

Proyecto final de carrera:
Entorno gráfico para la ejecución y depuración
de código intermedio

Autor: Alberto Domínguez
Dirigido por: D. Emilio Vivancos

Valencia, Septiembre de 2010

Resumen

El proyecto de final de carrera que se desarrolla a continuación consiste en un intérprete integrado en un entorno de desarrollo que de un modo enteramente visual ejecuta y facilita la depuración de programas escritos en un código intermedio de tres direcciones, potente y polivalente. A su vez detecta todos los errores que pueda tener el programa tanto en tiempo de carga como en tiempo de ejecución. Para ello se pueden usar herramientas como los puntos de ruptura, la inspección continua de los valores en memoria, la ejecución paso a paso o la entrada y salida de datos por consola. El entorno de desarrollo permite además una integración total con el compilador y permite la modificación del código durante su uso, mejorando claramente las herramientas disponibles hasta ahora para ello.

Se ha hecho especial incapié en conseguir que sea un programa de fácil uso y rápido aprendizaje, y que sea muy minucioso a la hora de detectar y mostrar los errores, pues el principal uso del programa es el docente o didáctico aunque no por ello el programa deja de tener toda la potencia necesaria para usarse con un propósito general, soportando varios tipos de datos y permitiendo una interacción cómoda con el usuario mientras se ejecutan los programas. Además, pensando en posibles cambios y ampliaciones del conjunto de instrucciones de código intermedio se ha programado y documentado pensando en la fácil ampliación del mismo por cualquier interesado.

Índice general

1. Introducción	5
1.1. Breve descripción y motivación del proyecto	5
1.2. Conceptos generales	6
1.2.1. Traductores: Compiladores e intérpretes	6
1.2.1.1. Traductores	6
1.2.1.2. Compiladores	7
1.2.1.3. Ventajas e inconvenientes entre compiladores e intérpretes	8
1.2.2. Lenguajes y gramáticas	9
1.2.2.1. Lenguajes	9
1.2.2.2. Gramáticas	9
1.2.3. El proceso de análisis del programa fuente	10
1.2.3.1. Análisis léxico	10
1.2.3.2. Análisis sintáctico	11
1.2.3.3. Análisis semántico	11
1.3. Funcionamiento teórico del intérprete	13
2. Análisis de diseño del intérprete gráfico	16
2.1. Análisis de requisitos	16
2.1.1. Requisitos funcionales	16
2.1.2. Requisitos de usabilidad	19
2.1.3. Requisitos de rendimiento	22
2.1.4. Requisitos de mantenimiento y soporte	22
2.2. Análisis de los casos de uso	23
2.2.1. Abrir programa	23
2.2.2. Recargar programa	25
2.2.3. Reiniciar estado del intérprete gráfico	25
2.2.4. Borrar todos los breakpoints	25
2.2.5. Borrar un breakpoint	25
2.2.6. Ejecutar paso a paso	26
2.2.7. Ejecutar de forma continua	26
2.2.8. Añadir breakpoint	26
2.2.9. Desplazar el visor de memoria	27
2.2.10. Solicitar ayuda	27

2.3.	Diagrama de estados del intérprete gráfico	27
2.4.	Construcción de un prototipo del intérprete gráfico	27
3.	Codificación del intérprete gráfico	30
3.1.	Una aproximación a Qt	30
3.1.1.	Historia de Qt	30
3.1.2.	Estructura de los programas en Qt/C++	31
3.1.3.	Ventajas e inconvenientes del uso de Qt	35
3.2.	Estructura del proyecto software	35
3.3.	Codificación del analizador léxico, sintáctico y semántico.	36
3.3.1.	Herramientas usadas	36
3.3.2.	Análisis del código intermedio	37
3.3.2.1.	Otros tipos de código intermedio	38
3.3.2.2.	Código intermedio de tres direcciones	38
3.3.3.	Codificación del analizador léxico de los ficheros de entrada	40
3.3.4.	Codificación del analizador sintáctico-semántico de los ficheros de entrada	41
3.3.5.	Comunicación entre el módulo de carga de programas y el núcleo del intérprete	43
3.4.	Tipos de datos en el intérprete gráfico	44
3.4.1.	Codificación de las instrucciones	44
3.4.2.	Codificación de la memoria	46
3.4.3.	Variables de estado del intérprete gráfico	47
3.5.	Instrucciones en la ruta de datos	47
3.5.1.	Proceso de decodificación de una instrucción y búsqueda de operadores	48
3.5.2.	Ejecución de la instrucción y almacenamiento del resultado	49
3.5.3.	Funciones que modifican el flujo de ejecución o los punteros del intérprete y el tratamiento de puntos de ruptura	49
3.6.	Aspectos de la interfaz de usuario	51
3.6.1.	QMainWindow: Ventana principal	51
3.6.2.	Estableciendo el widget central	52
3.6.3.	Usando la clase QTreeWidget	53
3.7.	Modificaciones a las funciones de generación de código	54
3.8.	Manual de ampliaciones	54
4.	Manual de usuario	56
4.1.	Introducción rápida	56
4.2.	Instalación	56
4.3.	Primera vista	57
4.4.	Carga de un programa	59
4.4.1.	Recargar un programa	60
4.4.2.	Posibles errores al cargar un programa	60
4.5.	Cambiar el ancho de los paneles y las columnas de datos	61
4.6.	Ejecutar paso a paso un programa	61
4.6.1.	Detalles en la ejecución de un programa	62

<i>ÍNDICE GENERAL</i>	4
4.6.2. Uso estricto de la memoria	63
4.6.3. Posibles errores en tiempo de ejecución	64
4.7. Ejecutar el programa de forma continua	65
4.8. Añadir y quitar breakpoints	65
4.9. Detener y reiniciar el estado	66
4.10. Desplazar automáticamente la representación de la memoria y las instrucciones	66
4.11. Mostrar la ayuda y la autoría del intérprete gráfico	67
4.12. Peticiones de datos por parte del intérprete gráfico	67
4.13. Ayudas al desarrollo	68
4.14. Persistencia de la configuración y opciones por defecto.	69
A. Instrucciones de código intermedio	71
A.1. Instrucciones aritméticas	71
A.2. Instrucciones que modifican el flujo del programa	74
A.3. Instrucciones que acceden a vectores	77
A.4. Instrucciones de entrada y salida de datos	78
A.5. Instrucciones que modifican la pila, el <i>frame pointer</i> y el estado del programa	79
B. Software necesario en la creación del proyecto	82
B.1. Dependencias de compilación y ejecución	82
B.2. Software utilizado durante la realización del proyecto	82
Bibliografía	84

Capítulo 1

Introducción

1.1. Breve descripción y motivación del proyecto

En la asignatura de procesadores de lenguajes de cuarto curso de ingeniería informática los alumnos como práctica deben realizar un compilador para programas fuente escritos en un lenguaje similar a *C*, llamado *MenosC*. Este lenguaje, que varía año tras año, incluye los tipos básicos de *C* para enteros y reales, estructuras de control tales como los bucles FOR, los bucles WHILE, las condiciones IF, llamadas a funciones y estructuras de datos como los vectores. Sin embargo se descartan conceptos más complicados como las estructuras de datos compuestas o los punteros que complicarían demasiado el compilador. De cualquier forma el lenguaje –un subconjunto de *C*– es suficientemente útil para programar cualquier algoritmo de forma imperativa con relativa sencillez.

Como para generar código objeto para un computador del laboratorio, basado en el juego de instrucciones del Intel i386 es demasiado complicado y lo que se pretende con la práctica es que el alumno se encargue de crear un compilador sin tener la necesidad de conocer los entresijos que implica generar un ejecutable de un sistema *Linux* o las numerosas instrucciones del procesador se decidió que la salida del programa no fuese un fichero ejecutable estándar sino un código intermedio con tres argumentos por instrucción y de sencillo acceso a memoria. Dicho código intermedio, muy similar en concepto al del procesador MIPS usado como ejemplo en las asignaturas de arquitectura de computadores lo denominaremos Código de tres direcciones. A partir de este código podría generarse código objeto, pero esto va más allá del objetivo del presente proyecto.

Para ejecutar nuestro programa *MenosC* compilado en código de tres direcciones se hace necesario un intérprete que vaya ejecutando cada instrucción para comprobar que el compilador realizado por el alumno genera las instrucciones oportunas. Es dicho intérprete el que ha de encargarse de la carga y ejecución de las instrucciones y de la entrada y salida de datos del programa.

Hasta ahora el intérprete facilitado para las prácticas es un programa de línea de comandos, con una funcionalidad muy limitada para el usuario, donde

la ejecución paso a paso es de difícil comprensión y no se avisa de ningún error, dejando a cargo del alumno su búsqueda. Para facilitar la tarea de crear el compilador se decidió crear otro intérprete. Un programa ejecutable con un entorno gráfico basado en ventanas donde además de ejecutar el código se pueda ver continuamente el estado de la memoria del proceso, el flujo de ejecución, añadir puntos de pausa en el programa o informando de los posibles errores semánticos que tenga el código. Se pasa así de una concepción de intérprete a la de un entorno de depuración completo, con elementos de ayuda al desarrollo, que va más allá de su objetivo docente y didáctico, para tener un uso general.

Es la programación de este intérprete gráfico el contenido de este proyecto.

1.2. Conceptos generales

1.2.1. Traductores: Compiladores e intérpretes

Antes de mostrar las diferencias y similitudes entre un compilador y un intérprete hay que dar una definición formal de los datos que procesan, programas escritos en lenguajes de alto nivel.

Para especificar formalmente un lenguaje de alto nivel debemos dar tres niveles de especificaciones para que los programas escritos en dicho lenguaje sean correctos y no provoquen situaciones de ambigüedad. Además pondré ejemplos basados en uno de los lenguajes más difundidos, C^1 .

- Especificación léxica: Contiene las reglas que deben cumplir las palabras que forman el lenguaje. Identificadores, palabras reservadas, operadores... Por ejemplo, en un programa escrito en C la palabra `while` está reservada para indicar un bucle mientras que cualquier otro uso de la palabra está prohibida, las variables no pueden empezar por un dígito, los corchetes se usan para desreferenciar una posición de un vector...
- Especificación sintáctica: Contiene las reglas que construyen el lenguaje a partir de su especificación léxica. Por ejemplo, en C un bucle `while` ha de tener una condición entre paréntesis y a continuación una instrucción o bloque de instrucciones, o la necesidad de cerrar todas las llaves que se abren.
- Especificación semántica: Da el significado y las reglas que transforman la especificación sintáctica en un programa consistente. Como ejemplo, en C no se puede declarar varias veces la misma variable, o un bucle `while` implica que debe ejecutarse el código mientras la condición se cumpla.

1.2.1.1. Traductores

Un traductor es un programa que traduce un código programado en un lenguaje a otro código equivalente escrito en otro. Por lo general el código de

¹Para profundizar más en las especificaciones de C leer el capítulo «Especificación del lenguaje» de [7].

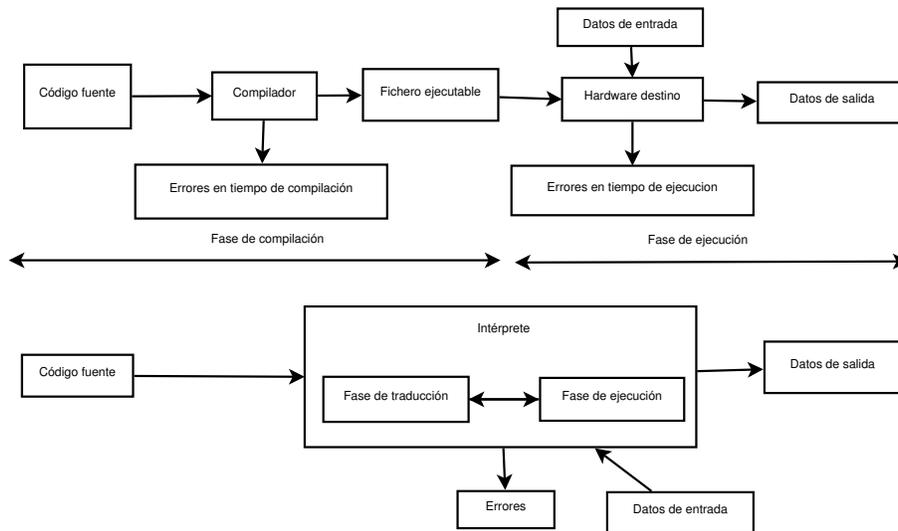


Figura 1.1: Diferencias entre compiladores e intérpretes.

entrada es de alto nivel y el de salida un código de más bajo nivel, aunque podrían haber casos donde fuera útil un traductor a un lenguaje de más alto nivel.

Si el traductor realiza toda la traducción de golpe, dejando un fichero en un código máquina como resultado estamos hablando de un compilador. Si en cambio el traductor traduce una o unas pocas instrucciones, y tras traducirlas las ejecuta y vuelve a por más instrucciones del fichero fuente se llama intérprete [1]. Puede verse la diferencia en la figura 1.1.

1.2.1.2. Compiladores

Definición: Un compilador es un programa informático que traduce ficheros programados en un lenguaje a otro distinto. Normalmente traducen programas escritos en lenguajes de alto nivel a un código objeto que podría ser ejecutado por el procesador de nuestro ordenador [1].

Como hemos dicho antes el compilador es un tipo de traductor que traduce de golpe y en una sola vez² las instrucciones de un programa fuente y crea un fichero ejecutable. Ese proceso puede detectar todo tipo de errores, tanto léxicos, sintácticos o semánticos, indicando al programador su posición en el código.

La cantidad de compiladores existente es enorme. Hay muchos lenguajes fuente (C, C++, Cobol, Pascal) e incluso han aparecido compiladores para lenguajes tradicionalmente interpretados como Basic o Java. El lenguaje objeto que genera un compilador a su vez puede ser distinto. Hay compiladores que generan

²Lo cual no implica que no existan compiladores de múltiples pasadas o fases. Lo importante es que el código máquina de la totalidad del programa esté traducido antes de su ejecución.

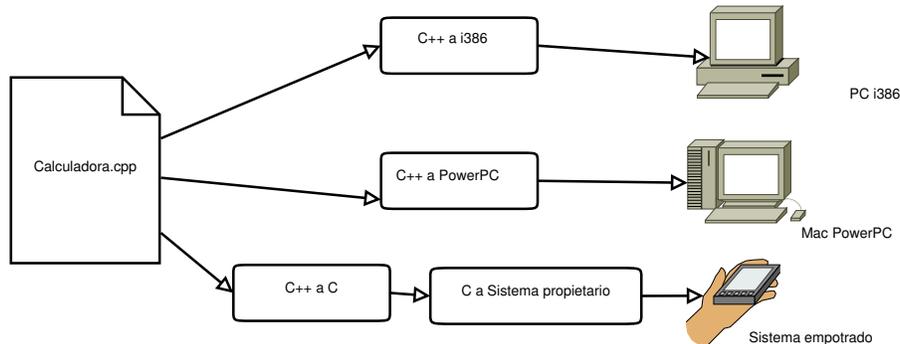


Figura 1.2: Cada lenguaje de programación y arquitectura de destino necesita un compilador distinto.

código para arquitecturas i386, AMD64, MIPS, PowerPC... Y para que un programa se pueda ejecutar en una máquina debe existir el compilador apropiado, tanto por el lenguaje que acepta como por el escribe.

Ejemplo: Tenemos una calculadora escrita en C++, un Mac basado en la arquitectura powerPC, un PC con procesador i386 y un sistema empotrado con un sistema propietario donde el fabricante solo nos facilita un compilador de C. Necesitamos tres compiladores distintos y un traductor. La imagen de la figura 1.2 nos ayuda a comprender la situación.

Para evitar toda esta multiplicidad de esfuerzos han aparecido compiladores divididos en bloques, donde la parte que acepta un lenguaje es independiente del lenguaje de salida.

1.2.1.3. Ventajas e inconvenientes entre compiladores e intérpretes

- El compilador no es necesario cuando se ejecuta el programa, dejando toda la potencia de la máquina al proceso.
- El código ejecutable compilado esta libre de errores léxicos y sintácticos. Pueden aparecer, eso sí, errores semánticos en tiempo de ejecución.
- Es posible optimizar todo lo posible el código compilado, ya que el tiempo de compilación no es crítico.
- Un programa interpretado puede ser multiplataforma. Basta con usar el intérprete apropiado para tu computador y se podrá ejecutar cualquier programa escrito en ese lenguaje.
- Un programa interpretado puede ser ejecutado incluso con errores, al menos hasta que el flujo de ejecución llegue al error.
- Un programa interpretado es más fácil de depurar.

1.2.2. Lenguajes y gramáticas

1.2.2.1. Lenguajes

Definición: Un lenguaje es el conjunto finito o infinito de frases, todas ellas de longitud finita y construidas por concatenación de un número finito de elementos, de modo que $L \subseteq \Sigma^*$. [5]

Desde un punto de vista informático un lenguaje es una notación formal para escribir algoritmos que serán ejecutados por un computador. El problema reside en que los lenguajes de programación usados por programadores suelen ser demasiado complicados para poder ser ejecutados en un computador y necesitaremos un traductor para pasar el código escrito en un lenguaje de *alto nivel* en un lenguaje que pueda ser ejecutado.

Tecnológicamente sería posible contruir computadores que ejecutasen directamente lenguajes de alto nivel, pero sería mucho más lento y más costoso que seguir el enfoque aplicado en la actualidad: procesadores que ejecutan operaciones muy simples pero rápidamente. Por lo tanto, mientras que los lenguajes de *alto nivel*³ son cada vez más potentes, fáciles de aprender y más cercanos al lenguaje humano, los lenguajes de *bajo nivel* continúan siendo igual de simples. Todo esa diferencia de complejidad debe resolverla el traductor.

1.2.2.2. Gramaticas

Llegado a este punto tenemos un problema. Si un lenguaje está formado por un conjunto finito o infinito de frases, ¿Cómo se puede demostrar que una frase cualquiera pertenece o no a un lenguaje?

Pensemos en los distintos programas escritos en C que pueden existir. Es un conjunto infinito de programas. ¿Qué mecanismo tiene un compilador para discernir si el programa de entrada está escrito correctamente -forma parte del lenguaje- o no? Para solucionar el problema debemos buscar un sistema que mediante reglas pueda generar y reconocer frases de cualquier tipo de lenguaje. Ese concepto se conoce como gramática.

El término gramática tiene su origen en los lingüistas que estudiaban los lenguajes naturales. Las gramáticas para lenguajes naturales no se acaban de concretar ya que el lenguaje natural tiene multitud de excepciones, justo al contrario que las gramáticas en las que se basan los lenguajes de programación.

Definición: Se define como $G = \{N, \Sigma, P, S\}$ siendo N un alfabeto de símbolos no terminales, Σ un alfabeto de símbolos terminales donde $N \cap \Sigma = \emptyset$ (en el caso de un lenguaje de programación Σ consistiría en el conjunto de tokens detectados por el analizador léxico), P el conjunto finito de

³Aunque la frontera entre los lenguajes de alto y bajo nivel es difusa, se define un lenguaje de alto nivel si tiene los elementos de abstracción suficientes para solucionar problemas de modo que no se tenga que pensar en los detalles internos de la máquina. Aun así la división es gradual. Por ejemplo, C y Java son lenguajes de alto nivel, pero los programas en C son mucho más cercanos a la máquina. [7]

producciones de forma $(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow N (\Sigma \cup N)^*$ y S un símbolo inicial donde $S \in N$. [5]

1.2.3. El proceso de análisis del programa fuente

1.2.3.1. Análisis léxico

En el análisis léxico se extraen de la cadena de entrada los símbolos (también conocidos como tokens) que componen el lenguaje. Los tokens pueden ser identificadores, palabras reservadas, símbolos matemáticos... Estas unidades de información son básicas del lenguaje y no importa el orden en el que estén escritos de cara a la fase del análisis léxico, ya que el encargado de discernir si los tokens detectados están en un orden aceptado por el lenguaje será el analizador sintáctico.

Para ello se emplean autómatas finitos definidos con la expresión $A = (Q, \Sigma, \delta, q_0, F)$, donde el autómata finito está definido por el conjunto de estados, símbolos de entrada, funciones de transición, y estados finales de aceptación. [5]

Los espacios en blanco, tabuladores, comentarios y símbolos supérfluos usados por el programador para facilitar la lectura del código fuente a otras personas son desechados.

En el análisis léxico se emplean gramáticas regulares –que definen los autómatas finitos antes descritos– para discernir cual es el token adecuado para los caracteres de entrada.

A continuación mostramos un ejemplo con una línea de lenguaje *MenosC*, los tokens encontrados y las expresiones regulares que los identifican.

```
if (3.14 <= limite) a = a + 1;
```

1. *Token if*. Palabra reservada del lenguaje. Detectada por la expresión “if”
2. *Token Abrir paréntesis*. Símbolo reservado del lenguaje. Detectado por la expresión “(“
3. *Token Constante real*. Detectado por la expresión “-”?[0-9]+.”[0-9]*
4. *Token Menor o igual*. Símbolo reservado del lenguaje. Detectado por la expresión “<=”. No es confundido por la concatenación del token Menor que seguido del token Igual, dado que el análisis léxico acepta por definición la expresión regular más larga.
5. *Token Identificador*. Detectada por la expresión $[a-zA-Z]+\{a-zA-Z0-9\}^*$
6. *Token Cerrar paréntesis*. Símbolo reservado del lenguaje. Detectado por la expresión “)”
7. *Token Identificador*.
8. *Token Asignación*. Símbolo reservado del lenguaje. Detectado por la expresión “=”

9. *Token Identificador.*
10. *Token Signo positivo.* Símbolo reservado del lenguaje. Detectado por la expresión “+”
11. *Token Constante entera.* Detectado por la expresión “-”?[0-9]+
12. *Token Punto y coma.* Detectado por la expresión “;”

1.2.3.2. Análisis sintáctico

El análisis sintáctico es el proceso que determina si la cadena de tokens detectados por el analizador léxico se ajusta a la gramática que define el lenguaje de entrada de nuestro traductor. Para ello trataremos de ajustar los tokens en grupos gramaticales que se pueden visualizar en formato de árbol. Si los tokens no se ajustan a ninguna regla gramatical definida es porque la cadena no pertenece al lenguaje de entrada y nos encontraremos con un error.

Es decir, el analizador sintáctico debe comprobar que:

$$L(G) = \left\{ w \in T^* \mid S \xrightarrow[G]{*} w \right\}$$

Significando que el lenguaje definido por la gramática G son todas las palabras que pertenecen al conjunto de palabras que se pueden formar con el alfabeto T y surgen al derivar el símbolo inicial de la gramática hasta formarla. [5]

Los tokens encontrados en el ejemplo anterior se pueden organizar en el siguiente árbol sintáctico, pudiendo derivarse usando las reglas de la gramática del lenguaje MenosC. Mostramos el árbol formado por el ejemplo de la sección 1.2.3.1 en la figura 1.3.

1.2.3.3. Análisis semántico

Ahora que sabemos que la cadena de entrada pertenece léxica y sintácticamente al lenguaje, debemos comprobar si tiene sentido. Alguna de las comprobaciones que realiza un traductor de C puede ser [7]:

- Comprobación de tipos: Debemos asegurarnos que los tipos empleados en las operaciones son compatibles. Por ejemplo, no podemos sumar un entero con un real.
- Comprobación de unicidad: Una variable o función solo puede ser definida una vez.
- Comprobación de flujo: Habría un error si por ejemplo usamos un break desde fuera de un bucle.

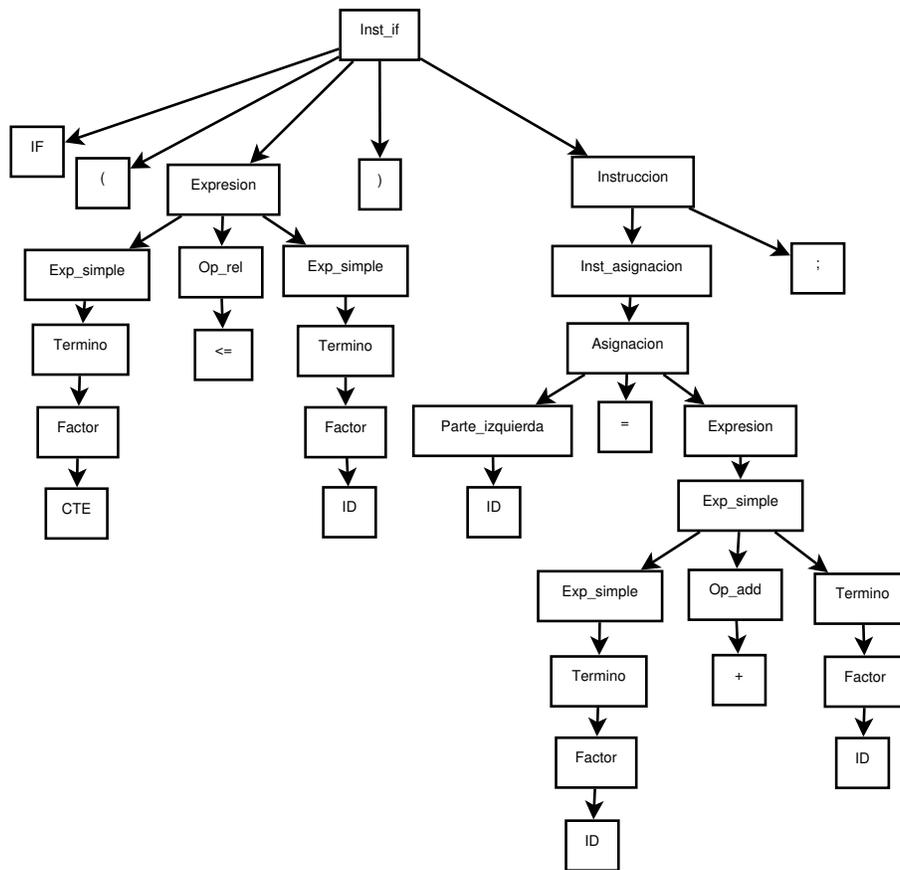


Figura 1.3: Árbol sintáctico formado por la instrucción de la sección 1.2.3.1

1.3. Funcionamiento teórico del intérprete

Nuestro intérprete ejecuta las instrucciones de código intermedio como una máquina de arquitectura Harvard, es decir, con la memoria separada para las instrucciones y para los datos. [4] La memoria se representa como un vector de mil posiciones donde en cada una se puede guardar tanto un entero como un número en punto flotante⁴, mientras que hay espacio disponible para programas de hasta quinientas instrucciones. La asignación de memoria se realiza por medio de una pila de control. Los registros de activación se introducen y se sacan cuando las activaciones comienzan y terminan respectivamente. La memoria para las variables locales en cada llamada de un procedimiento está contenida en el registro de activación de dicha llamada. Además, los valores de las variables locales se borran –quedan inaccesibles– cuando finaliza la activación. [1]

Para marcar el primer registro libre de la memoria emplearemos el puntero a cima (*top pointer*). Durante la ejecución un registro de activación se puede asignar y desasignar incrementando y decrementando el puntero el mismo número de posiciones que ocupe el registro de activación. El puntero a marco (*frame pointer*) marca siempre la primera posición de memoria del registro de activación actual. Gracias a dicho puntero podemos acceder a variables globales, a los que se accede en las primeras posiciones de memoria, o a variables locales, para los cual tendremos que acceder a la memoria sumando la dirección efectiva a la posición del puntero a marco.

Un registro de activación consta de los siguientes elementos:

- Valor de retorno
- Parámetros de la función actual
- Copia de la dirección de retorno
- Enlace de control (copia del antiguo *frame pointer*)
- Variables locales
- Variables temporales

Para profundizar mejor en las estructuras de control ejecutaremos un programa muy simple que solo calcula el factorial de un entero menor de 15. El código en MenosC es el siguiente:

```
int fact(x){
    if(x > 1) return x * fact(x - 1);
    else return 1;
}
int main(){
    x = read();
```

⁴Aunque en las arquitecturas más habituales un número en coma flotante ocupa más que un entero, nuestro intérprete supone que ocupan el mismo tamaño de bytes por razones de simplicidad.

```

    if(x > 15){
        print(0);
        return 0;
    }
    print(fact(x));
    return 0;
}

```

Si queremos calcular el factorial de 3 se deben producir 4 registros de activación. Primero se cargará el registro de activación para la función principal, y luego consecutivamente para `fact(3)`, `fact(2)` y `fact(1)`.

Registros de activación	Datos	Valores
R. Activación <code>main()</code>	Dato retorno	0
	Variable local	x
	Variables temporales	...
R. Activación <code>fact(3)</code>	Dato retorno	6
	Parametro	3
	Dirección retorno	Apilado por la instrucción CALL del <code>main</code>
	Dirección de enlace	Antiguo FP del Registro de activación de <code>main()</code>
	Variables temporales	...
R. Activación <code>fact(2)</code>	Dato retorno	2
	Parametro	2
	Dirección retorno	Apilado por la instrucción CALL de <code>fact(3)</code>
	Dirección enlace	Antiguo FP del registro de activación de <code>fact(3)</code>
	Variables temporales	...
R. Activación <code>fact(1)</code>	Dato retorno	1
	Parametro	1
	Dirección retorno	Apilado por la instrucción CALL de <code>fact(2)</code>
	Dirección enlace	Antiguo FP del registro de activación de <code>fact(2)</code>
	Variables temporales	...

Cuando se esté ejecutando el caso base de la recursión el *frame pointer* apuntará al registro de activación accesible en ese momento, es decir, al creado por la instancia `fact(1)`.

Sin embargo, aunque este método sea muy útil para traducir código de lenguajes no anidados no nos permitirá declarar funciones anidadas. Para ello en vez de tener un solo puntero *frame pointer*, deberíamos tener un conjunto de punteros referenciados como un vector llamados *display*. En ese caso si tenemos un Display de diez posiciones podríamos anidas hasta diez funciones. De todos

modos se puede observar que sólo se puede acceder a las variables contenidas en el registro de activación actual y en el caso de usar *displays*, en cada una de las funciones anidadas superiores a la actual.

Para profundizar en el tema acudir al capítulo “Ambientes para el momento de la ejecución” en [1].

Capítulo 2

Análisis de diseño del intérprete gráfico

Llegados a este punto debemos plantearnos la concepción de nuestro intérprete gráfico: Las librerías de interfaz gráfica, la especificación de requisitos, el formato de los ficheros de entrada, el tratamiento de errores...

2.1. Análisis de requisitos

A continuación se muestran un conjunto de requisitos imprescindibles que debería implementar el programa que se ha desarrollado. Para cada uno se especifica el grado de importancia de cara al usuario si el requisito se cumple en una escala de cero a cinco. Además se explica el motivo o la justificación de la existencia del requisito y por último se valida el grado de cumplimiento en la aplicación final. Para ello se seguirá el esquema recomendado por la norma IEEE Std. 830-1998. [6]

Los requisitos se dividen en:

- Funcionales
- De usabilidad
- De rendimiento
- De mantenimiento y soporte

2.1.1. Requisitos funcionales

Requisito 01

Descripción: Se ha de poder cargar un fichero fuente en código de tres direcciones en texto plano.

Importancia: 5

Justificación: Hasta ahora, en el intérprete proporcionado a los alumnos se debía introducir un programa guardado en forma binaria. Debíamos compilar las fuentes dos veces, una vez con salida en modo texto para poder leer el código y la otra en modo binario para ejecutarlo. El fichero que podíamos ejecutar por tanto no podía ser modificado.

Validación: Hecho. Nuestro intérprete se carga con ficheros en modo texto. En la sección 4.4 se explica cómo.

Requisito 02

Descripción: Se ha de poder ejecutar el programa paso a paso.

Importancia: 5

Justificación: El usuario debe poder ver como afecta cada instrucción al contenido de la memoria, los saltos en el flujo del programa y los posibles mensajes de error, sabiendo en cada momento por qué se han producido.

Validación: Hecho. La sección 4.6 muestra toda su utilidad.

Requisito 03

Descripción: Se ha de poder ejecutar el programa de forma continua.

Importancia: 4

Justificación: El usuario debe poder ejecutar de forma continua programas que por su longitud o elevada recursividad no sea factible ejecutarlos paso a paso por el tiempo que se tardaría.

Validación: Hecho. Puede ver la forma de uso en la sección 4.7.

Requisito 04

Descripción: Se ha de poder manejar distintos tipos de datos.

Importancia: 5

Justificación: Aunque los usuarios del intérprete gráfico actualmente solo usen datos de tipo entero, debe estar preparada para ejecutar programas que contengan instrucciones con operadores reales y estar preparada para que con ligeras modificaciones de código, admitir cualquier otro tipo de dato (por ejemplo, punteros).

Validación: Hecho.

Requisito 05

Descripción: El programa no debe fallar al pedir datos al usuario.

Importancia: 3

Justificación: Cuando el usuario cargue un programa y al ejecutarlo se llegue a una instrucción que pida un dato al usuario (ya sea entero, real o carácter) el intérprete debe validar que el dato de entrada corresponde al tipo solicitado.

Validación: Hecho. Para más información acudir a la sección 4.12.

Requisito 06

Descripción: Se ha de poder parar el programa en cualquier momento.

Importancia: 4

Justificación: Cuando el usuario vea que su programa tiene errores o no desee acabar su ejecución debe poder pararlo, reiniciando de este modo el estado del intérprete.

Validación: Parcialmente. El intérprete no puede reiniciarse si entra en un bucle infinito en el modo de ejecución continua.

Requisito 07

Descripción: Se deben poder establecer breakpoints.

Importancia: 5

Justificación: El usuario debe poder establecer breakpoints en cualquier instrucción. Así, en la ejecución continua del programa el intérprete gráfico se detendrá antes de decodificar la instrucción.

Validación: Hecho. Para informarse sobre sus usos o activación ver la sección 4.8 o la figura de la página 50.

Requisito 08

Descripción: El intérprete gráfico debe detectar cualquier fallo que contenga el programa de entrada en tiempo de carga y mostrarlo claramente.

Importancia: 5

Justificación: El usuario debe poder ver todos los errores que tenga su programa en tiempo de carga. De esta forma no perderá el tiempo ejecutando programas en los que se sepa que contienen errores de antemano, además de una inestimable ayuda al programador del compilador. Sin embargo es una característica que debe poder desactivarse por parte del usuario para hacer pruebas en programas parcialmente correctos.

Validación: Hecho. En la sección 4.4.2 se detallan y ejemplifican los errores en tiempo de carga que se detectan.

Requisito 09

Descripción: El intérprete gráfico debe detectar cualquier fallo o situación sospechosa que se produzca durante la ejecución de un programa y avisar.

Importancia: 5

Justificación: El usuario debe poder ver todos los errores que su programa cometa en tiempo de ejecución, ya que por lo general suelen ser errores sutiles o de concepto (uso de la pila de memoria, uso del *frame pointer*, etc.

Validación: Hecho. En la sección 4.6.3 se encuentra una lista con todos los errores detectables.

2.1.2. Requisitos de usabilidad

Requisito 10

Descripción: Se ha de poder recargar un fichero.

Importancia: 3

Justificación: El usuario puede volver a cargar el fichero que ya está ejecutándose. Esto es útil cuando se ha compilado una nueva versión del mismo.

Validación: Hecho. Puede ver su forma de uso en la sección 4.4.1.

Requisito 11

Descripción: No se debe visualizar el contenido de la memoria a partir del puntero a cima.

Importancia: 2

Justificación: Cualquier dato que esté por encima del puntero de pila no debería ser válido y por tanto debe ser ignorado en la memoria y ocultada su representación de cara al usuario. Se puede confiar en que el usuario sepa que los datos por encima del puntero a cima son residuales, advirtiéndolo al usuario que intente escribir o leer posiciones de memoria superiores al puntero a cima. Debe poder desactivarse por parte del usuario para poder depurar programas que fallen escribiendo o leyendo por encima del vector cima.

Validación: Hecho. Para conocer las implicaciones que resultan de activar esta opción leer la sección 4.6.2.

Requisito 12

Descripción: Se ha de introducir algún mecanismo de comentarios.

Importancia: 3

Justificación: Los programas escritos en código de tres direcciones suelen ser difíciles de comprender en una primera lectura. Se debe dejar al usuario la libertad para que su compilador emita comentarios que luego muestre el intérprete gráfico. Es de gran ayuda para comprender cual bloque de instrucciones del código MenosC está sustituyendo las instrucciones mostradas en el intérprete.

Validación: Parcialmente. En un fichero de código de tres dimensiones se aceptan dos tipos de comentario, basados en texto y en el uso de la instrucción NOP con un argumento entero identificativo. Sin embargo en la representación del programa en el intérprete sólo se mostrarán las instrucciones NOP.

Requisito 13

Descripción: El intérprete gráfico ha de ser fácil de instalar

Importancia: 2

Justificación: El intérprete depende de varias librerías, principalmente de Qt 4. Además para compilarlo se necesitan algunos programas adicionales al compilador de C++. Es por ello que es recomendable distribuirlo bien sea en un fichero .deb o con un makefile automático.

Validación: Hecho. Se ha conseguido que tan solo ejecutando el fichero de proceso por lotes *do* se compile todo el código dependiente del intérprete. Hay que tener en cuenta que se debe tener instalado en el sistema las librerías de desarrollo de Qt, Flex, Bison, GCC y G++ (Ver la sección B.1).

Requisito 14

Descripción: Se debe distribuir el intérprete gráfico en diversos idiomas.

Importancia: 1

Justificación: El intérprete va a ser fundamentalmente usado por alumnos a los que se les ofrece la asignatura en castellano, valenciano e inglés. El intérprete debería estar traducida al menos a esos idiomas.

Validación: Parcial. A ampliar en futuras versiones. El intérprete está en castellano, pero todo el texto que se muestra al usuario está marcado para que pueda traducirse sin esfuerzo usando la herramienta Qt Linguist. En cualquier caso la abundancia de iconos sencillos de entender no hace excesivamente importante la traducción.

Requisito 15

Descripción: Se debe poder llamar al compilador y al editor desde el mismo intérprete.

Importancia: 3

Justificación: En un entorno de desarrollo normalmente se recompila el código cada vez que se corrige algún error. Es por ello que la introducción de un orden que compilase el código y abriese un editor sería una buena ayuda.

Validación: Hecho. Además para ello se ha añadido un menú de configuración indicando cual es la ruta del compilador o editor que queremos usar (ver sección 4.13).

Requisito 16

Descripción: El intérprete debe guardar sus opciones de configuración cada vez que se cierra para recordarlas la próxima vez que se ejecute.

Importancia: 3

Justificación: Naturalmente no se debería tener que configurar las opciones del programa cada vez que se ejecuta, y el tamaño y posición de las ventanas y paneles debería mantenerse entre sesiones.

Validación: Hecho. Haciendo uso de métodos que aporta la librería Qt es posible guardar la configuración e iniciar el programa usando una configuración estándar la primera vez que se ejecuta en un equipo (ver sección 4.14).

2.1.3. Requisitos de rendimiento

Requisito 17

Descripción: El intérprete gráfico debe ejecutar el código en modo continuo de forma rápida.

Importancia: 3

Justificación: Aunque el intérprete va a ser utilizado principalmente para fines didácticos, es necesario que la ejecución de código de forma continua sea lo más rápida posible para evitar largas esperas en algoritmos poco optimizados.

Validación: Hecho. El intérprete mientras está ejecutando código en modo continuo no modifica ningún objeto de la interfaz de usuario. Los cambios se aplican cuando la ejecución finaliza, ya sea por llegar al fin del código, un error de ejecución o a un punto de ruptura.

2.1.4. Requisitos de mantenimiento y soporte

Requisito 18

Descripción: El código del intérprete gráfico debe ser modular, fácil de comprender y de ampliar.

Importancia: 5

Justificación: El intérprete cumple un propósito eminentemente didáctico. Aunque el conjunto de instrucciones de código de tres direcciones es suficientemente amplio para cualquier tarea, es posible que en un futuro se necesiten incluir más o cambiar su semántica.

Validación: Hecho. El intérprete gráfico centraliza el código referente a la implantación de las instrucciones y se detalla en su manual como codificar nuevas instrucciones, además el código está profusamente comentado.

Requisito 19

Descripción: El intérprete gráfico debe poder compilarse de forma sencilla.

Importancia: 4

Justificación: Aunque el código fuente del intérprete no es muy extenso, para compilarse necesita la presencia de varios programas y librerías auxiliares, ejecutándose en un orden determinado. No debemos presuponer que el usuario que necesite compilarlo pueda hacerlo.

Validación: Hecho. El manual hace referencia a los pasos y dependencias necesarias para compilar el intérprete. Además se incluye un fichero con las órdenes necesarias que automatizan su compilación.

2.2. Análisis de los casos de uso

Los casos de uso son las secuencias de interacciones que se desarrollarán entre un sistema y sus actores en respuesta a un evento que inicia un actor principal sobre el sistema, en este caso el usuario sobre el intérprete gráfico. Forma parte de la fase de diseño junto a la especificación de requisitos en cualquier desarrollo software [6].

En la figura 2.1 mostramos de forma esquemática los casos de uso del programa.

Cabe recordar que para ver en detalle todas las opciones disponibles en detalle puede consultar el manual de usuario en la sección 4.

2.2.1. Abrir programa

Precondiciones: El intérprete gráfico debe no debe estar en estado de ejecución.

1. El usuario indica al intérprete que desea abrir un fichero.
2. El intérprete gráfico muestra un diálogo de selección de fichero.
3. El usuario elige un fichero, pudiendo seleccionar el filtrado de archivos por extensión.
4. El intérprete gráfico carga el fichero en memoria y *LLAMA* a Reiniciar estado del intérprete gráfico.

Extensiones:

1. Si en 4) el intérprete detecta un error en el fichero de entrada:
 - a) Mostrar fichero de error
-

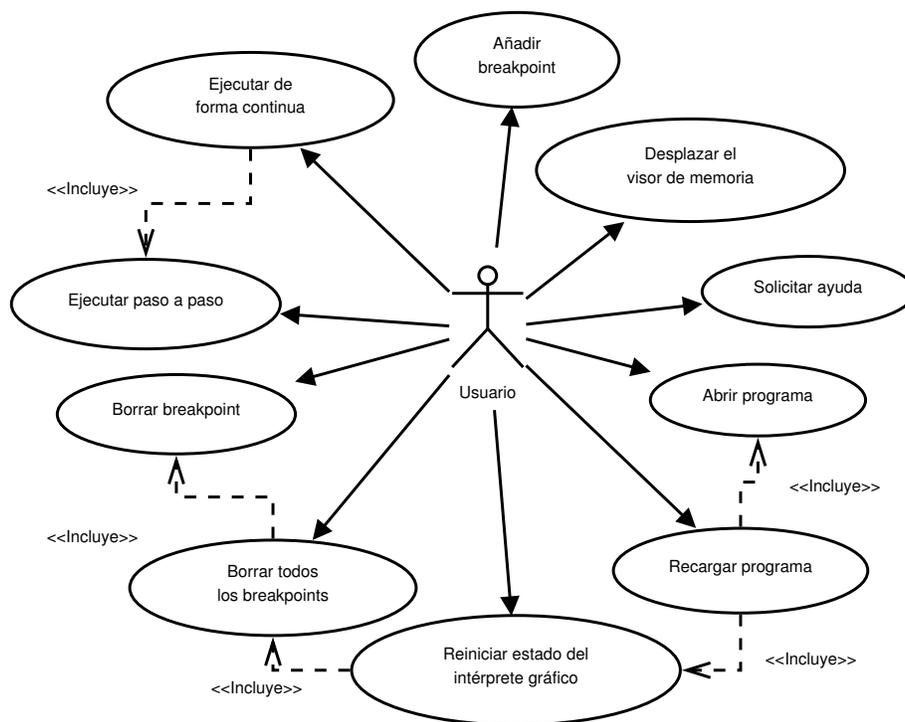


Figura 2.1: Casos de uso del intérprete gráfico.

2.2.2. Recargar programa

Precondiciones: El intérprete gráfico no debe estar en estado de ejecución y debe tener un fichero cargado.

1. El usuario indica al intérprete gráfico que se desea recargar el fichero cargado.
 2. El intérprete gráfico *LLAMA* a Abrir programa pasándole como parámetro el fichero actual.
-

2.2.3. Reiniciar estado del intérprete gráfico

Precondiciones: El intérprete gráfico no debe estar en estado de ejecución y debe tener un fichero cargado.

1. El usuario indica al intérprete gráfico que desea reiniciar su estado.
 2. El intérprete gráfico borra el contenido de la memoria y pone a cero el puntero a cima, el puntero a pila y el contador de programa. A continuación *LLAMA* a Borrar todos los breakpoints.
-

2.2.4. Borrar todos los breakpoints

Precondiciones: El intérprete gráfico no debe estar en estado de ejecución, debe tener un fichero cargado y algún breakpoint.

1. El usuario indica al intérprete gráfico que quiere borrar todos los breakpoints del programa.
 2. El intérprete gráfico *LLAMA* a Borrar breakpoint por cada breakpoint que tenga el programa.
 3. El intérprete gráfico muestra en la barra de estado la cantidad de breakpoints que se han quitado.
-

2.2.5. Borrar un breakpoint

Precondiciones: El intérprete gráfico no debe estar en estado de ejecución, debe tener un fichero cargado y algún breakpoint.

1. El usuario indica al intérprete gráfico que quiere borrar un breakpoint en concreto.
 2. El intérprete gráfico borra el breakpoint, indicándolo en la barra de estado.
-

2.2.6. Ejecutar paso a paso

Precondiciones: El intérprete gráfico no debe estar en estado de ejecución y debe tener un fichero cargado.

1. El usuario indica al intérprete gráfico que quiere ejecutar la siguiente instrucción del programa.
2. El intérprete gráfico decodifica la instrucción, la ejecuta, modifica la memoria y modifica el contador de programa.

Extensiones:

1. Si en 2) el intérprete gráfico detecta alguna condición que implique un warning o excepción:
 - a) *Si* la opción de Inhibir excepciones no está activada el intérprete gráfico mostrará un mensaje de error.
 - b) *Si* el contador de programa sale del límite dejará el intérprete gráfico en estado de error.
 - c) El usuario *solicitará* Reiniciar el estado del intérprete gráfico para continuar.
-

2.2.7. Ejecutar de forma continua

Precondiciones: El intérprete gráfico no debe estar en estado de ejecución, debe tener un fichero cargado y algún breakpoint.

1. El usuario indica al intérprete gráfico que quiere ejecutar el programa de forma continua.
 2. El intérprete gráfico *LLAMA* a Ejecutar paso a paso *hasta* finalizar la ejecución o llegar a un breakpoint.
-

2.2.8. Añadir breakpoint

Precondiciones: El intérprete gráfico no debe estar en estado de ejecución, debe tener un fichero cargado y alguna instrucción libre de breakpoints.

1. El usuario indica al intérprete gráfico que quiere añadir un breakpoint en una línea en concreto.
 2. El intérprete gráfico añade el breakpoint y muestra en la barra de estado el número de línea.
-

2.2.9. Desplazar el visor de memoria

Precondiciones: El intérprete gráfico no debe estar en estado de ejecución.

1. El usuario indica al intérprete gráfico que quiere desplazar el visor de la memoria e instrucciones para visualizar en cada momento los datos pertinentes.
 2. El intérprete gráfico indica en la barra de estado que ha activado la opción.
-

2.2.10. Solicitar ayuda

Precondiciones: El intérprete gráfico no debe estar en estado de ejecución.

1. El usuario indica al intérprete gráfico que necesita ayuda.
 2. El intérprete gráfico muestra un diálogo con el significado de las instrucciones.
-

2.3. Diagrama de estados del intérprete gráfico

El programa una vez empieza su ejecución se encuentra en cada momento en un estado determinado. Dependiendo del estado en que se encuentre, tendrá habilitados o deshabilitados unas opciones de menú, unos botones o unos eventos.

En la figura 2.2 se muestran los estados que puede alcanzar el intérprete gráfico en su funcionamiento y las transacciones que se dan para pasar de un estado a otro.

2.4. Construcción de un prototipo del intérprete gráfico

Tras establecer claramente las funcionalidades ofrecidas por el intérprete gráfico y las interacciones de ésta con el usuario, el siguiente paso antes de entrar en la fase de codificación es crear un prototipo adecuado en la que se ofrezca un esbozo de la apariencia y funcionalidad de la aplicación finalizada para mostrarla al futuro usuario. [6] Nuestro prototipo se ajusta a las siguientes definiciones:

Prototipo de baja fidelidad: Este prototipo se caracteriza por ser una maqueta estática no computerizada y no operativa de la interfaz de usuario.

Prototipo exploratorio: Prototipo no reutilizable dedicado a definir las metas del proyecto, identificar requerimientos y depurar la interfaz de usuario.

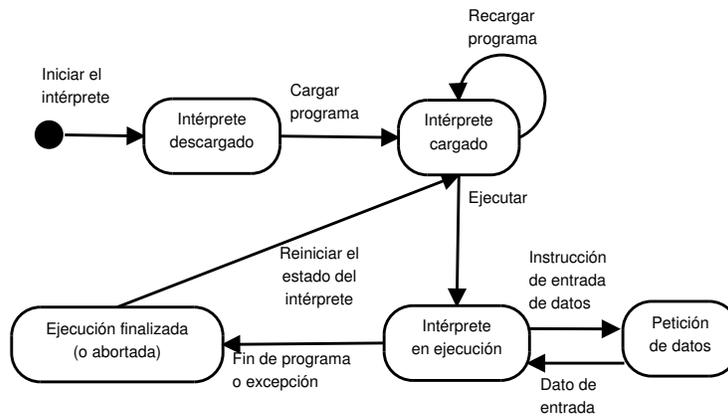


Figura 2.2: Diagrama con los estados del intérprete gráfico.

Prototipo horizontal: Muestra las características de un sistema sin entrar en detalles.

La figura 2.3 muestra la ventana principal de mi prototipo. Como podrá observarse más adelante, la interfaz de usuario sufrirá algunos cambios hasta su implementación final.

El uso de del prototipo ha sido muy útil, ya que me ha servido para empezar a programar usando la librería gráfica Qt4, dandome muestras de hasta dónde puede llegar una interfaz que haga uso de ella. También ha servido para mostrar a posibles usuarios del intérprete el aspecto general de la aplicación y que ellos comentasen cuales son las opciones que necesitan o echarían más en falta.



Figura 2.3: Prototipo del intérprete gráfico.

Capítulo 3

Codificación del intérprete gráfico

3.1. Una aproximación a Qt

Hemos decidido emplear Qt en su cuarta versión para construir la interfaz gráfica de usuario del intérprete. Su sencillez en su uso, su potencia y su rápida curva de aprendizaje han sido factores clave en la decisión. Aunque la librería está algo menos difundida que la ubicua GTK, forma parte de cualquier sistema con el escritorio KDE y es en cualquier caso muy fácil de instalar en cualquier distribución.

3.1.1. Historia de Qt

Qt es una librería multiplataforma para desarrollar interfaces gráficas de usuario aunque también es útil para el desarrollo de programas sin interfaz gráfica como herramientas de consola y servidores. Qt es muy conocida al ser usada en el entorno de ventanas KDE, además de otras muchas aplicaciones como KOffice, K3B, Opera o VideoLAN, entre otros. Es producido por la división de software Qt de Nokia, que entró en vigor después de la adquisición por parte de Nokia de la empresa noruega Trolltech, el productor original de Qt, el 17 de junio de 2008. [3]

Inicialmente Qt apareció como biblioteca desarrollada por Trolltech (en aquel momento «Quasar Technologies») en 1992 siguiendo un desarrollo basado en el código abierto, pero no completamente libre. Originalmente permitía desarrollo de software cerrado mediante la compra de una licencia comercial, o el desarrollo de software libre usando la licencia Free Qt. Esta última no era una licencia real de software libre dado que no permitía redistribuir versiones modificadas de Qt.

Se usó activamente en el desarrollo del escritorio KDE con un notable éxito y rápida expansión, camino de convertirse en uno de los escritorios más populares de GNU/Linux. Este hecho causaba preocupación desde el proyecto GNU,

ya que veían como una amenaza para el software libre que uno de los escritorios libres más usados se apoyase en software propietario. Para contrarrestar esta situación se plantearon dos iniciativas: por un lado el equipo de GNU en 1997 inició el desarrollo del entorno de escritorio GNOME con GTK+ para GNU/Linux. Por otro lado se intentó hacer una biblioteca compatible con Qt pero totalmente libre, llamada Harmony.

En 1998 desarrolladores de KDE se reunieron con Trolltech para establecer la Free Qt Foundation, que establecía que si Trolltech dejaba de desarrollar la versión gratuita y semi-libre de Qt la propia Fundación podría liberar la última versión publicada de la biblioteca Qt bajo una licencia tipo BSD.

En el año 2000 Trolltech comenzó a ofrecer la biblioteca Qt 2.1 bajo la licencia GPL en su versión para Linux. La versión para MacOS X no se publicó bajo GPL hasta 2003, mientras que la versión para Windows fue publicada bajo la licencia GPL en junio de 2005.

Qt cuenta actualmente con un sistema de triple licencia: GPL para el desarrollo de software de código abierto y software libre, la licencia de pago QPL para el desarrollo de aplicaciones comerciales, y a partir de la versión 4.5 una licencia gratuita pensada para aplicaciones comerciales, LGPL.

3.1.2. Estructura de los programas en Qt/C++

Aunque Qt utiliza el lenguaje de programación C++ de forma nativa, adicionalmente puede ser utilizado en varios otros lenguajes de programación a través de bindings, siendo los más populares QTJambi, para Java y PythonQt para Python.

Qt es una librería orientada a clases, habiendo clases para cualquier necesidad que tengamos: creación de diálogos, ventanas, widgets predefinidos y creados por el programador, gráficos 2D y 3D, proceso de eventos, entrada y salida, conexión a bases de datos, conexión a redes, programación concurrente...

Mirando más de cerca las funciones que vamos a emplear, es decir, las que se refieren a la creación de interfaces de usuario, Qt es sencillo de usar. Con Qt usaremos el concepto de acción, elemento que define un trabajo concreto y puede mostrarse tanto en la barra de menús, las cajas de herramientas o en los atajos de teclado.

A una ventana en Qt puede añadirse una barra de estado, diferentes paneles, widgets y otros elementos con las que el usuario puede interactuar. La forma de unir las interacciones del usuario con el código que escriba el programador es a través de la instrucción `connect` y `disconnect`. Estas instrucciones no pertenecen al lenguaje C++, lo que hace que el código deba preprocesarse por la herramienta de precompilación de Qt antes de ser compilado por un compilador de C++ como puede ser el g++.

A continuación se propone un ejemplo de clase Qt comentada para ver sus funcionalidades principales:

```
1 /* Fichero dialogo.h */
   #include <QDialog>
```



Figura 3.1: Diálogo que forma el código comentado en la sección 3.1.2

```

#include <QtGui>
class FindDialog : public QDialog {
5 Q_OBJECT
public:
    FindDialog(QWidget *parent = 0);
signals:
10 void findNext(const QString &str, Qt::CaseSensitivity cs);
    void findPrevious(const QString &str, Qt::CaseSensitivity cs);
private slots:
    void findClicked();
    void enableFindButton(const QString &text);
15 private:
    QLabel *label;
    QLineEdit *lineEdit;
    QCheckBox *caseCheckBox, *backwardCheckBox;
    QPushButton *findButton, *closeButton;
20 };

```

El fichero que mostramos es la definición de la clase que mostrará el cuadro de diálogo en pantalla mostrado en la figura 3.1. En él vemos como se incluyen las definiciones de las clases que nos permiten crear cuadros de diálogos en la línea 2 y las clases que nos permitirán añadir los widgets en la línea 3. Usaremos etiquetas de textos, cajas de entrada de línea, botones y botones de aceptar opciones.

La siguiente línea es la macro `Q_OBJECT`. Es importante añadirla en cualquier clase que necesitamos que tenga las funcionalidades de añadir slots y signals, los mecanismos que usa Qt para comunicar eventos del usuario.

Después definimos la clase `FindDialog`, que hereda de la clase predefinida `QDialog`. Esto nos permitirá acceder a multitud de métodos, entre ellos los que devuelven los valores al código que muestra el diálogo. Tras ello declaramos el constructor de la clase como un método público. Su único argumento es el padre del diálogo. Cuando se llame desde otra ventana deberemos pasarle el puntero `this`. En caso de no hacerlo Qt en tiempo de ejecución adscribirá el diálogo a la ventana principal de la aplicación.

A partir de la línea 9 definimos dos señales encargadas de comunicar a la ventana principal que el usuario quiere realizar una búsqueda hacia delante o hacia atrás y si queremos que la búsqueda diferencie entre mayúsculas o minúsculas.

En la línea 12 definimos slots privados. Los slots son métodos que se pueden activar mediante la interacción del usuario. En este ejemplo el slot `findClicked` lo activaremos cuando el usuario pulse el botón `find`, y el slot `enableFindButton` se activará cuando el usuario introduzca algún carácter en la caja de texto. Los slots los uniremos con las señales adecuadas usando `connect` más adelante.

Finalmente declaramos los seis widgets con los que cuenta el ejemplo.

```

01 FindDialog::FindDialog(QWidget *parent)
   : QDialog(parent){
   label = new QLabel(tr("Find &what:"));
   lineEdit = new QLineEdit;
05   caseCheckBox = new QCheckBox(tr("Match &case"));
   backwardCheckBox = new QCheckBox(tr("Search &backward"));
   findButton = new QPushButton(tr("&Find"));
   findButton->setDefault(true);
   findButton->setEnabled(false);
10   closeButton = new QPushButton(tr("Close"));
   connect(lineEdit, SIGNAL(textChanged(const QString &)),
           this, SLOT(enableFindButton(const QString &)));
   connect(findButton, SIGNAL(clicked()),
           this, SLOT(findClicked()));
15   connect(closeButton, SIGNAL(clicked()),
           this, SLOT(close()));
   QHBoxLayout *topLeftLayout = new QHBoxLayout;
   topLeftLayout->addWidget(label);
   topLeftLayout->addWidget(lineEdit);
20   QVBoxLayout *leftLayout = new QVBoxLayout;
   leftLayout->addLayout(topLeftLayout);
   leftLayout->addWidget(caseCheckBox);
   leftLayout->addWidget(backwardCheckBox);
   QVBoxLayout *rightLayout = new QVBoxLayout;
25   rightLayout->addWidget(findButton);
   rightLayout->addWidget(closeButton);
   QHBoxLayout *mainLayout = new QHBoxLayout;
   mainLayout->addLayout(leftLayout);
   mainLayout->addLayout(rightLayout);
30   setLayout(mainLayout);
   setWindowTitle(tr("Find"));
}

```

El constructor del diálogo hace lo que cabría esperar. Crea la etiqueta de texto que solicita la palabra a buscar, la caja de texto y los botones de opción que indican si se buscará distinguiendo mayúsculas de minúsculas o si se buscará hacia atrás. En la línea 7 se crea el botón `Find` y establece las opciones para que el botón sea el predefinido –se pulsará cuando el usuario aprete la tecla `intro`– y que aparezca desactivado. A continuación se declara el botón que cierra el diálogo.

En la línea 11 se declaran las conexiones, donde tenemos que unir una señal de una clase Qt con el slot de otra. En este ejemplo se hacen tres conexiones:

- La primera une la interacción del usuario con la caja de texto con un método que comprobará que la entrada sea válida y activará el botón `find`.
- La segunda conecta la acción de hacer clic sobre el botón `find` con un método que enviará los datos a la ventana principal.
- La tercera conecta el botón `cerrar` con el slot predefinido `close`, heredado de la clase `QDialog`.

A partir de la línea 17 se establecen diferentes capas para que Qt ordene los widgets de forma automática. Este mecanismo es muy cómodo ya que dejamos que recaiga sobre Qt la tarea de ordenar las ventanas de modo que los widgets queden perfectamente alineados y se redimensionen cuando se redimensione el diálogo. Por último se pone el título a la barra superior del diálogo.

```

01 void FindDialog::findClicked()
    {
        QString text = lineEdit->text();
        Qt::CaseSensitivity cs =
05         caseCheckBox->isChecked() ? Qt::CaseSensitive
                                     : Qt::CaseInsensitive;
        if (backwardCheckBox->isChecked()) {
            emit findPrevious(text, cs);
        } else {
10         emit findNext(text, cs);
        }
    }

```

El método `findClicked` se activa cuando el usuario pulsa el botón `find`. Como puede verse lo que hace es tomar el texto de la caja de texto, comprobar que el botón de opción `caseCheckBox` está o no pulsado y dependiendo si hemos activado la búsqueda inversa emitirá la señal `findPrevious` o `findNext` con los datos que hemos recogido en las líneas 3 y 4.

```

01 void FindDialog::enableFindButton(const QString &text)
    {
        findButton->setEnabled(!text.isEmpty());
    }

```

Por último, el método `enableFindButton` se ejecuta cada vez que el usuario escribe o borra algo en la caja de texto. Simplemente comprobamos que si la caja no está vacía debemos activar el botón de búsqueda.

3.1.3. Ventajas e inconvenientes del uso de Qt

Aunque las ventajas de Qt son grandes, especialmente si las comparamos contra herramientas con el mismo objetivo como GTK o Tcl/tk, no está desprovisto de inconvenientes. A continuación indico algunos argumentos tanto a favor como en contra del uso de Qt:

- Sencillez de su programación, y por lo tanto un rápido aprendizaje.
- Su abundante, centralizada y ejemplar documentación resuelve con rapidez cualquier duda que pueda surgir durante su uso.
- Multiplataforma. Es posible compilar el código para Windows, Mac y Linux en las arquitecturas más populares. Además es posible usar los programas algunos nuevos móviles Nokia.
- Existen multitud de entornos de desarrollo especializados para programar en C++ usando la librería Qt.
- Facilidad para la traducción de las aplicaciones a cualquier otro idioma a través de la herramienta Qt Linguist.
- Posibilidad de crear ventanas y diálogos sin necesidad de escribir código, mediante la herramienta Qt Creator.
- Aunque se vaya a programar en un paradigma no orientado a objetos, el uso de C++ obliga a ello, aunque solo en el código referente a la interfaz gráfica.
- La librería Qt no está por lo general instalada en una distribución GNU/Linux con Gnome y aun menos habitual es encontrarla en Windows.
- La librería para ordenadores personales es bastante pesada, no siendo recomendable su uso en ordenadores lentos o antiguos.

3.2. Estructura del proyecto software

El desarrollo del proyecto software se puede dividir en varios tramos. El primer paso sería diseñar una gramática para conseguir validar la cadena de entrada (cualquier código tres direcciones válido) y seguidamente implementar el analizador léxico y el analizador sintáctico para poder recorrer esa cadena de entrada y descubrir sus posibles errores.

Una vez se puede recorrer la cadena de entrada validando su estado o extrayendo sus errores, hay que almacenar todos los datos leídos en unas estructuras con un determinado formato para que a partir de ese momento podamos acceder automáticamente a cualquier instrucción para obtener cualquier parámetro o argumento.

Paralelamente se puede empezar a diseñar todas las ventanas, marcos, barras de herramientas, menús, etcétera de lo que va a ser el interfaz gráfico con el que

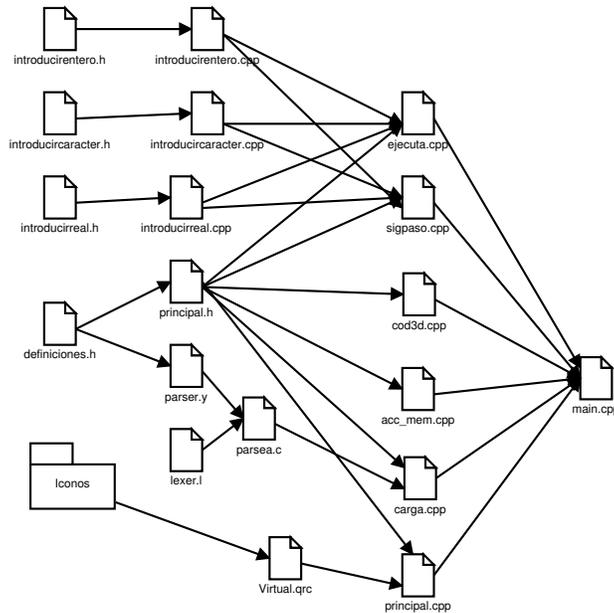


Figura 3.2: Diagrama de archivos del proyecto.

se va a encontrar el usuario. Una vez diseñado, hay que introducir todos los comandos de la barra de menú, los botones para realizar acciones directas y los eventos asociados a los elementos gráficos para que el usuario pueda interactuar con el programa.

Finalmente queda la parte más importante del trabajo, que constituye la programación de las acciones lógicas que el intérprete gráfico debe realizar con las instrucciones, basadas en el ciclo clásico constituido por la búsqueda de la instrucción, decodificación, búsqueda de operandos, operación, almacenamiento del resultado y detección de excepciones. [4]

El código se ha dividido en varios ficheros para facilitar su modificación y hacer más liviana la compilación. Un diagrama de las partes se muestra en la figura 3.2.

3.3. Codificación del analizador léxico, sintáctico y semántico.

3.3.1. Herramientas usadas

Para el analizador léxico y sintáctico he preferido usar Flex y Bison¹, herramientas de generación de código consolidadas en vez de nuevas herramientas

¹Se puede encontrar más información en su web dentro del proyecto GNU: <http://www.gnu.org/software/>

que permiten generar parsers en lenguajes orientados a objetos, como ANTLR o el generador experimental de Qt, QLALR, muy prometedor pero sin apenas documentación. La ventaja de usar Flex y Bison es bien clara: Funciona, y muy bien además. Sin embargo haber usado estas dos veteranas herramientas me ha supuesto algunos inconvenientes:

- Debemos compilar el código generado por Flex y Bison con un compilador de C, mientras que el código del intérprete gráfico será compilado en uno de C++. Sin embargo incluir código C en un programa en C++ no conlleva problemas mientras la función se declare de la siguiente forma:

```
extern "C" int parsea (char *fichero_entrada /*...*/);
```

Como puede verse al principio del fichero `mvirtual/carga.cpp`.

- El compilador no soportará caracteres unicode. En cualquier caso para nosotros no es un problema, dado que el código que aceptará nuestro analizador será texto ANSI.
- La herramienta de generación de Makefile en Qt, `qmake`, no resolverá la dependencia de forma adecuada, siendo necesario resolverla manualmente. Para ello, en el *script* de compilación del fichero `do` uso la orden `sed`² para añadir manualmente la dependencia.

3.3.2. Análisis del código intermedio

Las intrucciones en formato de código de tres direcciones son un tipo de representación intermedia que expresan las operaciones necesarias en la máquina de destino pero sin tener en cuenta los detalles específicos del hardware, como ya hemos introducido en la sección 1.2.1.2. Para elegir un buen tipo de código intermedio debemos comprobar que cumplan los siguientes requisitos:

- Debe ser fácil de escribir en la fase del análisis semántico.
- Ha de resultar sencillo traducir el código intermedio a un lenguaje máquina real, de arquitectura específica.
- Las instrucciones de código intermedio deben tener un significado claro y carente de ambigüedad para optimizar las transformaciones que optimizan el código intermedio. [2]

Hay tres tipos básicos de código intermedio que cumplen las premisas. El código intermedio en forma de árbol sintáctico, el código intermedio de notación postfija y finalmente el que ejecutará nuestro intérprete, el código intermedio de tres direcciones.

²Para más información se puede consultar *man sed* en cualquier distribución GNU/Linux.

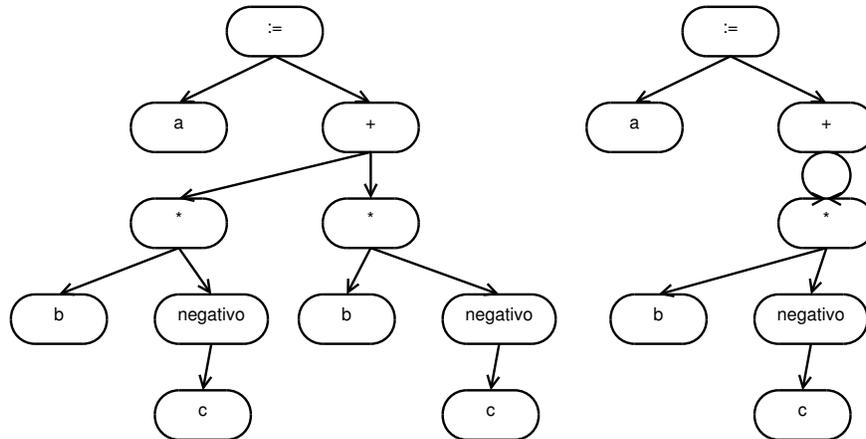


Figura 3.3: Representación gráfica de $a := b(-c) + b(-c)$ tanto de forma directa como compacta.

3.3.2.1. Otros tipos de código intermedio

El código intermedio en forma de árbol sintáctico se estructura como un grafo acíclico dirigido. En él se pueden identificar subexpresiones comunes y compactar las expresiones (ejemplo en la figura 3.3). [1]

En cambio el código intermedio en notación postfija es una representación linealizada de un árbol sintáctico. La lista de los nodos de un árbol aparece inmediatamente después de sus hijos. Vemos como ejemplo que la representación de la expresión de la figura 3.3 en notación postfija sería:

a b c negativo * b c negativo * + :=

Como las aristas del grafo en notación postfija no aparecen de forma explícita debemos saber el número de operandos que espera el operador de un nodo. Así la suma, multiplicación y asignación esperan dos nodos, pero el cambio de signo solo espera uno. La recuperación de las aristas se evalúa mediante una pila. [1]

3.3.2.2. Código intermedio de tres direcciones

El intérprete va a ejecutar programas escritos con instrucciones de código de tres direcciones sencillas y de formato invariable. Aunque todas están debidamente detalladas en el apéndice A es conveniente indicar los distintos tipos de instrucción y sus formatos.

Las instrucciones de código intermedio que acepta el intérprete están formadas por su código, que indica la operación que la instrucción realizará seguido de tres parámetros. Por lo general dichos parámetros aportarán los datos que serán procesados por la instrucción y la posición de memoria donde se almacenará el resultado. También podemos encontrarnos con instrucciones que modifican el flujo del programa, donde se comprueba cierta condición con respecto a los

datos de los dos primeros argumentos y en el caso de cumplirla se salta a la instrucción señalada por el tercer parámetro. Además hay instrucciones de entrada y salida, saltos incondicionales, instrucciones para el manejo de vectores, que modifican el puntero a pila, que modifican el puntero a *frame*...

Nuestro intérprete se comporta como una máquina de memoria en forma de pila, sin registros de propósito general. La memoria del intérprete puede representarse como un vector de mil posiciones donde en cada una de las cuales puede escribirse un número entero o real³ y se accede a él de forma aleatoria, aunque el código de tres direcciones está orientado a que se use en forma de pila de control. Los registros de activación se introducen y se sacan cuando los procedimientos empiezan y terminan respectivamente.

Además hay tres importantes punteros:

- El puntero a cima marca la posición de memoria donde se deberá escribir el próximo registro de activación. Se modifica con las instrucciones de tipo PUSH, POP, TOPFP, INCTOP y DECTOP.
- Nuestro código de tres direcciones no contiene instrucciones para el manejo de un display, así que no será posible traducir lenguajes de programación en los que se puedan anidar definiciones de funciones. (Ver el capítulo “Acceso a variables no locales” en [1]) Sin embargo tenemos un puntero de marco que separará las variables globales de las locales. Para modificar este puntero se deben usar las instrucciones FPTOP y FPPOP.
- El contador de programa, que siempre apuntará a la siguiente instrucción que deba ser ejecutada. Cada vez que se ejecute una instrucción deberá incrementarse en una unidad excepto en una instrucción de salto, donde el contador se actualizará al valor del destino del salto.

Todos los datos de las instrucciones serán datos inmediatos –se obtienen al decodificar la instrucción– o bien se indicará la posición de memoria donde se encuentra en un formato de nivel y desplazamiento. La forma de decodificar esta tupla a la dirección física es sumándole al desplazamiento el puntero a marco en el caso que nivel indique que buscamos una variable local (valor de nivel *true*) y solo el desplazamiento si lo que queremos es una variable global (valor de nivel *false*). Sin embargo para dejar los resultados se deberá indicar la dirección de memoria.

Hay varios tipos de instrucciones:

- Aritméticas, tanto para enteros como para reales.
- De conversión entre enteros y reales.
- De modificación del flujo del programa, tanto para saltos incondicionales como condicionales, incluyendo comparaciones entre reales y enteros.
- De modificación del puntero a pila y el puntero a marco.

³Aunque un entero y un real pueden no tener el mismo tamaño, el intérprete asume que todos los tipos de dato ocupan una posición de memoria.

- De llamada y retorno de funciones.
- De acceso a vectores.
- De entrada y salida y otros.

Todas las instrucciones se pueden ver en detalle en la sección A.

3.3.3. Codificación del analizador léxico de los ficheros de entrada

Las instrucciones de código intermedio pueden contener los siguientes tokens que serán identificados gracias al código Flex contenido en el archivo `parser_c3d/lexer.l`. Si en el fichero de entrada se encuentran tokens no definidos aquí, al cargar el programa se mostrará un error léxico. La especificación léxica del fichero de entrada se comprueba mediante las siguientes expresiones regulares:

- | | |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| “#”.\n | Comentario. Todo lo que se haya escrito entre la almohadilla y el salto de línea se deshecha. |
| [“ \t”+] | Cualquier aparición, separada o seguida de espacios en blanco o tabuladores son separadores sin ningún significado sintáctico. El analizador léxico los deshecha. |
| \n | Token nueva línea. Este token delimita una instrucción de la siguiente. |
| “-”?[0-9]+ | Token entero. Más tarde el analizador sintáctico lo empleará en varios casos. Le pasa al analizador sintáctico un número entero como valor semántico. |
| “-”?[0-9]+.”[0-9]* | Token real. Luego el analizador sintáctico lo empleará en el caso de un argumento real. Le pasa al analizador sintáctico un número en punto flotante como valor semántico. |
| “(” | Token abrir paréntesis. Sintácticamente se usa al delimitar un puntero, para resolver la ambigüedad de la coma, que separa tanto un argumento como las dos partes que componen una posición de memoria. |
| “)” | Token cerrar paréntesis. |
| “,” | Token coma. Sintácticamente se usa al delimitar los argumentos para evitar ambigüedad en el caso del argumento nulo, y para delimitar las dos partes del argumento posición. |
| “e:” “p:” “i:” “r:” | Tokens etiqueta, puntero, entero y real respectivamente. Se usan para darle un tipo al argumento que vendrá a continuación. |

Instrucciones También tendremos un token por cada código de instrucción que decodifica el intérprete, por ejemplo FIN, CREAD, ESUM, etc.

Hay que tener en cuenta que el analizador no distinguirá entre mayúsculas o minúsculas.

Flex traducirá el fichero `parser_c3d/lexer.l` en un fichero de código C en los cuales estarán programados los autómatas finitos correspondientes. Para más información sobre el proceso acudir a la sección 1.2.3.1.

3.3.4. Codificación del analizador sintáctico-semántico de los ficheros de entrada

Las instrucciones de código intermedio tienen la siguiente especificación sintáctica:

```
Num_linea Instruccion Arg1, Arg2, Arg3
```

Siendo el número de línea un número entero, la instrucción una palabra mnemotécnica (Ver sección A para un listado completo) y los argumentos pueden ser alguno de los cinco siguientes:

- Argumento nulo, en cuyo caso no habrá nada escrito.
- Argumento real. Se indicará con `r:` y a continuación el número real escrito sin usar notación científica. (Ej: `r: -4.52`)
- Argumento entero. Se indicará con `i:` y a continuación el número entero. (Ej: `i: 560`)
- Argumento etiqueta. Usado por las instrucciones que modifican el flujo de instrucciones del programa. Se indicará con `e:` y a continuación el número de una instrucción existente en nuestro programa.
- Argumento posición. Usado como una referencia a una posición de memoria. Se indica con `p:` y a continuación, separado por una coma y entre paréntesis, dos números enteros positivos.
 - El primero de ellos puede ser un cero o un uno e indica si la posición de memoria a la que apunta forma parte del marco global del programa o del marco de la función en ejecución (Cero indica global, un uno indica local).
 - El segundo es el desplazamiento. Es un número entero positivo o negativo que deberemos sumar al *frame pointer* para obtener la dirección de memoria efectiva.

A partir de los tokens detectados en el analizador léxico, el analizador sintáctico tiene que hacer su trabajo. Nuestro lenguaje es sintácticamente muy simple. Tanto que de hecho podría resumirse en las siguientes reglas:

```

programa: programa ENTERO instruccion argumento COMA
         argumento COMA argumento N_LINEA | ;
instruccion: ESUM | RSUM | EDIF | RDIF | EMULT
            | RMULT /* ... */ ;
argumento: T_ENTERO ENTERO
          | T_POSICION A_PARENTESIS ENTERO COMA ENTERO C_PARENTESIS
          | T_REAL REAL | T_ETIQUETA ENTERO | ;

```

Sin embargo esto complicaría sobremanera nuestro analizador semántico, ya que para cada instrucción deberíamos comprobar si sus argumentos, si los tiene, son del tipo correcto. La lista de errores semánticos que deberíamos comprobar antes de la carga los podemos encontrar en la sección 4.4.2. Para evitar la tediosa comprobación de los tipos hemos preferido aumentar el tamaño del analizador sintáctico, reduciendo el semántico a su mínima expresión. El analizador se guarda en el fichero `parser_c3d/parser.y` en nuestro proyecto. Las reglas están definidas siguiendo el esquema que muestro a continuación:

1. Defino una regla por cada tipo de parámetros que admite una instrucción. Así tendremos reglas como `inst_n_n_ip`, `inst_p_ip_p`, etc.

a) Los nombres de las reglas siguen el patrón de llamarse `inst`, seguido de los tres tipos de *parámetros* separados entre guiones bajos. Hay 6 tipos:

- 1) `n` para parámetros nulos.
- 2) `i` para parámetros enteros (*integer*)
- 3) `p` para direcciones de memoria (*posición*)
- 4) `e` para direcciones de salto (*etiquetas*)
- 5) `ip` si el parámetro puede ser una dirección de memoria o un entero.
- 6) `rp` si el parámetro puede ser una dirección de memoria o un real.

b) Cada uno de los no terminales del tipo `inst_x_x_x` deberá derivarse en cada una de las instrucciones que acepta este tipo de parámetros. Recordemos que cada instrucción solo deberá estar en un grupo. Por cada regla deberemos subir como valor semántico –la asignación a `$$`– el tipo de la instrucción. Por ejemplo:

```

inst_n_n_e : GOTOS_ { $$ = GOTOS; }
          | CALL_  { $$ = CALL;  } ;

```

Equivale a decir que solo hay dos instrucciones que admitan dos parámetros nulos y una etiqueta: `GOTO` y `CALL`.

2. Defino las reglas que admiten tanto un parámetro positivo como una posición, o real y posición:

```

arg_posi_ente : arg_posi { $$ = $1; }
             | arg_ente { $$ = $1; } ;

```

```
arg_posi_real : arg_posi { $$ = $1; }
              | arg_real { $$ = $1; } ;
```

Reglas que simplemente permiten elegir entre dos tipos y suben su valor semántico.

3. Por último, en la regla instrucción subo a la raíz del árbol semántico los valores para pasarlos al módulo principal del intérprete.

```
instruccion : inst_ip_ip_p arg_posi_ente COMA_
              arg_posi_ente COMA_
              arg_posi
            { $$ . cop = $1;
              $$ . arg[0] = $2;
              $$ . arg[1] = $4;
              $$ . arg[2] = $6; } | /* etc */ ;
```

En cualquier caso, para añadir funcionalidades al intérprete (ver sección 3.8: Manual de ampliaciones) solo deberíamos añadir las nuevas instrucciones en las reglas del tipo `inst_x_x_x`.

Para profundizar más en el tema recomiendo leer el capítulo sobre la creación de gramáticas en Yacc/Bison en [1].

3.3.5. Comunicación entre el módulo de carga de programas y el núcleo del intérprete

Para comunicar el resultado del análisis de nuestro fichero de entrada al intérprete gráfico en un principio se pensó en acceder desde el módulo a las estructuras de datos que existen en el núcleo del intérprete y rellenar los datos adquiridos. Sin embargo no podremos hacerlo porque el módulo está escrito en C (ver sección 3.3.1) y las estructuras de datos en el intérprete son clases. De este modo necesitamos crear una sencilla estructura de datos consistente en un vector de la siguiente estructura llamada `tipo_ctd` definida en el archivo `virtual/definiciones.h`:

```
struct tipo_ctd { int lin, cop; struct tipo_arg arg[3]; };
struct tipo_arg { int tipo, i_pos, niv; float r; };
```

La estructura principal guarda el número de línea, el código de la operación y los tres argumentos de la instrucción analizada. La estructura secundaria guarda cada argumento, consistentes en un código para el tipo (nulo, entero, posición, etiqueta, entero o real) y las variables mínimas necesarias para almacenarlo. Además el módulo manda al intérprete el número de instrucciones leídas, el número de errores detectados y un vector de caracteres con la descripción y el número de línea donde se encuentren los posibles errores.

Deberemos emplear la directiva de preprocesador `extern` para llamar a la función desde un código en C++, tal y como se explica en la sección 3.3.1.

3.4. Tipos de datos en el intérprete gráfico

En el intérprete se trabajará básicamente con dos tipos de datos, las instrucciones del programa cargado y los datos guardados en la memoria por dichas instrucciones. A su vez han de guardarse de dos modos distintos, uno para que al algoritmo de ejecución le sea fácil de manejar y otro para que pueda ser representado en los widgets que muestran los datos al usuario. De modo que en la clase principal que defina nuestra aplicación encontramos las siguientes instancias de objetos (en el archivo `principal.h`):

```
QList<QTreeWidgetItem *> rep_inst, rep_mem;
QList<c_tipo_ctd *> list_inst;
QList<c_tipo_mem *> list_mem;
```

Las cuatro instancias son listas. `Rep_inst` y `rep_mem` representan en los paneles las instrucciones y las memoria respectivamente. Contienen elementos del tipo `QTreeWidgetItem`, clase predefinida en Qt. En cambio las listas `list_inst` y `list_mem` contienen las intrucciones y la memoria que ejecuta el intérprete. Estas listas contienen instancias de las clases `c_tipo_ctd` y `c_tipo_mem`, que paso a describir a continuación.

3.4.1. Codificación de las instrucciones

Las instrucciones que se leen en el proceso de carga acaban finalmente guardadas en una lista que contiene elementos de la clase `c_tipo_ctd`. La clase está definida en el archivo `virtual/principal.h` como:

```
enum t_cod_ins {NULO = 0, ESUM, RSUM, EDIF,
               RDIF, EMULT, RMULT, EDIVI,
               RDIVI, RESTO, CONV /*...*/};
enum t_cod_arg {ARG_NULO = 0, ARG_ENTERO, ARG_REAL,
               ARG_POSICION, ARG_ETIQUETA};
enum error_mem {NO_ERROR = 0, ERROR_VACIA, ERROR_NOEXISTE,
               ERROR_TIPO, ERROR_ARG};
struct st_posmem {
    int i_e_pos;
    bool niv;
};
union un_arg {
    float r;
    st_posmem i_e_pos_niv;
};
struct st_arg {
    t_cod_arg tipo_arg;
    un_arg dato;
};
class c_tipo_ctd {
```

```

public:
    c_tipo_ctd();
    t_cod_ins r_cop();
    bool r_brp();
    error_mem r_arg(int p_arg, t_cod_arg *p_cod);
    error_mem r_int(int p_arg, int *p_val);
    error_mem r_eti(int p_arg, int *p_val);
    error_mem r_pos(int p_arg, bool *p_niv, int *p_pos);
    error_mem r_rea(int p_arg, float *p_fval);
    void s_cop(t_cod_ins p_cop);
    void s_brp(bool p_brp);
    error_mem s_int(int p_arg, int p_val);
    error_mem s_eti(int p_arg, int p_val);
    error_mem s_pos(int p_arg, bool p_niv, int p_pos);
    error_mem s_rea(int p_arg, float p_fval);
    void setnull(); // Resetea todos los campos
    error_mem setnull(int p_arg); // Resetea un argumento
private:
    t_cod_ins cop; // Código de instrucción
    bool breakpoint; // Indica la condición de punto de ruptura
    st_arg v_arg[3]; // Vector con los tres argumentos
};

```

En el código se puede apreciar tanto los datos privados como los métodos públicos que modifican su estado. Los métodos están diferenciados por el prefijo `r_` para los que leen un dato de la clase y el `s_` para los que escriben. Mientras no se advierta lo contrario, las llamadas pasan los datos por valor y devuelven por referencia, ya que se aprovecha el valor de retorno de las funciones para indicar errores.

El constructor de la clase crea una nueva instrucción con todos los valores nulos. Así después de crearlo debemos insertar con los métodos adecuados los valores. Si tenemos la siguiente instrucción:

```
23 EDIVI i:8, p:(0,2), p:(1,4)
```

Para almacenarla adecuadamente debemos ejecutar los siguientes pasos:

```

c_tipo_ctd *instruccion = new c_tipo_ctd;
instruccion->s_cop(EDIVI);
instruccion->s_int(0 /*Número de argumento*/, 8 /*Valor*/);
instruccion->s_pos(1 /*Número de argumento*/, false /*Nivel*/,
                 2 /*Desplazamiento*/);
instruccion->s_pos(2 /*Número de argumento*/, true /*Nivel*/,
                 4 /*Desplazamiento*/);

```

Aunque no es estrictamente necesario es una buena idea comprobar el tipo error devuelto para cerciorarse de que el método se ha ejecutado correctamente. Cada error tiene un significado distinto:

- Si como número de parámetro no introducimos un entero entre 0 y 3 devolverá el error `ERROR_ARG`.
- Si leemos un parámetro y su tipo no coincide con el método empleado (Ej: Leer un argumento entero con el método `r_rea`) devolverá el error `ERROR_TIPO`.

Las funciones están perfectamente comentadas en el código fuente, siendo útil su lectura en el caso de querer ampliar las funcionalidades del intérprete o conocer mejor su funcionamiento interno.

3.4.2. Codificación de la memoria

La memoria se concibe como una lista de instancias de la clase `c_tipo_mem`. La clase se define del siguiente modo en el archivo `virtual/principal.h`:

```
enum tipo_mem {MEM_UNUSED = 0, MEM_INT, MEM_REAL};
enum error_mem {NO_ERROR = 0, ERROR_VACIA, ERROR_NOEXISTE,
               ERROR_TIPO, ERROR_ARG};

union un_mem{
    int i;
    float r;
};
class c_tipo_mem{
public:
    c_tipo_mem();
    tipo_mem r_tip();
    error_mem r_int(int *p_val);
    error_mem r_rea(float *p_fval);
    void s_int(int p_val);
    void s_rea(float p_fval);
    void reset();
private:
    tipo_mem t;
    un_mem celda;
};
```

Su comportamiento es similar al la clase `c_tipo_ctd`. El constructor crea una celda de memoria vacía, el método `reset` pone otra vez a `MEM_UNUSED` el estado de memoria y los posibles errores son:

- Leer una celda de memoria con un método no válido (por ejemplo, leer un real con `r_int`) devolverá un `ERROR_TIPO`.
- Leer una celda de memoria sin usar (marcada como `MEM_UNUSED`) devolverá el resultado `ERROR_VACIA`.

3.4.3. Variables de estado del intérprete gráfico

El intérprete tiene declaradas ciertas variables que definen su estado. Están definidas como variables privadas de la clase `Principal`:

`textobarra`: Cadena de texto que se muestra en la barra de título de la ventana principal. Habitualmente es el nombre del archivo abierto, seguido del nombre de la aplicación (Ej: `codigo01.c3d - FalsPas`).

`ficheroactual`: Cadena de texto que contiene el nombre del fichero abierto con su ruta completa. Útil al usar la acción de recargar fichero.

`textosalida`: Contiene el texto que se mostrará en la ventana de salida de datos y avisos.

`totalbp`: Contiene el número de puntos de ruptura activos.

`total_inst`: Contiene el número total de instrucciones del programa cargado.

`inst_actual`: Contiene el número de la primera instrucción que se operará cuando se ejecute el programa. Al igual que un contador de programa de un procesador lo podrá modificar una instrucción de salto y se incrementará en una unidad cada vez que se ejecute una instrucción.

`ult_escr`: Booleano que indica si la última instrucción ejecutada ha escrito algún dato en memoria.

`p_top`: Puntero a cima de pila.

`p_fp`: Frame pointer.

`muestra_excepciones`: Booleano que en el caso de estar activo mostrará en el panel de salida de errores todos los warnings y excepciones que pueda ocasionar el programa en su ejecución.

`desp_mem`: Booleano que en caso de estar activo provoca que el panel de instrucciones y de memoria haga *scroll* de forma automática a la próxima instrucción en ejecutarse y al último dato escrito, respectivamente.

`pila_estricta`: Booleano que en caso de estar activado mostrará avisos en el panel de salida de errores si se intenta escribir en posiciones de memoria superiores al puntero a cima.

3.5. Instrucciones en la ruta de datos

En las secciones siguientes mostraremos como se ejecuta una instrucción desde el momento de su búsqueda hasta la actualización del contador de programa. El código puede encontrarse en los archivos `sigpaso.cpp` y `ejecuta.cpp`.

3.5.1. Proceso de decodificación de una instrucción y búsqueda de operadores

Cada vez que el intérprete necesite ejecutar una instrucción utiliza alguno de los siguientes métodos para decodificar la instrucción y buscar los operadores. Los métodos de esta sección se encuentran definidos en el archivo `principal.h`.

- Para leer el código de instrucción usamos `r_cop()`, como hemos visto en la sección 3.4.1.

```
t_cod_ins codigo = instruccion->r_cop();
```

- Comparamos el código de la instrucción en un `switch` para usar el método correspondiente con la semántica de la operación. Se puede encontrar en el archivo `ej_inst.cpp`.

```
/* la instrucción tiene un codigo == ESIG */
switch(codigo){
/*...*/
case ESIG:
```

- Recuperamos el único operador que tiene la instrucción `ESIG`. Recordemos que el analizador sintáctico nos evita tener que comprobar que sus parámetros sean correctos:

```
int op;
error_mem error = recupera_op(0, instruccion, &op);
```

Ahora en la variable `op` tendremos el dato correspondiente al primer argumento de la instrucción, sin importar si el argumento de la instrucción es un entero o una posición de memoria. El método `recupera_op` la recupera por nosotros.

El método `recupera_op` está sobrecargado para que sea posible usarlo con cualquier argumento:

- `error_mem recupera_op(int nargs, c_tipo_ctd *inst, int *val)` se usa para recuperar valores enteros, ya sea entero, etiqueta o posición de memoria que contenga un entero.
- `error_mem recupera_op(int nargs, c_tipo_ctd *inst, float *fval)` recupera reales, ya sea de un real o de una posición de memoria que contenga un real.
- `error_mem recupera_op(int nargs, c_tipo_ctd *inst, bool *niv, int *pos)` recuperará el nivel y el desplazamiento de un argumento tipo posición. Realmente lo usan las dos funciones anteriores.

Si ha habido problemas recuperando en memoria algún dato, el valor `error_mem` de retorno nos informa, ya bien sea `ERROR_NOEXISTE` si la posición de memoria es negativa o supera el límite, `ERROR_TIPO` si el dato en la memoria no era del tipo solicitado o `ERROR_VACIA` si intentábamos leer de una posición de memoria en la que aún no se ha escrito.

3.5.2. Ejecución de la instrucción y almacenamiento del resultado

Cuando ya hemos decodificado la instrucción y tenemos los operadores falta ejecutar la instrucción y guardar el resultado. Siguiendo el ejemplo de la sección anterior, esto sería:

```
error_mem err = escribe_mem(2, instruccion, -op);
```

Es decir, escribirá en la posición de memoria indicada en el tercer argumento de la instrucción el operando cambiado de signo.

Al igual que para leer en memoria, para escribir en memoria también tenemos la misma instrucción sobrecargada varias veces:

- `error_mem escribe_mem(int nargs, c_tipo_ctd *inst, int val, bool p_muestra = true)`
- `error_mem escribe_mem(int nargs, c_tipo_ctd *inst, float fval, bool p_muestra = true)`
- `error_mem escribe_mem(bool niv, int pos, int val, bool p_muestra = true)`
- `error_mem escribe_mem(bool niv, int pos, float fval, bool p_muestra = true)`
- `error_mem escribe_mem(int pos_abs, int val, bool p_muestra = true)`
- `error_mem escribe_mem(int pos_abs, float fval, bool p_muestra = true)`

Como podemos observar, podemos escribir en memoria un número, ya sea real o entero pasando un argumento de instrucción (que debe ser de tipo posición), pasando su nivel y desplazamiento o pasando la dirección absoluta de memoria. El último parámetro modifica el comportamiento del intérprete gráfico de cara al usuario. Un valor `true` (por defecto) modifica el panel de representación de la memoria de forma inmediata, mientras que un valor `false` no lo hace. Esto es así para cumplir con el requisito software número 14 (página 22), y se pondrá a valor `false` cuando se estén ejecutando instrucciones en modo continuo.

3.5.3. Funciones que modifican el flujo de ejecución o los punteros del intérprete y el tratamiento de puntos de ruptura

Además de las instrucciones típicas que leen datos y finalmente los escriben en la memoria también tenemos instrucciones que modifican el flujo de ejecución, el *frame pointer* o el puntero a cima. Para ello, después de decodificar la instrucción y buscar sus operadores, si una instrucción modifica la cima (por ejemplo INCTOP, EPUSH, EPOP, etc.) puede usar la instrucción:

```
error_mem set_top(int n_top, bool p_muestra = true)
```

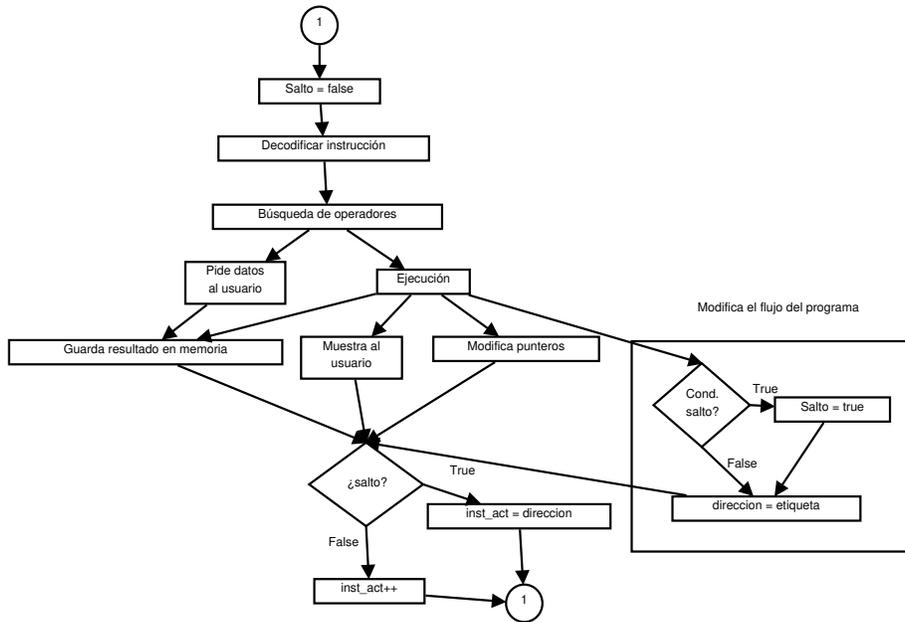


Figura 3.4: Flujo del programa con instrucciones de salto.

Además de modificar la variable `p_top` borrará de memoria cualquier dato por encima de la posición de la cima, no se representará en pantalla su acción si el segundo parámetro es `false` y el error que puede cometer la instrucción (llevar el puntero a una posición negativa o fuera de memoria) será avisado devolviendo la constante `ERROR_NOEXISTE`.

La instrucción que modifica el frame pointer es muy similar:

```
error_mem set_fp(int n_fp, bool p_muestra = true);
```

La única diferencia con la anterior es que ésta no borra de la memoria las posiciones superiores de memoria ocupadas.

El método que usa el intérprete gráfico para cambiar el flujo de las instrucciones es muy simple, basándose en una variable temporal de estado que indica si se ha ejecutado una instrucción de salto con la condición cierta. Se muestra con toda claridad en la figura 3.4.

Para parar el flujo de instrucciones ante una instrucción marcada con un punto de ruptura simplemente hay que comprobar con la orden `bool parar = instruccion->r_brp();` antes de su ejecución y abandonar el bucle en caso afirmativo.

3.6. Aspectos de la interfaz de usuario

En la interfaz he usado algunos de los aspectos más importantes que ofrece Qt de cara al usuario. Vamos a mostrar cada uno de sus elementos, indicando para cada uno de ellos sus mecanismos más significativos y los posibles problemas que pueden surgir durante su uso.

3.6.1. QMainWindow: Ventana principal

La clase `QMainWindow` es el eje donde gira nuestro programa⁴. La clase principal hereda directamente de la anterior. Entre sus propiedades se encuentra la de poder añadirle barras de estado, barras de menús, barras de herramientas, etc. En el proyecto la configuración de la ventana principal se ejecuta básicamente en el constructor de la clase principal. Entre todas las acciones destacamos:

- `menuBar()` devuelve un puntero a la barra de menús de la ventana. Podremos añadirle menús usando el método `addMenu()`. Por ejemplo, para establecer el menú archivo usaríamos:

```
menu_archivo = menuBar()->addMenu("Archivo");
```

Al que luego tendríamos que añadir las acciones del menú mediante el método `addAction()`:

```
menu_archivo->addAction(accion_abrir);
menu_archivo->addAction(accion_recargar);
menu_archivo->addAction(accion_salir);
```

- `addToolBar()` permite introducir una barra de herramientas en la posición de la ventana que queramos. Por defecto la añade arriba de la ventana, debajo de la barra de menús. El intérprete gráfico tiene tres barras de menú: `barra_ejecutar`, `barra_archivo` y `barra_Ayuda`. Un ejemplo de su uso puede ser el siguiente:

```
barra_ayuda = addToolBar("Ayuda");
```

Al que luego, como en el caso anterior, debemos introducir las acciones necesarias:

```
barra_ayuda->addAction(accion_ayuda);
barra_ayuda->addAction(accion_acerca);
```

- `statusBar()` nos permite añadir una barra de estado en la que podemos poner el widget que queramos. En el proyecto la barra de estado cuenta con una etiqueta de texto con la que lanzamos mensajes cada vez que ejecutamos una acción. La forma en la que inicializamos la barra es:

⁴Recomiendo acudir a la documentación Qt 4 Assistant –se instala a la vez que las librerías de desarrollo– para profundizar en la gran cantidad de posibilidades que ofrece.

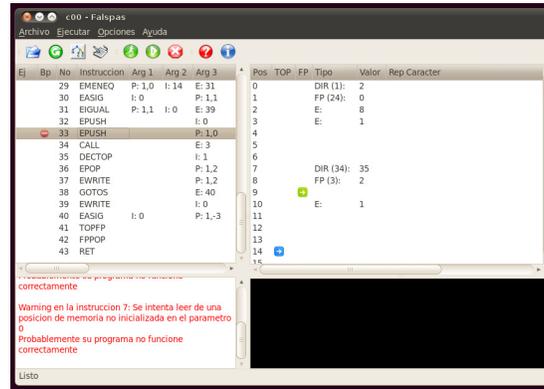


Figura 3.5: Como puede comprobarse, las tres barras desplazables que hay entre los cuatro espacios son la representación de los QSplitter, mientras que los dos cuadros de abajo son de texto y los superiores son QTreeWidget.

```
QLabel *texto_barra_estado = new QLabel("Listo");
statusBar()->addWidget(texto_barra_estado);
```

- Por último podemos poner un icono y un título a la ventana con `setWindowIcon()` y `setWindowTitle()` respectivamente, y establecer el widget principal de la ventana con `setCentralWidget()`.

La ventana principal se define en el archivo `principal.h` y se modifican sus parámetros para que muestre el estado que deseamos en el método `Principal::Principal()`.

3.6.2. Estableciendo el widget central

Tras definir la ventana principal debemos establecer un widget como elemento principal de la ventana. Para ello debemos hacer una aclaración. Hay dos tipos de widgets:

Widgets interactivos Son los widgets con los que el usuario interactúa, como los botones, botones de opción, cajas de texto...

Widgets contenedores Son widgets diseñados para contener otros widgets, como las carpetas de pestañas, *frames*, cajas de *scroll*...

Por lo tanto en la ventana principal hemos añadido tres QSplitter donde insertaremos en los cuatro huecos que dejan una caja de salida de texto para los datos de salida del intérprete, otra caja de texto para la salida de errores y dos QTreeWidget para representar los paneles de las instrucciones y la memoria. En la figura 3.5 se puede ver claramente, mientras que en la figura 3.6 se ve la organización jerárquica de la misma.

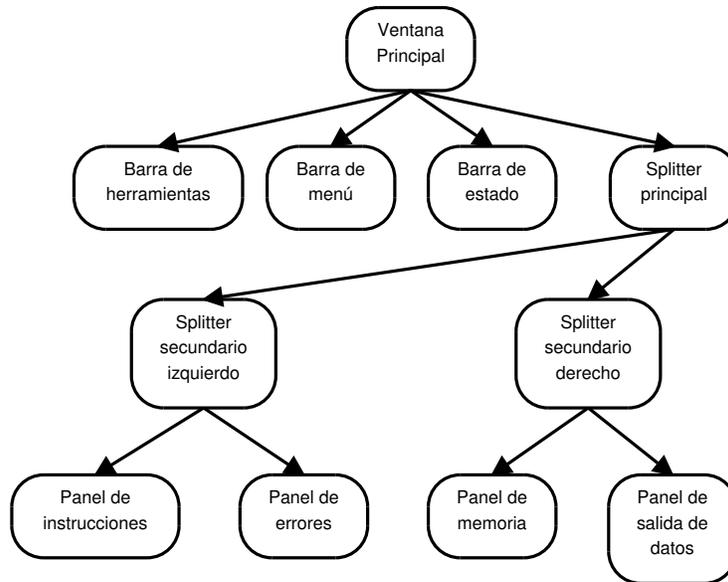


Figura 3.6: Diagrama que indica la jerarquía de widgets en la ventana principal del intérprete.

3.6.3. Usando la clase QTreeWidgetItem

La clase `QTreeWidgetItem` proporciona un espacio para mostrar de forma ordenada y jerárquica iconos y textos que estén empaquetados en instancias de la clase `QTreeWidgetItem`. Aunque estemos familiarizados con el widget gracias a su utilidad a la hora de mostrar los ficheros de un medio de almacenamiento, también se puede usar perfectamente para mostrar información que no esté jerarquizada. Simplemente asumiremos que todos los items representados forman parte del nivel inicial.

Para usarlo en nuestro proyecto es llamado de la siguiente forma, mostrado en el archivo `virtual/principal.cpp`:

```

QTreeWidgetItem *tree_inst = new QTreeWidgetItem();
tree_inst->setIndentation(0);
tree_inst->setHeaderLabels(QStringList()
    << tr("Ej") << tr("Bp") << tr("No")
    << tr("Instruccion") << tr("Arg 1")
    << tr("Arg 2") << tr("Arg 3"));
  
```

Con las instrucciones nos aseguramos un panel con una indentación de cero –no es necesaria porque no necesitamos el espacio donde se representa el árbol– y con siete columnas de representación de datos.

Más tarde cuando se necesite llenar de instrucciones el panel se usará el siguiente código:

```

QTreeWidgetItem *elemento = new QTreeWidgetItem();
elemento->setIcon(0, QIcon(style()->standardPixmap
                        (QStyle::SP_ArrowRight)));
//Esto mostrará un icono en forma de flecha en la posición 0
elemento->setText(2, '23');
elemento->setIcon(3, 'FIN');
tree_inst->addItem(elemento);

```

Además la clase nos ofrece señales que permiten interactuar con el árbol haciendo doble clic en un elemento, selecciones de los datos representados, cambiar los colores, ocultar y mostrar texto y un largo etcétera.

3.7. Modificaciones a las funciones de generación de código

Para mejorar la legibilidad del código intermedio generado por los compiladores realizados por los alumnos y evitar en medida de lo posible las confusiones que puedan acarrear que el compilador tenga una salida de programa binaria y otra ASCII, he decidido modificar el fichero `MenosC/Utils.c`. En este fichero encontramos funciones útiles para el desarrollador de un compilador, ya que contiene funciones para volcar el código, crear listas de argumentos no satisfechos o tablas de símbolos entre otras muchas posibilidades. Para que el programa se escriba de forma correcta hemos debido modificar unas pocas opciones. Las modificaciones son:

- Las posiciones de memoria pasan a escribirse de forma p: (nivel, desplazamiento)
- Dejan de emplearse tabuladores para separar los argumentos entre sí, pasando a ser separados por comas.
- Los ficheros pasan a tener una extensión `.c3d`
- La salida del fichero en modo binario muestra un aviso por la salida de error desaconsejando su uso.

3.8. Manual de ampliaciones

En esta sección vamos a mostrar como ampliar el intérprete gráfico empleando un ejemplo sencillo. Imaginémonos que queremos añadir una instrucción específica que calcule el seno de un ángulo dado un número real expresado en radianes. Para ello debemos tener en cuenta que el equipo donde pensemos compilar el intérprete debe contar con las herramientas necesarias, mostradas en la sección B.1. A partir de ahí, debemos modificar los ficheros siguientes:

1. Para empezar tendremos que asignarle un mnemotécnico a la instrucción. Siguiendo el esquema de nombres sería una buena idea llamarle RSIN. En el fichero `virtual/definiciones.h` añadiremos RSIN tanto a la definición `nombres` como a la enumeración `t_cod_ins`. Hay que tener en cuenta que el nombre en la definición deberá seguir el mismo formato que los ya presentes, es decir, << “RSIN”.
2. Abrimos el fichero `parser_c3d/lexer.l` y añadimos una nueva regla léxica. Tras todos los mnemotécnicos de instrucciones añadimos la nueva. Sería esta línea:

```
rsin {(return RSIN_); }
```

3. En el fichero `parser_c3d/parser.y` debemos añadir al final de la sección `%token` un nuevo token, `RSIN_`.
4. En el fichero `parser_c3d/parser.y` tendremos que añadir además una nueva regla sintáctica. Nuestra instrucción debe aceptar en el primer argumento un real o una dirección de memoria que contenga un real, el segundo debe ser nulo y el tercero debe ser una dirección de memoria (para dejar el resultado). Finalmente sería añadir las dos siguientes producciones:

- En la producción `inst_rp_n_p` añadiremos la regla:

```
RSIN_ { $$ = RSIN; }
```

- En cambio en la producción `instrucciones` no es necesario añadir nada. Puede parecer bastante complicado, pero en realidad es sólo indicar el tipo de argumentos que puede contener y “subir” los valores semánticos hacia la raíz del árbol. Para comprender el funcionamiento de la gramática consulta la sección 3.3.4.

5. Abrimos el fichero `virtual/ej_inst.cpp`. En la línea 39 encontraremos un bloque `switch` en el que se decodifican y ejecutan las instrucciones que tiene el intérprete cargadas. Añadimos un nuevo `case`:

```
case RSIN:
    err[0] = recupera_op(0, inst_exe, &fop1);
    err[2] = escribe_mem(2, inst_exe, sinf(fop1), false);
    break;
```

Debemos tener en cuenta que estamos empleando una función de la biblioteca estándar de C `math.h` (la instrucción `sinf`), que por defecto está incluida. Información más detallada sobre las funciones `recupera_op` y `escribe_mem` se pueden encontrar en la sección 3.5.1. En cualquier caso el código está profusamente documentado.

6. Compilamos el fichero simplemente ejecutando el script `do`. Ello debería bastar. Encontraremos el nuevo ejecutable en el mismo directorio.

Capítulo 4

Manual de usuario

4.1. Introducción rápida

El fin de este manual de usuario es el de informar a la persona que utiliza el programa de como sacarle el máximo partido posible a la aplicación del intérprete gráfico, además de detallar en profundidad todas las opciones que el intérprete gráfico ofrece. A continuación se explicará como usar la aplicación, sacando partido a todas las características que se han implementado y todas las funcionalidades que el programa permite realizar.

Se empezará explicando como lanzar la aplicación, cuales son los componentes de la ventana del programa y se explicará paso a paso cómo realizar una traza completa de un programa fuente de código tres direcciones, cómo poner y eliminar breakpoints y otras acciones de interés que se pueden realizar con la aplicación.

La mayor parte de las acciones pueden lanzarse de diferentes modos, ya bien sea desde botones directos, desde alguna opción del menú, con alguna combinación de teclas o con menús contextuales. Todas estas acciones se explicarán debidamente en el apartado donde corresponda.

Se detallará paso a paso cómo usar el programa, y se mostrarán ejemplos de todos los posibles errores que puedan darse, como por ejemplo errores léxicos o sintácticos en el programa fuente, errores semánticos en su carga o en su ejecución, errores al introducir un dato requerido, y el modo correcto de finalizar un programa.

El manual de usuario irá acompañado de ilustraciones que indicarán cuales son los elementos con los que hay que interactuar para llevar a cabo para realizar la acción que se esté explicando en ese momento.

4.2. Instalación

En un principio el programa consta solo de un ejecutable que no necesita instalación. Simplemente con copiar el fichero a un directorio donde se tenga

acceso y cambiar sus permisos deberíamos poder usarlo correctamente. Puede hacerlo con la siguiente orden.

```
cp /directorio_ejecutable/virtual ~
chmod +x virtual
```

La única dependencia que puede tener el programa con librerías no instaladas de forma predeterminada en un sistema Linux son con las librerías de ejecución Qt4. En el caso que el programa falle al iniciar indicando la falta de la librería debemos instalarla con una sencilla instrucción. Si nuestra distribución está basada en debian se instalará como `sudo apt-get install lib-qt4`. Si en cambio está basada en Red Hat será `sudo yum install libqt4`.

Para ahorrar todos los inconvenientes antes reseñados, también se distribuye el ejecutable en código fuente. Para compilar el programa sólo hay que ejecutar el script de compilación

```
chmod +x do
./do
```

Y tras el proceso de compilación el script dejará el ejecutable en la misma carpeta de entrada.¹

Finalmente, para ejecutar el intérprete gráfico deberemos escribir en la consola

```
./virtual
```

O bien haciendo doble clic en un entorno de ventanas. Recordamos que si está ejecutando el entorno de ventanas KDE 4, la librería Qt4 por definición estará seguro instalada en nuestro sistema. Recomendamos lanzar por primera vez la aplicación desde una consola, ya que así podremos leer cualquier mensaje de error que aparezca. Si lo ejecutamos haciendo doble clic los mensajes no aparecerán.

4.3. Primera vista

Al lanzar la aplicación se abre la ventana principal del programa, que será similar a la mostrada en la figura 4.1, dependiendo del tema que tenga su escritorio configurado:

Se puede observar que está dividida en el panel de instrucciones (1), un panel de memoria (2), panel de salida (3), panel de salida de errores (4), barra de menú, barra de herramientas y barra de estado. Antes de cargar ningún programa los tres paneles aparecen vacíos. Cada panel tiene multitud de información disponible.

- En el panel de instrucciones:

¹No hay que olvidar que para ello necesitaremos instalar algunas librerías y programas dependientes. Ver la sección B.1.

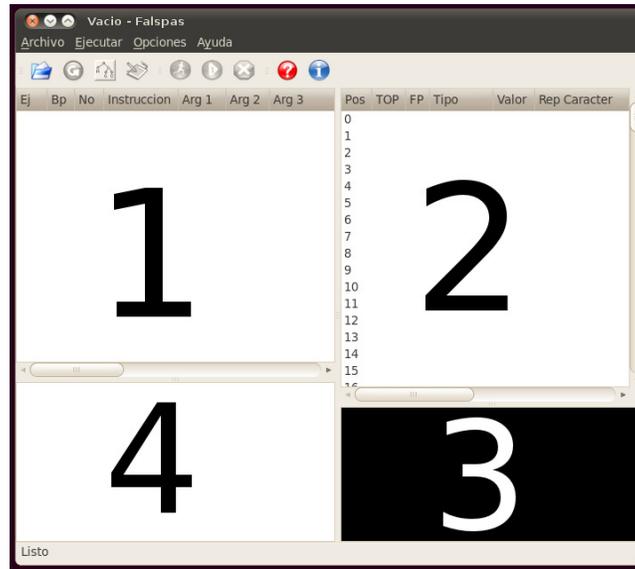


Figura 4.1: Intérprete gráfico descargado.

- En la columna Ej se marcará con una flecha la próxima instrucción a ser ejecutada.
 - En la columna Bp se indicará mediante un círculo rojo las instrucciones con un punto de ruptura asignado.
 - La columna No indicará el número de instrucción, empezando por cero.
 - La columna Instrucción mostrará el mnemotécnico de la instrucción.
 - En las tres columnas restantes se mostrarán cada uno de los argumentos de la instrucción.
 - Al pasar el puntero sobre un argumento de tipo posición, se indicará la dirección de memoria real.
- En el panel de memoria:
 - En la columna Pos se muestra la posición que ocupa la celda en la memoria. El intérprete gráfico por defecto está implementado con mil posiciones de memoria en la que se puede almacenar tanto un entero como un real.
 - La columna TOP será marcada con un círculo verde en la posición de memoria apuntada por el puntero a cima.
 - En la columna FP veremos un círculo azul en la posición de memoria apuntada por el *frame pointer*.

- La columna **Tipo** indica el tipo de valor almacenado en la posición de memoria correspondiente. Puede ser:
 - Para valores reales se mostrará **R**.
 - Para valores enteros se mostrará **E**.
 - Si el valor no ha sido inicializado no se mostrará texto. En ese caso no hay que suponer que el valor de la memoria en dicha posición sea cero.
 - Para el valor de retorno almacenado tras una instrucción **CALL**, aparecerá **DIR**, y entre paréntesis el número de la instrucción **CALL** que ha dejado este valor.
 - Para el valor de un **frame pointer** apilado por la instrucción **PUSHFP**, aparecerá **FP** seguido del número de la instrucción que ha dejado el valor, entre paréntesis.
 - La columna **Valor** muestra el número, mostrando el signo en caso de ser un número negativo y evitando la notación científica siempre que el número pueda representarse con claridad en el espacio disponible.
 - Por último la columna **Rep Caracter** mostrará el carácter que en representación *ASCII* mostraría el número, para los enteros positivos entre 32 y 126.
-
- En el panel de salida aparecerá todo lo que nuestro programa escriba con las instrucciones **EWRITE**, **RWRITE** y **CWRITE**, dando la apariencia de ser un antiguo monitor de fósforo verde.
 - En el panel de salida de errores aparecerá cualquier error en la carga de programas, además de cualquier *warning* o excepción.

4.4. Carga de un programa

Para cargar un programa fuente hay que realizar una de las tres acciones siguientes:

1. Usando la ruta de menús **Fichero**▷**Abrir**.
2. Pulsar el botón **abrir**, indicado con la imagen de una carpeta abierta. 
3. Con la combinación de teclas **Ctrl+A**.

Tras lo cual se muestra un diálogo pidiendo un fichero con extensión *.c3d* en el directorio actual. Si nuestro fichero tiene otra extensión debemos cambiar el filtro de **Mostrar ficheros *.c3d** a **Mostrar todos los ficheros *.***. Además podemos navegar por todo el espacio de directorios hasta encontrar el fichero necesario.

Tras elegirlo, el intérprete gráfico realiza un barrido del código para verificar que no existan fallos léxicos, sintácticos o semánticos detectables en tiempo de carga (ver sección 4.4.2 para profundizar en los errores detectados). Solo si pasa

la prueba se modificará el estado de la aplicación (véase página 27), es decir, no se modificará la memoria ni el programa que esté depurándose en ese momento. En caso contrario la memoria se vaciará, el *Frame Pointer* y el puntero a cima pasarán a apuntar a cero, los posibles breakpoints se borrarán, se cargará el programa en el panel de instrucciones y la instrucción disponible a ser lanzada será la número cero. Si todo ha ido bien se mostrará un mensaje de éxito en la barra de estado. En caso contrario en el panel de salida de errores se mostrará una lista con los errores detectados y el número de línea donde se encuentran.

4.4.1. Recargar un programa

El requisito para poder recargar un programa es haber cargado un programa correcto con anterioridad. Así podemos cargar varias veces un mismo programa sin tener que hacer uso del diálogo de selección de ficheros. Esta opción es útil cuando compilamos varias veces un programa, ya sea al comportarse éste de un modo erróneo o al añadirle nuevas funcionalidades. Se puede llamar con alguna de las siguientes acciones:

1. Usando la ruta de menús Fichero▷Recargar.
2. Pulsar el botón recargar, indicado con la imagen de una flecha circular.
 
3. Con la combinación de teclas Ctrl+R.

Las consecuencias de recargar un programa son las mismas que abrirlo. Se borrará la memoria y se inicializará el puntero a cima, el *frame pointer* y el puntero a la instrucción próxima a ejecutarse. También se borrarán los posibles breakpoints activos.

4.4.2. Posibles errores al cargar un programa

Puede ocurrir que el fichero fuente de código de tres direcciones esté mal generado. Ocurrirá especialmente si abrimos el fichero con un editor y modificamos las líneas sin meditar las consecuencias. En ese caso la carga del fichero se interrumpirá y una ventana nos avisará del problema. Los posibles errores detectados, junto a un ejemplo y la detección son los siguientes:

- Error léxico:


```
4 EADDI p:(1,-2), i:5, p:(1,1)
```

 No existe ninguna instrucción con ese mnemotécnico. El intérprete gráfico mostrará un error léxico en la línea 4.


```
4 RMULT p:(-1,-2), r:5.6, p:(1,1)
```

 El primer parámetro es inválido dado que el primer valor de una posición de memoria debe ser obligatoriamente un cero o un uno.
- Error sintáctico:


```
6 EMULT i:8 i:5 p:(1,1)
```

Las instrucciones no están separadas por comas. El intérprete gráfico mostrará un error sintáctico en la línea 6.

- Errores semánticos:
 - 8 `RMULT r:3.2, e:8, p:(0,0)`
El segundo parámetro es incompatible con la instrucción, debiendo ser un real o una posición de memoria (que deberá apuntar a un real). El intérprete gráfico mostrará un error semántico en la línea 8.
 - 12 `EREAD , , p:(1, -1)`
14 `EWRITE , , p:(1, -1)`
No deben faltar instrucciones. Los números de línea deben ser correlativos, sin saltos y empezar desde la instrucción número 0. El intérprete gráfico mostrará un error semántico en la línea 13.
 - 18 `GOTOS , , e:115 #Nuestro programa tiene 60 líneas`
No debe haber etiquetas que apuntes a instrucciones que no existan. El intérprete gráfico marcará como errónea la instrucción, avisando de un error en la línea 18 y no cargará el programa.
 - 153 `PUSHFP , ,`
Por definición solo se permiten programas de 150 líneas como máximo. El intérprete gráfico marcará como erróneo cualquier programa que lo supere, avisando de un error en la línea 150.

4.5. Cambiar el ancho de los paneles y las columnas de datos

En un principio el panel de instrucciones ocupa un cuarto de la ventana del intérprete gráfico, al igual que el panel de la memoria, de la salida de datos y de la salida de errores. Para cambiar las proporciones hay que situar el puntero del ratón encima de las líneas que lo dividen y pulsar sobre ellas. Moviendo el ratón se conseguirá ensanchar o estrechar el panel.

Las columnas que muestran las instrucciones y la memoria se fijan a una anchura óptima al cargar cada programa. Sin embargo se puede modificar su anchura pulsando y arrastrando entre dos de las etiquetas que tienen las columnas en su parte superior. Haciendo doble clic se dejará automáticamente una anchura óptima, de modo que se verá todo el texto disponible sin dejar más espacio vacío del necesario.

4.6. Ejecutar paso a paso un programa

Tras la carga correcta de un programa, podemos ejecutarlo paso a paso. Este sistema tiene la ventaja de poder ver de forma continua los cambios que provocan las instrucciones en la memoria, los saltos que toma el camino de ejecución de nuestro programa y poder estar más atento a los mensajes de

Pos	TOP	FP	Tipo	Valor	Rep	Caracter
0			DIR (1):	2		
1			FP (24):	0		
2		→	E:	70	F	
3			E:	1		
4						
5	→					
6						
7						
8						

Figura 4.2: Memoria con la tercera posición de memoria sombreada.

error en tiempo de ejecución que pueda dar nuestro código. Para ejecutar una instrucción deberemos usar uno de los siguientes métodos:

1. Usando la ruta de menú Ejecutar > Ejecutar la siguiente instrucción.
2. Pulsar el botón ejecutar la siguiente instrucción, indicado con la imagen verde de un triángulo con el dígito número uno inscrito. 
3. Con la combinación de teclas Ctrl+J.

4.6.1. Detalles en la ejecución de un programa

En un programa que está siendo depurado hay algunas marcas útiles para su depuración:

- La instrucción que será ejecutada en la siguiente iteración está marcada por una flecha². 
- La última posición de memoria donde se ha escrito un resultado aparecerá sombreada, como la tercera posición de memoria en en la figura 4.2. Si la última instrucción no ha escrito nada en memoria (por ejemplo, una instrucción de salto o entrada y salida de datos) la indicación desaparecerá hasta que no se vuelva a escribir en memoria.
- La posición del puntero a cima estará en cada momento señalada por una flecha azul en la columna TOP. Se debe tener en cuenta que la posición de memoria señalada es la primera libre, no la última escrita. 
- La posición del frame pointer está indicado en cada momento con una flecha verde en la columna FP. 
- Si pasamos el puntero sobre el panel de instrucciones y lo detenemos encima de un parámetro posición de memoria, aparecerá un pequeño cuadro

²El dibujo de la flecha puede ser ligeramente distinto dependiendo de la versión de la librería de ejecución Qt4 instalada en su sistema.

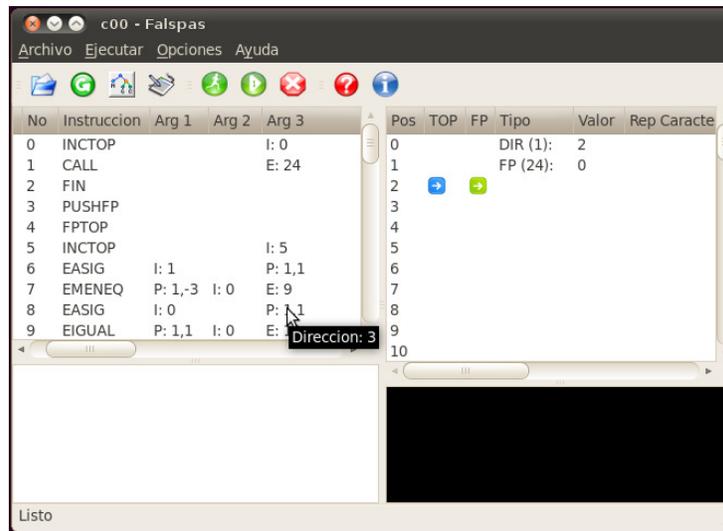


Figura 4.3: Pequeño cuadro de ayuda que indica la dirección física de memoria.

de ayuda indicando la dirección física de memoria en vez de la lógica. Evidentemente la información mostrada cambia cada vez varía la dirección del frame pointer. Un ejemplo se muestra en la figura 4.3. Como puede comprobarse la dirección física mostrada es la posición tercera, resultado de sumar al *frame pointer* el desplazamiento (2+1).

4.6.2. Uso estricto de la memoria

Un programa bien escrito en código de tres direcciones debería cumplir unas reglas semánticas en cuanto al uso de la memoria:

1. No debería poderse leer ni escribir en posiciones de memoria iguales o superiores al puntero a cima.
2. El entero que apila en memoria una instrucción CALL solo debería leerse (y a la vez desapilarse) por una instrucción RET.
3. Las antiguas posiciones del frame pointer apiladas en memoria por la instrucción PUSHFP solo deberían poder leerse (y desapilarse) por una instrucción FPPPOP.

Así, al activar el uso estricto de la memoria

- Usando la opción de menú Opciones▷Memoria estricta.
- O bien la combinación de teclas Ctrl+I.

Se comprobará que se cumplan estas tres reglas y en caso contrario se mostrarán las advertencias necesarias en el panel de salida de errores.

4.6.3. Posibles errores en tiempo de ejecución

Todas las instrucciones pueden dar mensajes de advertencia o excepciones en tiempo de ejecución cuando se den condiciones que compromentan la integridad semántica del programa. Estas advertencias se dan:

1. Cuando una instrucción busca un dato en memoria y éste no es del tipo correcto. Por compromiso se leerá un dato truncado, pero en realidad nuestro compilador no está bien escrito.
2. Cuando una instrucción busca un dato en memoria y en la posición no se ha escrito aún. Por compromiso se leerá un cero, pero en realidad nuestro compilador no está bien escrito. Todas las posiciones de memoria deberían inicializarse antes de leerlas.
3. El segundo argumento de una instrucción de división entera, real o la instrucción de resto es un cero. Por compromiso el resultado de la operación será de cero, pero es evidente que los datos que le hemos proporcionado a nuestro programa no son correctos. Para evitarlo deberíamos comprobar en nuestro programa fuente que no se de esta condición antes de efectuar la división.
4. Un programa acaba en una instrucción distinta a la instrucción FIN. Si la última instrucción de un programa no es de salto o una instrucción FIN, al actualizar el contador de programa se intentará leer una instrucción inexistente, finalizando el programa. Aunque puede parecer que el programa funciona, en realidad nuestro compilador no lo ha generado correctamente. Esta excepción es irrecuperable, debiendo reiniciar el estado del intérprete gráfico (Ver sección 4.9).
5. Ejecutar la instrucción EPOP cuando el puntero a cima esté en su posición inicial. Por compromiso devolveremos un cero y el puntero a cima no se modificará.
6. Ejecutar la instrucción DECTOP con un argumento superior al valor actual del puntero a cima. Esto quiere decir que por ejemplo, si el puntero a cima apunta a la posición de memoria 5, ejecutar DECTOP con un argumento mayor que 5 fallará. Por compromiso dejaremos el puntero a pila en la posición inicial.
7. Análogamente, cualquier instrucción que modifique o lea valores por encima del puntero a cima.
8. Ejecutar la instrucción RET con el puntero a cima apuntando a cero, a un dato no inicializado o a un real.
9. Cuando una instrucción escribe a posiciones de memoria que no existen. En este caso el dato no se escribirá.

10. Aunque sean enteros, el intérprete analiza que los números que escribe la instrucción CALL en la memoria solo puedan ser usados por la instrucción RET y por ninguna otra.
11. Aunque sean enteros, el intérprete también hace lo mismo con las cifras escritas por la instrucción FPTOP y TOPFP.

Sin embargo hemos de remarcar que no se indicará ninguna condición de *overflow* o *underflow* en la ejecución de los cálculos.

4.7. Ejecutar el programa de forma continua

Aunque es conveniente ejecutar antes el programa paso a paso para cerciorarse de que funciona correctamente, también podemos ejecutarlo de forma continua. Un programa así ejecutado solo parará ante una instrucción FIN, ante la excepción señalada con el número 4 de la sección anterior o ante una instrucción marcada como breakpoint. Hay que tener en cuenta que un programa ejecutado de forma continua no modificará la representación del panel de memoria ni la indicación de instrucción próxima a ejecutarse hasta que la ejecución finalice. Para ejecutar el programa de este modo usaremos cualquiera de estos métodos:

1. Navegando por el menú, haremos clic en Ejecutar ▷ Ejecutar.
2. Pulsar el botón ejecutar, indicado con la imagen de una persona corriendo dentro de un círculo verde. 
3. Con la combinación de teclas Ctrl+E.

Añadir que se pueden mezclar los dos tipos de ejecución (continua y paso a paso) en una misma traza sin ningún problema. También debemos tener la precaución que el programa no entre un estado de bucle infinito. En ese caso deberemos detener el proceso de nuestro intérprete gráfico usando una instrucción `kill -9` y abrirla de nuevo.

4.8. Añadir y quitar breakpoints

Los breakpoints son muy útiles para, por ejemplo, ejecutar de forma continua un bucle y parar la ejecución a la salida, pudiendo ver entonces el estado de la memoria. Evidentemente los breakpoints no tienen sentido cuando se ejecuta el programa paso a paso ya que el intérprete gráfico detiene su ejecución del código tras cada instrucción automáticamente. Para insertar un breakpoint basta con hacer doble clic en la instrucción que queremos que el intérprete pare *antes* de ejecutarla. Para quitar el breakpoint haremos también doble clic sobre la instrucción. Las instrucciones marcadas con breakpoints se distinguen por un pequeño símbolo en la columna Bp (breakpoint) similar a una señal de tráfico.



Para quitar todos los breakpoints de un programa de golpe –en el caso de haber alguno– se puede ejecutar la opción borrar todos los breakpoints navegando por el menú Opciones▷Borrar los breakpoints, o bien usar la combinación de teclas Ctrl+B. Tras su uso en la barra de estado se indicarán cuales son las líneas donde se ha añadido o quitado el breakpoint, o cuántos se han quitado de golpe.

4.9. Detener y reiniciar el estado

Si hemos introducido datos erróneos o por algún motivo queremos reiniciar el estado de nuestro intérprete gráfico podemos hacerlo con alguno de los siguientes métodos:

1. Desde el menú, navegaremos por las opciones Ejecutar▷Detener la ejecución y reiniciar el intérprete gráfico.
2. Pulsar el botón detener, indicado con la imagen de un hexágono rojo con un aspa. 
3. Con la combinación de teclas Ctrl+D.

Esta orden borra e inicializa la memoria, pone a cero el puntero a cima y el *frame pointer* y marca la instrucción cero como la próxima a ejecutarse. Sin embargo no modifica los breakpoints ni borra los datos de salida de anteriores ejecuciones del panel.

4.10. Desplazar automáticamente la representación de la memoria y las instrucciones

Si la ventana en la que ejecutamos el intérprete gráfico es pequeña o nuestro programa usa muchas instrucciones o memoria llegará un momento en que se escriba en posiciones de memoria que queden fuera del área de memoria representada en el panel o se ejecutarán instrucciones posteriores a las representadas en el panel de instrucciones. En ese caso tendremos que hacer scroll con las barras desplazadoras de los paneles. Como hacer eso durante la ejecución paso a paso de un programa es realmente incómodo, se puede usar una opción que garantiza que siempre estará visible la próxima instrucción a ejecutarse y el último dato que se ha escrito en memoria. Para activar la opción hay que hacer clic en la opción de menú Opciones▷Desplazar la memoria o bien usar la combinación de teclas Ctrl+Z.

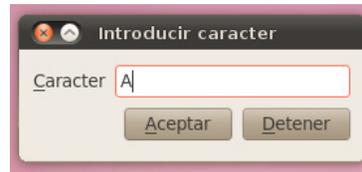


Figura 4.4: Petición de un dato por parte del intérprete gráfico

4.11. Mostrar la ayuda y la autoría del intérprete gráfico

El intérprete gráfico tiene tanto un pequeño cuadro de ayuda indicando el significado de las distintas instrucciones como un recuadro mostrando la autoría y el objeto de la aplicación. Podemos acceder a la ayuda del programa mediante cualquiera de los siguientes métodos:

1. Usando el menú, con la opción Ayuda▷Ayuda.
2. Pulsar el botón de ayuda, indicado con la imagen de un interrogante en un círculo rojo. 
3. Con la combinación de teclas Ctrl+Y.

Y para acceder a la autoría y descripción del programa podremos:

1. Navegando por el menú, usando la opción Ayuda▷Acerca de.
2. Pulsar el botón Acerca de, indicado con la imagen de una letra i en un círculo azul. 
3. Con la combinación de teclas Ctrl+C.

4.12. Peticiones de datos por parte del intérprete gráfico

Cuando el intérprete gráfico ejecute la instrucción EREAD, RREAD o CREAD mostrará un pequeño dialogo que consta de una caja de introducción de caracteres, un botón aceptar y otro botón detener como la mostrada en la figura 4.4.

La caja tiene un validador que hace imposible introducir una secuencia de caracteres que no corresponda al tipo de datos necesitado.

- En caso de un entero sólo acepta cadenas que cumplan la expresión regular `"-"?[0-9]+`
- El validador de un real acepta `"-"?[0-9]+". "[0-9]+`

- Y finalmente la petición de caracter acepta `[\x20-\x7E]{1}`, es decir, un y solo un caracter imprimible, lo que es equivalente a que su código ASCII esté comprendido entre el 32 y el 126.

Cuando se haya escrito un dato válido, el botón **Aceptar** del diálogo pasará a estar disponible. Pulsándolo, el intérprete gráfico recibirá el dato y la asignará a la posición de memoria indicada en el tercer argumento de la instrucción. Si en cambio se pulsa el botón **Detener** el intérprete no guardará dato alguno y el programa se detendrá si se estaba ejecutando en modo continuo.

4.13. Ayudas al desarrollo

He querido que mi herramienta tuviese opciones para facilitar la tarea de crear un compilador o programa adecuados. Para ello he añadido dos opciones que pueden resultar muy útiles:

- La opción de compilar puede iniciarse con alguno de estos dos métodos:
 - Con la opción de menú **Archivo**▷**Compilar**.
 - Pulsando el botón de compilar, donde se encuentra dibujado un árbol sintáctico. 

Esta herramienta compila de nuevo el programa que tengamos cargado en memoria y a continuación lo carga. Funciona de la siguiente manera:

- A través de la variable interna del intérprete `fichero_actual` (ver la sección 3.4.3 para más información) se obtiene la ruta absoluta y el nombre de fichero.
- Se elimina la posible extensión del fichero (casi con toda seguridad `.c3d`) y se cambia por la extensión `.c`.³
- Sin ninguna intervención del usuario el intérprete llama al compilador, recompila el archivo `.c` y recarga el archivo `.c3d` suponiendo que no contenga errores.

Tenemos que advertir que para usar esta opción el archivo de código fuente ha de tener la extensión `.c` y debe encontrarse en el mismo directorio que el archivo `.c3d`. De lo contrario no funcionará.

- La opción para editar puede iniciarse desde el menú o desde el botón de herramientas:
 - Mediante la orden del menú **Archivo**▷**Editar**.
 - Pulsando el botón de editar, donde se muestra el icono de una tableta de dibujo. 

³Ver la información de la clase `QFileInfo` en la documentación oficial de Qt4 (Qt 4 Assistant) para obtener más información.



Figura 4.5: Ventana de configuración.

Usando este método se abrirá en un editor de texto el fichero `.c3d` que tengamos cargado actualmente. Podremos modificarlo y tras guardarlo y cerrar el editor el intérprete volverá a cargar el programa modificado en caso de no contener errores. Esta opción deberá usarse con el único propósito de hacer pruebas en el caso de que estemos programando un compilador.

- Tanto para usar la opción de compilar como la de editar debemos antes establecer cual es el editor y compilador que queremos usar. Para ello tenemos que especificarlo haciendo uso de la ventana de configuración mostrada en la figura 4.5. Para invocar la ventana de configuración debemos introducir la orden de menú `Opciones > Configuración` que está marcado con el icono de una libreta y un engranaje. 📝⚙️

En esta ventana podemos hacer uso de las cajas de texto o de los botones que despliegan diálogos para buscar archivos. Finalmente al pulsar el botón aceptar se comprobará que las rutas sean correctas y los ficheros tengan permiso de ejecución. Pasado el test se guardarán las opciones. Para la elección de editor recomiendo el uso de `gedit`, en la ruta `/usr/bin/gedit`, mientras que la elección del compilador dependerá de cada usuario.

4.14. Persistencia de la configuración y opciones por defecto.

La primera vez que se ejecute el intérprete gráfico en nuestro equipo cargará las órdenes por defecto. Estas opciones incluyen el tamaño y posición de la ventana, el tamaño de los cuatro paneles con los que consta la ventana principal, la anchura de las columnas de los paneles de memoria e instrucciones, el uso estricto de la memoria, el desplazamiento automático de los paneles de instrucciones y memoria y finalmente la ruta del compilador y el editor, que por defecto son `./cmc` y `/usr/bin/gedit` (No es necesario escribir la ruta completa siempre que los ejecutables estén en el `PATH`). Al cerrar el intérprete si hemos cambiado alguno de estos aspectos durante su uso se guardará su nueva configuración para que la próxima vez que se ejecute se cargue de nuevo tal y como lo cerramos. Para ello el intérprete usa la herramienta de `QSettings` que es independiente del

entorno de ventanas que estemos ejecutando⁴.

Pero evidentemente hemos de señalar que si se está ejecutando el intérprete desde una distribución de GNU/Linux *live* desde un CD o similares, dichos cambios no se mostrarán cuando se reinicie el ordenador.

⁴Para más información sobre QSettings, lugar donde guarda los archivos de configuración y demás cuestiones mirar la sección QSettings en Qt 4 Assistant.

Apéndice A

Instrucciones de código intermedio

A continuación se mostrarán las instrucciones de código de tres direcciones que nuestro intérprete gráfico puede ejecutar. Se parte del hecho que todas las instrucciones tienen tres argumentos, aunque algunos de ellos, incluso los tres, pueden ser nulos. El esquema de las instrucciones sería:

```
Num_línea Instrucción Arg_1, Arg_2, Arg_3
```

A.1. Instrucciones aritméticas

- ESUM Suma dos enteros y deja el resultado en una posición de memoria.
- Argumento 1 y 2: Entero o puntero a una posición de memoria que contenga un entero.
 - Argumento 3: Puntero a una posición de memoria donde dejará el resultado.
- RSUM Suma dos reales y deja el resultado en una posición de memoria.
- Argumento 1 y 2: Real o puntero a una posición de memoria que contenga un real.
 - Argumento 3: Puntero a una posición de memoria donde dejará el resultado.
- EDIF Resta dos enteros y deja el resultado en una posición de memoria.
- Argumento 1 y 2: Entero o puntero a una posición de memoria que contenga un entero.
 - Argumento 3: Puntero a una posición de memoria donde dejará el resultado.

- RDIF Resta dos reales y deja el resultado en una posición de memoria.
- Argumento 1 y 2: Real o puntero a una posición de memoria que contenga un real.
 - Argumento 3: Puntero a una posición de memoria donde dejará el resultado.
- EMULT Multiplica dos enteros y deja el resultado en una posición de memoria.
- Argumento 1 y 2: Entero o puntero a una posición de memoria que contenga un entero.
 - Argumento 3: Puntero a una posición de memoria donde dejará el resultado.
- RMULT Multiplica dos reales y deja el resultado en una posición de memoria.
- Argumento 1 y 2: Real o puntero a una posición de memoria que contenga un real.
 - Argumento 3: Puntero a una posición de memoria donde dejará el resultado.
- EDIVI Divide dos enteros y deja el resultado de la division entera en una posición de memoria. Para conocer el resto se usará la instrucción RESTO
- Argumento 1 y 2: Entero o puntero a una posición de memoria que contenga un entero.
 - Argumento 3: Puntero a una posición de memoria donde dejará el resultado.
- RDIVI Divide dos reales y deja el resultado en una posición de memoria.
- Argumento 1 y 2: Real o puntero a una posición de memoria que contenga un real.
 - Argumento 3: Puntero a una posición de memoria donde dejará el resultado.
- RESTO Calcula el resto de la division entre dos enteros dejando el resultado en una posición de memoria.
- Argumento 1 y 2: Entero o puntero a una posición de memoria que contenga un entero.
 - Argumento 3: Puntero a una posición de memoria donde dejará el resultado.

- CONV Toma un número entero y lo deja en la memoria, esta vez asignándole el tipo de dato real.
- Argumento 1: Entero o puntero a una posición de memoria que contenga un entero.
 - Argumento 2: Argumento nulo.
 - Argumento 3: Puntero a una posición de memoria donde dejará el resultado.
- TRUNC Toma un número real y lo escribe en la memoria asignándole el tipo de dato entero, ignorando su parte decimal.
- Argumento 1: Real o puntero a una posición de memoria que contenga un real.
 - Argumento 2: Argumento nulo.
 - Argumento 3: Puntero a una posición de memoria donde dejará el resultado.
- ESIG Toma un número entero y le cambia el signo. Es la operación análoga a multiplicar el número por -1 usando la operación EMULT, pero lo efectúa de forma más rápida.
- Argumento 1: Entero o puntero a una posición de memoria que contenga un entero.
 - Argumento 2: Argumento nulo.
 - Argumento 3: Puntero a una posición de memoria donde dejará el resultado.
- RSIG Toma un número real y le cambia el signo. Es la operación análoga a multiplicar el número por -1.0 usando la operación RMULT, pero lo efectúa de forma más rápida.
- Argumento 1: Real o puntero a una posición de memoria que contenga un real.
 - Argumento 2: Argumento nulo.
 - Argumento 3: Puntero a una posición de memoria donde dejará el resultado.
- EASIG Toma un entero y lo asigna a una posición de memoria. Es un instrucción análoga a usar la instrucción ESUM para sumar un número con 0, o con la instrucción EDIF, pero lo realiza de forma más rápida y es semánticamente más claro.
- Argumento 1: Entero o puntero a una posición de memoria que contenga un entero.

- Argumento 2: Argumento nulo.
 - Argumento 3: Puntero a una posición de memoria donde dejará el resultado.
- RASIG Toma un real y lo asigna a una posición de memoria. Es un instrucción análoga a usar la instrucción RSUM para sumar un número con 0.0, o con la instrucción RDIF, pero lo realiza de forma más rápida y es semánticamente más claro.
- Argumento 1: Real o puntero a una posición de memoria que contenga un real.
 - Argumento 2: Argumento nulo.
 - Argumento 3: Puntero a una posición de memoria donde dejará el resultado.

A.2. Instrucciones que modifican el flujo del programa

- GOTOS Efectúa un salto incondicional del flujo del programa. Modifica el contador de programa para que la siguiente instrucción en ejecutarse sea la indicada por el tercer argumento, de tipo etiqueta.
- Argumento 1 y 2: Argumento nulo.
 - Argumento 3: Etiqueta al número de instrucción que saltará.
- EIGUAL Efectúa el salto a la etiqueta indicada en el tercer argumento a condición de que los dos enteros que tome en los argumentos uno y dos sean iguales.
- Argumento 1 y 2: Entero o puntero a una posición de memoria que contenga un entero.
 - Argumento 3: Etiqueta al número de instrucción que saltará dado el caso.
- RIGUAL Efectúa el salto a la etiqueta indicada en el tercer argumento a condición de que los dos reales que tome en los argumentos uno y dos sean iguales.
- Argumento 1 y 2: Real o puntero a una posición de memoria que contenga un real.
 - Argumento 3: Etiqueta al número de instrucción que saltará dado el caso.

- EDIST Efectúa el salto a la etiqueta indicada en el tercer argumento a condición de que los dos enteros que tome en los argumentos uno y dos sean distintos.
- Argumento 1 y 2: Entero o puntero a una posición de memoria que contenga un entero.
 - Argumento 3: Etiqueta al número de instrucción que saltará dado el caso.
- RDIST Efectúa el salto a la etiqueta indicada en el tercer argumento a condición de que los dos reales que tome en los argumentos uno y dos sean distintos.
- Argumento 1 y 2: Real o puntero a una posición de memoria que contenga un real.
 - Argumento 3: Etiqueta al número de instrucción que saltará dado el caso.
- EMEN Efectúa el salto a la etiqueta indicada en el tercer argumento a condición de que el entero del primer argumento sea menor que el segundo (Salta si $\text{Arg1} < \text{Arg2}$).
- Argumento 1 y 2: Entero o puntero a una posición de memoria que contenga un entero.
 - Argumento 3: Etiqueta al número de instrucción que saltará dado el caso.
- RMEN Efectúa el salto a la etiqueta indicada en el tercer argumento a condición de que el real del primer argumento sea menor que el segundo (Salta si $\text{Arg1} < \text{Arg2}$).
- Argumento 1 y 2: Real o puntero a una posición de memoria que contenga un real.
 - Argumento 3: Etiqueta al número de instrucción que saltará dado el caso.
- EMAY Efectúa el salto a la etiqueta indicada en el tercer argumento a condición de que el entero del primer argumento sea mayor que el segundo (Salta si $\text{Arg1} > \text{Arg2}$).
- Argumento 1 y 2: Entero o puntero a una posición de memoria que contenga un entero.
 - Argumento 3: Etiqueta al número de instrucción que saltará dado el caso.

- RMAY** Efectúa el salto a la etiqueta indicada en el tercer argumento a condición de que el real del primer argumento sea mayor que el segundo (Salta si $\text{Arg1} > \text{Arg2}$).
- Argumento 1 y 2: Real o puntero a una posición de memoria que contenga un real.
 - Argumento 3: Etiqueta al número de instrucción que saltará dado el caso.
- EMENEQ** Efectúa el salto a la etiqueta indicada en el tercer argumento a condición de que el entero del primer argumento sea menor o igual que el segundo (Salta si $\text{Arg1} \leq \text{Arg2}$).
- Argumento 1 y 2: Entero o puntero a una posición de memoria que contenga un entero.
 - Argumento 3: Etiqueta al número de instrucción que saltará dado el caso.
- RMENEQ** Efectúa el salto a la etiqueta indicada en el tercer argumento a condición de que el real del primer argumento sea menor o igual que el segundo (Salta si $\text{Arg1} \leq \text{Arg2}$).
- Argumento 1 y 2: Real o puntero a una posición de memoria que contenga un real.
 - Argumento 3: Etiqueta al número de instrucción que saltará dado el caso.
- EMAYEQ** Efectúa el salto a la etiqueta indicada en el tercer argumento a condición de que el entero del primer argumento sea mayor o igual que el segundo (Salta si $\text{Arg1} \geq \text{Arg2}$).
- Argumento 1 y 2: Entero o puntero a una posición de memoria que contenga un entero.
 - Argumento 3: Etiqueta al número de instrucción que saltará dado el caso.
- RMAYEQ** Efectúa el salto a la etiqueta indicada en el tercer argumento a condición de que el real del primer argumento sea mayor o igual que el segundo (Salta si $\text{Arg1} \geq \text{Arg2}$).
- Argumento 1 y 2: Real o puntero a una posición de memoria que contenga un real.
 - Argumento 3: Etiqueta al número de instrucción que saltará dado el caso.

A.3. Instrucciones que acceden a vectores

EAV El primer argumento indica la posición base del vector, el segundo argumento el desplazamiento o dirección relativa del vector. La acción que realiza es asignar en la posición de memoria indicada por el tercer argumento el elemento entero que se referencia con el vector (Es decir, $\text{Arg3} \leftarrow \text{Arg1}[\text{Arg2}]$).

- Argumento 1: Puntero a una posición de memoria en la que empiece un vector de enteros. No necesariamente debe estar inicializado, pero en caso contrario debe ser un entero.
- Argumento 2: Entero o posición de memoria que contenga un entero. Su valor no debería ser mayor que la longitud del vector.
- Argumento 3: Posición de memoria donde se escribirá el dato entero contenido en el vector.

RAV El primer argumento indica la posición base del vector, el segundo argumento el desplazamiento o dirección relativa del vector. La acción que realiza es asignar en la posición de memoria indicada por el tercer argumento el elemento real que se referencia con el vector (Es decir, $\text{Arg3} \leftarrow \text{Arg1}[\text{Arg2}]$).

- Argumento 1: Posición de memoria en la que empieza un vector de reales. No necesariamente debe estar inicializado, pero en caso contrario debería ser un real.
- Argumento 2: Entero o posición de memoria que contenga un entero. Su valor no debería ser mayor que la longitud del vector.
- Argumento 3: Posición de memoria donde se escribirá el dato real contenido en el vector.

EVA El primer argumento indica la posición base del vector, el segundo argumento el desplazamiento o dirección relativa del vector. La acción que realiza es asignar en la posición de memoria indicada por el vector y su desplazamiento el entero que indica el tercer argumento (Es decir, $\text{Arg1}[\text{Arg2}] \leftarrow \text{Arg3}$).

- Argumento 1: Posición de memoria en la que empieza un vector de enteros. No necesariamente debe estar inicializado, pero en caso contrario debería ser un entero.
- Argumento 2: Entero o posición de memoria que contenga un entero. Su valor no debería ser mayor que la longitud del vector.
- Argumento 3: Entero o posición de memoria donde hay un entero que se escribirá en el vector.

RVA El primer argumento indica la posición base del vector, el segundo argumento el desplazamiento o dirección relativa del vector. La acción que realiza es asignar en la posición de memoria indicada por el vector y su desplazamiento el real que indica el tercer argumento (Es decir, $\text{Arg1}[\text{Arg2}] \leftarrow \text{Arg3}$).

- Argumento 1: Posición de memoria en la que empieza un vector de reales. No necesariamente debe estar inicializado, pero en caso contrario debería ser un real.
- Argumento 2: Entero o posición de memoria que contenga un entero. Su valor no debería ser mayor que la longitud del vector.
- Argumento 3: Real o posición de memoria donde hay un real que se escribirá en el vector.

A.4. Instrucciones de entrada y salida de datos

ERead El intérprete gráfico muestra un diálogo en el que se pide un entero y tras validarlo se guarda en la posición de memoria apuntada por el tercer argumento

- Argumento 1 y 2: Argumento nulo.
- Argumento 3: Posición de memoria donde se escribirá el número entero escrito por el usuario.

RRead El intérprete gráfico muestra un diálogo en el que se pide un real y tras validarlo se guarda en la posición de memoria apuntada por el tercer argumento

- Argumento 1 y 2: Argumento nulo.
- Argumento 3: Posición de memoria donde se escribirá el número real escrito por el usuario.

CRead El intérprete gráfico muestra un diálogo en el que se pide un carácter y tras validarlo se guarda el entero que representa dicho carácter en código ASCII en la posición de memoria apuntada por el tercer argumento

- Argumento 1 y 2: Argumento nulo.
- Argumento 3: Posición de memoria donde se escribirá el número entero que equivale al carácter ASCII introducido por el usuario.

EWrite El intérprete gráfico muestra en el panel de salida un número entero definido en el tercer argumento.

- Argumento 1 y 2: Argumento nulo.
 - Argumento 3: Entero o posición de memoria donde se lee el entero a escribir en el panel de salida.
- RWRITE El intérprete gráfico muestra en el panel de salida un número real definido en el tercer argumento.
- Argumento 1 y 2: Argumento nulo.
 - Argumento 3: Real o posición de memoria donde se lee el real a escribir en el panel de salida.
- CWRITE El intérprete gráfico muestra en el panel de salida el caracter cuyo código ascii, guardado en un número entero está definido en el tercer argumento.
- Argumento 1 y 2: Argumento nulo.
 - Argumento 3: Entero o posición de memoria donde se lee el entero a escribir en el panel de salida. El número ha de estar comprendido entre 32 y 126.

A.5. Instrucciones que modifican la pila, el *frame pointer* y el estado del programa

- FIN El intérprete gráfico muestra en el panel de salida un mensaje indicando que el programa ha finalizado correctamente y deja su estado en modo *Ejecución finalizada* (Ver figura 2.2 en página 28).
- Argumento 1, 2 y 3: Argumento nulo.
- RET La instrucción semánticamente implica el retorno de una llamada a función. Internamente lo que realiza es desapilar un entero de la memoria y poner el dato en el contador de programa. Es decir:
Puntero a cima ← Puntero a cima - 1
Contador de programa ← Memoria[Puntero a cima]
- Argumento 1, 2 y 3: Argumento nulo.
- CALL La instrucción desvía el flujo del programa a la instrucción apuntada por la etiqueta del tercer argumento, pero guarda la dirección de la instrucción siguiente en la pila para poder ejecutar más tarde la instrucción RET. Detalladamente realiza:
Memoria[Puntero a cima] ← Contador de programa + 1
Puntero a cima ← Puntero a cima + 1
Contador de programa ← Arg3
- Argumento 1 y 2: Argumento nulo.

- Argumento 3: Etiqueta al número de instrucción que saltará (primera instrucción de la rutina).
- NOP La instrucción no realiza ningún trabajo. Sin embargo puede ser de ayuda al usuario para identificar las instrucciones que emite su compilador. El tercer argumento acepta un entero que sirve para distinguir una instrucción NOP de otra.
- Argumento 1 y 2: Argumento nulo.
 - Argumento 3: Número entero.
- EPUSH Apila un número entero. La instrucción realiza el siguiente trabajo:
Memoria[Puntero a cima]←Arg3
Puntero a cima←Puntero a cima + 1
- Argumento 1 y 2: Argumento nulo.
 - Argumento 3: Entero o posición de memoria donde hay un número entero.
- RPUSH Apila un número real. Su funcionamiento es análogo al de la instrucción anterior.
- Argumento 1 y 2: Argumento nulo.
 - Argumento 3: Real o posición de memoria donde hay un número real.
- EPOP Desapila un número entero y lo guarda en la posición de memoria indicada por el tercer argumento. La instrucción realiza el siguiente trabajo:
Puntero a cima←Puntero a cima - 1
Memoria[Arg3]←Memoria[Puntero a cima]
- Argumento 1 y 2: Argumento nulo.
 - Argumento 3: Posición de memoria donde se guardará el entero desapilado.
- RPOP Desapila un número real y lo guarda en la posición de memoria indicada por el tercer argumento. La instrucción realiza un trabajo análogo al de la instrucción anterior.
- Argumento 1 y 2: Argumento nulo.
 - Argumento 3: Posición de memoria donde se guardará el número real desapilado.

- PUSHFP** Apila el frame pointer en la pila. La instrucción realiza el siguiente trabajo:
Memoria[Puntero a cima] ← Frame pointer
Puntero a cima ← Puntero a cima + 1
- Argumento 1, 2 y 3: Argumento nulo.
- FPPOP** Desapila el frame pointer de la pila. La instrucción realiza el siguiente trabajo:
Puntero a cima ← Puntero a cima - 1
Frame pointer ← Memoria[Puntero a cima]
- Argumento 1, 2 y 3: Argumento nulo.
- FPTOP** Modifica el *frame pointer* para que pase a apuntar a la misma posición de memoria que el puntero a cima. Es decir:
Frame pointer ← Puntero a cima
- Argumento 1, 2 y 3: Argumento nulo.
- TOPFP** Modifica el puntero a cima para que pase a apuntar a la misma posición de memoria que el *frame pointer*. Es decir:
Puntero a cima ← Frame pointer
- Argumento 1, 2 y 3: Argumento nulo.
- INCTOP** Modifica el puntero a cima sumándole el número entero que le pasemos como tercer argumento. Es decir:
Puntero a cima ← Puntero a cima + Arg3
- Argumento 1 y 2: Argumento nulo.
 - Argumento 3: Entero o posición de memoria donde hay un número entero.
- DECTOP** Modifica el puntero a cima restándole el número entero que le pasemos como tercer argumento. Es decir:
Puntero a cima ← Puntero a cima - Arg3
- Argumento 1 y 2: Argumento nulo.
 - Argumento 3: Entero o posición de memoria donde hay un número entero.

Apéndice B

Software necesario en la creación del proyecto

B.1. Dependencias de compilación y ejecución

Para poder compilar el intérprete cómodamente usando el script de compilación `do` es necesario tener instalado ciertos programas y librerías:

- El generador de analizadores léxicos Flex.
- El generador de analizadores sintácticos Bison.
- El compilador GCC.
- El compilador G++.
- Las librerías de desarrollo de Qt4.

Todo ello es fácilmente instalable con tan solo una línea de comandos:

```
sudo apt-get install flex bison gcc g++ libqt4-dev
```

B.2. Software utilizado durante la realización del proyecto

El computador usado para realizar este proyecto de fin de carrera ha sido un PC con un procesador compatible i386 con un sistema operativo GNU/Linux. El software ha sido el siguiente:

- El sistema operativo ha sido la versión 9.04 de la distribución Ubuntu del sistema operativo GNU/Linux, para procesadores de 32 bits, aunque también se han hecho pruebas en los computadores de los laboratorios del DSIC, donde tienen la distribución de GNU/Linux CentOS 4.3 como sistema operativo.

APÉNDICE B. SOFTWARE NECESARIO EN LA CREACIÓN DEL PROYECTO83

- Flex, el generador libre de analizadores léxicos rápidos, en su versión 2.5.35, usado para generar el analizador léxico de los ficheros c3d.
- Bison, la alternativa libre a yacc para generar parsers de gramáticas LALR(1) en su versión 2.4.1, usado para generar el analizador sintáctico y semántico de los ficheros c3d.
- Las librerías de desarrollo Qt4 en su versión 4.6.2, adquiridas por Nokia y liberadas bajo GPL 3.0
- El generador gráfico de interfaces QtDesigner, de Nokia, en su versión 4.6.2, usado para generar la interfaz de la ventana principal del intérprete gráfico y los diálogos de petición de datos.
- La extensa documentación de las librerías Qt4 gracias al programa Qt Assistant, en su versión 4.6.2.
- QDevelop 3.2, un entorno integrado de programación para KDE que presta ayudas como la indentación automática, detección de errores o la predicción de texto.
- Gcc, el compilador libre de C, en su versión 4.4.3 para compilar el código generado por Flex y Bison.
- G++, el compilador libre de C++, en su versión 4.4.3 para compilar el código del intérprete gráfico.
- Ld, el enlazador libre del proyecto GNU, en su versión 2.20.1, para enlazar el código objeto de los analizadores sintáctico, léxico y el intérprete gráfico.
- LyX en su versión 1.6.5, herramienta de interfaz gráfica de usuario para redactar la memoria en L^AT_EX de un modo rápido y fácil.

Bibliografía

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compiladores. Principios, técnicas y herramientas. Segunda edición*. Pearson Educación, 2008.
- [2] Andrew W. Appel. *Modern compiler implementation in C*. Cambridge University Press, 1998.
- [3] Jasmin Blanchette, Mark Summerhill, and Others. *C++ GUI programming with Qt4*. Prentice Hall, 2006.
- [4] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Third edition*. Morgan Kaufmann Publishers, 2002.
- [5] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Teoría de autómatas, lenguajes y computación*. Pearson Addison Wesley, 2008.
- [6] Dean Leffingwell. *Managing software requirements, a use case approach. Second edition*. Addison-wesley, 2003.
- [7] Dennis M. Ritchie and Brian W. Kernigan. *The C programming language. Second edition*. Prentice Hall, 1988.
- [8] Cándido Zuriaga. *Proyecto fin de carrera. Facultad de Informática. Universidad Politécnica de Valencia: Máquina abstracta para la depuración y ejecución de código intermedio*, 1999.