Sistemas y Aplicaciones Informáticas

Tema 31. Utilidades para el Desarrollo y Pruebas de Programas. Compiladores. Intérpretes. Depuradores.

1. Ámbito de docencia.

- Sistemas informáticos monousuario y multiusuario (ASI 1).
- Sistemas informáticos multiusuario y en red (DAI 1).
- Sistemas operativos en entornos monousuario y multiusuario (ESI 1).

2. Utilidades para el desarrollo y pruebas de programas.

2.1.1. Entornos de programación. Características y componentes.

- Los entornos de programación son una combinación de herramientas que permiten automatizar el proceso de desarrollo y prueba de los programas. Sus características son las siguientes:
  - Facilitan las labores de programación.
  - Aportan herramientas automáticas de ayuda.
  - Son independientes respecto del entorno final de ejecución de la aplicación.
  - Facilitan la compatibilidad y la integración en familias de productos.
- Los componentes de un entorno de programación son los siguientes:
  - *Editor de código*. Es el programa utilizado para escribir el código fuente. El editor puede formar parte del entorno de programación, o bien ser una herramienta aparte que el usuario arranca por separado. El fichero creado con el código fuente será procesado posteriormente por otras herramientas del entorno de programación. Además de escribir código, el editor puede aportar otras facilidades tales como:
    - *Análisis del código*. El editor avisa al programador de los errores cometidos según va escribiendo. Estos errores son fundamentalmente sintácticos.
    - *Ayuda a la codificación*. El editor presenta información con la sintaxis de la instrucción que el programador ha comenzado a teclear, de tal manera que pulsando una tecla el programador completa la palabra en curso sin tener que escribirla completamente.
  - *Programas traductores*. Realizan la traducción del código fuente a código máquina convirtiéndolo en un programa ejecutable.
  - *Herramientas de depuración*. Una vez codificado el programa, es necesario probarlo para corregir los defectos que pueda tener. Para ello, facilitan la detección de éstos permitiendo al programador que ejecute el programa en desarrollo instrucción a instrucción y analice al mismo tiempo el estado de todas las variables y otros elementos importantes.
  - *Sistemas de control de versiones*. Son herramientas que controlan los cambios realizados en el código fuente durante el desarrollo de un proyecto, clasificándolos por versiones. Permiten la colaboración de diversos programadores en el desarrollo, siendo posible volver a un estado anterior de cualquier fichero en caso de ocurrir problemas.
  - *Otras herramientas.* Los entornos de programación pueden contener generadores de código, gestores de bibliotecas, generadores de datos de prueba, etc.

2.1.2. Tipos de entornos de programación. Integración de herramientas.

- Normalmente todas estas herramientas se integran en un único software conocido como entorno de programación integrado o IDE (Integrated Development Environment). Existen varios tipos de entornos de programación integrados:

    - *Entornos centrados en un lenguaje*. Son aquellos que son específicos para un lenguaje de programación. Suelen ser entornos fuertemente integrados y homogéneos, y funcionan como una herramienta única. Suelen ser fáciles de usar pero a veces son poco flexibles. El editor está orientado al lenguaje concreto y el desarrollo está basado en el código fuente. Ejemplos de este tipo de IDE son Delphi y Visual C++.
    - *Entornos basados en colección de herramientas*. Presentan una integración débil pensada para entornos heterogéneos, aunque son fáciles de adaptar y ampliar. No existe un control sobre la utilización de las herramientas. Normalmente se trata de interfaces de usuario que realizan llamadas a herramientas externas. Un ejemplo de este tipo de IDE es Eclipse.
    - *Entornos multilenguaje*. Son aquellos basados en máquinas virtuales que dan soporte a varios lenguajes de programación de forma integrada. Ejemplos de este tipo de IDE son Java Virtual Machine (basado en Java Runtime Environment) y Microsoft .NET (basado en Common Language Runtime).
- Existen varios criterios de integración de las herramientas de un entorno de programación:
    - *Datos*. El paso de datos evita redundancias e inconsistencias y proporciona interoperabilidad entre herramientas. La transferencia puede ser directa, mediante ficheros, mediante comunicación, a través de un repositorio común o mediante conversión de formatos.
    - *Control*. La integración de control permite invocar herramientas desde otras, utilizando la integración de datos para la comunicación entre ellas. Puede ser mediante la utilización de un lenguaje de script a través de un shell (intérprete de órdenes para un sistema operativo).
    - *Presentación*. Las herramientas deben tener una interfaz amigable y uniforme que limite las formas diferentes de interacción, proporcione las formas adecuadas al modelo mental del usuario, mantenga la información disponible y ofrezca los tiempos de respuesta adecuados.
    - *Proceso*. Consiste en compartir los elementos del modelo de proceso, automatizar la realización de tareas y forzar las restricciones. El modelo de proceso está formado por pasos (unidades de trabajo), sucesos (condiciones que pueden desencadenar la ejecución de una acción) y restricciones (condiciones impuestas por razones de metodología).

3. Herramientas de desarrollo y pruebas.

3.1.1. Compiladores.

3.1.1.1. Concepto. Compilación y enlazado.

- Son aquellos programas que se encargan de traducir el código fuente en su totalidad a lenguaje máquina antes de poder ser ejecutado. Esta operación se realiza en dos pasos:
    - *Compilación*. Se realiza un análisis léxico, sintáctico y semántico del programa fuente, seguido de un proceso de optimización de código Esto da lugar a código máquina no ejecutable denominado código objeto, ya que en compilación no es posible determinar las direcciones de memoria de las variables y las instrucciones, debido a lo siguiente:
        - Un programa fuente puede estar dividido en varios ficheros, que se interrelacionan de tal forma que pueden compartir algunas variables entre ellos, así como pueden utilizarse en unos ficheros funciones cuyo código se encuentra escrito en otros ficheros.

        - Todos los lenguajes disponen de librerías que contienen funciones para utilizar. Las librerías se encuentran en código objeto y no pueden tener asignadas las direcciones de las variables ni de las instrucciones, ya que pueden formar parte de cualquier programa.
    - *Enlazado*. Después de compilar todos los programas fuente, es necesario combinar los módulos objeto correspondientes en un solo programa ejecutable:
        - A las variables se les asigna su dirección en memoria y todas las referencias a dichas variables tienen que apuntar a sus correspondientes direcciones de memoria. Las funciones y los procedimientos también obtienen una posición de memoria.
        - Cuando el sistema soporta varios programas al mismo tiempo en la memoria, las direcciones de los símbolos no pueden ser fijas pues se producirían conflictos entre unos programas y otros. En este caso se asignan a los símbolos direcciones relativas respecto de la dirección que el sistema operativo le da al programa en el momento de ser cargado en memoria. Se dice entonces que el programa es reubicable.

3.1.1.2. Características del proceso de compilación. Tipo de compiladores.

- Las características del proceso de compilación son las siguientes:
    - Puede ser lento, ya que todo el código debe ser correcto antes de ejecutarse. Sin embargo el lenguaje máquina generado es eficiente porque está optimizado.
    - Pueden producirse errores al analizar el código fuente que impidan la generación del ejecutable, o advertencias (warnings) que sí permiten la generación del ejecutable. En cualquiera de los dos casos, el compilador crea una lista con los mensajes de error y advertencia, mostrando las líneas en los que se han detectado.
- Existen varios tipos de compiladores:
    - *Ensambladores*. Realizan la traducción de código ensamblador a código máquina.
    - *Cruzados*. Realizan una traducción a lenguaje máquina de otra máquina distinta.
    - *Incrementales*. Sólo compilan el código fuente modificado desde la última compilación.

3.1.1.3. Compilación.

3.1.1.3.1. Análisis léxico.

- El compilador somete al fichero fuente a un análisis léxico para detectar errores. Durante este proceso el compilador separa las diferentes unidades del lenguaje que componen el programa, también denominadas tokens. Estos tokens son las palabras reservadas, los nombres de las variables y constantes, los operadores y los nombres de los procedimientos y funciones.
- En esta fase del análisis se detectan varios tipos de errores:
    - Utilización de nombres ilegales para las variables o para los procedimientos y funciones, como nombres demasiado largos o que contienen caracteres no permitidos.
    - Símbolos no reconocidos en el lenguaje de programación.
    - Utilización incorrecta de constantes, como el uso de la coma en vez del punto decimal.
- El compilador crea una tabla de símbolos, que es una estructura de datos usada por el compilador para asociar a cada variable, función y procedimiento que aparece en el programa fuente una representación de sus atributos. La información guardada depende del elemento almacenado, del lenguaje y la estructura del compilador. Suele tener el nombre del símbolo, la dirección de

memoria, el tipo de dato y la información de localización. La tabla puede tener estructura de tabla consecutiva, de tabla ordenada, de tabla hash o de árbol binario de búsqueda.

3.1.1.3.2. Análisis sintáctico.

- Un lenguaje de programación especifica un conjunto de reglas sintácticas que deben ser respetadas. Una de las razones para que existan estas reglas es precisamente que el proceso de traducción a código máquina pueda ser efectuado de forma automática. Cuando el programador no respeta estas reglas el compilador no puede traducir.
- El analizador sintáctico (o parser) del compilador recibe los tokens y comprueba su ordenación correcta mediante reconocimiento descendente (desde el axioma gramatical a la sentencia) o ascendente (desde la sentencia al axioma gramatical), generando un árbol sintáctico en el cual los operadores se encuentran en los nodos no terminales y los operandos en los nodos terminales.
- Los errores detectados por el parser pueden ser:
  - Errores en la estructura de la frase. Por ejemplo, en una función se necesitan dos parámetros y sólo se ha dado uno, o paréntesis y llaves que no tienen su correspondiente pareja.
  - Utilización incorrecta de palabras reservadas, que tienen que estar situadas en el lugar correcto dentro de la instrucción.

3.1.1.3.3. Análisis semántico.

- Una instrucción sintácticamente correcta puede no tener ningún significado y por lo tanto no podrá ser traducida a un bloque de código máquina. Por tanto es necesario realizar una interpretación de las instrucciones del código fuente.
- El analizador semántico realiza una comprobación de reglas semánticas dentro del árbol sintáctico, y detecta incoherencias en el programa. Por ejemplo un bloque de código puede no ser accesible pues el flujo de ejecución de instrucciones nunca pasa por él. Utiliza la tabla de símbolos para implementar las restricciones típicas de los lenguajes de programación, como son:
  - Control de la unicidad de identificadores.
  - Control de variables no declaradas.
  - Verificación de tipos.
  - Implementación de reglas de ámbito.
- El objetivo del análisis semántico es asegurar que el tipo de construcción del programa coincida con el previsto en su contexto, es decir:
  - Compatibilidad de tipos en las asignaciones.
  - Operadores aplicados a datos del tipo adecuado.
  - Número y tipo correcto en los parámetros de funciones.

3.1.1.3.4. Código intermedio, optimización y código objeto.

- Después de los análisis, se crea un código intermedio que permite especificar operaciones sin entrar en detalles de bajo nivel específicos de la máquina. Está formado por construcciones claras, sencillas y uniformes, con significado unívoco. Ofrece las siguientes ventajas:
  - Facilita la optimización y la generación de código máquina para todas las máquinas objetivo, ya que ofrece una representación simple y uniforme.
  - Mayor modularidad, ya que aísla elementos de alto nivel de los dependientes de la máquina.

    - Mayor portabilidad, puesto que la generación de código no depende del lenguaje de programación original. Para crear un compilador para otra máquina basta crear una nueva etapa final de generación.
- Posteriormente el compilador intenta optimizar el código intermedio. Para ello elimina todas las instrucciones superfluas que se han generado en código máquina, y realiza la factorización de expresiones comunes. El compilador también toma decisiones para adaptar el código a la arquitectura de la CPU. Puede decidir reservar registros de la CPU para algunas variables en lugar de posiciones de memoria, a fin de que el programa sea más rápido.
- El último paso del compilador es el relativo al código objeto. Existen tres opciones:
    - *No generar código objeto*. En tal caso una máquina virtual interpretará el código intermedio. Como el código intermedio está más cerca del nivel de máquina que de un lenguaje de alto nivel, los programas se ejecutan más rápidamente que los programas completamente interpretados, aunque más despacio que los programas traducidos a código máquina.
    - *Generar código ensamblador*. De esta manera se facilita la traducción, pero necesita un ensamblador para generar el verdadero código objeto.
    - *Generar el código máquina directamente*. Con una alta dependencia del sistema operativo y de la arquitectura de la CPU, puede generarse:
        - **Código absoluto**. El programa se localizará en una posición fija de memoria. La ejecución en este caso es directa, pero muy restrictiva.
        - **Código relocalizable**. El programa puede situarse en cualquier posición, permitiendo la compilación separada de varios módulos. Sin embargo, necesita de procesamiento adicional (enlazado) e incluir información de localización en el fichero objeto.

3.1.1.4. Enlazado.

- El enlazador es la herramienta encargada de combinar los módulos objeto creados en la fase de compilación en un solo programa. Se encarga de repasar todo el programa, asignando valores a todas las referencias de la tabla de símbolos hasta que finalmente todas ellas han sido resueltas. Si se hace alguna llamada a un procedimiento o una función cuyo código no existe, el enlazador genera un error y el fichero ejecutable no se produce.
- El procedimiento básico consiste en disponer los módulos objeto uno tras otro en un solo fichero y a continuación dar direcciones de memoria a todos los elementos de la tabla de símbolos:
    - Todas las referencias a los distintos procedimientos, funciones y variables tienen que apuntar a sus correspondientes direcciones de memoria.
    - Cada vez que una instrucción del programa realiza una llamada a un procedimiento o una función, se coloca una instrucción de salto a la posición de memoria en la que se encuentra ubicado el código de la función. Los procedimientos no devuelven ningún valor, a diferencia de las funciones, y por tanto su mecanismo de salida es ligeramente más sencillo.
    - Cuando se llama a un procedimiento o una función, el código del mismo no necesariamente tiene que encontrarse en el mismo fichero objeto que la instrucción de llamada.

3.1.2. Intérpretes.

3.1.2.1. Concepto. Ventajas e inconvenientes.

- Son aquellos programas que traducen a lenguaje máquina y ejecutan inmediatamente línea por línea el código fuente, sin crearse por tanto un programa objeto para posteriores ejecuciones.
- Los programas interpretados son compactos, pero más lentos y menos eficientes que los programas compilados porque el código no se analiza de forma global, y necesitan del intérprete cada vez que se ejecutan. Sin embargo los intérpretes permiten una rápida depuración de los programas, ya que puede ejecutarse parcialmente el código fuente aunque no esté finalizado.

3.1.2.2. Máquinas virtuales.

- Una máquina virtual es una capa de abstracción que separa el funcionamiento de un ordenador de su hardware. Se pueden identificar dos tipos de máquinas virtuales concretas:
  - *De compilación*. Las que intervienen en la preparación de un programa para su ejecución.
  - *De ejecución*. Las que permiten la ejecución de dicho programa.
- Las máquinas virtuales se suelen implementar en términos de otras máquinas virtuales existentes. Por tanto no existen aisladas, sino como parte de una jerarquía.
- El proceso de interpretación de un programa se realiza a través de una máquina virtual que simula un ordenador cuyo código máquina es el lenguaje de alto nivel que está siendo interpretado. Típicamente esta máquina virtual se construye a través de un conjunto de programas de código máquina que representa los algoritmos y estructuras de datos necesarios para la ejecución de las instrucciones del lenguaje de alto nivel.

3.1.3. Depuradores.

- Son programas que ejecutan paso a paso un programa con el fin de corregir los errores lógicos. Normalmente son proporcionados por el mismo fabricante del compilador utilizado, ya que suelen ser de uso exclusivo para los programas generados por dicho compilador.
- Permiten realizar las siguientes operaciones:
  - Ver el contenido de las variables del programa y modificar de su valor.
  - Establecer puntos de ruptura y detener la ejecución en un punto determinado del programa.
  - Desviar condicionalmente el flujo de ejecución de un programa.
- Para poder ejecutar un programa bajo un depurador, generalmente es necesario compilarlo con una opción especial que incorpora la tabla de símbolos al fichero objeto, para poder acceder a las variables y a las funciones por su nombre y no por su posición de memoria.

4. Herramientas de desarrollo y pruebas en Linux.

4.1.1. Compilador gcc (GNU compiler).

- El compilador *gcc* es rápido, flexible y riguroso con el estándar de ANSI C. Puede funcionar como compilador cruzado para un gran número de arquitecturas distintas. En realidad *gcc* no genera código máquina, sino código ensamblado. La fase de ensamblado a código máquina la realiza el ensamblador de GNU (*gas*) y el enlazador de GNU (*ld*) se encarga de los objetos resultantes. Este proceso es transparente para el usuario, ya que a no ser que se especifique lo contrario, *gcc* realiza automáticamente el paso desde código en C a un fichero ejecutable.

- Su sintaxis es `gcc [opciones] fichero(s)` donde `fichero(s)` puede ser uno o varios ficheros fuente o ficheros objeto, y `opciones` puede ser:
  * `-o <ejecutable>.` El fichero ejecutable generado por *gcc* es por defecto `a.out`. Mediante este modificador, se especifica el nombre del ejecutable.
  * `-Wall.` No omite la detección de ningún aviso de compilación (warning).
  * `-g.` Incluye en el ejecutable información necesaria para utilizar un depurador.
  * `-c.` Preprocesa, compila y ensambla, pero no enlaza. El resultado es un fichero objeto con extensión *.o* y el mismo nombre que el fichero fuente.

4.1.2. Depurador gdb (GNU Debugger).

- Se trata de un depurador asociado a *gcc*, que necesita que el programa sea compilado previamente con la opción `-g`. El depurador permite establecer puntos de ruptura condicionales y detener la ejecución del programa cuando el valor de una expresión cambie.
- Su sintaxis es `gdb fichero` donde `fichero` es el nombre del ejecutable creado con *gcc*. Con esta instrucción se entra dentro del depurador, que se maneja mediante línea de comandos, aunque existe una interfaz gráfica de usuario para este depurador denominado *ddd* (GNU data display debugger) más intuitiva de manejar.

4.1.3. Herramienta make.

- Un programa en C normalmente está formado por varios ficheros fuente y ficheros cabecera. Cada vez que se modifica algún fichero fuente o cabecera, el programa debe recompilarse para crear un ejecutable actualizado. Sin embargo no es necesario recompilar todos los ficheros, sino sólo los afectados por la modificación.
- La utilidad *make* determina automáticamente qué ficheros del programa deben ser recompilados y las órdenes que se deben utilizar para ello. Para utilizar *make* es necesario escribir un fichero denominado *Makefile*, que describe las dependencias entre ficheros y las órdenes que se deben ejecutar con cada fichero. Si en un directorio existe un fichero *Makefile*, la orden *make* se encargará de ejecutar las órdenes que contiene.
- Un fichero *Makefile* simple está compuesto por reglas que tienen la siguiente forma:

```
<etiqueta>:<lista de dependencias>
    <orden>
    <orden>
    …
```

- Una etiqueta es un rótulo que generalmente se refiere al nombre del fichero objeto que se quiere actualizar con la regla. También hay etiquetas que indican la acción que realiza la regla. Una lista de dependencias es el conjunto de ficheros fuente u objeto que intervienen en la regla. Si cualquiera de estos ficheros es modificado, la regla se activa. Una orden es la acción que *make* realiza si la regla se activa. Las líneas de órdenes empiezan siempre con un tabulador, para distinguirlas de las líneas de etiquetas.
- La orden *make* sin etiquetas activa la primera regla del fichero *Makefile*, la orden *make* <*etiqueta*> activa la regla correspondiente a la etiqueta especificada. Las etiquetas que indican el

nombre de una acción deben declararse explícitamente como falsas con .PHONY para que make no compruebe la existencia de un fichero con ese nombre.

- Es posible definir variables para simplificar el fichero *Makefile*. La definición se realiza antes de la primera etiqueta de la siguiente manera: *nombre_variable=texto_al_que_sustituye*, y la referencia a una variable se realiza mediante *$(nombre_variable)*. También se puede emplear caracteres comodín para hacer referencias a nombres de ficheros: '*' equivale a cualquier cadena de caracteres y '?' equivale a un carácter. El carácter comodín se puede escapar con '\'.
- Un ejemplo de *Makefile* que permite generar un ejecutable llamado prog.exe a partir del código fuente contenido en tres ficheros (prog.c, vector.c y vector.h):

```
objs = prog.o vector.o
CC = gcc
todo: prog.exe
vector.o: vector.h vector.c
   $(CC) –c vector.c -Wall
prog.o: vector.h prog.c
   $(CC) –c prog.c -Wall
prog.exe: $(objs)
   $(CC) –o prog.exe $(objs)
.PHONY: limpiar
limpiar:
   rm prog.exe $(objs)
```

4.1.4. Sistema de control de versiones CVS.

- El sistema de control de versiones CVS (Concurrent Versión System) es un sistema descentralizado en el cual cada programador utiliza su propio directorio de trabajo. Permite la edición concurrente de ficheros e integra todos los cambios realizados en un fichero mezcla.
- Todas las versiones de un fichero se guardan de manera conjunta en un único fichero incrementalmente. Se basa en dos programas: *diff* (detecta diferencias entre dos ficheros y las vuelca a otro) y *patch* (reconstruye un fichero a partir del original y el fichero de diferencias).
- El repositorio es el directorio donde se almacenan todos los ficheros con las diferentes versiones de uno o varios proyectos. Es el depósito común de información del proyecto, y se recurre a él para recuperar ficheros y almacenar los cambios. El repositorio almacena información de control para CVS y las diferencias entre las versiones de cada uno de los ficheros del proyecto.
- El modelo de trabajo con CVS es el siguiente:
  - Un programador descarga una copia desde el repositorio con *checkout*. El programador edita libremente su copia. Al mismo tiempo, cualquier otro miembro del equipo puede trabajar sobre otra copia de trabajo.
  - El programador finaliza sus cambios y los entrega al repositorio de CVS con *commit*, junto con un texto explicativo de las modificaciones realizadas. CVS integra los cambios en la copia maestra del proyecto. Otros programadores pueden solicitar a CVS comprobar si la copia maestra ha sufrido cambios recientes con *update*.