cg_enri
www.wuolah.com/student/cg_enri

Tema3compilacion.pdf

Tema 3 Compilación

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 de programas

Un lenguaje de programación es un conjunto de símbolos y de reglas para combinarlos. Que se usan para expresar algoritmos:

- Son independientes de la arquitectura física del computador, lo que aumenta la portabilidad de los programas.
- Una instrucción en lenguaje de alto nivel, después de traducirse, da lugar a instrucciones en lenguaje máquina.
- En el lenguaje de alto nivel hay anotaciones que son reconocibles entre las personas que lo utilizan.

Gramática

Proporciona una especificación sintáctica precisa de un lenguaje de programación. La complejidad de la verificación sintáctica depende de la gramática que define el lenguaje.

$$Gramática = (V_N, V_{T,} P, S)$$

$V_N \rightarrow$ símbolos no terminales

$V_T \rightarrow$ símbolos terminales

$P \rightarrow$ producciones o reglas gramaticales

$S \rightarrow$ símbolo inicial (no terminal)

Traductor

Es un programa que recibe como entrada un texto en un lenguaje de programación concreto, y produces como salida un texto en lenguaje máquina equivalente.

- Entrada $\rightarrow$ lenguaje fuente $\rightarrow$ de la máquina virtual.
- Salida $\rightarrow$ lenguaje objeto $\rightarrow$ de la máquina real.

Hay dos tipos:

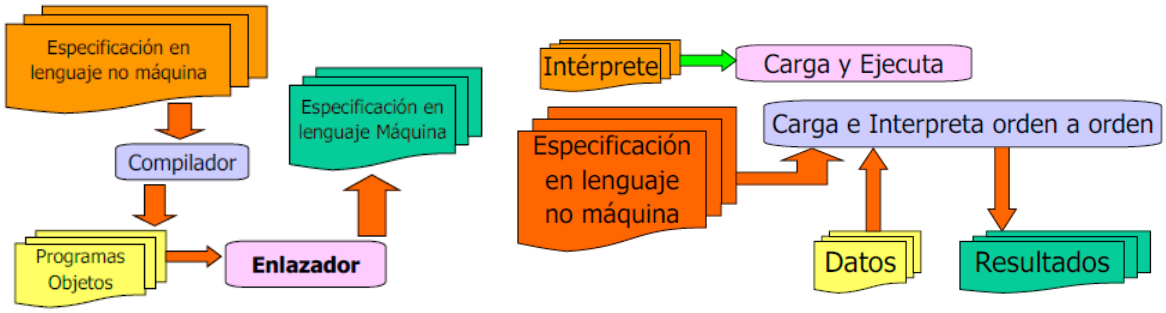
- **Compiladores:** Traduce la entrada a lenguaje máquina incompleto y con instrucciones maquina incompletas. (Se necesita un enlazador)
  - Enlazador (linker): liga las instrucciones máquina necesarias para completar el programa y genera un programa ejecutable para la máquina real.
- **Intérpretes:** Lee un programa fuente escrito para una máquina virtual y realiza y ejecuta de manera interna una a una las instrucciones obtenidas para la

INEAF
BUSINESS SCHOOL

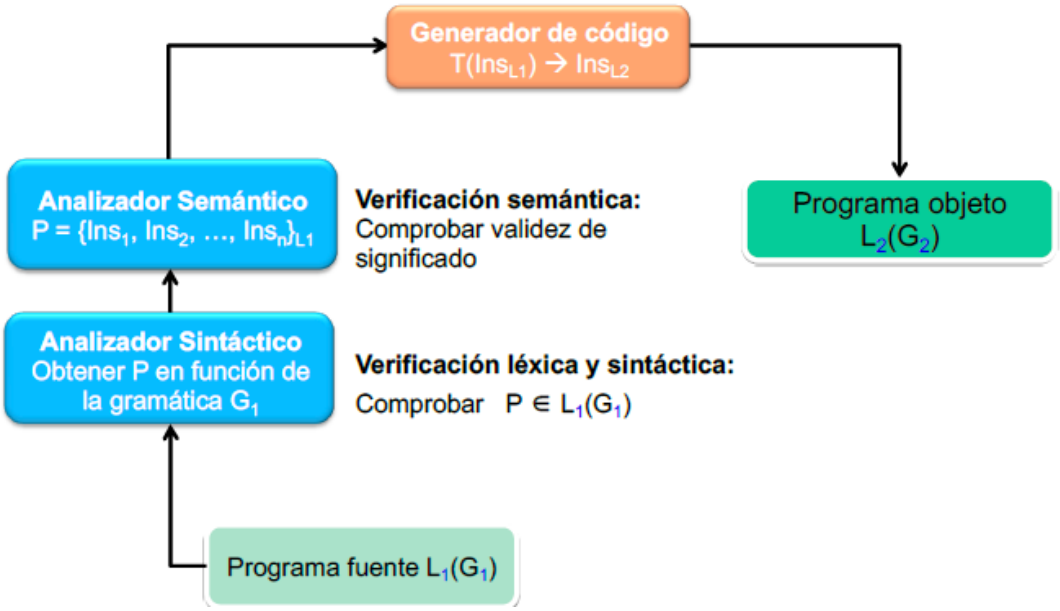
Es el momento
DE CRECER

Master en Asesoría Fiscal de Empresas

máquina real. No se genera ningún programa objeto equiparable al descrito en el programa fuente.



Esquema de traducción



Conceptos gramaticales

- **Alfabeto:** conjunto finito de símbolos. Ej: Binario {0,1}.
- **Cadena:** secuencia finita de símbolos de un alfabeto. Ej: (110, 0110, 0…).
- **Símbolos terminales:** elementos del alfabeto usados para formar cadenas. Ej: 0, 1…
- **Símbolos no terminales:** variables sintácticas que representan conjuntos de cadenas. Se utilizan en las reglas gramaticales y no son elementos del alfabeto.

$$N \rightarrow (N0 \mid N1 \mid 0 \mid 1)$$

Ejemplo:

Dada la siguiente gramática:

$\boldsymbol{P} = \{E \rightarrow EOE \mid (E) \mid id$

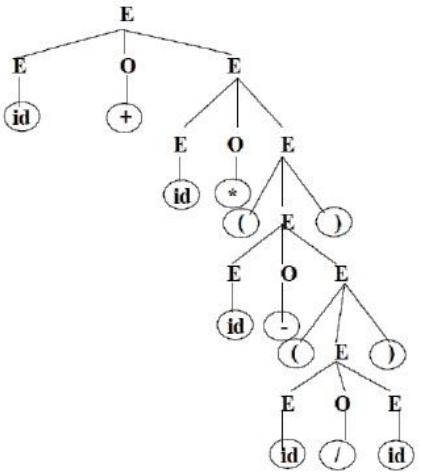
$\boldsymbol{V_N} = \{E\ O\}$

$\boldsymbol{V_T} = \{ (,) , id , +, -, *, / \}$

$\boldsymbol{S} = E$

Y el texto de entrada: id + id*(id-(id/id)).

Usando las reglas de formación gramatical, se obtendría una representación (**árbol sintáctico**) que valida la construcción del texto de entrada → verificación sintáctica correcta.



Fases en la construcción de un traductor:

Tabla de símbolos

- Fase de análisis: léxico, semántico y sintáctico
  - Análisis léxico
  - Análisis sintáctico
  - Análisis semántico
- Fase de síntesis: generación y optimización de código abierto.
  - Generación de código abierto
  - Optimizador de código intermedio
  - Generador de código
  - Optimizador de código

Análisis léxico

- Función: lee los caracteres de entrada del programa fuente y los agrupa en lexemas y produce como salida una secuencia de tokens para cada lexema en el programa fuente. (elimina caracteres superfluos y comentarios)
  - Lexema: secuencia de caracteres del alfabeto con significado.
  - Token: conjunto de lexemas que, según la gramática, tienen la misma función sintáctica.
  - Patrón: forma que pueden tomar los lexemas de un token.

  Tipos de token en muchos lenguajes:

  + Un token para cada palabra reservada
  + Tokens para los ordenadores
  + Un token que representa a todos los identificadores, tanto de variables como de subprogramas.
  + Uno o más tokens que representan constantes
  + Un token por cada signo de puntuación

ERROR LÉXICO: Se produce cuando el carácter de entrada no tiene asociado a ninguno de los patrones disponibles en la lista de tokens. Ejemplo: whi¿le

Se pueden expresar expresiones regulares para identificar un patrón del alfabeto de símbolos del alfabeto como pertenecientes a un token determinado.

- El operador *
- +: r* = r*/h
- Operador ?
- Una forma cómoda de definir clases de caracteres a|b|c…|z=[a-z]

Ejemplo:

$$
\begin{array}{rcl}
\text{S} & \rightarrow & \text{A} \mid \text{C} \\
\text{A} & \rightarrow & \text{id} = \text{E} \\
\text{C} & \rightarrow & \text{if E then S} \\
\text{E} & \rightarrow & \text{E O E} \mid (\text{E}) \mid \text{id} \\
\text{O} & \rightarrow & + \mid - \mid * \mid / \\
\text{id} & \rightarrow & \text{letra} \mid \text{id digito} \mid \text{id letra} \\
\text{letra} & \rightarrow & \text{a} \mid \text{b} \mid \cdots \mid \text{z} \\
\text{digito} & \rightarrow & 0 \mid 1 \mid \cdots \mid 9
\end{array}
$$

Token	Patrón
ID	`letra(letra|digito)*`
ASIGN	`"="`
IF	`"if"`
THEN	`"then"`
PAR_IZQ	`"("`
PAR_DER	`")"`
OP_BIN	`"+"|"-"|"*"|"/"`

Análisis sintáctico

Las gramáticas benefician a los diseñadores de lenguajes y a los que diseñan traductores:

- Aportan especificaciones sintácticas precisas de un lenguaje de programación
- A partir de ciertas clases gramaticales es posible construir un analizador sintáctico eficiente de forma automática
- Permite la evolución del lenguaje

Objetivo del analizador sintáctico

Analiza las secuencias de tokens y verifica que son sintácticamente correctas. A partir de una secuencia de tokens el analizador sintáctico nos devuelve:

- Si la secuencia es correcta o no sintácticamente.
- El orden en el que hay que aplicar producciones de gramática para obtener la secuencia de entrada (árbol sintáctico).

Si no se encuentra un árbol sintáctico la secuencia es incorrecta sintácticamente.

+ Decimos que la gramática es ambigua si admite más de un árbol sintáctico para una misma secuencia
+ La gramática es libre de contexto si solo admite tener un símbolo no terminal en la parte izquierda de las producciones

Ej: $P = \{A \rightarrow \alpha - A \in V_N, \alpha \subset (V_N \cup V_T) *\}$

Rafael García Parrado
Director de Marketing Digital

Donde aparezca A se puede poner α independientemente del contexto en el que se encuentre A.

Análisis semántico

La semántica es el significado dado a las distintas construcciones sintácticas. Está asociado a la estructura sintáctica de las secuencias.

Ejemplo: En una sentencia de asignación, según la sintaxis del lenguaje C, expresada mediante la producción siguiente:

$$\boldsymbol{sent\_asignacion \rightarrow IDENTIFICADOR\ OP\_ASIG\ expresion\ PYC}$$

donde IDENTIFICADOR, OP_ASIG y PYC son símbolos terminales (tokens) que representan, respectivamente, a una variable, el operador de asignación "=" y al delimitador de sentencia ";", deben cumplirse las siguientes reglas semánticas:

- IDENTIFICADOR debe estar previamente declarado.
- El tipo de la expresión debe ser acorde con el tipo del IDENTIFICADOR.

- Durante el análisis semántico se producen errores cuando se detectan construcciones sin un significado correcto
- En C podemos asignar varias variables de ≠ tipos, pero el compilador devuelve un *warning* si algo puede realizarse mal a posteriori.

Generación de código

- En esta fase se genera un archivo con un código de lenguaje objeto con el mismo significado que el texto fuente.
- Se puede intercalar una fase de generación de código intermedio para proporcionar independencia de las fases de análisis con respecto al lenguaje máquina o para optimizar el código.

Optimización de código

- Mejora el código mediante comprobaciones locales a un grupo de instrucciones a nivel global.
- Se pueden realizar optimizaciones de código tanto al código intermedio como al código objeto final.

Intérpretes

Hacen que un programa fuente escrito en un lenguaje vaya sentencia a sentencia, traduciéndose y ejecutándose directamente por el computador.

Consecuencias:

- Cada vez que se ejecuta el programa, hay que analizarlo.

- La ejecución del programa ejecuta en lenguaje fuente está supervisado por el intérprete.
- Las instrucciones de los bloques se analizan en cada información.
- La optimización se hace a nivel de instrucción.

¿Cuándo es útil un intérprete?

• El programador trabaja en un entorno interactivo y se desean obtener los resultados de la ejecución de una instrucción antes de ejecutar la siguiente.
• El programador lo ejecuta escasas ocasiones y el tiempo de ejecución no es importante.
• Las instrucciones del lenguaje tiene una estructura simple y pueden ser analizadas fácilmente.
• Cada instrucción será ejecutada una sola vez.

¿Cuándo no es útil un intérprete?

• Si las instrucciones del lenguaje son complejas.
• Los programas van a trabajar en modo de producción y la velocidad es importante.
• Las instrucciones serán ejecutadas con frecuencia.

Modelo de memoria de un proceso

Elementos responsables de la gestión de memoria:

- Lenguaje de programación
- Compilador
- Enlazador
- SO
- Hardware de gestión de memoria

Niveles de gestión de memoria

- **Nivel de procesos**: reparto de memoria entre los procesos. Responsabilidad del SO.
- **Nivel de regiones**: distribución del espacio asignado a las regiones de un proceso. Gestionado por el SO aunque la división en regiones la hace el compilador.
- **Nivel de zonas**: reparto de una región entre las diferentes zonas (nivel estático, dinámico basado en pila y dinámico basado en heap) de esta. Gestión del lenguaje de programación con soporte del SO.

Necesidades de memoria de un proceso

• Tener un espacio lógico independiente.
• Espacio protegido del resto de procesos.
• Posibilidad de compartir memoria.
• Soporte para diferentes regiones.
• Facilidades de depuración.

• Uso de un mapa amplio de memoria.
• Uso de diferentes tipos de objetos de memoria.
• Persistencia de datos.
• Desarrollo modular.
• Carga dinámica de módulos (por ejemplo, plug-in).

Modelo de memoria de un proceso

Estudiaremos aspectos relacionados con la gestión del mapa de memoria de un proceso, desde la generación del ejecutable a su carga en memoria:

• Nivel de regiones.
• Nivel de zonas.

Para ello veremos:

• Implementación de tipos de objetos necesarios para un programa y su correspondencia con el mapa de memoria.
• Ciclo de vida de un programa.
• Estructura de un ejecutable.
• Bibliotecas.

Tipos de Datos (desde el punto de vista de su implementación en memoria)

• Datos estáticos:
  – Globales a todo el programa, módulo o locales a una función (ámbito de visibilidad de una variable).
  – Constantes o variables.
  – Con o sin valor inicial – implementación con direccionamiento absoluto o direccionamiento relativo (PIC – código independiente de la posición).
• Datos dinámicos asociados a la ejecución de una función:
  – Se almacenan en pila en un **registro de activación (contiene variables locales, parámetros, dirección de retorno)**.
  – Se crean al activar una función y se destruyen al terminar la misma.
• Datos dinámicos controlados por el programa – **heap** (zona de memoria usada tiempo de ejecución para albergar los datos no conocidos en tiempo de compilación).

Código independiente de la posición

- Un fragmento de código cumple esta propiedad si puede ejecutarse en cualquier parte de la memoria.
- Es necesario que todas sus referencias a instrucciones o datos no sean absolutas sino relativas a un registro, por ejemplo, contador de programa.

Ciclo de vida de un programa

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

  – Preprocesado
  – Compilación
  – Ensamblado
  – Enlazado

- Carga y ejecución

Compilación

Procesa archivos de código fuente para generar archivo objeto:

- Genera código objeto y calcula cuanto espacio ocupan los diferentes tipos de datos.
- Asigna direcciones a símbolos estáticos y resuelve las referencias bien de forma absoluta o relativa.
- Genera tabla de símbolos e información de depuración.

Enlazador

Agrupa archivos objeto y resuelve las diferencias entre ellos:

- Utiliza la tabla de signos y resuelve símbolos externos.
- Se agrupan zonas con características similares.
- Se reubican módulos y forman regiones.

- Enlazado externo → visibilidad global
- Enlazado interno → visibilidad de fichero
- Sin enlazado → visibilidad de bloque

Carga en memoria principal y ejecución

La reubicación del proceso se realiza en la carga o en ejecución y es función del SO, ayudado del hardware específico. Depende del tipo de gestión de memoria que se realice:

- Segmentación
- Paginación

Diferencias entre archivos objeto y ejecutables

- 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.

Definiciones

- **Bibliotecas:** colección de objetos normalmente relacionadas entre sí.
- Las bibliotecas favorecen modularidad y reusabilidad de código.

Las podemos clasificar según la forma de enlazarlas:

- **Estáticas:** Se ligan con el programa en el enlazado (**.a**)
- **Dinámicas:** Se ligan con el programa en tiempo de ejecución (**.so**)

Bibliotecas estáticas

Conjunto de archivos objetos que se copian en un único archivo.

+ Construimos el código fuente
+ Generamos el código
+ Archivamos objeto (creamos la biblioteca)
+ Utilizamos biblioteca

Inconvenientes de las bibliotecas estáticas:

- El código de la biblioteca está en todos los ejecutables que la usan, lo que desperdicia disco y memoria principal.
- Si actualizamos una biblioteca estática, debemos recompilar los programas que la usan para que se puedan beneficiar de la nueva versión.
- Producen ejecutables grandes.

Bibliotecas dinámicas

Se integran en tiempo de ejecución, se realiza previamente la reubicación de módulos. Resuelven los inconvenientes que presentan las bibliotecas estáticas. El archivo correspondiente a una biblioteca dinámica se diferencia de un archivo ejecutable en los siguientes aspectos:

- Contiene información de reubicación
- En la cabecera no se almacena información del punto de entrada
- Contiene una tabla de símbolos

Al usar una biblioteca dinámica, en el proceso de montaje del programa ejecutable se incluye un módulo de montaje dinámico (enlazador dinámico): carga y monta las bibliotecas dinámicas usadas por el programa durante su ejecución.

+ Generamos el objeto de la biblioteca
+ Creamos la biblioteca
+ Usamos la biblioteca
+ Usamos la orden para ver las bibliotecas enlazadas con un programa

Automatización del enlazado y la compilación

Mejora la calidad del resultado final y permite el control de versiones de varias formas:

- Herramientas **make** $\rightarrow$ archivos makefile
- IDE (Integrated Development Environment – Entornos de Desarrollo Integrados), que embebe los guiones y el proceso de compilación y enlazado, p.e. **CodeBlocks**.