *)

signature LISTA =
sig
   structure Es  : ESTRLINEALB
   val ListaVacia : Es.estrlineal
   val Inserta    : (Es.Elem.elemento * Es.estrlineal) -> Es.estrlineal
   val resto      : Es.estrlineal -> Es.estrlineal
   val inicial    : Es.estrlineal -> Es.Elem.elemento
   val esvacia    : Es.estrlineal -> bool
   val longitud   : Es.estrlineal -> int
   val enesimo    : (int * Es.estrlineal) -> Es.Elem.elemento
   val concatena : (Es.estrlineal * Es.estrlineal) -> Es.estrlineal
end;

functor LISTA(E: ESTRLINEALB) : LISTA =
struct
   structure Es = E
   open Es
   val ListaVacia = EstrVacia
   val Inserta = Inserta
   val resto = resto
   val inicial = inicial
   val esvacia = esVacia
   fun longitud (l) = if esVacia (l) then 0
                      else 1 + longitud (resto(l))
   exception Enesimo
   fun enesimo (n,l) = if n=0 then raise Enesimo
                       else if n=1 then inicial (l)
```

```
                        else enesimo (n-1,resto(l))
   fun concatena (l,m) = if esvacia (l) then m
                         else if esvacia (m) then l
                              else Inserta ((inicial m),
                                   concatena (l, resto m))
end;
functor Lista ( E : ELEMENTO ) : LISTA =
struct
structure S =LISTA (Estrlineal (E))
open S
end;


(*   archivo  arbol.sml                          *)
(*   requiere llamar a elementos.sml             *)
(*   poner use["elementos.sml"];                 *)

signature ARBOLBIN =
sig
(*  parametro  *)
   type elemento
   val menor    : (elemento * elemento) -> bool
   val igual    : (elemento * elemento) -> bool
(*  genero que define  *)
   datatype arbolBin = ArbolVacio | CreaArbol of (arbolBin*elemento *
                                                  arbolBin)

(*  auxiliares  *)
   val izq      : arbolBin -> arbolBin
   val der: arbolBin -> arbolBin
   val raiz: arbolBin -> elemento
   val altura : arbolBin -> int
   val esVacio: arbolBin -> bool
end;

functor Arbolbin ( Elem:ELEMORDEN ): ARBOLBIN =
```

```
struct
   type elemento = Elem.elemento
   val menor = Elem.menor
   val igual = Elem.igual
   datatype arbolBin = ArbolVacio | CreaArbol of (arbolBin*elemento *
                                                  arbolBin)

  exception Izq
   fun izq ArbolVacio = raise Izq
    | izq (CreaArbol (ai,e,ad)) = ai

   exception Der
   fun der ArbolVacio = raise Der
    | der (CreaArbol (ai,e,ad)) = ad

exception Raiz
   fun raiz ArbolVacio = raise Raiz
    | raiz (CreaArbol (ai,e,ad)) = e

   fun altura ArbolVacio = 0
    | altura (CreaArbol (ai,e,ad)) = if altura (ai) <= altura (ad)
                                     then 1 + altura (ad)
                                     else 1 + altura (ai)

   fun esVacio ArbolVacio = true
    | esVacio _           = false

end;



(*   archivo extarbol.sml:    arboles binarios  extendidos    *)
(*  requiere llamar a arbol.sml        *)
(*  poner use["arbol.sml"];            *)

signature EXTIENDEARBOL =
sig
```

```
   structure Arbol : ARBOLBIN
(*  operaciones que se agregan  *)
   val eshoja : Arbol.arbolBin -> bool
   val numhojas: Arbol.arbolBin -> int
   val numnodos : Arbol.arbolBin -> int
   val estalleno: Arbol.arbolBin -> bool
end;

functor EXTARBOL (A: ARBOLBIN ) : EXTIENDEARBOL =
struct
   structure Arbol = A
   open Arbol

   fun eshoja (ArbolVacio) = false
    | eshoja (CreaArbol (ai,e,ad)) = esVacio (ai)
                                     andalso esVacio(ad)

   fun numhojas (ArbolVacio) = 0
    | numhojas (CreaArbol (ai,e,ad)) = if eshoja(CreaArbol
                                       (ai,e,ad)) then 1
                                       else numhojas(ai) + numhojas(ad)
   fun numnodos ArbolVacio = 0
    | numnodos (CreaArbol (ai,e,ad)) = 1 + numnodos(ai) + numnodos(ad)

   fun estalleno ArbolVacio = true
    | estalleno (CreaArbol(ai,e,ad)) = if (altura(ai) = altura(ad))
                andalso estalleno(ai) andalso estalleno(ad) then true
                else false
end;


functor Extarbol ( E: ELEMORDEN ) : EXTIENDEARBOL =
struct
structure S = EXTARBOL (Arbolbin (E))
open S
end;
```

```
(*   archivo heap.sml:     arboles piramidales o heap     *)
(*   necesita llamarse antes a extarbol.sml     *)
(*  poner  use["extarbol.sml"];                  *)

signature HEAP =
sig
   type elemento
   val menor : (elemento * elemento ) -> bool
   type arbolBin
(*  operaciones visibles de arboles amontonados  *)
   val Arbolvacio : arbolBin
   val inserta    : (elemento * arbolBin) -> arbolBin
   val max        : arbolBin -> elemento
   val esVacio    : arbolBin -> bool
   val sacamax    : arbolBin -> arbolBin
end;


functor HEAP (Aext:EXTIENDEARBOL ) : HEAP =
struct
   type elemento = Aext.Arbol.elemento
   val menor = Aext.Arbol.menor
   type arbolBin = Aext.Arbol.arbolBin
   val Arbolvacio = Aext.Arbol.ArbolVacio
(*  funciones ocultas  *)
   fun hacecompl (e,Arbolvacio) = Aext.Arbol.CreaArbol(Arbolvacio,e,
                                                       Arbolvacio)
     | hacecompl (e,Aext.Arbol.CreaArbol(ai,el,ad)) =
```

```
                if (Aext.Arbol.altura(ai) = Aext.Arbol.altura(ad)
                  andalso Aext.estalleno(ai))
                  orelse (Aext.Arbol.altura(ai)=Aext.Arbol.altura(ad))
                  andalso  not(Aext.estalleno(ad))
                then Aext.Arbol.CreaArbol(ai,e1,hacecompl(e,ad))
                else Aext.Arbol.CreaArbol(hacecompl(e,ai),e1,ad)



fun  haceheap (e,ai,Arbolvacio) = if Aext.Arbol.esVacio(ai) then
                Aext.Arbol.CreaArbol(Arbolvacio,e,Arbolvacio) else
                        if menor(Aext.Arbol.raiz(ai) ,e)
                        then  Aext.Arbol.CreaArbol(ai,e,Arbolvacio)
                        else Aext.Arbol.CreaArbol(Aext.Arbol.CreaArbol
          (Arbolvacio,e,Arbolvacio),Aext.Arbol.raiz(ai),Arbolvacio)

  | haceheap (e,ai,ad) = if menor(Aext.Arbol.raiz(ai) ,e)
                    andalso menor(Aext.Arbol.raiz(ad) ,e)
            then Aext.Arbol.CreaArbol(ai,e,ad)
            else if menor(Aext.Arbol.raiz(ad), Aext.Arbol.raiz(ai))
then Aext.Arbol.CreaArbol(haceheap(e,Aext.Arbol.izq(ai),
                        Aext.Arbol.der(ai)),Aext.Arbol.raiz(ai),ad)
else Aext.Arbol.CreaArbol(ai,Aext.Arbol.raiz(ad),
     haceheap(e,Aext.Arbol.izq(ad),Aext.Arbol.der(ad)))

exception Creaheap
fun creaheap ArbolVacio = ArbolVacio
  | creaheap (Aext.Arbol.CreaArbol(ai,e,ad))= if
              (Aext.Arbol.altura(ai)-Aext.Arbol.altura(ad)) <= 1
            then haceheap (e,creaheap(ai),creaheap(ad))
            else raise Creaheap
exception Ult
fun ult Arbolvacio = raise Ult
  | ult (Aext.Arbol.CreaArbol(ai,e,ad)) = if Aext.Arbol.esVacio (ai)
              then e
              else if Aext.Arbol.altura(ad) < Aext.Arbol.altura(ai)
```

```
                    then ult (ad)
                    else ult (ai)

fun eliminault Arbolvacio = Arbolvacio
   fun inserta (e,a) = creaheap ( hacecompl (e,a))

   exception Max
   fun max(ArbolVacio) = raise Max
    |  max (Aext.Arbol.CreaArbol(ai,e,ad)) = e

   val esVacio = Aext.Arbol.esVacio

   exception Sacamax
   fun sacamax (a) = if esVacio (a) then raise Sacamax
                     else haceheap (ult(a),Aext.Arbol.izq(eliminault(a)),
                                    Aext.Arbol.der(eliminault(a)))
end;



functor Heap ( E : ELEMORDEN) : HEAP =
struct
structure S =HEAP ( EXTARBOL (Arbolbin (E)))
open S
end;



(*    archivo arbus.sml:    arboles de busqueda    *)
(*     debe llamarse a arbol.sml     *)
(*     usar use["arbol.sml"];        *)

signature ARBOLBUSQ =
sig
   structure Ar : ARBOLBIN
```

```
(*  operaciones que se exportan  *)
   val ArbolVacio : Ar.arbolBin
   val esVacio    : Ar.arbolBin -> bool
   val insertab   : (Ar.elemento * Ar.arbolBin) -> Ar.arbolBin
   val minimo     : Ar.arbolBin -> Ar.elemento
   val maximo     : Ar.arbolBin -> Ar.elemento
   val sustraeb   : (Ar.elemento * Ar.arbolBin) -> Ar.arbolBin
   val miembro    : (Ar.elemento * Ar.arbolBin) -> bool
end;



functor ARBOLBUSQ (A: ARBOLBIN) : ARBOLBUSQ =
struct
   structure Ar = A
   open Ar
   val ArbolVacio = ArbolVacio
   val esVacio = esVacio
   fun insertab (e,ArbolVacio) = CreaArbol (ArbolVacio, e, ArbolVacio)
    | insertab (e,CreaArbol (ai,e1,ad)) = if igual(e,e1)
                          then CreaArbol (ai,e,ad)
                          else if menor(e, e1)
                               then CreaArbol(insertab(e,ai),e1,ad)
                               else CreaArbol(ai,e1,insertab(e,ad))
   exception Minimo
   fun minimo (ArbolVacio) = raise Minimo
    | minimo (CreaArbol(ai,e,ad)) = if esVacio ai
                                    then e
                                    else minimo ad

   exception Maximo
   fun maximo ArbolVacio = raise Maximo
    | maximo (CreaArbol(ai,e,ad)) = if esVacio ad
                                    then e
                                    else maximo ad

   fun sustraeb (e,ArbolVacio) = ArbolVacio
```

```
    | sustraeb (e,CreaArbol(ai,e1,ad)) = if esVacio ai
                    then sustraeb (e,ad)
                    else if esVacio ad
                         then sustraeb (e,ai)
                         else if igual(e,e1)
                              then CreaArbol (ai,minimo ad,
                                   sustraeb(minimo ad,ad))
                              else if menor(e,e1)
                               then CreaArbol(sustraeb (e,ai),e1,ad)
                               else CreaArbol(ai,e1,sustraeb(e,ad))


  fun miembro (e,ArbolVacio) = false
    | miembro (e,CreaArbol(ai,e1,ad)) = if menor(e,e1)
                                        then miembro (e,ai)
                                        else if menor(e1,e)
                                             then miembro (e,ad)
                                             else true
end;

functor Arbolbusq (E : ELEMORDEN ) : ARBOLBUSQ =
struct
structure S = ARBOLBUSQ ( Arbolbin (E))
open S
end;


(*   archivo avl.sml:     arbol balanceado o AVL     *)


(*   debe llamarse a arbol.sml antes       *)
(*   poner use["arbol.sml"];               *)



signature ARBOLAVL =
```

```
sig
   type arbolBin
   type elemento
   val menor      : (elemento * elemento ) -> bool
   val Arbolvacio : arbolBin
   val estabalan  : arbolBin -> bool
   val minim      : arbolBin -> elemento
   val inserta    : (elemento * arbolBin) -> arbolBin
   val sustrae    : (elemento * arbolBin) -> arbolBin

end;


functor ARBOLAVL (A: ARBOLBIN) : ARBOLAVL =
struct
   type arbolBin = A.arbolBin
   type elemento = A.elemento
   val menor = A.menor
   val Arbolvacio = A.ArbolVacio
   fun estabalan Arbolvacio = true
    | estabalan (A.CreaArbol(ai,e,ad)) = if A.altura(ai)=A.altura(ad)
                                         then true
                                         else false
(*  funciones ocultas   *)

   fun rotaizq Arbolvacio = Arbolvacio
    | rotaizq (A.CreaArbol (ai,e,ad)) = if A.esVacio (ad)
                      then A.CreaArbol (ai,e,ad)
                      else A.CreaArbol(A.CreaArbol(ai,e,A.izq(ad)),
                           A.raiz(ad), A.der(ad))

   fun rotader Arbolvacio= Arbolvacio
    | rotader (A.CreaArbol (ai,e,ad)) = if A.esVacio(ai)
                      then A.CreaArbol (ai,e,ad)
                      else A.CreaArbol (A.izq (ai), A.raiz (ai),
                           A.CreaArbol(A.der(ai), e,ad))
```

```
fun balanceizq Arbolvacio = Arbolvacio
 | balanceizq(A.CreaArbol(ai,e,ad))=if (A.altura(ad)-A.altura(ai))<=1
                  then A.CreaArbol(ai,e,ad)
                  else if A.altura(A.der(ad)) < A.altura(A.izq(ad))
                       then rotaizq (A.CreaArbol(ai,e,rotader(ad)))
                       else rotaizq (A.CreaArbol(ai,e,ad))

fun balanceder Arbolvacio = Arbolvacio
 | balanceder(A.CreaArbol(ai,e,ad))=if (A.altura(ai)-A.altura(ad))<=1
                  then A.CreaArbol(ai,e,ad)
                  else if A.altura(A.izq(ai)) < A.altura(A.der(ai))
                       then rotader (A.CreaArbol(rotaizq(ai),e,ad))
                       else rotader (A.CreaArbol(ai,e,ad))


exception Minim
fun minim Arbolvacio = raise Minim
 |  minim (A.CreaArbol (ai,e,ad)) = if A.esVacio (ai)
                                    then e
                                    else minim (ai)

fun inserta (e,Arbolvacio) = A.CreaArbol(Arbolvacio,e,Arbolvacio)
 |  inserta (e,A.CreaArbol(ai,e1,ad)) = if menor (e,e1)
  then if A.altura(ai) <= A.altura(ad)
       then A.CreaArbol(inserta(e,ai),e1,ad)
       else if menor (e,A.raiz(ai))
           then rotader (A.CreaArbol(inserta(e,ai),e1,ad))
           else if menor (A.raiz(ai),e)
            then rotader (A.CreaArbol(rotaizq(inserta(e,ai)),e1,ad))
            else A.CreaArbol(ai,e1,ad)
  else if menor (e1,e)
       then if A.altura(ad) <= A.altura(ai)
            then A.CreaArbol (ai,e1,inserta(e,ad))
            else if menor (A.raiz(ad),e)
                 then rotaizq(A.CreaArbol (ai,e1,inserta(e,ad)))
                 else if menor(e,A.raiz(ad))
                      then rotaizq(A.CreaArbol(ai,e1,rotader(inserta
```

```
                              (e,ad))))
                         else A.CreaArbol(ai,e1,ad)
          else A.CreaArbol(ai,e,ad)
  fun sustrae (e,Arbolvacio) = Arbolvacio
    | sustrae (e,A.CreaArbol(ai,e1,ad)) = if menor (e,e1)
          then balanceizq(A.CreaArbol(sustrae(e,ai),e1,ad))
          else if menor (e1,e)
               then balanceder(A.CreaArbol(ai,e1,sustrae(e,ad)))
               else if A.esVacio (ai)
                         then ad
                         else if A.esVacio(ad)
                              then ai
                              else balanceizq(A.CreaArbol(sustrae
                                   (minim(ai),ai), minim(ai), ad))
end;

functor ArbolAVL ( E : ELEMORDEN ) : ARBOLAVL =
struct
structure S = ARBOLAVL (Arbolbin (E))
open S
end;



(*   archivo conjunto.sml:      archivo de conjuntos          *)
(*   debe llamarse antes a elementos.sml          *)
(*   poner use["elementos.sml"];                  *)

signature CONJUNTO =
sig
   (*  parametro  *)
   type elemento
   val iguales : (elemento * elemento) -> bool
   (*  genero que se define  *)
   datatype conj = ConjVacio | Inserta of (elemento * conj)
    (*     funciones auxiliares       *)
```

```
   val remover      : (elemento * conj) -> conj
   val pertenece    : (elemento * conj) -> bool
   val union: (conj * conj ) -> conj
   val interseccion : (conj* conj) -> conj
   val diferencia : (conj * conj) -> conj
   val tamano       : conj -> int
   val esVacio      : conj -> bool
end;


functor Conjunto (Elem :ELEMCOMPAR ): CONJUNTO =
struct
   type elemento = Elem.elemento
   val iguales = Elem.igual
   datatype conj = ConjVacio | Inserta of (elemento * conj)

   fun remover (e,ConjVacio) = ConjVacio
    | remover (e,Inserta(f,c)) = if Elem.igual (e,f) then remover(e,c)
                                 else Inserta(f,remover(e,c))

   fun pertenece (e,ConjVacio) = false
    | pertenece (e,Inserta(f,c)) = Elem.igual(e,f)
                                          orelse (pertenece(e,c))
   fun union (c,ConjVacio) = c
    | union ( c,Inserta(e,d)) = Inserta (e,union(c,d))

   fun interseccion (c,ConjVacio) = ConjVacio
    | interseccion ( c,Inserta(e,d)) = if pertenece (e,c)
                                 then Inserta(e,interseccion (c,d))
                                 else interseccion (c,d)

   fun diferencia (c,ConjVacio) = c
    | diferencia (c,Inserta(e,d)) = if pertenece (e,c)
                                 then remover (e,diferencia(c,d))
                                 else diferencia (c,d)
```

```
   fun tamano ConjVacio = 0
    | tamano (Inserta (e,c)) = 1 + tamano (remover (e,c))
   fun esVacio ConjVacio = true
    | esVacio _          = false
end;


(*   archivo bolsas.sml            *)
(*  requiere de conjunto.sml       *)
(*  poner use["conjunto.sml"];     *)


functor BOLSA (C :CONJUNTO ) =
struct
   structure Conj = C
   open Conj

   fun remover (e,ConjVacio ) = ConjVacio
     | remover (d,Inserta (e,b)) = if iguales (e,d) then b
                                   else Inserta (e,remover (d,b))

   fun interseccion (c, ConjVacio) = ConjVacio
    | interseccion (c,Inserta (e,b)) = if pertenece (e,b)
                          then Inserta(e,interseccion(remover(e,b),c))
                          else interseccion (b,c)
   fun tamano ConjVacio = 0
    | tamano (Inserta (e,b)) = 1 + tamano (b)
end;

functor Bolsa (E : ELEMCOMPAR ) =
struct
structure S = BOLSA ( Conjunto (E))
open S
end;
```

```
(*    archivo  graficas.sml          *)
(*   requiere llamar antes a elementos.sml          *)
(*   poner use["elementos.sml"];                    *)

signature GRAFICA =
sig
   (*   parametros    *)
   type nodo
   val igual : (nodo * nodo) -> bool
(*   tipo que se define  *)
   datatype grafica = GrafVacia | Insertanodo of (nodo * grafica)
                    | Insertaarista of (nodo * nodo* grafica)
   val borranodo : (nodo * grafica) -> grafica
   val borraarista : (nodo*nodo*grafica) -> grafica
   val adyacente : (nodo*nodo*grafica) -> bool
   val contienenodo : (nodo * grafica) -> bool
   val esVacia  : grafica -> bool
end;
functor Grafica (N: ELEMCOMPAR) : GRAFICA =
struct
   type nodo = N.elemento
   val igual = N.igual
   datatype grafica = GrafVacia | Insertanodo of (nodo * grafica)
                    | Insertaarista of (nodo*nodo*grafica)
   fun borranodo (n,GrafVacia) = GrafVacia
    |  borranodo (n,Insertanodo (m,g)) = if igual (n,m)
                                then borranodo(n,g)
                                else Insertanodo(m,borranodo(n,g))
    |  borranodo (n,Insertaarista (p,q,g)) = if igual (n,p)
                  then Insertanodo(q,borranodo(n,g))
                  else if igual (n,q)
                       then Insertanodo (p,borranodo (n,g))
                       else Insertaarista (p,q,borranodo(n,g))
   fun borraarista (n,m, GrafVacia) = GrafVacia
    | borraarista(n,m,Insertanodo(p,g))=Insertanodo(p,
                                                borraarista(n,m,g))
```

```
    | borraarista (n,m,Insertaarista(p,q,g)) = if (igual (n,p) andalso
                  igual(m,q)) orelse (igual (n,q) andalso igual(m,p))
                  then borraarista (n,m,g)
                  else Insertaarista (p,q,borraarista (n,m,g))
  fun adyacente (n,m,GrafVacia) = false
    | adyacente (n,m,Insertanodo(p,g)) = adyacente (n,m,g)
    | adyacente (n,m,Insertaarista(p,q,g)) = if (igual(n,p) andalso
                  igual(m,q)) orelse (igual (n,q) andalso igual(m,p))
                  then true
                  else adyacente (n,m,g)
  fun contienenodo (n,GrafVacia) = false
    | contienenodo (n,Insertanodo(p,g)) = if igual(n,p)
                                          then true
                                          else contienenodo (n,g)
    | contienenodo (n,Insertaarista(p,q,g)) = if igual(n,p)
                                              orelse igual(n,q)
                                              then true
                                              else contienenodo (n,g)

  fun esVacia GrafVacia = true
    | esVacia _         = false
end;


(*   archivo digraficas.sml:  son  graficas dirigidas     *)
(*    se requiere de llamar a graficas.sml       *)
(*   poner use["graficas.sml"];                  *)



functor DIGRAFICA (G : GRAFICA) =
struct
   structure graf = G
   open graf
   val igual = igual
   fun essucesor (n,m,GrafVacia) = false
```

```
    | essucesor(p,q,Insertanodo(n,g))=if igual(n,p) orelse igual(n,q)
                                      then true
                                      else essucesor (p,q,g)
    | essucesor (p,q,Insertaarista(n,m,g)) = if igual(n,q)
                                              andalso igual(m,p)
                                              then true
                                              else essucesor (p,q,g)
  fun espredecesor (n,m,GrafVacia) = false
    | espredecesor (p,q,Insertanodo(n,g)) = if igual(n,q)
                                            andalso igual(n,p)
                                            then true
                                            else espredecesor (p,q,g)
    | espredecesor (p,q,Insertaarista (n,m,g)) =
                              if igual(n,q) andalso igual (m,p)
                              then true
                              else espredecesor (p,q,g)
end;



functor Digrafica ( E : ELEMCOMPAR) =
struct
structure S = DIGRAFICA ( Grafica (E))
open S
end;



(*   archivo tablah.sml:   tablas   hash   *)
(*   no requiere nada      *)

signature DATOS =
sig
   type llave
   type dato
   val igual : (llave * llave) -> bool
```

```
end;

(*   se debe generar una estructura que satisfaga a DATOS
     antes de llamar a diccionario     *)

signature DICCIONARIO =
sig
(*   parametros      *)
   structure Da : DATOS
(*   definicion del tipo dic    *)
   type dicc
   val DiccVacio : dicc
   val Inserta   : (Da.dato * Da.llave * dicc) -> dicc
   val elimina   : (Da.llave * dicc) -> dicc
   val recupera : (Da.llave * dicc) -> Da.dato
   val pertenece: (Da.llave * dicc) -> bool
   val esVacio   : dicc -> bool
end;



functor Dicc (D : DATOS) : DICCIONARIO =
struct
(*   parametros     *)

structure Da = D
open Da
(*  definicion del tipo y funciones   *)
   datatype dicc = DiccVacio | Inserta of (dato * llave * dicc)
   exception Elimina
   fun elimina (j,DiccVcio) = raise Elimina
    |  elimina (j, Inserta (d,k,dic)) = if igual (j,k)
                          then elimina (k,dic)
                          else Inserta (d,k,elimina(j,dic))
   exception Recupera
```

```
   fun recupera (j,DiccVacio) = raise Recupera
    | recupera (j, Inserta (d,k,dic)) = if igual (j,k)
                        then d
                        else recupera (j,dic)
   fun pertenece (j,DiccVacio) = false
    | pertenece (j, Inserta (d,k,dic)) = if igual (j,k)
                        then true
                        else pertenece (j,dic)
   fun esVacio DiccVacio = true
    | esVacio _          = false
end;



signature ELEM =
sig
   type elem
end;



signature RANGO =
sig
   type rango
   val iguales : (rango * rango) -> bool
end;



signature TABLA =
sig
(*   parametros     *)
   structure E : ELEM
```

```
   structure R : RANGO
(*    definicion del tipo tabla    *)
   datatype tabla = TabVacia | Asigna of ( E.elem * R.rango * tabla)
   val contenido : (R.rango * tabla ) -> E.elem
end;


functor Tabla (El: ELEM,  Ra:RANGO) : TABLA =
struct
(*    parametros    *)
   structure E = El
   structure R = Ra
   open E
   open R
(*    definicion del tipo y operaciones    *)
   datatype tabla = TabVacia | Asigna of ( elem * rango * tabla)
   exception Contenido
   fun contenido (j,TabVacia) = raise Contenido
    | contenido (j,Asigna (e,k,t)) = if iguales (j,k)
                                then e
                                else contenido (j,t)
end;

signature FUNHASH =
sig
   structure D : DATOS
   structure R : RANGO
   val hash : D.llave -> R.rango
end;


signature TABHASH =
sig
   structure T : TABLA
   structure F : FUNHASH
   structure Di : DICCIONARIO
```

```
   val incluye : (T.E.elem * Di.Da.llave *T.tabla) -> T.tabla
   val excluye : ( Di.Da.llave * T.tabla) -> T.tabla
   val inf     : ( Di.Da.llave * T.tabla) -> T.E.elem
   val existe  : ( Di.Da.llave * T.tabla) -> bool
   val asignado: ( Di.Da.llave * T.tabla) -> bool
end;



functor Tabhash (Ta:TABLA, Dic:DICCIONARIO, Fu:FUNHASH sharing (* type
                 Ta.E.elem = Dic.dicc *)
                 structure Dic.Da = Fu.D ) :TABHASH =
struct
   structure T = Ta
open T
   structure Di = Dic
open Di
   structure F = Fu
   open F
   fun incluye (d,k,TabVacia) = Asigna(Inserta (d,k,DiccVacio),hash(k),
                                TabVacia)
    | incluye (d,k,Asigna (dic,i,t)) = if F.R.iguales (hash(k),i)
                        then Asigna (Inserta(d,k,dic),i,t)
                        else Asigna (dic,i,incluye (d,k,t))

   fun excluye (k,TabVacia) = TabVacia
    | excluye (k,Asigna(dic,i,t)) = if F.R.iguales (hash(k),i)
                        then Asigna (elimina(k,dic), i,t)
                        else Asigna(dic,i,excluye(k,t))

   exception Inf
   fun inf (k,TabVacia) = raise Inf
    | inf (k,Asigna (dic,i,t)) = if F.R.iguales (hash(k) ,i)
                        then recupera (k,dic)
                        else inf (k,t)
```

```
   fun existe (k, TabVacia) = false
    |  existe (k,Asigna (dic,i,t)) = if F.R.iguales (hash(k) , i)
                        then pertenece (k,dic)
                        else existe (k,t)

   fun asignado (k,TabVacia) = false
    |  asignado (k,Asigna (dic,i,t)) = if F.R.iguales (k,i)
                        then true
                        else asignado (k,t)
end;




signature COLISION =
sig
   structure F : FUNHASH
   val sigue : F.rango -> F.rango
end;



functor TabHColis ( T: TABLAHASH, C:COLISION sharing type
                    T.E = (C.F.llave * C.F.dato)) =
struct
  datatype T.elem =elem of  ( C.F.llave * C.F.dato)
   exception Anade
   fun anade ((d,k), i,t) = if iguales (hash(k),i)
                        then raise Anade
                        else if asignado (i,t)
                             then anade ((d,k),sigue(i),t)
                             else asigna ((d,k),i,t)
   exception Busca
```

```
  fun busca (k,i,t) = if iguales (hash(k),i) or not(asignado(i,t))
                      then  raise Busca
                      else if igual (#1(inf(i,t),k))
                           then #2(inf(i,t))
                           else busca(k,sigue(i),t)

  fun incluye ((d,k),t) = if asignado(hash(k),t)
                          then anade((d,k),sigue(hash(k)),t)
                          else asigna ((d,k),hash(k),t)

  exception Inf
  fun inf (k,t) = if not asignado(hash(k),t)
                  then raise Inf
                  else if igual (#1(inf(hash(k),t)),k)
                       then #2(inf(hash(k),t))
                       else busca (k,sigue(hash(k)),t)
end;
```

REFERENCIAS BIBLIOGRAFICAS:

Abbott 83 R. J. Abbott. Program Design by Informal English Descriptions. Communications of the ACM, Vol. 26 No. 11 Nov. 83

Bergstra 89 Bergstra, Heering, Klint .Algebraic Specifications. ACM Press 89.

Biggerstaff 84 T. Biggerstaff, A. Perlis. Forward to Special Issue on Software Reusability. IEEE Trans on Software Engineering Vol SE-10, No. 5 84.

Biggertaff 87 T. Biggerstaff, Ch. Richter. Reusability Framwork, Assessment and Directions. IEEE Sotware march 87.

Burstall 81 R.M. Burstall, J.A. Goguen. An Infornal Introduction to Specifications using Clear. Academic Press 81

Cardelli 85 L. Cardelli, P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. Computing Surveys Vol. 17 No. 4 85.

Drossopoulou 88 S. Drossopoulou, S. Eisenbach, L. McLonglin. Module and Type Systen (a tour). Imperial College Feb 88.

Ehrig 85H. Ehrig, B. Mahr. Fundamentals of algebraic specifications .Vol I.Equations and initial semantics. Springer Verlag. 1985.

Goguen 78 J.A. Goguen, J.W. Thatcher, E> G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. Currents trends in programming Methodology IV. R. Yeh (ed) Prentice Hall 80-149. 1978.

Goguen 84 J. Goguen . Parametrized Programming. IEEE Trans. on Software Engineering. Vol SE-10 No. 5 84.

Goguen 86 J. Goguen . Reusing and Interconnection Software Components. IEEE Computer feb. 86.

Guttag 75 J.V. Guttag. The Specification and Aplication to Programming of Abstract Data Types. Ph. D. Thesis University of Toronto. 75

Guttag 78 J. V. Guttag, J.J. Horning. The algebraic specification of abstract datatypes. Acta Informatica 10, p.27-52. 1978.

Harper 86 R. Harper, D. MacQueen, R. Milner. Standard ML. Departamento de Ciencias de la computación. Universidad de Edinburgo. ECS-LFCS 86-2, marzo 1986.

Harper 86 R. Harper. Introduction to standard ML. ECS-LFCS 86-14 nov 1986.LFCS 86-14. 1986.

Harrison 89 R. Harrison. Abstract Data Types in Modula-2. Wiley 89

Hudak 89 P. Hudak . Conception, Evolution and Aplication of Functional Programming Languages. ACM Computing Surveys Vol. 21 No. 3 89.

Lins 90 C. Lins. The Modula-2 Software Component Library. Springer Verlag Vols 1 a 4 90.

Lipson 81 J.D. Lipson. Elements of algebra and algebraic computing. Addison Wesley. 1981.

Liskov 74 B. Liskov, S. Zilles. * Programmimg with Abstract Data types. ACM SIGPLAN Notices abril 74.

MacQueen 90 D. MacQueen. A Higher-Order Type System for Functional Programming. Research topic in Functional Programming. Edited by David A. Turner. Addison Wesley 90.

Martin 86 J. J. Martin. Data types and data structures. Prentice Hall. 1986.

Oktaba 88 H. Oktaba, L. Espitia, G. Ibargüengoitia, C. Velarde. Especificación formal en el diseño de programas. Comunicaciones Técnicas IIMAS, serie azul No. 107. 1988.

Oktaba 90 H. Oktaba 90. Componentes Reusables de software y su especificaciön formal. Memorias del I Taller de Programaciön Avanzada y Metodologùas, CIMAT 90.

Sannella 84 D. Sannella, A. Tarlecki. Program Specification and Development in Standard ML. ACM 1984.

Sannella 86 D Sannella, A Terlecki. Lecture Notes on Categories Specifications and Institutions. Draft version sept 1986.

Sannella 88 D. Sannella, A. Tarlecki. Toward formal development of ML programs: foundations and Methodology. . Versión preliminar dic 1988.

Tofte 88 M. Tofte. Two lectures on ML modules. Departamento de Ciencias de la Computación.Universidad de Edinburgo. Dic 1988.

Tracz 90 W. Tracz. Where Does Reuse Start?. ACM SIGSOFT Software Engineering Notes. Vol. 15 No. 2 1990.

Turski 87 W M. Turski, T.S.E. Mainbaum. The specification of computer programs. Addison Wesley 1987.

Van Horebeek 89 J.L. Van Horebeek. Algebraic Specifications in Software Engineering. Springer Verlag 89.

Wikstrom 87 A. Wikstrom. Functional programming. Using standad ML. Prentice Hall. 1987.

Wing 90 J.M. Wing. "A Specifier"s Intriduction to Formal Methods. Computer sept 90.

Zilles 74 S.N. Zilles.Algebraic specifications of data types. Computation Structures Group. Memo 119. Laboratory for Computer Science MIT. 1974.