Introducción a la Compilación

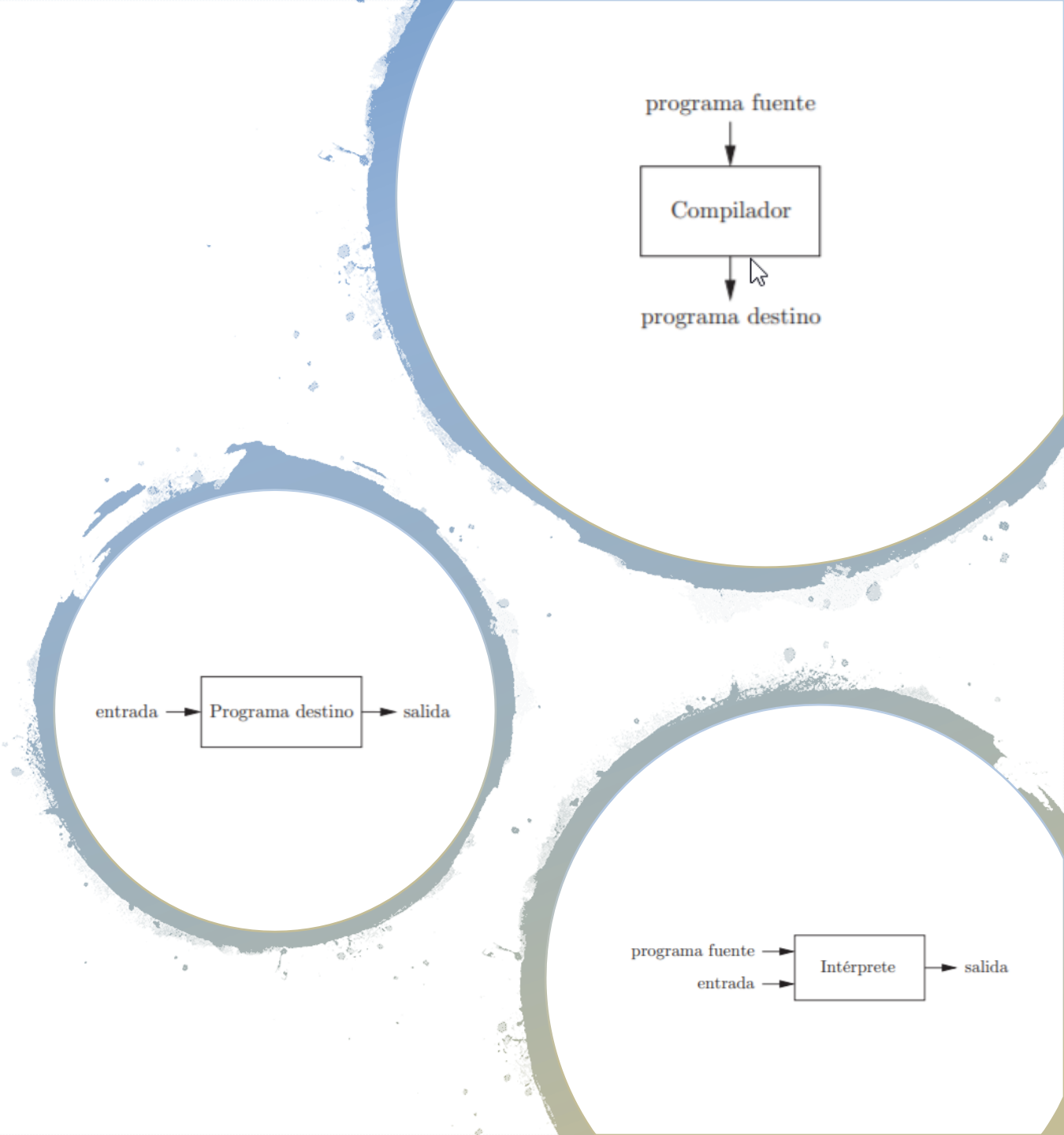
Unidad 1

Código Fuente

- El código fuente consiste en uno o más archivos que contienen las instrucciones de programación con las cuales un desarrollador de software creó una aplicación. Regularmente el código fuente fue escrito utilizando alguna herramienta de programación.

Procesadores de lenguaje

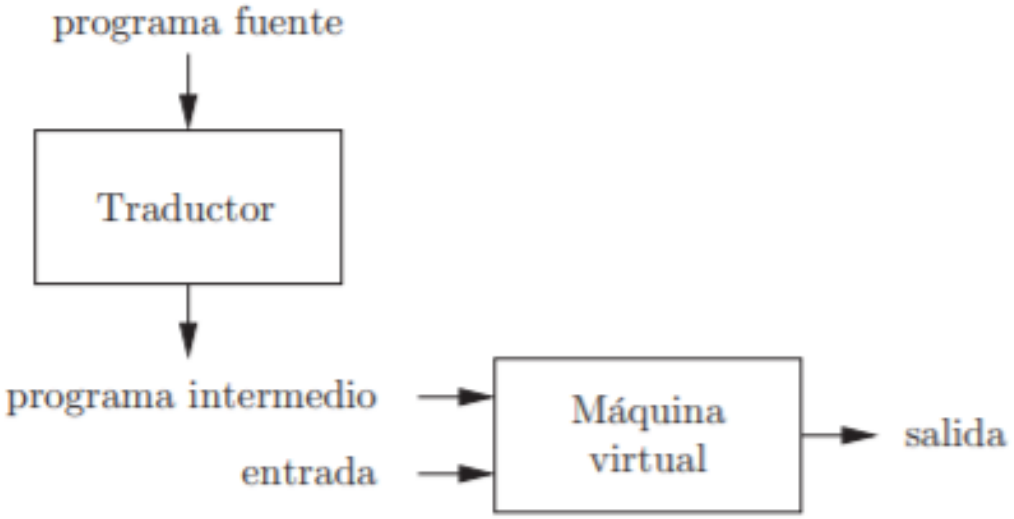
- Un compilador es un programa que puede leer un programa en un lenguaje y traducirlo en un programa equivalente en otro lenguaje.
- Si el programa destino es un programa ejecutable en lenguaje de máquina, entonces el usuario puede ejecutarlo para procesar las entradas y producir salidas.
- Un intérprete es otro tipo común de procesador de lenguaje. En vez de producir un programa destino como una traducción, el intérprete nos da la apariencia de ejecutar directamente las operaciones especificadas en el programa de origen con las entradas proporcionadas por el usuario
- Un programa compilado es más seguro que uno interpretado, porque no contiene el código fuente, que puede ser modificado incorrectamente por el usuario.



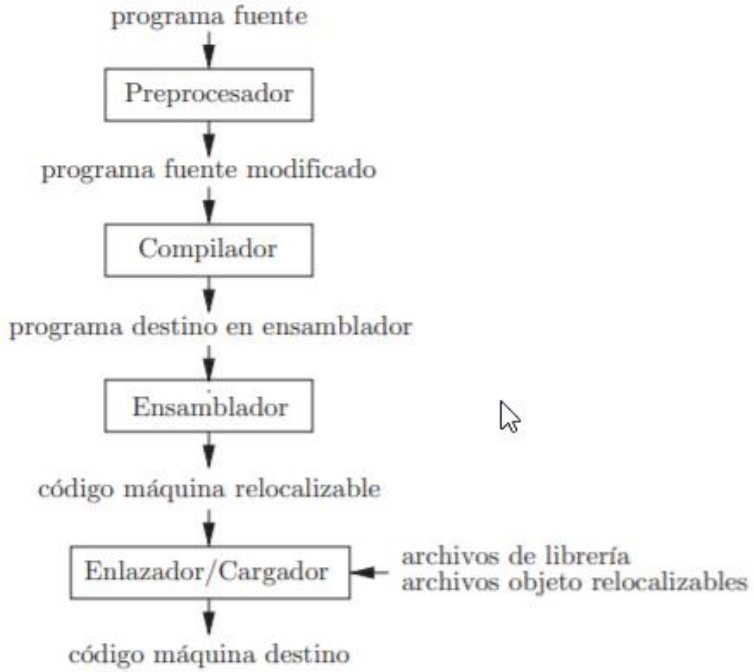
Compilador vs Intérprete

- El programa destino en lenguaje máquina que produce un compilador es, por lo general, más rápido que un intérprete al momento de asignar las entradas a las salidas. No obstante, por lo regular, el intérprete puede ofrecer mejores diagnósticos de error que un compilador, ya que ejecuta el programa fuente instrucción por instrucción.
- Un compilador genera un programa “stand-alone” es decir que tiene sentido y se puede ejecutar por sí solo, mientras que un programa interpretado siempre necesita su intérprete correspondiente para poder ejecutarse.
- Un intérprete traduce instrucciones de alto nivel en una forma intermedia para ser ejecutado. En contraste, un compilador, traduce instrucciones de alto nivel directamente en lenguaje de máquina.
- El intérprete traduce un programa línea a línea mientras que el compilador traduce el programa entero y luego lo ejecuta.
- El intérprete detecta si el programa tiene errores y permite su depuración durante el proceso de ejecución, mientras que el compilador espera hasta terminar la compilación de todo el programa para generar un informe de errores.

Procesadores Java



- Los procesadores del lenguaje Java combinan la compilación y la interpretación.
- Un programa fuente en Java puede compilarse primero en un formato intermedio, llamado bytecodes.
- Después, una máquina virtual los interpreta. Un beneficio de este arreglo es que los bytecodes que se compilan en una máquina pueden interpretarse en otra, tal vez a través de una red.
- Para poder lograr un procesamiento más rápido de las entradas a las salidas, algunos compiladores de Java, conocidos como compiladores just-in-time (justo a tiempo), traducen los bytecodes en lenguaje máquina justo antes de ejecutar el programa intermedio para procesar la entrada.



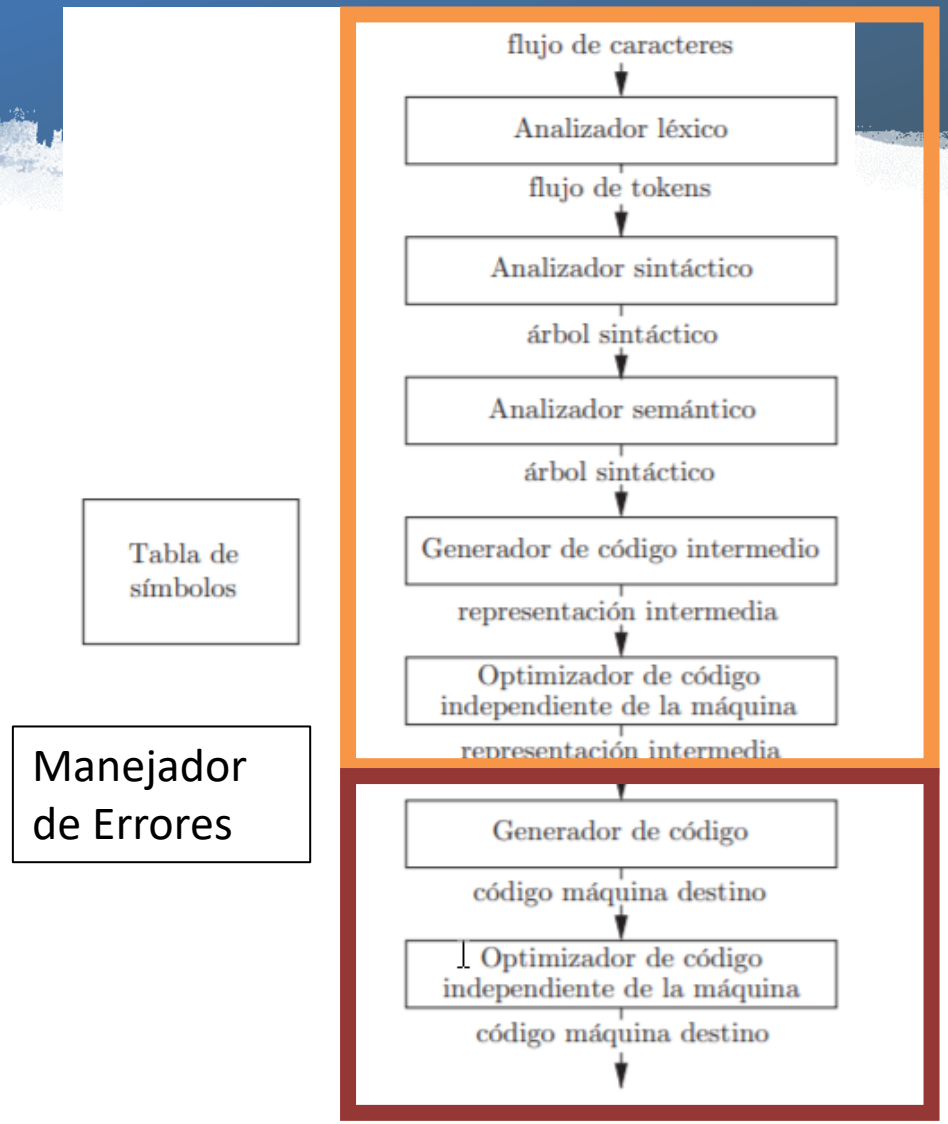
- Además de un compilador, pueden requerirse otros programas más para la creación de un programa destino ejecutable. Un programa fuente puede dividirse en módulos guardados en archivos separados. La tarea de recolectar el programa de origen se confía algunas veces a un programa separado, llamado preprocesador. El preprocesador también puede expandir algunos fragmentos de código abreviados de uso frecuente, llamados macros, en instrucciones del lenguaje fuente.
- Después, el programa fuente modificado se alimenta a un compilador. El compilador puede producir un programa destino en ensamblador como su salida, ya que es más fácil producir el lenguaje ensamblador como salida y es más fácil su depuración. A continuación, el lenguaje ensamblador se procesa mediante un programa llamado ensamblador, el cual produce código máquina relocalizable como su salida.
- A menudo, los programas extensos se compilan en partes, por lo que tal vez haya que enlazar (vincular) el código máquina relocalizable con otros archivos objeto relocalizables y archivos de biblioteca para producir el código que se ejecute en realidad en la máquina. El enlazador resuelve las direcciones de memoria externas, en donde el código en un archivo puede hacer referencia a una ubicación en otro archivo. Entonces, el cargador reúne todos los archivos objeto ejecutables en la memoria para su ejecución.

Un sistema de procesamiento de lenguaje

La estructura de un compilador

- Dentro del compilador hay dos procesos dentro del compilador: análisis y síntesis.
- La parte del análisis divide el programa fuente en componentes e impone una estructura gramatical sobre ellas. Después utiliza esta estructura para crear una representación intermedia del programa fuente. Si la parte del análisis detecta que el programa fuente está mal formado en cuanto a la sintaxis, o que no tiene una semántica consistente, entonces debe proporcionar mensajes informativos para que el usuario pueda corregirlo. La parte del análisis también recolecta información sobre el programa fuente y la almacena en una estructura de datos llamada tabla de símbolos, la cual se pasa junto con la representación intermedia a la parte de la síntesis.
- La parte de la síntesis construye el programa destino deseado a partir de la representación intermedia y de la información en la tabla de símbolos. A la parte del análisis se le llama comúnmente el front-end del compilador; la parte de la síntesis (propiamente la traducción) es el back-end

Fases de un compilador



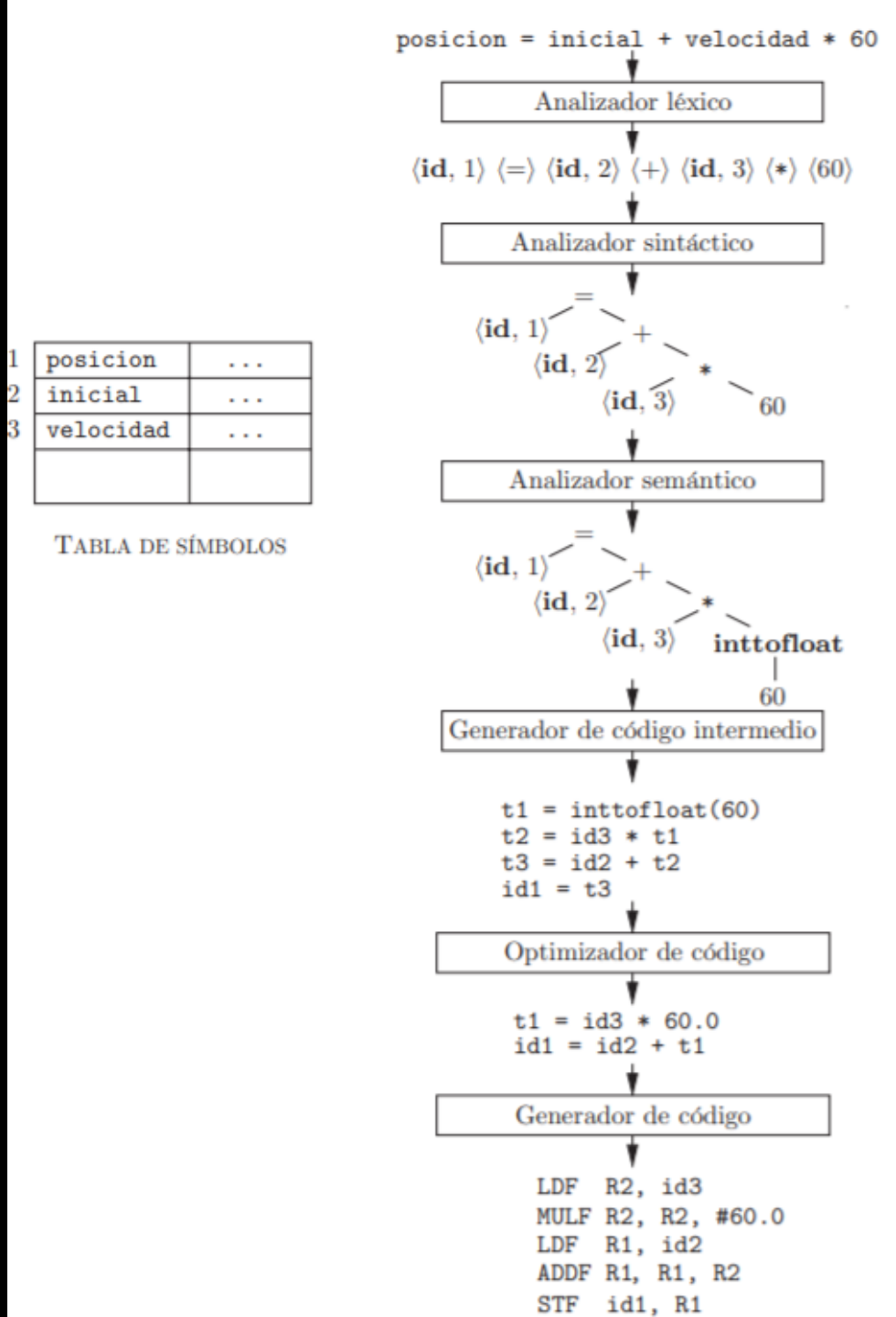
- Si examinamos el proceso de compilación con más detalle, podremos ver que opera como una secuencia de fases, cada una de las cuales transforma una representación del programa fuente en otro.
- En la práctica varias fases pueden agruparse, y las representaciones intermedias entre las fases agrupadas no necesitan construirse de manera explícita. La tabla de símbolos, que almacena información sobre todo el programa fuente, se utiliza en todas las fases del compilador.
- Algunos compiladores tienen una fase de optimización de código independiente de la máquina, entre el front-end y el back-end. El propósito de esta optimización es realizar transformaciones sobre la representación intermedia, para que el back-end pueda producir un mejor programa destino de lo que hubiera producido con una representación intermedia sin optimizar. Como la optimización es opcional

Análisis de léxico

- A la primera fase de un compilador se le llama análisis de léxico o escaneo. El analizador de léxico lee el flujo de caracteres que componen el programa fuente y los agrupa en secuencias significativas, conocidas como lexemas. Para cada lexema, el analizador léxico produce como salida un token de la forma: nombre-token, valor-atributo, que pasa a la fase siguiente, el análisis de la sintaxis.
- En el token, el primer componente nombre-token es un símbolo abstracto que se utiliza durante el análisis sintáctico, y el segundo componente valor-atributo apunta a una entrada en la tabla de símbolos para este token. La información de la entrada en la tabla de símbolos se necesita para el análisis semántico y la generación de código.

Análisis de léxico

- Por ejemplo, suponga que un programa fuente contiene la instrucción de asignación:
  - posición = inicial + velocidad * 60
- Los caracteres en esta asignación podrían agruparse en los siguientes lexemas y mapearse a los siguientes tokens que se pasan al analizador sintáctico:
  - posición es un lexema que se asigna a un token id, 1, en donde id es un símbolo abstracto que representa la palabra identificador y 1 apunta a la entrada en la tabla de símbolos para posición. La entrada en la tabla de símbolos para un identificador contiene información acerca de éste, como su nombre y tipo.
  - El símbolo de asignación = es un lexema que se asigna al token =. Como este token no necesita un valor-atributo, hemos omitido el segundo componente. Podríamos haber utilizado cualquier símbolo abstracto como asignar para el nombre-token, pero por conveniencia de notación hemos optado por usar el mismo lexema como el nombre para el símbolo abstracto.
  - inicial es un lexema que se asigna al token id, 2, en donde 2 apunta a la entrada en la tabla de símbolos para inicial.
  - + es un lexema que se asigna al token +
  - velocidad es un lexema que se asigna al token id, 3, en donde 3 apunta a la entrada en la tabla de símbolos para velocidad.
  - * es un lexema que se asigna al token *
  - 60 es un lexema que se asigna al token 60. Hablando en sentido técnico, para el lexema 60 formaríamos un token como número 4, en donde 4 apunta a la tabla de símbolos para la representación interna del entero 60, pero dejaremos la explicación de los tokens para los números hasta el capítulo 2. En el capítulo 3 hablaremos sobre las técnicas para la construcción de analizadores léxicos.
- El analizador léxico ignora los espacios en blanco que separan a los lexemas.
- <id, 1> = <id, 2> + <id, 3> <*> <60>
- En esta representación, los nombres de los tokens =, + y * son símbolos abstractos para los operadores de asignación, suma y multiplicación, respectivamente.



1	posicion	...
2	inicial	...
3	velocidad	...

TABLA DE SÍMBOLOS

Análisis sintáctico

- La segunda fase del compilador es el análisis sintáctico o parsing. El parser (analizador sintáctico) utiliza los primeros componentes de los tokens producidos por el analizador de léxico para crear una representación intermedia en forma de árbol que describa la estructura gramatical del flujo de tokens. Una representación típica es el árbol sintáctico, en el cual cada nodo interior representa una operación y los hijos del nodo representan los argumentos de la operación.
- Este árbol muestra el orden en el que deben llevarse a cabo las operaciones en la siguiente asignación: posición = inicial + velocidad * 60
- El árbol tiene un nodo interior etiquetado como *, con id, 3 como su hijo izquierdo, y el entero 60 como su hijo derecho. El nodo id, 3 representa el identificador velocidad. El nodo etiquetado como * hace explicito que primero debemos multiplicar el valor de velocidad por 60. El nodo etiquetado como + indica que debemos sumar el resultado de esta multiplicación al valor de inicial. La raíz del árbol, que se etiqueta como =, indica que debemos almacenar el resultado de esta suma en la ubicación para el identificador posición. Este ordenamiento de operaciones es consistente con las convenciones usuales de la aritmética, las cuales nos indican que la multiplicación tiene mayor precedencia que la suma y, por ende, debe realizarse antes que la suma.
- Las fases siguientes del compilador utilizan la estructura gramatical para ayudar a analizar el programa fuente y generar el programa destino. En el capítulo 4 utilizaremos gramáticas libres de contexto para especificar la estructura gramatical de los lenguajes de programación, y hablaremos sobre los algoritmos para construir analizadores sintácticos eficientes de manera automática, a partir de ciertas clases de gramáticas.

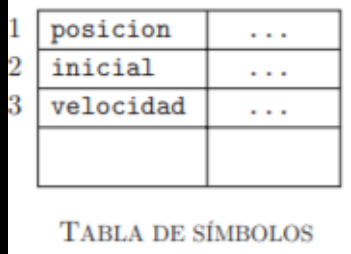
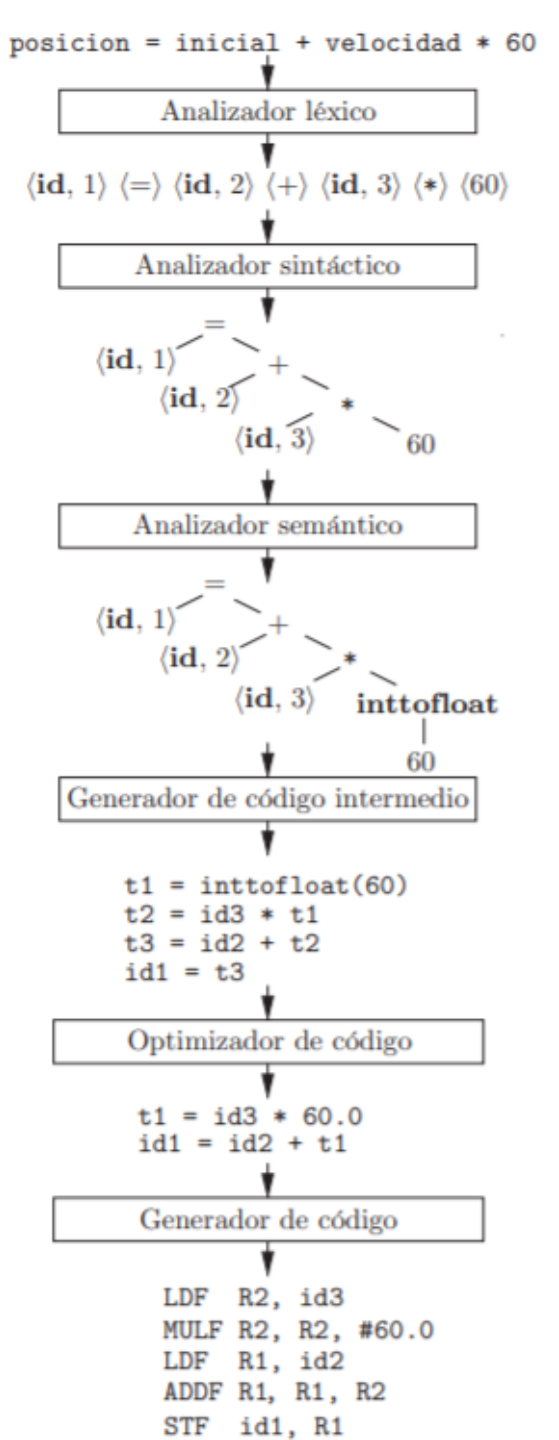
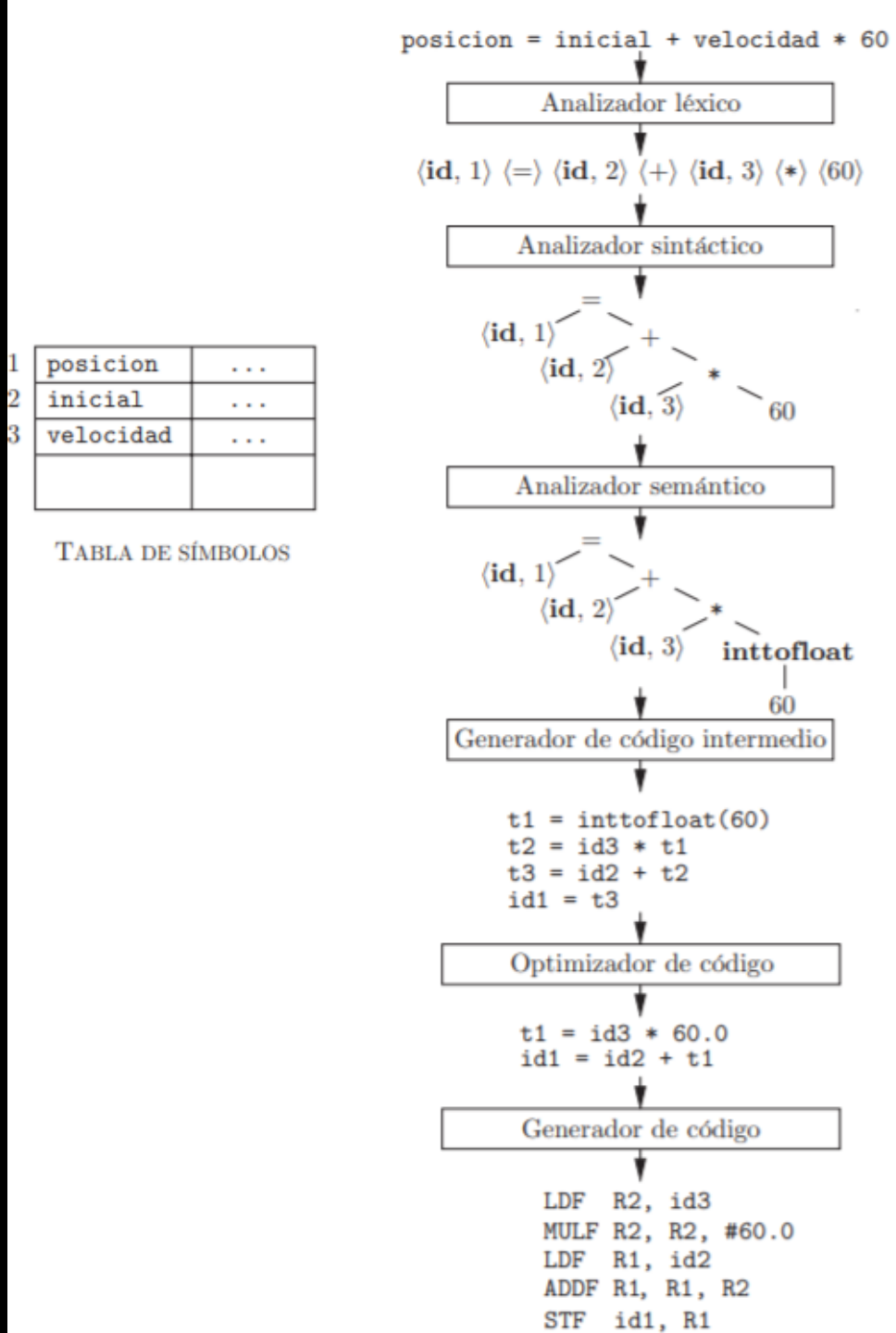


TABLA DE SÍMBOLOS



Análisis semántico

- El analizador semántico utiliza el árbol sintáctico y la información en la tabla de símbolos para comprobar la consistencia semántica del programa fuente con la definición del lenguaje. También recopila información sobre el tipo y la guarda, ya sea en el árbol sintáctico o en la tabla de símbolos, para usarla más tarde durante la generación de código intermedio.
- Una parte importante del análisis semántico es la comprobación (verificación) de tipos, en donde el compilador verifica que cada operador tenga operandos que coincidan. Por ejemplo, muchas definiciones de lenguajes de programación requieren que el índice de un arreglo sea entero; el compilador debe reportar un error si se utiliza un número de punto flotante para indexar el arreglo.
- La especificación del lenguaje puede permitir ciertas conversiones de tipo conocidas como coerciones. Por ejemplo, puede aplicarse un operador binario aritmético a un par de enteros o a un par de números de punto flotante. Si el operador se aplica a un número de punto flotante y a un entero, el compilador puede convertir u obligar a que se convierta en un número de punto flotante.
- Dicha conversión aparece en la figura 1.7. Suponga que posicion, inicial y velocidad se han declarado como números de punto flotante, y que el lexema 60 por sí solo forma un entero. El comprobador de tipo en el analizador semántico de la figura 1.7 descubre que se aplica el operador * al número de punto flotante velocidad y al entero 60. En este caso, el entero puede convertirse en un número de punto flotante. Observe en la figura 1.7 que la salida del analizador semántico tiene un nodo adicional para el operador inttofloat, que convierte de manera explícita su argumento tipo entero en un número de punto flotante. En el capítulo 6 hablaremos sobre la comprobación de tipos y el análisis semántico.



Generación de código intermedio

- En el proceso de traducir un programa fuente a código destino, un compilador puede construir una o más representaciones intermedias, las cuales pueden tener una variedad de formas. Los árboles sintácticos son una forma de representación intermedia; por lo general, se utilizan durante el análisis sintáctico y semántico.
- Después del análisis sintáctico y semántico del programa fuente, muchos compiladores generan un nivel bajo explícito, o una representación intermedia similar al código máquina, que podemos considerar como un programa para una máquina abstracta. Esta representación intermedia debe tener dos propiedades importantes: debe ser fácil de producir y fácil de traducir en la máquina destino.
- El código de tres direcciones, que consiste en una secuencia de instrucciones similares a ensamblador, con tres operandos por instrucción. Cada operando puede actuar como un registro. La salida del generador de código intermedio consiste en la secuencia de código de tres direcciones.
    - t1 = inttofloat(60)
    - t2 = id3 * t1
    - t3 = id2 + t2
    - id1 = t3
- Hay varios puntos que vale la pena mencionar sobre las instrucciones de tres direcciones. En primer lugar, cada instrucción de asignación de tres direcciones tiene, por lo menos, un operador del lado derecho. Por ende, estas instrucciones corrigen el orden en el que se van a realizar las operaciones; la multiplicación va antes que la suma en el programa fuente. En segundo lugar, el compilador debe generar un nombre temporal para guardar el valor calculado por una instrucción de tres direcciones. En tercer lugar, algunas “instrucciones de tres direcciones” como la primera y la última en la secuencia anterior, tienen menos de tres operandos.
- Las técnicas para la traducción dirigida por la sintaxis, para la comprobación de tipos y la generación de código intermedio en las construcciones comunes de los lenguajes de programación, como las expresiones, las construcciones de control de flujo y las llamadas a procedimientos.

Optimización de código

- La fase de optimización de código independiente de la máquina trata de mejorar el código intermedio, de manera que se produzca un mejor código destino. Por lo general, mejor significa más rápido, pero pueden lograrse otros objetivos, como un código más corto, o un código de destino que consuma menos poder. Por ejemplo, un algoritmo directo genera el código intermedio, usando una instrucción para cada operador en la representación tipo árbol que produce el analizador semántico.
- Un algoritmo simple de generación de código intermedio, seguido de la optimización de código, es una manera razonable de obtener un buen código de destino. El optimizador puede deducir que la conversión del 60, de entero a punto flotante, puede realizarse de una vez por todas en tiempo de compilación, por lo que se puede eliminar la operación inttofloat sustituyendo el entero 60 por el número de punto flotante 60.0. Lo que es más, t3 se utiliza sólo una vez para transmitir su valor a id1, para que el optimizador pueda transformar en la siguiente secuencia más corta:
- t1 = id3 * 60.0
- id1 = id2 + t1
- Hay una gran variación en la cantidad de optimización de código que realizan los distintos compiladores. En aquellos que realizan la mayor optimización, a los que se les denomina como “compiladores optimizadores”, se invierte mucho tiempo en esta fase. Hay optimizaciones simples que mejoran en forma considerable el tiempo de ejecución del programa destino, sin reducir demasiado la velocidad de la compilación.

Generación de código

- El generador de código recibe como entrada una representación intermedia del programa fuente y la asigna al lenguaje destino. Si el lenguaje destino es código máquina, se seleccionan registros o ubicaciones (localidades) de memoria para cada una de las variables que utiliza el programa. Después, las instrucciones intermedias se traducen en secuencias de instrucciones de máquina que realizan la misma tarea. Un aspecto crucial de la generación de código es la asignación juiciosa de los registros para guardar las variables.
- Por ejemplo, usando los registros R1 y R2, el código intermedio podría traducirse en el siguiente código de máquina:
  - LDF R2, id3
  - MULF R2, R2, #60.0
  - LDF R1, id2
  - ADDF R1, R1, R2
  - STF id1, R1
- El primer operando de cada instrucción especifica un destino. La F en cada instrucción nos indica que trata con números de punto flotante. El código encarga el contenido de la dirección id3 en el registro R2, y después lo multiplica con la constante de punto flotante 60.0. El # indica que el número 60.0 se va a tratar como una constante inmediata. La tercera instrucción mueve id2 al registro R1 y la cuarta lo suma al valor que se había calculado antes en el registro R2. Por último, el valor en el registro R1 se almacena en la dirección de id1, por lo que el código implementa en forma correcta la instrucción de asignación .
- En esta explicación sobre la generación de código hemos ignorado una cuestión importante: la asignación de espacio de almacenamiento para los identificadores en el programa fuente. La organización del espacio de almacenamiento en tiempo de ejecución depende del lenguaje que se esté compilando. Las decisiones sobre la asignación de espacio de almacenamiento se realizan durante la generación de código intermedio, o durante la generación de código.

Administración de la tabla de símbolos

- Una función esencial de un compilador es registrar los nombres de las variables que se utilizan en el programa fuente, y recolectar información sobre varios atributos de cada nombre. Estos atributos pueden proporcionar información acerca del espacio de almacenamiento que se asigna para un nombre, su tipo, su alcance (en qué parte del programa puede usarse su valor), y en el caso de los nombres de procedimientos, cosas como el número y los tipos de sus argumentos, el método para pasar cada argumento (por ejemplo, por valor o por referencia) y el tipo devuelto.
- La tabla de símbolos es una estructura de datos que contiene un registro para cada nombre de variable, con campos para los atributos del nombre. La estructura de datos debe diseñarse de tal forma que permita al compilador buscar el registro para cada nombre, y almacenar u obtener datos de ese registro con rapidez.

El agrupamiento de fases en pasadas

- El tema sobre las fases tiene que ver con la organización lógica de un compilador. En una implementación, las actividades de varias fases pueden agruparse en una pasada, la cual lee un archivo de entrada y escribe en un archivo de salida. Por ejemplo, las fases correspondientes al front-end del análisis léxico, análisis sintáctico, análisis semántico y generación de código intermedio podrían agruparse en una sola pasada. La optimización de código podría ser una pasada opcional. Entonces podría haber una pasada de back-end, consistente en la generación de código para una máquina de destino específica.
- Algunas colecciones de compiladores se han creado en base a representaciones intermedias diseñadas con cuidado, las cuales permiten que el front-end para un lenguaje específico se interconecte con el back-end para cierta máquina destinto. Con estas colecciones, podemos producir compiladores para distintos lenguajes fuente para una máquina destino, mediante la combinación de distintos front-end con sus back-end para esa máquina de destino. De manera similar, podemos producir compiladores para distintas máquinas destino, mediante la combinación de un front-end con back-end para distintas máquinas destino.

Herramientas de construcción de compiladores

- Al igual que cualquier desarrollador de software, el desarrollador de compiladores puede utilizar para su beneficio los entornos de desarrollo de software modernos que contienen herramientas como editores de lenguaje, depuradores, administradores de versiones, profilers, ambientes seguros de prueba, etcétera. Además de estas herramientas generales para el desarrollo de software, se han creado otras herramientas más especializadas para ayudar a implementar las diversas fases de un compilador.
- Estas herramientas utilizan lenguajes especializados para especificar e implementar componentes específicos, y muchas utilizan algoritmos bastante sofisticados. Las herramientas más exitosas son las que ocultan los detalles del algoritmo de generación y producen componentes que pueden integrarse con facilidad al resto del compilador. Algunas herramientas de construcción de compiladores de uso común son:
    1. Generadores de analizadores sintácticos (parsers), que producen de manera automática analizadores sintácticos a partir de una descripción gramatical de un lenguaje de programación.
    2. Generadores de escáneres, que producen analizadores de léxicos a partir de una descripción de los tokens de un lenguaje utilizando expresiones regulares.
    3. Motores de traducción orientados a la sintaxis, que producen colecciones de rutinas para recorrer un árbol de análisis sintáctico y generar código intermedio.
    4. Generadores de generadores de código, que producen un generador de código a partir de una colección de reglas para traducir cada operación del lenguaje intermedio en el lenguaje máquina para una máquina destino.
    5. Motores de análisis de flujos de datos, que facilitan la recopilación de información de cómo se transmiten los valores de una parte de un programa a cada una de las otras partes. El análisis de los flujos de datos es una parte clave en la optimización de código.
    6. Kits (conjuntos) de herramientas para la construcción de compiladores, que proporcionan un conjunto integrado de rutinas para construir varias fases de un compilador.