Utilidades para el desarrollo y pruebas de programas

Alberto García Díaz

18 de septiembre de 2011

Índice

1. Utilidades para el desarrollo y pruebas de programas

Como vimos en el tema introductorio a los lenguajes de programación, estos han ido evolucionando y por tanto también lo han hecho las herramientas para su desarrollo. Las aplicaciones informáticas son cada vez más grandes y complejas, produciéndose muchos errores, etc. Existe un amplio abanico de herramientas que facilitan la labor del programador, las veremos ahora, algunas de ellas son realmente imprescindibles.

1.1. Herramientas básicas

A continuación veremos las herramientas básicas y algunas imprescindibles para el desarrollo de programas.

1.1.1. Editores

Evidentemente, para escribir un programa necesitaremos un editor, y aunque en principio cualquiera podría valer, podemos encontrar una gran variedad de editores con características específicas para la escritura de código. Entre otras características cabe destacar:

- Resaltado de sintaxis, es decir, palabras reservadas, constantes, comentarios, etc. se enfatizan de alguna manera: distinto color, negrita,...
- Navegación de código, ofreciendo el listado de funciones, métodos, procedimientos del programa de manera que podemos acceder a ellos rápidamente.
- Macros de escritura, que permiten escribir rápidamente las sentencias básicas del lenguaje de una manera rápida y sencilla, mediante autocompleción por ejemplo.

Algunos ejemplos de editores: `vim`, `emacs`, `UltraEdit`, `Notepad++`, etc.

1.1.2. Compiladores

Como ya sabemos, los lenguajes de alto nivel no pueden ser ejecutados directamente por la computadora, sino que es necesario un proceso de traducción al lenguaje máquina. Los compiladores son los encargados de realizar esta traducción, ofreciendo la posibilidad de realizar optimizaciones (que puede incluso decidir el programador) en el código máquina resultante.

1.1.3. Enlazadores

Una aplicación puede dividirse en diferentes módulos, de manera que cada uno de ellos puede desarrollarse y compilarse de manera independiente, obteniendo un fichero de código objeto por cada uno de ellos.

1.1.4. Depuradores

Un depurador permite monitorizar la ejecución de un programa y se utiliza para localizar errores en la lógica del programa, mal uso de la memoria dinámica, etc.

1.1.5. Gestión de proyecto

En aplicaciones grandes, tendremos una gran cantidad de módulos y posiblemente en la compilación y enlazado de unos no se podrá hacer sin haber compilado antes otros. Es decir, podemos encontrar ciertas dependencias entre módulos a la hora de compilar. Para evitar tener que compilar cada módulo a mano y poder especificar tales dependencias entre módulos se utilizan utilidades como `make`, que usa un fichero (`makefile`) dónde se expresa, con una sintaxis específica, cómo obtener el ejecutable de la aplicación a partir de su código fuente. Otra ventaja adicional es que es capaz de determinar qué archivos de código fuente se han modificado y sólo recompila aquellos que lo necesitan, ahorrando de esta forma tiempo en la compilación y eliminando posibles errores.

1.2. Herramientas avanzadas

Las siguientes herramientas integran o mejoran a las anteriores para hacer la labor de programación más sencilla.

1.2.1. Entornos de desarrollo integrado

Aplicaciones que encapsulan todas las fases de desarrollo, incluyendo editores, compiladores, depuradores, etc. Tienen mucha importancia en el diseño e implantación de aplicaciones gráficas. Ejemplos: C++ Builder, Microsoft Visual StudioX, Anjuta, Kdevelop, Eclipse, NetBeans, ...

1.2.2. Sistemas de control de versión

En las aplicaciones grandes trabajan siempre varios programadores, a veces decenas o centenas de ellos. En estos casos es imprescindible una herramienta de control de versión. También para generar versiones válidas en un

momento dado y tener un historial de cambios. Las herramientas de control de versión funcionan con el sistema cliente-servidor. Consisten en un servidor que gestiona el código fuente y se accede a él por red desde los clientes, que normalmente son los que manejan los programadores. Esto permite el trabajo colaborativo incluso a través de Internet. Ejemplos de este tipo de herramientas son: CVS y MicrosoftSourceSafe.

1.2.3. Generadores de documentación

Son utilidades capaces de extraer información descriptiva del código fuente de un programa bien documentado, es decir, con sus correspondientes comentarios en un formato adecuado. Entre otra información, se puede obtener:

- Manuales de referencia, con la interfaz y la descripción de todas las funciones y procedimientos del código.
- Grafos de dependencia entre funciones o procedimientos.
- Diagramas de herencia y colaboración en lenguajes Orientados a Objetos.

Ejemplos de este tipo de herramientas son: Doxygen, JavaDOc.

1.2.4. Desemsambladores

Son programas que permiten obtener el código fuente en ensamblador a partir del ejecutable de una aplicación. Se suelen utilizar en nociones de ingeniería inversa, ya que en lugar de estudiar el código máquina directamente es mucho más cómodo hacerlo sobre ensamblador. Existen muchas utilidades de este tipo. Veamos un pequeño ejemplo sobre *Linux*. Lo primero es buscar un fichero ejecutable, obviamente si intentamos desemsamblar otro tipo de fichero como una imagen, documento de texto, etc. no vamos a obtener un listado en ensamblador ya que dichos ficheros no son programas, sino simplemente datos organizados de una determinada manera. Por ejemplo, creamos un fichero llamado *prueba.c* con el siguiente contenido:

```
#include <stdio.h>
int main() {
    printf("Esto es una prueba.\n");
    return 0;
}
```

Una vez creado lo compilamos y *linkamos* para obtener el ejecutable mediante `gcc -o prueba.exe prueba.c` y deberíamos obtener un nuevo fichero

prueba.exe al que deberemos darle permisos de ejecución haciendo: `chmod u+x prueba.exe`. Finalmente lo ejecutamos para comprobar que todo ha ido correcto: `./prueba.exe` Como vimos anteriormente podríamos abrir el ejecutable mediante un editor hexadecimal y veríamo el contenido binario de dicho ejecutable. Obviamente, analizar lo que hace este código es bastante complicado, por no decir casi imposible. Generemos pues el código en ensamblador, que aunque tiene una relación 1 a 1 con las instrucciones del microprocesador son más legibles y aparecen organizadas. Ejecutamos `objdump -d prueba.exe`. Por pantalla se nos mostrará el código desemsamblado de dicho programa, entre lo que podemos ver:

```
add     $0x0,%al
add     %al,(%eax)
adc     $0x0,%al
add     %al,(%eax)
add     (%eax),%eax
add     %al,(%eax)
inc     %edi
```

2. Compiladores

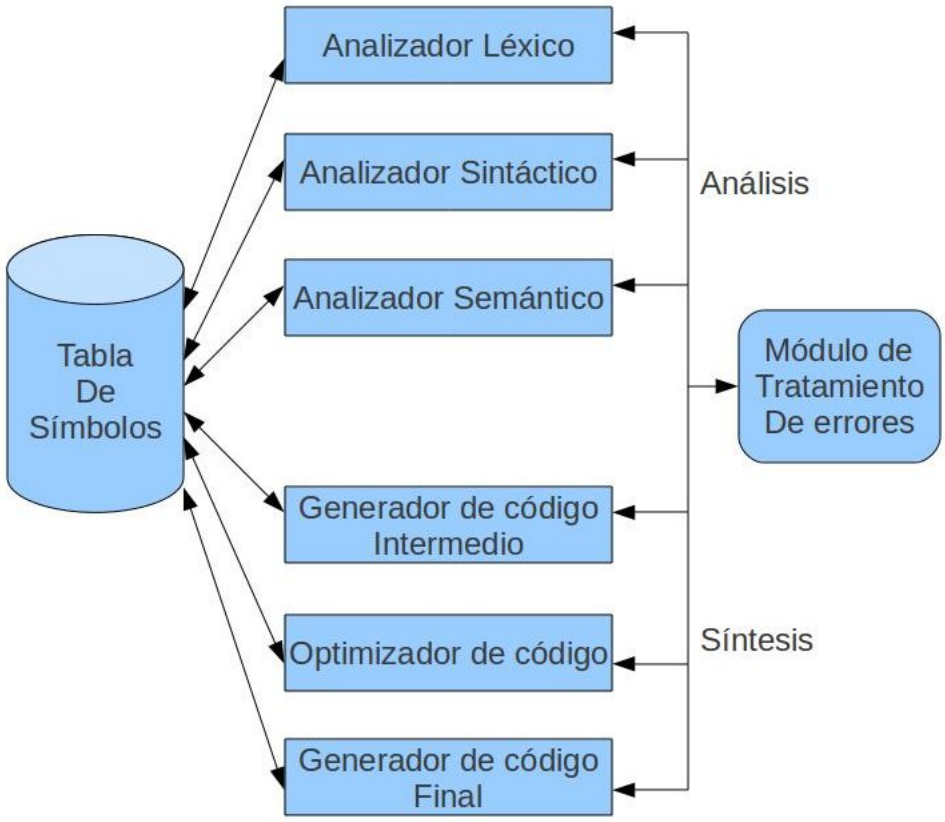
2.1. Definición de compilador

Genéricamente, podemos definir un **traductor** como una máquina teórica que tiene como entrada un texto escrito en un lenguaje L1 y como salida un texto escrito en un lenguaje L2. Habitualmente se denomina a L1 **lenguaje fuente** y a L2 **lenguaje objeto**. Un **compilador** no es más que un traductor que convierte texto escrito en un lenguaje fuente de alto nivel en un programa objeto escrito en código máquina. Este programa objeto puede almacenarse en memoria auxiliar o masiva para ser ejecutado posteriormente sin necesidad de volver a realizar la traducción.

2.2. Etapas y módulos de un compilador

En el proceso de compilación pueden distinguirse dos fases principales: una fase de **análisis**, en la que cuál se lee el programa fuente y se estudia la estructura y significado del mismo; y otra fase de **síntesis**, en la que se genera el programa objeto. En un compilador pueden distinguirse, además, algunas estructuras de datos comunes, la más importante de las cuales es la **tabla de símbolos**, junto con las funciones de gestión de ésta y de los demás elementos del compilador, y de una serie de rutinas auxiliares para

detección de errores. La compilación es un proceso complejo y que puede consumir un tiempo considerable. En cualquiera de las fases de análisis el compilador puede dar mensajes sobre los errores que detecta en el programa fuente, cancelando en ocasiones el proceso. Podemos observar gráficamente las interrelaciones entre las distintas fases del compilador:



A continuación, vamos a describir las diferentes fases que componen el proceso de compilación.

2.2.1. Analizador léxico

El analizador léxico, es la parte del compilador que lee el programa fuente, carácter a carácter, y construye a partir de éste unas entidades primararias llamadas **tokens**. Es decir, el analizador léxico transforma el programa fuente en tiras de tokens. Estos elementos son los símbolos de lenguaje: palabras reservadas, símbolos de operadores, identificadores de variables, números con-

stantes, etc. La información que da esta secuencia de tokens es suficiente para el análisis de la estructura de la sentencia, pero no basta para un análisis de su significado, para lo cuál hay que tener cierta información de cuáles son las variables que entran en juego en esta sentencia. El analizador léxico aisla los símbolos, identifica su tipo y almacena en las tablas de símbolos la información del símbolo. Por ejemplo, si consideramos la expresión en lenguaje C:

```
long = 2 * PI * radio;
```

el analizador léxico generará la siguiente secuencia de símbolos:
`Identificador[0] Asignación Número[0] Operador * Identificador[1] Operador * Identificador[2]`
Además, el analizador léxico realiza otra serie de tareas secundarias:

- Identifica y salta comentarios, espacios en blanco, tabuladores, etc.
- Identifica posibles errores de léxico
- Relaciona los posibles errores que produce el compilador en sus diversas fases con la parte de código fuente correspondiente.
- Si el lenguaje incorpora macros, el analizador puede incorporar un preprocesador.

Hay diversas razones por la que se separa el análisis léxico del análisis sintáctico:

1. El diseño del analizador sintáctico es más fácil.
2. Se mejora la eficiencia del compilador en su conjunto. Es usual realizar el analizador léxico en ensamblador, que proporciona una gran rapidez en la lectura de los símbolos.
3. Aumenta la portabilidad del compilador. Pensad en que el código fuente puede estar realizado en diversos tipos de codificación de caracteres.

2.2.2. Analizador sintáctico

La sintaxis de un lenguaje de programación especifica cómo deben escribirse los programas mediante un conjunto de reglas de sintaxis o gramática del lenguaje. Un programa es sintácticamente correcto cuando sus estructuras (expresiones, sentencias declarativas, asignaciones, etc.) aparecen en un orden correcto. Por ejemplo, en C, la expresión:

```
a + b = 30;
```

es incorrecta, pues antes del operador de asignación (=) debe aparecer un único identificador. Notar también que una sentencia puede ser sintácticamente correcta pero carecer de sentido, por ejemplo asignar una cadena a una variable de tipo entero. Se han definido varios sistemas para definir la sintaxis de los lenguajes de programación (metalenguajes), entre ellos destacan la notación BNF y los diagramas sintácticos. En ambos casos se definen las construcciones del lenguaje (símbolos no terminales) a partir de los símbolos terminales, reconocidos por el analizador léxico, por ejemplo, la construcción formada por un identificador seguido del signo = es la expresión de una sentencia de asignación. Un diagrama sintáctico es un grafo dirigido, que describe, al recorrerlo, las distintas posibilidades de creación de una construcción del lenguaje. Con este esquema de análisis, el analizador puede generar mensajes de error cuando se encuentre un símbolo no esperado.

2.2.3. Análisis semántico

El análisis semántico, no se realiza claramente diferenciado del resto de tareas del compilador, más bien podría decirse que completa a las dos anteriores. La semántica de un lenguaje de programación es el significado dado a cada una de las distintas construcciones sintácticas. Está íntimamente ligada a la semántica de la sintaxis. Desde el punto de vista del compilador, las rutinas de análisis semántico pueden ser de dos tipos:

Estáticas Se realizan durante la compilación del programa. Ejemplos: Comprobaciones de tipos, unicidad de los nombres en identificadores declardos, etc.

Dinámicas Son las que el compilador incorpora al programa traducido. Añade en el código objeto ciertas instrucciones que se ejecutarán simultáneamente a la ejecución del programa. Ejemplos: Estado de la máquina, fallos al abrir ficheros, errores numéricos, errores de direccionamiento de memoria, etc.

2.2.4. Generación de código intermedio

Habitualmente, los compiladores generan un código intermedio, puede definirse como el lenguaje de una máquina abstracta que el diseñador del compilador define. La definición de este código intermedio se realiza de forma que:

- Evitar limitaciones propias de una máquina real en cuanto a número de registros, alineación de datos, etc.

- Facilitar la tarea de optimización de código. Al ser más flexible en cuando direccionamiento del código en la máquina real.
- Facilitar la portabilidad del compilador.

El incoveniente principal: No hay un aprovechamiento de algunas instrucciones específicas de la máquina objeto, que se podría resolver en algunos casos si se realiza una adecuada optimización del código. En esta etapa se realiza la asignación de memoria de los datos definidos por el programa.

2.2.5. Optimizador del código

Se recibe el código intermedio y se mejora atendiendo a factores como la velocidad de ejecución, tamaño del programa objeto, etc. Ciertos compiladores permiten al usuario omitir o reducir las optimizaciones, disminuyendo así el tiempo de compilación. Se realizan analizando el programa objeto globalmente. Ya vimos en clase un ejemplo de optimización eliminando una asignación constante dentro de un bucle.

2.2.6. Generador de código final

En esta etapa se genera el código objeto mediante la traducción del código intermedio optimizado.

2.2.7. Módulo de tratamiento de errores

Facilita la detección, y en algún caso, la recuperación de los errores producidos en las distintas fases de compilación. Cuando se detecta un error, trata de localizar su posición exacta y su posible causa para presentar al programador un mensaje de diagnóstico, que será incluido en el listado de compilación.

2.2.8. Tabla de símbolos

Es una estructura de datos usada para asociar a cada símbolo del programa fuente (identificadores, constantes,...) un contenido semántico. Suele estar formado por registros de longitud fija. Los campos dependen del lenguaje a compilar y de la estructura del compilador. A modo de ejemplo, algunos campos que pueden definirse serían:

- Nombre del símbolo
- Dirección de memoria: Dónde se guardarán los valores de las variables, será una dirección relativa al programa en cuestión.

- Número de línea de la declaración o de dónde se usa el símbolo. Útil para la depuración, es una lista encadenada de números de línea.

Hay que tener en cuenta que en la tabla de símbolos no se guarda el valor de las variables, sólo su dirección.

2.3. Compiladores cruzados

Se denominan traductores cruzados a aquellos traductores que efectúan la traducción de programas fuente a programas objeto en una computadora distinta (A) a aquella en la que se ejecutará el programa objeto (B). A la máquina A se le llama huésped (guest) y a la B computadora anfitriona (host). Los emuladores son programas que simulan el comportamiento de otra computadora.

3. Intérpretes

Son traductores que realizan su tarea por bloques, de forma que se van analizando bloques de programa fuente, se genera el código máquina correspondiente y se ejecuta dicha porción de código. Este ciclo se repite hasta el final del programa. Por tanto, análisis, traducción y ejecución están fuertemente ligadas. En la etapa de generación del código, se busca una combinación de instrucciones en lenguaje máquina que hagan exactamente lo mismo que haría la instrucción de alto nivel. En la etapa de ejecución, se le pasa el control al código objeto generado hasta que termine, momento en el cuál, el control vuelve a pasar al intérprete. Tienen las siguientes características:

- Los programas objeto se ejecutan de forma lenta.
- los programas objeto se pueden detener, modificar el fuente y seguir ejecutando.
- Son fácilmente portables entre distintas máquinas, pues lo que se mueve es el código fuente

Ejemplos de lenguajes de programación interpretados son `LISP` y `BASIC`.

4. Depuradores

Aunque la compilación sea correcta, no implica que el programa esté libre de errores, tanto técnicos como funcionales. Los depuradores son programas

que facilitan al programador encontrar y corregir dichos errores. Permiten observar llamadas a funciones, inspeccionar valores de variables, depurar funciones miembro y aplicaciones multi-hebra, controlar la ejecución paso a paso (a nivel de instrucción máquina o ensamblador), instalar puntos de control (breakpoints). Para trabajar necesitan el ejecutable y el código fuente del programa. El compilador ha de generar cierta información de depuración; inclusión del número de línea del código fuente de cada instrucción, etc.

5. Máquinas virtuales

Hemos visto que los lenguajes interpretados y compilados tiene distintas características, ventajas e inconvenientes. Un intento de combinar lo mejor de ambos surge durante la década de los 90; en enfoque de máquina virtual. La filosofía de la máquina virtual es la siguiente: el código fuente se compila, detectando los errores sintácticos y se genera una especie de código ejecutable, con un código máquina dirigido a una CPU imaginaria. A este código se le denomina código intermedio, lenguaje intermedio, p-code o byte-code. Para poder ejecutar ese código se construye un intérprete. Este intérprete es capaz de ejecutar cada una de las instrucciones del código intermedio en una CPU real. A este intérprete se le llama máquina virtual.

Portabilidad Existe una máquina virtual para cada plataforma, y por tanto, es capaz de ejecutarse dicho código ya compilado. Será más rápido pues ya no tiene que comprobar la existencia de errores y tampoco realizar la compilación en sí.

Estabilidad Al ser ejecutado por una CPU virtual existe mejor control sobre el código, evitando poner en peligro la estabilidad de la plataforma real.

Ejemplos: Java de Sun Microsystems, lenguajes de la plataforma .NET. Cabría destacar la plataforma Mono, que permite la ejecución de proyectos .NET en plataformas distintas a la de Windows.