ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Ingeniero Técnico de Informática de Sistemas

GENERADOR AUTOMÁTICO DE INTERFACES GRÁFICOS DE USUARIO

Realizado por

Carlos Óscar Sánchez Sorzano

Dirigido por

Oswaldo Trelles Salazar

Departamento

Arquitectura de Computadores

UNIVERSIDAD DE MÁLAGA

MÁLAGA, 25 de febrero de 2.000

UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Ingeniero Técnico de Informática de Sistemas

Reunido el tribunal examinador en el de la fecha, constituido por:

Presidente D/Dª._____

Secretario D/Dª._____

Vocal D/Dª. _____

para juzgar el proyecto Fin de Carrera titulado:

GENERADOR AUTOMÁTICO DE INTERFACES GRÁFICOS DE USUARIO

del alumno D.: Carlos Óscar Sánchez Sorzano

dirigido por D.: Oswaldo Trelles Salazar

ACORDÓ POR _____ OTORGAR LA CALIFICACIÓN DE _____

Y PARA QUE CONSTE, SE EXTIENDE FIRMADA POR LOS COMPARECIENTES DEL TRIBUNAL, LA PRESENTE DILIGENCIA.

Málaga, a _____de _____de

El Presidente _____El Secretario_____ El Vocal_____

Fdo: _____Fdo:_____ Fdo:_____

¿De qué te sirve ganar el mundo
si pierdes tu alma?

A todos los que han hecho
que sea ahora la persona que soy,
en especial a mi familia, mis amigos,
todos los buenos y malos momentos
y en definitiva las personas
a las que tanto quiero.

Índice de contenido

Presentación.....5
Capítulo 1. Introducción.....6
1.1. Soluciones anteriores.....8
1.2. Especificaciones.....10
1.2.1. Estructura de los interfaces de línea de comando.....10
1.2.2. Requerimientos sobre los tipos de datos.....14
1.2.3. Requerimientos sobre la gramática.....15
Capítulo 2. Implementación.....16
2.1. Herramientas de programación.....16
2.2. Opciones de diseño.....18
2.2.1. Convención de nombres.....19
2.3. Gramáticas.....20
2.3.1. Gramática de entrada.....21
2.3.2. Gramática intermedia.....25
2.4. Estructura modular.....26
2.5. Módulos, funciones y clases.....28
2.5.1. Funciones generales.....28
2.5.2. Manejo de la tabla de símbolos.....31
2.5.3. Compiladores.....48
2.5.4. Generador de interfaz.....52
2.5.5. Manejo de la interfaz gráfica.....56
2.6. Resultados.....73
2.6.1. Módulo principal de la aplicación Colimate.....74
2.6.2. Menús compartidos.....77
2.6.3. Menús complejos.....80
2.6.4. Programas con distintos modos de funcionamiento.....85
Capítulo 3. Conclusiones.....91
Bibliografía.....92

Presentación

El presente proyecto se desarrolla en el seno de la Unidad de Biocomputación del Centro Nacional de Biotecnología (Consejo Superior de Investigaciones Científicas) en Madrid. Dicha unidad actúa en tres frentes diferentes. Por un lado, se dedica al estudio estructural de macromoléculas a partir de imágenes obtenidas con técnicas de microscopía electrónica, en concreto ahora mismo se centra en el trabajo con helicasas examéricas aunque también ha hecho otras aportaciones al campo de la biología estructural. Un segundo campo de acción es el desarrollo de algoritmos de procesamiento de imagen que faciliten dichos estudios estructurales, para ello se está trabajando activamente en algoritmos de clasificación de imágenes con redes neuronales y algoritmos iterativos de reconstrucción tridimensional a partir de proyecciones bidimensionales (tomografía). Por último, el grupo participa activamente en el establecimiento de bases de datos que recojan y organicen la inmensa cantidad de información estructural que actualmente se está generando a nivel mundial.

Es en este contexto de computación científica en el que surge la necesidad cubierta por el presente proyecto. Dada la marcada vocación internacional del grupo en el que se ha desarrollado, el proyecto no tiene más opción que ser escrito íntegramente en inglés. Al mismo tiempo, el carácter de servicio nacional de la unidad obliga a que dicho proyecto esté abierto públicamente y que el uso del mismo no se atenga más que al espíritu de licencias abiertas que suele imperar en la comunidad científica.

En este sentido se ha publicado el proyecto en Internet (http://www.cnb.uam.es/~bioinfo/Colimate/Extra_Docs) bajo el nombre de Colimate (The COmmand LIne MATE) y sirve de apoyo a Xmipp (X−Windows based Microscopy Image Processing Package), un paquete realizado también por el grupo.

Capítulo 1. Introducción

Tradicionalmente los entornos de investigación, caracterizados por una carga pesada de computación, se han desenvuelto a nivel de grandes estaciones de trabajo en red bajo alguna de las múltiples versiones del sistema operativo Unix. Es éste un sistema muy robusto que ofrece muchas ventajas a la hora de soportar una computación intensiva. Sin embargo, habitualmente el entorno Unix se basa, o por lo menos lo ha hecho hasta ahora, en un interfaz de usuario en modo texto. Las aplicaciones gráficas estaban muy orientadas a tareas concretas y a problemas específicos. De este modo, en entornos científicos los usuarios suelen interaccionar con los programas por medio de una línea de comandos por la que suministra todos los parámetros que le son necesarios al programa, normalmente una buena cantidad, para ejecutar un determinado algoritmo. Esta filosofía de interacción, sin embargo, tiene sus aspectos positivos como es una gran facilidad de ser integrado en procesamientos automáticos sin necesidad alguna de intervención de los usuarios.

Adicionalmente la mayor parte de los paquetes de computación en entornos de investigación no son concebidos desde el principio como un sistema global por medio de técnicas de ingeniería de sistemas, sino que la mayor parte de ellos se convierten en sistema *de facto* al ir creciendo en módulos independientes, normalmente realizados por investigadores diferentes y diferentes filosofías. Además, el carácter científico de los desarrolladores, la no comercialización de los productos y la falta de una fuente estable de recursos refuerzan la tendencia de dedicar esfuerzos casi exclusivamente a lo que es mejora e implementación de nuevos algoritmos, dejando un poco de lado el aspecto de interfaces de usuario.

A la discusión anterior hay que unir el hecho de que en dichos entornos hay una alta probabilidad de que el usuario de los algoritmos sea el propio programador de los mismos, con lo cual el tema de interfaz a posibles usuarios pierde aun más interés.

El resultado final es que si el conjunto de programas tiene éxito en la comunidad científica, el número de usuarios se multiplica y una cuestión que en

principio era secundaria como era la presentación de un interfaz amigable pasa a un plano más relevante. Desafortunadamente, la accesibilidad de los usuarios al grupo de desarrollo de los programas suele ser nula, y por otra parte, el esfuerzo que este equipo está dispuesto a dedicar a un tema puramente de presentación es más bien escaso. A esto hay que añadir el hecho de que a poco que se quiera, la realización de cualquier tipo de interfaz gráfica con las herramientas actuales de programación supone un esfuerzo importante, aunque empieza a ser el momento de romper una lanza en favor de numerosos entornos y librerías que procuran hacer estas tareas cada vez más sencillas.

El objetivo de este proyecto es el de rellenar de algún modo el vacío existente entre las herramientas de programación gráfica, los interfaces de línea de comando y los usuarios finales de los productos. Para ello, se pretende realizar un lenguaje empotrado en un lenguaje matriz que facilite al máximo a los desarrolladores de software científico, y en el extremo a los usuarios, la realización de sencillos pero eficaces interfaces gráficos sin perder de vista las ventajas de un interfaz de línea de comandos. La restricción a interfaces en líneas de comandos simplifica enormemente la variedad de interfaces, de este modo las posibilidades de cambio son muy limitadas y encontramos que la estructura de casi todas las líneas viene a ser más o menos parecida. La idea es, pues, que en el mismo programa en el que se realice la función deseada, se describa textualmente y del modo más simple posible cuál es dicha estructura. Esta especificación se compilará y como resultado de dicha compilación se dispondrá de una serie de ventanas y menús más aptos para la interacción hombre−máquina a nivel de usuarios que no tienen por qué estar muy relacionados con el mundo de los ordenadores y para los que los interfaces de línea producen ciertos sentimientos de aversión, sobre todo si los comparamos ahora con el tan extendido Windows. Nótese que estas ventanas aunque son definidas dentro del programa principal del algoritmo no pertenecen al mismo, sino que son abiertas por una aplicación externa que se basa en las descripciones empotradas que hemos hecho. Por medio de estas ventanas el usuario deberá proporcionar los parámetros adecuados para la ejecución del algoritmo en cuestión. Una vez facilitados deberá pulsar una tecla de ejecución, el equivalente al ENTER en el interfaz de línea, y la aplicación gestora de los menús se encarga de escribir y lanzar la línea de comandos deseada.

Creemos que la necesidad identificada queda de este modo cubierta y que se logra establecer un puente de unión entre una dedicación de esfuerzos mínimo a un aspecto que dentro de la comunidad científica no goza de demasiada consideración y la necesidad real de acercar los programas a un conjunto amplio de usuarios.

1.1. Soluciones anteriores

Antes de ponernos manos a la obra será necesario hacer una revisión de las soluciones que ya se han propuesto en el campo de los interfaces gráficos de usuario (GUI). Tras una búsqueda exhaustiva por Internet encontramos que efectivamente existen herramientas muy potentes de generación de gráficos que podríamos categorizar según la siguiente taxonomía:

1. **Librerías gráficas**: en general suelen ser potentes mecanismos de tratamiento de gráficos, con una visión muy amplia del concepto de interfaz, y propósito inespecífico. Es precisamente por esta versatilidad por lo que la programación en estos entornos se vuelve un tanto compleja y hay que controlar muchos parámetros. El resultado es que los programas se hacen difíciles de mantener, están sujetos a errores un tanto "esotéricos" y en el entorno científico tienen además una fuerte dependencia con el programador que los creó, de forma que cuando éste desaparece el programa no mantiene en lo sucesivo su interfaz.
   Sin embargo, no debemos llevarnos una visión negativa de estas librerías pues son el punto de referencia obligado a la hora de implementar cualquier interfaz. Las diferencias entre ellas radican principalmente en el lenguaje nativo que emplean para su definición, la encapsulación de datos y estructuras, comunicación entre diferentes componentes, filosofía de las aplicaciones, potencia de la funcionalidad ofrecida, extensiones y relaciones con otros estándares, el carácter abierto o comercial de la librería, organismo o empresa que la respalda, ...
   En esta categoría nos encontramos los grandes estándares de librerías gráficas (Qt, Tcl/Tk, Motif, X11, Microsoft Foundation Classes, GTK, OpenGL, ...) pero también se alinean librerías "envoltura" (wrappers) que tratan de simplificar el acceso a las funciones gráficas (OpenAmulet, EasyGTK, Gtk−−, KDE, Gnome,

Motif++, VDK, Fresco, WxWindows, Fox, SCO Visual TCL, ...)

2. **Generadores automáticos de código**: El objetivo de este segundo grupo de soluciones es radicalmente distinto al anterior. En esta familia se restringe el conjunto de interfaces abordables, pero la forma de crearlos es mucho más sencilla. La idea es que el programador define de forma gráfica cuáles son los elementos de que desea se componga una determinada ventana, y una vez hecho esto, una herramienta CASE genera un código válido en un lenguaje de programación (C++, C, ADA, ...) que implementa dicha estructura. Principalmente podríamos apuntar como desventaja de esta segunda solución el hecho de que nos tendríamos que atener a los componentes gráficos que proporcione la librería ya que el manejo de nuevas estructuras definidas por el usuario todavía es una materia incipiente en la que se tiene que avanzar mucho; y segundo, que aunque hemos disminuido en un grado cualitativo la complejidad de la programación aún ésta no se ha reducido lo suficiente para el nivel que se pretende puesto que el siguiente paso suele ser editar los ficheros generados automáticamente, normalmente complejos, insertar los trozos de código que nos interesen, y por último compilarlo todo junto.

   En esta segunda familia de soluciones podríamos mencionar Lxb, Andrew, Xforms, QtArch, Ez, Glade, ... De nuevo conviene resaltar la extrema utilidad de dichas herramientas CASE en el prototipado rápido de aplicaciones. Sin embargo, no es ése el asunto que nos ocupa en este caso.

3. **Lenguajes híbridos**: un último conjunto de soluciones estaría formado por una posición intermedia entre los dos grupos anteriores. Así nos encontramos con lenguajes extendidos a una funcionalidad gráfica, con un entorno de desarrollo estilo a las herramientas CASE anteriores pero con características de lenguaje, de hecho, dieron lugar a una nueva filosofía de programación como fue la programación orientada a eventos. Este es el caso de los conocidos Visual Basic y Visual C++ o de sus precursores como el CVI, y en el lado Unix, Xview.

Para concluir, podríamos decir que ninguna de las soluciones propuesta hasta ahora cumple con las especificaciones de nuestra situación problema: extrema sencillez en la programación con la idea de no depender fuertemente de algún programador determinado, disminuir al máximo la etapa de aprendizaje, y orientación específica a servir únicamente como un interfaz más amigable de las líneas de comando. Será este último requisito el que precisamente nos lleve a la consecución del primero. Visto de un modo simplista podríamos expresarlo como que cuando la funcionalidad y la versatilidad a cubrir aumenta, también lo hace la complejidad del lenguaje a utilizar. El habernos restringido al campo concreto de los interfaces de línea de comando será la clave que distinga la aplicación realizada en este proyecto frente a las respuestas que ya se han dado en el terreno de los interfaces gráficos de usuario.

1.2. Especificaciones

En esta sección describiremos la funcionalidad a cubrir por el presente proyecto fin de carrera, daremos las especificaciones técnicas que luego serán plasmadas en código y estructuras de datos concretas. Comenzaremos por definir la estructuración global de los interfaces a cubrir y los tipos básicos de datos que necesitaremos, finalmente revisaremos las necesidades expresivas de la gramática.

1.2.1. Estructura de los interfaces de línea de comando

Básicamente podemos reducir la estructura de una línea de comando a un programa que se llama con algunos parámetros (en notación BNF):

<command_line> ::= <exec_prog> {<arg>}

Sin embargo, esta definición de línea de comando es muy general, podemos concretar algo más sobre los argumentos que se emplean. De tal forma que podríamos declararlos como

<arg> ::= <param> | <flag> | <flag> <param>

<flag> ::= (”−”|”−−|”"/") <id> [“=”]

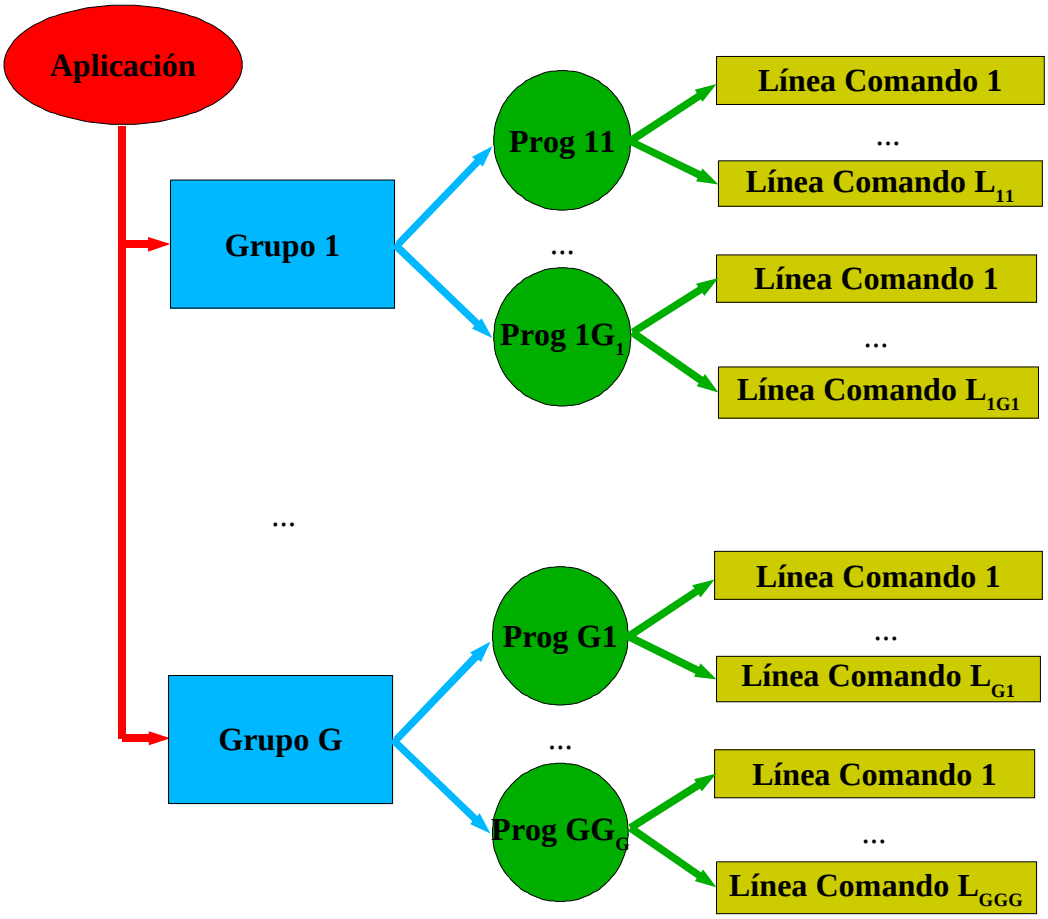
<param> ::= <id> | <quoted text> | <number> | <filename>

donde flag recoge el concepto de las opciones de línea de comando que comienzan por “−, por“−−, o por /”seguidos de algún identificador (no es el propósito de esta sección el de dar una definición exhaustiva de la gramática por lo que nos conformaremos por el momento con la concepción estándar de identificador). Por otro lado, los parámetros serán por regla general un identificador, un texto entrecomillado, un número, un fichero o patrón de nombre de fichero.

Así, una línea de comando o línea a parsear se modela como una lista de tokens, algunos tokens son invariantes, es decir, siempre son los mismos y el usuario no tiene ninguna capacidad para modificarlos (por ejemplo, el nombre del programa o un determinado flag), mientras que otros son variables. Un token variable representa un parámetro introducido por el usuario que debe proporcionar el valor adecuado en cada ejecución del comando.

Por otra parte, podemos distinguir dentro de un mismo programa diferentes modos de funcionamiento, en las que a cada uno corresponda un conjunto de parámetros distinto. Cada uno de estos modos será asociado con una línea a parsear. Al mismo tiempo, habrá programas que podamos agrupar dentro de una funcionalidad parecida. Así nace el concepto de grupo. Al conjunto de grupos lo llamaremos aplicación, que es en última instancia, el elemento al que pretendemos dotar de interfaz gráfico.

Para resumir, podemos decir que nuestro objetivo es el de dotar de un interfaz gráfico adecuado a una aplicación, que se compone de varios grupos de programas, cada programa a su vez puede tener diferentes tipos de comportamientos para los que la línea a parsear en cada caso será distinta. La siguiente figura representa dicha estructura.



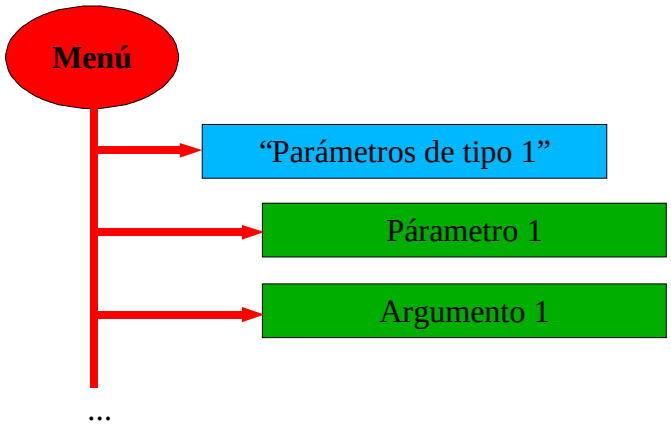
Un aspecto todavía no considerado hasta este punto es cómo se solicitarán los parámetros al usuario. Dentro de la estructura de la línea de comandos presentada la única capacidad de maniobra de que dispone el usuario es la posibilidad de introducir diferentes valores para los parámetros variables, y la elección o no de los argumentos opcionales. De este modo, el interfaz a desarrollar deberá de algún modo guiar al usuario a través de la estructura de árbol mostrada con anterioridad hasta que seleccione una determinada línea de comando. Una vez allí deberá preguntarle cuáles de los argumentos opcionales desea incluir y cuáles son los valores para las variables. Estos interfaces forman entidades independientes llamadas menús.

Así, pues, un menú es el interfaz específico que se presenta al usuario para la obtención concreta de valores y argumentos opcionales orientada a la generación automática de la línea de comando.

La estructura más sencilla en la que podemos modelar un menú es como una lista de variables y argumentos opcionales. Sin embargo, de alguna forma tendremos que separar un tipo de parámetros de otros por lo que los menús admitirán también textos fijos que nos indiquen la clase de parámetros que estamos introduciendo. Aunque no atañe directamente al modelo de menú, es conveniente resaltar llegado este punto, que deberá existir un mecanismo que permita habilitar y deshabilitar todos los parámetros que dependan de un determinado argumento opcional. Por lo tanto, nos quedaremos con la idea de que un menú es una serie de argumentos, parámetros y separadores uno debajo de otro.

Una característica adicional de los menús será que deberán ser compartidos entre las diversas líneas de comando de un mismo programa, y que eventualmente podrán ser compartidos por diferentes programas. Así, se deberá implementar un dispositivo automático que deshabilite aquellos parámetros y argumentos que aun siendo mostrados por el menu porque pertenecen a él, sin embargo, no participan en la línea de comando actual seleccionada por el usuario. O dicho de otro modo, de todos los parámetros y argumentos del menu, sólo se debe permitir al usuario actuar sobre aquellos que afectan directamente a la línea de comando que actualmente ha sido seleccionada.

La siguiente figura muestra esquemáticamente un ejemplo de menú.



Adicionalmente habrá que definir un modo que permita asociar de alguna forma los parámetros necesitados por un programa con los mostrados en un menú.

1.2.2. Requerimientos sobre los tipos de datos

En esta sección concretaremos algunos requerimientos adicionales sobre los tipos de datos y estructuras soportados.

Deseamos que los parámetros variables posean una serie de características, a saber:

- Etiqueta: que sea la que realmente aparezca en el interfaz
- Ayuda on−line: a petición del usuario
- Valores por defecto: en caso de que el parámetro así lo requiera.
- Control de validez del valor: la validez de un parámetro depende de su tipo, por ejemplo, para un argumento numérico la validez consistiría en un control de tipo y rango, mientras que para un fichero podría ser el control de existencia o no existencia del fichero en disco.

Los parámetros variables podrán ser de algunos de los siguientes tipos:

- **Textuales**: de longitud máxima fijada o libre
- **Numéricos**: naturales, enteros o reales. Opcionalmente se podrá limitar su valor a alguno de los siguientes rangos [a,b], [a,∞) ó (∞, b], donde a y b son dos valores reales.
- **Fichero**: nombre de fichero o patrón de nombres de fichero que incluyan los caracteres comodín “?” o “*” con su significado habitual en los sistemas operativos. En el caso de tratarse de un fichero simple se desea poder ejercer un control sobre la existencia o no del fichero, así por ejemplo, se podrán requerir ficheros que efectivamente existan.
- **Lista**: es éste un tipo de parámetro un tanto especial ya que no producirá texto a la hora de generar la línea de comando a ejecutar. Sin embargo, las variables tipo lista llevan asociadas una lista de acciones, que en principio estarán orientadas a modificar el valor de otras variables. Es decir, cuando se seleccione un elemento de estas listas se podrá asignar a otra variable del menú un valor prefijado o el contenido de otra variable.

Por último, deseamos que cuando el parámetro introducido en pantalla sea un fichero se puedan realizar sobre él operaciones muy sencillas como por ejemplo llamar a otros programas del sistema, externos a la aplicación definida, con este fichero como parámetro. Ejemplos de estas operaciones podrían ser edición, visualización como imagen, visualización como fichero PostScript, ...

1.2.3. Requerimientos sobre la gramática

Se desea que la sintaxis del lenguaje con el que se definan los interfaces sea lo más simple posible, pero al mismo tiempo que no se encasille ni ponga propias limitaciones a futuras ampliaciones.

Por otro lado, necesitaremos que la gramática definida permita que un argumento opcional se componga de varios parámetros e incluso de otros argumentos opcionales, es decir, que el anidamiento en los argumentos opcionales sea contemplado por la gramática y la estructura de datos posterior para la línea de comandos.

El lenguaje definido estará empotrado (embedded) en uno anfitrión (host) que en este caso será C de tal manera que el código para generar la estructura de menús sea transparente a un compilador de C normal.

Además se debe poder incluir documentación (comentarios) del programador en el propio código de generación de menús e incluir otros ficheros fuentes donde se definan otras partes de la aplicación.

Capítulo 2. Implementación

En este capítulo comenzaremos analizando las herramientas de programación utilizadas en el desarrollo del presente proyecto. A continuación comentaremos los aspectos más relevantes tanto de la estructura modular, estructuras de datos definidas, y la gramática a compilar. Una descripción pormenorizada de todos estos aspectos se puede encontrar en la ayuda y páginas de definición hipertextuales que complementan a la presente memoria.

2.1. Herramientas de programación

Debemos seleccionar el lenguaje para codificar el proyecto que satisfaga en mayor medida el siguiente conjunto de condiciones: potencia de expresividad, optimalidad de ejecución, extensamente portable, amplia documentación, bien soportado por librerías externas, a ser posible de dominio público, y en el que la interacción con herramientas que faciliten el diseño de compiladores sea relativamente sencilla. Tanto C como C++ cumplen el conjunto de requisitos, sin embargo, nos hemos decantado por un desarrollo en C++ debido a la flexibilidad del paradigma de orientación a objetos de la que podemos obtener beneficios adicionales.

El requerimiento de portabilidad y dominio público es fácilmente conseguible ateniéndonos al compilador de GNU, nuestro programa será portable a todas las máquinas y sistemas a los que el compilador de C++ de GNU haya sido portado. Además, es éste un compilador tan extendido que la mayoría de las librerías y otras herramientas de programación procuran ser compatibles con él.

Una vez decidido que trabajaremos en el C++ de GNU, podremos usar las versiones de LEX y YACC de GNU para C++ que se llaman FLEX++ y BISON++. Como principales diferencias respecto a LEX y YACC podemos resaltar una mayor expresividad de los analizadores léxicos y sintácticos, mayor flexibilidad para el programador y que en las nuevas versiones los analizadores son objetos de una clase que tiene el comportamiento del analizador requerido, la comunicación entre el

analizador léxico y el sintáctico ya no se realiza por medio de constantes de compilación sino a través de una clase enumerada perteneciente al analizador sintáctico. Esta filosofía permite, por ejemplo, tener varios analizadores independientes de una forma muy sencilla.

Respecto a la librería gráfica que emplearemos nos hemos decidido por Qt, primero, por ser uno de los grandes estándares en el mundo de los interfaces gráficos de usuario (por ejemplo, el extendido KDE es uno de los proyectos donde se emplea Qt como librería base); segundo, porque es una librería realmente potente y robusta; por último, tiene una filosofía de programación y una estructura interna muy lógica y está desarrollado íntegramente en C++ con lo que su integración con más código en C++ es inmediata. Por otro lado, Qt está sostenido por una empresa, sin embargo, es de libre distribución en entornos Unix y está portado a Windows, aunque esta última versión sí que no es gratis.

Por el hecho de trabajar en C++ podemos beneficiarnos de el uso de otras librerías como la STL (Standard Template Library). Es una librería de propósito general que da soporte a un amplio espectro de tipos de datos y algoritmos básicos distintos. Actualmente, dicha librería forma parte del propio compilador del GNU por lo que su instalación y uso son bien sencillos.

Para terminar, mencionaremos el empleo de una herramienta de documentación automática llamada DOC++. Es ésta una herramienta que permite la inclusión de comentarios especiales insertos en los propios programas de C++ tales que dichos comentarios definen una documentación estructurada del código escrito en forma de páginas hipertextuales. De este modo, es muy sencillo generar una documentación organizada y de calidad inserta en el propio código.

Resumiendo, el conjunto de herramientas a utilizar a lo largo del proyecto será: compilador de C++ de GNU, Lex++, Bison++, Qt, STL y DOC++. En la página web de distribución de Colimate se proporcionan enlaces para la consecución de cualquiera de estas herramientas.

2.2. Opciones de diseño

En esta sección veremos como Colimate cumple con algunos requisitos del sistema a diseñar tales como el empotramiento en un lenguaje matriz, la posibilidad de inclusión de código procedente de otro fichero o como definir comentarios dentro de Colimate.

El código para Colimate se encapsulará dentro de comentarios en C. Sin embargo, para distinguir las directivas para Colimate del resto de comentarios normales del programa los comentarios de Colimate comenzarán por "***/*Colimate:***", nada que esté fuera de este tipo de comentarios será procesado para la generación de menús. De ahora en adelante, todo lo que se diga respecto a la gramática a procesar se entenderá que está incluido dentro de un comentario en C válido para Colimate.

Como opción de diseño se ha pretendido que en la medida de lo posible la gramática de Colimate recuerde a la de C o C++. Así, por ejemplo, la inclusión de código definido en otros ficheros se hace por medio de la directiva

#include "ruta/fichero"

y los comentarios de Colimate tienen el mismo comienzo que los comentarios cortos de C++, "//".

El siguiente código ejemplifica lo que hasta ahora se ha definido de la gramática, podría ser perfectamente la distribución clásica de un menú Colimate inserto en un programa en C.

```
/* Esto es parte de un programa en C normal */
...
int main(int argc, char **argv) {
...
}

/*Colimate:
      // Incluye otro fichero
      #include “"otro_fichero"”

      // Continua definiendo el menu
      ...
*/
```

Para terminar de dar una pincelada sobre el estilo de Colimate diremos que el menú a definir está íntegramente contenido en los comentarios Colimate correspondientes. Sin embargo, estos comentarios pueden estar empotrados en el código de los propios programas a los que sirven de interfaz en caso de disponer sus fuentes. En caso contrario, simplemente escribiremos la especificación Colimate en un fichero aparte con un único comentario en C.

2.2.1. Convención de nombres

Antes de pasar adelante comentaremos la convención de notación empleada a lo largo de todo el proyecto. En principio, una convención coherente de nombres es útil porque facilita la lectura del código ya que simplemente viendo la tipografía del identificador sabemos si corresponde a un parámetro de la función, a un miembro de una clase, a un método, una variable, una macro, ... Además, si diseñamos nuestros identificadores con cierto arte evitaremos colisiones con otros identificadores particulares de cada sistema.

Comenzaremos por decir que todas las funciones, clases y estructuras de datos que se definan en el programa deberán empezar por la letra M de menús. Así, si queremos redefinir la función atoi de C, la nueva función se llamará Matoi. Vemos en el ejemplo una ventaja muy clara de esta notación, no perdemos el acceso a las

funciones originales. Una segunda es que es improbable que haya símbolos en el sistema que comiencen con la letra M. En este sentido, se añade la restricción de que todos los argumentos de funciones tengan como mínimo dos caracteres alfanuméricos, hay algunos símbolos definidos en algunos sistemas, como IRIX, que se llaman "_P" por ejemplo.

La siguiente tabla resume la tipografía empleada en cada clase de identificador.

Clase de identificador	***Ejemplo tipográfico***	***Descripción tipográfica***
Macro	MMACRO_DEF	Comienza por M y todas las letras son mayúsculas
Funciones independientes	Mfuncion_def	Comienza por M y todas las letras son minúsculas
Clases y Estructuras de datos	MClase_Def	Comienza por M y las primeras letras de cada palabra son mayúsculas el resto minúsculas
Métodos de clase	metodo_def	Todas las letras minúsculas
Argumentos	_arg_def	Todas las letras minúsculas, comienza por "_"
Miembros de clase	__miembro_def	Todas las letras minúsculas, comienza por "__"

2.3. Gramáticas

Nos referiremos como gramática de entrada a la gramática que deben usar los programadores para la definición de los interfaces. Se introduce esta apreciación en oposición a la gramática intermedia que es la que usa el compilador de Colimate para comunicarse con el generador de interfaces, esta última gramática se definirá más adelante aunque su estudio carece de valor práctico.

2.3.1. Gramática de entrada

La gramática de entrada, pues, debe reflejar los requerimientos sobre la gramática expresados en el apartado de Funcionalidad. Utilizaremos en esta sección notación de expresiones regulares, BNF y en algún caso en que la notación se simplifique un híbrido entre ambos. Una revisión exhaustiva de dichas notaciones se puede encontrar en [1].

Comenzaremos por definir los elementos más simples de la gramática que luego nos servirán a modo de ladrillos elementales con los que construir finalmente el lenguaje implementado. A continuación se muestra en notación BNF dichas definiciones, a menudo la parte derecha de la definición se encuentra como una expresión regular debido a que para algunas expresiones dicha notación es considerablemente más compacta y que el único objetivo de esta sección es el de transmitir la potencialidad del lenguaje.

```
<id>          ::= [a-zA-Z_][a-zA-Z0-9_]*
<var>         ::= "$"<id>
<flag>        ::= ["-""--""/"]<id>"="?
<small_text>  ::= \"[^\"\n]*\"
<text>        ::= \"[^\"]*\"
<int>         ::= -?[1-9][0-9]* | -?[0-9]
<float>       ::= -?([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?
<comment>     ::= "//"[^\n]

<group_name>  ::= <id>
<program_name>::= <id>
<menu_name>   ::= <id>
<line_name>   ::= <id>
<exec_prog>   ::= <id>
<number>      ::= <int> | <float>
```

Las definiciones anteriores no responden sino a los conceptos intuitivos y estándares de identificador, número entero, real, texto entrecomillado en una sola línea (small_text) y en varias líneas (text), los comentarios son del estilo de los de C++, comienzan por "//" y no ocupan más de una línea, las variables son

identificadores que comienzan por el carácter "$" y los flags para la línea de comando comienzan por "−", "−−" o "/" y pueden terminar opcionalmente en un carácter "=".

Excluyendo los comentarios que son ignorados y las sentencias de inclusión de otro fichero que son manejadas de forma diferente, podríamos definir la gramática de entrada como la siguiente (enteramente en notación BNF):

```
<Application> ::= {<Appl_def>} [<rightbutton_def>]
<Appl_def>    ::= <group_def> | <program_def> | <menu_def>

<group_def>   ::= GROUP <group_name> "{"
                     {PROGRAM <program_name> ;}
                  "}"

<program_def> ::= PROGRAM <program_name> "{"
                     OPEN MENU <menu_name> ;
                     COMMAND LINES "{"
                        <command_line> {<command_line>}
                     "}"
                    [PARAMETER DEFINITIONS "{"
                        <prm_def> {<prm_def>}
                     "}"]
                  "}"

<command_line>::= "+" <line_name> ":" <exec_prog> <arg_list>
                      ["<" <valued_arg>] [">" <valued_arg>]
<arg_list>    ::= {<arg> | "[" <arg_list> "]"}
<arg>         ::= <flag> | <valued_arg>
<valued_arg>  ::= <var>  | <id>

<prm_def>     ::= <var> "{"
                     LABEL      "=" <small_text> ";"
                    [TYPE       "=" <type_def> ";"]
                    [HELP       "=" <text> ";"]
                    [BY DEFAULT "=" <small_text> ";"]
                  "}"
```

```
<type_def>    ::=   TEXT    [MAXLENGTH "=" <int>]
                  | NATURAL [<range>]
                  | INTEGER [<range>]
                  | FLOAT   [<range>]
                  | FILE    [EXISTING | NON EXISTING | PATTERN]
                  | LIST "{"
                       {<small_text> ["{" {<action>} "}"]}
                    "}"

<range>       ::= "[" [<number>] "..." [<number>] "]"
<action>      ::= <var> "=" (<small_text> | <var>) ";"

<menu_def>    ::= MENU <menu_name> "{"
                     {<var> | <small_text>}
                  "}"

<rightbutton_def> ::= RIGHTBUTTON "{"
                        <command_line> {<command_line>}
                      "}"
```

El objetivo es construir una aplicación, la cual se compone básicamente de definiciones de grupos, programas y menús más una definición opcional de cómo tratar el botón derecho del ratón cuando nos encontramos sobre parámetros de tipo fichero. En líneas generales, se puede decir que se ha adoptado un estilo de lenguaje que recuerda en cierto modo a C++, esto es así con la intención de disminuir aun más la curva de aprendizaje de Colimate.

Para especificar un grupo basta con nombrar los programas que pertenecen a él (me remito a la modelización de las líneas de comando realizada en la sección 2.1 y a los requerimientos sobre la gramática para una justificación de la que aquí se muestra). Aunque todos los elementos del grupo actualmente son programas, hemos introducido la etiqueta PROGRAM delante, lo cual nos permite futuras ampliaciones sin tener que modificar interfaces que estuvieran ya definidos en una versión anterior de Colimate.

Los programas son algo más complejos de definir puesto que suponen la práctica totalidad de la información sobre el interfaz. En ellos tenemos que especificar a qué menú se debe llamar, cuáles son las diferentes líneas de comando posibles y definiremos los distintos parámetros.

Las líneas de comando permiten el anidamiento de parámetros opcionales con un mecanismo muy simple de anidación de corchetes. Al mismo tiempo, se permite la utilización de los mecanismos de redirección de entrada y salida soportados tanto por Unix como por MS−DOS.

En cuanto a la definición de parámetros sólo hay un campo obligatorio, el de etiqueta, el resto son opcionales, gramaticalmente hablando, es decir, el tipo es obligatorio en todos los parámetros a menos que sean de tipo opcional. En este caso, Colimate puede determinar automáticamente el tipo.

Dependiendo del tipo de parámetro emplearemos unos modificadores u otros. Salvo para el tipo lista, que explicaremos más adelante, el resto de los tipos mantiene una correlación bastante evidente entre la gramática y los requerimientos sobre los tipos de datos definidos en la sección 2.2.

Las listas se han modelado como una etiqueta más una lista de acciones entre llaves, el único tipo de acciones considerado es la asignación del valor de las variables. La idea es que al seleccionar alguna de las etiquetas se ejecute la lista de acciones asociada.

La sintaxis de los menús está relacionada directamente con el modelo de interfaz considerado. Por último, la posibilidad de realizar operaciones básicas sobre parámetros de tipo fichero se ha implementado por medio de la definición de un programa especial llamado "Rightbutton" para el que no hace falta el prefijo PROGRAM y en el que la única variable permitida es $FILE. Este tipo de controles semánticos no se muestran directamente en la gramática ya que la idea no es dar una notación exacta de la gramática sino una impresión de lo que se puede hacer y no. Sin embargo, aunque no esté explicitado en la gramática estos controles semánticos están

perfectamente documentados en las páginas de ayuda y en esta memoria.

2.3.2. Gramática intermedia

La gramática intermedia es la que utiliza el compilador de menús para comunicarse con el generador de interfaces a través de un fichero intermedio en el que se encuentran todos los resultados de la compilación. En realidad se trata tan sólo de la impresión de la tabla de símbolos generada en tiempo de compilación de tal forma que una vez compilada la aplicación gráfica se pueda ejecutar sin necesidad de volver a realizar una lectura de una fuente que puede ser errónea o estar corrupta. De este modo, estamos independizando la ejecución de la aplicación con su desarrollo, esto conlleva claras ventajas.

El fichero de comunicación entre los dos procesos se ha escrito en texto puro y con una sintaxis legible desde el punto de vista humano. Así facilitamos una alternativa a la detección de errores de código desde el punto de vista de Colimate. Sin embargo, este fichero de comunicación no debería ser manipulado en ningún caso puesto que el generador de interfaces asume su corrección y no implementa mecanismo alguno de protección contra errores.

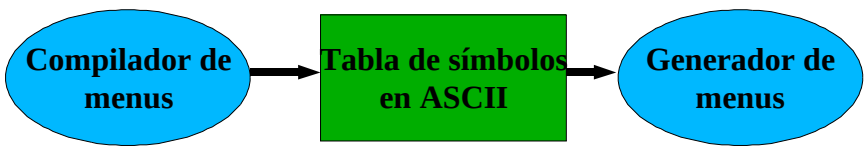
El estudio detallado de la gramática a este nivel carece de interés puesto que nunca nadie tendrá que escribir en ella, y para comprenderla basta con que sepamos que se muestran todas las tablas de símbolos (grupos, programas, variables y menús). Ya veremos más adelante cómo se construyen dichas tablas, con todos sus elementos y campos.

Por lo tanto, la idea que debemos quedarnos de esta sección es que necesitaremos un segundo compilador, más sencillo que el primero, que interprete la tabla de símbolos escrita en ASCII y la provea como una representación interna al generador de menús.

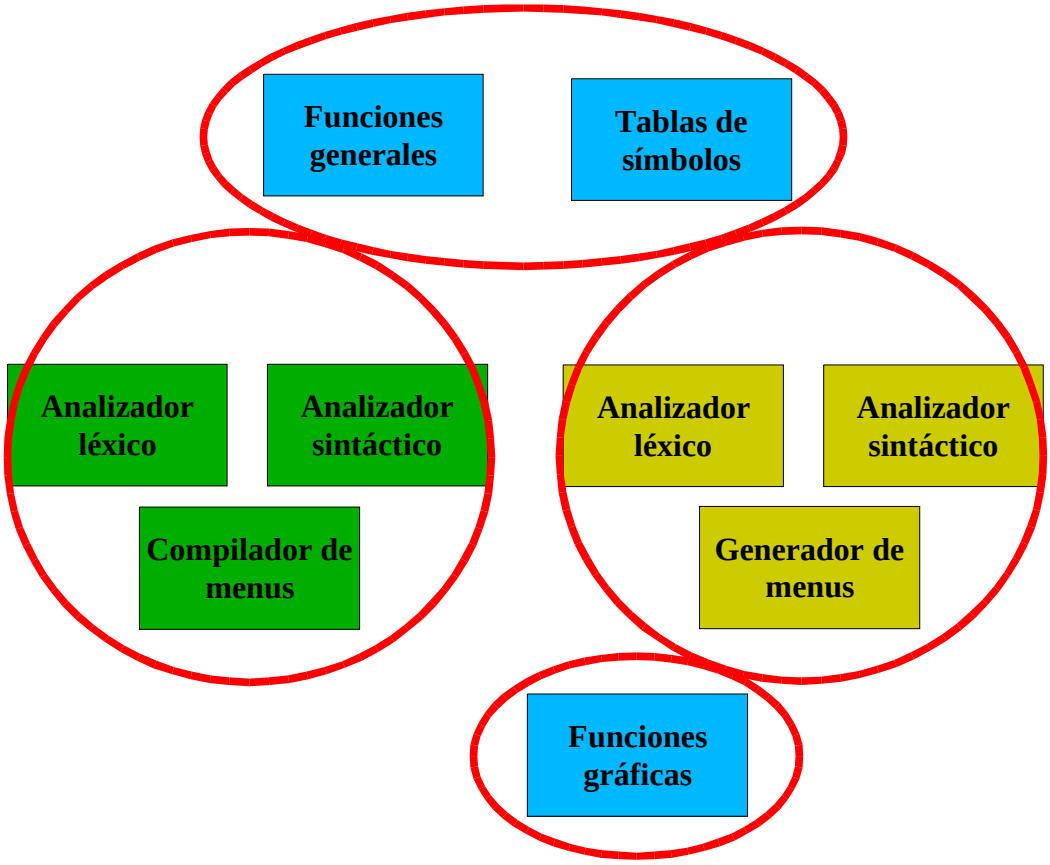
2.4. Estructura modular

En la funcionalidad a cubrir se distinguen claramente dos tareas: por un lado, la interpretación de la entrada textual codificada en forma de comentarios, y por otro, la generación de las ventanas con los componentes gráficos necesarios para la solicitud de los parámetros y posterior producción de la línea de comando correspondiente. Si acoplásemos ambas tareas en un único proceso tendríamos que cada vez que un usuario deseara abrir el interfaz de la aplicación, Colimate compilaría todos los fuentes en busca de la definición de dicho interfaz. Fuera de la pérdida de tiempo que esto supone, que con los ordenadores actuales no sería gran cosa, el gran inconveniente de esta aproximación es que se necesitarían distribuir los fuentes de la aplicación gráfica, lo cual no es siempre factible, y que además este proceso está mucho más sujeto a modificaciones erróneas posteriores del código que impidan la apertura de dicho interfaz.

Por ello, hemos decidido independizar las dos tareas. Por un lado, tendremos un compilador de menús que leerá el texto directamente escrito en los ficheros fuentes y generará un fichero intermedio de comunicación con el siguiente módulo de Colimate; por otro, un generador de menús que leerá el fichero intermedio anterior y abrirá el interfaz correspondiente. Este fichero intermedio se puede salvar fácilmente y constituye todo lo que hace falta para la generación de menús. De esta manera, hemos soslayado de forma muy sencilla los inconvenientes que detectábamos previamente. El fichero intermedio de intercambio de información no es otra cosa que la tabla de símbolos generada por el compilador escrita en forma de texto ASCII comprensible por un humano. Así facilitamos, incluso, la inspección visual por los programadores en caso de que el interfaz no se estuviera generando correctamente.



Hasta este punto hemos dado una estructura muy general de lo que es el proyecto. Ahora podríamos concretarla algo más y especificar en módulos de menor nivel cuáles son las tareas a desarrollar, prácticamente cada módulo de los que ahora se indiquen se define en un fichero de implementación (.cc) más otro de cabecero (.hh). Atenderemos en primer lugar a un pequeño diagrama que establece las relaciones entre módulos y el número de éstos, más tarde se irán detallando la funcionalidad de cada uno de ellos.



En este diagrama vemos que hemos desdoblado el compilador y generador de menús en submódulos, al mismo tiempo que hemos añadido pequeños subsistemas de apoyo. En cuanto a las estructuras del compilador y generador son muy similares entre sí: ambas poseen un analizador léxico, otro sintáctico y un elemento integrador que combina ambas funciones. En el caso de del compilador de menús, el texto a reconocer es la descripción del interfaz dada por el programador de la aplicación; mientras que para el generador será la tabla de símbolos intermedia.

Las funciones de apoyo a estos dos grandes bloques se pueden dividir en:

- **Funciones generales**: donde se engloban funciones de conversión de tipos (de formatos numéricos a cadenas y viceversa), eliminación de espacios superfluos, eliminación de comillas en una cadena, gestión de errores, ...
- **Tablas de símbolos**: en este módulo se implementan todas las funciones relacionadas con la gestión de la tabla de símbolos asociada al interfaz definido.
- **Funciones gráficas**: aquí se encuentran aquellas funciones relacionadas con la interacción con Qt, apertura de ventanas, componentes gráficos, ...

Una descripción más detallada de cada módulo se puede encontrar en la siguiente sección.

2.5. Módulos, funciones y clases

En este apartado estudiaremos detalladamente cada uno de los módulos anteriores viendo a grandes rasgos su estructura, filosofía, clases y relación entre ellas. Una descripción pormenorizada de cada clase, cada método y cada función se puede encontrar en las páginas de ayuda hipertextual que complementan a la presente memoria. A lo largo de la sección iremos mostrando ejemplos que ilustren el uso de los conceptos introducidos. Aquellas funciones que estén definidas en Colimate y que se pretenden mostrar estarán siempre en negrita.

2.5.1. Funciones generales

En este módulo se han incluido algunas funciones de propósito inespecífico que complementan algunas limitaciones de C. Bajo el nombre de funciones generales se encierra el manejo de errores, la definición de algunas constantes y macros útiles, la conversión de tipos y el manejo de argumentos provenientes de la línea de comando.

Manejo de errores

El manejo de errores implementado en Colimate se basa en el concepto de excepción de C++. Muy rápidamente podríamos resumirlo en lo siguiente: cuando se realiza una operación inválida una función lanza una excepción, si hay alguna función dentro de la cuál se haya llamado a la que lanza la excepción que contemple la posibilidad de capturar excepciones, entonces la excepción es capturada y se le da el tratamiento definido por la captura. En caso de que nadie recoja la excepción el programa termina volcando el estado actual de la máquina.

Colimate define la clase *MError* que contiene un código de error que especifica el tipo de error producido, más un mensaje que informa sobre quién produjo el error, qué error en concreto es y, a ser posible, sobre qué dato se produjo el error. En la documentación de hipertexto de Colimate se da una tabla con todos los tipos de errores, sus códigos y significado. A continuación se da un ejemplo en el que se emplean las clases, funciones y macros definidas por Colimate para el manejo de excepciones.

```
// Esta función no admite valores negativos
void funcion1(int a) _MTHROW {
   if (a<0)
     MREPORT_ERROR(5000,(string)"funcion1: no puedo procesar "
        "valores negativos: "+MItoA(a));
}

// Programa principal desde el que se contempla la posibilidad
// de una operación inválida
int main (int argc, char **argv) {
   int  a;
   // Lee línea de comandos
   try {
      a=Matoi(Mget_param(argc, argv, "-a"));
   } catch (MError ME) {
      cout << "Uso: programa -a <parámetro entero>\n";
      exit(1);
   }
```

```
   // Procesamiento
   try {
      ...
      funcion1(a);
      ...
   } catch (MError ME) {cout << ME; exit(ME.errorno());}
}
```

En este programa se han introducido dos funciones que todavía no se han presentado: Matoi que pertenece al grupo de funciones de conversión de tipo, y Mget_param que trata con la lectura de parámetros desde la línea de comandos.

Obsérvese la diferencia entre los dos tipos de tratamiento de errores: en el primer caso estamos leyendo de la línea de comandos y los errores posibles son: o que no haya puesto un flag “−a” , que no haya parámetro detrás de él o que el parámetro introducido no sea entero. En cualquiera de las situaciones el tratamiento a seguir es recuperarse del error y mostrar cuál es la línea de comandos correcta. Sin embargo, una vez dentro del procesamiento las operaciones inválidas pueden ser múltiples con lo que la única opción es mostrar el error producido y salir del programa con el código de error.

Macros útiles

En el caso de que no lo estén ya, se definen las constantes TRUE y FALSE, así como las funciones numéricas mínimo o máximo de dos valores, valor absoluto, y redondeo hacia $-\infty$ (FLOOR).

Conversiones de tipo y manejo de cadenas

Se implementan aquí funciones más inteligentes que las proporcionadas por C para la conversión de formatos numéricos (float o int) a cadenas y viceversa. Estas últimas incluso lanzan excepciones en el caso de que la cadena introducida no represente un valor válido del tipo numérico requerido. En cuanto al procesamiento de cadenas se han introducido funciones básicas como la simplificación de espacios

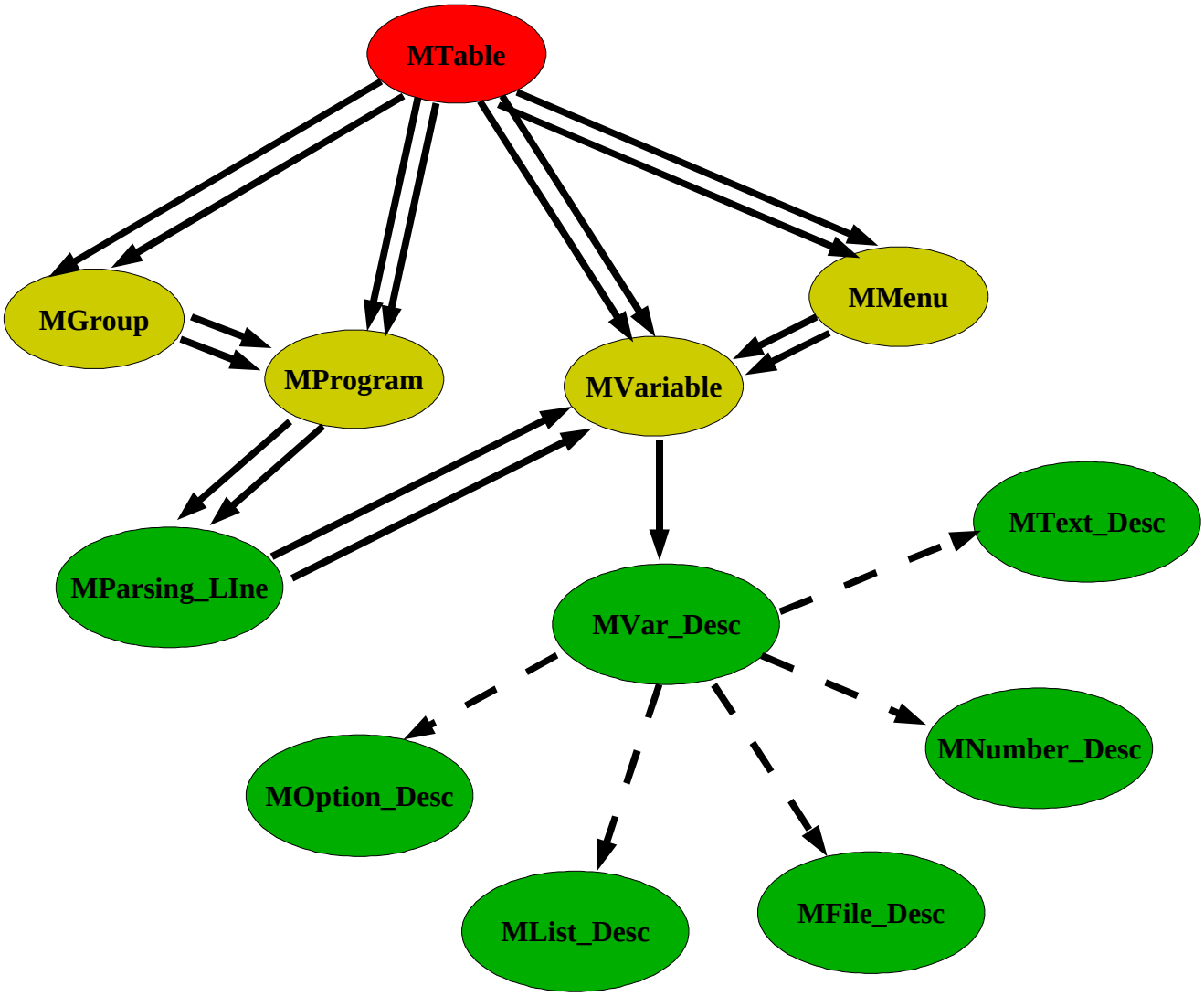
múltiples a un único espacio, eliminación de las comillas de un texto entrecomillado, o llevar una cadena de mayúsculas a minúsculas.

Manejo de argumentos de la línea de comandos

Se han implementado dos funciones tales que dada una línea de comandos en la que se supone que todos los parámetros van precedidos por un flag que indica lo que es, o bien devuelve el parámetro asociado a dicho flag (Mget_param) o bien simplemente indica si tal flag se encuentra presente o no en la línea. Mget_param es una función muy versátil que permite la omisión de parámetros, parámetros por defecto, y lanza excepciones cuando un parámetro obligatorio ha sido omitido.

2.5.2. Manejo de la tabla de símbolos

Las tablas de símbolos juegan un papel fundamental en Colimate, no son sólo el resultado de la compilación de la definición del interfaz, sino que además son la fuente utilizada para crear efectivamente el interfaz gráfico. Las tablas de símbolos empleadas tanto por el compilador de menús como por el generador son las mismas, y están programadas siguiendo una fuerte orientación a objetos. Esto quiere decir que las diferentes tablas de símbolos son entidades en sí mismas a las que les podemos solicitar que realicen operaciones, y que el contenido de las tablas son a su vez objetos que tienen también comportamientos propios. El siguiente diagrama trata de explicar las clases creadas, sus contenidos y relaciones. Cuando se dice que una clase referencia a otra puede ser una referencia en el sentido de puntero de C o una simple anotación del nombre de un objeto perteneciente a la otra clase. El diagrama intenta proporcionar una visión de conjunto del manejo de tablas, el presente apartado dará una explicación más particularizada a cada clase y, por último, como viene siendo habitual la descripción al detalle de cada clase, método y función se encuentra en las páginas hipertextuales de ayuda.



La línea continua significa contiene una referencia o el objeto en sí mismo, en caso de ser doble línea lo contenido será una lista de objetos del tipo indicado. La línea discontinua representa "es heredado por".

Una descripción muy somera del diagrama anterior nos revela que tenemos cuatro tablas de símbolos diferentes, cada una según el tipo de datos contenido. Por un lado, tenemos la tabla grupos, de programas, de variables y de menús. Los grupos entre otras cosas comprenden una lista de programas. A su vez los programas poseen una lista de líneas de comando o líneas a generar (*parsing lines*). Ya vimos que las líneas de comando se modelaban por una lista de símbolos o tokens entre los que se encontrarán variables que deberemos buscar en la tabla de variables, así mismo como

con los menús. Por último, las variables poseen una única descripción de variable que mantendrá información sobre las condiciones a imponer al contenido de la variable. Debido a algunas restricciones de los lenguajes orientados a objetos, deberemos crear una clase genérica que contenga dichas condiciones (MVar_Desc) y luego particularizarla para cada uno de los casos concretos (las herencias indicadas en el grafo).

Clase Tablas (MTable)

La estructura interna de la clase está compuesta por los siguientes miembros (a partir de ahora daremos un extracto del código en C++ asociado a la clase para ilustrar mejor la naturaleza de la misma)

```
template <class T>
class MTable {
   string      __type;
   vector<T *> __list;
   T *         __current_item;
   int         __current_index;
}
```

Se trata de una clase plantilla, es decir, es una clase genérica de manejo de tablas en la que los objetos contenidos en ella pueden ser casi de cualquier tipo. Está especialmente diseñada para tablas de símbolos por lo que el único requisito sobre los datos guardados es que respondan a los métodos requeridos para el modelo soportado de símbolos, como por ejemplo que tengan un “nombre propio”, o técnicamente expresado, que respondan a un método name() con una cadena indicando el nombre del objeto. La tabla guarda información sobre el tipo de datos contenido en ella (*__type*), una lista de punteros a los elementos contenidos en ella (*__list*) y una anotación sobre cuál ha sido el último elemento introducido (*__current_item* y *__current_index*). El modo normal de trabajar con la tabla será el de crear un objeto, comprobar que es correcto (por ejemplo, si necesita información de otra tabla, que dicha información está presente) y sólo entonces introducirlo en la tabla con la seguridad de que no vamos a tener que eliminarlo de ella por un error posterior de

compilación o cualquier otro.

En general la funcionalidad de la tabla se puede dividir en tres categorías:

- Información sobre la propia estructura: asignar el tipo de datos contenidos, acceso aleatorio a uno de los objetos, número de objetos en la lista, mostrar cada uno de los objetos, acceso al último elemento introducido, ...
- Funciones específicas de lista: vaciar la lista, extraer por detrás, por el frente, devolver el objeto al frente de la lista, borrado de un elemento que no ocupa el frente o la cola, liberación de la memoria ocupada por el objeto al frente de la lista, ...
- Funciones específicas para tablas de símbolos: estas funciones descansan normalmente sobre funciones con el mismo nombre de los objetos guardados en la tabla. La tabla lo único que hace es llamar en cascada a cada uno de los métodos correspondientes en cada objeto.
  - Está completamente relleno: esta función devuelve VERDADERO si todos los objetos en la tabla están completamente rellenos. El concepto de "completamente relleno" hace alusión al hecho de que todos los campos de la correspondiente estructura estén debidamente rellenos con valores válidos para su tipo. Es una validación interna, en el sentido de que sólo se comprueban aquellos campos de la estructura que no hacen referencia a ningún otro objeto.
  - Dependencias correctas: esta función devuelve VERDADERO si todas las dependencias de todos los objetos son correctas. Es ésta la contrapartida externa de la función anterior. La idea es comprobar que todas las referencias que se hacen a otros objetos son resolubles: por ejemplo, si en un menú se referencia una variable, comprobar que dicha variable está presente en la tabla de variables.
  - Añadir o buscar un objeto a la lista: la lista sólo guarda un puntero al objeto a introducir, el cual se añade normalmente al final de la misma. A la hora de buscar un objeto en la lista se le pregunta a cada objeto su nombre y se compara con el nombre buscado. Supone un error el hecho de introducir en la lista dos objetos distintos con el mismo nombre.

Clase Grupo (MGroup)

Una aplicación se compone de varios grupos de programas y el interfaz lo primero que debe determinar es a qué grupo pertenece el programa que el usuario desea ejecutar. Los miembros de la estructura son

```
class MGroup {
   string         __name;
   vector<string> __prog_list;
}
```

Vemos que el grupo posee una configuración muy sencilla: únicamente posee el nombre del propio grupo (*__name*) y una lista con los nombres de los programas que lo conforman (*__prog_list*). Con respecto a los métodos miembros, de nuevo los podemos dividir en:

- Información sobre la propia estructura: asignar u obtener el nombre del grupo, acceso aleatorio a uno de los programas, número de programas en el grupo, mostrar la lista de programas, ...
- Funciones específicas de símbolo: un grupo se dice que está completamente relleno cuando tiene un nombre no vacío y todos los programas en la lista también tienen nombres no vacíos. Por otra parte, las dependencias de un grupo son correctas cuando todos los nombres de programa en la lista del grupo se pueden encontrar en la lista de programas.
- Funciones específicas de grupo: añadir un programa al grupo y buscar un programa dentro de él.

Clase Programa (MProgram)

Un objeto programa debe mantener información algo más compleja, por un lado debe tener una lista con todas las posibles líneas de comando para este programa (*__parsing_line*) así como una referencia rápida a la última línea introducida en la lista (*__current_line* y *__current_index*), por otro deberá generar automáticamente etiquetas para las variables opcionales y tendrá que saber cuántas se han generado ya

(__*current_autom*), además tendrá información sobre su propio nombre (__*name*) y el nombre del menú a abrir cuando el programa sea seleccionado (__*menu*).

```
class MProgram {
   string                  __name;
   string                  __menu;
   vector<MParsing_Line *> __parsing_line;
   MParsing_Line *         __current_line;
   int                     __current_index;
   int                     __current_autom;
}
```

Debe observarse que el menú abierto será el mismo independientemente de la línea de comando seleccionada, es decir, todas las líneas de comando de un programa toman los parámetros del mismo menú.

Otro aspecto a resaltar es que, aunque en general las variables siempre pertenecen a un determinado menú (ya veremos más adelante en la clase variable cómo se consigue esto) y no al programa en que se definen (esta restricción surgía de la necesidad de que varios programas compartieran el mismo menú), las variables opcionales no son así. Pertenecen al programa y debe ser éste el que se encargue de generar etiquetas automáticamente para ellas. Sin embargo, la implementación de la "pertenencia" en Colimate permite que otros programas puedan hacer uso de opciones incluso de otros programas. Dentro de un programa las variables opcionales se numeran comenzando por 0 y la numeración es única. Esto se consigue asignando las etiquetas de forma consecutiva. Así, en el código Colimate siguiente

```
PROGRAM prog {
     OPEN MENU menu;
     COMMAND LINES {
          + linea1: prog [−a $PARAM_A];
          + linea2: prog [−b $PARAM_B];
     }
     ...
}
```

la opción “−a” sería la variable de etiqueta $*menu_prog_00000* y a la opción “−b” le corresponde la etiqueta $*menu_prog_00001*. Me remito a la clase de línea de comando para una descripción más en profundidad de cómo se codifican las mismas, y a la clase de menús para una explicación de por qué las etiquetas generadas llevan por delante el prefijo “$menu”. En cuanto al resto de la etiqueta no es difícil reconocer el nombre del programa en ella, de hecho el nombre de una variable opcional es el nombre del menú a abrir, más el nombre del programa al que pertenece, más una etiqueta numérica que la distingue del resto. Es éste precisamente el mecanismo implementado para la pertenencia de una variable opcional a un programa: al generar automáticamente la etiqueta le anteponemos el nombre del programa que la generó, así aseguramos que programas distintos generarán etiquetas distintas. Por otro lado, las variables son entes propios residentes en la tabla de variables, de este modo no hay ningún inconveniente para que un programa haga referencia a una variable opcional de otro siempre que ambos programas compartan el mismo menú (puesto que el nombre del menú será siempre el prefijo para cualquier variable encontrada en el contexto de un programa).

Dividiendo la funcionalidad del objeto en categorías, como viene siendo habitual:

- Información sobre la propia estructura: asignar u obtener el nombre del programa o del menú a abrir, acceso aleatorio a una de las líneas de comando, número de líneas en el programa, mostrar la información del programa, ...
- Funciones específicas de símbolo: se dice que un programa está completamente relleno si tiene un nombre no vacío, el nombre del menú tampoco lo es y todas las líneas de comando están completamente rellenas. Al mismo tiempo, un programa

tiene todas las dependencias correctas si el menú se encuentra en la tabla de menús y todas las líneas de comando tienen sus dependencias correctas.

- Funciones específicas de programa: añadir espacio para una nueva línea de comandos (se considera un error que haya dos líneas con el mismo nombre), referencia rápida a la última línea introducida, búsqueda de una línea y generación automática de etiquetas para variables opcionales.

Clase Línea de Comando (MParsing_Line)

La línea de comandos es el objeto que tiene la información sobre lo que hay que generar en cada momento, cuáles son los campos que componen un determinado modo de funcionamiento de un programa, cuáles son opcionales y cuáles no. Ya vimos en un apartado anterior que la línea de comando se modelaba básicamente como una lista de símbolos (tokens) que podían ser bien textuales bien variables. En esta implementación se distinguen unos de otros en que las variables comienzan siempre por el carácter "$" y los textos no. La estructura en C++ correspondiente responde al siguiente código

```
class MParsing_Line {
   string                    __name;
   vector<string>            __token;
   const MTable<MVariable> * __VT;
}
```

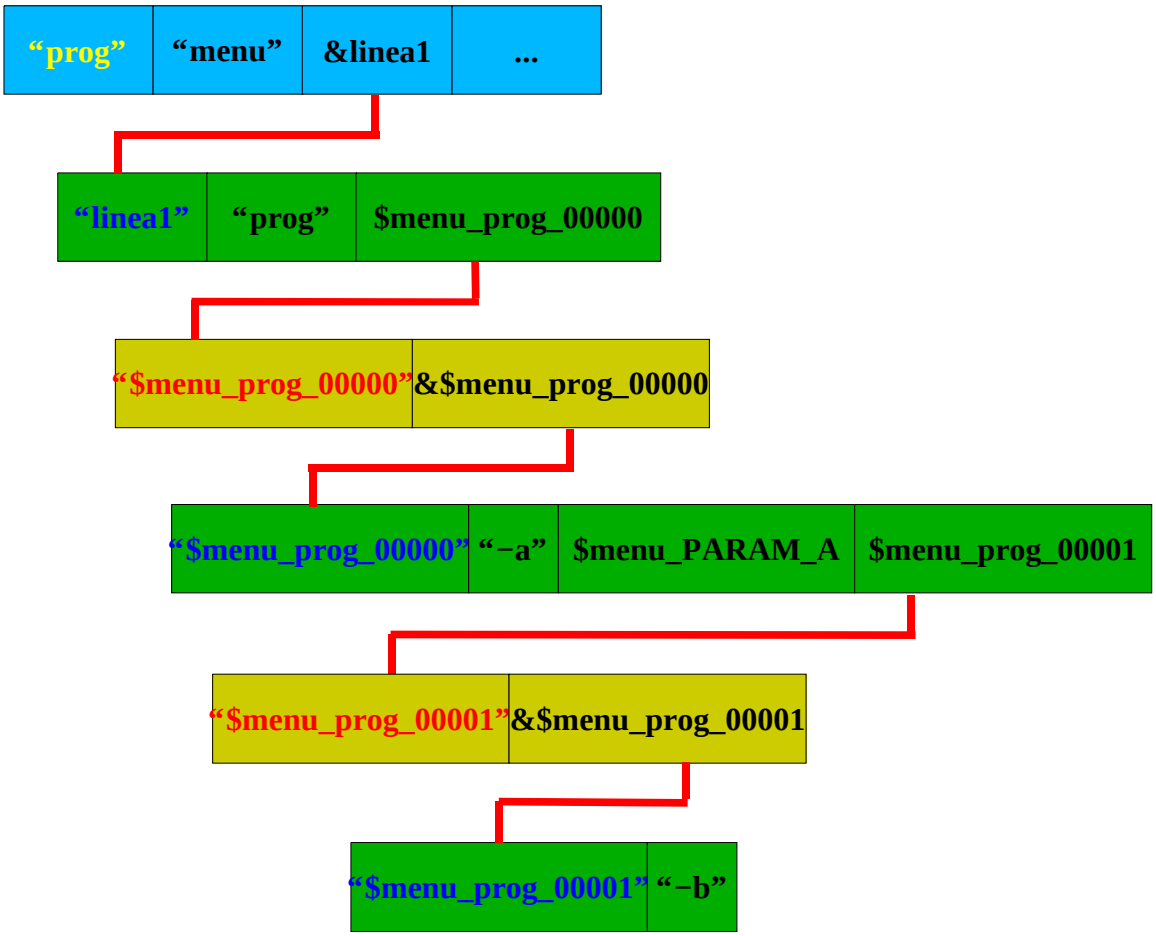
Vemos como esencialmente lo que se guarda en la estructura es una lista de cadenas (*__token*) que son los símbolos. Adicionalmente se guarda el nombre de la línea de comandos (*__name*, dos programas diferentes pueden tener dos líneas de comando que se llamen igual), y una referencia a la tabla de variables (*__VT*) en la que se deben buscar todos los símbolos variables de la línea.

Explicaremos ahora en detalle la implementación concreta de los argumentos opcionales. Cuando Colimate encuentra un argumento opcional lo encapsula completamente en una variable de tipo OPCIONAL, y es esta variable la que aparece

en la lista de símbolos de la línea de comando. Luego, una variable de tipo opcional posee una línea de comando a su vez, es decir, el contenido de una variable de tipo opcional es la propia lista de símbolos a la que sustituye. Este mecanismo permite el anidamiento de argumentos opcionales y así dentro de un argumento opcional podemos tener otros argumentos opcionales o no. Veamos con un ejemplo cómo se traduce a nivel de tabla de símbolos el siguiente código Colimate:

```
PROGRAM prog {
     OPEN MENU menu;
     COMMAND LINES {
          + linea1: prog [-a $PARAM_A [-b]];
     }
     ...
}
```

Los aspectos más relevantes de la tabla de símbolos se puede resumir en el siguiente diagrama esquemático (en oscuro las líneas de comando y en claro las variables)



Vemos como la línea 1 del programa “prog” contiene un único símbolo que es una variable opcional llamada *$menu_prog_00000* (ya se ha explicado el origen de tal nombre). El contenido de esta variable es a su vez una referencia a una nueva línea de comando con el mismo nombre que la variable opcional cuyo contenido es el segmento de la línea de comando comprendido entre los corchetes más externos. Esta estructura se repite al anidarse otro argumento opcional dentro de éste, dando lugar a la variable opcional y línea de comando llamadas como *$menu_prog_00001*.

Dejando el ejemplo y volviendo a la clase de líneas de comando, podemos clasificar las funciones de esta clase como:

- Información sobre la propia estructura: asignar u obtener el nombre de la línea, asignar y obtener la referencia a la tabla de variables, acceso aleatorio a una de las líneas de comando, número de símbolos o variables en la línea, mostrar la información de la línea, ...
- Funciones específicas de símbolo: una línea de comandos está completamente rellena si su nombre y todos los símbolos son cadenas no vacías. Mientras que sus dependencias son correctas si los nombres de variable que en ella aparecen son encontrados en la tabla de variables.
- Funciones específicas de línea de comando: añadir o buscar un símbolo en la lista, éste es de los pocos casos en que da igual que el símbolo esté repetido en lo que ya hayamos introducido en la lista de símbolos, pues ahora sí tiene sentido poner dos veces el mismo valor; generar la línea de comando con los valores actuales de las variables en memoria y ejecutarla; y generar el trozo de línea de comando asociado a una determinada variable opcional.

Clase Variable (MVariable)

Esta clase es la de los parámetros de tipo variable de las líneas de comando. Ya hemos mencionado que las variables son objetos en sí mismos independientes de cualquier otra estructura, y están almacenados en la tabla de variables. De este modo, las variables no dependen de ningún programa o menú. Esto no es del todo cierto puesto que toda variable comienza por prefijo que indica en qué menú es mostrada y en cierto modo define implícitamente una pertenencia. Es importante resaltar que esta

dependencia se da con los menús y no con los programas. Así dos programas pueden compartir una misma variable sobre un mismo menú, pero programas que abren menús diferentes nunca podrían compartir una variable. Esta característica es deseable en el desarrollo de nuestro interfaz para evitar recoger valores de menús que no se encuentren en pantalla.

Un extracto de la clase C++ que representa a estos objetos nos revela

```
class MVariable {
   string            __name;
   Var_Type          __type;
   string            __help;
   string            __label;
   string            __default_val;
   string            __val;
   MVar_Desc *       __pt_desc;
}
```

donde

```
typedef enum {
   NOT_ASSIGNED      = -1,
   TEXT              =  0,
   NATURAL           =  1,
   INTEGER           =  2,
   FLOAT             =  3,
   FILENAME          =  4,
   LIST              =  5,
   OPTION            =  6 } Var_Type;
```

Las variables tienen un nombre (*__name*) que como ya hemos dicho siempre comienzan por el nombre del menú en que se solicitan precedidas por el carácter “$”, un tipo indicando la naturaleza del contenido de la variable (*__type*), una etiqueta más informativa que el nombre de la variable (*__label*) que es lo que realmente se muestra al usuario, una ayuda extra con más información que complemente a la etiqueta

(*__help*), un valor por defecto (*__default_val*), el valor actual de la variable (*__val*) y un puntero a un descriptor de variable dependiente del tipo de variable (*__pt_desc*), es decir, las variables textuales poseen un descriptor de variable diferente a las variables numéricas.

En un entorno de programación como C el problema de tener varios tipos de descriptores según el tipo de variable se resolvería usando una unión o un puntero a void, sin embargo, en una filosofía orientada a objetos podemos resolver este asunto de una forma más elegante mediante mecanismos de herencia. La variable posee un puntero a un descriptor de variable que es una clase padre para los diferentes tipos de descriptores.

Debemos resaltar el hecho de que para Colimate todos los valores de las variables son cadenas de caracteres, es el tipo de la variable el que luego determina que dicha cadena sea correcta o no.

Atendiendo, como viene siendo habitual, a las funciones miembro de la clase, podemos dividir éstas en:

- Información sobre la propia estructura: asignar u obtener el nombre de la variable, su tipo, la etiqueta, la ayuda, su valor por defecto y su valor, el puntero al descriptor; mostrar la variable.
- Funciones específicas de símbolo: una variable está completamente rellena si su nombre y etiqueta son cadenas no vacías y el tipo está asignado a algún tipo válido (diferente de NOT_ASSIGNED). Por otra parte, la variable cumple todas sus dependencias siempre salvo en el caso de variables de tipo OPTION en las que se comprueba que las variables que aparecen en la línea de comando asociada se encuentran en la tabla de símbolos.
- Funciones específicas de variable: aparece un nuevo concepto el de valor correcto. Una variable tiene un valor correcto si su valor como cadena cumple todos los requisitos impuestos por el tipo de variable y el descriptor (por ejemplo, si es una variable entera con rango, pues que la cadena correspondiente al valor represente a un número entero perteneciente al citado rango). Esta función descansa sobre otra de igual nombre en los correspondientes descriptores de tipo.

Clase Descriptor Genérico de Variable (MVar_Desc)

Esta clase es el fruto de una solución orientada a objetos del problema de tener varios tipos de descriptores distintos en dependencia del tipo de variable. La solución adoptada es que el objeto variable tenga un puntero a una clase descriptor genérico de variable de la que heredarán los diferentes tipos específicos de descriptores. En esta solución estamos haciendo uso de la propiedad de los punteros en C++ que dice que un puntero a una clase padre es también un puntero a cualquiera de las clases que heredan de ella. Esta clase está vacía de contenido y lo único que obliga es que sus hijas tengan un método que determine la corrección del valor de la variable.

Las clases que heredan de ésta son un descriptor textual, un descriptor numérico, otro de fichero, lista y, por último, de variable opcional. Veremos que todos las variables numéricas, independientemente de su tipo, tienen el mismo descriptor.

Clase Descriptor Textual (MText_Desc)

La única restricción que se ha implementado sobre un texto es la posible limitación de su longitud. En caso de que no deseemos limitarla basta con poner la longitud máxima a −1 (valor por defecto). Vemos que la estructura de esta clase es muy sencilla.

```
class MText_Desc: public MVar_Desc {
   int    __max_length;
}
```

Aparte de tener funciones para asignar y preguntar por la longitud máxima, y mostrar el contenido del descriptor, la única función interesante de este descriptor es la de decidir si el valor de la variable tiene un valor correcto, en este caso, que su longitud no exceda la máxima permitida en el caso de que alguna restricción se haya impuesto.

Clase Descriptor Numérico (MNumber_Desc)

El descriptor numérico define límites para el valor de la variable, así podemos obligar a dicho valor a pertenecer a un determinado rango que se identifica con alguno de los siguientes intervalos numéricos: (∞,b], [a,b] o [a,∞), o ninguno de ellos. Los extremos de los intervalos son valores reales, a pesar de que la variable pueda únicamente tomar valores enteros, por ejemplo. La estructura interna de la clase viene determinada por

```
class MNumber_Desc: public MVar_Desc {
   bool         __left_bounded;
   float        __f0;
   bool         __right_bounded;
   float        __fF;
}
```

Los extremos de los intervalos a y b son *__f0* y *__fF* respectivamente, un flag habilita o no la presencia de dicho extremo (*__left_bounded* y *__right_bounded*), las diferentes combinaciones de activo/desactivo van dando la configuración de los distintos tipos de intervalos.

La clase provee funciones suficientes para asignar el tipo de intervalo y sus extremos. Una variable numérica tiene un valor correcto según este descriptor si se encuentra dentro del intervalo para ella definido (si es que hay alguno) y si cumple las restricciones de tipo (ser entero, ser natural, o representar un valor real).

Clase Descriptor de fichero (MFile_Desc)

El tipo de restricción abordado por este descriptor se atiene más bien a dar un atributo sobre el valor de la variable. Así tenemos que

```
class MFile_Desc: public MVar_Desc {
   File_Type __file_type;
}
```

donde

```
typedef enum {
   DONT_MIND     =  0,
   EXISTING      =  1,
   NON_EXISTING  =  2,
   PATTERN       =  3 } File_Type;
```

Es decir, lo que imponemos a un fichero es el hecho de que exista o no, o le damos el atributo de ser un patrón de nombre de fichero. Por defecto, diremos que no nos importa el estado de existencia o no del fichero. Como es natural existen funciones para el manejo del objeto (asignación, lectura del estado, mostrar, ...). Una variable de tipo fichero se dice que tiene un valor correcto si cumple la condición de existencia, es decir, si debe existir, que efectivamente exista y si no debe existir, que no exista. Si es un patrón, realmente no se está poniendo ninguna limitación a dicha variable.

Clase Descriptor de Lista (MList_Desc)

Recordemos que una lista era un tipo especial de datos que no generaba código alguno en la línea de comandos y que servía para modificar el valor de otras variables en el mismo menú. El descriptor de la lista lo que guarda es una serie de etiquetas (*__label*) a las que se les asocia una lista de acciones (*__AL*), por último, como ayuda al proceso de compilación se mantiene información sobre cuál es la última etiqueta introducida (*__current_label*).

```
class MList_Desc: public MVar_Desc {
   vector<MAction_List *> __AL;
   vector<string>         __label;
   int                    __current_label;
}
```

Veremos en la siguiente clase cómo se almacenan las acciones asociadas. De nuevo nos encontramos en esta clase la funcionalidad normal de mantenimiento de la

clase (acceso aleatorio a los datos contenidos en él, mostrado del objeto, ...) más una serie de funciones especiales debido a la naturaleza del objeto como añadir una nueva etiqueta con un conjunto de acciones asociado (se considera un error introducir dos veces la misma etiqueta) o ejecutar la lista de acciones asociada. Una variable de tipo lista tiene un valor correcto si representa un índice válido de la lista de etiquetas y acciones.

Clase Lista de Acciones (MAction_List)

El único tipo de acciones considerado por Colimate es el de la asignación de variables ora desde un valor textual entrecomillado ($a=a;} ora desde el contenido de otra variable ($a=$b;). Aparte de las operaciones normales para acceder a la lista de acciones, determinar las asignaciones, contar el número de ellas, ... podemos ejecutar el contenido de la lista en sí misma, es decir, la propia lista de acciones se encarga de acceder a la tabla de variables y realizar las asignaciones necesarias.

Clase Descriptor de variable Opcional (MOption_Desc)

Las variables opcionales ya han sido comentadas a lo largo de la memoria, sin embargo, recordemos cuáles eran sus principales características: son variables que se generan automáticamente en Colimate y sustituyen a los trozos de la línea de comandos que se encuentran entre corchetes. Así el valor de la variable opcional es únicamente activado o desactivado. En el caso de estar activa todo el segmento al que sustituye está habilitado y podremos introducir valores para las variables que cuelguen de él. Dentro de una opción se puede dar otro argumento opcional. Para poder asignar valores a los parámetros que se encuentran en este segundo nivel de opcionalidad deberemos activar las dos variables de las que cuelgan. Ver la clase de las líneas de comando en la que se propone un ejemplo de uso y generación de este tipo de variables.

Recordemos también que las variables opcionales comienzan por el nombre del menú en el que se muestran, el nombre del programa en que se definen y por último un número que indica el orden que ocupa entre todas las variables opcionales

de dicho programa.

El contenido del descriptor se reduce prácticamente a una línea de comando a la que sustituye.

```
class MOption_Desc: MVar_Desc {
   MParsing_Line __parsing;
}
```

Existen funciones para acceder a la línea de comando sustituida, asignar su nombre (el compilador de Colimate da el mismo nombre a la línea de comando sustituida que a la variable opcional correspondiente), introducir nuevos símbolos en el trozo, y mostrar el descriptor. Una variable de tipo opcional se dice que tiene un valor correcto cuando su valor es 0 o 1 y todas las variables que se hallan debajo de ella tienen un valor correcto.

Clase Menu (MMenu)

La clase menú es la que contiene cada uno de los interfaces de usuario en los que se solicitarán los valores de los parámetros. El modelo de menú ya fue discutido en la sección 2.1, veíamos allí que un menú no era más que una lista de cadenas fijas y de variables que deseamos presentar. Vemos esta estructura perfectamente reflejada en la definición de la clase: la lista de objetos a presentar (*__obj_list*) agrupados en un sólo que recibe un nombre de menú (*__name*).

```
class MMenu {
   string __name;
   vector<string> __obj_list;
}
```

Además de las funciones lógicas para el acceso a cada uno de los campos de la clase, se proporcionan funciones para añadir un objeto o para comprobar si determinado objeto está o no. Un menú se dice que está completamente relleno si tiene un nombre no vacío y todas las variables que aparecen están completamente

rellenas (para esta segunda condición se requiere además que las variables se encuentren en la tabla de variables). Por otra parte, un menú tiene todas las dependencias correctamente si todas sus variables se encuentran en la tabla de variables.

2.5.3. Compiladores

Como ya se ha visto se habla de compiladores en plural debido a que efectivamente se han tenido que desarrollar dos compiladores diferentes, uno por cada gramática utilizada: la de entrada y la intermedia. Sin embargo, se ha invertido un esfuerzo importante en mantener una estructura genérica y común para ambos compiladores. Comentaremos en primer lugar aquellas características comunes y pasaremos más tarde a considerar algunas de las particularidades propias de cada uno. Sin embargo, no entraremos en esta sección a un comentario exhaustivo de cada regla para no dejar que "los árboles no nos dejen ver el bosque". Para tal nivel de detalle me remito al propio código comentado debidamente y presente en las páginas de ayuda hipertextual que acompañan a la presente memoria.

Los compiladores se han desarrollado haciendo uso de los lenguajes Flex++ y Bison++ que no son sino evoluciones de Lex y Yacc respectivamente, por lo que la sintaxis de dichos lenguajes recuerda mucho la de sus antecesores. Sin embargo, la filosofía y estructura de los analizadores generados son muy diferentes a los producidos por Lex y Yacc.

En Lex y Yacc, el analizador léxico y sintáctico residen en funciones independientes que se comunican a través de llamadas a ciertas funciones conocidas por ambos y por medio de variables y macros globales. En concreto, el analizador sintáctico reside en una función llamada *yyparse*, el analizador sintáctico en *yylex*, y el analizador sintáctico determina cuáles serán los símbolos a devolver por medio de la definición de unas macros cuyos valores son precisamente los códigos asociados a cada símbolo. Esta estructuración obliga a que solamente haya un compilador por programa, puesto que no pueden repetirse nombres de función. Además, las tablas de símbolos también deben tener un ámbito global, impidiendo la existencia de varias

tablas del mismo tipo. Si atendemos al uso indiscriminado que se hace del concepto "global" (tablas globales, variables globales, macros globales, funciones globales, ...) vemos que esta práctica atenta contra las características deseables en un sistema modular.

Flex++ y Bison++ solucionan este problema. Aparte de que el lenguaje base es C++, lo cual ya es una ganancia respecto a Lex y Yacc que emplean C, los compiladores son concebidos como entidades propias, es decir, como clases. Habrá una clase que se llame como nosotros deseemos nombrarla que ejercerá las funciones de analizador sintáctico. Uno de sus métodos será el clásico *yyparse*, además las herramientas nos permiten definir nuevos métodos con lo que podemos dotar a las clases de una funcionalidad tan compleja como queramos. De igual modo, el analizador léxico compone en sí mismo una clase con un método *yylex*. Los símbolos son ahora miembros de un tipo enumerado que pertenece a la clase del analizador léxico. Nótese como ahora la modularidad es absoluta, podremos tener varios compiladores de gramáticas distintas en un mismo programa e incluso varios compiladores del mismo lenguaje corriendo a la vez sobre diferentes textos fuente. Además dos compiladores diferentes podrían tener el mismo nombre de símbolo y, sin embargo, tratarse de símbolos diferentes al estar dentro de la clase particular del analizador sintáctico.

Haciendo uso de la nueva filosofía de construcción de compiladores, en el caso del analizador léxico de entrada se ha extendido la funcionalidad ofrecida por Flex++ para los analizadores léxicos de tal manera que ellos mismos gestionen la inclusión de ficheros y mantengan el número de línea que se está procesando en cada uno de ellos. Cada analizador léxico mantiene una pila de los ficheros fuentes que él ha abierto y otra pila con las líneas activas en cada uno de ellos. Vemos de nuevo la potencialidad de la orientación a objetos introducida por Flex++, estas pilas son internas a cada objeto de esa clase, por lo que si hubiese dos compiladores del mismo lenguaje o de otro distinto, cada uno sabría cuáles son sus ficheros a procesar de forma independiente.

El analizador léxico realiza algunas operaciones básicas sobre el texto introducido como detectar el comienzo y el fin del código fuente para Colimate, ignorar los espacios y saltos de línea entre símbolos, ignorar los comentarios, y gestionar la inclusión de ficheros (sólo el de la gramática de entrada). Además en cuanto a las palabras claves los analizadores léxicos son insensibles al estado de mayúscula o minúscula del símbolo. Todo esto servirá para simplificar el analizador sintáctico.

En cuanto a los analizadores sintácticos podemos decir que son ellos mismos los que poseen como miembros a las tablas de símbolos. De nuevo, hemos ganado fuertemente en modularidad, cada analizador sintáctico gestiona sus propias tablas, o mejor dicho posee sus propias tablas, la gestión de la tabla en sí es realizada por la propia tabla, éste es uno de los pilares básicos de la programación orientada a objetos. También se ha implementado un modo de depuración invocable desde la línea de comando (al estilo de modo *verbose*) del compilador de forma que obtengamos una salida rica en información sobre el proceso realizado por el analizador sintáctico en cada instante.

Aprovechando la orientación a objetos, hemos construido una clase más específica sobre los analizadores sintácticos proporcionados por Bison++. La nueva clase llamada compilador (cada una de las gramáticas tiene su propia clase compilador) hereda del analizador sintáctico por lo que tiene el mismo comportamiento que ella pero además incluye el analizador léxico. Además, se encarga de servir de envoltura de las funciones que necesitan ser accedidas desde el exterior, así, por ejemplo, ya no le pedimos al analizador léxico que comience a leer de un determinado buffer de entrada sino que se lo solicitamos al compilador. El compilador es también quien se encargará de comprobar la consistencia de las dependencias entre tablas de símbolos y que todos las entradas se encuentren completamente rellenas. Con esta nueva clase, desde el exterior únicamente necesitamos instanciar un objeto de tipo compilador e interaccionar con él.

Una vez comentadas las características comunes y principios básicos de diseño de los compiladores en el presente proyecto fin de carrera, procedamos a revisar las

particularidades de cada uno de los dos compiladores implementados.

Compilador de Gramática de entrada

Como veíamos la gramática de entrada es la que usan los programadores para especificar a Colimate la estructura de las líneas de comando e interfaces a manejar. Este código fuente puede estar repleto de errores por lo que el compilador correspondiente tendrá que ser lo más robusto posible ante ellos. Para ello se han empleado diferentes técnicas y controles semánticos que verifican la corrección del código fuente introducido.

Por una parte se hace uso de la recuperación de errores sintácticos ofrecido ya por YACC con el símbolo *error* y la directiva *yyerrok*. Se suelen utilizar en este contexto los símbolos ";" y "}" como símbolos de recuperación.

Por otra parte, unos flags nos indican si ya se produjo un error en la presente definición de elemento para no introducirlo en la tabla de símbolos correspondiente, por ejemplo, si en la definición de un grupo se produjo un error pero ya nos recuperamos de él, este indicador nos ayuda a conocer tal situación de manera que cuando termine la definición del grupo no lo introduzcamos en la tabla de grupos puesto que contiene algo con sintaxis errónea.

Por último, a la hora de introducir símbolos en las tablas debe ser bastante cuidadoso de manera que únicamente se añadan símbolos de los que conocemos a ciencia cierta que no contienen ningún tipo de error sintáctico. Esta condición no se puede saber hasta el final de la definición del símbolo por lo que tendremos que esperar a este momento para introducirlo. Un caso extremo de esta precaución es el caso de las variables, conforme las vamos reconociendo dentro de un programa necesitamos almacenarlas en alguna parte, pero hacerlo en la tabla global de variables es un tanto peligroso por si posteriormente se produce un error que nos haga desechar el programa. Por ello, se van almacenando en una tabla temporal de variables, cuando el programa se ha reconocido como correcto todas las variables que se introdujeron en dicha tabla temporal pasan a la tabla "oficial" de variables.

Cada vez que se produce un error en algún punto se informa al usuario de cuál símbolo es el que ha dado el error, en qué línea y de qué fichero.

Compilador de Gramática intermedia

El compilador de gramática intermedia es mucho más simple que su homólogo de gramática de entrada debido fundamentalmente a tres motivos: el primero es que el texto a reconocer se ha generado automáticamente por lo que la posibilidad de errores léxicos o sintácticos es imposible, el segundo es que la posibilidad de errores semánticos tampoco es posible ya que el compilador de la gramática anterior se ha encargado de detectarlos, y por último que la riqueza expresiva del lenguaje a definir es menor ya que se limita a reconocer el contenido de unas tablas fijas y bien delimitadas. Por ello, este compilador no incluye protección contra errores aunque sí se le puede ir pidiendo que nos dé una salida rica en información (modo *verbose*).

La misión principal del compilador de gramática intermedia es la de servir de puente entre la verdadera compilación y el generador de interfaces. De hecho, esta segunda compilación constituye la primera fase de la generación, como veremos a continuación.

2.5.4. Generador de interfaz

Colimate únicamente tiene dos programas visibles por el usuario: el compilador de menús que genera la tabla de símbolos intermedia que ya hemos visto, y el generador de interfaz que recoge dicha tabla de símbolos y construye a partir de ella un interfaz gráfico de usuario.

El generador de interfaz se puede dividir en tres partes bien diferenciadas en el tiempo: en una primera fase, interpreta el texto que le llega del compilador de menús por medio del propio compilador de gramática intermedia; en un segundo paso, se construye una representación interna en memoria de todo el interfaz, se crean

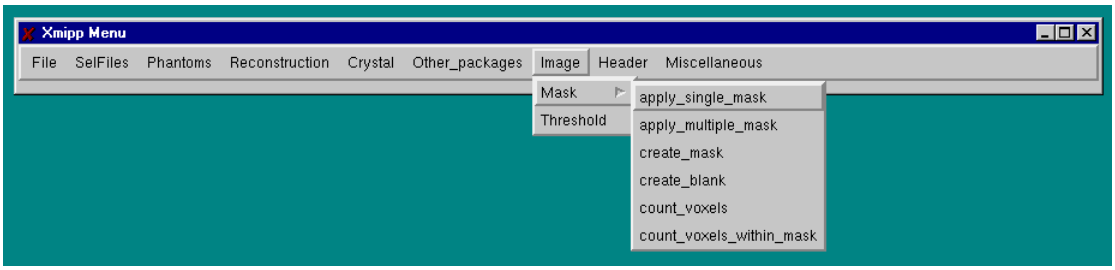
todos los objetos y se les asigna las propiedades y dependencias adecuadas. Por último, se muestra dicho interfaz y se deja que funcione de forma "autónoma", para Qt un programa de interfaz normalmente se compone de una clase específica que resume las características de la aplicación (*QApplication*).

Podemos profundizar un poco en esta gestión "automática". Qt mantiene internamente una lista de los objetos propios de Qt y que se hayan definido heredando de clases de Qt. Para estas nuevas clases podemos definir o no nuevos métodos que oculten los heredados por defecto de la clase Qt correspondiente, como por ejemplo el método *show()* que indica cómo mostrar un objeto. Al mismo tiempo Qt proporciona un mecanismo de comunicación entre objetos por medio de señales y bahías (*signals* y *slots* en inglés). De este modo, cuando pulsamos un botón en el interfaz se genera la señal *pressed()* que puede ser conectada a un *slot* de la clase que contiene el botón, por ejemplo, *slotButtonPressed()*. Algunas señales y slots vienen por defecto en Qt, otros tendremos que añadirlos nosotros. Es de resaltar que la comunicación se realiza entre objetos y no entre clases. Como nota sobre este mecanismo de comunicación diremos que Qt extiende el lenguaje C++ con algunas palabras reservadas para este cometido, por lo que antes de poder compilar nuestro programa tendremos que pre−compilarlo con el Meta Object Compiler proporcionado por Qt que traduce del C++ extendido a C++ estándar.

De los párrafos anteriores deberemos extraer la idea de que la misión del generador de menús es la de leer las especificaciones del interfaz a desarrollar, crear en memoria todos los objetos Qt y de clases derivadas que implementen dicho interfaz, realizar las conexiones entre objetos necesarias y posteriormente dejar que sea el propio interfaz el que vaya generando los eventos que al entrar por slots a las clases derivadas son capaces de acceder a las tablas de símbolos y finalmente producir la línea de comando que el usuario ha seleccionado.

Siguiendo las directivas y requerimientos del interfaz, lo hemos diseñado de la siguiente manera: el interfaz primario de usuario será el conjunto de grupos dispuestos en una barra de menús. Al seleccionar alguno de los menús se nos mostrarán los diferentes programas que pertenecen a dicho menú. Si un programa

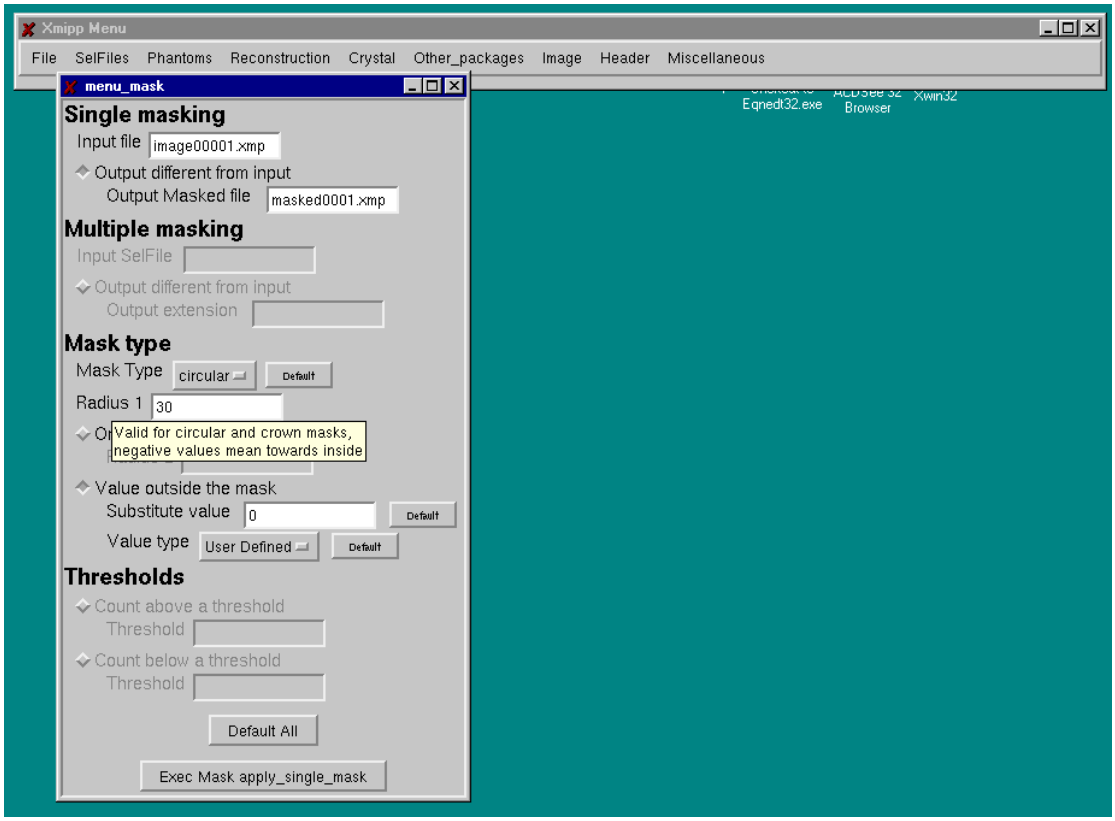
posee varias líneas de comando, al seleccionarlo se nos pedirá (siempre en forma de menús desplegables) cuál de ellas deseamos activar. En caso de tener una única línea de comando directamente pasaremos a abrir el interfaz correspondiente con los parámetros necesarios. La siguiente figura ilustra esta primera fase de interacción.



Nótese que el primero de los grupos es "File", este grupo es común a todas las aplicaciones Colimate y posee funciones generales a realizar tales como:

- Cargar/salvar: el estado actual de las variables. Esto sirve para mantener configuraciones especiales sobre cada uno de los parámetros de manera que podamos volver rápidamente a ellas en caso de que sea necesario. De todos modos, la propia aplicación al cerrarse salva el estado actual de las variables en el mismo fichero de tabla de símbolos con el que fue invocado
- Cambiar de directorio: por medio de esta opción podremos cambiar el directorio de trabajo con respecto al que llamó a la aplicación.
- Comando del sistema: esta utilidad nos permitirá introducir y lanzar cualquier comando del sistema, hace las funciones de interfaz del sistema.
- Salir de la aplicación: Termina la aplicación, recordemos que al salir la aplicación guarda el estado de las variables en el mismo fichero de tabla de símbolos desde el que se le invocó.

Una vez seleccionada la línea de comando deseada aparecerá el menú correspondiente, véase el ejemplo mostrado en la figura a continuación.



Este ejemplo nos servirá para mostrar la mayor parte de las especificaciones deseadas para nuestro interfaz de línea de comando. Por un lado, sabemos que el modelo de menú es una lista de textos separadores y variables. Vemos como los separadores aparecen en negrita y con un tamaño de letra algo mayor que el resto. Entremezclados aparecen las diferentes variables. La indentación en Colimate es fundamental y refleja la jerarquía en la dependencia.

Los parámetros habitualmente aparecen con una etiqueta junto con un campo editable en el que deberemos poner el valor adecuado para dicho parámetro. Los argumentos opcionales están deshabilitados al principio, sin embargo, poseen un botoncito a la izquierda que los activa y una etiqueta que nos indica la naturaleza del argumento. Una vez activado un argumento se generará línea de comando para él. Los argumentos opcionales pueden anidarse, en este caso el nivel de jerarquía es también indicado por la indentación. Si detenemos el cursor sobre una etiqueta obtendremos ayuda adicional para el argumento o parámetro como la mostrada sobre fondo amarillo para el parámetro "Radius 1". Los parámetros con valores por defecto tienen al lado derecho un botón que si es pulsado asigna dicho valor por defecto al

actual contenido por la variable. Un botón general para todo el menú (default_all) fuerza a todas las variables que posean valores por defecto a adoptarlos. Por último, el botón "Exec" corresponde al "Enter" del interfaz del sistema, su función es la de lanzar el comando correspondiente a los parámetros actualmente activos en el interfaz. El propio botón de ejecución hace una referencia al programa que abrió el menú y por medio de qué modalidad de línea de comando.

Debemos indicar que todos los campos editables, además de la mera edición de texto, añaden la posibilidad de acceder a valores pasados al mantener un histórico de parámetros, y que en el caso de corresponder a un parámetro de tipo fichero con el botón derecho del ratón pulsado sobre ellos accedemos a un menú desplegable que permite realizar alguna acción particular sobre el fichero. Estas acciones están definidas por la directiva RightButton de la gramática y como mínimo siempre ofrecerá la posibilidad de seleccionar el fichero de la estructura de directorios por medio de una interfaz gráfica estándar.

En capítulos anteriores hemos estudiado el compilador de gramática intermedia como primera fase de este generador de menús. Para terminar con el análisis de la implementación de este módulo estudiaremos las clases y estructuras relacionadas con la interfaz gráfica.

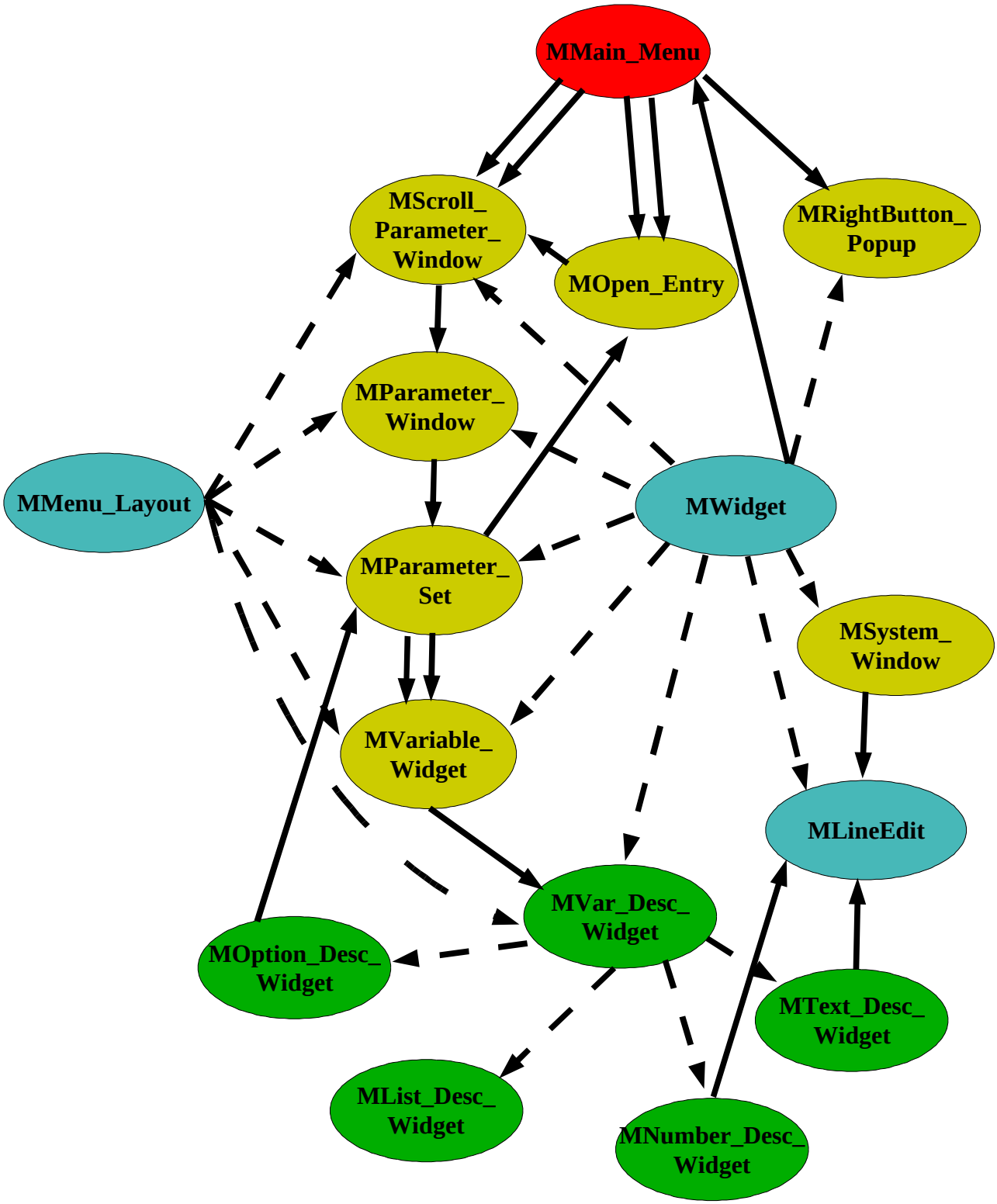
2.5.5. Manejo de la interfaz gráfica

En esta sección comentaremos los hechos más relevantes de las clases, funciones y estructuras relacionadas con el manejo de la interfaz gráfica. Estas clases son las que interactúan directamente con Qt, consisten en general en una especialización de las clases de Qt al tipo de interfaz específico desarrollado en este proyecto. Comenzaremos por representar gráficamente las dependencias y herencias entre las diferentes clases gráficas de Colimate (obviando para no complicar aún más el grafo las dependencias con clases de Qt y con clases de manejo de la tabla de símbolos del propio Colimate), para pasar posteriormente a un estudio más profundo de cada una de ellas. De todos modos, me remito a la ayuda hipertextual para una definición exhaustiva de cada clase.

La línea continua significa que contiene una referencia o el objeto en sí mismo; en caso de ser doble línea continua lo contenido será una lista de objetos del tipo indicado. La línea discontinua representa "es heredado por".

El gráfico en sí mismo es suficientemente complejo como para ser autoexplicativo. Sin embargo, con un poco de paciencia iremos desgranando cada una de las relaciones y comprenderemos mejor qué es lo que viene a decirnos. En esta primera aproximación daremos una visión de conjunto de todo el sistema.

Podemos empezar reconociendo tres clases de apoyo al sistema, a saber, MMenu_Layout, MWidget y MLineEdit. La primera de ellas es una extensión del concepto de Layout de Qt, se encarga de mantener la jerarquía entre argumentos, y disponer las diferentes etiquetas y elementos de un menú o variable físicamente en la pantalla. En segundo lugar, MWidget únicamente nos servirá para mantener en cada objeto una referencia a la aplicación de la que dependen. Vemos como la mayoría de las clases heredan de MWidget, esto significa que la mayoría de las clases objetos físicos de la aplicación y necesitan conocer quién las ha creado y de quién dependen. Veremos más adelante que esta referencia a la aplicación es de vital importancia pues es en ella en la que se almacenan referencias a las tablas de símbolos de manera que en el momento que cualquier objeto necesita acceder a ellas no tiene más remedio que pedirle a la aplicación que le proporcione la dirección de dichas tablas. Por último MLineEdit es una extensión del editor de línea de Qt al que se añade la funcionalidad de mantener un histórico de valores al que se accede por medio de los cursores y la de desplegar un menú con diferentes acciones en el caso de tratarse de un fichero.


MMain_Menu
MScroll_ Parameter_ Window
MOpen_Entry
MRightButton_ Popup
MParameter_ Window
MMenu_Layout
MWidget
MParameter_ Set
MSystem_ Window
MVariable_ Widget
MLineEdit
MVar_Desc_ Widget
MOption_Desc_ Widget
MText_Desc_ Widget
MList_Desc_ Widget
MNumber_Desc_ Widget

El principal objeto del generador de menús, la aplicación, es de tipo MMain_Menu. Es ella quien mantiene las referencias a todas las tablas de símbolos empleadas, interfaz a desarrollar (MScroll_Parameter_Window), barra para grupos, y la conexión entre los diferentes programas y líneas de comando y los menús a abrir (MOpen_Entry). El menú desplegable con el botón derecho es único para toda la aplicación y también reside en este objeto (MRight_Button). Las funciones correspondientes al grupo "File" también se encuentran en esta clase, salvo cuando se solicita una ventana para introducir comandos del sistema que se crea un nuevo objeto de tipo MSystem_Window y se le pasa a él el control. Es la aplicación la encargada de ir creando cada uno de los interfaces a partir de las tablas de símbolos. Vemos, pues, como este objeto es de vital importancia en la concepción del menú Colimate del que constituye su elemento central.

El concepto de menú para línea de comando según el modelo ampliamente comentado a lo largo de esta memoria es implementado por MParameter_Set. Sin embargo, por encima de ésta se establecen dos clases adicionales: MParameter_Window que añade los botones de "Default All" y "Exec" y MScroll_Parameter_Window que encapsula la MParameter_Window en un sólo objeto de Qt con scroll. La aplicación para abrir un interfaz solicita la apertura a MScroll_Parameter_Window, que es quien abre una ventana con scroll y pide a los objetos que tiene por debajo que se muestren. MParameter_Window es una clase intermedia sin más funcionalidad que la de comunicar MScroll_Parameter_Window con MParameter_Set, aparte de la ya señalada de añadir el botón global de valores por defecto y de ejecución. MParameter_Set es la encargada de lanzar la línea de comando, para ello necesita una referencia a la línea de comando que abrió el menú. Esta referencia viene dada por MOpen_Entry que relaciona el grupo, programa, y parsing_line con el MScroll_Parameter_Window abierto. Al ser MParameter_Set quien implementa el concepto de menú como lista de etiquetas y variables vemos como es él quien mantiene una lista de MVariable_Widget.

MVariable_Widget es la clase que representa gráficamente una variable, independientemente de su tipo. Hereda de MMenu_Layout porque es un layout horizontal de objetos, en lugar de los layout verticales utilizados en

MScroll_Parameter_Window, MParameter_Window y MParameter_Set. La principal misión de MVariable_Widget es la de servir de intermediario entre la lista de objetos del menú y el manejo específico de dichas variables según su tipo descrito por MVar_Desc_Widget, así como el mantenimiento del layout general para la variable.

MVar_Desc_Widget es una clase intermedia genérica que define el comportamiento básico de cualquier descriptor de variable. Serán las clases herederas de ésta (MOption_Desc_Widget, MList_Desc_Widget, MNumber_Desc_Widget, y MText_Desc_Widget) las que efectivamente definan las acciones a realizar y los Qt widgets a mostrar en cada momento. Es importante resaltar en este punto que son estas clases específicas las que manejan el valor de las variables que a su vez se encuentran en dos niveles diferentes. Tenemos, por un lado, el valor almacenado en la tabla de símbolos, lugar para el valor "oficial" de la variables, sin embargo el valor que está siendo editado en este momento en pantalla no es el "oficial". Este valor reside en el propio editor de línea de Colimate hasta que se pulsa la tecla de ejecución de la línea de comando, en este momento el valor de pantalla pasa a ser el oficial. La comunicación en sentido inverso, del oficial al de pantalla, también será útil en algún momento.

Obsérvese que los argumentos opcionales (representados por MOption_Desc_Widget) poseen un MParameter_Set. Éste es el mecanismo gráfico que permite la anidación de argumentos opcionales e implícitamente define una jerarquía de argumentos basada en la indentación. Es éste también el motivo por el que se ha desdoblado el modelo de menús en dos (MParameter_Set y MParameter_Window) que de otro modo podrían haber permanecido juntas.

A continuación se irán describiendo cada clase una por una. Daremos una impresión de conjunto de lo que realiza y el tipo de funciones que podemos encontrar dentro. Sin embargo, y como viene siendo habitual, me refiero a la ayuda hipertextual para una descripción pormenorizada.

Clase MMenu_Layout

Esta clase es la que se encarga de distribuir los objetos (de ahí el nombre de layout) en pantalla. Tendrá que mantener información sobre si los objetos de distribuyen en vertical u horizontal (*__vertical*), además del espacio ocupado por los objetos ya distribuidos hasta el momento (*__min_width* y *__min_height*). Hereda de QFrame que en Qt es una clase bastante general y se define como aquellas clases susceptibles de tener un marco.

```
class MMenu_Layout: public QFrame {
   Q_OBJECT
protected:
   float __min_width;
   float __min_height;
   bool  __vertical;
public:
   typedef enum {
      LEFT          =  0,
      HCENTER       =  1,
      CENTER        =  2,
      VCENTER       =  3,
      RIGHT         =  4,
      INDENT        =  5,
      DOUBLE_INDENT =  6,
      BUTTON        =  7 } ML_Position;
}
```

La forma natural de trabajar con esta clase es que la estructura en la que queremos definir una disposición de elementos herede de ésta. En la inicialización de la clase específica normalmente se debería inicializar ésta con una configuración vertical u horizontal. Después se le va pidiendo que añada los objetos (que deben ser hijos de QWidget, otra clase muy genérica de Qt) de uno en uno. A la hora de posicionar los objetos podemos elegir entre las posiciones indicadas por ML_Position y si queremos que al posicionar el siguiente objeto tomemos el recién introducido con un tamaño algo superior (10%).

Las posiciones centradas hacen referencia a los objetos ya introducidos por lo que puede ser que en el conjunto final un determinado objeto no esté absolutamente centrado. La posición CENTER equivale a HCENTER o VCENTER según la configuración del layout sea vertical u horizontal. La posición BUTTON es una posición centrada horizontalmente y ligeramente más separada en la dirección vertical de los objetos que hayan sido ya introducidos. Las posiciones de INDENT y DOUBLE_INDENT se utilizan para marcar la jerarquía de los argumentos opcionales.

Hay que hacer notar que cualquier posición a la que se hace referencia en un determinado MMenu_Layout siempre está referido al sistema de coordenadas del QWidget padre y no al global del monitor.

Clase MWidget

Esta clase decíamos que únicamente servía para mantener una referencia a la aplicación principal de forma que cualquier objeto pueda tener acceso a temas generales como pueden ser las tablas de símbolos o el menú de botón derecho. Vemos como la estructura de la clase refleja exactamente esta funcionalidad.

```
class MWidget {
protected:
   const MMain_Menu *__main;
}
```

La referencia debe ser asignada en el constructor del objeto.

Clase MLineEdit

Es ésta una extensión de la clase QLineEdit de Qt, de la cual hereda. Tiene un flag que le indica si este editor corresponde a un parámetro de tipo fichero (*__file_mode*), porque en tal caso tendrá que atender las pulsaciones del botón

derecho del ratón. Por otro lado, también es esta clase la que gestiona el histórico de parámetros introducidos en este editor, cada objeto mantiene su propio histórico. Los parámetros en sí se almacenan en *__history* mientras que *__current_history* es un índice dentro de dicha historia para el valor actual y *__max_history* nos ayudará a establecer una historia circular con los cursores, es decir, cuando lleguemos al último parámetro introducido y pidamos el siguiente con la tecla de cursor hacia abajo, entonces devolveremos el primero. Y viceversa al pasar del primero al último. Si no nos gusta ninguna de las opciones del histórico pulsaremos ESC para introducir uno nuevo.

```
class MLineEdit: public QLineEdit, public MWidget {
   Q_OBJECT
   bool   __file_mode;
   string __history[MAX_SYSTEM_HISTORY];
   int    __max_history;
   int    __current_history;
}
```

Clase MMain_Menu

Ya hemos comentado que éste es el tipo principal de la aplicación gráfica. Cuando el generador de menús termina de leer las especificaciones del interfaz, el siguiente paso es crear en memoria todos los elementos necesarios, realizar las conexiones entre ellos y posteriormente mostrar la barra de con los grupos y dejar ya que sea Qt el que gestione la apertura y cierre de las ventanas.

En esta clase se mantendrán las referencias básicas a todos los objetos del interfaz. Siguiendo una cadena descendente de referencias desde este punto se puede llegar a cualquier objeto de la aplicación. De ahí la importancia de la clase MWidget, que desde cualquier objeto nos lleva de nuevo a la aplicación principal.

```
class MMain_Menu: public QWidget {
   QMenuBar *                          __menu_bar;
   vector<MScroll_Parameter_Window *>  __SPWT;
   vector<MOpen_Entry *>               __OET;
   menu_gencompiler *                  __compiler;
   string                              __fn_in;
   MRightButton_Popup *          __rightbutton_popup;
}
```

En la estructura de la clase vemos como se tiene por un lado un puntero al menu de grupos (*__menu_bar*), punteros a todos los menús (*__SPWT*), punteros a todas las relaciones entre grupos, programas y líneas de comando y menús abiertos (*__OET*), puntero al compilador (*__compiler*) en cuyo interior se encuentran las tablas de símbolos, el nombre del fichero desde el que se leyeron las especificaciones del interfaz y por último, puntero al menú de botón derecho (*__rightbutton_popup*), que como ya indicamos es común a toda la aplicación.

Entre la funcionalidad de la clase encontramos:

- Crear las ventanas de cada menú a partir de las especificaciones de los menús. El proceso básicamente es el que sigue: se van leyendo cada uno de los menús de la tabla de menús. Para cada uno de ellos se crea un objeto nuevo de tipo MScroll_Parameter_Window y se le solicita a ese nuevo objeto que introduzca los elementos necesarios para representar al menú en cuestión. Esta llamada se va pasando de clase en clase (lo veremos con más detalle al estudiar cada clase) hasta que el interfaz queda completamente configurado para este menú. Entonces se pasa al siguiente elemento de la lista.
- Crear el interfaz primario de usuario, es decir, la barra de grupos y programas. Conforme se va creando se anotan en forma de MOpen_Entry las relaciones entre los grupos, programas y líneas de comando. Una MOpen_Entry lo que viene a decir en resumen es "La línea de comando tal del programa cuál del grupo más allá abre el interfaz contenido en tal MScroll_Parameter_Window".
- Crear el menú de botón derecho
- Asignación del compilador y acceso a cada una de las tablas de símbolos
- Acceso al botón derecho

- Acceso a cada menú
- Slots para recibir los eventos asociados al grupo común "File". Estos mismos slots se encargan de manipular dicho evento. En el caso de solicitarse un editor de línea para comandos del sistema, se crea una nueva MSystem_Window y se le cede el control. Así podemos disponer de varias ventanas desde las que lanzar comandos directamente al sistema operativo.
- Otras funciones relacionadas con la funcionalidad de esta clase como objeto de Qt.

Clase MOpen_Entry

El objetivo de esta clase es meramente el de relacionar las línea de comando y programas del interfaz primario con los menús a desplegar. Su estructura es muy sencilla y básicamente contiene la de la línea de comando (*__Gr*, *__Pr*, *__PL*) y el menú a abrir (*__SPW*).

```
class MOpen_Entry {
   const MGroup *                __Gr;
   const MProgram *              __Pr;
   const MParsing_Line *         __PL;
   MScroll_Parameter_Window *    __SPW;
}
```

Las funciones de la clase únicamente atienden al acceso y asignación de los valores de estas variables.

Clase MScroll_Parameter_Window

Esta clase junto con las dos siguientes (*MParameter_Window* y *MParameter_Set*) forman los interfaces gráficos para los menús. Se ha tenido que dividir esta funcionalidad en varias clases por razones diversas: en primer lugar, porque un *scroll* en Qt se puede aplicar a un único objeto con lo que ya hay tenemos una división. La segunda responde al hecho de que dado un argumento opcional puede existir un conjunto de parámetros a los que no podemos añadirles botones.

La estructura de la clase ilustra la primera de las divisiones mencionadas: un objeto (*__pw*), al que se le añade un scroll (*__scroll*).

```
class MScroll_Parameter_Window:
   public MMenu_Layout, public MWidget {
   QScrollView *                        __scroll;
   MParameter_Window *                  __pw;
}
```

Además, en Qt todos los objetos gráficos tienen un padre. Los únicos objetos que no tienen padre son los que abren ventana al pedirles que se muestre, el resto se muestran en la ventana del primero de sus ascendientes. Pues bien, estos objetos de tipo *MScroll_Parameter_Window* no tienen padre, es decir, serán ellos los que abran la ventana y el resto de los objetos se mostrarán en ella.

Entre las funciones de esta clase se cuentan:

- Crear el menú correspondiente a partir de una especificación de menú (*MMenu*). Esta función realmente descansa en la de *MParameter_Window*, una vez que *MParameter_Window* ha creado el menú como un único objeto, *MScroll_Parameter_Window* la posiciona en la ventana y añade el scroll.
- Obtener valores de la tabla o del interfaz: estas funciones las encontraremos a lo largo de toda la estructura de clases hasta llegar a las descripciones particulares de los tipos de parámetros que son realmente quienes saben cómo resolverlas. De modo que lo que irán haciendo cada una de las clases es propagar la llamada a todos los objetos que tengan por debajo capaces de responder a este método, así hasta que llega a los descriptores de variable. El significado de estas funciones es la de pasar de los valores mostrados en el interfaz a los valores oficiales de la variable y viceversa (ver la introducción a las clases hecha al comienzo de esta misma sección).
- Abrir la ventana. Se entiende por abrir la ventana no al simple hecho de mostrarla sino al resultado de una selección por parte del usuario de un comando de línea en el interfaz primario. Así, pues, al abrir la ventana tendremos que anotar (esta anotación se realiza en la clase *MParameter_Set* por lo que se propaga la llamada a métodos equivalentes hasta que llega a esta clase) quién fue la línea de comando

que solicitó la apertura de la ventana. Esto es así debido al hecho de que varios programas y diferentes líneas de comando pueden compartir un mismo menú. Al final al pulsar el botón de ejecución tendremos que dilucidar cuál de las posibles líneas de comando debe lanzarse.

Clase MParameter_Window

Esta clase muestra un conjunto de parámetros (*__ps*) y le añade los botones de recoger todos los parámetros por defecto (*__default*) y de ejecución de la línea de comandos (*__exec*), correspondiente al Enter en el línea del sistema.

```
class MParameter_Window:
   public MMenu_Layout, public MWidget {
   MParameter_Set                    *__ps;
   QPushButton                       *__exec;
   QPushButton                       *__default;
   MOpen_Entry                       *__oe;
}
```

Las funciones de esta clase se limitan fundamentalmente a propagar las llamadas y a conectar los botones con los slots adecuados en de MParameter_Set:

- Crear el interfaz a partir del menú: lo que es creación del interfaz se solicita a *MParameter_Set*. Esta función crea los dos botones ya mencionados, distribuye espacialmente los tres objetos y conecta las señales producidas por cada botón a los slots de *MParameter_Set* encargados uno de cargar todos los valores por defecto y otro de generar y lanzar la línea de comando.
- Obtener valores de la tabla o del interfaz: simple propagación de los métodos hacia *MParameter_Set*.
- Abrir la ventana. En este momento se cambia el texto del botón de ejecución para que dé información sobre el programa y la línea de comando que abren el menú. Se pide a *MParameter_Set* que deshabilite los argumentos no válidos para esta línea de comando, que haga la anotación de la *MOpen_Entry*, y que capture todos los valores de las variables de la tabla de variables.

Clase MParameter_Set

La clase *MParameter_Set* será la encargada de mantener un conjunto de parámetros en la ventana. Tiene principalmente dos usos: por una parte, organiza las variables y etiquetas presentes en un menú, por otra, gestiona y crea el interfaz correspondiente a un argumento opcional. Es por esta razón por la que se ha tenido que subdividir aún más la funcionalidad de mostrar un menú. En cuanto a su estructura vemos que del modelo de menú con el que hemos venido trabajando hasta el momento, únicamente se almacenan las referencias a los parámetros (*__prm_list*). Las etiquetas de texto que sirven como separadores de argumentos son también creadas por esta clase, se le informa a Qt de que el padre de dichas etiquetas (de tipo *QLabel*) es este objeto, pero no necesitamos mantener una referencia explícita en la estructura puesto que no realizaremos en ningún momento operaciones con ella. También almacenaremos como parte de la estructura un puntero a la *MOpen_Entry* que actualmente nos indica qué línea de comando deberemos lanzar cuando se nos pida. En caso de que el *MParameter_Set* pertenezca a un argumento opcional esta referencia se mantiene a NULL puesto que no hay ninguna línea que generar.

```
class MParameter_Set:
   public MMenu_Layout, public MWidget {
   vector<MVariable_Widget *>  __prm_list;
   const MOpen_Entry *         __oe;
}
```

Las funciones desarrolladas por la clase son:

- Crear el conjunto de parámetros desde un menú (*MMenu*) o desde una línea de comandos. Las etiquetas separadoras son hijas de este objeto y las variables (*MVariable_Widget*) que se necesiten, también. Una vez creado el objeto manipulador de variables se le solicita a él mismo que se configure con arreglo a la variable que le ha tocado.
- Deshabilitar los argumentos no usados por una determinada línea de comando. De este modo evitaremos errores involuntarios por parte del usuario. Esta función suele ser llamada en el momento de abrir la ventana.
- Comunicar un estado de habilitación/deshabilitación a todas las variables que

cuelguen de él.

- Anotar la línea de comando que abre la ventana. Esta anotación también se realiza en la apertura de la misma.
- Lanzar dicha línea de comando. Lo que hace esta función realmente es pedirle a *MParsing_Line* que lance el comando.
- Obtener valores de la tabla, del interfaz, o de los valores por defecto aquellos que los tengan. Simplemente se propaga la llamada hacia las variables hijas.

Clase MVariable_Widget

Esta clase sirve de interfaz entre el resto de la estructura gráfica y la gestión específica de variables. Su principal misión es la de dada una variable (*MVariable*) decidir a qué tipo específico de manipulación gráfica de variables pertenece, crear un nuevo objeto de ese tipo (heredero de *MVar_Desc_Widget*) y pedirle que se configure con arreglo a la variable concreta asignada. Esto se hace en el constructor de la clase. El resto de los métodos solicitados a una variable (tomar el valor del interfaz, de la tabla, de un posible valor por defecto, cambiar su estado de habilitación, ...) son propagados hacia el descriptor de variable.

Obsérvese que a pesar de conocer distintos tipos de manipulación de variable, esta clase contiene un único puntero a MVar_Desc_Widget, una clase abstracta de manipulación gráfica de la que heredan los manipuladores efectivos. Esto es así gracias a las características hereditarias proporcionadas por C++, de otro modo sería imposible mantener una estructura elegante que representase esta dependencia.

En cuanto a la estructura de esta clase vemos como tiene únicamente una referencia a la variable que representa (*__Vr*) y otra al descriptor y manipulador gráfico (*__var_desc_widget*).

```
class MVariable_Widget:
   public MMenu_Layout, public MWidget {
   MVar_Desc_Widget   *__var_desc_widget;
   MVariable          *__Vr;
```

```
}
```

clase MVar_Desc_Widget

Es ésta una clase abstracta en el sentido de que no implementa ninguna de las características básicas necesarias para el manejo del tipo de variables que nos concierne, a pesar de que posee ciertos objetos gráficos que ella misma es capaz de manejar. La idea es que los controladores específicos de los tipos de datos hereden de ésta. Parte de la funcionalidad vendrá implementada aquí mientras que los métodos más relacionados con el tipo de dato en concreto se implementará en la clase derivada.

Esta clase posee una referencia a la variable a la que representa (*__Vr*), la etiqueta que se muestra en pantalla como descripción del parámetro (*__label*) junto con la ayuda adicional y un botón para capturar el valor por defecto (*__default*) que sólo existirá en el caso de que haya un valor por defecto que tomar.

```
class MVar_Desc_Widget:
   public MMenu_Layout, public MWidget {
   protected:
      MVariable *       __Vr;
      QLabel *          __label;
      QPushButton *     __default;
}
```

Salvo la creación y habilitación de la etiqueta y el botón de valor por defecto que son gestionados por este objeto, el resto de las llamadas son transmitidas a los manipuladores específicos gracias al mecanismo de herencia y funciones virtuales proporcionado por C++.

Clases de Manipulación gráfica según el tipo de Variable

En este parágrafo incluimos las clases *MText_Desc_Widget*, *MNumber_Desc_Widget*, *MList_Desc_Widget*, y *MOption_Desc_Widget*. Salvo aquellos aspectos esenciales para la comprensión del funcionamiento del generador de

menús, no entraremos en las estructuras particulares de cada una de ellas. Nos basta con conocer que tendrán un editor de línea (*MLineEdit*), alguna etiqueta informativa sobre el rango (por ejemplo, en números), una lista de Qt (*MList_Desc_Widget*), o un botón para habilitar un conjunto de variables (*MVariable_Set*) en los argumentos opcionales.

Son estas clases las que saben qué objetos gráficos de Qt son necesarios para presentar el parámetro al usuario, cómo pasar los valores en el interfaz a la tabla de variables y viceversa, cómo leer los valores por defecto, y cómo habilitar/ un argumento.

En el caso concreto de las listas, cada vez que se selecciona un nuevo elemento de la lista se ejecutan las acciones asociadas a el mismo. En cuanto a los argumentos opcionales, toda la funcionalidad descansa sobre *MParameter_Set*, de este modo aseguramos una jerarquía sólida y bien fundamentada. Los botones para habilitar/deshabilitar la opción transmiten el cambio de estado al conjunto de parámetros consiguiendo así la habilitación/deshabilitación de todo el bloque.

Clase MSystem_Window

Esta clase es la de las pequeñas de ventanas que hacen de editor de línea para luego lanzar los comandos al sistema. No tienen padre desde el punto de vista de Qt, es decir, que abrirán ellas mismas la ventana. Se abren tantas ventanas de sistema como veces se seleccione la opción "System Command" del grupo "File". Su estructura es muy simple, tan sólo se guarda información del editor de línea utilizado mientras que el botón de ejecución asociado carece de referencia y su señal de pulsado se conecta al slot de ejecución de esta clase.

```
class MSystem_Window:
   public QFrame, public MWidget {
   MLineEdit  * __command;
}
```

Clase MRightButton_Popup

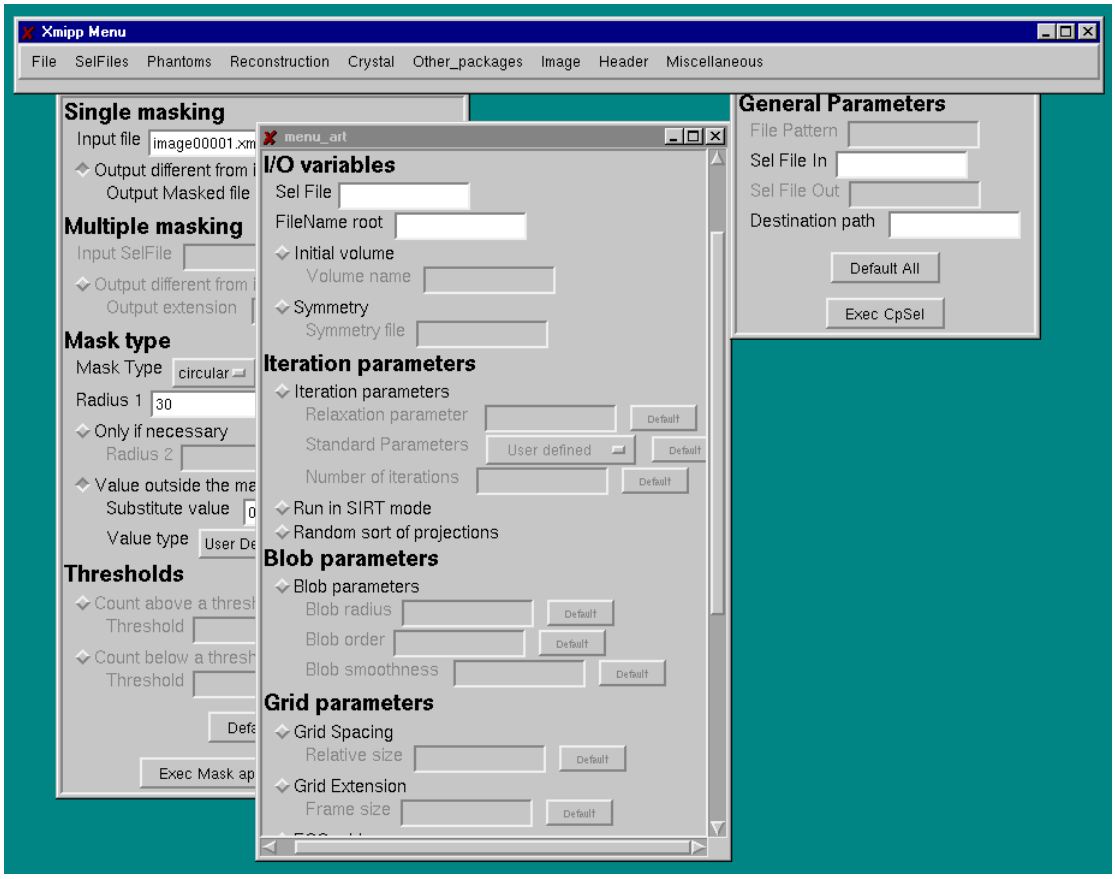
Esta clase es la del menú que aparece al pulsar el botón derecho del ratón, en él aparecerán las diferentes programas posibles a ejecutar sobre parámetros de tipo fichero. Como mínimo siempre estará la opción de seleccionarlo de la estructura de directorios por medio de un interfaz gráfico. Este menú se crea a partir de un programa especial llamado "RightButton" de la tabla de programas y es compartido por todos los parámetros de tipo fichero de la aplicación. Cada vez que pulsamos el botón derecho sobre uno de ellos, el editor de línea de dicho parámetro solicita a la aplicación que comunique al menú de botón derecho quién realizó la solicitud de apertura y que despliegue el menú en el punto en que se pulsó el botón. El contenido del parámetro se volcará en la variable "$rightbutton_FILE" de la tabla de variables que es la que se utiliza para generar las líneas de comando asociadas a las diferentes acciones.

La estructura de la clase refleja exactamente el comportamiento anteriormente descrito:

```
class MRightButton_Popup:
   QPopupMenu, public MWidget {
   MLineEdit *    __editor;
   MProgram  *    __rightbutton_prog;
}
```

2.6. Resultados

Colimate ha sido empleado con éxito en el desarrollo de un interfaz gráfico más amigable que el comando de línea. El paquete elegido a tal efecto es Xmipp (X−Windows based Microscopy Image Processing Package) del propio grupo al que pertenezco en el Centro Nacional de Biotecnología. La siguiente figura ilustra algunas de las ventanas producidas por Colimate para dicha aplicación.



A modo de resumen rápido del paquete utilizado como ejemplo podemos indicar que se trata de un conjunto de programas orientados a la reconstrucción tridimensional de macromoléculas a partir de imágenes de proyección obtenidas con un microscopio electrónico. Estas reconstrucciones se consiguen al combinar adecuadamente miles de partículas, formando un sistema de ecuaciones inmenso de cuya resolución obtenemos el volumen deseado. Aunque sea éste el objetivo final de Xmipp, no podemos olvidar otros módulos del mismo tales como el preprocesado bidimensional, clasificación de imágenes, extracción de características, programas de

servicio, ...

En este capítulo estudiaremos los ficheros más relevantes de la aplicación (desde el punto de vista del desarrollo del interfaz gráfico) a modo de ejemplo de uso de Colimate. Analizaremos por un lado un menú simple pero compartido por varios programas como es el de los SelFiles (son éstos unos ficheros de selección de partículas dentro de Xmipp) mostrado en la figura anterior a la derecha, otro más complejo específico para el programa de reconstrucción ART (encargado de la resolución del sistema de ecuaciones cuyo resultado es el volumen tridimensional de la proteína), y por último, un caso de programa con varios modos de funcionamiento (el programa de enmascaramiento de imágenes/volúmenes que además realiza funciones como la cuenta de valores dentro de una máscara por encima de un determinado valor, generación de la máscara sin aplicarla, o generación de una imagen/volumen en blanco del mismo tamaño que el fichero de entrada). Aunque la estructura dada al programa Colimate ejemplo es la recomendada, no es la única y cualquier programador puede acomodar la estructuración de las directivas de Colimate según sus gustos y necesidades.

2.6.1. Módulo principal de la aplicación Colimate

La aplicación general Colimate para Xmipp tiene un programa principal cuyo contenido es

```
/*Colimate:
   // Groups ---------------------------------------------
   GROUP SelFiles {
      PROGRAM DoSel;
      PROGRAM CpSel;
      PROGRAM MvSel;
      PROGRAM RmSel;
   }

   GROUP Phantoms {
      PROGRAM Create_phantom;
      PROGRAM Evaluate;
      PROGRAM Project;
```

```
   PROGRAM Surface;
   PROGRAM Random_phantom;
   PROGRAM Recons_test;
}

GROUP Reconstruction {
   PROGRAM Ang_distribution;
   PROGRAM Art;
   PROGRAM Surface;
   PROGRAM Symmetrize;
}

GROUP Crystal {
   PROGRAM Spots2RealSpace2D;
}

GROUP Other_packages {
   PROGRAM Adapt_for_spider;
   PROGRAM Raw22spi;
}

GROUP Image {
   PROGRAM Mask;
   PROGRAM Threshold;
}

GROUP Header {
   PROGRAM Infogeo;
   PROGRAM Assign_angles;
}

GROUP Miscellaneous {
   PROGRAM DocFile_Histogram;
   PROGRAM Are_different;
   PROGRAM Combine_stats;
}

// Programs ------------------------------------------
#include "../Batches/Do_Selfile/do_sel.mnu"
#include "../Src/CpSel/cpsel.cc"
#include "../Src/MvSel/mvsel.cc"
#include "../Src/RmSel/rmsel.cc"
```

```
   #include "../Src/Create_phantom/create_phantom.cc"
   #include "../Src/Evaluate/evaluate.cc"
   #include "../Src/Project/project.cc"
   #include "../Src/Surface/surface.cc"
   #include "../Src/Random_phantom/random_phantom.cc"
   #include "../Src/Recons_test/recons_test.cc"

   #include "../Src/Ang_distribution/ang_distribution.cc"
   #include "../Src/Art/art.cc"
   #include "../Src/Symmetrize/symmetrize.cc"

   #include "../Src/Spots2RealSpace2D/spots2realspace.cc"

   #include "../Src/Adapt_for_Spider/adapt_for_spider.cc"
   #include "../Src/Raw22spi/raw22spi.cc"

   #include "../Src/Mask/mask.cc"
   #include "../Src/Threshold/threshold.cc"

   #include "../Src/Infogeo/infogeo.cc"
   #include "../Src/Assign_angles/assign_angles.cc"

   #include "../Src/DocFile_Histogram/docfile_histogram.cc"
   #include "../Src/Are_different/are_different.cc"
   #include "../Src/Combine_stats/combine_stats.cc"

   // Right button ----------------------------------------
   RIGHTBUTTON {
      + xv:        xv        $FILE
      + nedit:     nedit     $FILE
      + ghostview: ghostview $FILE
      + acroread:  acroread  $FILE
   }
*/
```

En él vemos que hemos definido los grupos de programas del interfaz primario, indicamos dónde encontrar todos y cada uno de los programas organizados por grupos (en realidad no es tan sólo una indicación de dónde encontrarlos sino que se incluyen y se compilan en ese momento) y, por último, damos las acciones a realizar sobre los parámetros de tipo fichero cuando se pulse el botón derecho del ratón sobre ellos (además de estas opciones explícitamente definidas siempre

tendremos la posibilidad de seleccionarlo con un interfaz gráfico de entre la estructura de directorios).

Todos los programas se desarrollan de forma análoga por lo que únicamente veremos un par de ejemplos en el que se ilustren las principales ideas de programación con Colimate.

2.6.2. Menús compartidos

Comenzaremos por analizar el menú del programa DoSel. DoSel corresponde a un script de Unix, no es un programa C++, cuya línea de comando habitual es

```
do_selfile "<file_pattern>" > <selfile>
```

Al tratarse de un programa no escrito en C++ tiene un fichero aparte con las especificaciones a Colimate. El contenido de do_sel.mnu es

```
/*Colimate:
   PROGRAM DoSel {
      OPEN MENU menu_selfile;
      COMMAND LINES {
        + usual: do_selfile $FILE_PATTERN > $SELFILE_OUT
      }
      PARAMETER DEFINITIONS {
        $FILE_PATTERN {
           label="File Pattern";
           help="Files which will form the SelFile";
           type=file pattern;
        }
        $SELFILE_OUT {
           label="Sel File Out";
           help="Generated Sel File";
           type=file;
        }
      }
   }
```

```
MENU menu_selfile {
      "General Parameters"
      $FILE_PATTERN
      $SELFILE_IN
      $SELFILE_OUT
      $PATH
   }
*/
```

Vemos como el programa sigue la gramática de Colimate para definir cada uno de los parámetros a utilizar, cómo emplearlos, cómo decir menú se abre cuando se llame a este programa, etc. La estructura del menú también es sencilla y refleja fielmente el modelo de interfaz gráfica propuesto. Sin embargo, en la definición del propio menú vemos que aparecen unas variables que todavía no han sido definidas, a saber, $SELFILE_IN y $PATH. Esto se debe a que este menú está compartido por otros programas de manera que será en ellos en los que se definan estos argumentos. Cuando la aplicación para Xmipp entre a este menú a través de DoSel, estos dos parámetros que no participan en la línea de comando de DoSel se inhabilitarán de suerte que no se lleva a confusión al usuario. Es de reseñar el hecho de que se pueden utilizar variables en un menú sin haberlas definido, esto se permite para simplificar la programación en Colimate y es tan sólo después de haber leído todo el programa Colimate que se realiza una comprobación de que todos los elementos referenciados realmente existen y tienen las propiedades adecuadas.

Veamos cómo hay que especificar el resto de los programas que comparten el interfaz, como por ejemplo CpSel y MvSel. CpSel es el siguiente programa que lee Colimate, y hace uso de las otras dos variables del interfaz. Es por ello que al no haber sido definidas todavía, son explícitamente especificadas en este punto. Sin embargo, obsérvese que ya no hace falta declarar nada referente al menú, salvo que CpSel abre ese menú.

```
/*Colimate:
   PROGRAM CpSel {
      OPEN MENU menu_selfile;
      COMMAND LINES {
        + usual: cpsel $SELFILE_IN $PATH
      }
      PARAMETER DEFINITIONS {
         $SELFILE_IN {
            label="Sel File In";
            help="Input Selfile";
            type=file existing;
         }
         $PATH {
            label="Destination path";
            help="Where to perform operation";
            type=file;
         }
      }
   }
*/
```

En cuanto a MvSel, la situación es aún más sencilla, puesto que ya todos los elementos del interfaz han sido especificados. Lo único que tendremos que hacer será definir su línea de comando y comunicarle a Colimate que debe abrir el mismo menú que para los programas anteriores.

```
/*Colimate:
   PROGRAM MvSel {
      OPEN MENU menu_selfile;
      COMMAND LINES {
        + usual: mvsel $SELFILE_IN $PATH
      }
   }
*/
```

2.6.3. Menús complejos

En este segundo ejemplo desarrollaremos un menú más complejo en el que se hace un uso intensivo de los argumentos opcionales. Aunque éste pueda parecer el punto más oscuro de la definición Colimate, veremos cómo no entraña ninguna dificultad. El programa de reconstrucción ART con multitud de opciones aunque con un sólo modo de funcionamiento nos servirá de base para el ejemplo. El interfaz Colimate correspondiente es mostrado en la figura con la que comenzamos este capítulo como el menú intermedio, mientras que la línea de comando se resume en la siguiente tabla.

```
art [Parameters]
```

donde los parámetros son

Input/Output parameters	Iteration parameters
-i <selection file>	**-l <lambda=0.047>**
-o <output file root name>	**-n <no_iterations=1>**
-start <starting blob volume="">	**-SIRT**
-sym <symmetry file="">	**-random_sort**
Blob parameters	Grid parameters
-r <blob_radius=2>	**-g <grid_relative_size=2.26>**
-m <blob_order=2>	**-FCC | -CC**
-a <blob_alpha=3.6>	**-ext <proj_ext=0>**
Debugging options	
-show_error	
-show_stats	
-save_at_each_step	
-save_intermidiate	
-save_blobs	
-manual_order	

No es éste el momento de explicar el significado de cada uno de los parámetros, las combinaciones que podemos hacer con ellos ni como afectan a la resolución del sistema de ecuaciones. Bástenos saber que de todos los parámetros los únicos obligatorios son el fichero de entrada con la selección de imágenes y el

nombre de los ficheros de salida. El código para generar el interfaz correspondiente se muestra en las páginas siguientes.

```
/*Colimate:
   PROGRAM Art {
      OPEN MENU menu_art;
      COMMAND LINES {
      + usual: art -i $SELFILE -o $FNROOT
          [-start $STARTVOL]                                  //00
          [-sym $SYMFILE]                                     //01
          [-l $LAMBDA $STANDARD_PRMS -n $NO_IT]               //02
          [-SIRT]                                             //03
          [-random_sort]                                      //04
          [-r $BLOB_RADIUS -m $BLOB_ORDER -a $BLOB_ALPHA]     //05
          [-g $REL_SIZE]                                      //06
          [-ext $EXT]                                         //07
          [-FCC]                                              //08
          [-CC]                                               //09
          [-show_error]                                       //10
          [-show_stats]                                       //11
          [-save_at_each_step]                                //12
          [-save_intermidiate]                                //13
          [-save_blobs]                                       //14
          [-manual_order]                                     //15
      }
      PARAMETER DEFINITIONS {
        $SELFILE {
           label="Sel File";
           help="File with all input projections";
           type=file existing;
        }
        $FNROOT {
           label="FileName root";
           help="A .hist and .vol files are generated";
           type=file;
        }
        $Art_00000
           {label="Initial volume"; help="Voxel Xmipp format";}
           $STARTVOL {
              label="Volume name";
              help="Blob Xmipp format";
              type=file existing;
```

```
   }
$Art_00001 {label="Symmetry";}
   $SYMFILE {
      label="Symmetry file";
      help="This file has got a complex structure, "
           "better see Web help";
      type=file existing;
   }
$Art_00002 {label="Iteration parameters";}
   $LAMBDA {
      label="Relaxation parameter";
      help="Controls the convergence speed";
      type=float [-2...2];
      by default="0.0047";
   }
   $NO_IT {
      label="Number of iterations";
      help="Usually no more than 1 or 2";
      type=integer [1 ...];
      by default="1";
   }
   $STANDARD_PRMS {
      label="Standard Parameters";
      help="Some useful combinations";
      type=list {
          "User defined"
          "Negative staining" {$LAMBDA="0.047";}
          "Cryo microscopy"   {$LAMBDA="0.0047";}
      };
   }
$Art_00003 {label="Run in SIRT mode";}
$Art_00004 {
   label="Random sort of projections";
   help="Only valid in ART";
}
$Art_00005 {label="Blob parameters";}
   $BLOB_RADIUS {
      label="Blob radius";
      help="in Voxels";
      type=float [0...];
      by default="2";
   }
```

```
      $BLOB_ORDER {
         label="Blob order";
         help="Derivative order of the Bessel "
              "function, must be a natural";
         type=natural;
         by default="2";
      }
      $BLOB_ALPHA {
         label="Blob smoothness";
         help="The higher the value, the "
              "sharper the blob";
         type=float [0...];
         by default="3.6";
      }
   $Art_00006 {label="Grid Spacing";}
      $REL_SIZE {
         label="Relative size";
         help="in voxels";
         type=float;
         by default="2.26";
      }
   $Art_00007 {label="Grid Extension";}
      $EXT {
         label="Frame size";
         help="in voxels, must be a natural";
         type=natural;
         by default="6";
      }
   $Art_00008 {label="FCC grid";}
   $Art_00009 {label="CC grid";}
   $Art_00010 {label="Show error on each projection";}
   $Art_00011
      {label="Show statistics on each projection";}
   $Art_00012 {label="Save volumes at each projection";}
   $Art_00013 {label="Save volumes at each iteration";}
   $Art_00014 {label="Save blob volumes";}
   $Art_00015 {label="Projection order manually given";}
 }
}
```

```
MENU menu_art {
      "I/O variables"
      $SELFILE
      $FNROOT
      $Art_00000
      $Art_00001
      "Iteration parameters"
      $Art_00002
      $Art_00003
      $Art_00004
      "Blob parameters"
      $Art_00005
      "Grid parameters"
      $Art_00006
      $Art_00007
      $Art_00008
      $Art_00009
      "Debugging options"
      $Art_00010
      $Art_00011
      $Art_00012
      $Art_00013
      $Art_00014
      $Art_00015
   }
*/
```

Aunque a primera instancia se antoja un poco largo, ni que decir tiene cuál sería la extensión, la complejidad y el esfuerzo de programación que deberíamos emplear para programar un menú de semejantes características utilizando cualquiera de las herramientas habituales de programación.

Vemos en primer lugar que la línea de comando para ART es ciertamente compleja, aunque con todo se puede modelar sin anidación de argumentos opcionales. La capacidad de Colimate para desechar comentarios nos permite numerarlos de forma cómoda, de manera que luego podamos hacer referencia a ellos inequívocamente.

Acto seguido, iremos especificando cada uno de los argumentos y parámetros. Mención especial requieren en este caso los argumentos opcionales. Únicamente necesitan que se les asigne una etiqueta, que será la que figure al lado del botón que lo habilita, ya que el tipo es automáticamente determinado por Colimate. Eventualmente podrá tener una ayuda adicional. Cuando hay parámetros dependientes de un argumento opcional, se puede indentar su definición para constatar gráficamente dicha dependencia. Por último, cuando en el menú queremos hacer referencia a los parámetros opcionales, tendremos solamente que incluir la variable opcional de la que dependen, así vemos que para representar los parámetros asociados con la definición del blob, únicamente hacemos referencia a $Art_00005, Colimate se encargará de desarrollar todos los parámetros dependientes en ese momento.

La variable $STANDARD_PRMS nos muestra el uso de las variables tipo lista. Vemos cómo está diseñada para asignar el valor de la constante de relajación del algoritmo ART en dos casos habituales de microscopía electrónica. Sin embargo, no generará texto en la línea de comando cuando ésta sea lanzada.

2.6.4. Programas con distintos modos de funcionamiento

El programa Mask de Xmipp, cuyo interfaz es el de la izquierda en la figura que inicia este capítulo, es un claro ejemplo de programa con diferentes modos de funcionamiento: por un lado enmascara una imagen simple, un conjunto de imágenes, cuenta los voxels/pixels con una determinada característica ya sea dentro de una máscara o en todo el volumen/imagen), crea una máscara o crea una imagen en blanco. La estructura de su línea de comandos se deduce del extracto de la página de ayuda de dicho programa mostrado a continuación.

```
mask [Parameters]
```

Where Parameters are:

`-i <image or volume> -o <output image or output volume>`

Use that image or volume as input for the program, the output, in case it is necessary and you want it to be different to the input file, must be given by the user.

`-i <selfile> -oext <output extension>`

Use the images and volumes in the selfile as inputs for the program, the outputs, in case it is necessary and you want them to be different to the input files, must be given by the user.

`-mask`

Generates a mask with the given specifications (See below how to define a mask)

`-save_mask`

Apply a mask and save the masks generated for the selected regions with names: mask2D and mask3D

`-create_mask`

Don't apply the mask, only generate it and save it.

`-count_above <threshold>`

Count the number of pixels/voxels within the selected region with a value greater or equal than the threshold

`-count_below <threshold>`

Count the number of pixels/voxels within the selected region with a value smaller or equal than the threshold

`-substitute <val=0>|min|max`

When a mask is applied susbstitute the values outside the mask by a value (by default, 0), the minimum or the maximum of that image/volume

El modo de funcionamiento se elige a través de los parámetros que suministremos, si damos unos entonces hará tal cosa mientras que si proporcionamos otros, la acción será diferente. El siguiente programa Colimate refleja la forma de trabajar con esta funcionalidad múltiple.

```
/*Colimate:
   PROGRAM Mask {
      OPEN MENU menu_mask;
      COMMAND LINES {
        + apply_single_mask: mask -i $FILE_IN
            [-o $FILE_OUT]                                  //00
             -mask $MASK_TYPE $MASK_TYPEL $R1 [$R2]         //01
            [-substitute $VAL $MINMAX]                      //02
        + apply_multiple_mask: mask -i $SELFILE_IN
            [-oext $OEXT]                                   //03
             -mask $MASK_TYPE $MASK_TYPEL $R1 $Mask_00001
        + create_mask: mask -i $FILE_IN
             -mask $MASK_TYPE $MASK_TYPEL $R1 $Mask_00001
             -create_mask
        + create_blank : mask -i $FILE_IN -create_mask
        + count_voxels: mask -i $FILE_IN
            [-count_above $ABOVE_TH]                        //04
            [-count_below $BELOW_TH]                        //05
        + count_voxels_within_mask: mask -i $FILE_IN
             -mask $MASK_TYPE $MASK_TYPEL $R1
             $Mask_00001 $Mask_00004 $Mask_00005
      }
      PARAMETER DEFINITIONS {
        $FILE_IN {label="Input file"; type=file existing;}
        $Mask_00000 {label="Output different from input";}
           $FILE_OUT {
              label="Output Masked file";
              help="If not given, the output is the same "
                   "as input";
              type=file;
           }

        // Definitions of $MASK_TYPE $MASK_TYPEL $R1 and $R2
        #include "mask.mnu"


        $Mask_00001 {label="Only if necessary";}

        $Mask_00002 {label="Value outside the mask";}
           $VAL {
              label="Substitute value";
              help="For values outside the mask";
              type=text;
              by default="0";
```

```
            }
            $MINMAX {
               label="Value type";
               type=list {
                  "User Defined"
                  "min" {$VAL="min";}
                  "max" {$VAL="max";}
               };
            }
         $SELFILE_IN {label="Input SelFile"; type=file existing;}
         $Mask_00003 {label="Output different from input";}
            $OEXT {
               label="Output extension";
               help="If not given, the output files are the "
                    "same as the input ones";
               type=text;
            }
         $Mask_00004 {label="Count above a threshold";}
            $ABOVE_TH {label="Threshold"; type=float;}
         $Mask_00005 {label="Count below a threshold";}
            $BELOW_TH {label="Threshold"; type=float;}
      }
   }

   MENU menu_mask {
      "Single masking"
      $FILE_IN
      $Mask_00000
      "Multiple masking"
      $SELFILE_IN
      $Mask_00003
      "Mask type"
      $MASK_TYPEL
      $R1
      $Mask_00001
      $Mask_00002
      "Thresholds"
      $Mask_00004
      $Mask_00005
   }
*/
```

Nótese como desde una línea de comando se pueden emplear argumentos opcionales que han sido definidos en otra línea diferente (desde la línea "apply_multiple_mask" se hace uso de $Mask_00001 definida en la línea anterior, o desde "count_voxels_within_mask" se pueden utilizar los argumentos para el umbral). También es reseñable en este programa la inclusión de otro fichero con definiciones Colimate, esto es así porque las máscaras son utilizadas por varios programas al mismo tiempo, por lo que los parámetros para definirlas son comunes a todos ellos. De esta forma podemos centralizar todas las definiciones referentes a la forma de las máscaras en un único fichero cuyo contenido se muestra a continuación.

```
/*Colimate:
        $MASK_TYPE {
           label="Mask type, not shown!!";
           type=text;
        }
        $MASK_TYPEL {
           label="Mask Type";
           type=list {
              "circular" {$MASK_TYPE="circular";}
              "crown"    {$MASK_TYPE="crown";}
           };
        }
        $R1 {
           label="Radius 1";
           help="Valid for circular and crown masks,
                 negative values mean towards inside";
           type=float;
        }
        $R2 {
           label="Radius 2";
           help="Valid for crown masks";
           type=float;
        }
*/
```

La gestión de modos de funcionamiento múltiple por parte de Colimate tiene la ventaja adicional de que el gestor de menús se encarga de deshabilitar aquellos argumentos y parámetros que no van a ser usados por la línea de comando que abrió el interfaz evitando de este modo que el usuario introduzca valores para variables que

realmente no se van a usar pudiendo llevar este hecho a confusión. Con la gestión realizada por Colimate aseguramos que únicamente se puedan editar aquellos campos que efectivamente contribuirán a la formación de la línea de comando deseada.

Capítulo 3. Conclusiones

En este proyecto se ha diseñado con éxito un lenguaje de generación de interfaces gráficas de usuario orientado exclusivamente al desarrollo de interfaces amigables que sirvan de nivel intermedio entre el usuario y el interfaz de línea ofrecido por el sistema operativo. Se ha cumplido el requisito inicial de sencillez en la programación frente a una cierta potencia del interfaz, lo cual lo podemos resaltar como su principal característica. El precio pagado por tal sencillez de programación ha sido el sacrificio de la versatilidad de los interfaces, solamente se pueden diseñar interfaces de un tipo. Sin embargo, este inconveniente es perfectamente asumible en la inmensa mayoría de las aplicaciones basadas en un interfaz de línea de comando, al mismo tiempo que no perdemos la posibilidad de interactuar con ellos a través de ficheros por lotes.

Se ha realizado una prueba completa y suficientemente compleja que ha permitido valorar la capacidad semántica de Colimate, la calidad de la interfaz que ofrece y el ahorro sustancial de tiempo/esfuerzo de programación.

La portabilidad del lenguaje se ha basado en la portabilidad de las herramientas empleadas para su desarrollo, por lo que se ha demostrado la posibilidad de funcionar en una amplia gama de plataformas Unix, al mismo tiempo que se ha procurado la compatibilidad con el sistema operativo Windows.

También se ha documentado ampliamente dicho lenguaje, tanto a nivel de usuario del lenguaje Colimate, como de usuario de aplicaciones desarrolladas con Colimate, como de programador del propio lenguaje.

El proyecto fue concebido para funcionar bajo una licencia libre académica, es decir, de libre distribución siempre que no se emplee su contenido para fines comerciales. El lenguaje completo junto con toda la documentación asociada es accesible Internet.

Bibliografía

[1] Alfred V. Aho, Revi Sethi, Jeffrey D. Ullman. *Compilers: principles, techniques and tools*. Addison Wesley, 1988.

[2] Tony Mason, Doug Brown, *Lex & Yacc*. O'Reilly & Associates, Inc. 1991

[3] Herbert Schildt. *C++, the complete reference*. McGraw−Hill, 1995.

[4] David R. Musser. *STL Tutorial and reference guide*. Addison Wesley, 1996

[5] Matthias Kalle Dalheimer. *Programming with Qt*. O'Reilly, 1999.

[6] Ayuda on−line de STL. http://www.sgi.com/Technology/STL

[7] Ayuda on−line de Qt. http://www.troll.no/qt/

[8] Ayuda on−line de Doc++. http://www.zib.de/Visual/software/doc++

[9] Ayuda on−line de Flex. http://www.gnu.org/manual/flex−2.5.4/flex.html

[10] Ayuda on−line de Bison. http://www.gnu.org/manual/bison−1.25/bison.html

[11] Ayuda on−line de Xmipp. http://www.cnb.uam.es/~bioinfo

[12] Ayuda on−line de Colimate. http://www.cnb.uam.es/~bioinfo/Colimate/Extra_Docs

áéíóú