IvanVR
www.wuolah.com/student/IvanVR

Tema 3.pdf

Resumen Tema 3.- Compilación y enlazado

1° Fundamentos del Software

Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

TEMA 3. COMPILACIÓN Y ENLAZADO

FUNDAMENTOS DEL SOFTWARE

No nos vemos pero ahora estamos más cerca #QuédateEnCasa

Primera academia especializada en estudios de la Facultad de Informática UCM

academia@mathsinformática.com

C/Andrés Mellado, 88 duplicado

www.mathsinformatica.com

academia.maths

91 399 45 49

615 29 80 22

Material online

- Resolución de ejercicios en vídeo
- Clases virtuales Skype y Hangouts
- Grupos de asignatura en Whastssap

Maths informática

1.| LENGUAJES DE PROGRAMACIÓN

Un lenguaje de programación es un conjunto de símbolos y reglas para combinarlos, que se usan para expresar algoritmos. Estos lenguajes poseen un léxico (vocabulario o conjunto de símbolos permitidos), una sintaxis (indica cómo realizar construcciones del lenguaje) y una semántica (determina el significado de cada construcción correcta).

Sus características fundamentales son:

1.Son **independientes de la arquitectura física del computador**, por tanto, no obligan al programador a conocer los detalles del computador que utiliza y permiten utilizar los mimos programas en computadores diferentes, con distinto lenguaje máquina (portabilidad)

2. Normalmente, una sentencia en un lenguaje de programación de alto nivel da lugar, al ser traducida a varias instrucciones en lenguaje máquina.

3. Utilizan **notaciones cercanas a las habituales** en el ámbito en que se usan. Con estos lenguajes las operaciones se expresan con sentencias o frases muy parecidas al lenguaje matemático o natural. Por lo general, se suelen usar palabras o términos en inglés.

2.| CONSTRUCCIÓN DE TRADUCTORES.

2.1-DEFINICIÓN DE TRADUCTOR

Como el computador puede interpretar y ejecutar únicamente código máquina, existen programas traductores, que traducen programas escritos en lenguajes de programación a lenguaje máquina. Un *traductor* es un programa que recibe como entrada un texto en un lenguaje de programación concreto, y produce, como salida, un texto en lenguaje máquina equivalente. El programa inicial se denomina *programa fuente*, y el programa obtenido, *programa objeto*.

2.2-EL PROCESO DE TRADUCCIÓN

2.2.1 GRAMÁTICAS

Una gramática se compone de **4 elementos**:

1. V_N : Conjunto de símbolos no terminales
2. V_T : Conjunto de símbolos terminales
3. P: Conjunto de reglas de la gramática
4. S : El símbolo inicial (símbolo no terminal a partir del cual se aplican las reglas de la gramática para obtener las distintas cadenas del lenguaje).

Hay varias gramáticas por las que atraviesa el análisis, entre ellas están:

Regulares: Son las gramáticas más restrictivas, reconocidas por *Autómatas finitos.*

Independientes de contexto: Son aquellos que pueden ser reconocidos por un *autómata de pila*.

Sensibles al contexto: Aquellos que pueden ser reconocidos por las *Autómatas Linealmente Acotados*.

2.2.2 COMPILACIÓN

La compilación es un proceso complejo y que consume a veces un tiempo muy superior a la propia ejecución del programa. En cualquiera de las fases de análisis el compilador puede dar mensajes sobre errores que detecta en el programa fuente, cancelando la compilación para que el usuario realice las correcciones oportunas en el archivo fuente.

La traducción por un compilador (la compilación) consta de dos etapas fundamentales: la etapa de análisis del programa fuente y la etapa de síntesis del programa objeto. Cada una de estas etapas conlleva la realización de varias fases:

Análisis lexicográfico: Consiste en descomponer el programa fuente en tokens (caracteres o secuencias de caracteres que tienen un significado concreto en el lenguaje). El analizador lexicográfico aísla los símbolos, identifica su tipo y almacena en las tablas de símbolos la información que pueda ser necesaria durante el proceso de traducción. Como resultado del análisis de léxico se obtiene una representación del programa formada por la descripción de símbolos en las tablas y una secuencia de símbolos junto con la referencia a la ubicación del símbolo en la tabla. Además el analizador de léxico realiza otras funciones secundarias como saltar sobrantes e informar sobre errores de léxico.

Análisis sintáctico: la sintaxis de un lenguaje de programación específica es cómo deben escribirse los programas mediante un conjunto de reglas de sintaxis o gramática del lenguaje. Un programa es sintácticamente correcto cuando sus estructuras aparecen en un orden correcto. A nivel formal, se irá comprobando si cada nuevo símbolo reconocido por el escáner puede ser correcto. Este esquema de análisis genera mensajes de error cuando se encuentre un símbolo no esperado.

Análisis semántico: La semántica de un lenguaje de programación es el significado dado a las distintas construcciones sintácticas. El proceso de traducción, en esencia, es la generación de un código en lenguaje máquina con el mismo significado que el código fuente. En el proceso de traducción, el significado de las sentencias se obtiene de la identificación sintáctica de las construcciones sintácticas y de la información almacenada en las tablas de símbolos. Durante esta fase se puede producir errores cuando se detectan construcciones sin un significado correcto. Los errores semánticos no son detectados por el compilador o intérprete, suelen aparecer cuando un programa termina de forma anormal o salta una excepción.

Generación y optimización de código: en esta fase se crea un archivo con un código en lenguaje objeto (normalmente lenguaje máquina) con el mismo significado que el texto fuente. Dependiendo del compilador este archivo puede ser ejecutable o necesitar más pasos previos (ensamblado, encadenado y carga). En la generación de código intermedio se completan y consultan las tablas generadas en fases anteriores y se realiza la asignación de memoria a los datos definidos en el programa. En la fase de optimización se mejora el código analizando el programa objeto globalmente.

2.3- COMPILADORES E INTÉRPRETES.

Un **compilador** traduce un programa fuente, escrito en lenguaje de alto nivel, a un programa objeto, escrito en lenguaje ensamblador o máquina. Una vez traducido el programa, su ejecución es independiente del compilador. con el compilador, se deben corregir las instrucciones erróneas en el archivo fuente y volver a compilarlo, empezando la ejecución de nuevo.

Un **intérprete** hace que un programa fuente escrito en un lenguaje vaya, sentencia a sentencia, traduciéndose y ejecutándose directamente por el computador. El intérprete da lugar a una ejecución inmediata y no se crea un programa objeto. La ejecución del programa está supervisada por el intérprete. Son preferibles cuando el número de veces a ejecutar el programa es bajo y no hay problemas de velocidad, además con los intérpretes es más fácil desarrollar programas ya que podemos parar la ejecución para conocerlos valores de las variables en cada momento y qué instrucción se ha ejecutado.

Depurador/Debugger : Hace de intérprete para eliminar errores en el código a la hora de usar un compilador.

Compiladores interactivos : Permiten la edición, compilación, ejecución y depuración de programas escritos en lenguaje de alto nivel. Sus ventajas son que la comodidad para crear y depurar programas, permite al compilador realizar las comprobaciones mientras se edita para detectar con antelación los errores y cuando se hace una modificación no hay que retraducir todo el programa.

2.4- DATOS E INSTRUCCIONES

2.4.1 DATOS

Se estructuran en dos **tipos de tipos de datos** :

Primitivos : Son valores simples, manipulados con operaciones proporcionadas directamente por el lenguaje de programación y que normalmente se pueden representar y manipular con las herramientas disponibles en el lenguaje máquina del computador. Están formadas por el género, las operaciones y las propiedades.

Estructurados : agrupaciones de datos primitivos manipuladas mediante operaciones proporcionadas por el lenguaje de programación y representadas en el lenguaje máquina mediante colecciones de datos primitivos.

También podemos clasificar los datos **según la estancia durante la ejecución del programa:**

1. Estáticos: Se trata de datos que existen durante toda la vida del programa. Pueden ser globales (se usan en todo el programa o en toda una zona del código), constantes (valores que ya están asignados) o variables. Pueden tener (o no) un valor asignado previamente.

- PIC/Código independiente de la posición : *un fragmento de código cumple esta propiedad si puede ejecutarse en cualquier zona de memoria sin requerir un proceso de reubicación.*

2. Dinámicos asociados a la ejecución de una función : Se crean cuando se invoca la función correspondiente en la pila del proceso en una estructura de datos denominada *registro de activación*, donde, además de las variables locales y los parámetros, se guarda la información necesaria para retornar al punto de llamada cuando termina la ejecución de la función.

3. Dinámicos controlados por el programa: Se trata de datos dinámicos que el programa crea cuando considera oportuno. Habitualmente, los datos se guardan en una región denominada *heap*, donde se van almacenando todos los datos dinámicos usados por el programa. El espacio asignado se liberará cuando ya no sea necesario, o bien porque así lo indique el programa, usando el mecanismo que le ofrece el lenguaje para ello o mediante un mecanismo de detección automático denominado *recolección de basura*.

2.4.2 INSTRUCCIONES

Se clasifican en **compuestos**, formados por los condicionales *(if/else)* y los bucles *(for, while, do ... while)* ; y las **expresiones**, como sumas, productos... *(a+b*c/d)*.

3.| CICLO DE VIDA DE UN PROGRAMA

A partir de un código fuente, un programa debe pasar por varias fases antes de poder ejecutarse:

3.1- PREPROCESADO

El preprocesado es una fase que comparten algunos lenguajes de programación, anterior a la fase de compilación propiamente dicha. Se aplica generalmente sobre el archivo principal que contiene el código fuente. La función principal de los preprocesadores es hacer inteligible para el compilador el código fuente, cambiando las directivas de preprocesamiento por valores para el compilador. Estas directivas siempre están señaladas por caracteres especial para que solo sean modificadas por el preprocesador.

3.2- COMPILACIÓN

El compilador procesa cada uno de los archivos de código fuente para generar el correspondiente archivo objeto. Realiza las siguientes acciones:

- Genera código objeto y calcula cuánto espacio ocupan los diferentes tipos de datos
- Asigna direcciones a los símbolos estáticos (instrucciones o datos) y resuelve las referencias bien de forma absoluta o relativa (necesita reubicación).
- Las referencias a símbolos dinámicos se resuelven usando direccionamiento relativo a pila para datos relacionados a la invocación de una función, o con direccionamiento indirecto para el *heap*. No necesitan reubicación al no aparecer en el archivo objeto.
- Genera la Tabla de símbolos e información de depuración

3.3- ENSAMBLADO

Es la recolección de uno o más archivos o ficheros, agrupándolos juntos para formar una unidad lógica.

3.4- ENLAZADO

El enlazador (linker) debe agrupar los archivos objetos de la aplicación y las bibliotecas, y resolver las referencias entre ellos. En ocasiones, debe realizar reubicaciones dependiendo del esquema de gestión de memoria utilizado.

3.4.1 FUNCIONES

Se completa la etapa de resolución de símbolos externos utilizando la tabla de símbolos.

• Se agrupan las regiones de similares características de los diferentes módulos en regiones (código, datos inicializados o no, etc.)

• Se realiza la reubicación de módulos – hay que transformar las referencias dentro de un módulo a referencias dentro de las regiones. Tras esta fase cada archivo objeto tiene una lista de reubicación que contiene los nombres de los símbolos y los desplazamientos dentro del archivo que deben aún parchearse.

• En sistemas paginados, se realiza la reubicación de regiones, es decir, transformar direcciones de una región en direcciones del mapa del proceso.

3.4.2 TIPOS DE ENLAZADO Y ÁMBITO

Atributos de enlazado: *externo*, *interno* o *sin enlazado*.

Los tipos de enlazado definen una especie de **ámbito**:

• Enlazado externo --> visibilidad global

• Enlazado interno --> visibilidad de fichero

• Sin enlazado --> visibilidad de bloque

3.4.3 REGLAS

1. Cualquier objeto/identificador que tenga ámbito global deberá tener enlazado interno si su declaración contiene el especificador static.
2. Si el mismo identificador aparece con enlazados externo e interno, dentro del mismo fichero, tendrá enlazado externo.
3. Si en la declaración de un objeto o función aparece el especificador de tipo de almacenamiento extern, el identificador tiene el mismo enlazado que cualquier declaración visible del identificador con ámbito global. Si no existiera tal declaración visible, el identificador tiene enlazado externo.
4. Si una función es declarada sin especificador de tipo de almacenamiento, su enlazado es el que correspondería si se hubiese utilizado extern (es decir, extern se supone por defecto en los prototipos de funciones).
5. Si un objeto (que no sea una función) de ámbito global a un fichero es declarado sin especificar un tipo de almacenamiento, dicho identificador tendrá enlazado externo (ámbito de todo el programa). Como excepción, los objetos declarados const que no hayan sido declarados explícitamente extern tienen enlazado interno.
6. Los identificadores que respondan a alguna de las condiciones que siguen tienen un atributo sin enlazado:

   • Cualquier identificador distinto de un objeto o una función (por ejemplo, un identificador typedef).

   • Parámetros de funciones.

   • Identificadores para objetos de ámbito de bloque, entre corchetes {}, que sean declarados sin el especificador de clase extern.

3.5- CARGA Y EJECUCIÓN

El enlazador no completa la secuencia de reubicaciones requeridas para ejecutar el programa, por tanto, será en la etapa de carga en memoria del programa o durante su ejecución cuando se complete esta secuencia. Hay tres casos, dependiendo del esquema de gestión de memoria utilizado:

- **Si se usa segmentación**, el enlazador sólo ha realizado la etapa de reubicación de módulos y quedan pendientes la reubicación de regiones y la de procesos. Sin embargo, en la segmentación se fusionan

ambas reubicaciones en la misma etapa. El cargador copiará el programa en memoria sin modificarlo y será el hardware de gestión de memoria (MMU) el encargado de realizar esta reubicación combinada en tiempo de ejecución.

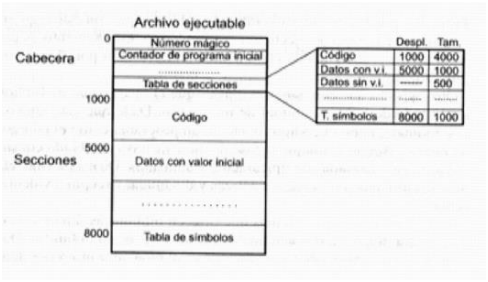
- En el **resto de los casos**, el enlazador ya ha realizado la reubicación de regiones, quedando pendiente de hacer la reubicación de procesos cuando se conozca en qué direcciones físicas ejecutará finalmente el proceso. En caso de que el hardware de gestión de memoria (MMU) sea capaz de realizar la reubicación de procesos (como ocurre en sistemas basados en registros valla o paginación), el cargador copiará el programa en memoria sin modificarlo y será el hardware el que lleve a cabo esta reubicación en tiempo de ejecución.

- **Si no se utiliza hardware de gestión de memoria (MMU) para llevar a cabo la reubicación**, ésta será realizada cuando el programa se copie en memoria. El cargador consultará la información de reubicación presente en el ejecutable (en los otros dos casos, la información no existiría en el ejecutable) y procederá a realizar la reubicación de procesos, modificando todas las referencias teniendo en cuenta las direcciones físicas donde va a ejecutar el proceso. Será por tanto, un proceso similar al caso previo, pero en este caso se trata de una reubicación software.

4.| ARCHIVOS OBJETO Y ARCHIVOS EJECUTABLES

4.1- ¿CUÁL ES LA DIFERENCIA?

Los archivos objeto (resultado de la compilación) y ejecutable (resultado del enlazado) son muy similares en cuanto a contenidos.



Sus principales diferencias son:

• En el ejecutable la cabecera del archivo contiene el punto de inicio del mismo, es decir, la primera instrucción que se cargará en el PC.

• En cuanto a las regiones, sólo hay información de reubicación si ésta se ha de realizar en la carga.

4.2- FORMATOS DE ARCHIVO

Tipo	Descripción
a.out	Es el formato original de los sistemas Unix. Consta de tres secciones: *text, data y bss* que se corresponden con el código, datos inicializados y sin inicializar. No tiene información de depuración.
COFF	El *Common Object File Format* posee múltiples secciones cada una con su cabecera pero están limitadas en número. Aunque permite información de depuración, ésta es limitada. Es el formato utilizado por Windows
ELF	Executable and Linking Format es similar al COFF pero elimina algunas de sus restricciones. Se utiliza en los sistemas Unix modernos, incluido GNU/Linux y Solaris.

4.3- SECCIONES DE UN ARCHIVO

.text – Instrucciones. Compartida por todos los procesos que ejecutan el mismo binario. Permisos: r y w. Es de las regiones más afectada por la optimización realizada por parte del compilador.

.bss – Block Started by Symbol: datos no inicializados y variables estáticas. El archivo objeto almacena su tamaño pero no los bytes necesarios para su contenido.

.data – Variables globales y estáticas inicializadas. Permisos: r y w

.rdata – Constantes o cadenas literales

.reloc – Información de reubicación para la carga.

Tabla de símbolos – Información necesaria (nombre y dirección) para localizar y reubicar definiciones y referencias simbólicas del programa. Cada entrada representa un símbolo.

Registros de reubicación – información utilizada por el enlazador para ajustar los contenidos de las secciones a reubicar.

5.| BIBLIOTECAS

Biblioteca: colección de objetos, normalmente relacionados entre sí. Las bibliotecas favorecen modularidad y reusabilidad de código. Podemos clasificarlas según la forma de enlazarlas:

• Bibliotecas estáticas - se enlazan con el programa en la compilación (.a)

• Bibliotecas dinámicas – se enlazan en ejecución (.so)

5.1- BIBLIOTECAS ESTÁTICAS

Una biblioteca estática es básicamente un conjunto de archivos objeto que se copian en un único archivo.

Las bibliotecas estáticas tiene algunos inconvenientes:

• El código de la biblioteca esta en todos los ejecutables que la usan, lo que desperdicia disco y memoria.

• Si actualizamos las bibliotecas, debemos recompilar el programa para que se beneficie de la nueva versión.

5.2- BIBLIOTECAS DINÁMICAS

Las bibliotecas dinámicas se integran en ejecución, para ello se ha realizado la reubicación de módulos. Su diferencia con un ejecutable: tienen tabla de símbolos, información de reubicación y no tiene punto de entrada. Pueden ser:

• Bibliotecas compartidas de carga dinámica – la reubicación se realiza en tiempo de enlazado.

• Bibliotecas compartidas enlazadas dinámicamente – el enlazado se realiza en ejecución.

6.| AUTOMATIZACIÓN EN LA CONSTRUCCIÓN DE SOFTWARE

Automatizar la construcción es la técnica utilizada durante el ciclo de vida de desarrollo de software donde la transformación del código fuente en el ejecutable se realiza mediante un guión (script).

La automatización mejora la calidad del resultado final y permite el control de versiones.

Hay varias formas de automatizar el proceso, entre ellas:

• Herramienta **make**

• **IDE (Integrated Development Environment)**, que embebe los guiones y el proceso de compilación y enlazado. Incluye todo lo necesario para que un programador pueda hacer su código y probarlo, todo desde el mismo programa. Entre todas las herramientas puede haber un compilador, un depurador, un editor de código... *Por ejemplo, Dev-C++.*